Pós-Graduação em Ciência da Computação

“Domain-Specific Game Development”

Por

André Wilson Brotto Furtado

Tese de Doutorado

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

Universidade Federal de Pernambuco
CENTRO DE INFORMÁTICA
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ANDRÉ WILSON BROTTO FURTADO

Domain-Specific Game Development

ESTE TRABALHO FOI APRESENTADO À PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DO CENTRO DE INFORMÁTICA DA UNIVERSIDADE FEDERAL DE PERNAMBUCO COMO REQUISITO PARCIAL PARA OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIA DA COMPUTAÇÃO.

ORIENTADOR: André Luís de Medeiros Santos
CO-ORIENTADOR: Geber Lisboa Ramalho

RECIFE, Fevereiro/2012

Catalogação na fonte

Bibliotecária Jane Souto Maior, CRB4-571

Furtado, André Wilson Brotto
Domain-Specific game development / André Wilson Brotto Furtado. - Recife: O Autor, 2012.
xxv, 233 p. : il., fig., tab.

Orientador: André Luis de Medeiros Santos.
Tese (doutorado) - Universidade Federal de Pernambuco. CIn, Ciência da Computação, 2012.
Inclui bibliografia e apêndice.

1. Ciência da Computação - Engenharia de Software. 2. Linguagem de programação. 3. Jogos digitais. I. Santos, André Luis de Medeiros (orientador). II. Título.

005.12 CDD (23. ed.) MEI2012 – 034

“[...] a good programmer in these times does not just write programs. [...] a good programmer does language design, though not from scratch, but building on the frame of a base language.”

-- Guy Steele Jr

“Language defines the boundary to our world: it sets what we can describe and also what we can’t.”

-- Ludwig Wittgenstein

"Enjoyment is not a goal; it is a feeling that accompanies important ongoing activity."

-- Paul Goodman

To Juliana.

ACKNOWLEDGEMENTS

Among all of the few thousand pages I've probably written in the last years, this is the one I was the most looking forward to. It means a very big journey is about to reach its destination. It means it's time to wrap up the efforts and lessons learned from a long chapter. It means it's time to resume.

Since the first words of this document have been typed, a lot has changed, from my country of residence to my civil status. Finally standing here, on the other edge, enables me to look back and realize the amount of people who made it possible for me to cross the bridge. For very real, folks: I'll carry your encouragement and support forever!

To my wife Juliana, words won't be enough. I know this moment means as much to you as it means to me. Sorry for all the moments that did not happen. Or could have been better. Thanks for your patience, support and for simply being there. I love you!

To my family overseas, thanks for the continuous incentive and support, for letting me to share this with you, be it the good, the bad or the ugly. Your will for making this moment to happen refueled my efforts, even when the sunrise reminded how hard things can be.

To my friends and advisors André Santos and Geber Ramalho, thanks for opening the door and making this possible. You started it all. Without your plan, trust, knowledge and motivation, this would have never gotten beyond an "apparently interesting academic idea".

To my experiment's subjects… wow! 14 sessions, 125 hours, 32 versions of games, a lot of pizzas and an unlimited amount of camaraderie. Angelo Ribeiro, Carlos Rodrigues, Gustavo Andrade, Gustavo Magalhães, Julio Lins, Livar Cunha, Renato Ferreira and Rodrigo Silveira: this research would have a third of its confidence and validity if it wasn't for you. Consider your contributions a legacy to the SPL and DSL communities. Enjoy Section 5.4.

To all of those from the research groups who contributed to this thesis, especially the RiSE group and UFPE's Center of Informatics, my sincere thanks. You helped me to connect the dots when I needed the most. The network you created is very noble and invaluable to Software Engineering researchers.

To my new and old friends, now spread around multiple continents, as well as my teammates: thanks for making this a lighter and happier path. You are there even when you are not here, and you know what I mean. Being connected to you all is a gift. Thanks for sharing so much energy and growth.

Finally, I'd like to register a modest, humble thanks to my past self. Sorry for anything. Thanks for pursuing the goal. Thanks for believing.

André Furtado

December 31, 2011

ABSTRACT

This thesis introduces the concept of Domain-Specific Game Development, which is an approach that harvests the benefits of Software Product Lines (SPLs) to create digital games belonging to a same family more effectively. The need for such an approach is justified by the fact that introducing reuse and SPL concepts into digital games development, in fact into any other domain, is not a straightforward task, due to the peculiarities of each domain. Specifically for games, traditional Requirements Engineering and use cases cannot be applied as is. Business requirements are trumped by prototypes, rapid experimentation and emotion-based requirements, such as immersion and nostalgia. The so popular concept of game genres is nevertheless too vague and ambiguous to define the scope of a family of games from a SPL perspective. The end-user's (player's) experience is much more based on surprises and ruptures than the adherence to standards. And no approach aimed at improving game development can ignore game engines, which have become the state-of-the-art development resource for digital games by bringing to the area the benefits of Software Engineering and object-orientation. However, the abstraction level provided by game engines could still be made less complex to consume by means of language-based tools, the use of visual models as first-class citizens (in the same way as source code) and a better integration with development processes. With such a motivation, Domain-Specific Game Development bridges SPLs to game development, culminating with domain-specific languages (DSLs) and generators streamlined for game sub-domains and aimed at automating more of the digital games development process. It focuses on employing Domain Engineering, Model-Driven Development and software reuse to encapsulate the common and best practices in game development, yet supporting variable and unforeseen behavior. In order to evaluate the proposed approach, we present exploratory and confirmatory case studies, as well as a controlled experiment performed with software engineers with industry experience. With a measured development effort improvement of more than five times in average, we suggest Domain-Specific Game Development as a viable alternative for same-family game development scenarios in order to reduce the complexity of consuming game engines, to break down game development tasks into more granular and automatable chunks, to enable the creation of expressive yet executable game specifications, to deliver incremental value for prioritized game sub-domains, to build effective domain-specific assets tailored to the unique characteristics of an envisioned family of games and, finally, to still provide flexibility and extensibility for game developers and designers to focus on what makes each game unique and distinct.

Keywords: game development, software reuse, software product lines, domain engineering, domain-specific languages, experimental study.

RESUMO

Este tese apresenta o conceito de Domain-Specific Game Development, uma abordagem que emprega os benefícios de linhas de produção de software (SPLs) para criar mais eficientemente jogos digitais pertencentes a uma mesma família. A necessidade por essa abordagem é justificada pelo fato de que a introdução de conceitos de reuso e SPL em jogos digitais, na verdade em qualquer domínio, não é trivial, devido às peculiaridades de cada domínio. Especificamente para jogos, a Engenharia de Requisitos tradicional e casos de uso não podem ser aplicados como são. Prototipagem, rápida experimentação e requisitos baseados em emoção são preferidos em relação a requisitos de negócio. O tão popular conceito de motores de jogos é muito vago e ambíguo para definir o escopo de uma linha de produção de jogos. A experiência de jogadores é mais baseada em surpresas e rupturas do que na aderência a padrões. E nenhuma abordagem focada na melhoria do desenvolvimento de jogos pode ignorar motores de jogos, atual estado-da-arte no desenvolvimento de jogos digitais. Por outro lado, a abstração provida por eles poderia ser menos complexa de ser consumida, através de ferramentas baseadas em linguagens, o uso de modelos como cidadãos de primeiroa classe (assim como código fonte) e uma melhor integração com o processo de desenvolvimento. Dada essa motivação, Domain-Specific Game Development faz a ponte entre SPLs e o desenvolvimento de jogos, culminando com linguagens de domínio específico (DSLs) e geradores alinhados para sub-domínios de jogos e focados em automatizar mais o processo de desenvolvimento de jogos digitais. A abordagem emprega Engenharia de Domínio, Model-Driven Development e reuso de software para encapsular as melhores e mais comuns práticas do desenvolvimento de jogos, ainda suportando comportamento variável e imprevisto. Para avaliar a abordagem, apresentamos casos de estudo confirmatórios e exploratórios, assim como um experimento controlado realizado com engenheiros de software da indústria. Com uma melhoria de produtividade observada de mais de 5 vezes em média, sugerimos Domain-Specific Game Development como uma abordagem viável para o desenvolvimento de jogos que pertencem a uma mesma família, de modo a reduzir a complexidade no consumo de motores de jogos, quebrar tarefas de desenvolvimento em blocos mais automatizáveis, habilitar a criação de especificações expressivas porém executáveis, entregar valor incrementalmente para sub-domínios priorizados, construir artefatos de domínio específico alinhados às características únicas da família-alvo de jogos e, finalmente, prover flexibilidade e extensibilidade para que desenvolvedores e designers de jogos possam focar no que torna cada jogo único distinto.

Palavras-chave: desenvolvimento de jogos, reuso de software, linhas de produção de software, engenharia de domínio, linguagens de domínio específico, estudo experimental.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

There is a vital difference between an application's problem domain and its code [Jackson, 1995]. These are two different worlds, each with its own language, experts and ways of thinking. A finished application forms the intersection between these worlds [Metacase, 2009].

However, most of the information about a software application above the level of source code is typically captured in an informal manner or lost, in spite of the fact that this is the information that best tells what is being built and why developers are building it a certain way [Greenfield et al., 2004]. This means that when a new application is being developed, mappings from the higher-level problem domain to the lower-level solution domain are performed from scratch over and over again, although most software products are more similar than different to each other. Too much time and effort is spent manually rediscovering and reinventing solutions to common domain requirements, which has led to the situation where software is still built more or less in isolation, and the majority of software projects are late, over budget and defect ridden [Lenz & Wienands, 2006].

In order to tackle such a challenge, Parnas [1976] was the first to introduce the idea of family-based production strategies. According to him, a set of programs is considered to constitute a family whenever it is worthwhile to study the programs from the set by **first studying the common properties** of the set and **then determining the special properties** of the individual family members. Building on top of that, Czarnecki & Eisenecker [2000] advocated that the first step in the transition from single systems to system families is to adopt a Domain Engineering or software product line (SPL) process.

Instantiating the concepts of reuse and application families to the digital games development industry is the main theme of this thesis. We are interested in investigating how the peculiarities of digital games development impact the applicability of SPLs and software factories to such a domain, proposing an approach entitled Domain-Specific Game Development to improve the effectiveness of creating games belonging to a same family. We also have a profound interest in *measuring* the benefits and shortcomings of the approach.

This chapter contextualizes the problem domain by first investigating high-level Software Engineering approaches for avoiding software development in isolation, then provides a background on how the discussion is relevant to digital games development, highlighting some of the area's specific needs and challenges. That sets up the motivation for the research. Following that, a clearer definition of this thesis' goal is stated, along with its challenges and the structure through which the proposed solution will be presented.

1.1 Motivation

Stating this research's question requires reflecting on a couple of other fundamental questions related to Software Engineering and digital games development. The first question is *"What does Software Engineering propose against software development in isolation?"*. It relates to the aforementioned problem of reinventing solutions over and over again, and leads to the concept of software industrialization.

The second question then brings into discussion *"Why should digital games development care about software industrialization?"*. By answering such a question, it is possible to understand the needs and specific challenges faced by digital games development upon software industrialization.

Finally, the third question, *"What is the business relevance of the digital games industry?",* contextualizes the macro-domain this thesis deals with, from a consumer and market perspectives. Its answer makes it possible to envision the range to which the contributions of this research can be applied.

What does Software Engineering propose against software development in isolation?

In order to avoid manually rediscovering and reinventing solutions, an increasing interest on software reuse, known as the process of creating software systems from existing software rather than building them from scratch [Krueger, 1992], has been observed as an approach to improve quality, productivity and consequently reducing costs in software development. The software reuse process is more effective when systematically planned and managed in the context of a specific domain, in which applications belonging to the same "family" share functionality [Almeida, 2007]. In other words, systematic software reuse is a paradigm shift in software development from building single systems to application families of similar systems [Frakes & Isoda, 1994].

A major addition to existing reuse approaches since the 1990s is the concept of software product families or Software Product Lines (SPLs) [Clements & Northrop 2001] [Bosch, 2000] [Weiss & Lai, 1999], defined as "a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way". Since instances of a product family share the same problem domain and consequently the same root problems, Software Product Lines are aligned with the concepts of Domain-Specific Development, an approach for solving problems that can be applied when a particular problem occurs over and over again [Fowler, 2005]. Industrial experiences of SPLs, Domain-Specific Development and Modeling report major improvements in productivity (3 to 10 times), lower development costs and better quality [Kelly & Tolvanen, 2008] [Weiss & Lai,

1999]. Key contributing factors are the higher level of design activities that have to be performed, in contrast to low-level implementation details.

The concept of software factories, as introduced by Greenfield et al. [2004], builds on several already established concepts such as Software Product Lines, reusable software assets (application blocks and frameworks), Model-Driven Development (MDD) and automated context-based guidance. It heavily relies on integration with development environments and on a more graphical approach that, unlike Computer-Aided Software Engineering (CASE) tools, is seriously interested in semantics and control over code generation [Fowler, 2005]. That is enabled by means of visual Domain-Specific Languages (DSLs), limited form of computer languages designed for a specific class of problems that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain [van Deursen et al., 2000]. Software factories use visual DSLs to elevate program specifications to compact visual domain-specific notations that are easier to write and maintain, raising the abstraction level of system development beyond programming by specifying the solution directly using domain concepts. Model transformation, such as code generation, is applied to derive other work products from the designed models.

Although successful cases of software reuse can be found in the industry [SPL Hall of Fame, 2012], Almeida [2007] reports that experiences in software reuse are often related to individuals and small groups, who practice it in an ad hoc way, with high risks that can compromise future initiatives in this direction. Lenz & Wienands [2006] point out that, even though the products are similar, reuse mostly happens at a limited scope, like copy and paste of code snippets and reuse of class libraries. Nascimento [2008] states that apart from certain specific domains, such as mathematical libraries, in general the benefits of the traditional software reuse approach have been limited. Reuse is still relatively low and the emphasis is on low-level (code) reuse.

One the other hand, digital games development, despite being a prominent industry, is a field typically characterized by ad hoc, low-level development [Reyno & Cubel, 2008]. Historically, excessive high performance constraints together with very tight schedules forced digital games development to trade more refined Software Engineering techniques for a result-oriented but less organized development process, as well as reusability for in-house development, in a methodology that became known as “pedal to the metal” [Rollings & Morris, 2000]. Together with other digital games development peculiarities, such as a non-traditional requirement engineering, such factors provide an indication that additional efforts might be required in order to leverage the aforementioned Software Engineering reuse techniques in digital games development as a means to satisfy the increasing demands on such an industry. The next subsection addresses the topic.

Why should digital games development care about software industrialization?

Neward [2008] points out that there are a number of challenges coming up that we cannot solve with our current set of languages and tools, and that we stand on the threshold of a "renaissance in programming languages". Greenfield et al. [2004] estimate that the total global demand for software will grow by an order of magnitude over the next decades: *"design patterns and specialized tools demonstrated limited but effective knowledge reuse; however, without deeper increases in productivity, total software development capacity seems destined to fall far short of total demand"*. Those and other authors believe that there is evidence that the current development paradigm is near its end, and that a new paradigm is needed to support the next leap forward in software development technology.

On the other hand, the expectations on digital games are already extremely high [Folmer, 2007]. Innovative hardware (new input devices, more powerful graphics cards, etc.), improved "software as a service" business models (online player community memberships such as the Xbox Live, on-demand game titles rental without late fees, etc.), applicability to multiple domains (entertainment, education, training, etc.) and innovative gameplay make digital games to be perceived as one of major streams where bleeding-edge technologies and ideas are showcased. Digital games are a cultural phenomenon, and that continuously pushes the boundaries of what is expected from the game development ecosystem.

As a result, it seems that the exponential growth of the total global demand for software is a trap waiting for the game development industry, since:

- The hardest part of making a game has always been the engineering [Blow, 2004];
- Game development is a field typically characterized by ad hoc, low-level development [Reyno & Cubel, 2008];
- Many game developers struggle with component integration and managing the complexity of their architectures, while expanding deadlines and escalating costs have notoriously plagued the game industry [Folmer, 2007].

Almost 50 years after the development of the first computer game, Wiering [1999] pointed out that game development teams were still spending a considerable amount of time in solving programming problems, instead of creatively designing the game. Game development has turned, decade after decade, into a much more complex experience, while the game industry has a perpetual shortage of qualified people due to its increasing expertise requirements [Blow, 2004].

Before the commoditization of digital games by app stores and marketplaces, only the top 5% of products make a profit in the game industry [Gillin, 2006]. Numerous games may start development but are canceled, or perhaps even completed but never published. Video

game companies have been accused of excessive invocation of "crunch time" [Frauenheim, 2004], point at which the team is believed to be failing to achieve the milestones required to launch the game on time, causing disrupt in the developers' work-life balance. Informal sources[1] also point out that experienced game developers may work for years and yet never ship a title: "such is the nature of the business". This volatility is likely inherent to the artistic nature of games, and the complexity of the work flow in video game creation makes it very difficult to manage the team's schedules.

Digital games have always evolved toward increased technical complexity to deliver to their users (players) features they have never experienced before. As a result, each wave of games is attempting several technical feats that are mysterious and unproven [Blow, 2004]. Thus game developers carry a lot of *technical* risk (it is not possible to accurately schedule the unknown or predict how it will interact with the rest of the system) as well as *game design* risk (how will unprecedented features feel to end-users?). Rather than being discouraging, the challenge involved in making a game is a major part of the reason why so many people are attracted to the field. Therefore, game development teams need to focus on such risks and new features rather than wasting time on menial and routine tasks that should be performed repeatedly.

In contrast, game engines are the state-of-the-art game development resource. They are an important step toward game development automation by bringing together the benefits from Software Engineering and object-oriented technologies. They provide an additional abstraction layer by encapsulating common knowledge and providing a reusable game development foundation. Nevertheless, this abstraction level could be made less complex to consume by means of language-based tools, the use of visual models as first-class citizens (in the same way as source code) and a better integration with development processes. Additionally, in many circumstances game engines and integrated environments in which games are developed, including the recent additional of integrated graphical environments such as Unit [Menard, 2011], are too generic to allow the benefits of application families development to be exploited.

The SPL Hall of Fame compiled by the Software Engineering Institute [SPL Hall of Fame, 2012] does not contain any entries related to the digital games development industry. Nevertheless, the same problems and trends that raised solutions such as Software Product Lines and Domain-Specific Development seem to be present in digital games. For instance, Blow [2004] points out that one of the major roots of difficulties in digital games development lies on complexity and problems due to high **domain-specific requirements**. Sometimes, for

[1] http://en.wikipedia.org/wiki/Video_game_development

defining game content and behavior, such as for building the geometry of the game world, **domain-specific editors** are written from scratch. To write good engine code, there is a lot of **domain-specific knowledge** required. Often the application program interfaces (APIs) consumed by digital games are difficult to deal with because they embody some **conceptual model** that is a poor fit for the way the game needs to work. Moreover, all-or-nothing approaches for reuse are common in game development, where an entire game engine is licensed instead of **more granular, reusable components**.

If from one side Calheiros et al. [2007] point out that tool support development is a pre-requisite for a widespread adoption of SPL practices, on the other hand Blow [2004] complains that "excellent development tools", which would be able to tackle some of the complexities faced by the game industry, are simply not there yet. For instance, prototyping and playtesting are game development activities already recognized as fundamental pillars of game design [Henderson, 2006] [Fullerton et al., 2004]. However, ad hoc processes and tools, many times built from scratch, are used to perform such activities, instead of models and reusable tools integrated into a domain-specific, context-aware process. Finally, the need for more abstraction beyond source code in the digital games development process is evident, since "creating a game will always expand until it exceeds our implementation abilities" [Blow, 2004]. In summary, we believe that in order to satisfy the software demands of the next decades, the game development industry needs more specialized tools, languages, frameworks, integration and automation, developed in the context of a product line, allowing game developers to work more productively with more abstraction and closer to their application domain.

Now that some motivation has been set for employing reuse and SPL approaches in game development, the next subsection discusses the actual dimension of the digital games market and industry. Only a market that is demanding enough justifies upfront SPL investments and enables the exploitation of economies of scope.

What is the business relevance of the digital games industry?

The Entertainment Software Association (ESA) 2011 report on *Essential Facts about the Computer and Digital Games Industry* [ESA, 2011] shares that digital games (both computer and console games) were responsible in 2010 for 25.1 billion dollars in sales. Even with the uncertain economic scenario that unfolded on the end of the first decade of the XXI century, the growth in the video game industry has not stopped [Klotz, 2009]. The U.S. video game industry presented a 19% growth year over year, while one out of four dollars spent on entertainment in the U.S. goes to gaming.

According to the last data available as of the writing of this research, the size of the game industry corresponds to 7% of the software industry [ESA, 2011] [Datamonitor, 2011]. The numbers are a match even for the colossal music and movie industries, while studies reveal that more is spent in digital games than in musical entertainment [Slocombe, 2005]. In short, digital games are one of the most profitable industries in the world.

Additional data from ESA reveals that:

- About **72%** of American households play computer or video games;
- Differently from what it could be expected, the average game player is **37 years old**, while players under 18 years old represent only 25% of the market;
- The average age of the most frequent game purchaser is **41 years**;
- The average number of years adult gamers have been playing computer or video games is **12 years**;

Therefore, it is possible to conclude that the digital games development industry is definitively consolidated as a serious business. Misconceptions related to the maturity of this industry and its target audience vanishes when faced against the aforementioned numbers.

Data from other sources corroborate with ESA's facts. McGonigal [2010] shares that 3 billion hours are spent every week in online games, while the average teen gamer has played 10,000 hours when reaching 21 years old. According to her, almost 6 million years were spent in one single game alone: World of Warcraft.

From a development effort standpoint, the growth in the digital games industry is exponential [Blow, 2004]. In the mid-80s, a game could be developed in 3 months by one programmer who also did the design and art, from conception to final implementation [Reyno & Cubel, 2008]. Decades later, digital games development requires teams up to 100 multidisciplinary specialists, including programmers, game designers, artists, writers, voice actors, musicians and other roles. The budget of an AAA game is estimated in a dozen million dollars, requiring up to 4 years of development and an outsourcing network that spans through the globe. That is also a match for many other industries.

1.2 Goals

Given the context provided by the previous section on Software Engineering reuse techniques and their relevance to digital games development, this research's question can be stated as follows:

How to harvest and measure the benefits of software product lines to create digital games belonging to a same family more effectively, taking into account the peculiarities of the domain?

The process employed to answer such a question leveraged multiple approaches and processes that exploit application families and domain-specific development, culminating with a practical approach to develop game SPLs, named **Domain-Specific Game Development**. We are not aimed though at redefining a complete Domain Engineering process per se, comprehending all of its three macro-activities (Domain Analysis, Domain Design and Domain Implementation). Otherwise, our focus is on the Domain Engineering tasks that have the most impact and relevance to digital games development, and therefore need to be elaborated or adapted. We essentially focus on bridging the Game Domain Analysis phase to the creation of core domain assets for game SPLs, such as domain-specific languages and generators, which are still underexplored by the game development industry. In fact, Frakes & Kang [2005] mention that there is a need for a seamless integration between the models output from Domain Analysis and the inputs needed for Domain Implementation such as components, domain-specific languages and applications generators. Domain-Specific Game Development attempts to fill such gaps, but in the specific context of digital games development.

The topic of creating processes for Domain Engineering and reuse is not new. Such processes have already being cataloged and new processes were defined in other researches [Almeida, 2007] [Nascimento, 2008] to solve traditional deficiencies of reuse processes, such as the lack of activities, sub-activities, roles, inputs and outputs of each step in a systematic way, as well as the lack of comprehensive approaches to encompass the three classical macro-activities of Domain Engineering. Therefore, rather than reinventing the wheel, this work leverages state-of-the art research in the area [Greenfield et al., 2004] [Kelly & Tolvanen, 2008]. The main distinction of the approach is that differently from generic SPL processes, the discussion presented in this research is specific to the digital games development macro-domain, dealing with many of the peculiarities that make such an area so different from others in Computer Science. The need for specific guidance on game SPLs is also evident from the fact that current SPL processes lack details for specific Domain Engineering tasks, eventually causing SPLs to be an unviable approach [Nascimento, 2008], especially for domains with more specific constraints.

It is worth noticing that a very important goal of this research is to actually measure the outcomes of Domain-Specific Game Development in practice. We not only employ exploratory and confirmatory case studies [Easterbrook et al., 2007], but also evaluate the approach by means of an experimental study [Wohlin et al., 1999] performed with software engineers with industry experience. Such an evaluation is aimed at measuring the development effort improvement implied by the proposed approach, among other metrics, situating its ef-

fectiveness against other SPL-based approaches and providing an indication on the approach's return of investment.

1.3 Challenges

From one side, digital games development is one of the most creative disciplines in the Computer Science domain. Uniqueness and innovation are intrinsic attributes of successful titles. On the other hand, by promoting automation, enforcing predictability and stimulating reuse, software factories and SPLs are concerned with turning the current software development paradigm, based on craftsmanship, into a manufacturing process. Hence, understanding how the variability support provided by SPLs and the software factories paradigm can satisfy the creativity of digital games development is one of the challenges of this research. Similarly, we are challenged to assess how much of "The Art of Computer Game Design" is subject to industrialization.

Since digital games constitute a broad universe of game genres and mechanics, the proposed approach is more applicable to some types of games than others. For instance, the approach's concept of a Domain-Specific Game Architecture builds atop game engines, promoting them to domain frameworks that can be more seamlessly consumed by domain-specific languages and other assets. If the target game domain does not use game engines, which could be the case for some casual games developed for the mobile market[2], then the approach's tasks related to the Domain-Specific Game Architecture should be customized. On the other hand, state-of-the-art toolsets in such mobile domains, such as integrated graphical environments like Unity [Menard, 2011], are backed up by game engines and therefore still part of the Domain-Specific Game Development's scope.

Finally, deploying a complete, usable software factory or SPL demands substantial resources. Therefore, considerable efforts were incurred on the confirmatory case study of this research, as there was a commitment on not restricting it to the research environment, but actually releasing it to the digital games development community with an acceptable quality level, in order to collect feedback. In the same way, the controlled experiment demanded a considerable investment (236 man-hours and 32 versions of games developed) to enable measurements of the effectiveness of the approach as well as understanding when upfront investments in a game SPL break-evens with the creation of one-off[3] game instanc-

[2] 36.5% of game content sales in the game industry belong to mobile apps, social networking gaming, subscriptions and other formats beyond traditional computer games and video games [ESA, 2011].

[3] Something that is done or created only once, and often quickly, simply, or improvisationally; occurring once; one-time.

es. Detailed information on similar experiments is not easy to find in the SPL and DSL literatures. More details are presented in Chapter 5 and Appendix B.

1.4 Thesis Organization

This thesis is organized in six chapters. Apart from this introductory chapter, the remaining chapters are organized as follows:

- Chapter 2 presents a comprehensive history of game development technologies and processes, ranging from the assembly language, multimedia APIs, click-n-play tools, game engines, waterfall game development processes, spiral processes, agile processes and model-driven process to upcoming tendencies.
- Chapter 3 discusses techniques for building application families, such as software factories, SPLs and one of their most important foundations: Domain-Specific Development. A discussion about how software factories are suitable for digital games development is also provided.
- Chapter 4 details the proposed approach, building on the two background topics previously presented (game development evolution and application families).
- Chapter 5 presents an evaluation of the proposed approach, including exploratory and confirmatory case studies, as well as a controlled experiment to measure its effectiveness.
- Chapter 6 is the last chapter of this thesis. It investigates future work that can be carried out from this research and concludes about its obtained results.

This thesis also contains six appendixes, all related to the evaluation of the proposed approach. They are organized as follows:

- Appendix A presents the detailed feature model of the ArcadEx case study.
- Appendix B provides a compilation of metrics and evaluation approaches for Domain-Specific Development and MDD.
- Appendix C presents the questionnaire used in the controlled experiment.
- Appendix D presents the cheat sheet used by subjects during the controlled experiment.
- Appendix E presents the game development tasks checklist used by subjects during the controlled experiment.

2. DIGITAL GAMES DEVELOPMENT

A major evolution in game development technologies and processes has occurred since its early days. This chapter presents a relatively brief but comprehensive history of digital games development, pointing out the advantages and limitations of each era. Following that, it analyses tendencies and suggests proposals built atop previous technologies and processes to improve the future of the game development.

2.1 From Assembly to Doom

Since the early days of digital games development, game developers were faced with unique challenges and scenarios, which turned game development into a very peculiar domain when compared to software development in general. While the majority of software was developed and run on big mainframes, with plenty of resources (at least for that era), computer games development were targeted at smaller computers, with many limitations[4] [Rocha, 2003].

Computer games programming soon became an art of overcoming memory and processing constraints. To get satisfactory results from the existing hardware was the great boundary to be overcome. During the '80s, for example, games were meant to be played in 8-bit processor computers, with a speed of just 4MHz and a memory of 48KB to 64KB [Rollings & Morris, 2000]. Performance and (small) size were the two most important factors in game programming; therefore, code optimizations were heavily performed on demand[5].

Since programs generated by C compilers were too big and slow for game development, most computer games were programmed in assembly language. Hence, their code was not portable to other platforms. Besides that, game debugging was also a big challenge, since the debugger and the computer game generally did not fit together into the target memory.

Once computer games were developed based on hackers' method to build low-level applications, focused basically on appearance and performance, important Software Engineering concepts and design goals were systematically overlooked, such as reusability and modularity. Furthermore, assimilating Software Engineering was made even more troublesome since game programmers were wounded by the *Not Built Here (NBH) Syndrome*,

[4] Actually, some games were developed for mainframes in the early days [Bellis, 2009], but these were a few experimental attempts before computer games became primarily designed to be played at home, which made it possible to create games of longer duration and to better explore them commercially [Juul, 2009].

[5] Years later, games targeted at mobile devices made this problem to be revisited.

which stated that everything should be developed in-house because components developed by others were supposedly “slower, worse and/or would not work”. Such hardware-driven and ad hoc approach, which traded the productivity of higher-level languages for assembly language performance, as well as reusability for “in-house” development, became known as “pedal to the metal” [Rollings & Morris, 2000]. Such an approach was characterized by the lack of an organized game development process.

The growth in consumer demands, technical and esthetical complexity, as well as resources invested in the development of digital games caused the awareness of how important the use of Software Engineering concepts and high-level languages is to game development [Furtado & Santos, 2002]. An important preliminary milestone toward such a reality was the creation of the first successful game that adopted C as its native programming language: Doom (Figure 1), in 1993. For the first time, developers suffering from the *NBH Syndrome* had to recognize that a compiler generated a final code as competitive as code written in assembly language. In fact, by using a 32-bit compiler, Doom overcame the 640KB limitation of 16-bit programs and set a new era for game development.

Figure 1 – Doom, by *id Software*

In the subsequent years, the increase of hardware capabilities as well as the creation of more powerful compilers to produce better optimized executables made it possible for programmers to focus on more abstract game development concepts (such as Artificial Intelligence and Computer Graphics). Low-level implementation details, driver peculiarities and the *NBH Syndrome* were losing appeal and Software Engineering finally had a real chance in game development.

The game industry soon realized the importance of making the development process more productive and searched for new solutions by means of creating and consuming innovative technologies and tools. The most expressive of such development assets created by the game industry are presented in the following sections, as well as a discussion regarding their effectiveness (advantages and especially drawbacks) in game development.

2.2 Multimedia APIs

Multimedia Application Program Interfaces (APIs) are programming libraries that can be used to directly access the machine hardware (graphics devices, sound cards, input devices, etc.). Such APIs are not only useful to provide means to create games with good performance, but also for enabling the portability of digital games among devices manufactured by different vendors. Therefore, by using a Multimedia API, game developers are provided with a standard device manipulation interface and do not need to worry about low-level peculiarities of each possible target device. Furthermore, since these APIs are flexible to accommodate device enhancements and extensions, hardware support becomes a responsibility of the hardware manufacturers themselves and game programmers are abstracted from such changes [Madeira, 2003]. Another interesting feature of multimedia APIs is their capability to emulate devices which are needed by a digital game, but are not necessarily present in a given computer, like 3D graphics accelerator cards [Pessoa, 2003].

The usage of multimedia APIs provided a new abstraction layer in game development, as illustrated in Figure 2. In fact, the majority of digital games today are developed, directly or indirectly, by means of an underlying multimedia API.

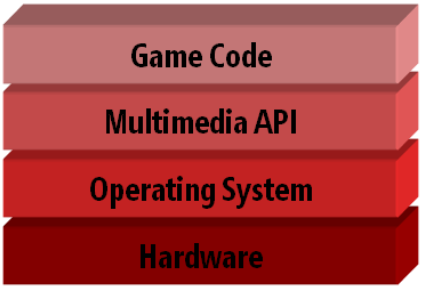


Figure 2 – A multimedia API provides abstraction layer to game development

Most multimedia APIs provide graphics, sound and input handling features. These features are usually highly optimized to get the best results from the hardware, and parts of the library are often rewritten for specific hardware devices to improve speed even more. The most important graphics functions found in just about every multimedia API include setting the display mode, drawing (and reading) pixels, drawing (transparent) bitmaps, scrolling the screen, page flipping[6], reading user input and playing background music and sound effects. Multimedia APIs played a key role in moving the digital games industry from DOS applica-

[6] Page flipping is a technique used in most games to avoid flickering. First, all graphics are drawn on an off-screen buffer (the virtual page), then by swapping (flipping) this virtual page with the visual page (the screen memory) the new graphics all come together as a new frame. The flipping is synchronized with the screen refresh [Ambrosine, 2009].

tions to more robust 32-bit operating systems, starting with Windows 95. For the PC, the Allegro API was a reference for creating DOS games, while today the most used Multimedia APIs are Microsoft DirectX and OpenGL.

2.2.1 Discussion: Multimedia APIs Effectiveness in Game Development

Although Multimedia APIs handle almost all the desired low-level tasks of a game, the game itself still has to be programmed. The APIs provide features that are generalized for multimedia applications development and therefore do not offer the best desired abstraction level for game programmers. For example, they do not provide features to trigger the transition between game states (phases), entity behavior modeling nor artificial intelligence. In other words, the semantic gap between game designers and the final code remains too high if multimedia APIs are the only abstraction mechanism used.

Additionally, interaction with such APIs can only be done programmatically, not visually. Such a limitation prevent automation and productivity in the execution of some tasks (such as specifying the tiles of a tiled background map), which would have to be executed by exhaustive "copy and paste" commands and through counter-intuitive actions[7].

2.3 Click-n-Play Tools

With the goal of simplifying game development and making it more broadly accessible, the concept of **click-n-play tools** became very popular. They aim at creating complete games with no programming at all, but through a "visual programming" approach (Figure 3). The end user is aided with intuitive interfaces for creating game sprites, defining entity behavior, the game flow and adding sound, menus, text screens and other resources to the game. Such tools generally make use of an underlying multimedia API, by compiling the visual game specification into a series of API calls.

A click-n-play tool can be either generic, such as Game Maker (Figure 4) or focused on the creation of games belonging to a specific game genre, such as first-person shooters, role-playing games (RPGs), adventure games and so on. Either way, there have been several attempts to make the ultimate game creation tool [Ambrosine, 2009].

[7] Common sense agrees that humans are generally much better suited to solving problems presented "visually" (e.g., in pictures) than those presented in text or numbers [Mongan & Suojanen, 2000].

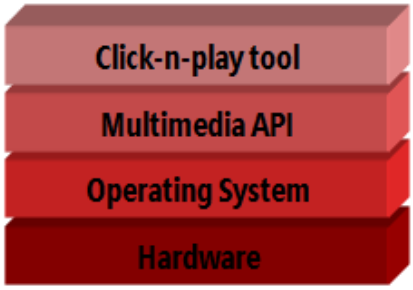


Figure 3 – Click-n-play tools modify the top abstraction layer

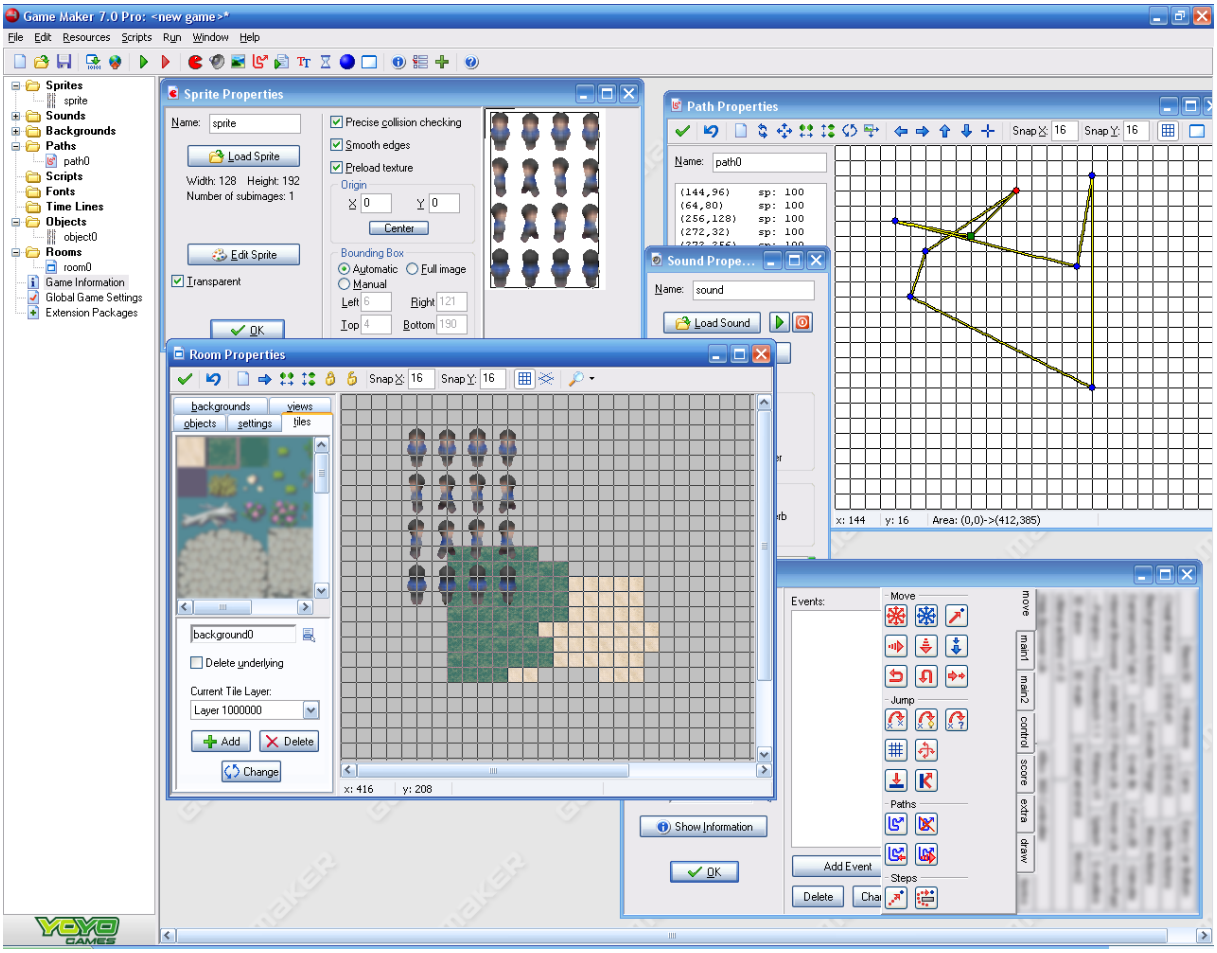


Figure 4 – Game Maker

2.3.1 Discussion: Click-N-Play Tools Effectiveness in Game Development

Click-n-play tools were certainly a great achievement in order to help beginner or amateur game designers and programmers to accomplish their tasks. The slogan employed by GameSalad[8], a web game creation tool for Mac OS X, summarizes such an accomplishment very well: “Game Creation for the Rest of Us”. Developer communities created around some of the tools are very inspiring, while other tools can even transcend the game development

[8] http://gamesalad.com/landing/overview

domain by turning the game creation process into an educative discipline. That is the case of Kodu[9], a visual game programming environment by Microsoft Research designed for children.

Being able to finish the creation of a complete game with a few mouse clicks is very impressive indeed. However, the possibilities turn out to be limited. Some types of games can certainly be made, but this approach does not seem adequate for real-world games [Wiering, 1999]. Click-n-play tools currently do not address the complexity required by the creation of more sophisticated games, and this is reflected by the lack of their adoption by the game industry. For instance, Mark Overmars, the creator of Game Maker, revealed that while the tool is downloaded more than 100.000 times per month, most people use the free (limited) versions and "very few companies use it"[10]. The licensed versions are generally ordered by schools, not by the game development industry.

In many cases, click-n-play tools do not provide a better way to create games than "real" programming languages. On the other hand, some of these tools provide visual aids for programming language syntax and constructs (variable declaration, variable assignment, instruction blocks, if-then-else branches, loops, etc.), as shown in Figure 5. By trying to simplify programming concepts with visual counterparts, such an approach can even be appropriate to simple cases, but it is inflexible and a not very productive way to program in the majority of cases, where more elaborated behaviors are desired.

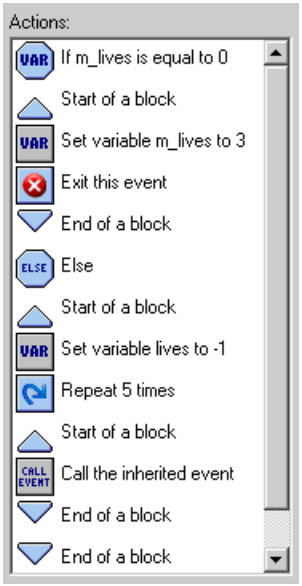


Figure 5 – Specifying flow control visually in Game Maker

[9] http://research.microsoft.com/en-us/projects/kodu/

[10] Personal contact by e-mail.

Such a lack of programming intuitiveness made end-users and click-n-play tool manufacturers to realize that the creation of digital games demands “more flexibility and control than the standard actions” [Overmars, 2004]. The solution to this problem was to combine script languages with click-n-play tools, in order to make game behavior programming more natural, at the price of renouncing to a full visual game development. RPG Maker XP, for example, is equipped with the Ruby Game Scripting System (RGSS), a Ruby-based script language. The latest versions of Game Maker, on the other hand, provide a built-in programming language to end-users, named Game Maker Language (GML). Such languages enable users to extend the designed game with code, such as building expressions that can be used as values in game actions (move actions, draw actions, score actions, etc.).

When programming with such languages, users have some code editing support, such as syntax highlighting and code completion (Figure 6). However, while they certainly provide more power to click-n-play tools, some problems can be pointed out. First, they require end-users to learn a new language (perhaps their first language) and to have some programming skills. This may diverge from the original purpose of such tools (to be “visual programming” environments).

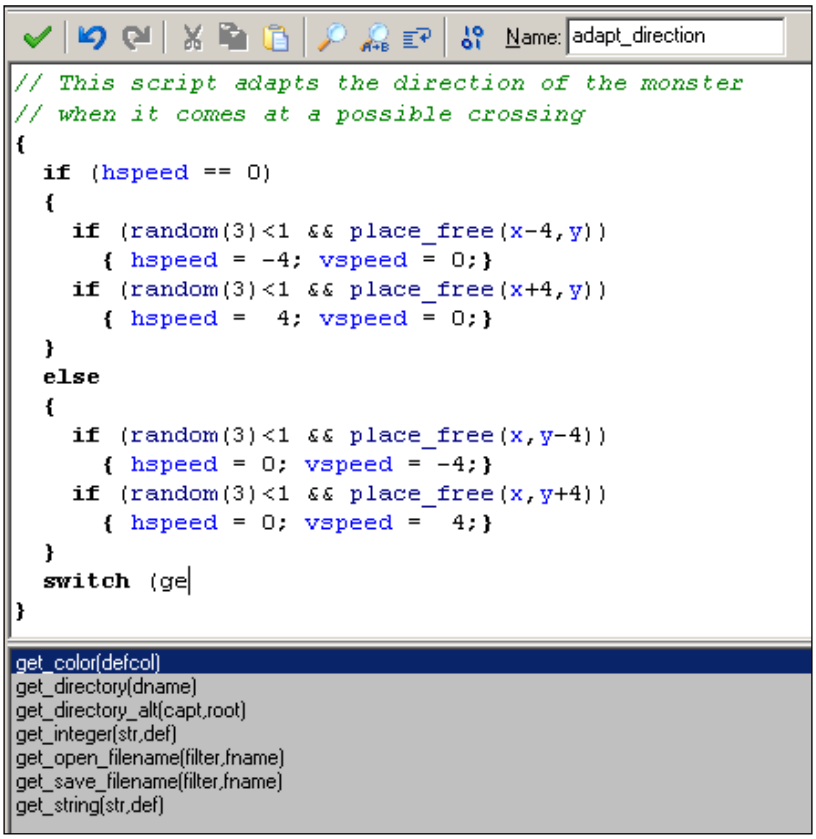


Figure 6 – Creating a script with GML, the Game Maker built-in programming language

Some may say that these built-in languages are not intended to be employed by all users, but only by advanced users. But once earning programming expertise, however, users might prefer to have the benefits of true object-oriented programming languages, with the support of robust integrated development environments with full editor and debugging support, instead of working with error-prone scripting languages inside an environment which was not originally conceived for codification. Moreover, development productivity is much more than just having script keywords highlighted. It is supported by a set of integrated concepts and features, such as refactoring, code and modeling synchronization, test automation, configuration management, quality assurance, real-time project monitoring, domain-specific guidance and process integration.

2.4 Game Engines

If from one hand click-n-play tools majorly branched out to enable non-programmers in the creation of simpler games, **game engines** were created by the actual game industry as the result of applying Software Engineering concepts to the digital games development. Game engines are focused on assisting development teams in the creation more complex games, being considerably more flexible and powerful than click-and-play tools. Engines build on top of multimedia APIs to hide low-level implementation details and support more abstract game development tasks (entity rendering, world management, game events handling, etc.) through a programmatic interface in which the game logic can be plugged in (Figure 7). Commercial game engines can reach an acquisition cost of about half a million dollars, while others are completely free. A comprehensive list of (3D) game engines can be found in the DevMaster.net engine database[11].

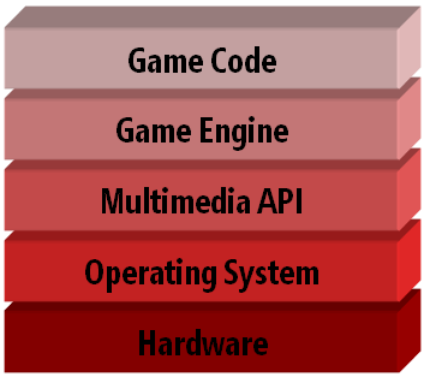


Figure 7 – Game engines introduced a new abstraction layer in game development

In order to be more effective, game engines typically narrow their focus down to a subset of digital games. For example, a 3D game engine is different from a 2D isometric

[11] http://www.devmaster.net/engines

game engine. If the engine is built in a modular architecture, it can be reused to create a great diversity of games belonging to its target domain, which consume and configure only the required modules [Rollings & Morris, 2000].

Some game engines provide visual tools to game programmers to help them to accomplish a specific task, such as level editors and sprite editors. These tools, however, are different from previously analyzed click-n-play tools. Game engine tools are focused on a specific aspect of the game development process, not being targeted at the creation of a complete game, and their output can be modified and consumed programmatically by developers.

2.4.1 Discussion: Game Engines Effectiveness in Game Development

Game engines became the state-of-the-art in the development of many industrial titles. By providing more abstraction, knowledge encapsulation and a reusable game development foundation, they allowed the game industry to reach an unparalleled productivity level. Multiple successful games such as Counter-Strike and Team Fortress, for example, were only able to satisfy time-to-market demands because they were built upon a powerful game engine.

On the other hand, due to the inherent complexity of game engines, the learning curve for mastering such tools is somewhat high. The demands for understanding the game engine architecture, interaction paradigm and programming peculiarities can turn their use into an unintuitive experience at first. That is the reason why many of today's game engines still present complexity and lack of usability as one of their most cited deficiencies.

Subsequently, using a game engine may involve considerable costs, such as acquisition costs, training costs, customization costs and integration costs [Albuquerque, 2005]. In addition, one of the major difficulties in game engine development is the industrial secrecy. Since such projects involve great investments, many organizations hide their architectures and tools in order to have some advantage over their competitors [Rocha, 2003]. For example, it may be difficult to find comprehensive studies about the applicability of design patterns in game engines [Madeira, 2003]. Game engine developers are not close from having something like "game engine workbenches" to aid the creation of such tools.

Game Development processes (see Section 2.6) have traditionally been lacking awareness and deeper integration with game engines. Despite using visual modeling extensively, such processes also do not focus on code visualization or generation. They commonly use models as documents, not as source artifacts. Therefore, the reusability provided by game engines is still attached to one-off development approaches. Game engines and the

game development processes are not benefited from deeper traceability and automation mechanisms, such as code generation and synchronization.

Game engines may also depend upon the (generic) development environment used, which may not provide all the desired foundation for the target game domain(s). In short, the multidisciplinary environment in which digital games development is inserted as well as the great diversity of tools used throughout the project life-cycle (game engines, level editors, sound editors, 3D modelers, etc.) demands not only a richer tool integration but a closer alignment with the development process as well, in order to enhance developer experience and productivity.

2.5 Industry Alternatives to Game Engines

As an alternative to development approaches solely based on game engines, the game industry has been observing the rise of **generic game development toolsets**, such as the Microsoft XNA [Reed, 2010], and **integrated graphical environments** backed up by game engines, such as Unity [Menard, 2011]. With a strong focus on productivity, both are targeted at lowering the entry barrier for game development, like click-n-play tools, but still enable game developers to create industrial-level games with an acceptable quality level for the market, as it happens with game engines.

Such approaches offer a set of tools and building blocks to handle game development tasks with more abstraction than a game engine, such as handling the content pipeline, multiplayer networking, etc. They either integrate with development environments, turning them into a broader framework aimed at game development, or provide their own graphical environments. Generic toolsets are majorly based on a programmatic paradigm and therefore underexplore code generation capabilities. Integrated graphical environments, on the other hand, use visual editors as the primary method of development.

From a software product line perspective, one of the main disadvantages of such approaches is that they are not domain specific, in the sense that any type of game can be created from them (arcade, adventure, casual, etc.). While starter kits for some specific genres are available, many opportunities are still under-explored by such platforms with regards to reuse and abstraction in the context of a family of similar games. In summary, although more streamlined APIs and visual editors atop game engines provide increased abstraction, we believe other automation assets are still underexplored, such as domain-specific languages and guidance automation. As a consequence, it is not possible to find in the literature guidance for developing such toolsets and graphical environments for a specific game domain.

2.6 Game Development Processes

While the previous sections in this chapter discussed game development technologies and tools, game development *processes* also play a key role in understanding the challenges and state-of-the-art of digital games development. Some of such processes are described in this section, which therefore contributes to assess more work related to this research.

2.6.1 The Early Phases: Ad Hoc Game Development

In the very beginning (1950-1960), rudimentary visual systems and the lack of complex rule systems made game development to be characterized by *ad hoc* approaches: there was no systematic approach or scientific method employed in the development of digital games [Araujo, 2006]. However, as noticed in Section 2.1 (From Assembly to Doom), there were many engineering challenges especially due to low-level programming, memory and performance constraints. The lack of an industry made digital games development to be low-scale and experimental, in which small teams of one or two people employed a *code-and-fix* approach, which was chaotic but presented results capable of fulfilling the demands of that time [Flynt, 2005].

2.6.2 Waterfall Processes

With the popularization of videogames in the 70s, the demand for more powerful, complex, immersive and multidisciplinary games made development processes in the area to evolve from ad hoc to waterfall. The game industry was then following the typical linear sequence of activities employed in waterfall processes: Requirements, Design, Implementation, Verification and Maintenance [Sommerville & Flynt, 2007]. Development teams evolved from a small team of essentially programmers to professionals with distinct skills and expertise, such as engineers (programmers, testers, architects), designers (game designers, graphical designers, level designers, usability designers, sound designers), managers (project managers, functional managers, producers) and, depending on the game, business analysts, psychologists, anthropologists and other profiles [Chandler, 2006].

The Game Waterfall Process (GWP) was introduced as an adaptation of the generic waterfall model [Flood, 2003] [Flynt, 2005]. It introduced, for the first time, game-specific development artifacts such as game specification documents and story bibles. GWP suggested five main distinct phases: Requirements (creation of the Game Design Specification, Art, Technology and other documents), Design (refining of specifications, creation of prototypes, sketches and conceptual art), Implementation (development of scripts, source code, graphics and audio assets), Validation (gameplay and usability tests) and Maintenance (deployment, distribution and marketing activities).

2.6.3 Iterative, Incremental and Agile Processes

In the 90s, iterative and incremental development processes were adopted by the digital games industry, as such processes welcomed changes in the game lifecycle much more than waterfall processes. The “divide and conquer” approach of iterative and incremental processes, which delivered to end-customers partial solutions by means of increments, also contributed to incorporate early feedback into a game lifecycle.

Initially, iterative and incremental processes such as the Unified Process (UP), eXtreme Programming (XP), Feature-Driven Development (FDD) and Scrum were adopted by the game development industry as is. In a next step, the Game Unified Process (GUP) [Flood, 2003] proposed a spectrum of process approaches ranging from the RUP (Rational Unified Process) to XP, according to the game development organization. Special needs for documentation and process formalisms would be more aligned to RUP, whereas more informal and flexible environments would lean toward XP. Nevertheless, GUP still did not address some critical issues in game development, such as the integration between design disciplines (game design, graphical design, level design and usability design) and computer science disciplines (Software Engineering and programming). GUP also does not specify a flow of activities to create a digital game.

The eXtreme Game Development process (XGD) [Demachy, 2003] proposes an interpretation of the XP methodology to digital games. However, such an interpretation is criticized as being superficial [Araujo, 2006] and not providing enough structured guidance, in part because it is an agile methodology.

Although GUP and XGD had a considerable repercussion in game industry circles, such as in the Game Developers Conference, there are no indications, recommendations nor reference models for employing such processes, in part due to the secrecy nature of the game industry. On the other hand, some related initiatives in the academy can still be found, such as the Agile Game Process (AGP) [Araujo 2006] and the Prescriptive Methodology for Computer Games Development [Carvalho, 2006], although they lack an evaluation process to determine their validity. Other authors also applied agile methodologies to game development in order to enhance change management during development and iterations in game design [Keith, 2006] [McGuire, 2006] [Miller, 2008].

2.6.4 Model-Driven and Componentized Processes

As an attempt to improve reuse, Folmer [2007] applied Component-Based Development (see Section 3.1) to digital games development. His proposal consists in establishing a Commercial of the Shelf (COTS) culture for digital games development, in which pre-built components such as physics or 3D engines can be assembled and customized by game developers, in-

stead of being developed from scratch. He refined the idea of developing games with components, presenting a reference architecture that outlines the relevant areas of reuse, taking game development peculiarities into account.

Folmer's architecture is inspired by the published architectures of two games and a real time system, along with the layered reference architecture for component-based development as proposed by Collins-Cope & Matthews [2000]. However, his architecture is generic: it encompasses different game genres and is targeted at digital games in general. This may overlook automation and reuse possibilities for specific game sub-domains. For example, the top ("game interface") layer relies on a generic database and generic concepts such as "game logic", without getting into more specific details such as game entities and flow. Moreover, his approach was built from the solution domain (actual digital games architecture and code), not taking into account the problem domain or its related tasks such as Domain Analysis. In fact, the approach is not contextualized in the literature of Software Product Lines.

The author also recognizes that the validity, accuracy and complements of the reference architecture are open for discussion, since it was based on a limited number of samples. Finally, integration and adoption costs are recognized by the author as one of the main challenges of the COTS approach. While it was found out that cost reductions and more importantly reductions in development time can be achieved when developers use off-the-shelf components rather than develop them from scratch, many game developers struggle with component integration and managing the complexity of their architectures.

As game engines evolved from APIs to a more comprehensive toolset encompassing tools and editors to help developers in the creation of digital games, game development processes started to rely on a certain level of modeling as long as such engines were used. The Unreal engine, for instance, provides UnrealScript, a game scripting language that supports game-specific concepts (states, time, properties, networking, etc.), as well as UnrealKismet, a visual scripting language aimed at level designers. However, scripting languages like UnrealScript remain at a fairly low programming level, which raises concerns as to the abstraction level that can be attained for a game DSL [Dobbe, 2007]. Moreover, since engines like Unreal have applicability in many different game sub-domains, their somewhat generic built-in languages do not benefit from the expressiveness of languages more focused in specific (sub-)domains.

Focusing on more domain-specific approaches, the XML Game Consortium (XGC) created GameXML, a collection of XML specifications which describe and script computer simulation engines. The consortium maintains XML-based game languages for specific domains such as simple board games (GameXML/ABG) and role-playing strategy games

(GameXML/RPS). Nonetheless, although XML is a valid way of expressing a DSL syntax [Fowler, 2005], its verboseness and lack of a more intuitive visual representation limits adoption and productivity. This was promptly found out by the authors of the Video Game Language (ViGL) [Sutman & Schementi, 2005]. The authors of such an academic project realized that XML is not ideal for control-flow features and has a bloated syntax, and moved to a mixed declarative (XML) and imperative (embedded code) approach. On the other hand, having embedded code inserted into the declarative language syntax makes it difficult to have a clear separation of concerns, therefore reducing the benefits of the abstraction provided by domain-specific languages.

Dobbe [2007] introduces a new domain-specific language for computer games, hosted in its own special environment (instead of an IDE) and integrated with a proprietary game engine (Cannibal Game Engine). The author separates game development in three major areas: arts, sound and design. He points out that although tooling is provided in the game industry for the arts and sound areas, game development is still performed by generic languages and tools. An interesting finding from the author which is aligned to the guidelines of this research is that his Cannibal Game Engine had to go through some modifications prior to the DSL integration, such as a better extensibility support and the implementation of an event-driven paradigm.

Although the author's experience provides lessons learned on applying DSLs to digital games, it does not intend to define a process nor even guidelines on how to perform Domain-Specific Game Development. The work is not contextualized under the software reuse and SPL literature. Through an ad hoc approach, the author determined that the following areas need to be covered by a game DSL: objects, interaction, rules and storyline. Cross-DSL integration is not supported in the language level, but through the underlying framework. Finally, the details of the proposed DSL are removed from the public version of his report, due to industry secrecy.

Reyno and Cubel [2008] propose the use of Model-Driven development, by means of Platform-Independent Models (PIMs) and Platform-Specific Models (PSMs), to create prototype 2D platform games for PC. In their approach, the authors used the Unified Modeling Language (UML) diagrams, eventually extended with stereotypes to define the structure and behavior of a platform game. While the effectiveness of UML stereotypes as a mechanism to raise abstraction levels in Model-Driven Development is an open discussion [Greenfield et al., 2004] [Kelly & Tolvanen, 2008], Reyno and Cubel acknowledge that such diagrams are closer to software engineers than to game developers, therefore the process lacks conceptual models targeted at game developers. Moreover, the expressiveness of such diagrams is constrained by the visual syntax of UML elements (activities, classes, etc.), which may not be

suitable for cases in which other richer visual designer alternatives are desired. Finally, their game control (input) meta-model is not scaled to support game event triggers other than player input.

Barros et al. [2006] used models and frameworks toward the creation of simulation games aimed at teaching project management. They propose five steps as part of a three-layered model approach for describing the behavior, story and graphical representation of an educative simulation game. While experiments concluded that the approach abstracts low-level details, enables the separation of responsibility areas and enhance productivity in the creation of educative simulation games, it does not encompass extensibility for supporting the implementation of more complex behaviors. Likewise, the proposal does not provide graphical designers or deeper tool integration to aid developers in the creation of the game models. Moreover, applying such an approach beyond the educative simulation games domain is yet to be explored.

Maier & Volk [2008] discuss first findings of an ongoing case study, in which language workbench concepts are applied to the creation of level editors for “classic” games, such as Pac-Man. The case study encompasses two steps: first, a product line developer designs the meta-model (domain concepts) and visual representation of a level-editing DSL. That generates a level editor tool used by product developers to create games. There is an additional textual DSL used by product developers to describe game logic. Code extensions to the level editor are also supported. The authors mention that one advantage of their toolset is its “evolutionary methodology” support, in which the DSL specification can be updated at will by the product line developer in order to re-generate an updated version of the level editor. However, the authors do not discuss the impact of that on configuration management (more specifically artifact versioning) neither the implied costs of interactions between product line developers and product developers during the development lifecycle of the same game.

As far as evaluation goes, the authors mention that the generated level editors implied in a “plain reduction of development time”, enabling prototypes to be “literally created within hours”. They also argue for increased productivity and simplicity. Mastering meta-modeling and fine-tuning (extending) factory functionality are mentioned as challenges faced by developers. Finally, the authors recognize that the need for multiple levels in the same game requires launching multiple instances of the level editor. On the other hand, the authors are not concerned with defining an approach for creating game SPLs, but focus on a specific sub-domain (level editing) in which only one DSL is created per “factory”. The work does not take into account any integration with other game assets such as game engines.

Moreno-Ger et al. [2008] suggests a storyboard-driven approach, called e-Adventure, as a methodology for developing educational games. They aim at moving beyond the current

state of edutainment products which “combine the entertainment value of a bad lecture with the educational value of a bad game”. The authors focus on the specific sub-domain of adventure game storyboards, advocating that content writers (such as instructors, in educative games) are never placed in the center of game development processes, which typically focus on technical roles. e-Adventure is pretty much a game SPL instance that supports dynamic updates across iterations. It provides a collaboration model in which domain (content) experts and developers work together using documents that describe the game’s contents and other relevant features. Developers equip domain experts with a suitable domain-specific markup language and an application generator. Domain experts mark up documents with the language and process them automatically with the generator, yielding the final executable application. An end product in the e-Adventure process includes the DSL, documents with the marked-up storyboard, art assets, the game engine and the final game.

The authors recognize that the functionality of these environments becomes a factor that limits creativity, because the language constrains what can and cannot be done. As it happened with Maier & Volk [2008], their process advocates for having not only the game but also the language and the game engine to be refined at each iteration. They mention that although this could be considered a bad practice from a Software Engineering perspective, that’s not true in every case and they prefer such “change embracing” approach. On the other hand, we believe that the language design phase should incorporate enough prototyping, sample analysis and validation iterations in order for a more stable language to emerge, simplifying configuration management tasks such as asset versioning. Required modifications can then be handled as game SPL extensions, whose incorporation into the game SPL can be evaluated as part of the factory feedback cycle. The authors did not mention whether an evaluation process was used, however they do present a complete case study in which the toolset is used to develop an adventure game that acts as an initiation module for a course on safety regulations in construction.

Hernandez & Ortega [2010] developed a Domain-Specific Language (Eberos Game Modeling Language 2D, or GML2D) targeting the modeling of two-dimensional games. The language supports sprites, animations, entities (whose logic is defined by means of state machines), “action scripts” that can be coded to extend the language, collision detection, background music and sound effects. The authors are not concerned with providing a systematic approach to the creation of DSLs for game domains, but focus on sharing their experience with a specific game DSL. For instance, they did not detail how the DSL and its syntax were designed, apart from mentioning “discussions from domain experts and five years of game development experience”. The approach was evaluated by modeling two games and comparing the difference between the amount of work required to write the games from

scratch versus the amount required using the DSL. Details are provided in the Appendix B, which also presents a compilation of Domain-Specific Development evaluation for other domains.

We disagree with two main points raised by the authors of Eberos GML2D. First, it is mentioned that they believe the 2D gaming domain is specific enough to be expressed by a single DSL. On the other hand, one of the lessons learned from the spike solution used in this research (SharpLudus Adventure) was that this is not always the case. A 2D game has many sub-domains that can be more effectively expressed with specific DSLs for each of them. Partitioning a domain into more specific sub-domains is vital to evaluate the automation potential of such a domain through a divide-and-conquer approach, enabling game SPL designers to focus on sub-domains with the best ROI and come up with more specific and effective languages and tools. Secondly, the authors consider that game engines should be consumed as is by the generated code. Our experience shows on the other hand that in many situation adapters are required not only to make the generated code more easily to be consumed by the game engine, but also to reduce the complexity of the generation scripts, whose development is error prone. Rather than assuming that all game engines are ready to be efficiently consumed by generated code, our approach is concerned with promoting game engines to genuine domain frameworks, more aligned to Model-Driven Development techniques such as framework completion. This opinion is also shared by Dobbe [2007].

Some works in the literature can also be found about the application of Software Product Lines to the development of games for mobile devices. Such approaches commonly tackle the high variability of this sub-domain caused by the big diversity of phones and manufactures, bringing up variation points such as different screen sizes, different API implementations and limited application size. Nascimento [2008] defined a practical approach for implementing core assets in a mobile games software product line. Encompassing component modeling, component implementation and component testing, it defines a prescriptive approach with well-defined phases, activities, inputs, outputs and roles. Alves et al. [2005], on the other hand, used mobile device games to evaluate the combination of reactive and extractive approaches for developing Software Product Lines [Krueger, 2001], handling variations with aspect-oriented constructs.

Finally, some game development approaches deal with modularization at a higher, conceptual level, taking game design elements into account. One example is the description of games according to the following attributes trio: mechanics, dynamics and interface [Araujo, 2009]. The game mechanics refer to the game rules, challenges, incentives and world model. The game dynamics refer to the game script, i.e., its narrative or context (e.g., space adventure *versus* medieval war). Finally, the game interface refers to its graphical rep-

resentation (realistic *versus* cartoonist, adult *versus* childish, 2D *versus* 3D, etc.). Such a separation is relevant for game SPLs since it might lead to identifying variability points.

2.7 The Future of Game Development: Tendencies and Proposals

Trends in the abstraction of software development (in general) provide indications that game engines could still play a bigger role in game development automation. Roberts & Johnson [1996], for example, described a recurring pattern that reveals how software development automation is carried out:

- **Phase 1:** after developing a number of systems in a given problem domain, a set of reusable abstractions for that domain is identified, and then a set of patterns for using those abstractions is documented.
- **Phase 2:** a runtime is then developed, such as a framework or server, to codify the abstractions and patterns. This allows the creation of systems in the domain by instantiating, adapting, configuring, and assembling components defined by the runtime, hence automatically applying the patterns.
- **Phase 3:** languages are then defined, and tools are built to support the runtime, such as (visual) editors, compilers and debuggers, which automate the assembly process. This helps a faster response to changing requirements, since part of the implementation is generated, and can be easily changed.

Game engines are situated in the second of these three "pattern-runtime-language" phases. However, as Roberts & Johnson point out, although a framework (such as a game engine) can reduce the cost of developing an application by an order of magnitude, using one can be difficult. Mapping the requirements of each product variant onto the framework is a non-trivial problem that generally requires the expertise of an architect or senior developer. Language-based tools (the third stage) automate this step by capturing variations in requirements using language expressions, encapsulating the abstractions defined by a framework, helping users think in terms of the abstractions and generating framework completion code. Language-based tools also promote agility by expressing concepts of the domain (such as the properties or even features of digital games) in a way that customers and users better understand, and by propagating changes to implementations more quickly.

Aligned with the creation of language-based tools, an emerging trend is to make models as first-class citizens of game development, similar to source code. Models provide a richer medium for describing relationships between abstractions, delivering more efficiency and expression power than source code. By using a visual DSL, models can be used not only as documentation but as input that can be processed by tools in other stages of the development process, promoting more automation during the project life-cycle.

Every new paradigm builds on the strengths of their predecessors, while addressing some of the weaknesses that give raise to their chronic problems [Greenfield, 2004]. This was true for the introduction of multimedia APIs, click-n-play tools and now for game engines. As with multimedia APIs, game engines were a great contribution to game development, risen from a deeper integration with Software Engineering concepts, and will also last for a long time. However, this work believes that as with multimedia APIs, game engines can act as an important foundation upon which more abstract layers can be built. This sounds especially applicable considering the hiatus faced by game development today (Figure 8), in which easy-to-use script languages and click-n-play tools are many times not flexible enough, contrasting to the powerful yet many times too complex world of game engines. Such a trend (the move to Figure 8's second quadrant) can be already observed in integrated graphical environments, such as Unity (see section 2.5).

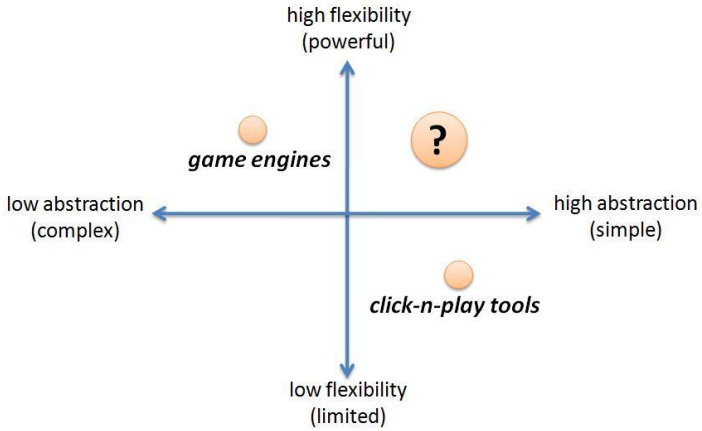


Figure 8 – Contextualizing the hiatus addressed by this research

The key claim of this research, therefore, is that game engines can be further explored, by means of domain-specific processes, patterns, frameworks, tools and especially languages toward a SPL-based approach to situate game development in an industrial stage, reuse these assets systematically and automate more of the game development.

Figure 9 presents the abstraction layers envisioned by this research for the future of digital games development. Building atop current techniques, the vision relies on models, created with visual languages, to provide a higher level of abstraction for development tasks specific to a given game domain. The models generate work products such as code that "completes" a game engine, whose complexity is abstracted by means of the models themselves. Developers can still provide their own custom code, directly interacting with the game engine or its underlying multimedia API, to define more complex game behavior which is not supported by the game SPL and its asset as built-in. The vision also encompasses non-prescriptive guidance integrated with development assets (visual designers, semantic validators and game engine themselves) that span throughout the game development life-cycle.

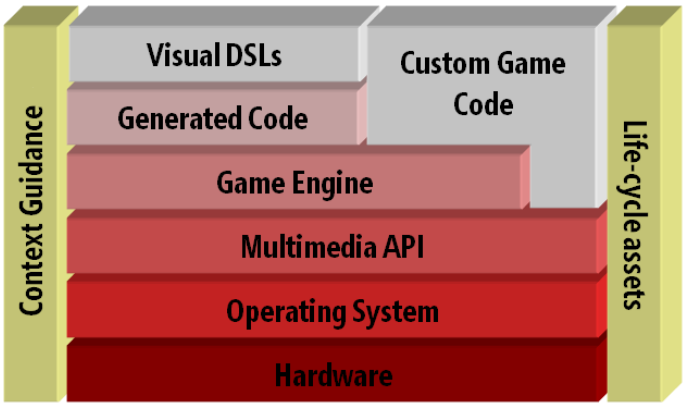


Figure 9 – A new context: higher abstraction through DSLs and process integration

2.8 Chapter Summary

This chapter focused on the evolution of digital games development. From assembly language to the advent of game engines, techniques were discussed along with their respective advantages and drawbacks. Game development processes were also presented, contributing to situate this research with related work. At the end of the chapter, some trends and proposals envisioned by this research were also presented.

In order for digital games development to accomplish such a vision and fully reach the aforementioned third automation level designed by Roberts & Johnson [1996], it is important to understand the applicability of Domain Engineering activities to the digital games development, and how to instantiate them. This requires a better comprehension of both software product lines and domain-specific languages concepts, and whether they are a viable fit for digital games development. Such areas are the focus of the next chapter.

3. BUILDING APPLICATION FAMILIES

SPLs and software factories are concerned with turning the current software development paradigm, based on craftsmanship, into a more effective manufacturing process. In order to raise the level of abstraction toward software industrialization, an investment in reusable production assets is required, encapsulating knowledge in languages, patterns, frameworks and tools. However, an ad hoc approach to reuse tends to produce frustration and marginal results [Greenfield et al., 2004], reinforcing, for example, the *Not Built Here Syndrome*. To realize the benefits of reuse, a more mature approach should be adopted. It should involve the identification of the common sub-problems in a given domain and develop integrated collections of production assets that can be reused to solve those problems predictably, especially in the context of a family of systems [Parnas, 1976].

This chapter discusses some approaches in the literature for achieving such a systematic reuse and also for building on top if, culminating with one of the core foundations of SPLs and software factories: domain-specific languages. Later on, a discussion is carried out to investigate whether, and to what extent, software factories are a good fit for digital games development.

3.1 Component-Based Development

Component-Based Development (CBD) explores the benefits of reusing software components, involving the selection of components from an in-house library or the marketplace to build products. When the term “software reuse” was coined for the first time [McIlroy, 1968], the idea of software components was presented. Components were at that time usually compared to routines, available in families arranged according to precision, robustness, generality and performance.

Almost thirty years later, Sametinger [1997] defined components as reusable, self-contained and clearly identifiable artifacts that describe and/or perform specific functions and have clear interfaces, appropriate documentation and a defined reuse status. Szyperski [2002] adds that a software component is a unit of composition with contractually specified interfaces and explicit context dependencies only, which can be deployed independently and is subject to composition by third parties.

Some CBD methods include *Catalysis* [D'Souza & Wills, 2001], which introduces three CBD phases: problem domain elaboration, components specification and components internal design. The *UML Components* approach [Cheesman & Daniels, 2001], on the other hand, is based on Unified Modeling Language (UML) extensions by means of stereotypes. It introduces and refines component-specific activities such as component identification, com-

ponent interaction and component specification, followed by provisioning and assembly phases. Such CBD methods are criticized by lacking guidelines on how to apply variability implementation techniques [Anastasopoulos & Gacek, 2001], not being detailed enough as a process (lacking steps, inputs, outputs, roles, etc.), how to structure an extensible architecture considering commonalities and variability and, finally, how to combine components in order to derive products [Nascimento, 2008].

3.2 Domain Engineering

Instead of reusing an individual component, it is much more advantageous to reuse a whole design or subsystem, consisting of the components and their interconnections [Gomaa, 2005]. This means reuse of the control structure of the application, including artifacts of requirements architecture, code and tests. Thus, the reuse of such artifacts has much greater potential than component reuse because it is large-grained reuse.

Domain Engineering is the activity of collecting, organizing, and storing past experience in building systems or parts of systems in a particular domain in the form of reusable assets, as well as providing an adequate means for reusing these assets when building new systems [Czarnecki & Eisenecker, 2000]. The first Domain Engineering approach was proposed by Neighbors [1980], along with a prototype called *Draco*. The main ideas introduced by Draco include Domain Analysis, domain-specific languages and components as sets of transformations.

From there, the Domain Engineering concept and approaches evolved incrementally. The *Conceptual Framework for Reuse Processes* [STARS, 1993] established a framework for considering reuse-related Software Engineering processes, how they interrelate, and how they can be integrated with each other and with non-reuse-related processes to form reuse-oriented life-cycle process models that can be tailored to organizational needs. The *Organization Domain Modeling* method [Simos et al., 1996] established three main phases (Plan Domain, Model Domain and Engineer Asset Base) that can be mapped to today's Domain Engineering phases (Domain Analysis, Design and Implementation). The reuse-driven *Software Engineering Business* (RSEB) [Jacobson et al., 1997] employed a UML and use case-driven approach for reuse, with distinct processes of Domain Engineering and Application Engineering, and focus on modeling variability and maintaining variability traceability across different development disciplines. *FeatuRSEB* [Griss et al., 1998] overcame some of the RSEB limitations, by refining its Domain Analysis activities and by integrating Feature-Oriented Domain Analysis (FODA) [Kang et al., 1990] into the approach.

The *Feature-Oriented Reuse Method* (FORM) [Kang et al., 1998] was also based on capturing domain commonality and variability through features. It then used the domain anal-

ysis results to develop domain architectures and components. The *Odyssey Domain Engineering* [Villela, 2000] leveraged Domain Engineering approaches toward a systematic sequence of activities to search and recover components in software reuse environments. It also provided a traceability mechanism between requirements and other work products using tool support. Almeida [2007] proposed the *RiSE Process for Domain Engineering* (RiDE), comprehending the three phases of the Domain Engineering but with a special focus on the Domain Design phase, in which the product line architecture is established based on similarity functions according to feature models.

As opposed to Lucrédio [2009], who presents a Model-Driven Development approach for software reuse, the RiDE process does not explore the use of domain-specific languages. While RiDE claims that the essence of the idea behind DSLs ("to build reusable assets") is similar to Domain Engineering or Software Product Lines, it is possible to identify at least two major peculiarities in the use of visual DSLs as part of Domain Engineering approaches:

- In contrast to domain architectures, which are described by means of code-level concepts such as components, classes and interfaces, DSL models are expressed in more abstract syntax and semantics, using concepts closer to the problem domain. Therefore, there is a distinction between the creation of *application* core assets (such as architectures) and *development* core assets (such as DSLs) [Lenz & Wienands, 2006].
- While DSLs still benefit from the encapsulation of common behavior in reusable components, they actually play a major role as a **variability** mechanism that enables developers to model the distinct behaviors of each instance of the SPL. DSL transformations into lower level assets, such as generated code, are then responsible for configuring the reusable components, task that otherwise would have to be done manually at code level.

3.3 Software Product Lines

By 1999, it was concluded that Domain Engineering approaches had not proved to be as effective as expected. According to Bayer et al. [1999], there are basically three reasons for this: misguided scoping of application area, lack of operational guidance and overstressed focus on organizational issues.

Until that time, software reuse processes were only related to Domain Engineering. Following that, a new trend called Software Product Lines (SPL) started to be explored and began to be seen as one of the most promising advances for efficiency in software development. According to Clements & Northrop [2001], a software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs

of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. In a SPL approach, the reuse is planned, enabled, and enforced – the opposite of opportunistic. All the assets are designed to be reused and are optimized for use in more than a single system. The reuse in SPLs is comprehensive and profitable [Nascimento, 2008].

SPLs have proved to be a very successful approach to intra-organizational software reuse [Bosch, 2002]. However, until the late 90's, there were few available guidelines or methodologies to develop and deploy product lines beyond existing Domain Engineering approaches. One of the first attempts to use component-based development in the context of Software Product Lines is *GenVoca* [Batory & O'Malley, 1992], a design methodology for building architecture-extensible software via component addition and removal. The *Product Line Software Engineering* (PuLSE) methodology [Bayer at al., 1999] defined elements for enabling a SPL infra-structure, providing the technical know-how to make the SPL operational and enabling adaptation, evolution and deployment of the SPL. The *Family-Oriented Abstraction, Specification and Translation* (FAST) process [Weiss & Lai, 1999] provided a systematic approach to analyze potential families and to develop facilities and processes for generating family members. It introduced a Domain Qualification sub-process consisting of an economic analysis of the family, which requires estimating the number and value of family members and the cost to produce them. The *Komponentenbasierte Anwendungsentwicklung* (KobrA) approach [Atkinson et al., 2000] was created to address circumstances where no processes or well-defined products pre-exists in an organization before it attempts to establish a SPL. The *Component-Oriented Platform Architecting Method* (CoPAM) [America et al., 2000], on the other hand, enabled developers from different product families to share know-how from their respective family engineering methods.

Clements & Northrop [2001] compiled several concepts of software reuse and product lines into the *Framework for Product Line Practice*, proposed by the Software Engineering Institute (SEI). It introduced management activities, which act on both technical and organizational levels taking into consideration budgets, schedules, team effort, and all the managerial aspects. The *Pervasive Component Systems* (PECOS) project [Winter et al., 2002] enabled component-based development for embedded systems. Kang et al. [2002] presented an extension of their previous work, FORM [Kang et al., 1998], to support the development of Software Product Lines. It proposes two sub-processes: Asset Development and Product Development, but is also concerned with business aspects such as the Marketing and Product Plan (MPP). The *Product Line UML-Based Software Engineering* (PLUS) [Gomaa, 2005] extends the UML-based methods for single systems development to support Software Product Lines, providing modeling techniques and notations for product line engineering. It pro-

poses three SPL macro-processes (Requirements, Analysis, and Design Modeling) followed by Application Engineering. Finally, Pohl et al. [2005] introduced the *Software Product Line Engineering* (SPLE) Framework, based on the two traditional Domain Engineering and Application Engineering macro-processes.

3.4 Visual Modeling and Domain-Specific Languages

Model-Driven Development (MDD) supports complex domain variability and automatic implementation of software artifacts by using models that can be processed by tools and are first-class development citizens in the same way as source code. MDD is the combination of generative programming, domain-specific languages and software transformations, which were already being explored back in 1980 [Neighbors, 1980]. Lucrédio [2009] provides a compilation of the main MDD approaches in the industry, such as the Object Management Group's Model-Driven Architecture (MDA), Sun Microsystems' Java Metadata Interface (JMI) and MetaData Repository (MDR), IBM's Eclipse Modeling Framework (EMF) and others.

The purpose of models in MDD is twofold: (i) to serve as a way to capture domain concepts, thus facilitating communication between the different stakeholders (mainly between domain and technology experts); and (ii) to serve as input to automatic processors, for validation, optimization, transformation, code generation or interpretation. In Domain Engineering, the most important artifacts used to achieve this purpose are domain-specific languages (DSLs) and transformations.

A domain-specific language (DSL) is a limited form of computer language designed for a specific class of problems [Fowler, 2005]. It is a usually small and declarative programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain. The key characteristic of DSLs is their focused expressive power. In many cases, DSL programs are translated to calls to a common subroutine library and the DSL can be viewed as a means to hide the details of that library.

Examples of popular domain-specific languages include SQL (Structured Query Language), HTML (Hypertext Markup Language) and BNF (Backus-Naur Form). XML configuration files and graphical user interface (GUI) builders, in which the user experience is quite different from textual programming languages, can also be pointed as examples, in spite of not being usually perceived as DSLs. Today, DSLs span multiple domains, such as financial products, software architectures, databases, video device driver specifications, operating system specialization, web computing, image manipulation, 3D animation, drawing, communication protocols, telecommunication switches, simulation, mobile agents, robot control, par-

tial differential equations and digital hardware design, just to mention a few [van Deursen & Klint, 1998].

Graphical or visual DSLs, as illustrated in Figure 10, use graphical notations for their concrete syntax instead of text. Generating source code from graphical visualizations provides value above working directly with the source code only if the visualizations contain higher-level abstractions, such as business entities constructed from multiple classes, or call processing protocols for telecommunication systems [Greenfield et al., 2004]. In other words, changing the representation of a construct without increasing the abstraction level does not improve productivity. The Unified Modeling Language (UML), for example, does not provide a higher level of abstraction by using a rectangle symbol to illustrate a class in a diagram and later creating the equivalent code representation in a programming language [Tolvanen, 2005]. Adding resources such as stereotypes and notations to UML does not solve the problem. On the other hand, this may increase its complexity in some scenarios.

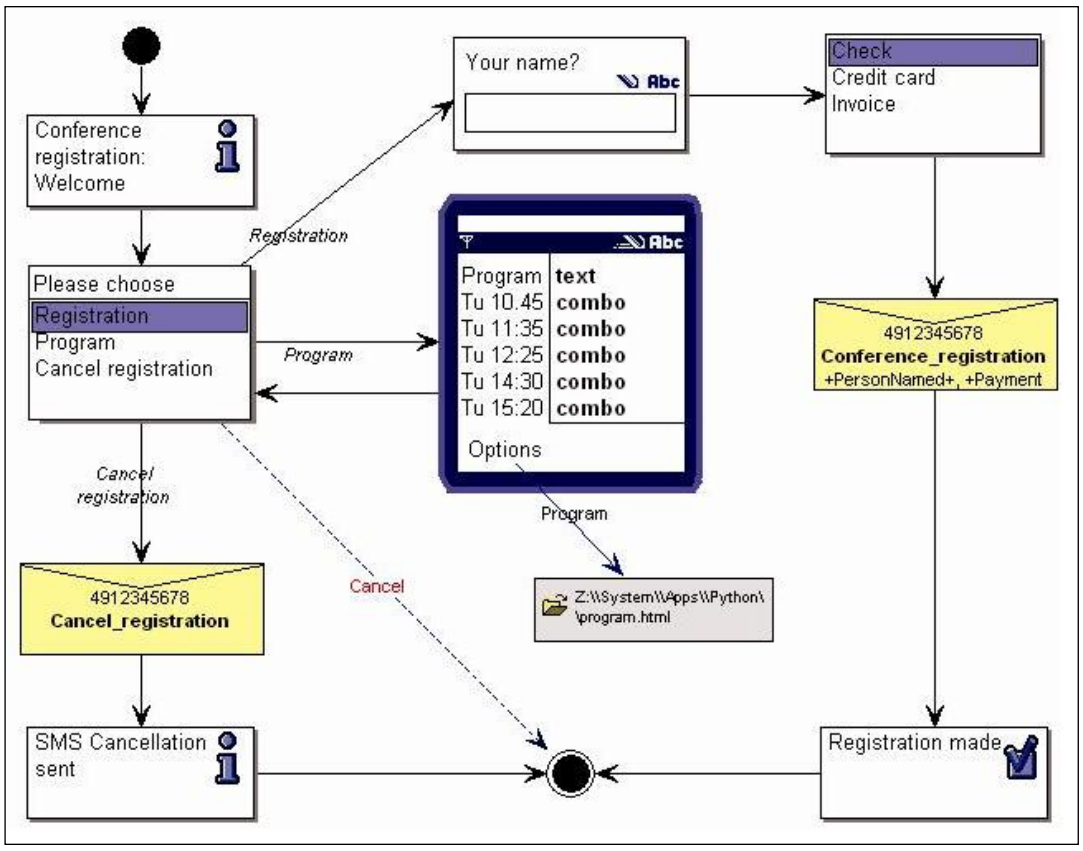


Figure 10 – Model created using a DSL for smart phones [Tolvanen, 2005]

As opposed to UML, domain-specific modeling (DSM) and visual DSLs are used by software factories to provide real abstraction to the development process, not only “visual syntax-sugars”. These languages follow domain abstractions and semantics, allowing developers to perceive themselves as working directly with domain concepts [Kelly & Tolvanen,

2008]. In many cases, the complete final product code can be automatically generated from these high-level specifications with domain-specific code generators.

Adopting a DSL approach to Software Engineering (i. e., adopting a language oriented programming[12] approach) involves both risks and opportunities. Besides the aforementioned abstraction and variability expressiveness, DSLs improve reliability, maintainability [Kieburtz et al., 2006] [van Deursen & Klint, 1998], portability [Herndon & Berzins, 1988], testability [Sirer & Bershad, 1999] and allow validation and optimization at the domain level [Menon & Pingali, 1999] [Bruce, 1997] [Basu et al., 1997]. On the other hand, some disadvantages and challenges of using DSLs can be also pointed out, such as [Krueger, 1992] [van Deursen et al., 2000] [Spinellis, 2001] [Gray et al., 2008]:

- The high costs of designing, implementing and maintaining a DSL. For instance, extending a DSL for unanticipated changes can be substantial.
- The high costs of education for DSL users. A new language implies at reluctant customers. Although they are generally able to understand the return of investment after the using the language for a while, lowering the entry barrier is still a challenge.
- The limited availability of DSLs.
- The difficulty of understanding the domain and defining the DSL scope (DSLs targeted at too broad domains are ineffective).
- The difficulty of balancing between domain-specificity and general-purpose programming language constructs (conditional branches, loops, etc.) which might be expected by the DSL users but may not make sense to the domain.
- The potential loss of efficiency, in some situations and especially without tool support, when compared with hand-coded software.
- Cross-DSL integration.
- Difficulty to interoperate with mainstream languages.
- The common pitfall of creating DSLs based on solution domain (code), instead of the problem domain.
- Immaturity of tools for iterative development and use of DSLs.
- Planning and assessing the return on investment (cost-benefit analysis).
- Lack of systematic approach to replicate processes related to DSLs design and implementation.

[12] Language Oriented Programming is the general style of development which operates about the idea of building software around a set of domain-specific languages. [Fowler, 2005]

- Unclear guidance on how to fit the use of a DSL into standard software development processes.
- Backward compatibility: it is hard to evolve the language when models (specs) created with it are already in place.

Creating DSLs comes at a cost to design the language, build the translator, and consider tools to support programming. This is especially true for visual DSLs, which demand more tooling such as graphical editors. This is where language workbenches come into play [Fowler, 2005]. They contrast the early days of domain-specific modeling, when no tools were available to create domain-specific languages and support modeling with them in a cost-effective manner [Tolvanen, 2005], requiring many man-years of development and left as an option only for large organizations that could commit to such an undertaking. Language workbenches, on the other hand, use IDE tooling to make language oriented programming a viable approach, making it easy to build tools that match the best of modern IDEs. They provide a toolset that supports the creation of the DSLs' abstract and concrete syntax, generators, semantic validators and other assets. Examples of language workbenches include JetBrains' Meta Programming System[13], Intentional Software[14] and Microsoft Visual Studio Team System (VSTS) [Guckenheimer & Perez, 2006].

3.5 Contextual and Automated Guidance

In general, software process guidance is about helping developers to solve a current design or coding problem. It can take many different forms, such as help pages, developer journal articles, code samples and patterns [Gamma et al., 2005].

When dealing with *prescriptive guidance*, developers have to absorb a large amount of documentation in advance, but are left on their own on how to apply it in practice. In contrast, software factories introduce the concept of *contextual and automated guidance*, which suggests and/or automates process activities that developers are supposed to perform in a particular context, such as changing the project structure to add unit tests, and displays on-demand, context-specific help to developers [Lenz & Wienands, 2006]. This guidance can be created by domain experts, systems engineers and software architects to convey the best practices and semantic constraints to developers.

Such guidance can be fully or partially automated. It collects input from developers by means of wizards, optionally applies transformations and validators to the input, mashes-up input from different sources (such as a web service) and finally launches automated and pa-

[13] http://www.jetbrains.com/mps

[14] http://intentsoft.com

rameterized tasks by feeding them with the transformed input. The result can be made available to the developer, cause a change in the development environment or be chained into other guidance tasks. Therefore, automated and contextual guidance heavily rely on integration with the development environment, commonly exploring its extensibility APIs.

The Guidance Automation Toolkit (GAT) and the Guidance Automation Extensions (GAX)[15] support the creation and execution, respectively, of contextual and automated guidance. Using a XML file, the domain architect responsible for packaging and automating the guidance can define recipes to automate activities that developers would usually perform manually, wizards, (code) generators and type converters to manipulate the user input. Guidance automation libraries implemented by the architects can also be linked from the XML file and launched as part of a recipe.

The Microsoft Blueprints[16] was another initiative which also supports contextual and automated guidance. The target development environment (Visual Studio) displays the available guidance by means of tasks in a specific tool window (Figure 11). The guidance tasks can launch commands packaged by software architects to execute a task, such as transforming the development environment by adding to the current project a brand new customized class file. Domain architects can also define the availability of guidance tasks by modeling an activity diagram using the Windows Workflow Foundation [Shukla & Schmit, 2006].

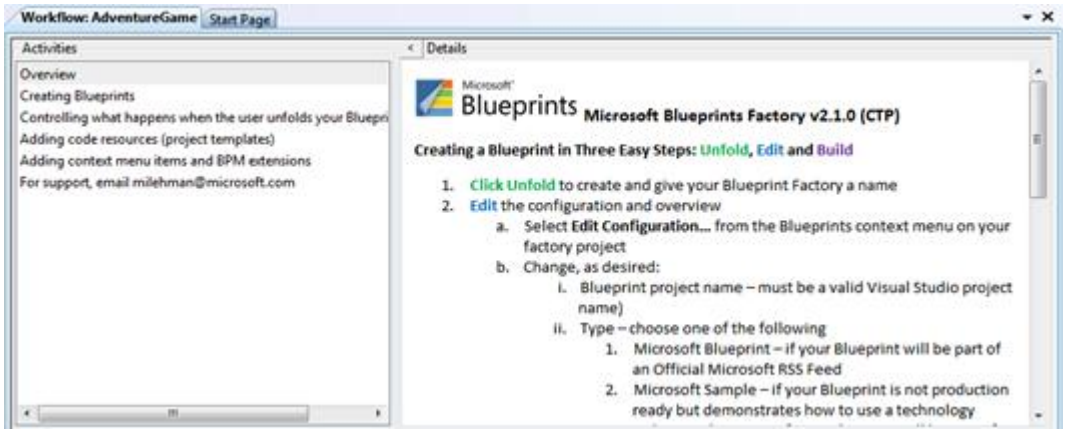


Figure 11 – Microsoft Blueprints guidance workflow tool window

Finally, semantic validators embedded into visual DSL designers can provide contextualized guidance by displaying high-level domain-specific errors and warnings related to the diagram being modeled. For example, a visual DSL for modeling the game flow could raise an alert pointing that no “game over” criteria was specified. This way, developers can be guided on how to properly satisfy the domain constrains expressed by the semantics of a

[15] http://msdn.microsoft.com/en-us/library/ff687174

[16] http://msdn.microsoft.com/en-us/architecture/blueprints.aspx

given visual DSL. That is necessary, for instance, to ensure the code generators which receive the modeled diagrams as input can properly run.

3.6 Software Factories

Some different definitions for “software factories” have been used and realized along the years, and they have even been compared [Aaen et al., 1997]. One of the most recent incarnations of the concept, by Greenfield et al. [2004], builds on several already established software engineering concepts, many of them presented in the previous sections, such as Software Product Lines, reusable software assets (such as application blocks and frameworks), Model-Driven Development, automated context-based guidance, patterns, languages and tools.

Software factories rely on integration with development environments and a more graphical approach based on MDD and DSLs that, unlike Computer-Aided Software Engineering (CASE) tools, is seriously interested in semantics and control over code generation [Fowler, 2005]. In summary, a software factory can be defined as a software product line that configures extensible development environments and processes with packaged content and guidance, carefully designed for building and maintaining variants of an archetypal product by adapting, assembling and configuring framework-based components [Greenfield et al., 2004]. It also reduces the time and effort required to follow that guidance by providing a partial solution, such as templates to complete or components to assemble, and by automating tasks performed to produce the finished product. Equipped with a software factory, a development team can rapidly punch out a variety of domain-specific applications, each containing unique features based on the unique requirements of specific customers.

A software factory is similar to a template loaded into a Microsoft Office application, like Word or Excel, to customize it for a specific task such as writing a resume, or computing mortgage payments. However, instead of customizing an Office application with a document template, a factory customizes an integrated development environment (IDE) like Visual Studio or Eclipse with a template containing class libraries, projects, help files, wizards, web pages, patterns and visual designers. Instead of information workers, its users are developers, project managers, architects, business analysts, testers and systems administrators. Instead of word processing or numerical modeling, it supports software life-cycle activities, like specifying, building, testing, deploying, operating and maintaining systems.

The software factory elements as structured by [Lenz & Wienands, 2006] are presented in Figure 12. It contemplates both the factory designer perspective as well as the factory consumer (product developer) perspective. While the factory designer is described as someone who selects and packages a collection of core assets along with a development

process for developing instances of the software product line, the product developer instantiates and uses the factory to create members of the product line more effectively and predictably. Although the general schematic is similar to a software product line, the specifics of a software factory relies more on tool support, development environment integration, visual domain-specific languages empowered by code generators and contextualized automated guidance.

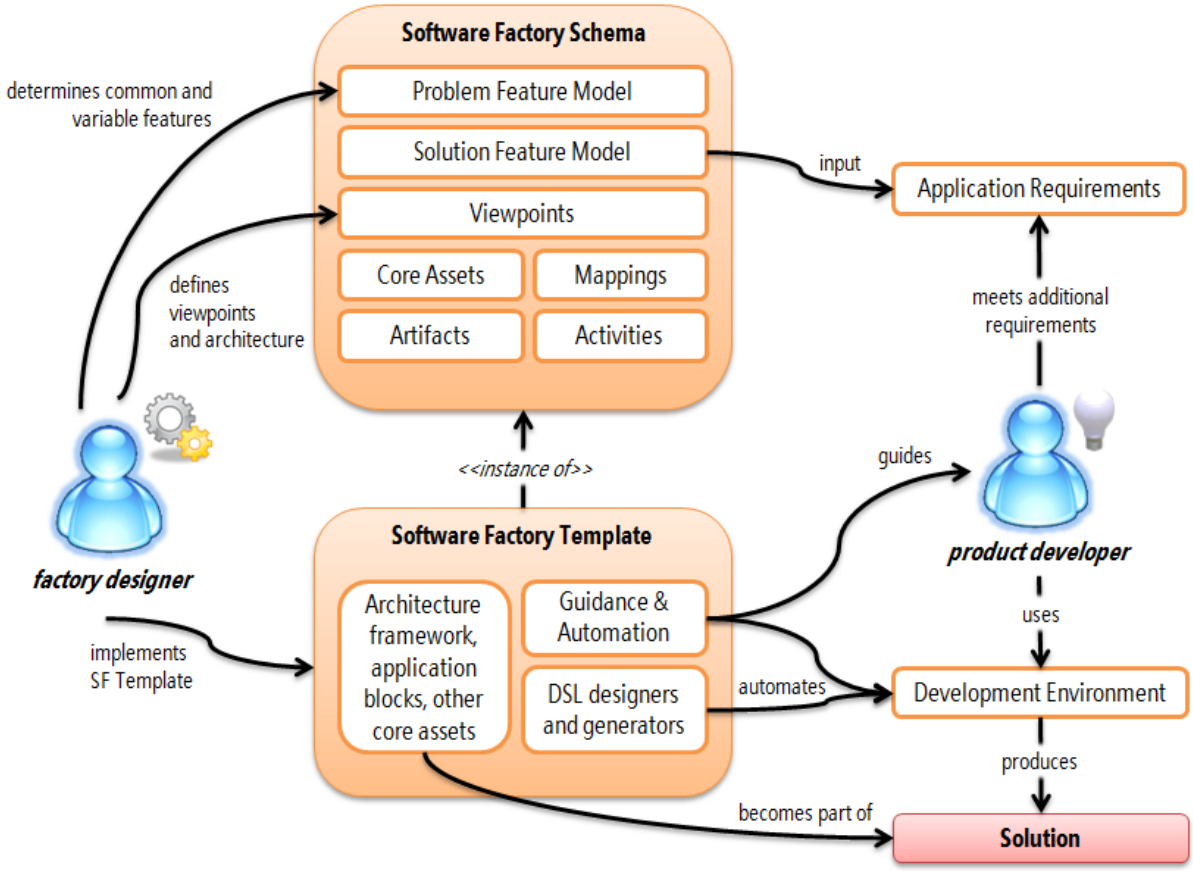


Figure 12 – Software Factory Overview [Lenz & Wienands, 2006]

While the software factories approach brings a number of benefits, they come at a price. Potential issues and challenges have to be considered when employing software factories, in special for digital games development. For instance, a “more predictable and uniform user experience” may be a disadvantage in digital games. From one hand, uniform behavior is still desired for games in some scenarios, such as the physics of platform and racing games, or the menu options of games deployed or hosted together such as in the Xbox Live Arcade network. However, since surprising players with new features is a desired goal for digital games, uniform behavior is not desired in many other game elements, such as the game flow or narrative, for instance. It is up to the game SPL designer, therefore, to identify where predictable and uniform experience is desired and where to provide variability and ex-

tension points to game developers. Others challenges related to software factories include an overwhelming theory that spans many software development disciplines, considerable resource demands and complexity in integrating existing assets yet remaining extensible.

3.7 The Applicability of Software Industrialization to Digital Games

A question that naturally arises in the context of this research is how applicable software industrialization concepts are to the digital games development domain. Since digital games are targeted at players, not at enterprises (which mainly exploit economies of scope), the tendencies are that not all of the claimed industrialization benefits can be applied to games. For example, some concerns exploited by software factories, such as business requirements, are more critical to enterprise software development than to digital games development.

In order to provide a clearer picture of such a discussion, this research analyzed the main challenges targeted by software factories as well as the expected major implications of software factories, according to Greenfield et al. [2004], and how relevant they are to digital games development. The results are displayed in Table 1 (digital games vs. challenges targeted by software factories) and Table 2 (digital games vs. major implications of software factories). For each challenge and expectation, our analysis provides a digital games relevance score of **High**, **Medium** or **Low**. Comments are also provided.

The results of this informal evaluation show that 19% of the challenges have **Low** applicability to digital games, 33% have **Medium** applicability and the remaining 48% have **High** applicability. Likewise, 14% of the major software factories implications were evaluated as having **Low** applicability to digital games, 29% as having **Medium** applicability and the remaining 57% as having **High** applicability.

The conclusions are that although digital games development is clearly a domain in which software factories are less effective than domains which deeply exploit economies of scope and customization, the number of addressed challenges and positive implications for games seems to make it worth to employ the approach.

Table 1 – Software development challenges and relevance to digital games

Trigger	Challenge	Relevance to Game Development
New business requirements	Support reengineered business processes and an increasing focus on process-oriented applications	**Low** (games are generally not focused on business processes)
	Expose existing systems to massive user loads created by web-based front ends	**High** (online games are considerably common)
	Design protocols (valid message sequences) and enforceable service level agreements to support processes than can span multiple enterprises	**Medium** (communication protocols are needed, but services that span multiple enterprises are rare in games)
	Determine strategies for versioned data and snapshots, such as price lists, that are widely distributed yet have limited lifetimes	**High** (in games, such data are represented by items such as demos, maps, scenarios, high-score tables, saved games and so on)
	Cope with the complexity created by transformed business models from business leaders such as Wal-Mart, who insist on deep integration with their partners	**High** (impacts revenue sharing; genres, gameplay experiences and interaction paradigms keep evolving, as well as engines and tools)
	Integrate heterogeneous application stovepipes and avoiding lossy transformations between them	**Medium** (integration is done for components like servers and game engines, but it is not so heterogeneous)
	Avoid reintroducing the batch era problem of multiple file layouts and lack of data format consistency in the rush to describe XML schemas for every software service	**High** (standards are desirable for saving games, defining scenarios, distributed communication and so on)
	Determine strategies for wrapping older applications executing on heterogeneous platforms	**Low** (legacy is not a recurring problem in the game domain)
	Customize packaged software to satisfy proprietary requirements	**High** (customization can arise in the form of map/level editors, skins and MODs[17])
	Address the special need of data warehouse and business intelligence	**Medium** (while games generally do not involve data warehousing, business intelligence is crucial to some domains such as the mobile)
	Demonstrate return on investment in custom software in the face of rising software development costs	**High** (game engines are a formidable example of a high effort which can provide return of investment by means of customization)

[17] MOD is short for Modification. It is a package that can be applied to a game to modify its appearance (graphics, sounds, texts, etc.) or even its behavior.

Trigger	Challenge	Relevance to Game Development
Increasing focus on security	Protect corporate data from hackers and viruses, while giving customers and partners direct access to the same resources	**High** (games should not attempt to access unauthorized data or execute unauthorized operations; cheaters and hackers are a constant annoyance in online versions)
	Mitigate the increasing risk of legal liability from the improper use of corporate data	**Medium** (although game data are generally not corporate and/or confident data, part of user profiles must be private)
Increasing deployment complexity	Satisfy operational requirements, such as security, reliability, performance and supportability, in rapidly changing applications	**High** (some of such concerns are critical to digital games)
	Integrate new and existing systems using a wide range of architectures and implementation technologies	**High** (integration with game engines, tools and even servers is heavily demanded)
	Understand the effects of partitioning and distribution on aggregate qualities of service	**Medium** (quality of service is impacted by distribution in games by only a few variables, such as performance)
	Design multitiered applications that deploy correctly to server farms on segmented networks partitioned by firewalls, with each server running a mixture of widely varying host software configurations	**Medium** (may appear only in some scenarios of online games, such as Massively Multiplayer Online Games)
Decentralized software development	Support applications developed by end users (for example, spreadsheets to 4GLs) while enforcing corporate policy and maintaining the integrity of corporate data	**High** (this concern rises in game development by means of map/level editors, skins, MODs or event behavior definition by players through script languages)
	Integrate personal productivity applications such as word processors and spreadsheets, with back end systems	**Low** (such applications are generally not integrated with digital games)
	Work with applications or components that are increasingly outsourced to development centers in remote locations forcing a discipline on requirements designs and acceptance tests that was often neglected in the past	**Medium** (outsourcing may appear in the form of graphics modeling, artificial intelligence programming, level design, and other tasks, but that is not common unless for AAA games with a high budget)
	Make departmental application integrate effectively and scale to satisfy enterprise requirements	**Low** (enterprise scalability makes not much sense to digital games)

Table 2 – Software development implications and relevance to digital games

Implication	Relevance to Game Development
Development by assembly: only a small part of each application will be developed from scratch	**High** (many aspects of game development, such as map generation and entity rendering, for example, are positively impacted by this implication)
Software supply chains: each participant consumes logical and/or physical products from one or more upstream suppliers, adds value, usually by incorporating them into more complex products, and supplies the results to downstream consumers	**Low** (despite supply chains can be applied to some scenarios, such as graphics modeling, they more applicable to other domains than game development)
Standardization of specification formats, packaging formats, architectures and patterns	**High** (games development deeply welcomes such standardization, however this challenge may be hard to overcome due to industrial secrecy)
Relationship management: managing costumer and relationship will become more important	**Medium** (a "digital game requirement" is not so well defined as in other domains, while beta-testers and publishers still have to be dealt with)
Domain specific assets: product line developers will build assets used by product developers	**High** (game engines, language-based tools and other frameworks/tools are genuine examples of product line assets used by game developers)
Organizational changes: much about development and development organizations will change	**High** (such an implication is relevant to virtually all software development domains)
Mass customization of software: software may eventually be mass customized like PCs ordered on the web today	**Medium** (letting the player to completely configure and customize a game genre in order to create its own instance, and then order it, is still a novel scenario)

3.8 Chapter Summary

This chapter presented the Software Engineering foundations this research is based on, with focus on SPLs, software factories and one of their most valuable underlying assets: domain-specific languages. Previous literature on reuse, Domain Engineering and visual modeling was also presented. Finally, the applicability of software factories to digital games development, based on the implications and challenges software factories are supposed to bring and solve, was also discussed.

By outlining the state-of-the-art of digital games development and software reuse targeted at application families, Chapters 2 and 3 provided the necessary background that enables the description of the proposed Domain-Specific Game Development approach, aimed at streamlining the development of games belonging to a same family. The next chapter focuses on introducing and detailing the elements of the approach.

4. DOMAIN-SPECIFIC GAME DEVELOPMENT

Many cases of successful Software Product Lines in practice can be found for different domains [SPL Hall of Fame, 2012], such as consumer electronics, printing machines and avionics. However, this is not true for domains related to digital games. We believe that this happens in part due to the fact that current application-family development approaches do not address the specific constraints and peculiarities related to digital games development. Examples of such peculiarities are:

- The concept of “genres” is extremely popular in digital games. It is commonly used as an attempt to define taxonomies into whose branches games can be classified. However, genres are blurry and imprecise: there is no agreement on a universal set of genres neither on the individual meanings of specific genres [Oxland, 2004]. Interpreting genres as sub-domains or solely taking them as is for scoping a game SPL can imply in abnormal outcomes for the product line.
- The development of a digital game is not a direct outcome of user requirements or business needs, which may not even exist for a given project in the domain. Games are typically *not* focused on solving end users’ problems, but to entertain and “seduce” them. On the other hand, emotion-based requirements such as immersion, surprise, delight and nostalgia are very present in digital games. This way, traditional Requirements Engineering cannot be applied as is to game development. For example, the well-known concept of “use cases”, with well-defined roles, flows and input/output artifacts does not make sense to game development processes. In practice, the high interactivity, randomness and unpredictability of digital games would imply in thousands of use cases, which are unviable to manage. Game Design documents, experimentation processes and focus on functionalities, many times volatile, are more realistic in this area. Atypically, game development emphasizes non-functional requirements [Callele et al., 2005] and lowers the stability of functional ones [Maier & Volk, 2008].
- Software is only one of the elements that constitute a digital game. Besides deeply relying on resources of multiple kinds (graphical, audio, etc.), games have peculiar challenges from the design standpoint, such as level-design, replay-value and immersive (plus interactive) storytelling. Synchronizing technical and artistic branches is an additional challenge that requires the coordination of multidisciplinary teams. Moreover, since digital games are many

times concerned with mimetically representing the reality, instead of abstracting it, analysis and design activities are not trivial.

- High-performance constraints together with very tight schedules forced digital games development to historically trade more refined Software Engineering techniques for a result-oriented but less organized development process, as well as reusability for in-house development, in a methodology that became known as “pedal to the metal” [Rollings & Morris, 2000]. Extra optimization efforts, for instance, are vital to enable digital games to render their virtual environments in real time.
- In game development, general-purpose programming languages such as C++ and C# have to be used in combination with script languages, frameworks and tools (DirectX, OpenGL, etc.) for specific tasks such as low-level graphics processing [Moreno-Ger et al., 2008]. Managing multiple assets like those increases the game development complexity.
- The user interaction is unique in digital games, especially when compared to other types of software generally based on mouse, keyboard and limited graphical interface standards (e.g., windows-based GUIs). In digital games, adherence to standards is many times trumped by the desire to provide innovative experiences. However, such experiences should still ensure usability, making it possible for players, with no previous training, to perceive the game controls and interaction through intuition. User interaction challenges range from one-button gameplay in casual domains to complex controller systems in simulators, or even controller-free experiences as in Kinect[18]. Game feedback to players many times relies on bleeding-edge audio and graphical experiences, which are typically not widespread to software in general. Finally, real-time multi-user interaction (local or online) is also peculiar in digital games.
- The usage of game engines in the design and implementation phases is heavily disseminated in game development. Virtually all successful titles in the industry rely on game engines in order to satisfy time-to-market constraints. Even integrated graphical environments such as Unity make use of engines in the back-end. Engines handle many low-level tasks and should be configured by the development team, sometimes using a script language.
- Digital games have unique deployment challenges including the diversity of target platforms (PC client, web, mobile, multiple consoles, etc.), media (car-

[18] http://www.xbox.com/kinect

tridges, CDs/DVDs, network, etc.) and distribution channels (publishers, online platforms for community content hosting, etc.).

- An impressive diversity of game instances, even from decades ago, is still in use today in the digital games domain. In contrary to the majority of other software domains, nostalgia causes “retro” instances to be kept alive for generations. Many samples are also available due to other reasons peculiar to game development, such as the diversity of platforms, prototyping culture and user-generated content. This translates into rich, valuable and available input for designing future software in the domain. On the other hand, it also implies in more samples that should be analyzed or filtered out by Domain Analysis tasks.
- Studies on the applicability of Software Product Lines in the game development domain are still incipient. Industrial secrecy to support competitive advantage is very high in the domain, since such projects involve great investments [Rocha, 2003]. For example, it is difficult to find comprehensive studies about the applicability of design patterns in game engines [Madeira, 2003].
- Specific game sub-domains also have additional peculiarities. For example, the big diversity of target devices is an issue for mobile domains [Nascimento, 2008], while reaching the perfect *easy-to-play/hard-to-master* balance is imperative to casual games more than any other domain [Araujo, 2009].

Taking the aforementioned peculiarities into account, as well as the motivations, background and state-of-the-art from previous chapters, this chapter proposes and details a Domain-Specific Game Development approach. Section 4.1 provides an overview of the approach. Section 4.2 introduces the concept of Game Domain Envisioning. Section 4.3 presents Game Domain Analysis. Section 4.4 discusses how to bridge Game Domain Analysis to application core assets, while Section 4.5 discusses how to bridge it to development core assets. Section 4.6 comments on cross-SPL game assets and Section 4.7, on Application Engineering. Finally, Section 4.8 concludes about this chapter.

4.1 Approach Overview

Taking as evidence the fact that some SPL approaches are too generic and lack implementation details that would enable them to be more effectively employed in practice [Nascimento, 2008], this research endorses that SPL approaches should be specialized and streamlined when the target product lines are scoped to a narrower macro-domain, such as digital games development. In the same way that the effectiveness of a SPL depends on how it is focused on a specific domain, this research considers that SPL approaches can be more ef-

fective if bound by macro-domains, instead of being designed to target software development in general.

By instantiating such a discussion to the digital games development macro-domain, this research describes a practical SPL-based approach for bridging the Domain Analysis of a game family to the implementation of core domain assets, area not comprehensively approached by other SPL or Domain Engineering processes [Nascimento, 2008]. The approach has a special focus on domain-specific languages, which are key to realize the software factories vision but are currently underexplored in the context of game development. An outline of the proposal is presented in Figure 13.

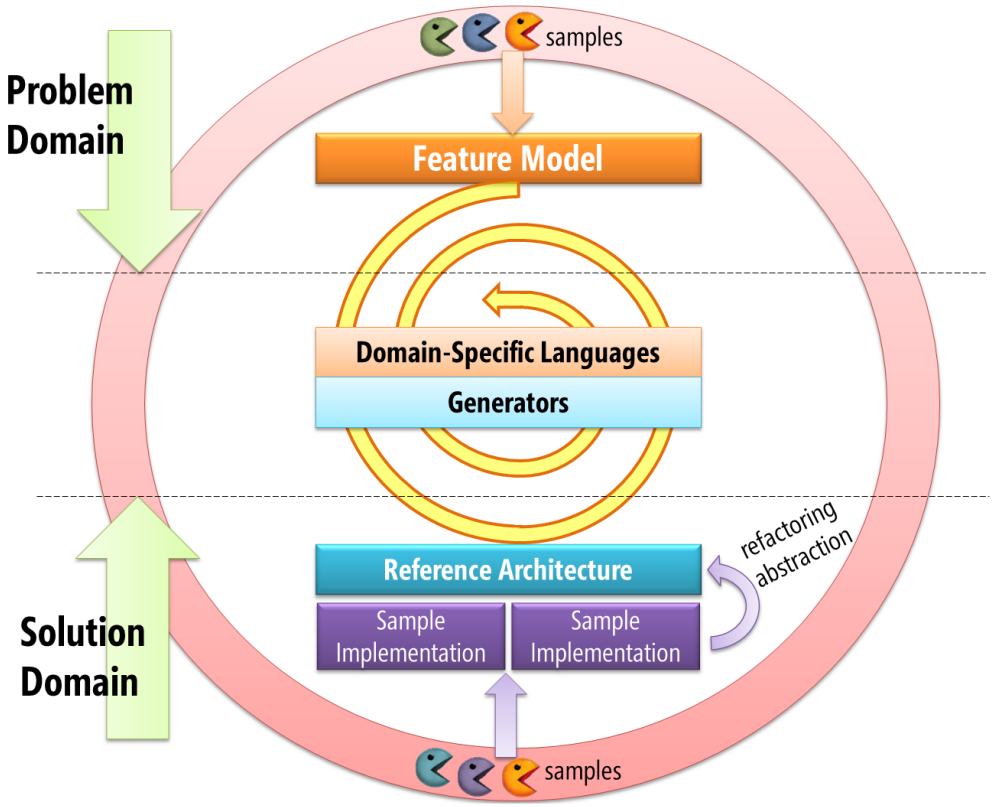


Figure 13 – A high-level overview of the Domain-Specific Game Development approach

The main goal of the approach is to enable the development of core domain assets for a game SPL, such as domain-specific languages, generators and reference architectures, starting from the Game Domain Envisioning and Analysis, which are then bridged to Domain Design and Implementation. Most notably, the proposal builds atop strict top-down and bottom-up Domain Engineering approaches, combining them toward the development of core assets in a spiral and iterative process denominated “edge-center”. Its goal is to work incrementally from each side (problem and solution domain) to avoid the risk of a big upfront investment in any of them.

The evidences that justify such an approach come from previous experiences in SPL projects for digital games and the nature of digital games development itself. From one hand, a strict top-down approach generates domain-specific languages majorly taking Domain Analysis (typically structured by means of feature models) as input. It considers the domain as a collection of interrelated concepts, represented in the DSLs' domain model, and has a number of potential advantages [Cook et al., 2007]: it gets much more quickly and directly to a substantial DSL, it tends to produce a more self-consistent and complete result and it tends to ensure that variations are expressed in terms of requirements rather than implementation, so that incidental variations in implementation are factored out. Kelly & Tolvanen [2008] mention that although it may be tempting to use concepts that originate from the source code as a starting point for language definition, higher abstraction and productivity can be achieved if the modeling concepts come from non-implementation concepts.

However, in such top-down approaches, many abstractions and details that could be identified from experience with actual product code may not be caught by the DSL syntax, semantics and generators, leading to missed opportunities to empower the modeling experience. For example, the top-down DSL implementation approach used in the SharpLudus Adventure game SPL [Furtado, 2006], which was used as a spike solution for this research, originally resulted in a very rigid DSL support for implementing game event triggers and reactions, which developers defined by simply enumerating them in a list. If product code was taken into account, the different scenarios and contexts where triggers and reactions can be combined would have been made clearer, leading to a richer and more flexible DSL support to that. In fact, after such an experience we concluded that whenever a new feature is to be supported by the DSLs of a game SPL, it is imperative that such a feature is implemented first in source code as an extension of the SPL's built-in feature set, or identified and extracted from an already existing implementation. This enables SPL designers to understand what parts of the domain framework (game engine and other components) is required by the feature and hence what code needs to be generated. Such information is then used as input to create or update the DSLs' syntax and semantics, and therefore is valuable input for designing or improving a game SPL.

Likewise, since reference implementations and architectures are trumped by feature models in top-down approaches, "biased" DSL generators can be inadvertently created, often leading to deficient implementations. In other words, in strict top-down approaches, generated code merely becomes a direct output of the DSL constructs, many times just a refined serialization, rather than a real mapping from higher to lower levels of abstraction which take architectural, performance and other low-level concerns into account. Once again, event triggers and reactions in the SharpLudus Adventure spike solution can be mentioned as an ex-

ample: since no reference implementation was taken into account in the process, the generated code created inefficient lists of event trigger objects whose launch conditions were checked every game cycle and were quite difficult to manage. This was an alarming issue for the generated games, since performance is commonly one of the most important concerns in game development, perhaps more than in other domains.

On the other hand, bottom-up approaches begin with specific application code and gradually parameterize it [Cook et al., 2007]. Existing code is turned into a set of templates, whose pieces are gradually replaced by template expressions. The DSLs develop alongside, as the means to express these statements' parameters. Nonetheless, implementing a DSL solely based on such template expressions, abstractions and refactorings identified in the product code can compromise the experience of game designers and developers, since a DSL is not supposed to deal with low-level implementation details [Gray et al., 2008]. In such strict bottom-up approaches, the DSLs' syntax and semantics can become only an alternative (many times graphical) representation of the code, leading to an awkward user experience that does not provide any abstraction at all.

Besides taking lessons learned from strict top-down and bottom-up approaches, the proposed "edge-center" approach also exploits some of the peculiarities of the digital games development macro-domain. For instance, game engines are considered as a vital piece in defining the reference architecture. Likewise, the approach helps game developers and designers to explore, in its both edges (problem and solution domain), the numerous game samples widely available thanks to the diversity of platforms, prototyping culture, abundance of user-generated content or simply nostalgia. More details on how the process leverages such peculiarities are described in this chapter.

Although this research provides a starting foundation for conceiving a comprehensive Game Domain Engineering process that contemplates the details of all Domain Engineering phases (Domain Analysis, Design and Implementation), it is out of our scope to do so. Hence, our approach should not be taken as a comprehensive Domain Engineering process. Our focus is on activities that impact Model-Driven Development in the context of digital games development, toward the creation of application core assets in a game SPL, such as DSLs and generators. This is illustrated by Figure 14. The darker areas illustrate the amount of activities covered in each phase, while the remaining ones are out of scope. It is worth noticing that although the phases below are presented sequentially, they are carried out in multiple iterations responsible for refining domain artifacts such as models, architectures and implementations [Lucrédio, 2009]. Likewise, the approach is focused on the *engineering* aspect of game development, majorly on game developers and designers. It has boundaries with but does not address other areas in game development such as the artistic ones.

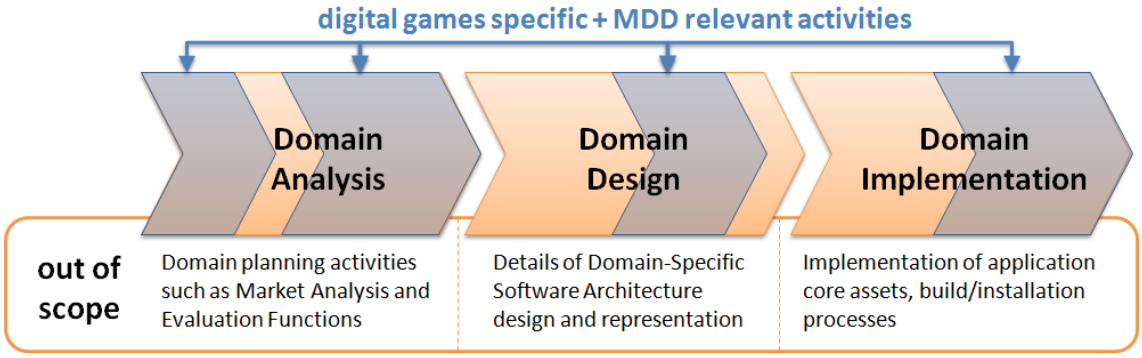


Figure 14 – Domain Engineering coverage from the proposed approach

Table 3 presents a summary of the proposed approach, presenting its activities and, for each of them, inputs, outputs and the role(s) in focus. Although the activities are presented sequentially, a combination of them should be executed in parallel. Moreover, the activities are revisited during each iteration. The next sections provide more detail on that.

4.2 Envisioning the Game Domain

Before analyzing domain products, we suggest that some envisioning work is carried out as the first task of planning a digital game SPL, in order to guide and better contextualize subsequent Domain Engineering activities. We define **Domain Envisioning** as an activity focused on establishing a high-level overview and common understanding of the domain to be approached, creating a vision to communicate the initial expectations for domain products. Such a vision is then refined by the activities in Domain Analysis, resulting in the game SPL scope.

The need for domain envisioning is quite evident in digital games, where the concept of “game genres” (blurry, ambiguous and scattered across multiple dimensions [Lindley, 2003]) can lead to a troublesome Domain Analysis unless a proper vision is established upfront. In addition, domain envisioning provides a baseline upon which domain scoping activities can start, in special by defining a **set of expectations** that can be used to select products and features to analyze. This is only possible because the product line instances in this context belong to a macro-domain (digital games, instead of software in general), for which categories of expectations can be defined upfront. Section 4.2.2.1 (*Setting Expectations for Core Game Dimensions*) provides more details. Envisioning also helps solving a chicken-and-egg problem, since understanding the domain requires analyzing domain products, but selecting domain products to analyze requires understanding the domain.

Reaching a complete domain specification is not a goal of Domain Envisioning, especially because the domain might be new or only partially understood. In fact, this is an iterative process: as game samples are analyzed and catalogued (Section 4.3), the domain be-

comes better understood, its scope can be refined and, as a consequence, game SPL assets can finally emerge to capture variability and commonality in the domain.

4.2.1 The Unreliability of Game Genres

As previously mentioned, the software reuse process is more effective when systematically planned and managed in the context of a specific domain, where application families share some functionality. America et al. [2000] state that a large population in a SPL only makes sense if the products in the family have enough in common to make it profitable and manageable by the organization.

On the other hand, the great diversity of games created so far has turned the digital games universe into an extensive, too broad domain. Creating a SPL targeted at digital games in general, ranging from 2D platform games to 3D flight simulators, constitutes a too wide-ranging and ineffective endeavor. In such a scenario, the SPL process and its tools would not be able to fully exploit SPL benefits such as component reuse and assemblage. In other words, a game SPL should focus on a narrower game domain, better scoping the possible games that the SPL users (game designers and developers) will create and generate.

Table 3 – Summary of the Domain-Specific Game Development activities

Phase	Activity	Inputs	Outputs	Role in Focus
Game Domain Analysis	Envision the game domain	Game domain, business goals and constraints, stakeholder information, market analysis, game samples, expert knowledge	Game domain vision (vision statement, expectations for core game dimensions, negative scope, target platforms), go/no-go decision	Domain analyst, domain expert, market specialist
	Build the game domain vocabulary	Game samples and other elicitation sources, expert knowledge	Game domain vocabulary	Domain expert
	Define and refine game domain features	Game domain vocabulary, game samples	Domain features	Domain analyst
	Select domain samples	Game samples, domain vision, expert knowledge	Refined list of game samples	Domain expert

Phase	Activity	Inputs	Outputs	Role in Focus
Game Domain Analysis (cont.)	Analyze game samples and model game domain	Refined list of game samples	Evaluated game samples, feature model, domain understanding churn	Domain analyst
	Partition the game domain into sub-domains	Domain, sub-domains, evaluated game samples, feature model	Sub-domains	Domain analyst
	Revisit the game SPL domain scope	Business goals and constraints, game domain vision, evaluated game samples, feature model, expert knowledge	Updated game SPL scope	Domain analyst
	Test sample analysis against stop criteria	Feature model, domain understanding churn	Decision on whether to pursue additional sample analysis	Domain analyst
	Validate the game domain	Feature model, game domain vocabulary	Refined features, validated feature model	Domain analyst, Domain expert
	Assess game domain automa-tion potential	Sub-domains, domain framework, reference implementations	Prioritized sub-domains	Domain analyst, domain architect
Design & Implement. of Application Core Assets	Create Domain-Specific Game Architecture	Simple implementa-tions, game engines, tools	Domain-Specific Game Architecture, new or updates game engines	Domain architect
	Promote game engines to domain frameworks	Domain-Specific Game Architecture, prioritized sub-domains	Domain framework, reference implementations	Domain architect, domain implementer
	Create reusable game components	Domain framework, prioritized sub-domains, reference implementations	Updated domain framework	Domain architect, domain implementer

Phase	Activity	Inputs	Outputs	Role in Focus
Design & Implement. of Development Core Assets	Characterize sub-domain variability	Prioritized sub-domains, feature model	Characterized sub-domains	Domain expert, domain implementer
	Decide upon MDD development	Characterized sub-domains, existing tools and languages	Selected sub-domains	Domain expert, domain implementer
	Define DSL and supporting assets	Selected sub-domains	DSLs	Language expert
	Develop trans-formations and refine DSLs	Selected sub-domains, DSLs, existing tools and languages	Transformations, refined DSLs	Transformation expert
	Design and implement IDE integration	Selected sub-domains, DSLs, transformations, existing tools and languages	IDE Integration	Integration expert
Application Engineering	(All related to creating a single game instance)	Game SPL, project goals, requirements and constraints, experimentation results	Game	Project manager, architect, developers, tester, configuration manager, producer, game designer, sound designer, art designer, etc.

One of the most often used attempts to classify digital games is the concept of *game genres*, which defines a game *taxonomy* [Crawford, 1984]. Some of the most popular game genres [Wolf, 2002] [Sawyer, 1995] [Crawford, 1984] include adventure games, platform games, fighting games and strategy games.

Nevertheless, defining genres can be a quite difficult task as many people have different opinions on the meaning of a genre or various ways of stereotyping them [Oxland, 2004]. Likewise, it is not rare for a game to fall into more than one category. Some authors even argue that describing different types of games requires different dimensions of distinc-

tions (narratology, ludology, simulation, gambling, etc.), i.e., orthogonal taxonomies which allow design concerns to be separated [Lindley, 2003].

Classifying games into genres is a difficult task not only because a game can be hybrid, but also due to the fact that some genres are “horizontal”, such as casual games, educative games, serious games, adult games and advergames. The high evolution speed experienced by game genres is also an issue. Crawford [1984] pointed out that, due to the dynamic nature of game taxonomies, they can be expected to become obsolete or inadequate in a short time.

Furthermore, there is a lack of consensus at the level of the classification schemas, some being more popular than others. For instance, some schemas are largely semiotic, while others rely more strongly on configurative patterns of interface and mechanics. In short, due to a general lack of commonly agreed-upon genres or criteria for the definition of genres, the classification of games under schemas are not always consistent or systematic and sometimes outright arbitrary between sources.

One interesting example of game genre disagreement is the *“Shoot’em up”* genre, in which the player controls a single character, often a spacecraft, shooting large numbers of enemies while dodging their attacks. Such a genre encompasses various types or subgenres and critics differ on exactly what design elements constitute a *shoot 'em up* game. Some restrict the definition to games featuring spacecrafts and certain types of character movement; others allow a broader definition including characters on foot and a variety of perspectives. The game *Sharky’s Air Legends* (Figure 15), on the other hand, provides additional gameplay elements, such as different depth levels in which airplanes can be in a given moment. Therefore, figuring out which genre suits best *Sharky’s Air Legends*, arguably *Shoot’em ups*, would bring yet another polemic discussion.

Figure 15 – Sharky’s Air Legends: hybrid evolution from Shoot ‘em Up genre

In short and in contrast to intuition, game genres *alone* are not reliable enough to comprehensively describe and envision a domain in the context of a digital game SPL. They could lead to ambiguities and difficulties in other Domain Engineering activities, such as when selecting domain samples to analyze during Domain Analysis. For example, a loosely defined “Racing Games SPL” can mean different things, as presented in Figure 16. Such a title does not really tell if the game SPL is aimed at producing perspective racing games such as GeneRally, arcade racers such as Daytona USA or simulation-style racing games such as EA Sports’ F1 series, not to speak about other possibilities such as street racing games, off-road racing games and others specifically focused on motorcycles, trucks, speedboats or even horses. On the other hand, the game SPL may still mention a game genre in order to introduce some level of familiarity to its stakeholders, but needs to supplement it with clear expectations as presented in subsection 4.2.2.1.

Figure 16 – GeneRally, Daytona USA and EA Sports F1

4.2.2 Describing Game Domains through a Game Domain Vision

Instead of embracing the challenge of reaching the perfect game genre taxonomy, we are actually focused on ensuring that digital games development can benefit from systematic planning and management in the context of a specific application family. Taking as motivation the ambiguous universe of game genres and the need to constrain the big sample set of game instances and features to be analyzed, we propose an alternative solution: the definition of a **game domain vision** to *describe* a game family that share functionality. Such a vision intends to reach a common agreement on what ultimately defines the domain to be approached, independent of game genres. It does so by means of the following tasks, detailed in the next subsections: setting expectations for core game dimensions, establishing a negative scope, identifying target platforms and creating a vision statement.

4.2.2.1 Setting Expectations for Core Game Dimensions

If from one hand the digital games macro-domain is broad, requiring game SPL designers to narrow it down so that it becomes viable in the context of Domain Engineering, from the other

hand such a macro-domain is still more specific than the broader “software” concept, lying somewhere in between. Hence, game SPL designers can use **core game dimensions**, which are ubiquitous to digital games development yet not overly specific or generic, to drive the envisioning phase of a digital game SPL. From a feature modeling perspective [Kang et al., 1990], core game dimensions can be understood as very abstract, high-level mandatory root features that each specific digital game domain should address in its own way. While we believe core domain dimensions and domain envisioning in general can also be applied to envision macro-domains other than digital games, such a discussion is out of the scope of this research.

The list below presents the suggested digital games core dimensions whose expectations should be established by game SPL designers:

- **Player:** concepts related to the game player(s), such as number of players, co-playing modes (e.g., in turns or simultaneously, cooperative or “death-match”, etc.), score, high-score, lives and others. This should not be confused with “main character” entities controlled by the players.
- **Graphics:** what players are supposed to see, including the world view (2D, isometric, first-person, etc.), heads-up displays (HUDs) and eventually more advanced techniques such as particle systems to simulate fire, dust, rain, etc.
- **Flow:** how the plot or storyline of the games evolve as perceived by players, encompassing levels, phases, missions, screens, transitions, rooms, scenes, etc.
- **Entities:** the underlying types and mechanics of beings and things that players are supposed to control and interact with, such as main characters, non-playable characters, items, projectiles, etc.
- **Events:** triggers and reactions that drive the behavior of the world, screens and entities of the games belonging to the domain.
- **Input:** how players provide input to interact with the SPL games, encompassing (a combination of) devices and eventually more advanced options such as speech recognition and controller-free systems.
- **Audio:** what audio feedback players are supposed to get from the generated games, including sound effects, background music and optionally more advanced technologies such as 3D sound, speech synthesis (text-to-speech) and special effects (echo, pitch, reverb, bullhorn, etc.).
- **Physics:** the physical mechanics of the produced games, including collision detection and acting forces, such as gravity, attrition, slippery surfaces, etc.

- **Artificial Intelligence:** artificial intelligence behaviors performed by entities and the world of the SPL games, such as path following, context awareness, etc.
- **Networking:** whether the produced games are standalone applications or interact with servers (to store high scores, for instance) and/or other running game instances.
- Any other **custom core dimension** that applies to the specific game domain to be explored. For instance, an important core dimension that constrains role-playing games (RPGs) is the *Battle System*, which determines whether fights against enemies happens in turns or as real action, randomly or planned, etc. On the other hand, card games can have special constraints based on card decks, such as the number of decks, usage of full or partial decks, etc.

The expectations for such core dimensions come from multiple informal sources, such as expertise from domain experts, previous knowledge from domain analysts, trends and influences from successful game titles, requirements from game developers or designers and an overall assessment of the high-level goals of the game SPL. Once again, by no means the resulting set of expectations should be considered final or totally accurate. On the contrary, it will very likely be modified and refined by subsequent activities and iterations.

4.2.2.2 Establishing a Negative Scope

A negative scope is focused on stating expectations that will **not** belong to the specific game domain being defined, and therefore will have no built-in support from the SPL assets, such as DSLs and transformations. This task is especially useful regarding expectations that SPL users (game developers and designers) would implicitly take for granted, but are out of scope by design. For example, a SPL focused on a domain involving racing games may explicitly state, through its negative scope, that refueling and campaign modes are out of its expectations.

Initially, negative expectations can be defined for each core game dimension. Later, the Domain Analysis activities can refine the negative scope, explicitly stating the domain features that should not be taken into account. Concerns belonging to different points of view influence the negative scope, such as:

- Management concerns: risky implications to the game SPL project schedule or lack of resources due to expectation/feature complexity;
- Manufacturing concerns: expectation/feature is hard or impossible to automate and configure, reducing the SPL effectiveness for it;

- Requirements concerns: expectation/feature is not critical to end-users or is rarely used;
- Analysis concerns: expectation/feature is so vague or ambiguous that it cannot be considered unless better refined;
- Conciseness concerns: strong mismatch between expectation/feature with the current SPL vision, leading to a loosely related set of SPL assets.

Defining a negative scope is not only useful to help clarifying the boundaries of the approached domain, but also to avoid wasted work during subsequent activities. For example, during Domain Analysis, a deeper investigation of some features of a game can be discarded upfront if such features are identified as falling into the game SPL negative scope.

Negative scoping, however, does not avoid game SPL users from adding the out-of-scope features to their game instances, as extensions to the SPL. For example, a game entity movement can be originally restricted to a built-in set or formula (e.g., 8-directions movement through arrow keys), but be extensible enough to enable game developers to define alternative possibilities (such as mouse position-based movement) to move entities around. In fact, the SPL can still provide extensibility mechanisms ("hooks"), such as parameterization, partial classes, events, sub-classing, polymorphism, dependency injection [Fowler, 2004] or specifically defined join points that could be useful to extensions based on Aspect-Oriented Programming [Kiczales et al., 1997]. This discussion is detailed in Section 4.4.

4.2.2.3 Identifying Target Platforms

We suggest that the game domain vision includes the **target platforms** to be supported by the SPL, such as consoles, mobile devices, PC (i.e., a client operating system), web (i.e., browsers), digital TV, etc. Constraints on the target platforms, such as the need for a specific browser technology (Flash, Silverlight, etc.), operating system or runtime can also be described if the information is available, or as a result of refining the domain envisioning in future iterations. *Product portfolios* [Nascimento, 2008] can be used to describe all families supported by the game SPL, by means of their target platforms. A *product map* can also be conceived in order to map capabilities and restrictions of core game dimensions into the target platforms. For example, one of the target platforms in a mobile games domain may have its graphics expectations restricted to some specific screen sizes.

It is worth noticing that infra-structure decisions such as the target programming language or architectural elements to be used (such a specific version of a game engine) should not be included at this time, since they are not really part of the vision, but are actually a consequence of (and means to enable, or implement) the domain vision and requirements.

4.2.2.4 Creating a Vision Statement

A comprehensive yet concise **vision statement** summarizes the essence of a game domain vision. Although the vision statement is established only after other game domain vision elements, it will generally be presented first when introducing the game SPL to stakeholders, as it happens with an executive summary of a business plan.

As highlighted by agile development methodologies [Cockburn, 2001], analogies are a powerful mechanism to facilitate understanding. In order to clarify and illustrate the expectations for a core dimension, the vision can still establish analogies to game genres or well-known titles in the industry. However, as previously mentioned, the whole vision itself should not solely rely on genres for describing the domain.

The vision should not be too broad so that it attempts to encompass too many genres and instances, since that would result in excessive variability and ineffective game SPLs. On the other hand, if the vision is too narrow, it may be difficult to reuse processes, components and tools, making it more difficult to achieve a return on the game SPL upfront investments. Likewise, the vision should be comprehensive, but not meticulously precise. False positive samples (games that are initially evaluated as complying with the vision but are later discarded by Domain Analysis) can still exist. As the vision gets refined by Domain Analysis, the domain boundaries are made clearer.

The vision is independent from game-world content, unlike other works of fiction such as films or books. For example, an action game is still an action game, regardless of whether it takes place in a fantasy world or outer space [Rollings & Adams, 2006]. If game SPL designers foresee that different player profiles (or even other types of end-users) will be addressed by the game SPL, Personas [Bonnie, 2007] can be modeled and become part of the vision. Examples of Personas include teachers and students in educative games, hardcore and casual gamers for games with multiple levels of difficulties, different content generator roles for customizable games, etc. The specific needs of each Persona can be used as input when conceiving game SPL assets later on.

Game SPL designers have an additional dilemma to deal with when envisioning the domain. If from one side each wave of games is attempting several mysterious and unproven technical feats (and experiences) to surprise players [Blow, 2004], on the other hand there should be a threshold on any core dimension innovation, in order to avoid it from becoming a rupture. For example, if a game makes use of well-known mechanics, such as the use of two directional sticks (one to move and the other to aim) in first-person shooter games, then the learning curve for the players will be considerably lower. Some game developers even argue

informally[19] that 95% of games should really be sticking with what players know to reduce the barrier of entry. However, this should not rule out a completely new gameplay experience, especially if it is solidly based on user experience research. Nintendo's Wii controller, which recognizes the players' gestures, as well as Microsoft's Kinect, which brings the player's body movements to the game screen, are successful examples of innovation which did not become ruptures.

Finally, the end product of Domain Envisioning can be customized by the game SPL in order to more properly fit the contents and layout of a Game Design Document already in use by the organization. In fact, they can be used as a Game Design Document *template*, to be refined by game SPLs and instantiated during actual application development (Section 4.7).

4.3 Analyzing the Game Domain

For almost three decades, the area of Domain Analysis has been serving other Software Engineering initiatives such as Component-Based Development, Software Product Lines and Software Reuse in general. The term was first introduced by Neighbors [1980] as "the activity of identifying the objects and operations of a class of similar systems in a particular problem domain." It can also be defined [Prieto-Diaz, 1990] as a process by which information used in developing software systems is identified, captured and organized with the purpose of making it reusable when creating new systems.

Evolving the game domain envisioning discussion presented in the previous section, the next subsections discuss the peculiarities of performing Domain Analysis for game SPLs. The final purpose is to gather a set of abstractions that comprehensively represent the games that belong to the chosen domain and, therefore, that can be used as input to design and implement core SPL assets.

4.3.1 Building the Game Domain Vocabulary

Virtually all tasks in Domain-Specific Game Development, from analyzing domain features to designing DSL concepts, require understanding the game domain vocabulary. Kelly & Tolvanen [2008] mention that most people already have a domain-specific vocabulary in use and it exists for a good reason: it is relevant when discussing a family of systems. The authors point out that there is no need to introduce a whole new language to the domain since one is probably already in use, albeit implicitly and partially. Hence, the domain analyst

[19] Doolwind's Game Coding Journal, http://www.doolwind.com/blog/?p=85

should be responsible for capturing such a domain vocabulary that will later result in other SPL assets, such as the DSL concepts, properties and relationships.

Nonetheless, Wiegers [1999] points out that “one problem with the software industry is the lack of common definitions for terms we use to describe aspects of our work”. Church [1999] enforces that game development is not an exception to the rule. He states that game design, an activity which includes the definition and use of a common vocabulary, is the least understood aspect of computer game creation, and that the primary inhibitor of design evolution is the lack of a common design vocabulary, despite the fact that understanding requires that designers are able to communicate precisely and effectively with one another. In short, Church says, “we need a shared language of game design”.

The demands for creating a game design “critical language”, a way to analyze games, to understand them and to understand what works and what makes them interesting is not new. In 1994, Costikyan [1994] already mentioned that such a challenge relates not only to digital games, but to games in general (digital, tabletop RGPs, virtual reality, sports, mass-market adult, card games and so on). However, compared to the vast body of operational knowledge found in the world of filmmaking, for example, the game design community is just beginning to articulate the concepts and techniques specific to its medium in order to establish methods of game design [Kreimeier, 2003]. Kreimeier [2003b] concludes that “[…] while knowledge about computer games has grown rapidly, little progress has made to document our individual experiences and knowledge, documentation that is mandatory if the game design profession is to advance. Game design needs a shared vocabulary to name the objects and structures we are creating and shaping, and a set of rules to express how these building blocks fit together”.

While game designers have called for a design language, noting that they currently lack a unified vocabulary for describing existing games and thinking through the design of new ones, many approaches to address such request have risen. Zagal et al. [2005] identified that many of the proposed approaches focus on offering aid to the designer, either in the form of design patterns [Kreimeier, 2003b] [Björk & Holopainen, 2005] [Björk et al., 2003], which name and describe design elements, or in the closely-related notion of design rules, which offer advice and guidelines for specific design situations [Fabricatore et al., 2002] [Falstein, 2006]. Other analyses draw methods and terminology from various humanistic disciplines. For example, games have been analyzed in terms of their use of space [Jenkins, 2003], as semiotic systems [Kücklich, 2003], as a narrative form [Church, 1999] [Murray, 1997], in terms of the temporal relationships between actions and events [Eskelinen, 2001] or in terms of sets of features in a taxonomic space, using clusters in this space to identify genres [Aarseth et al., 2003].

Such approaches present valuable contributions to the digital games development community, but they are not committed to abstracting the commonalities and differences in design elements across concrete examples belonging to a specific domain. They either offer imperative advice to designers, intend to describe rules for creating good games, or try to develop definitions to distinguish between games and non-games (or among different types of games). Such goals do not properly address the task of creating a common vocabulary for a game domain and, therefore, a game SPL.

Zagal et al. [2005], through the Game Ontology Project, aim at defining a broad ontology for digital games in general. Such an ontology cannot be used in a game SPL, since it is not domain-specific. Other less formal attempts for building a game vocabulary (or glossary), such as the GameDev.NET Game Dictionary[20] and The Game Programming Wiki[21], are also too broad in scope, including topics not directly related to game design and development, such as the name of celebrities, companies or general network infra-structure definitions.

Given that as a motivation and coping with the need to establish a common terminology for the game domain to be approached, we suggest the creation and maintenance of domain-specific vocabularies as part of the game SPL's Domain Analysis. This encompasses the domain concepts elicitation, which can come from game development tools, game samples, documentation associated with games (manuals, reviews, screenshots, etc.), game magazines, academic articles, interviews with game players and developers, core game dimensions and game domain features. Special focus should be given on ambiguity and contradiction elimination, as well as the identification and merging of synonymous concepts. Game designers, developers and other domain experts should be encouraged to validate the vocabulary.

Without agreeing on a vocabulary for the specific game domain to be approached, the different roles involved in the creation of the game SPL (domain expert, analyst, architect, implementer, etc.) may reach different definitions for the domain terms, which is a negative outcome for game SPL since creating domain-specific assets is all about clearly understanding and agreeing on definitions.

4.3.2 Defining and Refining Game Domain Features

At each iteration, the group of domain features to be analyzed should be defined and/or refined. When the domain vision was defined (Section 4.2), the expectations of a series of core game dimensions were established and can now be used as a starting point for this task.

[20] http://www.gamedev.net/dict/

[21] http://www.gpwiki.org

The iterative nature of the process and a consequent deeper understanding of the domain add, remove, merge, split and change the features in the feature set.

The execution of this task has some peculiar implications to digital games due to their nature. Software factories commonly approach the modeling of domain features by making a clear separation between the problem domain and the solution domain [Greenfield et al., 2004]. Modeling the former is an input to modeling the latter, which is ultimately used to bridge the Domain Analysis to subsequent phases (Domain Design and Implementation).

In digital games, however, the *problem* domain is unusual since the concept of user requirement is not clear. Raph Koster, in his acclaimed book *A Theory of Fun for Game Design* [Koster, 2004], concludes that we play games not to get end-user requirements satisfied, but simply because games offer "juicy patterns for our brains to consume". As the human brain is addicted to learning new patterns, it welcomes any activity that teaches it something new, until it loses the interest due to a mismatched difficult level, i.e., it becomes too easy or too hard to assimilate new patterns. As a consequence, the problem domain for digital games permeates the subjective topics of pattern-matching, fun, immersion, escapism, delight, competitiveness and others, combining areas that range from social behavior to adrenalin/dopamine, which are better addressed by psychochemical sciences rather than Software Engineering. If in other industries the requirements and design patterns (such as "safety" and "comfort" in the automobilist industry) clearly map to the solution domain (airbags, anti-break systems, automatic transmission, hydraulic steering, etc.), in digital games such mapping is not evident (or mature?) enough.

Likewise, while requirements and SPLs in general evolve as a direct consequence of identifying new user needs, in digital games such evolution is mostly based on **experimentation** and **creativity**, which typically refine an abstract theme such as "time manipulation" or "seek and hide". This way, although it is well known that software in general is very likely to change during the development process, the churn seems to be higher for digital games since interim experimentation and exploratory results can radically change and shape the final product. For instance, in some professional game development studios, designers are not required to come up with a detailed game specification until the first playable prototype is approved.

The implications of that to game SPLs are twofold. First, modeling the problem domain for emotion-based features does not seem to be useful to the game SPL, due to their subjective and cross-discipline nature. Typical examples of emotion-based problem domain features are "appealing physics" and "nostalgia", which can only benefit from Domain Engineering processes if their understanding and underlying requirements are made more concrete and specific to the domain. On the other hand, some non-emotional game features can

still exist and be traced back from the solution domain to the problem domain, as illustrated inTable 4 (the list can increase for domains in which games have secondary goals, such as in educative and serious games). In such a case, we suggest SPL designers to employ the same guidelines proposed by Greenfield et al. [2004], which evolves the problem domain along with the solution domain for the SPL.

Table 4 – Non-emotional features: tracing between problem and solution domains

Problem Domain Features	Solution Domain Features
Allow breaks to avoid having players loose progress	Save/Load, Pause/Resume, "Continues"
Register player performance	High-scores table, achievements
Provide social interaction	Multiplayer mode (online and local)
Establish a player identity	Avatar, game elements customization
Provide availability (to play independent from time/space)	Mobile platforms, digital convergence (multi-device experience for a same game)
Ensure readiness to play	Intuitive/one-click installers, zero-deployment games
Offer replay-value	Multiple narrative paths, multiplayer support, achievements
Establish a low learning curve	Tutorials, scaffolding (hints and tips that stop being offered as players acquire experience)
Advertise a specific brand (typical for advergames)	Hooks for brand insertion, which can end up as patterns: background of “loading” screens, mid-action fly-outs, specific areas or canvases designated for branding, etc.
Teach or train the player on a given real-world topic	Missions and problem-solving challenges that incorporate the topic contents, notorious in serious end educative games.

The second implication of the emotion and experimentation nature of many game features to game SPLs is that virtually every instance of the SPL will require extensions, not supported as built-in. Later on, of course, extensions may be retrofitted to the game SPL, as part of its feedback process.

Braid (Figure 17) is an Xbox 360 Live Arcade game that very conveniently illustrates this discussion. While it has all typical gameplay elements of a platform game, it innovates by adding **time interaction and manipulation** to the gameplay experience. For example, in one of Braid’s phases, if the main character moves to the right, time advances for all other entities

of the game. On the other hand, if the main character moves left, time goes back for other entities, i.e., they undo the actions they have previously done, such as by moving back to their original positions. Other game features are also impacted by time flow manipulation, such as the background music, which is played in reverse mode when time goes back.

Figure 17 – Braid: time flow manipulation extends built-in feature set of platform games

The features that realize such “time manipulation” abstract theme were very likely *not* conceived as the result of a well-defined Domain or Application Engineering process focused on user requirements. More probably, such a feature came from exploratory processes leveraging previous game design experience, and was gradually validated by experimentation through prototypes. Supposing that Braid was created in the context of a platform game SPL, however, such time flow manipulation feature could be retrofitted into the SPL as part of its feedback process. The feature could be refined into more detailed and well-understood features, such as “entity actions recording”, “entity actions playback” and “entity actions rollback”. On the other hand, it could also impact already existing features. For example, the “background music” feature could be parameterized in order to allow the game background music to be played in reverse mode. As a practical consequence of that, SPL assets (languages, frameworks, etc.) would be adjusted to become compliant with and enact the updated feature set.

4.3.3 Selecting Domain Samples

Selecting domain samples refers to identifying existing, under development or anticipated products to be analyzed. Such products were probably not created as part of a product line approach. On the contrary, they may have been produced through one-off development processes with little or no reuse intended at all.

The number of games to be analyzed is mainly constrained by the game SPL project resources and schedule. Therefore, a very important concern is to select games which are the most representative. For instance, a game which was either re-released through "re-makes", had many sequels or received broad industry and media recognition can probably be considered a representative sample.

An important question that is raised during the selection of product line samples is where to discover them. The elicitation task may benefit from the involvement of domain experts, end-users and specialized sources of information. As a rule of thumb, successful previous titles in the industry, which can be classified as belonging to the game SPL domain, are the most reliable samples. The ArcadEx case study (Chapter 5) illustrates this discussion.

4.3.4 Analyzing Game Samples and Modeling the Game Domain

The Game Domain Analysis registers information about each one of the selected product line samples against the currently specified feature set, in order to identify the commonality and variability[22] of the domain. The gathered information is typically organized in a catalogue and is used as input to create **feature models** that abstract the features of the individual domain samples into a feature hierarchy that represents the games that belong to the domain. As for tool support, Lisboa et al. [2010] have compiled a systematic review on Domain Analysis tools, including feature modeling ones.

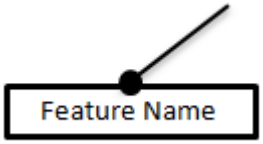
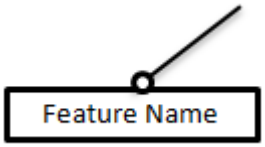
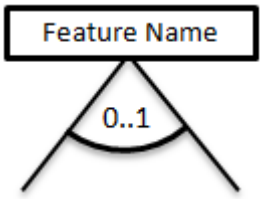
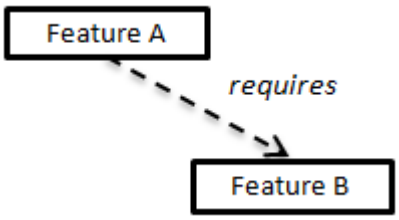
Feature modeling was proposed as part of the Feature-Oriented Domain Analysis (FODA) method [Kang et al., 1990], and since then, has been applied to a number of domains including telecom systems, template libraries, network protocols, and embedded systems. A feature is a system property that is relevant to some stakeholder and is used to capture commonalities or discriminate between instances. The features are organized in feature diagrams, which are trees whose root represents a concept (e.g., a software system), and whose descendent nodes are features.

A number of extensions and variants of the original FODA notation have been proposed [Czarnecki et al., 2004b]. This research suggests the use of the cardinality-based feature model notation [Czarnecki et al., 2004], especially because it provides expressiveness for establishing constraints between features and for telling the range of possible features that belong to an alternative node. Elements of such feature model notation include mandatory features, optional features and alternative feature sets that group similar or interchangeable features. A cardinality constraint (introduced in feature models by Riebisch et al. [2002]) can be used to avoid ambiguity in alternatives. Finally, constraints between features (such as

[22] Variability is the ability to change or customize a system [Svahnberg et al., 2001].

requires or *excludes*) can be applied as well. A summary of such a notation is presented in Table 5.

Table 5 – Cardinality-based feature model notation

Notation	Description
	**Mandatory feature.** All instances of the domain should have such a feature. For example, all game entities should have a position on the screen.
	**Optional feature.** An instance of the domain *can* have such a feature. For example, a game entity may have an opacity value.
	**Alternative with explicit multiplicity.** Multiplicities enable a more powerful and less ambiguous model and are similar to those proposed by the Unified Modeling Language (UML), such as *at most one* (**0..1**), *exactly one* (**1**), *any* (**0..***) and *at least one* (**1..***). As an extension, custom ranges are also allowed, such as **1..3**.
	**Constraint.** In this case, the *requires* constraint indicates that whenever Feature A is present in a domain product, Feature B should be present as well. Other constraints introduced in other works [Kang et al., 1990] [Czarnecki & Eisenecker, 2000] [Riebisch, 2003], such as *excludes*, and *refinement*, can be used as well.

A feature model represents an abstract view onto properties of all instances of a domain [Riebisch, 2003]. Every feature covers a set of requirements. By selecting a set of optional features, an instance of that domain can be defined. All mandatory features are part of the instance by definition. Finally, hierarchical relations between a feature and its sub-features control the inclusion of features in instances. If an optional feature is selected for an instance, then all of its mandatory sub-features have to be included as well, and optional sub-features can be included.

Figure 18 presents a typical feature model diagram contemplating the aforementioned notations elements. Mandatory features have a black circle decorator in their connection to the parent feature, while optional features have a white circle. Feature sets are denoted by

arcs, displaying the minimum and maximum number of options that can be selected. Decorators do not apply to the root concept neither to alternative options, since occurrence semantics for them are well-understood (a root concept always occurs; alternative options occurrence is determined by the alternative cardinality).

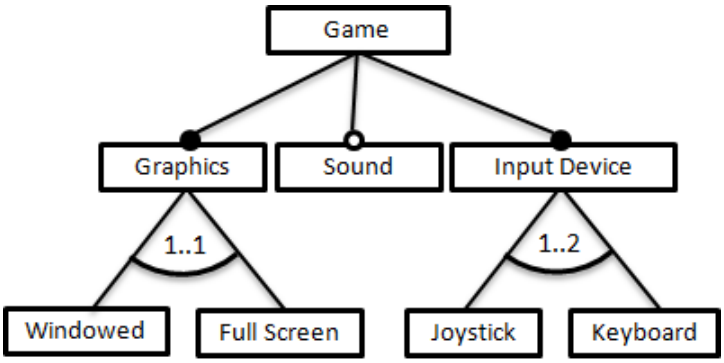


Figure 18 – Example of feature model diagram

It is important to point out a couple of peculiarities that stand out when analyzing samples for game SPLs. Sometimes, a game has sequels consisting of very similar titles, such as Pac-Man, Mrs. Pac-Man and derived variations. In such a case, SPL designers can opt for analyzing them as a single group, considering unique sequel features as extensions or variations of the original game. If the sequel/variation games have too expressive peculiarities on their own, this may be an indication that the SPL domain can be partitioned into subdomains, as explained in the next subsection.

The analysis of samples includes activities majorly focused on, but not limited to *playing* the actual games. If the hardware is not available, emulators should be considered. It is worth noticing that the SPL designers should take into account any legal implications of using such tools. Some examples of emulators are:

- DOSBox[23], an open source x86 emulator with DOS, primarily focused on running DOS games;
- MAME[24] (Multiple Arcade Machine Emulator);
- Specific video game console emulators (Stella for Atari systems, ZSNES for Super Nintendo, etc.);
- Operating system virtualization/emulation suites such as Microsoft Virtual PC[25].
- Online ROM emulators, typically implemented in Java, Flash and Silverlight.

[23] http://dosbox.sourceforge.net

[24] http://www.mamedev.org

[25] http://www.microsoft.com/windows/products/winfamily/virtualpc/default.mspx

As opposed to software in general, whose design if focused on enabling features to be easily reached and explored by end-users, many functionalities in a digital game are locked from the beginning. For instance, some levels should be cleared in order to unlock advanced levels. Such advanced levels, on the other hand, might reveal additional behavior (features) not originally noticed in previous levels. The web game *RunMan: Race Around the World*[26] is an interesting example: the concept of a world map, which links different playable areas, is not known by the player until all levels from the first area are cleared.

That said, it might not be trivial to explore all features of a digital game, and consequently perform Domain Analysis properly, without playing (and many times mastering) the game. The lack of specifications or access to game design documents negatively impacts such an already challenging concern. This research suggests the following techniques to enable game domain analysts to overcome such issues:

- Enabling "god modes" or activating "cheat codes", if available. Those resources give enhanced powers to players such as the freedom to teleport across game levels or an unlimited number of lives, ammo, etc. However, when using such techniques, the game domain analyst should keep track on what a built-in game behavior is versus what was modified, so that the game analysis is not jeopardized. It is also worth noticing that god modes and cheat codes are, by themselves, features that can be addressed by the game SPL.
- Exploring official and "underground" literature related to the game, which is very vast since addicted players worldwide do a very good job in documenting the behavior, scripts, characters and many other attributes of their favorite games. This includes strategy guides released by the publisher or others, playbooks[27], specialized magazines, reviews, recordings of previous play sessions, online forum conversations and logs. For instance, McGonigal [2010] highlights that the World of Warcraft wiki, with more than 8,000 thousand articles and 5 million contributors, is the second biggest wiki of the world, behind only Wikipedia. There is more information about the World of Warcraft game than any other topic covered by any other wiki in the world.
- Interviewing experienced players who master the game or the game domain. Such players can also be interviewed for eliciting anticipated game features that do not exist yet (Section 4.3.6).

[26] http://whatareyouwait.info

[27] An instruction book containing play scripts or diagramming various possible plays to be performed.

While the feature model notation tells variability by means of documenting whether a feature is common or optional in a domain, it cannot tell to what extent an optional feature is variable. In other words, it may be impossible to document, using a feature model, all possible flow configurations among the screens of the games belonging to a domain (e.g., only one screen, two screens where the first screen leads to the second, two screens configured in a loop, three screens, etc.). This is not specific to the digital games development domain, though. Since understanding how variability behaves for the features has a huge impact on how core domain assets such as DSLs are conceived, it is recommended that the game domain analyst annotates the feature model at least with textual information describing the variability universe for a given feature. Section 4.5 clarifies how such information is used to bridge the feature model and the Domain Analysis to the core assets implementation.

4.3.5 Partitioning the Game Domain into Sub-Domains

During Domain Analysis execution, SPL designers will commonly be able to partition the SPL domain into sub-domains, which define an even more related set of features. For example, in a game domain based on old-fashioned arcade games, a couple of sub-domains can be identified. Games such as King & Balloon, Space Invaders, Galaga and Galaxian (Figure 19) fall under the same sub-domain, that could be called *Bottom-up Shooter*. On the other hand, a *Maze* sub-domain could be defined to encompass games in which entities are confined in a field composed by a set of blocking walls (a maze) and should collect items, navigate to the exit or solve a puzzle, while escaping from NPCs. Variations of Pac-Man, Crash and Side Track (Figure 20) are samples that belong to such a sub-domain.

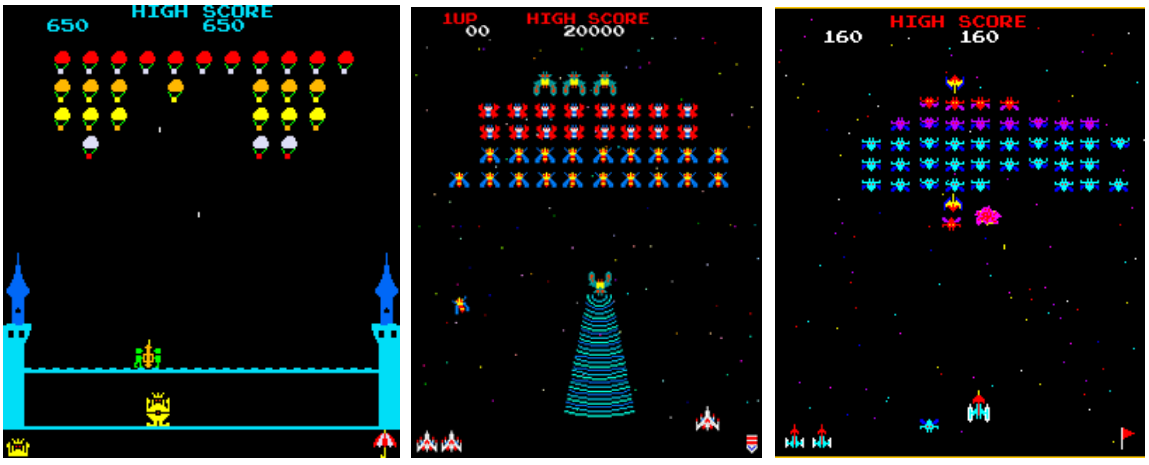


Figure 19 – King & Ballon (left) and other Bottom-up Shooter games

The greatest contributions of sub-domains to a game SPL design is that they are even more specific, leading to more expressive and effective assets, such as domain-specific languages. If from one side sub-domains are important to reinforce commonalities previously

identified, from the other side they provide a new ground to look up for special variability which was not yet taken into account.

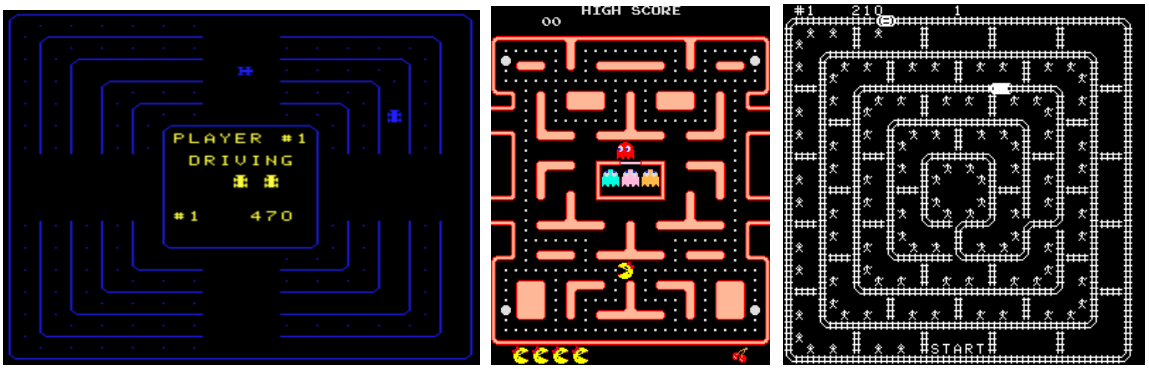


Figure 20 – Crash, Mrs. Pac-Man and Side Track belongs to the Maze sub-domain

For example, in spite of belonging to the aforementioned created *Bottom-up Shooter* sub-domain, the game Teeter Torture (Figure 21) presents a unique concept: a seesaw platform, which impacts both the main character movement as well as the game logic, introducing a more elaborated physics modeling. This can be considered a special variability for the game's sub-domain.

Figure 21 – Teeter Torture

From the SPL design point-of-view, many paths can be taken once game sub-domains are identified. A game sub-domain may be too complex or not strongly aligned to the original domain, causing the domain analysts to rethink the SPL's (negative) scope and cut out from it the features that distinguish the sub-domain. On the other hand, the sub-domain may stand out from others as an interesting challenge which would provide a better return of investment, implying in its promotion as the new SPL domain, while all other sub-domains now become part of the negative scope. Finally, domain analysts can decide that the game sub-domains will coexist. In such a case, Domain Analysis turns into a recursive

task: the same process applied to the original SPL domain should be applied to each of the sub-domains.

The analysis of game sub-domains, however, can turn into a challenge for some reasons. First of all, it can be intriguing to classify a sample into game sub-domains. The game Defender (Figure 22), for example, is a hybrid of *collector* and *projectile-based* game, and would therefore fall into both possible sub-domain classifications. As a result, similarly to the previous discussion on game genres, it can be concluded that there is not necessarily a 1-to-1 correspondence between game samples and sub-domains.

Figure 22 – Defender: hybrid game belonging to multiple sub-domains

Conversely, sub-domains can also unfold into other more specific, "sub-sub-domains". As domains and sub-domains get nested into deeper levels, the Domain Analysis effort inevitably grows. Another relevant challenge relates to when to consider a feature set variation strong enough to justify the creation of sub-domains. The two extremes are:

- One single domain with high feature variability.
- Many sub-domains, each one with low feature variability.

Game domain analysts should find a balance between these extremes, using a mid-term approach in which the need for sub-domains is justified by a more modularized creation and usage of SPL assets, such as domain-specific languages and frameworks (game engines).

4.3.6 Revisiting the Game SPL Domain Scope

The previous subsection presented how the Domain Analysis can identify sub-domains to reinforce previously identified commonalities and discover special variability which was not yet taken into account. On the other hand, it may be the case that the opposite happens, in which a sample is considered to be out-of-scope since it does not belong to the game SPL

vision. Such cases are an opportunity to reinforce or adjust the negative scope of the SPL with more concrete examples of out-of-scope expectations for core game dimensions, which were still not clear due to blurry vision boundaries. Three interesting scenarios when revisiting the game SPL domain scope follow below. They are illustrated in more details by the ArcadEx case study (Chapter 5).

- **Conflicting features:** some analyzed game samples may be initially considered as compliant to the game SPL scope. However, further investigation might reveal some not-so-obvious feature variability that actually conflicts with the SPL vision. Creating sub-domains to encompass such games with special features is not an interesting approach. A sub-domain is a specialization of the main domain and respects the SPL vision and scope, while such samples actually present features that conflict with the SPL vision and scope. Moreover, adding out-of-scope features to the game SPL may be dangerous, especially if the SPL increases its scope in a way that it gets too generic, loses its specific focus and consequently provides a reduced effectiveness, while becoming harder to maintain. A better approach would be to create other SPLs to address the currently conflicting features. Some assets, however, can still be shared across SPLs (Section 4.6), especially if properly modularized and provided with parameterization and customization capabilities.
- **Incorporating features:** on the other hand, adding an out-of-scope feature to the SPL scope may be acceptable if the feature is actually a variation or builds upon previous features to make them even more specific, instead of adding an extra feature overload that would make the SPL generic. In such a case, the game SPL scope is increased.
- **Anticipating features:** the identification of commonality and variability should not be restricted to the features identified in the analyzed samples. Game domain experts, together with game domain analysts, may foresee innovative features that could enhance the generated games and therefore extend the feature model with new *anticipated* features, or even new anticipated sub-domains. Therefore, besides existing applications, both future applications (i.e., applications where the requirements are rather clear, but development has not yet started) and potential applications (i.e., applications for which no clear requirements exist yet, but that are seen as relevant) should be considered in the Domain Analysis. Examples include supporting additional input devices in the game SPL (such as light guns, steering wheels, or even controller-free experiences) or “warp zones” (portals through which players can “tele-

port" from one place or level to another). Some useful techniques can be employed for anticipating features for game domains:

- **Retrospection and trend analysis** [Araújo, 2009]: in such a technique, the goal is to identify the first ancestors of a given artifact (such as a game sample or feature) and understand how they evolved along time. From this information, it may be possible to project future trends, new resources and functionality that can be built atop the current artifact state. The high availability of game samples (for instance, due to the "retro" branch of the game developer and player communities) makes this technique not only valid but encouraged.
- **Brainwriting:** differs from brainstorming by having each participant to think and record ideas individually, without any verbal interaction. Such a technique is motivated by studies revealing that participants working in isolation consistently outperformed participants working in groups, both in quantity and quality of ideas generated [Diehl & Stroebe, 1987]. In a round table, each participant has a limited amount of time (such as 3 minutes) to write down his ideas. Then each participant passes his sheet of paper to the participant to the left, who must come up with three new ideas. After the idea-gathering phase is completed, the ideas are read, discussed and consolidated with the help of a moderator, just like in traditional brainstorming.
- **Morphologic box** [Zwicky, 1948]: created by the Swiss astrophysicist Fritz Zwicky, this technique is a systematic form of idea finding where a solution for a problem is searched by trying out combinations from a matrix containing solution topics (or variables) and their possible values. The matrix is called a morphologic box and the combination of its values can result in unusual or "wild" solutions, coping with one of the basic ingredients of creativity, essential to game development and design.

4.3.7 Testing Sample Analysis against Stop Criteria

In Domain Analysis, experience and knowledge are accumulated until it reaches a threshold. However, understanding whether a domain is sufficiently mature to develop reusable assets (and build a SPL upon it) is a challenge. Almeida et al. [2006] assume that such maturity *"can be initially defined by the organization; however, a better solution must be investigated"*.

Therefore, this subsection provides some discussion and guidelines on the subject, which are believed to span beyond game development.

As previously mentioned, the Domain Analysis process, instantiated or not to game SPLs, should be incremental. Incremental processes, however, should count on well-defined stop criteria, such as the completion of an artifact or the achievement of a milestone, so that the methodology can proceed to the next phase. Of course, if the incremental process is also iterative, the phases are also revisited, in contrast to waterfall methodologies.

Stop criteria for a Domain Analysis iteration in SPLs can be defined beforehand, as a consequence of the project constraints (schedule and/or resources), based either on:

- Time (“analyze as many samples as possible in timeframe X”);
- Budget (“analyze as many samples as possible within budget Y”);
- A combination of criteria;
- A pre-defined sample amount (“analyze Z samples”), which may be the result of other project constraint function.

Independent from the project constraints above, there is an optimal number of samples which maximize the Domain Analysis results. A number of samples which is fewer than the optimal number negatively impacts the SPL effectiveness; more samples than the optimal number implies in wasted resources. Discovering the optimal number of samples, therefore, is the greatest challenge to overcome.

An important indicator that can be used to empirically tell if the Domain Analysis can stop is the **domain understanding churn**, i.e., how much the domain understanding changes after each sample is analyzed. If, after the analysis of the latest samples, new features and variations were identified, it means that the domain understanding is not comprehensive enough and the analysis should continue. On the other hand, if no relevant additions to the domain understanding happen after samples are analyzed, it can be assumed that the most important commonalities and variability were already covered and the effort to discover new ones, if they exist at all, may not be worth. In other words, after a certain number of iterations, the analysis of new samples in the game SPL stops contributing with more information to the domain understanding, but testifies that the already collected and classified data were indeed accurate and comprehensive. New analyzed games just reveal patterns already registered in previous sample analysis, although the contents of each game (sprites, story, etc.) are obviously different. At that moment, the analysis can be considered satisfactory and concluded for that iteration.

4.3.8 Validating the Game Domain

A series of activities can be taken in parallel to ensure the analyzed domain is validated. Such activities include discovering feature synonyms and homonyms, documenting the features (generally taking the domain expert knowledge as input) and documenting the domain itself. Additional details and guidance are provided by Almeida et al. [2006]. Ensuring the feature model and sub-domains are consistent with the game domain vocabulary (Section 4.3.1) should also be included as a domain validation task.

Nevertheless, tracing features back to requirements may be trickier to perform in the context of game SPLs, due to the nature of digital games. As previously mentioned in Section 4.3.2, the concept of end-users and requirements for digital games is considerably different from other software, since most of the language used to describe a game and its goals is already in the solution domain, rather than approaching the problem domain, i.e., user requirements. In such a case, the general guideline is to ensure that the analyzed features realize and are in accordance with the game SPL vision (which can be based in Game Design Documents, as aforementioned), while also ensuring the problem domain features which were actually modeled (Table 4) can be traced to solution domain features.

4.3.9 Assess Game Domain Automation Potential

Domain Analysis contributes to building reusable domain assets by identifying the main concepts from a domain and its scope, i.e., what will be included and what will be excluded from the domain. From a feature model perspective, the identification of reuse opportunities should consider the following [Almeida et al., 2006]:

- A feature model with AND-nodes at an upper level and OR-nodes at a lower level usually indicates a high level of reuse opportunity.
- On the other hand, alternatives (i.e., OR-nodes) at the upper level usually mean that applications in the domain do not share much commonality in terms of services and functions provided by them. This indicates that the domain might not have much reuse opportunity at the application level, although there might still be opportunities for reuse at low level.
- Additionally, alternatives (OR-nodes) at a lower level indicate different ways of designing and implementing certain reusable information.

On the other hand, Model-Driven Development (MDD) also promotes software reuse by reducing the semantic gap between the problem domain and the solution domain, using high-level models that shield software developers from the complexities of the underlying implementation platform [France & Rumpe, 2007]. MDD also uses transformations to automati-

cally generate implementation assets that reflect the solution expressed in the models [Schmidt, 2006].

MDD is a good fit to Domain Analysis as it can encapsulate the knowledge of how to implement features, their relationships and constraints [Ledeczi et al., 2001]. However, some parts of the domain are better candidates for MDD-based automation, when compared to others. Identifying and prioritizing such sub-domains is a joint responsibility of the game domain analyst and the domain expert. Such two activities is described next.

4.3.9.1 Identify Sub-Domain Candidates for Automation

Section 4.3.5 provided some initial discussion on what to take into account when partitioning the game domain into sub-domains. At that time, the goal was to identify more specific sub-domains into which games could be classified more precisely. Such an activity considers every game as an atomic entity that belongs, or not, to a given sub-domain.

In this activity, however, the game domain analyst should identify new sub-domains by fragmenting the atomicity of the sub-domains previously identified in order to more deeply explore game features and their relations. Examples of such sub-domains include the transition between game scenes or screens, the collision relationship between game entities and the possible graphical representations of UI elements such as head-up displays and menus. Such sub-domains alone are not comprehensive enough to define a game. On the contrary, different features of a single game fall under such sub-domains. This will ultimately lead to sub-domains whose automation potential, by means of MDD, can be individually and more confidently assessed.

There is not much information in the literature regarding how MDD fits into Domain Analysis, and not surprisingly this is as challenging for digital games development. Lucrédio et al. [2008] present a Domain Analysis approach for model-driven Domain Engineering projects, which is generic and can be instantiated to any macro-domain. Knodel et al. [2005] present an approach for using Model-Driven Development in Software Product Lines, called Pulse-MDD, in which the development of transformations and modelers is tailored to the architecture of the product line. However, in such an approach, MDD concerns arise only in Domain Design. Deelstra et al. [2005] describe an approach for model-driven product lines development, but do not present details on how to systematically create transformations that lead from domain models to concrete software artifacts.

Using such related work as input, the guidelines below aim at helping in identifying game sub-domains as automation candidates.

- Use the natural categorization of the domain [Maiden & Sutcliffe, 1996], by means of scanning the feature model that resulted from extracting the domain

commonality/variability (Section 4.3.4). Lucrédio et al. [2008] mention that closely related features are normally good candidates to pertain to a same sub-domain, while isolated features also provide good hints.

- Rely on the knowledge of the domain expert [Lee at al., 2002] [Maiden & Sutcliffe, 1996] [Haddad & Tesser, 2003] in order to further break down the characteristics of the game samples.
- Consider core game dimensions (Subsection 4.2.2.1) and features directly derived and elicited from them as sub-domain candidates.
- Investigate how types (classes, interfaces, enumerations, etc.) are modularized in sample implementations and game engines. For example, game engine modules and sub-modules can provide hints on possible sub-domain candidates.
- Investigate repetition in sample implementations [Lucrédio et al., 2008]. If some piece of design or code repeatedly appears in a sample or across samples, even if the repetition instances are not exactly the same, it is likely that a machine can do some parameterized copying and pasting, and it is worth to try to find a sub-domain here. Another technique is to search for implementation or design patterns [Knodel et al., 2005].

While some of such guidelines may seem to appear too low-level and early in the process, especially the last two, it is important to recap that the proposed "edge-center" approach is iterative, as introduced by Figure 13. As sample implementations and a reference architecture are built from the low-level bottom layers (Domain Design and Implementation), the current top-level discussion on identifying and prioritizing sub-domains for automation (Domain Analysis) becomes more informed.

4.3.9.2 Prioritize Sub-Domain Candidates for Automation

Once sub-domains are identified, they can be further refined following the techniques described in Section 4.3.4 (*Analyzing Game Samples and Modeling the Game Domain*) in order to improve their accuracy and level of detail, resulting in an enriched feature model diagram. However, the major focus of this activity is to understand the feasibility and return of investment on automating such sub-domains with Model-Driven Development. Domain analysts should weight the following variables in order to prioritize the sub-domain candidates:

- **Previous automation evidence:** are there existing modeling languages and modeling/generation tools for the sub-domain? Many times, prototyping and click-n-play tools (Section 2.3 Click-n-Play Tools) are a good input for that, as well as game engines and their built-in tools (Section 2.4).

- **Integration:** how easily can the modeling languages and tools (new or already existent) be plugged into the game product line? It may be necessary to adapt languages, tools, their inputs and outputs in order for them to be compliant with the envisioned target platforms (Subsection 4.2.2.3) and the product line environment, especially game engines.
- **Coverage:** does the sub-domain cover a bigger amount of features when compared to other sub-domains? If domain features were prioritized somehow [Moon et al., 2005] [Almeida 2007], does the sub-domain cover more important features than other sub-domains?
- **Development productivity:** how much effort will be saved for developers if the sub-domain is automated? This can be measured by the expected size of the artifact (such as the number of classes and lines of code) the sub-domain automation is supposed to generate, in average, for the developed games.
- **Development abstraction:** if implemented manually, how complex or error-prone are the artifacts supposed to be generated by the sub-domain automation? Examples of artifacts with a high error-prone and complexity rates are:
    - Code or configuration files that deals with too many literal values and constants, such as using string, character or enumeration types to define a map of tiles.
    - Code or configuration files that require lots of repetition but yet a few customizations that could be missed.
    - Code that requires application of design patterns.
    - Code that ensures a non-functional concern (such as performance or security) is satisfied.

This research is not aimed at providing a one-size-fits-all formula with precise weight values for each one of the criterion above, since their relevance may vary according to the software product line. However, we do suggest that all of such items are taken into account in order to prioritize sub-domains for automation. A relevant point that should be emphasized is that the prioritization is not final due to, once again, the iterative nature of the process. As a reference architecture is evolved (next section), more is known about the low-level details of the domain. Such a new knowledge impacts many of the criteria above, such as development productivity and abstraction, which should be revisited.

4.4 Bridging Game Domain Analysis to Application Core Assets

Before the implementation of domain-specific languages and generators can happen, knowledge should be gathered and compiled not only from the problem domain (top-down)

but also from the solution domain (bottom-up), resulting in a better understanding of the game domain also from an implementation (source code) perspective.

According to the Domain Engineering literature [Neighbors, 1980] [Czarnecki & Eisenecker, 2000] [Almeida 2007], the activities presented in this section and Section 4.5 belong to the Domain Design and Domain Implementation phases. Their definitions follow below:

- The goal of Domain Design is to produce a domain-specific architecture, defining its main elements and their interconnections [Bosch, 2000]. Focused on supporting commonality and variability to improve reuse, the designed architecture must not only predict the variation points, but also effectively provide the required support for their implementation [Bass et al., 2003]. In the ideal scenario, features defined in the feature model can be used to parameterize domain architectures and components [Almeida et al., 2006], making components to be developed almost free of design decisions by putting the features in the components as instantiation parameters.
- Domain Implementation is aimed at providing the detailed design, implementation and documentation of reusable software assets, based on the domain-specific software architecture [Almeida, 2007]. The Domain Implementation provisions and packages two types of assets [Lenz & Wienands, 2006]: *application core assets* (such as components, frameworks and prototypical applications) are building blocks for product line members, while *development core assets* (such as domain-specific languages and automated guidance) are integrated into a highly customized development environment to provide guidance, automation and abstraction to the product development.

The iterative "edge-center" nature of the proposed approach requires highlighting, customizing or introducing new Domain Design and Domain Implementation activities that directly relate to the Domain Analysis activities previously presented. Therefore, although this section and Section 4.5 have no ambition to define a comprehensive and systematic process for Game Domain Design nor Game Domain Implementation, they continue and complement the Domain Analysis discussion carried out so far, now in the design and implementation spaces, in order to culminate with the realization of application core assets for game product lines.

4.4.1 Toward a Domain-Specific Game Architecture

Some Game Domain Analysis activities, focused on analyzing the game domain and especially on the identification of sub-domain candidates for automation, touch lower-level con-

cepts such as development productivity and abstraction. That characterizes a transition from the top-down Domain Analysis to the bottom-up construction of a Domain-Specific Game Architecture.

According to Tracz [1995], a Domain-Specific Software Architecture (DSSA) is defined as an assemblage of software components, specialized for a particular type of task (domain), generalized for effective use across that domain, composed in a standardized structure effective for building successful applications. Gomaa [2005] states that developing a product line architecture is the most promising approach for architecture reuse, since it explicitly captures the commonality and variability in the family of systems that constitutes the product line. It is out of the scope of this research to define guidelines and processes of how to create a DSSA, as other works already address the particularities of DSSA design [Almeida, 2007]. This work does have interest, however, in how a DSSA for game domains can be conceived in order to support Model-Driven Development.

From experience in game development projects and based on the state-of-the-art design and implementation techniques of the game industry, we argue that a Domain-Specific Game Architecture (or just “reference game architecture”) should be built from the composition and adaptation of game engine(s) chosen and/or used for the game domain. In other words, game engines are the heart of a Domain-Specific Game Architecture. Reusable game components (Section 4.4.3 Creating Reusable Game Components) complement game engines in order to compose the final reference game architecture.

As opposed to the reference architecture proposed by Folmer [2007] for digital games, this research believes that there is no one-size-fits-all game architecture. In other words, reference game architectures have to be built, although not from scratch, for their target (sub)domains. If the game SPL aims at multiple platforms (e.g., console and PC), languages (e.g., C# and Objective-C), APIs (e.g., XNA and DirectX) or other variations in the target runtime environment, then there is a good chance that different game engines will be consumed by the generated games, therefore more than one domain-specific game architecture can emerge as a result. This may add a considerable complexity to the SPL, since its assets (especially code generators) will have to take such target platform variations into account.

The reference game architecture is also built from analyzing, refactoring and abstracting sample implementations in the game domain. Ideally, the sample implementations must cover examples of all possible common and variable aspects of the domain. Guidelines for that are provided by Lucrédio, Fortes, Furtado et al. [2009]. Typically, organizations that intend to establish game product lines have already implemented game instances in the past. Such instances may have already been consuming game engines, or otherwise were devel-

oped using one-off development processes with little or no reuse at all. In the latter case, the domain architect has to refactor the commonalities of the game instances toward creating (or reusing) a game engine. Game engine design and implementation is already discussed in the literature [Madeira, 2003] [Rocha, 2003] [Rollings & Morris, 2000] [Fan et al., 1996], therefore it is also out of the scope of this research.

If no sample implementations are available, the organization should employ a reactive approach [Krueger, 2001] for the game SPL, in which the SPL assets (including game engines and the final reference game architecture) are gradually built and incremented as domain instances are developed. Starter kits such as the XNA ones[28] are also a good resource, especially because they already encapsulate common knowledge from previously implemented instances, leaving only the customization part to game developers.

4.4.2 Promoting Game Engines to Domain Frameworks

In the first iterations, the Domain-Specific Game Architecture, which is based on game engines, very likely does not provide yet a seamless interface through which sub-domains' automation can be implemented or plugged-in. For instance, Dobbe [2007] had to add a better extensibility support and an event-driven paradigm to his Cannibal Game Engine before he was able to integrate it with his proprietary DSL.

In fact, the Domain-Specific Development literature [Cook et al., 2007] argues that mandatory sub-domain features identified during Domain Analysis should be encapsulated in a reusable component, which is called a **domain framework**, i.e., a framework created specifically for the domain. The domain framework is consumed by artifacts generated from diagrams used to model the variability of the sub-domain features. Lucrédio [2009] mention that most Domain Engineering approaches do not give enough attention to the process of designing an architecture that is well suited for MDD, but rely on the identification of variability as described in terms of features instead. As a result, this research suggests that game domain analysts and architects should primarily focus on promoting game engines (and their encompassing reference game architecture) into a reusable domain framework, instead of implementing other types of components or APIs.

Nevertheless, in the context of MDD and code generation, there are no guarantees that a game engine interface is in a suitable state to be easily consumed by generated code, as illustrated in Figure 23. An adapter layer is typically required to complement the game engine, in order to make it to be more easily consumed by generated code. As a result, this moves complexity away from code generators, which turns out to be extremely important

[28] http://creators.xna.com/en-US/education/starterkits/

since code generators are typically more difficult to maintain than a framework. In summary, the adapter layer promotes the game engine to a domain framework.

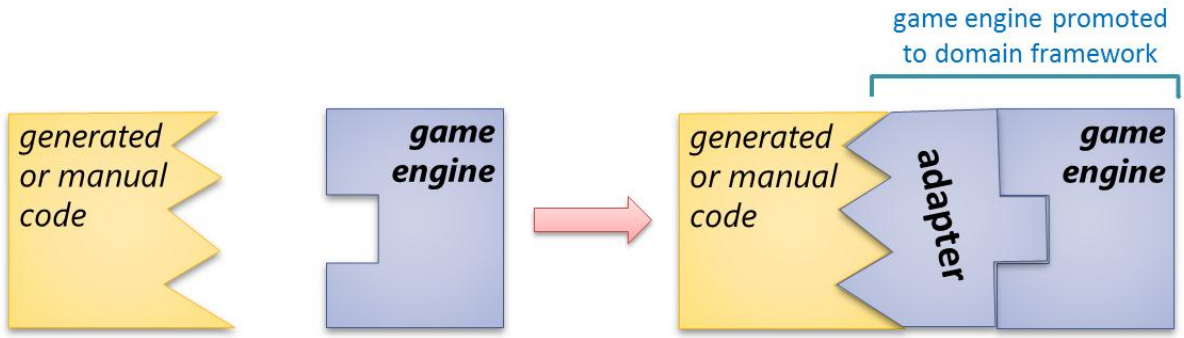


Figure 23 – Promoting game engines to domain frameworks

Turning a game engine into a SPL asset, however, may not be a straightforward task. Independent on whether the game SPL designers decide for implementing a game engine from scratch for the SPL or for promoting an already existing one to a domain framework, the following high-level questions arise and should be taken into account:

- Does the domain framework support the variability space of the game domain? According to Bass et al. [2003], the designed architecture must allow the construction of applications by predicting the existence of variation points and effectively supporting its implementation.
- Is the domain framework suitable for *framework completion*? In other words, the game engine should expose an interface which is expressive and concise enough so that code that consumes (configures) it can be easily generated by Model-Driven Development techniques (Figure 24). The goal here is to avoid complex code generators by providing more configurable and parameterizable domain frameworks.
- Does the domain framework support custom developer-added code, which is then combined and compiled with the SPL-generated code? This is important since the features of digital games typically present a high variability, which can be hard to express in models but could still be provided by the developer as an extension to the SPL.
- Can more specific frameworks be built atop the domain framework? This can provide more productivity and abstraction to the generation of features belonging to more specific sub-domains.
- Can the domain framework be integrated or even composed with other domain frameworks? This enables the common product code to have multiple configuration points, each one approached by a distinct model (for example,

one model for screen transition, other for entity state machine manipulation, other for physics modeling, etc.).

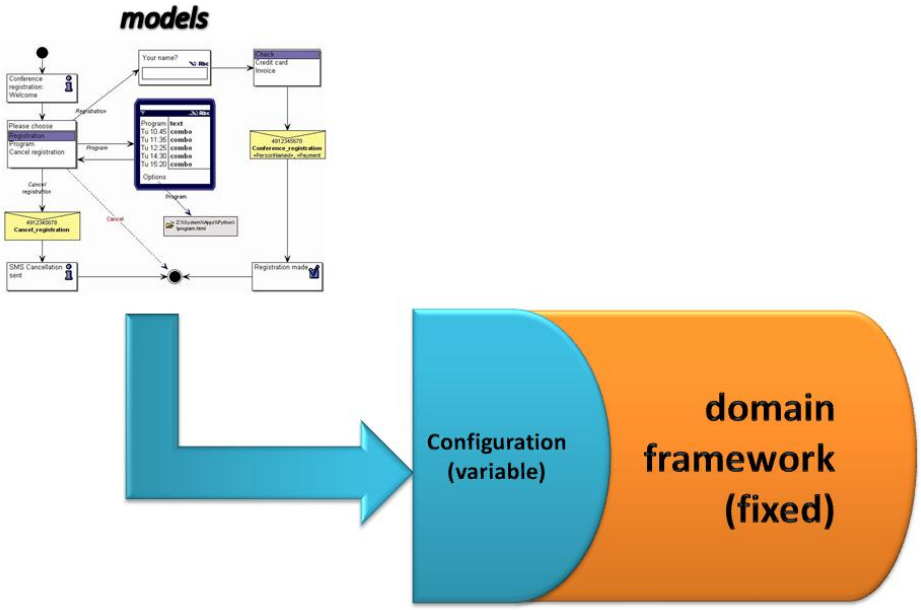


Figure 24 – Framework completion

In the lower (implementation) level, promoting game engines to domain frameworks involves refining the design of game engines in order to decompose them into well-defined modules [Parnas, 1972] [Almeida, 2007]. Game engines are already modular by definition, however further module decomposition might be necessary to provide a better mapping to sub-domain candidates for automation. Similarly, the interface of game engines might be refined in order to provide “variability hooks”, which can be more easily consumed by generated code. Anastasopoulos & Gacek [2001] describe multiple variability techniques. Refactoring game samples into a common yet customizable reference game architecture gives hints on the contexts where a technique fits better than others. For example, parameterization is a straightforward technique and can be used to configure simple properties of the domain such as the screen resolution. Inheritance is a great mechanism to make the generated code to specialize more abstract concepts, such as entities and screens, defined in the domain framework. Conditional compilation whose flags’ values are generated by models can be used to enable or disable behaviors that belong to a related context (e.g., compilation for console vs. compilation for PC) but are scattered across the source code. Other techniques include aggregation/delegation, overloading, properties, dynamic class loading, static libraries, dynamic link libraries, frames, reflection, aspect-oriented programming and design patterns.

Regardless of the approach used to promote the game engine(s) to a domain framework, some domain framework validation should be performed. In such a task, the sample

implementations should be updated in order to make them to consume the game engine, now promoted to a domain framework. Hence, such sample implementations now become **reference implementations**. Ideally, but constrained by the available resources, a representative set of games containing the sub-domain features should be implemented using the promoted game engine. If the game engine as a domain framework provides a smooth framework completion experience to implement the games, then it is suitable for the domain. On the other hand, a possible outcome of such a validation task is to refine the adapter layer so that more confidence is acquired on casting the game engine as a domain framework.

A game engine can also come with supporting tools or even a more complete development kit. Hence, the Domain Design and Implementation phases should also take such tools into account, in order to evaluate if they can also be incorporated as SPL assets and integrated into the game SPL schema. Finally, since the whole Domain Engineering process is iterative, the requirements for the domain framework and its Domain Design can be further refined in subsequent iterations. On this topic, we suggest the use of the Attribute-Driven Design (ADD) method [Bass et al., 2003], which promotes the successive refinement of the domain in two dimensions:

- From common to variable points: first, only the common points are considered in the design. Next, at each iteration, a variation point is included in the design;
- From module to sub-module: in this dimension, the domain framework is successively divided in modules and sub-modules, until a satisfactory level of details is obtained.

At each refinement iteration, the domain architect and domain implementer should create new or refine previous reference implementations in order to validate the game engine as a domain framework.

4.4.3 Creating Reusable Game Components

While game engines are the heart of a Domain-Specific Game Architecture, they are not the only implementation mechanism to realize Domain-Specific Game Development. Some functionalities of the domain may require more fine-grained implementation artifacts, which are not built-in elements of the game engine. That can be realized by **reusable game components**, which can co-exist with (and hopefully be pluggable into) game engines, toward establishing an enriched and more comprehensive domain framework.

Sametinger [1997] defines reusable software components as “self-contained, clearly identifiable artifacts that describe and/or perform specific functions and have clear interfaces, appropriate documentation and a defined reuse status.” The typical definition of component-

based development involves the selection of components from an in-house library or the marketplace to build products [Nascimento, 2008]. Although the products in Software Product Lines certainly are composed of components, these components are all specified by the product line architecture. Moreover, the components are assembled in a prescribed way, which includes exercising built-in variability mechanisms in the components to put them in use in specific products.

A game component provides support for a well-defined (and many times simple) game-related task or behavior. It can belong to a horizontal domain, such as an on-screen keyboard (Figure 25), or a vertical domain, such as a bi-dimensional radar for arcade games or a 3D camera movement behavior for first-person shooters, such as the “CNN camera view” feature that is launched when a main character is running in a first-person shooter game like Gears of War. Having a visual representation is not required for a game component. For instance, a game component can be a special timer or a more refined collision detection algorithm that can be applied to two entities.

Figure 25 – On-screen keyboard is a reusable game component

Conceiving reusable game components for a game product line arises from the need of satisfying the automation of a sub-domain that otherwise cannot be mapped to a game engine. Game components complement or provide a higher level of abstraction to functionalities provided by game engines. In short, reusable game components can fill in the gaps when game engines are not able to provide the desired automation for a sub-domain. Together, both contribute to promote the Domain-Specific Game Architecture (game engine + components + adapters) into a domain framework, therefore compliant with Model-Driven Development.

Once a game engine and reusable game components are properly combined toward the first version of the domain framework, the design and implementation of development

core assets, such as domain-specific languages, can finally start. DSLs, especially visual ones, are key to the process of sub-domain automation since they improve the way in which the domain framework is configured. Via the DSL's code generator, a model (visual DSL diagram) is transformed into the code (or a XML file) that consumes and configures the domain framework. Figure 26 is an updated version of Figure 24, explicitly calling out that since the domain framework is composed by multiple artifacts (game engines and reusable game components), it can be configured from multiple sources.

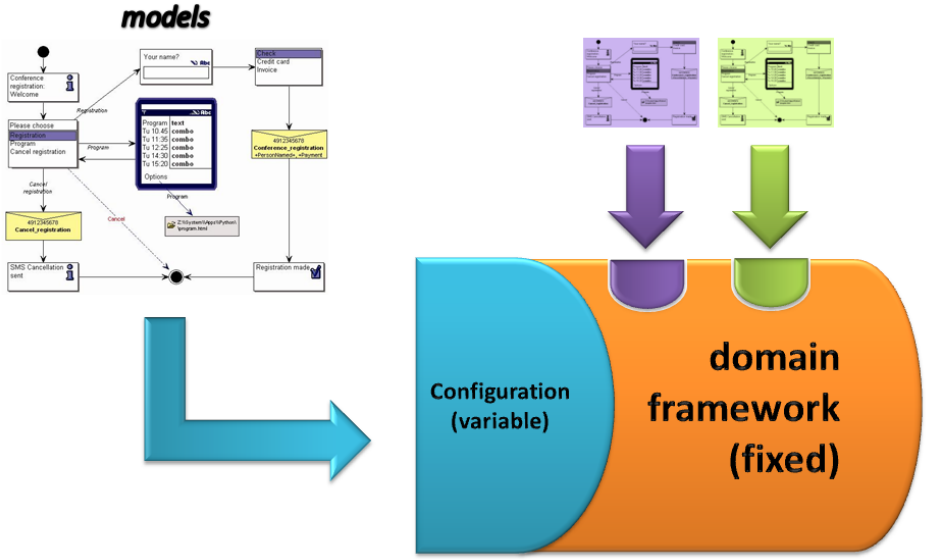


Figure 26 – Game domain framework configured from multiple sources

Figure 26 also reveals that the final reference architecture must be prepared for the existence of multiple sub-domains and possibly multiple DSLs. Lucrédio [2009] suggests the definition of a single meta-model for multiple sub-domains, and then the development of different concrete syntaxes for each one of them, so that they can integrate well but still have different views. We actually consider that if the sub-domains are too distinct, an alternative consists in the creation of one meta-model per DSL, so that each sub-domain is approached in a more modular and concise way. Such an alternative might also be more viable to enable the reuse of a DSL across SPLs (Section 4.6). Finally, the resulting DSLs can always be empowered with cross-DSL integration, allowing concepts in a language to transcend to other DSLs.

4.5 Bridging Game Domain Analysis to Development Core Assets

Application core assets, such as game engines and reusable game components, are building blocks that become part of the final products generated by the factory. On the other hand, *development* core assets, such as DSLs and generators, are those which contribute to the

product line pipeline in order to aid the development process, being integrated into a highly customized development environment to provide guidance, automation and abstraction to the product development.

DSLs are capable of extending the variability space not covered by most Domain Analysis approaches (such as feature modeling [Kang et al., 1990]), and at the same time serve as input to a generator or product configurator. However, DSL development is considered a science itself [Czarnecki & Eisenecker, 2000], given its complexity. Besides, it is not a very predictable process, as it requires a high degree of creativity [Völter, 2003]. We believe that the "edge-center" approach presented in this research contributes to tackle both challenges, complexity and creativity. At a given iteration, the sub-domain with the highest priority is chosen and mapped against the reference architecture. The type of variability for such domain is identified, ranging from routine configuration to creative construction. Such variability assessment on the reference architecture and on the feature knowledge enables the domain and language experts to conceive DSLs and generators for the chosen sub-domain, finalizing one iteration. Next iterations, based on other feature sets, can not only lead to new DSLs and generators, but also modify the ones conceived by previous iterations, as well as realize cross-subdomain relationships.

The steps described in the following subsections are suggested to realize the design and implementation of domain-specific languages based on the proposed edge-center approach. Some of such steps were defined in a joint research partnership [Lucrédio, Fortes, Furtado et al., 2009]. The discussion is instantiated to the digital games development domain when applicable.

4.5.1 Characterizing Sub-domain Variability

Depending on the range of variation required by a sub-domain and supported by the implementation, the structure of a DSL can be more simple or complex. The spectrum ranges from **routine configuration** (where simpler, tree-like DSLs, such as wizards or feature-based configuration, are used to select a subset of features when configuring a product) to **creative construction** (where complex, graph-like DSLs, such as programs and models, are created using textual or visual languages) [Völter & Groher, 2007] [Czarnecki, 2005]. This spectrum is illustrated by Figure 27.

For each sub-domain, a choice between routine configuration and creative construction (or somewhere in between) must be done, so that a proper DSL and its corresponding transformations can be implemented. In order to situate each sub-domain within the variability spectrum, the domain expert's role is very important, but some techniques can help. One of them is to look for **feature configurations that do not change between existing appli-**

cations. If a feature represents a variation point, its configuration must change in some way when different applications vary in that point. For example, if a game is played in full screen mode, and a second game is played in windowed mode, the feature configuration for these applications will be different. This indicates that the variability can be represented as features. However, if two applications differ at some point, but the features configurations that describe that point are the same, this might indicate that there is some variability that cannot be represented as features, and maybe a DSL is needed.

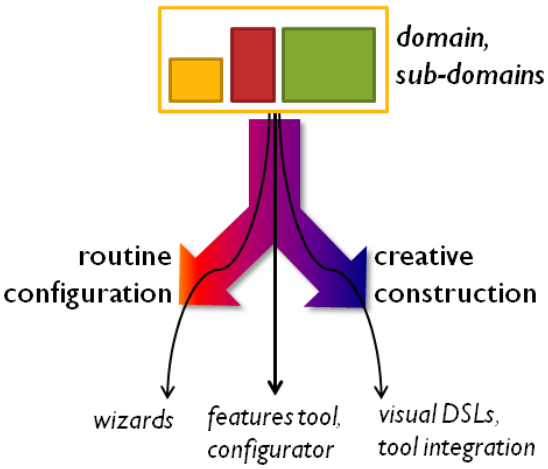


Figure 27 – Variability spectrum: from routine configuration to creative construction

Alternatively, SPL designers can look for **state machines.** Many sub-domains can be represented as state machines, such as the screens of a game or a game entity's behavior (for instance, walking → running → jumping → flying). If this is the case, this sub-domain will probably require a DSL (state machine) for its variability. Likewise, **hierarchical and containment structures** should also be sought after. Part-of relationships are commonly present in a domain. Although they can be normally represented in the feature model, some part-of relationships may require extra information. A classic example is the contents of a game screen. Although a feature mode can associate a screen with entity instances, heads-up displays and other elements, it does not capture all the information such as the location of such elements, their sizes, colors, and associated events. In such cases, a dedicated DSL is probably needed.

The output of this activity is a characterization of the variability type that is inherent to each sub-domain. Most importantly, this activity identifies which sub-domains will require a dedicated DSL.

4.5.2 Deciding upon MDD Development

For some mature sub-domains, languages and/or tools may already exist. For example, in a first-person shooter domain, tools for modeling tridimensional scenarios (game maps) and

saving their definitions in standard formats, to be consumed by game engines, can already be available. Other sub-domains may count on already existing languages, but no tools or code generators. In the worst case, there may not be anything at all. In this activity, the goal is to identify which sub-domains require further development in terms of DSL and transformations, and how such a MDD support should look like.

For each sub-domain, the existing languages and tools are analyzed. The goals of this task are twofold. First, to checking whether there are any available tools that could be reused, even if that involves some adaptation as it was already discussed in Section 4.4.2 for game engines. Second, to gather knowledge on the state-of-the art tools and usability for tasks related to the sub-domain (for example, creating a game entity animation from a set of pictures). Such a knowledge can be leveraged when creating the DSL concrete syntax (Section 4.5.3.2) and implementing IDE integration (Section 4.5.5), toward providing a better developer experience to game developers.

Languages and tools can be analyzed using human-computer interaction techniques such as task analysis [Preece et al., 1994]. A task analysis is formally defined as a detailed description of the goals an end-user should reach when using a system, as well as the activities and tasks required to fulfill such goals, with the intention to create a user experience that is the most intuitive and requires a minimum amount of assumptions and interactions. Task analyses should be observed from a real end-user experience, preferably from experts on the domain and on the tools under analysis.

In the sub-domains where variability is characterized as simple routine configuration, a feature modeling tool or a product configurator, such as Pure::Consul [Beuche & Spinczyk, 2003], may be sufficient to represent the variability and to perform some code generation and configuration. However, domain and language experts might face some technical limitations on such tools, such as their lack of integration with a code generation engine. For such cases, a new DSL for feature-based variability may be needed. For the other types of variability, characterized as creative construction, a DSL and code generators will probably be needed.

4.5.3 Defining DSLs and Supporting Assets

Depending on the type of the variability for each sub-domain, its DSL(s) will be more or less complex. In simpler routine configuration cases, the DSL may be composed of symbols that represent individual features, in order to indicate their presence/absence. Czarnecki et al. [2004] propose a method to derive a context-free grammar for a feature model. This method can be used to create a DSL that is capable of describing all kinds of variability that fits into a feature model. A parser generator or a DSL workbench can be used to easily create the language support, thus providing the necessary tools.

In more complex, open-ended variability cases, the DSL must define which concepts can be used, how they can relate to each other, and possible constraints that may exist. This can be done exclusively through a top-down approach. However, the DSL must also be able to produce models that serve as input to transformations and generators, what requires many details that are specific to the platform and chosen architecture. Thus, the bottom-up activities of the proposed approach is used to refine the initial DSL.

A DSL may be textual (programs) or visual (diagrams), and is normally composed of three elements: the abstract syntax, which defines the domain concepts, their relations and constraints; the concrete syntax, which provides a system (such as textual or graphical symbols) to represent the domain concepts in a concrete form; and the semantics, which defines the meaning of the language elements. This activity deals with the development of the abstract and concrete syntaxes, as well as a modeler that can be used to create instances (programs or diagrams) of that DSL.

In textual DSLs, the abstract and concrete syntaxes are normally represented as grammars and lexical rules. In visual DSLs, the abstract syntax corresponds to a meta-model [Guizzardi et al., 2002], while the concrete syntax corresponds to the visual aspect of the elements, normally using figures, icons, lines, arrows, among other notations. In order to correctly represent its variability, a sub-domain may require a system of concepts (abstract and concrete syntaxes) that is totally different from feature modeling, possibly requiring also a more complex modeler. But the feature model, even not being enough to identify DSL concepts [Tolvanen & Kelly, 2005], can serve as a starting point [Czarnecki, 2005], being later complemented with information from other artifacts, such as the domain architecture and experts' knowledge. The following subsections present four sub-steps are suggested in order to realize this activity.

4.5.3.1 Design the DSL Abstract Syntax

For visual domain-specific languages, the elements of an abstract syntax, such as concepts, relationships (including roles, cardinality, etc.), attributes and others, are typically structured as graph or tree, designed in a meta-modeling language. Examples of meta-modeling languages are the GOPRR (Graph Object Property Relationship and Role) language [Tolvanen, 1998] and the Microsoft DSL Tools meta-modeling language [Cook et al., 2007]. Kelly & Tolvanen [2008] suggests multiple sources for eliciting modeling concepts that will result in the elements of the DSL abstract syntax. Those include the (description of the) system family architecture, existing products and their features and manuals, available specifications, pattern catalogs, target environment and interfaces (libraries, frameworks, interfaces, etc.) and source code itself.

The most representative features of the sub-domain will start shaping the DSL's abstract syntax (the meta-model). For example, in a screen flow sub-domain, it is expected that screens and transitions (or links) are part of the initial syntax. Following that, features should be further analyzed to determine how they relate to each other and if additional DSL concepts are needed. Sub-features can be mapped to their own concepts, with additional "part-of" relations being used to represent the containment, or to properties of the containing concept, becoming meta-attributes in a meta-model. Concept attributes may require special domain types, rather than traditional "built-in" types such as *integer* and *string*. For example, optional and *"or"* sub-features [Lee et al., 2002] may be mapped as domain enumerations. On the other hand, a property can belong to custom types such as *Point* and *Size*. Related features may be connected by a new concept that describes the relation, the cardinalities, the participating concepts and their roles in the relation. Feature dependency analysis [Lee & Kang, 2004] may be used to identify such relations initially, but new ones may appear afterwards.

When a concept can be specialized by child concepts, which is commonly the case when the concept is derived from a feature with multiple OR alternatives, it is wise to include a "custom" child concept as one of the specialization options. Such custom concept contributes to the game SPL's flexibility by introducing an extensibility hook to the modeling languages, which enable developers to provide their own, custom implementation for the concept. The custom child concept typically has a method name as a property, or other means to associate it with a programming element that implements it. For example, the "screen transition trigger" concept can be specialized by child concepts such as a timer timeout, an entity position that was reached, an input button that was pressed or a custom action programmed by the game developer.

Sensible defaults should be provided for the concept properties (e.g., in platform games the scrolling direction property of a screen can be set to "Right" as default, while in bottom-up shooters "Up" makes more sense for its default value). The end goal is to capture the most common values for such properties and save time from DSL users, which will need to change the property values only for the less common scenarios. Likewise, some programming concepts, such as constants and variables, may be useful as DSLs concepts as well. For example, a DSL for defining game entities may enable developers to specify a constant value for the maximum amount of hit points (health) an entity can have. Once such a constant is exposed in the models, it can be referenced by other DSLs and model elements, such as event reactions (e.g., a health potion was collected by the main character), reducing the amount of literal values in the models and hence the duplicated work required to update them.

Some guidelines based on experience are also useful to evaluate the meta-model. For instance, two concepts of same DSL should not present a too similar list of properties. Otherwise, this is an indication that the concepts can be merged or a common, base concept can be introduced to refactor the similar properties of the specialized (child) concepts. Moreover, if too similar relationships (with the same cardinality, source or target concepts, roles, and/or properties) can be observed across the DSL concepts, this might be an indication that a common, base relationship can be introduced, or that once again the concepts belonging to the similar relationships may be merged or abstracted into a parent concept. Finally, if the abstract concepts tree of a visual DSL is too deep (i.e., it has too many levels from the root DSL concept to a concept leaf node), this might be an indication that the sub-domains were not properly partitioned. An abstract tree with more than three levels is not only difficult to manage but will also present challenges related to mapping the deeper concepts to the elements in the DSL's concrete syntax.

4.5.3.2 Define the DSL Concrete Syntax

This sub-step is not very difficult to derive. Features that are mapped to domain concepts may be represented as blocks. Pre-defined images can be used instead if they provide a richer representation for the feature. Vector graphics should be used when symbols need to be scaled Kelly & Tolvanen [2008]. More complex graphical representations can be used according to the nature of the sub-domain. For example, DSLs for sub-domains resembling activity diagrams, with roles and steps, can use "swim lanes" in order to graphically represent their features. Feature relationships may be represented as lines, which can vary in thickness, color and style (dotted, dashed, continuous, etc.).

Sub-features that can be mapped into boolean values may be represented as decorators in graphical shapes (such as an icon in front of a geometric shape) or in lines (such as a diamond on either end). On the other hand, sub-features that can be mapped to string values may be represented as textual decorators, such as a text inside the block or over a line. Kelly & Tolvanen [2008] mention that rectangular shapes are a better option when showing text is required, when compared to shapes such as ellipses, clouds, triangles, etc. If the sub-feature represent a lists of items, it may be mapped in the concrete syntax to compartments in blocks (such as UML attributes and methods, for example).

The "sensible defaults" defined for concepts and their properties should be respected and encouraged by the DSL's concrete syntax. Kelly & Tolvanen [2008] suggests additional guidance on the concrete syntax of a visual DSL, such as: show only relevant data and hide the details with techniques such as filters and property sheets, leverage scaffolding (hints and tips that stop being offered as users acquire experience), leverage colors, do no abuse

from especial effects such as shading, inspect corporate documentation standards to provide a consistent look-and-feel and consult with graphic designers if needed.

4.5.3.3 Manage Cross-DSL Integration

The need for cross-DSL integration arises when dealing with multiple, yet related sub-domains. Cross-DSL integration must be managed so that the developer can specify models in both languages using related concepts. For example, the concept of a game entity can be used by a DSL that defines collision between entities as well as by another DSL that links entity properties (such as number of lives and hit points) to elements of a heads-up display (HUD).

Cross-DSL integration can be achieved through name-based references or model bridges [Warmer & Kleppe, 2006], because they are simple to implement:

- Name references consist of a simple String attribute, in the referencing DSL, which points to the name of an element in the referenced DSL. Type and consistency checking has to be dealt with manually.
- Model bridges are not much more powerful, consisting of a proxy element in the referencing DSL that is an exact copy of an element in the referenced DSL. Type checking can be automated, but consistency checking still has to be performed manually.

Another possibility is to use a “model bus”. Such a solution provides a client-server infrastructure in which DSLs (clients) can register their shared (cross-DSL) concepts in a bus (the server), as well as consume concepts from the bus. A more powerful solution is presented by Hessellund et al., who propose the use of Prolog rules to establish inter-DSL relations [Hessellund et al., 2007], allowing higher order queries and facilitating consistency checking.

4.5.3.4 Build Domain-specific Modeler

As pointed out in Chapter 3, language workbenches contrast the early days of domain-specific modeling, where no tools were available to create domain-specific languages and support modeling with them in a cost effective manner [Tolvanen, 2005]. They make it easy to build tools that match the best of modern IDEs and make language oriented programming much easier to build and support, lowering the barriers that have made language oriented programming so awkward for so many.

Once the abstract and concrete syntaxes are defined, language workbenches are the technology of choice for implementing the domain-specific modeler, because they require little knowledge on language engineering to rapidly produce practical results. Examples include the Microsoft DSL Tools [Cook et al., 2007], MetaEdit+ [Metacase, 2009], GMF and

openArchitectureWare, among others. However, more complex DSLs may require a more active participation of language experts.

Again, the “sensible defaults” defined for concepts and their properties should be respected and encouraged, this time by the DSL’s modeler. For example, if the instances of a game domain always contain a start screen and a game over screen, then whenever a new diagram is created for modeling the game flow, it should include such screens by default as a starting point.

4.5.4 Developing Transformations and Refining DSLs

The output of the previous activities is one or more DSLs that allow developers to represent the different types of variability in each identified sub-domain, from simpler feature-based to more complex, open-ended variability. But the DSLs are not finished yet, since very likely there are still additional details that need to be included before they are suitable for being used with automated transformations. This is where the bottom-up aspect of the approach comes in hand again.

Design by-example techniques [Varró, 2006], [Wimmer et al., 2007], [Robbes & Lanza, 2008] can be used to produce transformations and possibly refine the initial DSLs. Instead of defining model-to-text transformations directly, the transformation expert starts with an example of how the generated code must look like, and then generalizes it to make it applicable in other contexts. This reduces the overhead by allowing the transformation expert to work on concrete instances of the problem, which is easier than having to figure out the details from a higher perspective [Robbes & Lanza, 2008].

The game sample implementations and their resulting reference architecture, which combines game engines and reusable game components toward a domain framework (Sections 4.4.2 and 4.4.3), are vital inputs for this activity. The transformation expert should investigate mappings from elements in the domain framework code to elements in the DSLs of the domain. During this inspection, he may discover new information that needs to be included in the original DSLs before they can be used as inputs to the transformations (for example, a game screen may introduce details like the start position of the main character). If this is the case, the domain implementer has to refine the DSLs (abstract and/or concrete syntaxes and associated tools) to include this new information. He may also realize that some modification or refactoring is needed in the reference implementation, in order to better support the variability. This can be done in parallel, avoiding the introduction of inconsistencies with other assets.

To find out DSL mappings to the reference architecture and implementations, we suggest identifying variability in the source code first. Differences in reference implementations can be a hint that variability is happening:

- **Addition of new functionality:** inclusion of new components, classes, functions, data structures or pieces of code. For example, a class is created for every game entity, or a new "if" branch is added to the *Activity* method of a screen class whenever a transition rule to other screen is introduced.
- **Removal of functionality:** removal of components, classes, functions, data structures or pieces of code. For example, code that sets the bounding box of a game entity is removed if the entity will not be colliding with any other.
- **Functionality substitution:** substitution of components, classes, functions, data structures or pieces of code. For example, code to set the size of the game window is replaced by code that runs the game in full screen mode.
- **Platform/environment:** when the inclusion of a variant requires modifications in the platform or execution environment. For example, if a game supports the concept of high-scores, then a persistent storage such as a database, the file system or even the cloud (web) should be enabled.

Once the DSL is refined, then the domain implementer produces a template-based code generator [Czarnecki et al, 2002], by migrating the code from the reference implementation to templates, annotating it with tags [Muszynski, 2005] and scriptlets that bind the code to the DSL. Techniques for code generation, like intermediate model-to-model transformations and patterns for code generation, can be used to produce better generators [Völter, 2003] [Völter & Bettin, 2004]. The annotated templates are responsible for implementing the variability logic, outputting the correct piece of code depending on the variants specified in the DSL. The use of DSL workbenches can greatly help the transformation expert. The result is a set of transformations that produce code that correctly implements the variability as specified in one or more DSLs.

Code generators are also responsible for the good quality of generated code, outputting code with documentation, using good names for identifiers, generating the correct indentation, and ensuring the generated code adheres to the required organizational standards, such as rules enforced by code analysis tools like StyleCop and FxCop. If such good qualities are already present in the reference implementation, then the template-based approach presented above should be able to keep them. Special care should be taken when generating identifiers (class names, method names, etc.) from the domain concept names entered by the game developer. For instance, the generators should filter out invalid characters (such

as spaces), ensure consistent casing is used across the board (lower vs. upper case), and so on.

Similarly, the code generator should keep and comply with the best Software Engineering practices (hopefully) employed in the Domain-Specific Game Architecture, such as modularization and reuse. The generated code should by no means allow violations of the model premises.

Generation scripts are many times hard to maintain due to different reasons, such as lack of tool support, scriptlet code interleaved with actual output game code and cross-language integration. This way, it is strongly recommended that the generators are designed to be as much light-weight and simple as possible, avoiding the generation of unnecessary code. Whenever possible, implementation complexities that arise from sub-domain variability should be refactored to the domain framework (game engine + reusable game components + adapters) and exposed in an easy way to be consumed by generators. Similarly, domain constraints are easier to implement as semantic validators targeted at the models, instead of the generated code. This discussion is typically revisited in future iterations of the *Promoting Game Engines to Domain Frameworks* step (Section 4.4.2).

Another challenge arises when the templates are ready, but changes are necessary. In such a case, the templates and the reference implementations should be kept consistent. This can happen either by changing the reference implementations first and propagating changes to the templates, or changing the templates first and replacing the reference implementations by regenerating them from the templates.

Some open-ended variability may be considered to be too complex to be covered by MDD assets. In such a case, the generated code should ensure that it supports extensibility mechanisms, such as partial classes or event handling, to allow developers to further customize and complement the generated code. The employment of the double-derived design pattern [Cook et al., 2007] helps to achieve such a goal. In such a pattern, instead of a single class, two classes are generated for a given domain concept. The base class contains all of the generated method definitions as virtual functions; the derived class contains no method definitions but is the one that is instantiated, which allows the user, in a partial class, to override any of the generated functions with their own version. The same techniques previously explored for supporting variability, such as inheritance, parameterization, overloading and others [Anastasopoulos & Gacek, 2001] can also be employed for extensibility purposes. Most importantly, the chosen technique should enable unexpected variability to be implemented in a non-intrusive way, i.e., developers are not required to modify the generated code at all. Similarly, the code generation process should not modify, delete or otherwise lose any work on the hand-written code or code generated by other DSLs.

When extending the generated code with manually added code, game developers typically program a behavior that is not provided as built-in by the game SPL. However, developers may want sometimes to call built-in behaviors from their extensions as well. For example, in a custom reaction for an entity timer event, developers may want to play a sound effect that depends on the entity's current state. Although switching on the current entity state is not a built-in behavior and requires custom code, playing sound effects might be supported as built-in by the game SPL. In order to call a built-in task in their manual code, such as playing a sound effect, developers should be able to easily reproduce the generated code approach for accomplishing the same task. In other words, the generated code should be as simple as possible, consuming the domain framework's interface in a concise way, without any "plumbing" and in the same intuitive way that developers are expected to do. For that to happen, game SPL designers may need again to move complexity away from generators to the domain framework, therefore enabling developers' manual code to be consistent with generated code, improving the extensibility experience.

4.5.5 Designing and Implementing IDE Integration

With the growing academic and industrial interest in Software Product Lines (SPL), one area demanding special attention is tool support development, which is a pre-requisite for widespread SPL practices adoption [Calheiros et al., 2007]. The true potential of a SPL can only be achieved if it integrates all modeling experiences into the same environment where other development tasks, such as coding and debugging, are performed.

Many language workbenches have the added benefit of already being hosted in an IDE (Integrated Development Environment). For example, the Microsoft DSL Tools [Cook et al., 2007] is integrated into the Visual Studio IDE, while other language workbenches are implemented as plug-ins to the Eclipse IDE. If no IDE integration is provided by default, we recommend that this goal is pursued by the integration expert.

The following integration areas are highlighted as being vital to provide an improved developer experience in the context of a SPL:

- **Project and build system:** an IDE project template defines a preliminary set of items for a new software project, such as starting files (classes, configuration files, etc.), default references to third-party libraries (such as .NET assemblies or Java .jar packages) and outputs (binary libraries, executables, etc.). They can be customizable and collect parameters from the developers through wizards or dialogs before being "unfolded". When using a game SPL, game developers should count on project templates. In the context of MDD, project templates should also include default instances of DSL diagrams,

which can already contain some information (concepts and relationships) as a starting state. Game SPL designers should also ensure that tasks such as compiling, linking, debugging and launching transformations can be done from within the IDE. Finally, special attention should be given to IDE operations such as renaming files and adding new resources (textures, sound assets, etc.), which may cause the modeling experience to become out of sync, especially in a cross-language environment. Some DSL workbenches and IDE object models provide "rules" that are triggered by such operations and to which developers can plug event handlers.

- **Custom property editors:** editing the properties of a DSL concept can be as simple as entering a string value. Many times, however, the values of a property might belong to a more complex type, or a simple type whose values are constrained, such as the buttons of a controller. In such cases, custom property editors should be provided for developers in order to guide them in specifying a property value. Those editors, typically linked from the IDE property grid, range from drop-down lists ("comboboxes") containing a set of possible values to more complex, GUI-based dialogs or wizards that present a richer graphical experience. An example for the latter case is a dialog presenting a tile-based world map, allowing the developer to select a tile (or map position) in which a game event should happen.
- **Semantic validators:** the developer experience can be dramatically improved if validation rules are defined and implemented for the domain-specific languages. Semantic validators can be defined to catch semantic errors at modeling-time, therefore guiding the diagram modeler (developer) on the creation of DSL instances [Cook et al., 2007]. For instance, a semantic validator can enforce that all game screens are reachable, or that the behavior of a game entity, modeled through a visual DSL, has a default starting behavior (state) always defined. Errors and warning found by the validators can be integrated into the IDE, being presented, for instance, in the same toolbars or tool windows responsible for showing compilation errors.
- **Contextual automated guidance:** as introduced in Section 3.5, contextual automated guidance are IDE extensions that empower product developers with suggestions on what activities to perform in a particular context, such as the automatic display of context-specific help, tutorials or checklists to guide product development and problem solving tasks. For example, when adding a custom event trigger to a game, developers can press the F1 key to make the

IDE show examples of custom triggers, code templates and concerns that they must be aware of. If developers want to, they can order the insertion of a custom trigger code template into the code under development. In the context of MDD and SPLs, contextual automated guidance can help with configuring feature models and complex tasks such as when creating DSLs diagrams or manually completing generated code. Technologies such as GMF's dashboard, openArchitectureWare's recipes and Microsoft's Guidance Automation Toolkit (GAT) are implementation possibilities for contextual automate guidance.

4.6 A Note on Cross-SPL Game Assets

Some "horizontal" game SPL assets can be re-used in other product lines. For example, the same domain-specific language used to model screen transition may be applied to an arcade game SPL as well as an adventure game SPL. Designing cross-factory assets, however, require the use of best practices such as making the asset flexible, extensible and configurable across game macro-domains, whose effort may not be worth. For example, instead of going through the complexity of making an arcade heads-up display graphical component to be compliant with a RPG heads-up display (which is definitively more elaborated), it may be wiser for game SPL designers to have specific components for each one of such domains.

4.7 Application Engineering

All the activities described so far belong to Domain Engineering, which is focused on analyzing systems in a domain and creating reference architectures and reusable components based on the analysis results. Application Engineering, on the other hand, is concerned on creating applications reusing these SPL assets, defining a product development process that guides the development of products through a software factory or product line. In the context of a game SPL, such a process should be defined by the game SPL designers in order to enable SPL users (game developers and designers) to fully exploit the capabilities of the SPL in the creation of digital games. Defining an Application Engineering process is out of the scope of this research. However, some pointers are presented below in order to provide more context to the proposed approach

Traditional product development processes start with requirement analysis. In the context of game SPLs, this actually translates into configuring the feature model for a specific game instance, besides entailing experimentation and prototyping activities that will shape unique characteristics for the instance. Breyer [2008] synthesizes a list of heuristics used to

evaluate game prototypes according to different variables such as interface, gameplay, narrative and game mechanic.

Following, a Game Design document should be instantiated for the game under development. Customizing the game domain vision (Section 4.2) for the specific game under development is a possibility. This task also contemplates other specific game development activities such as elaborating game characters, level design, sound concept, art concept and planning the publishing channels for the game. Personas, scenarios and immersion techniques can also be explored [Neves et al., 2008].

The software design and implementation should leverage the Domain-Specific Game Architecture (domain framework) and its customization by means of DSL diagrams, which will be used as input to generators. IDE integration, such as contextual automated guidance, play an important role in ensuring the game SPL assets are used effectively.

It is very important to continuously improve the game SPL assets and keep them versioned [Lenz & Wienands, 2006]. The feedback cycle, which is a combination of application development and the SPL's review and assessment, is key to evolve the product lines assets. This process of extracting knowledge from application development experience, also called mining or harvesting, is illustrated in Figure 28. Game SPL assets that should be targeted by the feedback process include the SPL scope, architecture, tooling and the application development process itself.

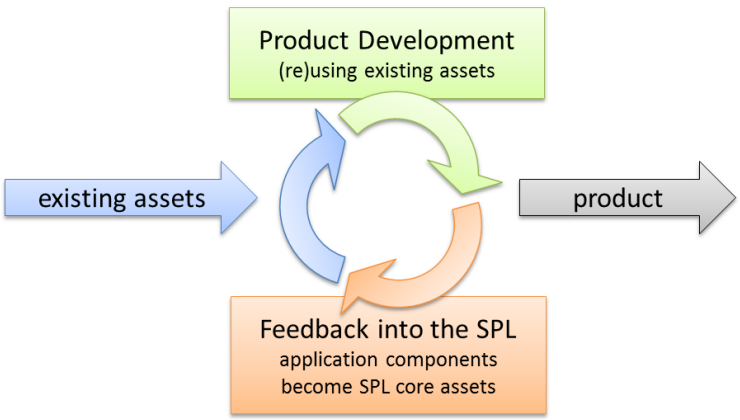


Figure 28 – Retrofitting feedback from developed games into the game SPL

Backward and forward compatibility is an important concern when evolving a game SPL. Not rarely, refined versions of the DSLs can break existing model instances. For such scenarios, we suggest that migration tools are developed in order to assist game developers and designers to move their models to the new versions of the DSLs.

As a final note, Araujo [2006] defines a game application process with roles, phases, activities, guidance and workflows. Although his focus is on casual games, much of his discussion can be scaled to digital games development in general.

4.8 Chapter Summary

After discussing the peculiarities of digital games when compared to the software in general, this chapter outlined an approach for Domain-Specific Game Development. Its focus is on activities that are relevant to Model-Driven Development in the context of digital games development, toward the creation of application and development core assets in a game SPL, such as domain-specific languages and generators.

The approach's activities were presented in four major groups. First, we discussed how game domains can be envisioned, having as major motivations the lack of precision and agreement on game genres, the desire to better bootstrap the domain scoping process and the need to have an upfront baseline for ensuring that the game domain's features and samples to be analyzed fit into the domain scope.

Next, guidance on analyzing game domain samples was presented, aimed at making the game SPL vision more concrete by identifying domain commonality and variability, as well as documenting them through feature models. Special implications of Domain Analysis due to the nature of digital games, such as experimenting and exploring game domain features, were discussed.

The relevance of identifying and prioritizing sub-domains for automation through Model-Driven Development was also presented. Following that, the Domain Analysis activities were bridged to the design and implementation of development core assets, in which game engines play a major role toward the creation of a domain framework. Then, the design and implementation of application core assets (such as domain-specific languages and generators) from the prioritized sub-domains was discussed. Comments were also provided for cross-SPL game assets and Application Engineering.

With Domain-Specific Game Development approach described, we switch our focus to evaluating its effectiveness. Chapter 5 discusses how such a task was accomplished, including evaluation techniques such as case studies and a controlled experiment, as well as the obtained results.

5. EVALUATION

This chapter presents the methodology employed to evaluate the suitability and effectiveness of the proposed Domain-Specific Game Development approach in the development of digital games, along with the obtained results. Its goal is to gather evidence on whether the approach can bridge the Game Domain Analysis to the development of core domain assets which are more streamlined and effective to the digital games development domain.

We employed a three-step evaluation process for our approach, detailed in the next sections. Section 5.1 investigates what are the good properties of domain-specific development and DSLs, the most relevant deliverables of the proposed approach. We openly evaluate whether and how the approach takes such properties into account.

Furthermore, we have been continuously developing a set of case studies with the proposed approach, whose purposes are twofold: exploratory and confirmatory [Easterbrook et al., 2007]. Section 5.2 presents exploratory case studies, typically used to drive the investigations that derive hypothesis and build theories. Section 5.3 presents a confirmatory case study, typically used to test existing theories.

Finally, taking into account that the major weakness of case studies is that the data collection and analysis is more open to interpretation and researcher bias, Section 5.4 describes the steps executed for an experimental study, such as its definition, design, instrumentation, threats, operation and analysis, which gathered very relevant data and feedback from software engineers in the industry on the proposed approach. The experiment is focused on measuring the benefits of an outcome of employing the approach (i.e., a game SPL instance) rather than the approach itself, which would be an order of magnitude more costly to evaluate, even empirically. Finally, Section 5.5 summarizes this chapter.

5.1 What Makes Good Domain-Specific Development Assets?

Several papers can be found on the advantages and disadvantages of domain-specific languages and related assets [van Deursen & Klint, 1998]. Section 3.4, for instance, presented a discussion on the many of the benefits (and drawbacks) of using a DSL, compiled from multiple sources. However, although such benefits are a consequence of properly designing and employing domain-specific development, it is not straightforward to directly infer from them what a “good” DSL or generator is constituted of.

Based on the multiple sources of the domain-specific development and SPL literature used throughout this research, interviews with some experienced domain-specific developers who work on leading tools in the area and on our personal experience, we compiled a list of the desirable characteristics (not necessarily metrics) of domain-specific development as-

sets. We then evaluated whether such good characteristics, or properties, are enforced or encouraged by the proposed Domain-Specific Game Development approach. The possible evaluation outcomes are **High** (following the process predictably results in the characteristic), **Medium** (following the process likely, but not certainly, results in the characteristic) or **Low** (there is no indication that following the process results in the characteristic). The result is presented in Table 6.

Despite of being informal, such an evaluation was very useful to identify points of strength and weakness of the proposed approach. For instance, while the promotion of game engines to domain frameworks aids many concerns related to code generation, it is clear that the approach lacks more details on areas such as testability and model-to-model transformation. The evaluation also establishes an important baseline upon which enhancements of the approach can be prioritized. In fact, we recommend that such "good properties" of Domain-Specific Development assets are taken into account in the very beginning of the creation of Domain Engineering processes, even for specific domains other than game development.

5.2 Exploratory Case Studies

A set of exploratory case studies was developed by Computer Science undergraduate students, under our guidance, in order to aid the investigations that shaped the proposed Domain-Specific Game Development approach and provide some "early and often" evaluation to it.

The *Commander Assembler* game SPL [Marques de Almeida, 2008] employed the approach toward the creation of Tactical Role-Playing Games (TRPGs), a domain that incorporates elements of traditional role-playing games and strategy games. The author of this game SPL justified the creation of a custom game engine from scratch by the lack of broadly available TRPG engines. The Commander Assembler game engine was then implemented as a domain framework and is consumed by a DSL used to model maps and entities of TRPGs (Figure 29). The screenshot of a TRPG sample generated by Command Assembler is presented in Figure 30.

Elegy [Azevedo et al., 2009] is a game SPL aimed at a broader set of RPG games. It has multiple languages and assets (Figure 31) for modeling RPG entities, maps, quests and other concepts. The domain framework consumed by the generated artifacts is the XNA's RPG Start Kit[29]. Figure 32 presents the screenshot of a sample game developed with the Elegy SPL.

[29] http://creators.xna.com/en-US/starterkit/roleplayinggame

Table 6 – Evaluation against desirable properties of domain-specific development assets

Desirable property of Domain-Specific Development assets	Approach evaluation against the property
Reuse of existing standards to avoid reinventing the wheel.	**High**. This is enforced by Domain Analysis tasks that are concerned with understanding the state-of-the-art in the domain, as well as DSL creation tasks that explore already existing automation evidence.
Early and often validation from end-users.	**Medium**. Although end-users are involved by multiple tasks, the approach does not detail user-centered tasks related to early validation and deeper interaction with them, especially for user interfaces.
Model partioning capabilities, for the sake of editing and processing scalability.	**Low**. Although some of the assets used in the approach handle such a concern, model partitioning capabilities are not mentioned as part of the process.
Asset evolution without breaking editors, processors and existing models.	**Medium**. Some rough notes are provided for backward and forward compatibility of the generated assets, but the approach lacks more details on those.
Powerful yet scoped (well-defined) expressiveness, representing a clear separation of concerns: only one aspect of the domain per DSL.	**High**. Enforced by the sub-domain breakdown and the iterative nature of the edge-center process.
Intuitive DSL notation that matches the domain concepts.	**High**. The solution domain is taken into account by multiple tasks of the process (vocabulary definition, feature modeling, etc.). Special concern is given to avoid DSLs and generators from becoming just an alternative representation of the source code.
Concise DSL syntax (limited concept tree depth, no duplication of concepts and relationships, sensible defaults)	**High**. Enforced by tasks related to the creation of the DSL syntax and its domain-specific modeler.
Strictly defined cross-DSL connection points, which are limited in number and dependencies.	**Medium**. The approach suggests cross-DSL techniques such as model bridges and model buses, but lacks details. Guidelines for coming up with and limiting connection points are not presented.

Desirable property of Domain-Specific Development assets (cont.)	Approach evaluation against the property
Model-to-model transformations to transform models into others and move complexity away from code generators.	**Low**. Model-to-model transformations are superficially mentioned, but not covered by the proposed approach.
Constraints to validate models with different severity levels (e.g., warnings, errors), which can be checked in the process workflow as early as possible.	**High**. Covered by tasks related to creating semantic validators.
Model library provided for reuse.	**High**. Covered by tasks related to promoting game engines to domain frameworks.
Transfer of complexity away from code generators.	**High**. Covered by tasks related to promoting game engines to domain frameworks and the creation of code generators.
Good quality of generated code (e.g., documentation, good names for identifiers, correct indentation, adherence to the required standards, best Software Engineering practices).	**High**. Covered by the tasks related to the creation of code generators and a Domain-Specific Game Architecture prior to the creation of code generators.
Code generation that does not allow violation of the model premises.	**High**. Covered by the tasks related to the creation of code generators.
No monolithic code generation: partitions and viewpoints can be processed separately and on-demand, one by one.	**Low**. Although the sub-domain breakdown avoids monolithic code generation, individually processing partitions and viewpoints on-demand is not explored.
Clear separation between generated and hand-written (extension) code.	**High**. Covered by tasks related to the creation of code generators and extensibility hooks for the game SPL.
Guidance or control of the developer efforts in complementing generated code with hand-written code.	**High**. Covered by tasks related to extensibility hooks and IDE integration.
Teamwork support such as versioning, tagging, branching, locking, comparing, merging and daily build.	**Low**. Such concerns were left out of the scope of the proposed approach.
Testability (e.g., unit tests to validate the generated code, test models that consumes all features of the language).	**Low**. Such concerns were left out of the scope of the proposed approach.

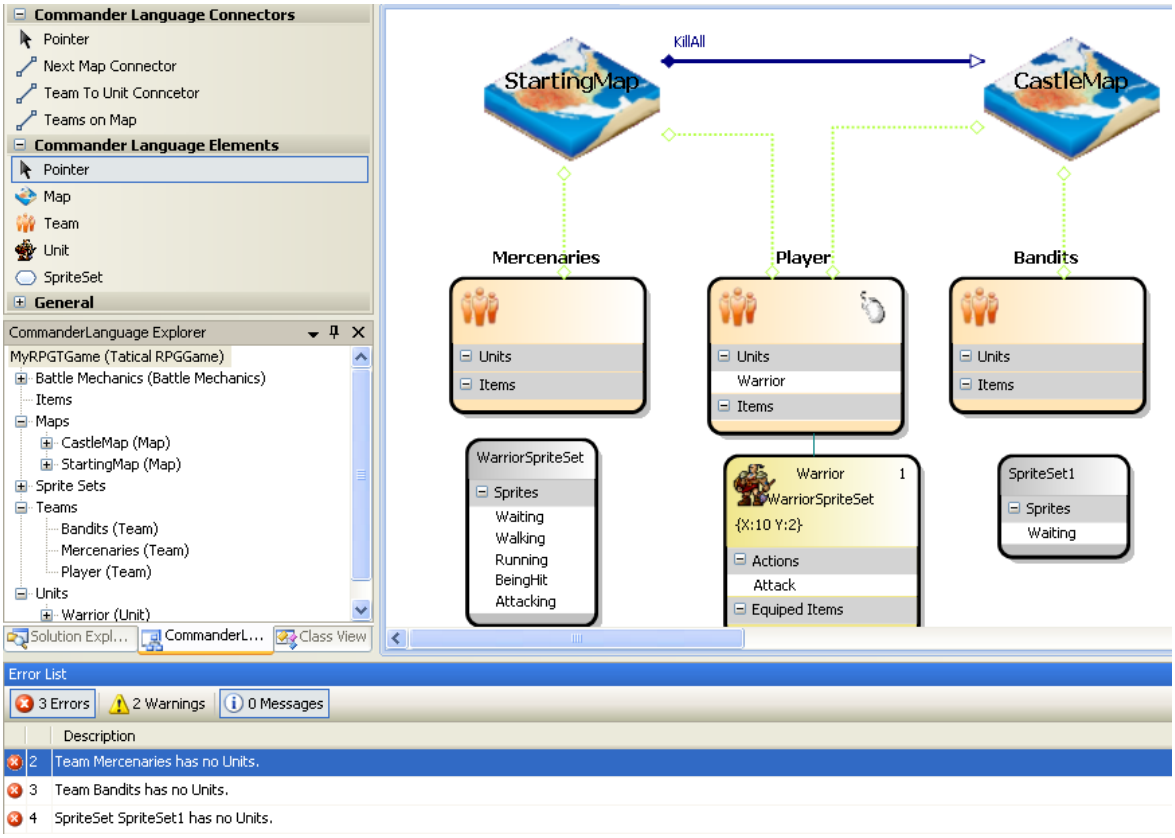


Figure 29 – Main DSL of the Commander Assembler game SPL

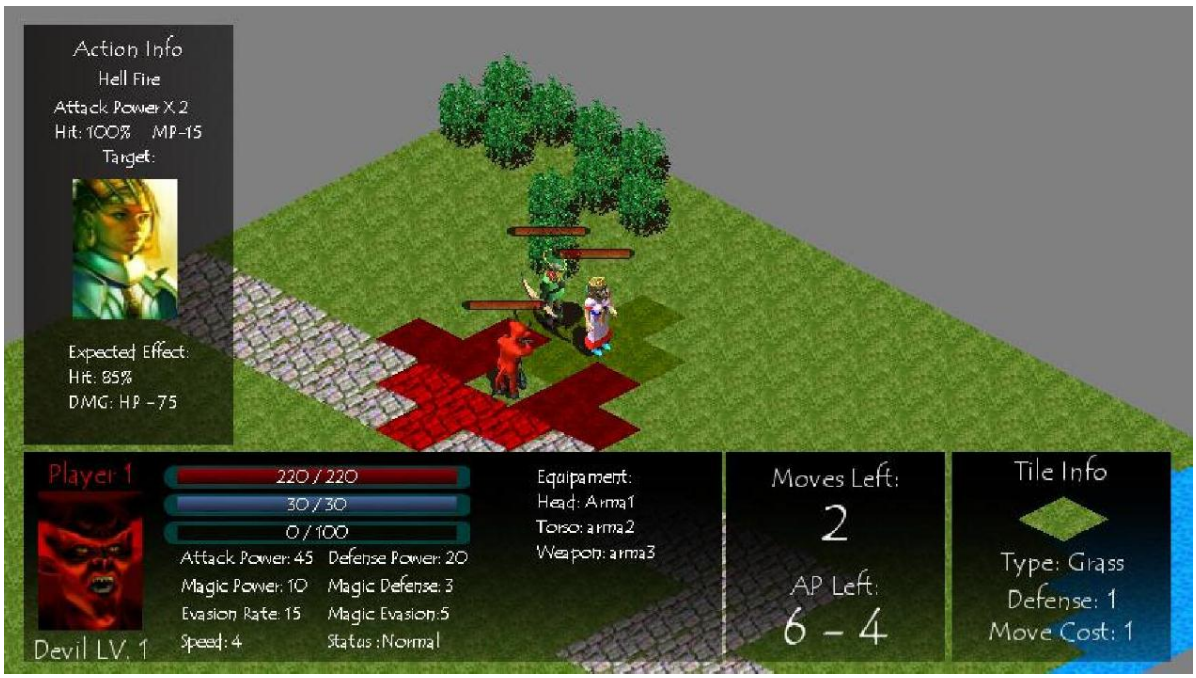


Figure 30 – Sample game developed with the Commander Assembler game SPL

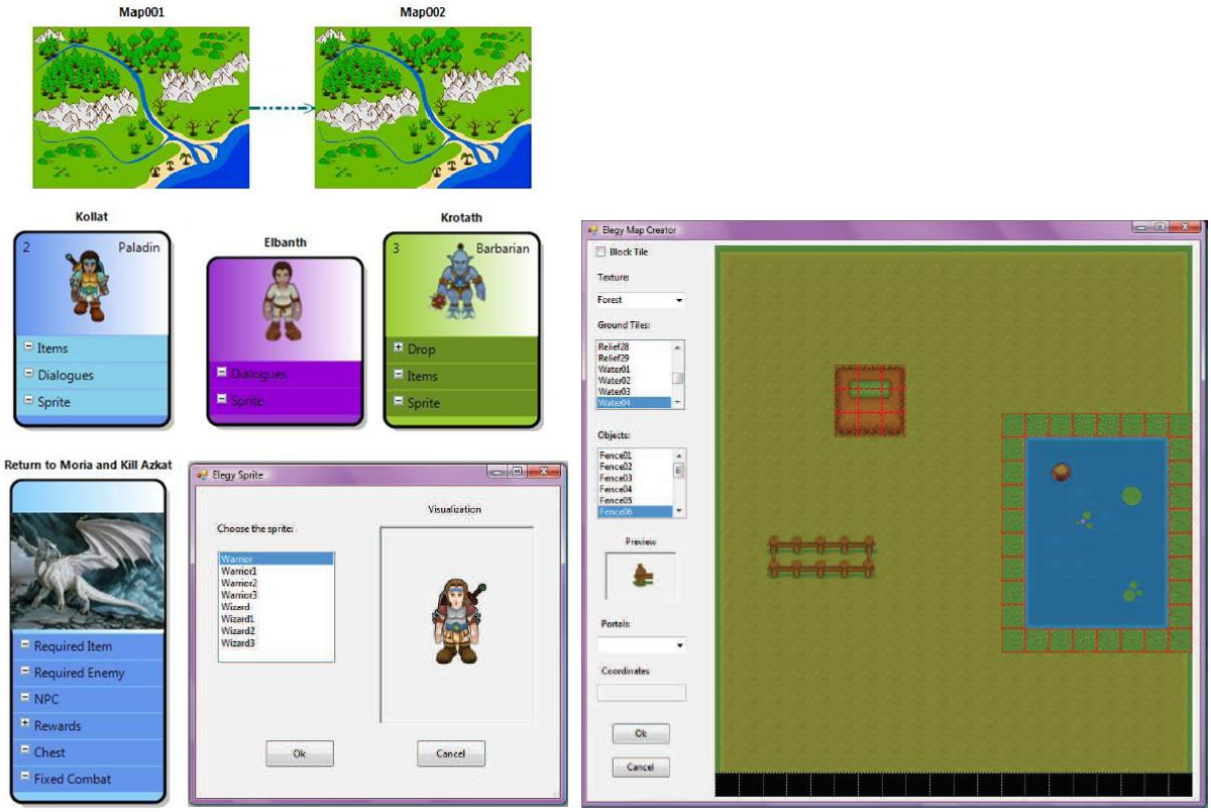


Figure 31 – Assets of the Elegy game SPL

Figure 32 – Sample game developed with the Elegy game SPL

Marinho [2010] partially employed the proposed approach to the creation of a game SPL targeted at multi-touch games. As a result, the author conceived a DSL entitled *Gesture Aggregation Language* (GAL). Such a DSL generates code targeted at Apple's Gesture Rec-

ognizers API[30], promoted to a domain framework through the “GAL Framework” adapter layer. Since GAL is a textual DSL (Figure 33), the author had to customize some of the proposed Domain-Specific Game Development activities in order to design and implement the DSL. Finally, in order to evaluate GAL, the author used an approach based on implementing and analyzing new versions of already existing games, such as the *Undead Attack! Pinball* game, presented in Figure 34.

```
context: Pinball
<no inputs>
gestures
  InvButton in SCREEN[1:2] with
    <no conditions>
    name : "LeftFlipper"
  end
  InvButton in SCREEN[2:2] with
    <no conditions>
    name : "RightFlipper"
  end
  InvButton in SCREEN[2:3][10:10] with
    <no conditions>
    name : "GateWeapon"
  end
```

Figure 33 – Sample code written in the Gesture Aggregation Language DSL

Figure 34 – Sample game re-implemented with the Gesture Aggregation Language

[30]http://developer.apple.com/library/ios/#documentation/EventHandling/Conceptual/EventHandlingiPhoneOS/GestureRecognizers/GestureRecognizers.html

In a parallel yet overlapping work, Araujo [2009] used some of the concepts presented in this research toward creating a casual games factory, entitled Play4Fun. His focus was not on domain-specific languages and frameworks, but on a detailed process for casual games development with roles, responsibilities, workflows, disciplines, phases, activities and work products.

5.2.1 SharpLudus Adventure Revisited

The *SharpLudus Adventure* game SPL [Furtado, 2006], developed as a spike solution for this research, can be considered as an early instance of the proposed Domain-Specific Game Development approach and provided a considerable amount of exploratory and evaluation data as well. This subsection revisits such a SPL with the goal of providing some retrospection on which of the original guidelines are still valid as proposed and which were improved to increase their efficiency and/or applicability.

SharpLudus Adventure was targeted at creating 2D adventure games, with rooms filled with enemies and items that had to be explored by a main character. SharpLudus' main deliverable was the SharpLudus Game Modeling Language (SLGML), presented in Figure 35, whose generated code consumed a custom engine developed on the top of a managed version of Microsoft's DirectX. A sample game developed with the SharpLudus Adventure SPL is presented in Figure 36.

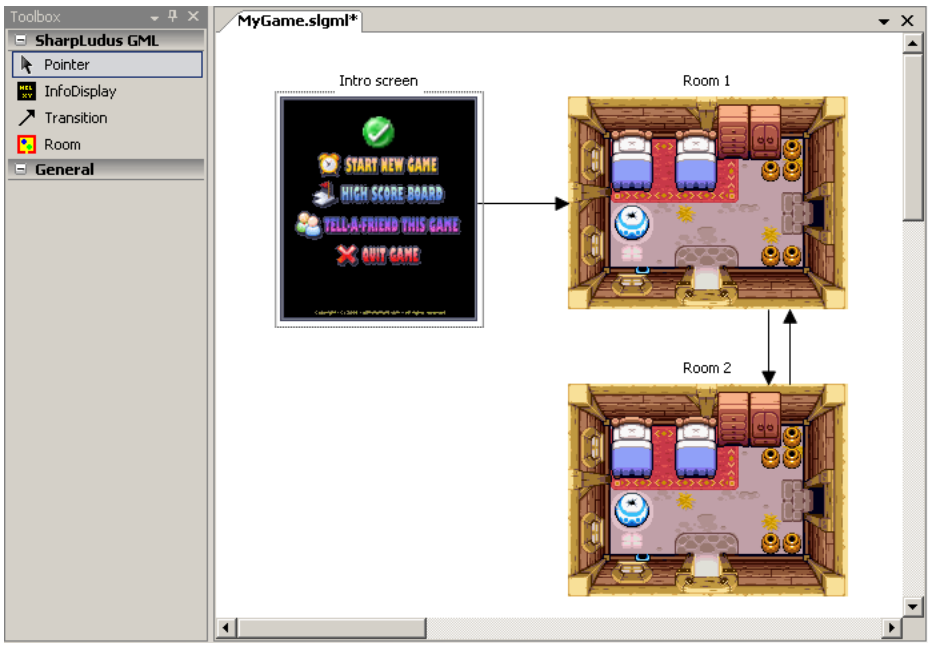


Figure 35 – SharpLudus Game Modeling Language (SLGML)

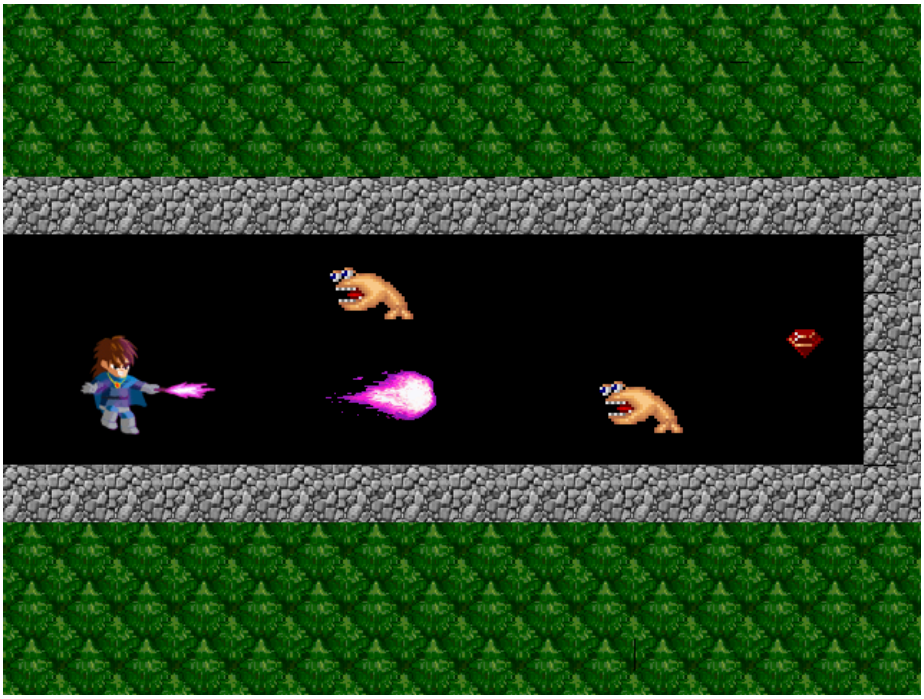


Figure 36 – *Ultimate Berzerk* adventure game, created with SharpLudus Adventure

Thinking in retrospect, many lessons were learned in the journey from the original SharpLudus Adventure project to the current Domain-Specific Game Development approach. From one hand, the original assumption that DSLs are underexplored in the context of the digital games development domain still seems to be true. In the lack of better metrics, game samples originally developed with SharpLudus reported the generation of dozens of classes and thousands of source lines of code after a couple of hours of modeling experience. As it will be described in the next session, the Domain-Specific Game Development approach results are also encouraging, reporting a more than five times faster development than approaches employing only game engines.

On the other hand, the original SharpLudus project and the current approach diverge in some relevant points. Obviously, the main difference is that while SharpLudus was an **instance** of a game SPL, Domain-Specific Game Development defines guidelines for the creation of game SPLs, being a more comprehensive and mature approach whose foundations were built from Domain Engineering and other software reuse concepts.

Already concerned with the problems entailed by the ambiguity of game genres, SharpLudus suggested that the target game domain had to be described by means of a "product line definition". The current approach evolved the concept by requiring the game SPL to be described by means of the expectations of *core game dimensions*, which are neither overly generic nor specific to a game domain (Subsection 4.2.2.1). Nonetheless, a more important difference is how such assets are used once created. While the product line definition (together with the domain vocabulary) was used by SharpLudus as a direct input to DSL

design, in Domain-Specific Game Development the core game dimension expectations, together with other assets such as the identified non-emotional requirements for the domain, end up as input for the creation of feature models. Such intermediate step makes the identification of the commonality and variability of the domain much more evident, and is key to identify and prioritize sub-domains, leading to more expressive and effective DSLs and generators. In fact, SharpLudus lacked a more structured Domain Analysis phase in which guidelines are provided for the selection and analysis of game samples.

Probably one of the most notable evolutions from SharpLudus to the current approach is the sub-domain breakdown employed by the latter. Although the SharpLudus Game Modeling Language (SLGML) is a *domain-specific* language, thinking in retrospect we concluded it is not atomic enough, but a bit bloated. It encompasses too many concepts that, despite of being related, could have been explored by SPLs in a much more effective way if separated. In short, SharpLudus lacks more specific, atomic yet integrated languages.

For instance, SLGML encompasses the concepts of audio, entities, events and game flow altogether. Having all of them in the same modeling canvas would provide a confusing user experience. As a result, the core of SLGML's concrete syntax focused on only one of such concepts: game flow (Figure 35). In other words, the modeling experience covered only a subset of the domain. The management of other concepts such as entities and events was performed in normal lists and dialogs (Figure 37), launched as custom property editors assigned to the properties of the domain's root concept (the "adventure game").

We believe this approach is not optimal. To start with, custom dialogs and lists built from standard UI controls do not typically provide the desired level of abstraction for a specific domain. In such an approach, the concrete syntax of the domain concepts gets mixed with concepts of the user interface API domain, such as buttons and list boxes. Likewise, in such an approach game developers and designers are more likely to deal with instances of the concepts in isolation, as Figure 37 shows: the entities of the game are described one by one in the list box, but interesting relationships between them, such as whether anything happens if they collide, are not described. The aforementioned problems can be mitigated by the creation of custom, refined UI controls. However, such a task may require a considerable amount of effort. In fact, the excessive creation of custom UI controls for modeling purposes seems to be a duplicated effort, considering that this is supposed to be the role of language workbenches (toolset through which DSLs and generators can be effectively implemented) [Fowler, 2005]. Finally, having all instances of a concept to be described in the same list may decrease the overall cohesion of the models. This was observed at least in two opportunities in SharpLudus. First, all animations of a game were defined together in the same list, but each set of animations was used only by a specific game entity, i.e., the sets had no relation

to each other. Hence, it makes more sense to have them managed from their respective entity instead of together. Second, game events were defined in a same list, but their triggers actually came from multiple sources (entity collision, screen timer, player input, etc.). Hence, the modeling experience could have been improved and made more cohesive if each event was managed from the source concept that triggers it.

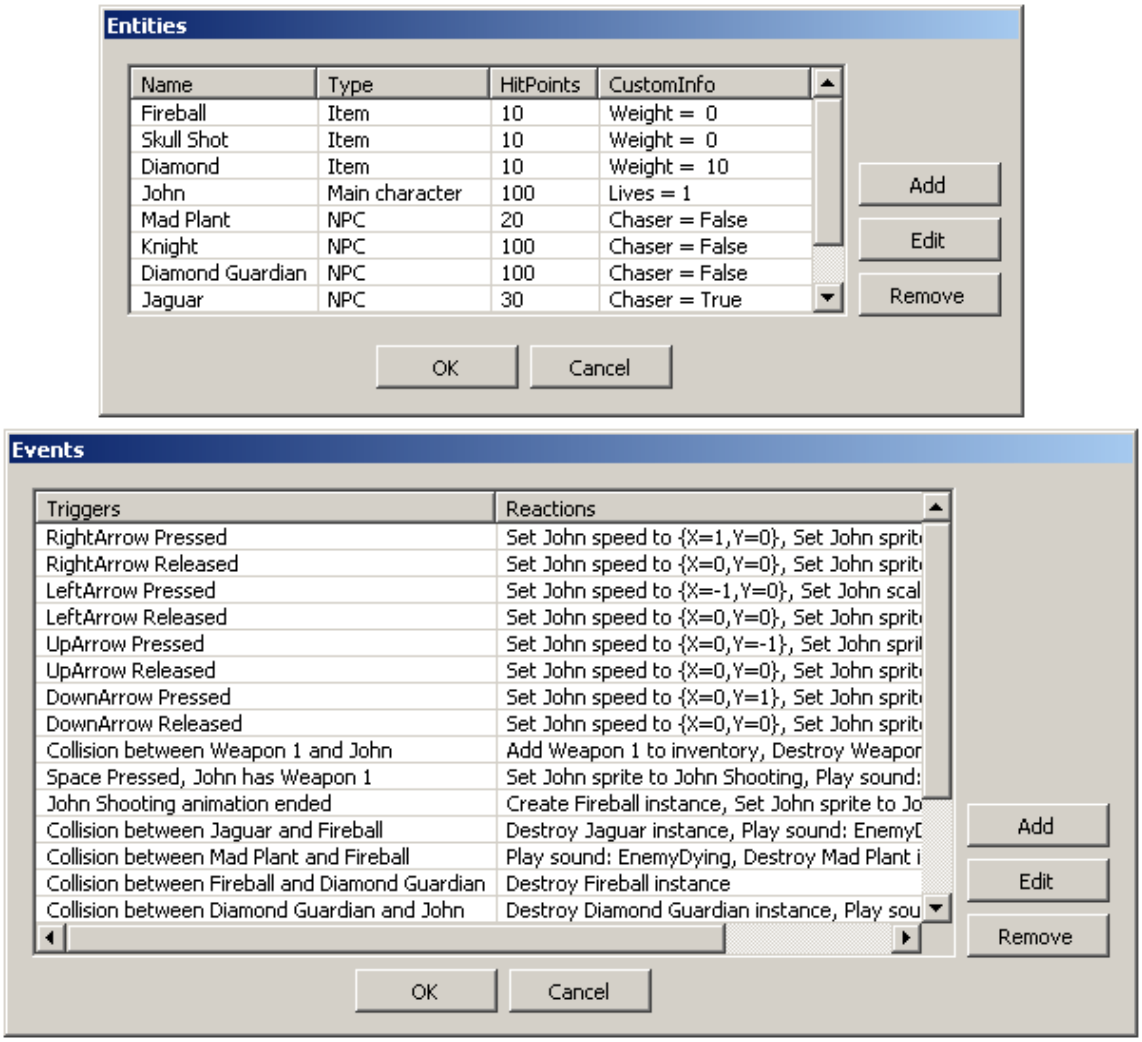


Figure 37 – SLGML concepts, managed through lists and dialogs

Learning from this experience, Domain-Specific Game Development advocates that the target game SPL domain should be broken down into atomic sub-domains. Examples of such sub-domains are the transition between game scenes or screens, entity or screen timers responsible for triggering events, the collision relationship between game entities and the possible graphical representations of menus and heads-up displays. Such sub-domains are too atomic to comprehensively define a game by themselves. On the contrary, the different features of a game fall under such atomic sub-domains, i.e., the game is the sum of the features distributed in the sub-domains.

As the target game SPL domain is broken down in sub-domains, the approach's edge-center spiral (Figure 13) focuses on addressing one prioritized sub-domain at a time. The sub-domain chosen for a given iteration has its feature model detailed, is mapped against existing source code from samples, has corresponding modules implemented in the domain-specific game architecture to support its commonality and variability, and ultimately leads to the creation of very specific, atomic DSLs and generators.

The Domain-Specific Game Development approach also provides guidance on how to characterize the variability of a sub-domain, which will determine the concrete syntax of its DSL(s). As previously mentioned in Chapter 4, the variability ranges from *routine configuration* to *creative construction* (Subsection 4.5.1). Similarly, techniques for developing transformations are more detailed in Domain-Specific Game Development, which elaborates how template-based code generators can be achieved by migrating source code from the reference implementation to templates, annotating it with tags and scriptlets that bind the code to the DSL.

As a result of the sub-domain breakdown, some benefits can be observed, such as **more expressive DSLs and generators**, since each is responsible for a well-defined subset of the target SPL domain. The most important entities of the target SPL domain end up being represented as first-class concepts in DSLs, instead of lower-level abstractions based on lists and UI controls, as it happened in SharpLudus Adventure. Second, it allows an **incremental delivery of value**: even if the first version of a game SPL automates only one sub-domain, delivering a single DSL and generator, game designers and developers can already start harvesting the benefits from it. Finally, each sub-domain is evaluated for its automation potential, providing **more confidence to ensure the sub-domains with the best return on investment are the ones prioritized for automation**.

On the other hand, the sub-domain breakdown may require cross-DSL integration, which has a lot of challenges on its own. For instance, while it is quite straightforward to consume one class from another in the source code level, it is not similarly simple to make one DSL to access the concepts of another DSL, as well as ensuring the references are always in sync. While cross-DSL integration is not a topic approached by SharpLudus, Domain-Specific Game Development provides some guidance on how that can be achieved, exploring the concepts of name-based references, model bridges and model buses.

Some SharpLudus contributions to the *development core assets* area still apply to the current approach, such as the creation of semantic validators so that DSL users can catch modeling errors in design time. Another contribution that remains valid is the guidance for game SPL designers to choose a language workbench [Fowler, 2005].

In the *application core assets area*, both SharpLudus and Domain-Specific Game Development advocated for using game engines, state-of-the-art resources in game development, as a central piece of the Domain-Specific Game Architecture. However, the current approach brings an important improvement: promoting the game engine(s) to a domain framework which can be seamlessly consumed by generated code and therefore is able to move complexity away from code generators. This turns out to be extremely important since code generators are typically more difficult to maintain than a framework.

SharpLudus already had concerns related to making the generated games flexible and extensible enough so that the built-in SPL features could be complemented with custom, developer-added features as a result of creative processes in the domain. It suggested the double-derived design pattern [Cook et al., 2007] to be employed, along with more variability techniques and extensibility channels to the domain framework [Anastasopoulos & Gacek, 2001]. Such guidelines remain applicable to Domain-Specific Game Development.

5.3 Confirmatory Case Study: The ArcadEx Game SPL

While the aforementioned exploratory case studies provided an ad hoc, yet useful evaluation on what parts of the proposed approach were the most effective and what others had to be improved, we also developed a confirmatory case study. Its goal was to more comprehensively employ and evaluate the proposed Domain-Specific Game Development approach, as well as to become a reference game SPL for practitioners who intend to reproduce it[31]. Such a game SPL was entitled **ArcadEx** and the relevant details of its development are presented in the next subsections. It was developed by one software engineer and took around 8 man-days from inception to completion.

5.3.1 ArcadEx Domain Envisioning

The game domain vision for ArcadEx is presented in Table 7. Following the guidelines in Section 4.2 (*Envisioning the Game Domain*), its vision is able to clearly communicate the expectations for its generated products (games), by means of the core game dimensions. Although such a vision is not enough to determine how every single possible generated product may look like, it provides a comprehensive high-level overview of what is expected from the SPL, and it does not solely depend on a specific game genre. At the same time, it provides a baseline for starting a more refined domain scoping.

The motivation behind ArcadEx's domain is twofold: it represents a popular and well-known subset of the universe of digital games, yet not too complex, therefore being a feasi-

[31] The assets of this game SPL can be downloaded at http://sharpludus.codeplex.com

ble alternative for case studies and experiments with limited resources. Although some desired event triggers and reactions were identified, the *Event* core game dimension was explicitly recognized as needing more input from the Domain Analysis process. This exemplifies the fact that the resulting set of expectations should not necessarily be final or totally accurate. Additionally, the analyzed samples will not take into account the *Networking* core dimension, since ArcadEx will not offer built-in support for adding connectivity to its generated games, as noticed in the negative scope. Finally, custom core dimensions were not included for this case study, but could be explored in future versions of ArcadEx. We envisioned that *End-User Customization* could be a custom dimension, enabling players (not to be confused with developers) to customize the generated games. For instance, by using a visual assistant, players would be able to edit the appearance of their main characters or provide their own sound effects and background music to the game.

Table 7 – Domain Vision for the ArcadEx SPL

ArcadEx - Domain Vision	
Vision Statement: The ArcadEx SPL will be focused on generating single or multiplayer bi-dimensional arcade games for PC, with short levels composed by screens containing entities and surrounding walls, quick play action (in contrast to more in-depth gameplay or stronger storylines), simple, easy to grasp controllers, iconic characters and eventually rapidly increasing difficulty. Players control main characters who, or whose projectiles, collide with other entities such as non-player characters (NPC) or items. Victory condition is specified by the game designer as (a set of) game events: enemies are defeated, a score is reached, etc.	
Target Platforms: PC (Windows)	
Core Dimensions Expectations	
Dimension	***Expectation***
Player	Single player or local multiplayer, up to four players, which can play simultaneously or in turns. Each player has his/her own score.
Graphics	Bi-dimensional (2D) world. Background scrolling is supported. Heads-up Displays (HUDs) based on progress bars, text, icons or radars can be used to display game, player or entity properties, such as score, number of remaining hit points or timers.
Flow	ArcadEx games are composed by a series of screens. A screen can display information or host actual game action. A screen can lead to one or more screens and be reached from one or more screens. A starting screen should always be defined.
Entities	Each player controls one or more main characters. Other entity types are items and non-player characters (NPCs). Entity attributes include position, velocity, direction and rotation. Animations (superposition of images at a given frame rate) are supported.
Events	Events include: entities can be created or destroyed, collision detection, screen transition and changing an entity attribute value. Other events to be defined and refined by domain-analysis.

Core Dimensions Expectations (Continued)	
Dimension	***Expectation***
Input	Keyboard and/or Xbox 360 gamepad controller.
Audio	Sound effects are supported as event reactions; background music can be associated with game screens and can be played in loop.
Physics	Collision detection, bouncing, velocity and acceleration. Screens contain blocking walls.
Artificial Intelligence	Primitive AI concepts such as chasing a main character are expected.
Networking	ArcadEx games are standalone. All data and play modes are locally constrained.
Negative Scope	
Dimension	***Negative Expectation***
Player	Player scores are not stored across game sessions. No high-score concept is supported as built-in.
Physics	There will not be built-in support for elaborated physics models, such as fluids and friction. Platforms will not be supported.
Audio	Audio in ArcadEx will be as simple as playing background music and sound effects, without any built-in support to add special effect such as echo, 3D sound, etc.
Graphics	Some arcade games explore isometric (2D ½) or even 3D views. ArcadEx games, however, will stick to bi-dimensional games, typically viewed from above. No built-in support is provided for UI controls, such as menus, textboxes or drop-down lists.
Networking	There is no built-in support for any kind of network connectivity.

5.3.2 ArcadEx's Domain Analysis

About 30 games were selected for ArcadEx's Domain Analysis, such as Pac-Man, Space Invaders, Asteroids, Defender, Geometry Wars, 1942, Missile Command and Rally-X, among others. Since many of the most successful arcade games were firstly (and sometimes only) made available in arcade cabinets, also known as arcade machines or "coin-op" (coin-operated machines), some of the consulted sources included the International Arcade Museum[32] (the world's largest museum of the art, inventions, and history of the amusement and coin-operated machine industries) and its video-game division, KLOV[33] ("Killer List of Videogames"), which has created an authoritative database on coin-operated video-games.

Other domain sources included specialized websites (such as The Dot Eaters[34] and the Arcade History database[35]), the Wikipedia list of arcade games[36], specialized magazines,

[32] http://www.arcade-museum.com

[33] http://www.klov.com

[34] http://www.thedoteaters.com

[35] http://www.arcade-history.com/index.php?page=database

[36] http://en.wikipedia.org/wiki/List_of_video_arcade_games

social networking communities focused on the subject and specialized events, such as the California Extreme[37], which is USA's largest show of classic (and playable) coin-op video-games. The history of arcade games was also studied and brought some interesting insights [Discovery, 2007]. Since some of the analyzed samples are as old as information technology itself (dating from the 60s and 70s), emulator tools were employed to enable a better exploration of some of the games and classifying them as belonging or not to the domain. The final list of game samples selected to be analyzed included games which are representative, unique and whose importance to the domain were (sometimes worldwide) recognized by the industry. Many of them were re-released in popular remakes [Kent, 1996].

Due to the simpler nature of the ArcadEx SPL domain (encompassing arcade and casual samples) it took less than a couple hours to analyze each domain game. However, more complex domains might require more efforts, since they encompass more complex products which demand more time to have its features properly understood and studied. For instance, the aforementioned Elegy case study for role-playing games (Section 5.2), pointed out that some RPG games required almost 30 analysis hours per game [Azevedo et al., 2009].

The proposed Domain-Specific Game Development approach turned out to be very useful for discarding samples, filtering out conflicting features and refining the SPL scope. The expectations set for ArcadEx's core game dimensions avoided resources and effort to be spent on the analysis of games and features that do not actually belong to the SPL scope, hence aided the Game Domain Analysis to keep focus and stay on track. For instance, the games BattleZone and Star Wars for the Atari system (Figure 38) are considered to be some of the most successful coin-op arcade games ever. Game SPLs solely based on game genres would be misguided to include them as samples to be analyzed. However, the Domain-Specific Game Development approach made it evident that the popular concept of "arcade games" is much broader than the actual scope of the ArcadEx SPL. As an outcome, the expectations set for the ArcadEx's core game dimensions promptly identified that the first-person gameplay and the 3D camera view of BattleZone and Star Wars are out of scope, no matter how such games are informally classified.

The ArcadEx game SPL also initially considered the games Paperboy and Pole Position (Figure 39) for Domain Analysis. Nevertheless, the employment of the proposed approach revealed that the isometric scrolling action of Paperboy presented challenges beyond the ArcadEx's scope. The same happened for Pole Position, whose camera view, differenti-

[37] http://www.caextreme.org

ated gameplay and collision detection mechanisms imply in unsupported built-in variability for the ArcadEx SPL. Both games were also discarded from Game Domain Analysis.

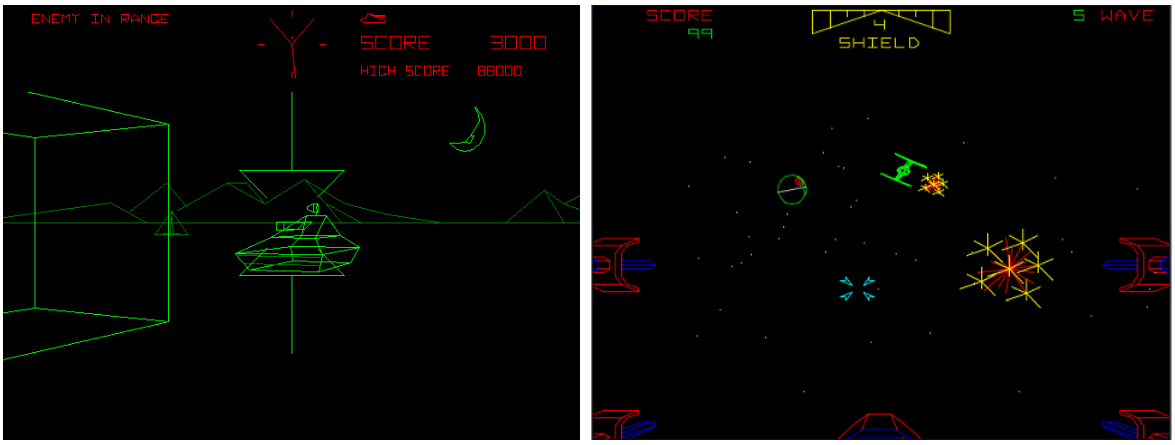


Figure 38 – BattleZone and Star Wars, by Atari

Figure 39 – Paperboy and Pole Position: also out-of-scope

Other samples that fall into this case are Gauntlet and Airlock (Figure 40). Airlock contains characteristics of platform games, such as surfaces on which the main character run, elevators, and gravity. On the other hand, Gauntlet contains more refined adventure and RPG elements, such as special characters, each one with specific skills (Warrior, Wizard, Elf and Valkyrie), item inventory, potions, doors, keys, rooms and teleporters. Adding Airlock or Gauntlet to the ArcadEx's Domain Analysis process would be troublesome, since this would result in many new features that conflict with the original scope and contribute to turn the SPL into a more generic one. The gravity system and more elaborated physics of Moon Patrol, Donkey Kong and Joust (Figure 41) made such games to be discarded from Domain Analysis for the same reasons.

Following the proposed approach, we concluded that creating sub-domains to encompass such games with special features would not be an interesting approach. A sub-

domain is a specialization of the main domain and respects the SPL vision and scope, while the above games present features that conflict with them. Adding conflicting features to the SPL later on may be dangerous, especially if the game SPL increases its scope in a way that it gets too generic, loses its specific focus and consequently provides a reduced effectiveness, while becoming harder to maintain. A better approach would be to create other SPLs to address the conflicting features, such as SPLs specifically for platform and racing games.

Figure 40 – Airlock and Gauntlet better belong to other game SPLs

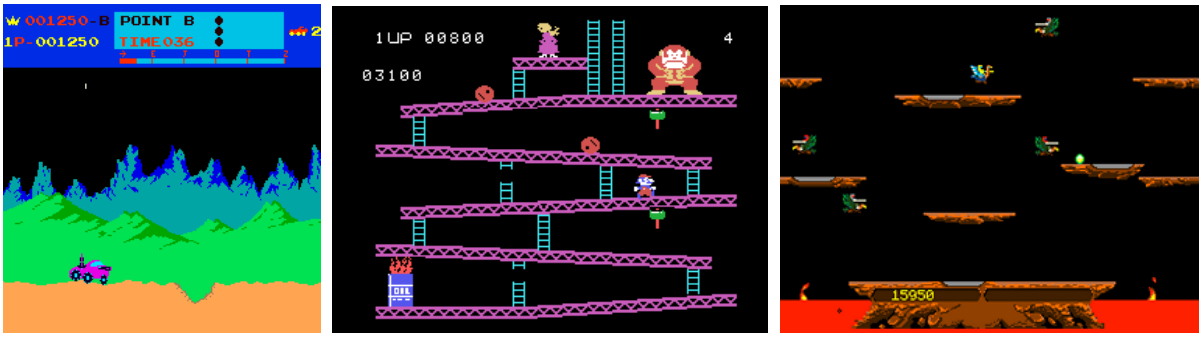


Figure 41 – Moon Patrol, Donkey Kong and Joust: false positive ArcadEx game samples

In the first iteration, the features identified by the Game Domain Analysis were a direct outcome of the core dimensions defined in Table 7: Player, Graphics, Flow, Entities, Events, Input, Audio, Physics and Artificial Intelligence. Features were then refined after iterations and we eventually came up with a feature model with almost 150 features to describe the domain's commonality and variability, which is detailed in Appendix A. Such a feature modeling work was carried out in parallel with other activities, as sub-domains were identified, refined and had their potential assessed for automation, resulting in DSLs and generators. The achieved feature model is not only an outcome of the analyzed game samples, but also takes into account possible "future features" as suggested by the approach. It is worth noticing that the feature model is by no means final, as it can be enhanced with the analysis

of new samples and the envisioning of additional future features (e.g., adding multiple scrolling layers to a screen background).

In order to assess whether enough information was minimally collected from the samples and to determine whether Domain Analysis could stop, we employed the suggested "domain understanding churn" approach, i.e., we continuously measured how much the domain understanding was changing after each sample was analyzed.

Initially, the churn was very high, indicating that the analysis should continue. For example, the entity movement analyzed in the first game samples was continuous, i.e., we assumed an entity position was the result of a velocity function in the world space. However, games such as Frogger and Tapper (Figure 42) revealed this is not always the case: entity movement can be based on tiles or restricted to specific world (screen) regions. In other words, entities in such a special case move through specific positions in the game background, never being able to stop between them. Tapper also reveals that enemies can, similarly, appear from a set of pre-defined locations, albeit randomly.

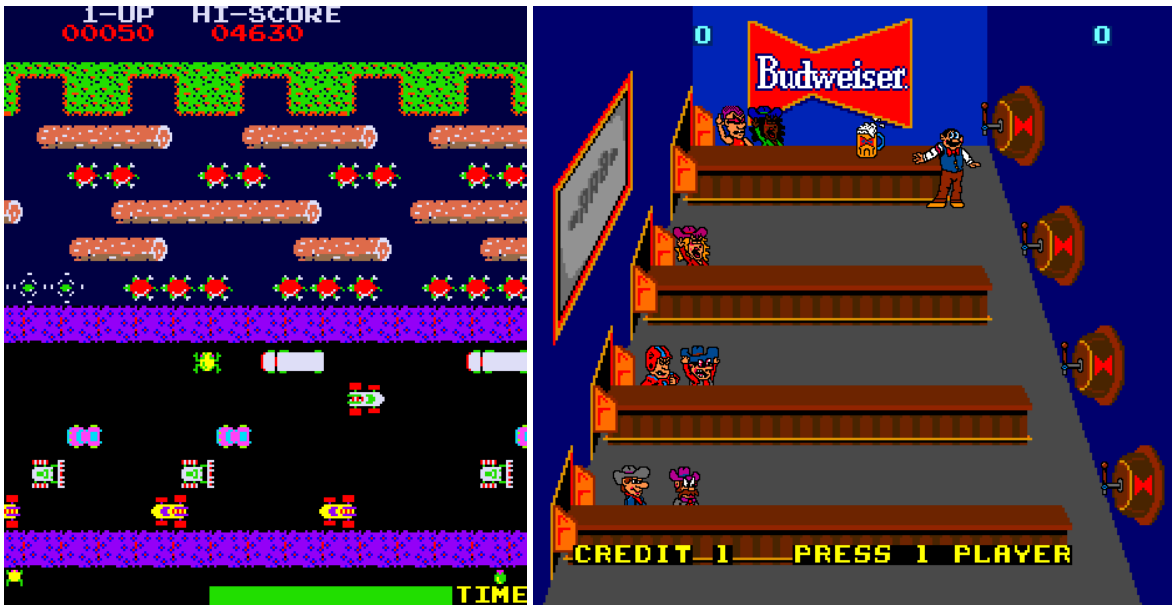


Figure 42 – Frogger and Tapper: new movement variability

Another example relates to the games Time Pilot and Bosconian (Figure 43), which broke some background scrolling assumptions of the ArcadEx SPL. In those games, even though the game world is bigger than the game screen, the screen camera is fixed in the player's ship. When the player asks the ship to move left, actually all the scenario and enemies move right and the ship always remains in the center of the screen.

In a given moment, however, the inclusion of new samples in the ArcadEx SPL Domain Analysis was not adding more information to the domain understanding, but only attesting that the already collected and classified data were indeed accurate and comprehensive.

New analyzed games such as Sea Quest, Asteroids (including sequels) and River-Raid (Figure 44) reveled patterns which were already registered in previous game evaluation records, although the contents of each game (sprites, story, etc.) were obviously different. At that moment, the Domain Analysis was considered satisfactory and concluded, but still open to future iterations based on the need to more deeply analyze sub-domains.

Figure 43 – Time Pilot and Bosconian: new scrolling variability

Figure 44 – Sea Quest, Asteroids and River-Raid: no new variability

As the ArcadEx feature model was refined, it already revealed some reuse opportunities. Almeida et al. [2006] mentions that a feature model with AND-nodes at an upper level and OR-nodes at a lower level usually indicates a high level of reuse opportunity. This seems to be the case of ArcadEx: in the upper-level many AND-nodes can be observed (e.g., the Entity feature has a role AND a shape AND collision targets AND a movement type AND other sub-features), while OR-nodes are present in the lower levels (e.g., multiple options for the entity movement, multiple options for an entity shape, etc.). See Appendix A for details.

5.3.3 Bridging the ArcadEx Domain Analysis to Application Core Assets

While the Game Domain Analysis and feature modeling were being performed from a top-down perspective, we employed the "edge-center" nature of the Domain-Specific Game Development approach to also evaluate game samples from a bottom-up perspective, aimed at achieving sample implementations, a reference architecture and ultimately a domain framework.

In the specific case of the ArcadEx SPL, source code was available only for a few of the analyzed samples, while we had to implement the most representative samples from scratch. In the pursuit of establishing a domain framework for the ArcadEx game SPL, different game engine possibilities came out, such as the XNA Parallax Engine[38] for background scrolling, Farseer Physics Engine[39] for physics modeling and the Saq 2D[40] engine for tiling. Other game engines and libraries available in the domain include the Torque Game Builder[41], BlitzMax[42], PTK[43] and the PopCap framework[44].

Among all the game engine options considered, the FlatRedBall Game Engine[45], targeted at the Microsoft XNA game platform [Carter 2007], presented the most comprehensive coverage of the features analyzed with the lowest learning curve. FlatRedBall is a 2.5D game engine focused on ease of use and asset management. Basic 3D games can be developed using the FlatRedBall Engine, although most of its functionality is built with the assumption that the game is 2D. The engine provides a starting project template that can be integrated into Visual Studio, so that users can get into programming right away. Later on, such a possibility was explored by ArcadEx in order to create a development core asset corresponding to a new project template in the IDE. Sample games developed with the engine can also be integrated to Visual Studio and accessed as if they were starter kits. A series of managers to handle specific game tasks are provided, such as the input manager, sprite manager, shape manager, text manager, file manager, time manager and screen manager. Other interesting built-in features are keyboard-joystick mapping, entity bouncing after collision, animation and general entity rendering support (rotation, translation, transparency, etc.), GUI elements (but-

[38] http://www.codeplex.com/xnaparalax

[39] http://www.codeplex.com/FarseerPhysics

[40] http://saqscrap.com/saq2d.aspx

[41] http://www.garagegames.com/products/torque/tgb

[42] http://www.blitzmax.com

[43] http://www.phelios.com/ptk/

[44] http://developer.popcap.com

[45] http://www.flatredball.com

ton, textbox, combobox, etc.) and tile-based movement. The engine makes use of cameras, even if the game under creation is 2D.

Figure 45 presents some of the implemented games as part of creating a Domain-Specific Game Architecture. Instead of just implementing the analyzed game samples as is, we majorly created sample implementations for *new* games, containing a combination of features present in the analyzed samples as well as features we anticipated in the feature models. Our main goal was twofold: (1) to create sample implementations covering the modeled features, toward a reference game architecture, and (2) to validate FlatRedBall as an engine capable of supporting the commonality and variability identified for the domain.

The sample implementations revealed that FlatRedBall required an adapter layer in order to simplify its consumption by the generated code. Such a layer, entitled **ArcadEngine**, was implemented in 2500 lines of code (1050 excluding comments and whitespaces). It promotes the FlatRedBall game engine to a domain framework, by specializing its interface to the envisioned ArcadEx domain, making it more compliant with the concept of framework completion and enabling it to be more easily configured. In fact, common code was identified across multiple implementation samples and then refactored into ArcadEngine. This way, as illustrated by Figure 46, code generated from the ArcadEx models can consume ArcadEngine to configure FlatRedBall, while the full power of FlatRedBall is still available to game developers who require the implementation of more complex game behaviors through complementary custom code.

Figure 47 details ArcadEngine's architecture. The *Arcade2DGame* class specializes the XNA's *Game* class for the arcade domain. Utility and helper classes such as the *AudioManager* complement the other FlatRedBall manager classes (*InputManager*, *SpriteManager*, etc.). The *Arcade2DScreen* class specializes FlatRedBall's *Screen* class for the arcade domain, and is complemented by classes with screen-related concepts such as heads-up displays (HUDs) and walls. Finally, the *Entity* class, along with its sub-classes and the *EntityState* class, specializes FlatRedBall's *Sprite* class.

5.3.4 Bridging the ArcadEx Doman Analysis to Development Core Assets

Multiple iterations were used to identify and prioritize the ArcadEx's sub-domains for automation. Besides analyzing samples, extracting and detailing features, inspecting and implementing code, and refining a domain-specific game architecture, we ultimately came up with four DSLs and code generators, instead of a single bloated DSL that lacks conciseness and maintainability. The ArcadEx DSLs were implemented with the Visual Studio Team System (VSTS) language workbench technologies, called DSL Tools [Cook et al., 2007]. It provides a framework and toolset that enable partners to build custom visual designers and do-

main-specific language designers using Visual Studio. A description of the approached sub-domains and how they contributed to the creation and/or refinement of development core assets, such as DSLs, is presented in this subsection. The identified sub-domains are highlighted in **bold**.

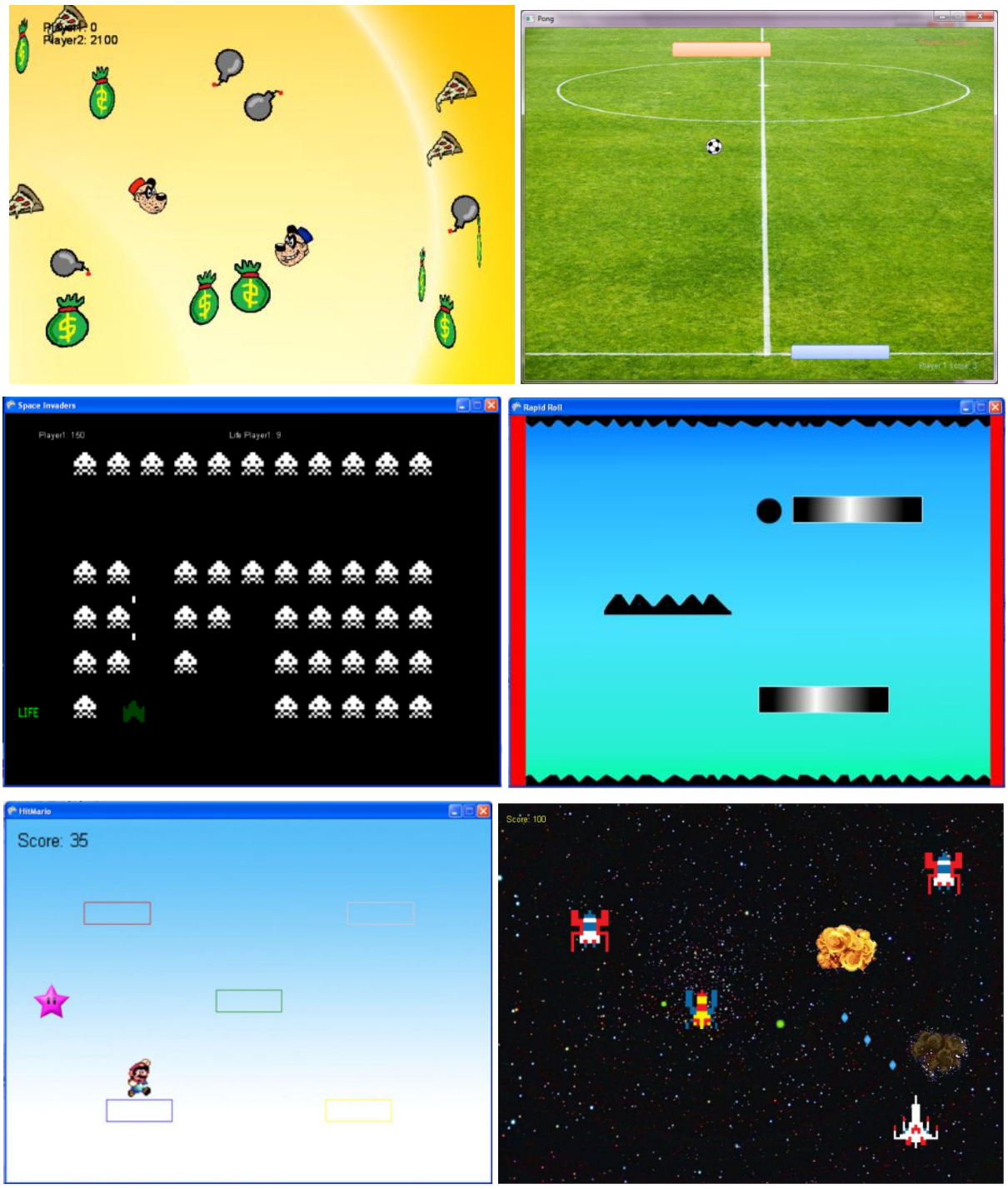


Figure 45 – Some games implemented toward ArcadEx's reference architecture

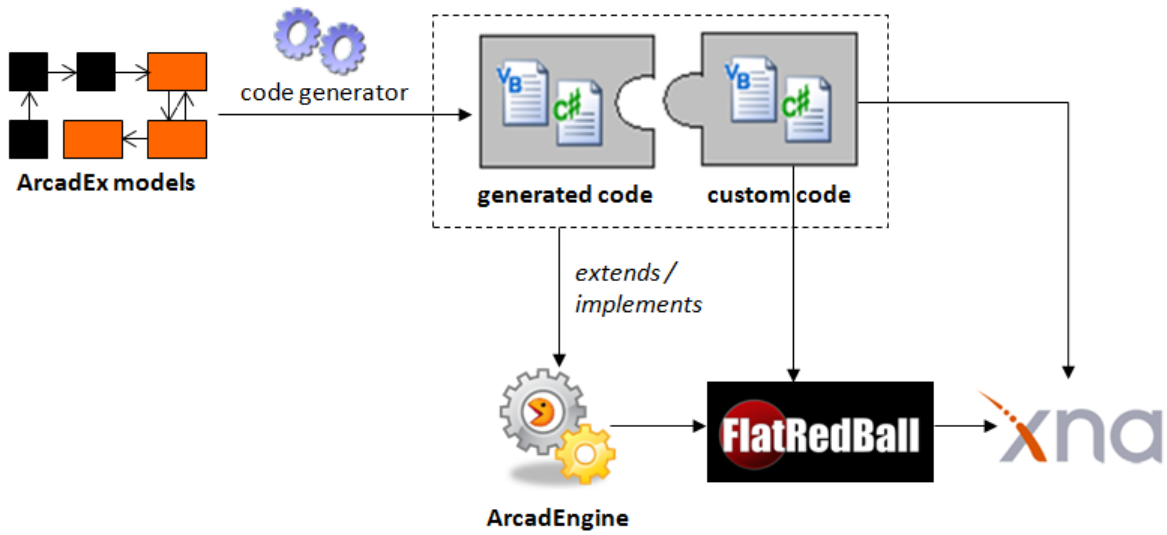


Figure 46 – ArcadEx assets overview, including ArcadEngine and FlatRedBall

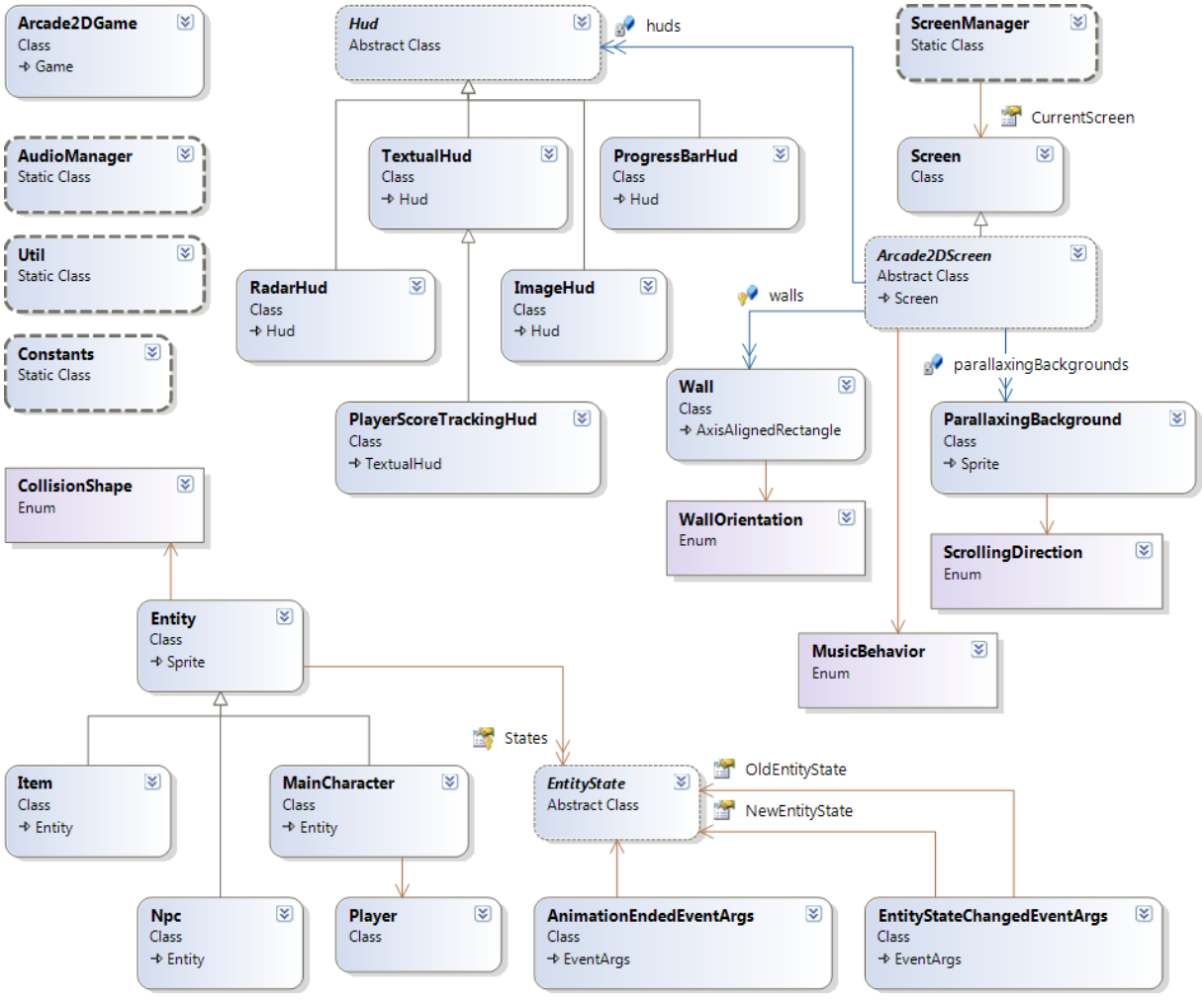


Figure 47 – ArcadEngine architecture

In the first iteration, we approached the **screen transition** sub-domain, including the different triggers that make an ArcadEx game to move from one screen to the other, such as an input action or a timer. Following the extensibility guidelines, we added support for custom transition events, which can be programmed by game developers but still be referenced from the models. This resulted in a first version of the *ScreenTransitionDSL*, which was then renamed to *GameDefinitionDSL* (Figure 48) in the second iteration after we concluded that such a DSL was the one through which developers could also specify the **top-level properties of a game** (such as its window mode, resolution, etc.).

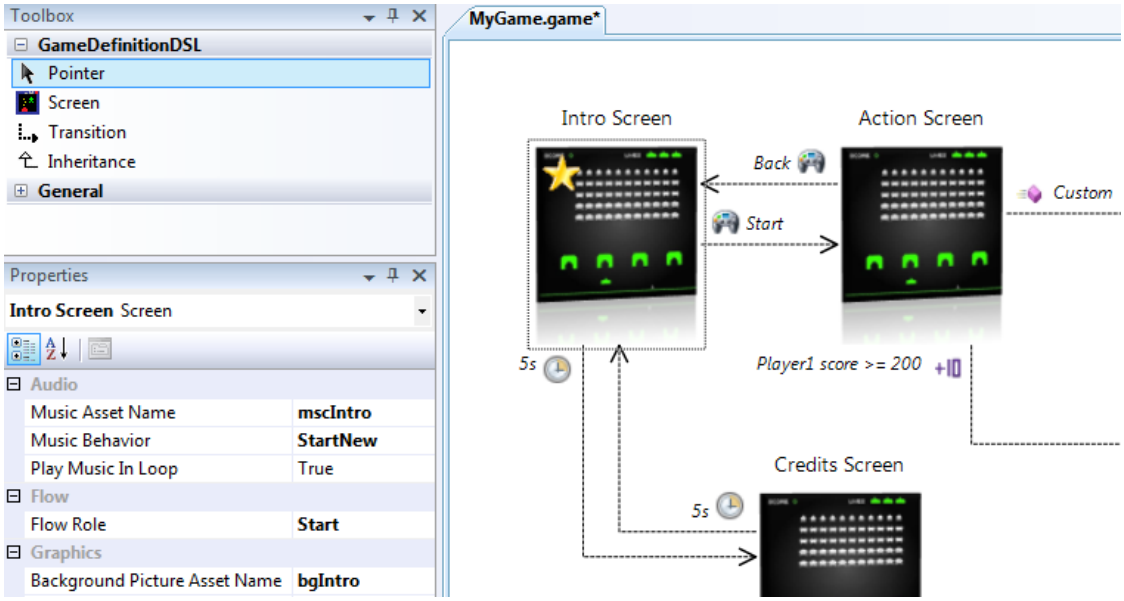


Figure 48 – Diagram modeled with GameDefinitionDSL

Since the ArcadEx DSLs were not developed in an isolated manner, but in the context of a game SPL, they integrate seamlessly with the development environment and other assets. For example, more complex domain concepts can be edited from the IDE's Properties Window, together with custom property editors provided by ArcadEx. This is illustrated by Figure 49 (left hand-side), in which a custom property editor was implemented for editing screen transition events in the GameDefinitionDSL. In addition, two other assets were implemented in parallel with each DSL: (1) semantic validators that raise errors at compile time and are displayed to game developers through the IDE Error List (Figure 49, right hand-side), and (2) transformations such as code-generators that receive a diagram modeled with the DSL as input, and output other artifacts such as code that will consume and configure the ArcadEngine adapter layer. For instance, Figure 50 illustrates the GameDefinitionDSL's code generator. By reading the properties of the game concepts modeled in a GameDefinitionDSL diagram, the scriptlets output a base game class that initializes the graphics mode of the game (Figure 51). Developers launch such transformations from within the IDE.

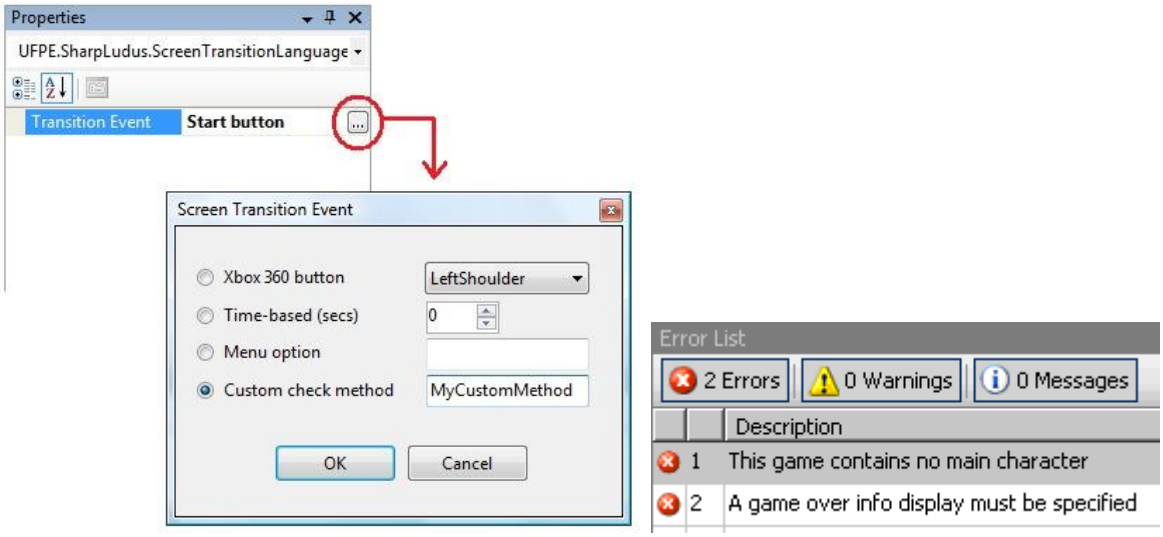


Figure 49 – Integrated tool support for domain-specific modeling experience

```
/// <summary>
/// Base game class that actully implements the behavior of the game.
/// Having a base game class and a derived game class is an implementation
/// of the double-derived design pattern.
/// </summary>
public abstract class <#=this.Game.Name#>Base : Arcade2DGame
{
  protected override void Initialize()
  {
    base.Initialize();
    InitGraphicsMode(
      <#=this.Game.Resolution.Width#>,
      <#=this.Game.Resolution.Height#>,
      <#=(this.Game.DisplayMode == DisplayMode.FullScreen).ToString().ToLower()#>);
    DefaultInputMapping.ApplyMappings();
  }
}
```

Figure 50 – Excerpt of GameDefinitionDSL's code generator

```
/// <summary>
/// Base game class that actully implements the behavior of the game.
/// Having a base game class and a derived game class is an implementation
/// of the double-derived design pattern.
/// </summary>
public abstract class PongBase : Arcade2DGame
{
  protected override void Initialize()
  {
    base.Initialize();
    InitGraphicsMode(
      800,
      600,
      false);
    DefaultInputMapping.ApplyMappings();
  }
}
```

Figure 51 – Example of code generated by the GameDefinitionDSL's code generator

In the third iteration, we refined the GameDefinitionDSL as a result of approaching the **screen background** sub-domain, allowing game developers and designers to assign static

background pictures to screens instead of manually programming code to render them. In the fourth iteration, we explored the **background music** sub-domain. Properties such as what music asset to play as background music were added to each screen, along with the background music behavior such as “start new music”, “keep playing the current one” or “inherit background music behavior from the parent screen”.

Once again, refining the IDE integration as a result of iterating through such sub-domains was key to improve the modeling experience. For example, the GameDefinitionDSL was further integrated into the IDE by means of being aware of music, sound effect and texture assets added by game developers and designers to the current game solution. Figure 52 reveals that as soon as a new music asset is added to the *Audio* solution folder, the asset becomes available through the “Music Asset Name” property of game screens.

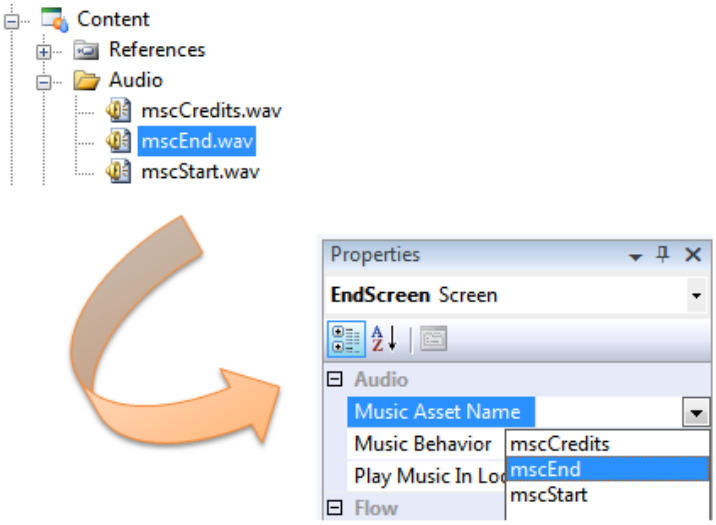


Figure 52 – Further GameDefinitionDSL IDE integration

After these four iterations, game developers and designers had a suitable version of the GameDefinitionDSL, which allowed them to perform various screen flow management tasks in a higher level of abstraction, via DSLs and models. Nonetheless, other game features still had to be programmed in the low level.

In order to improve the (manual) testability and expedite experimentations of the game SPL, we then approached the **input mapping** sub-domain, i.e., the mapping of the gamepad (controller) buttons to keyboard keys. The implementation of this sub-domain resulted in a new DSL, called *InputMappingDSL* (Figure 53). This DSL saves game developers effort by enabling the modeled gamepad input triggers and reactions to be reused for handling keyboard input as well. In other words, once the gamepad-to-keyboard mapping is defined, players can enjoy ArcadEx games even if no controller is available, by using the keyboard, although input triggers and reactions in the models are still specified only by means of gamepad buttons. Developers can map controller buttons to keyboard keys one by one, or

apply “common mappings”, which are multiple button-key mappings commonly used. An example is mapping a controller analog stick to the keyboard arrow keys (up, down, left and right).

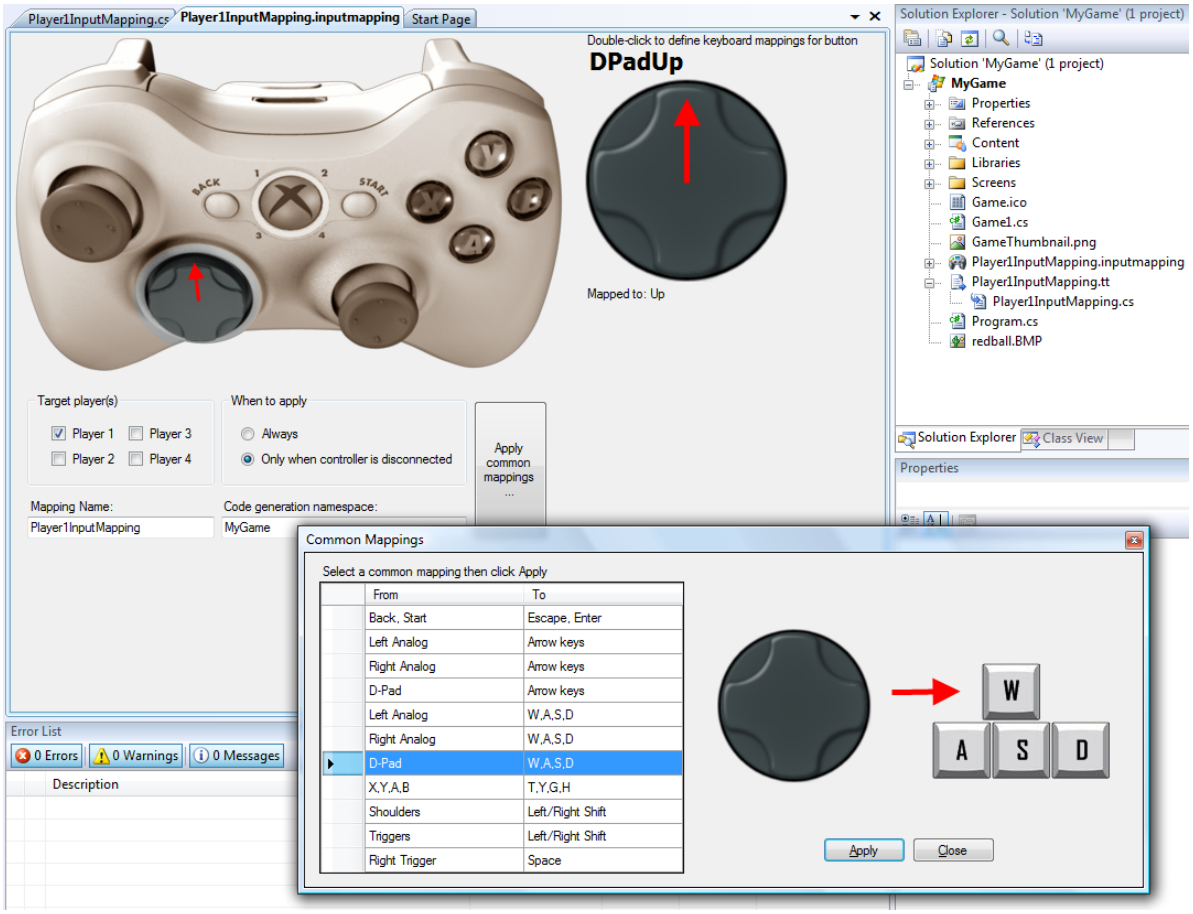


Figure 53 – InputMappingDSL modeling experience and IDE integration

InputMappingDSL is also a good case of domain variability whose modeling requires creative construction, instead of routine configuration. As a result, a more powerful, custom modeler was created for this DSL, as we concluded that the built-in visual syntax provided by the DSL Tools, based on arrows, image shapes and geometry shapes would not suffice for this language.

Subsequent iterations explored the **entity definition** sub-domain, including entity states and animations, resulting in the *EntityDSL* (Figure 54) through which the “things and beings” of ArcadEx games could be modeled, instead of programmed. Many iterations were required to refine this DSL, approaching related sub-domains such as the **declaration of collision interest** between entities, **entity input handlers** (single-button, thumbstick-based entity movement, etc.), **entity event reactions** (create entity, destroy entity, switch state, etc.), **entity-based timer events** and others.

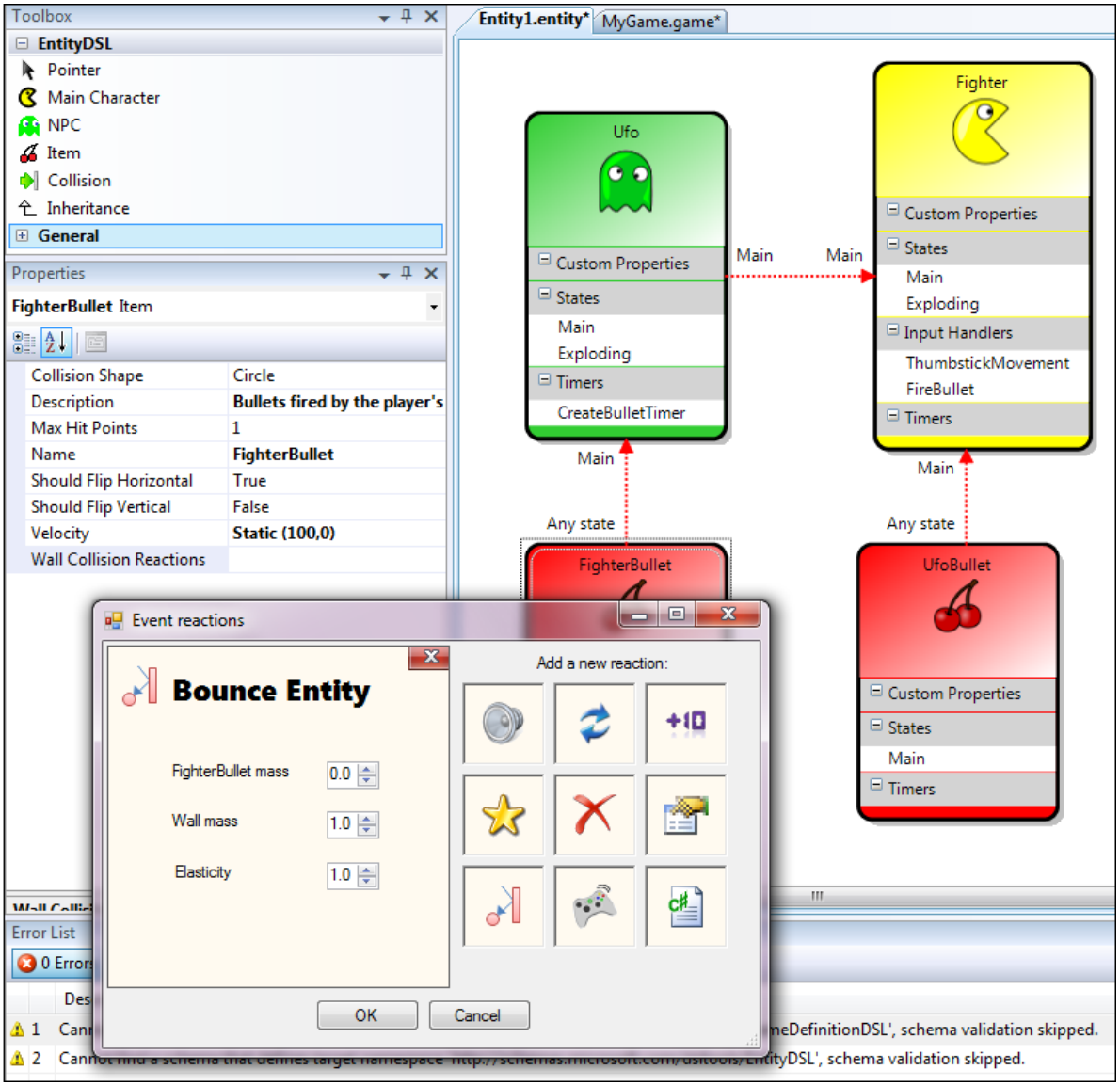


Figure 54 – EntityDSL modeling experience and IDE integration

Some modeling experience based on creative construction was required for EntityDSL as well, especially with regard to modeling *reactions* (play a sound, set an entity state, etc.) that handle *events* (input event, timer event, etc.). Following the suggested task analysis process [Preece et al., 1994] (Section 4.5.2) and inspired by Kodu[46], a visual game programming environment designed by Microsoft Research for children, ArcadEx allows developers to define a list of reactions for a given event by picking each reaction at a time, from a grid of representative icons, and configuring it by filing parameters in just a few UI interactions. This shows the value of performing task analysis on already existing languages and tools, toward designing intuitive user experiences.

[46] http://research.microsoft.com/en-us/projects/kodu/

Due to the sub-domain prioritization guidelines, some of the EntityDSL refinements were actually alternated with the creation and refinement of another DSL, called *ScreenDSL* (Figure 55 and Figure 56). Such a DSL is the result of prioritizing sub-domains related to screen contents, such as **heads-up displays** (textual, icon or progress bar), the **placement of entity instances** in a screen and **screen-based timer events**.

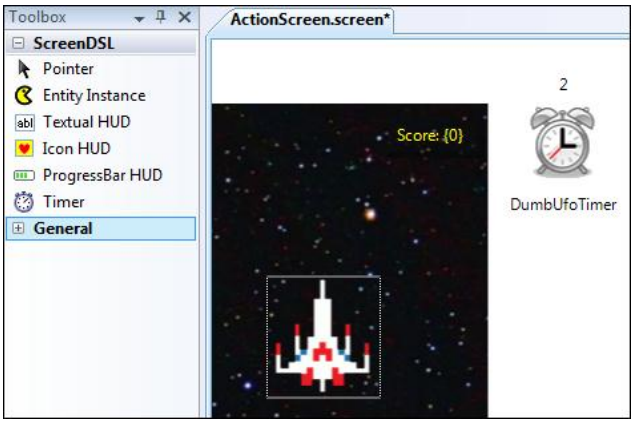


Figure 55 – ScreenDSL modeling experience and IDE integration for the game "2942"

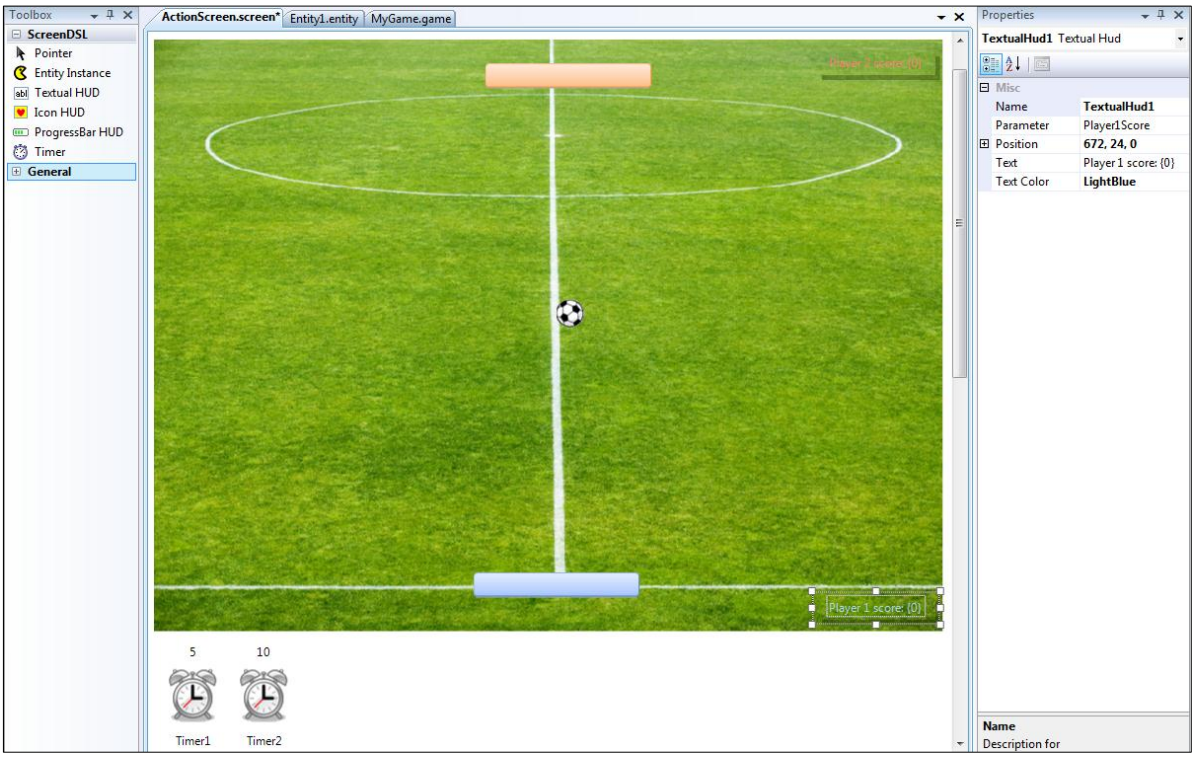


Figure 56 – ScreenDSL modeling experience and IDE integration for the game "Pong"

Having a rich and dynamic syntax, ScreenDSL provides a good case of cross-DSL integration and related challenges. As Figure 55 and Figure 56 suggest, a ScreenDSL diagram needs to query the GameDefinitionDSL for the background asset of the screen being mod-

eled, so that it can be dynamically rendered in the background. Similarly, when the developer drags the “entity instance” concept to the screen canvas, ScreenDSL needs to query EntityDSL for the available game entities, so that a list of entities can be displayed to the developer for him to pick one. ScreenDSL then queries EntityDSL for the graphical representation of the entity, in order to properly render it in the screen canvas. In addition, when a heads-up display is modeled in ScreenDSL, it can be associated to a player or entity property, such as the number of remaining hit points, which needs to be retrieved from EntityDSL. Coordinating cross-DSL integration was not a trivial task. While it is quite straightforward to consume one class from another in the source code level, it is not similarly simple to make one DSL access the concepts of another, as well as to ensure their references are always in sync. The use of model buses to publish and retrieve concepts aided the process.

Since ArcadEx’s DSLs were not developed one after another, as it happens in waterfall approaches, we continuously refined and revisited the DSLs as the game SPL evolved and new sub-domains were identified and prioritized. For example, the GameDefinitionDSL was updated after the **scrolling backgrounds** sub-domain was chosen for automation. Likewise, EntityDSL was revisited to support some AI behavior, enabling developers to switch a flag in order to make a NPC to **chase main characters**.

Likewise, we improved IDE integration incrementally, such as enabling ScreenDSL diagrams to be opened after the developer double-clicks screen concepts in the GameDefinitionDSL. A new Visual Studio project template (Figure 57) was also created and added to the IDE’s “New Project” dialog, so that ArcadEx’s users could have a starting point in the creation of ArcadEx games. Once unfolded, such a project template creates a new ArcadEx game with a start screen, a credits screen and a couple of background music and background texture assets. Developers can compile and run such starting code straight away.

Similarly, in order to move complexity away from the multiple code generators, improvements to the ArcadEngine adapter layer were developed incrementally, ensuring the FlatRedBall game engine was seamlessly consumed by new generated code as the prioritized sub-domains were implemented and DSLs were created and/or refined. Special attention was given to offer easy-to-consume extensibility hooks plugged into ArcadEngine and FlatRedBall for unforeseen game behaviors. In summary, each iteration was finalized with the high and low-level work meeting halfway, culminating with the design and implementation (or refinement) of one or more DSLs and generators for the iteration’s sub-domain.

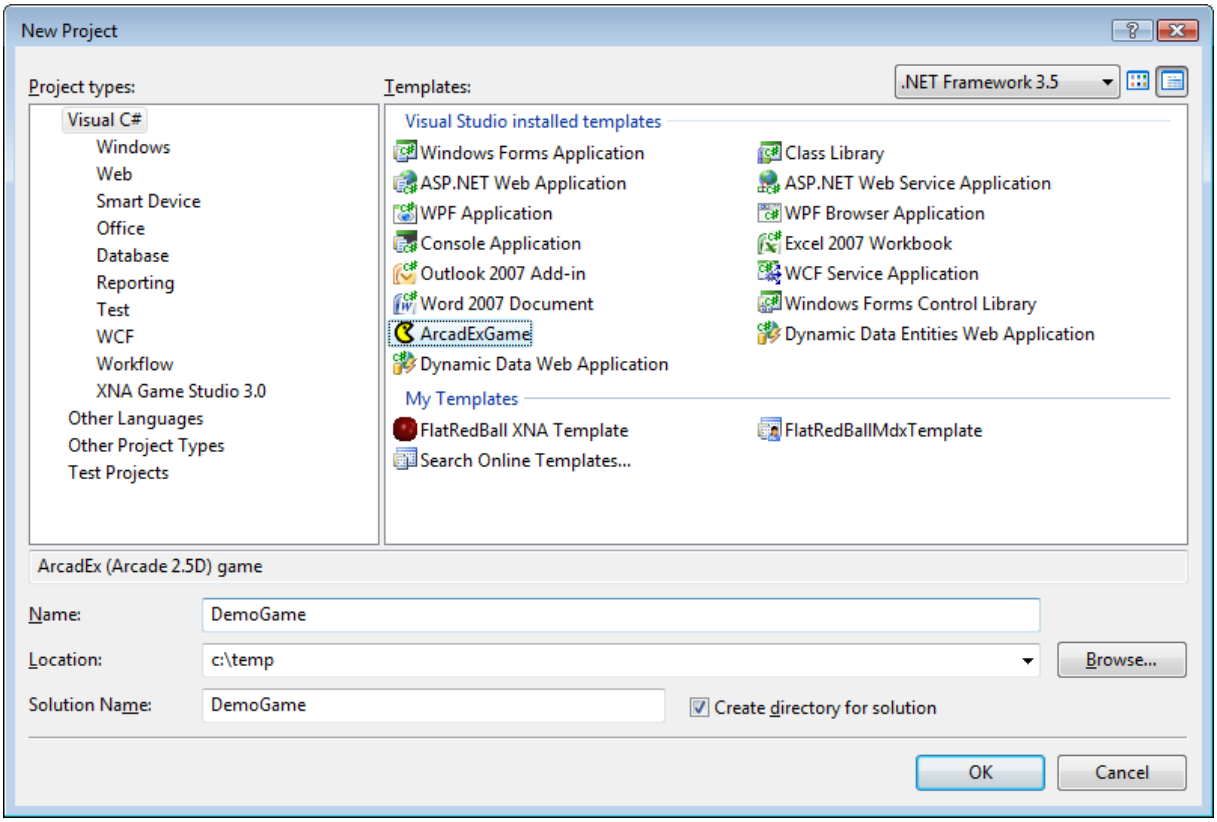


Figure 57 – ArcadExGame project template integrated into the Visual Studio IDE

5.3.5 ArcadEx Evaluation

The development of the ArcadEx case study took around 8 man-days. Since its inception, we have been collecting data to assess its effectiveness. Some of the benefits we were able to observe upfront include an incremental delivery of value via the prioritized sub-domains automation, a reduced complexity to consume game engines (promoted to domain frameworks) from the generated code and domain-specific assets tailored to the unique characteristics of the envisioned family of games.

Three examples are given next as an illustration of the abstraction level provided by ArcadEx assets. To start with, Figure 58 shows three UI actions required to make a non-player character (NPC), such as a UFO, to fire in every second a bullet that chases the player's main character, such as a Fighter. First, a new timer for the UFO NPC is added. Via the Properties tool window, the timer's interval is set to 1 second, and finally a "Create Entity" event reaction is added to the timer's tick handlers. Such an event reaction enables developers to set the velocity behavior of the newly created entity (bullet) to chase a main character.

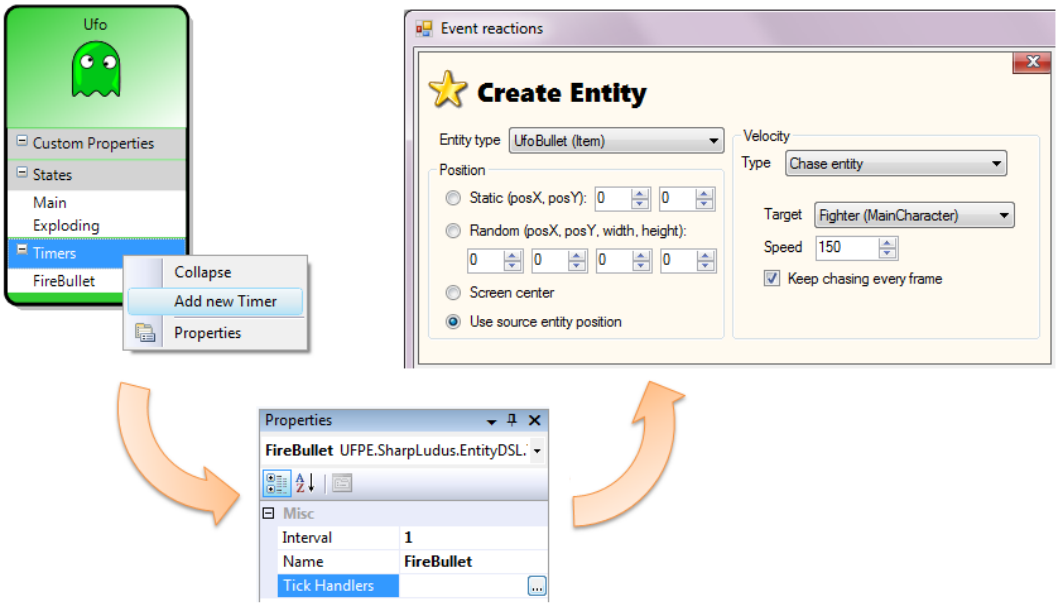


Figure 58 – UI actions required for making a NPC to fire a bullet every second

In contrast to the UI actions above, Figure 59 presents the generated code responsible for implementing the modeled behavior, which would have to be programmed manually otherwise. The code handles the initialization, checking and resetting of the timer variable, as well the creation and initialization of the new UFO bullet instance, which is set to chase the main character. The *Chase* method is provided by the ArcadEngine adapter layer – the game engine alone does not provide built-in support for making one entity to chase another. As it can be noticed, the three simple modeling tasks above abstract more than 15 lines of code.

```
public abstract partial class UfoBase : Npc {
   protected double fireBulletStartTime = TimeManager.CurrentTime;
   protected double fireBulletInterval = 1f;

   protected override void RunTimerChecks(){
      base.RunTimerChecks();
      if (TimeManager.CurrentTime - fireBulletStartTime > fireBulletInterval) {
         UfoBullet ufoBullet = new UfoBullet();
         ufoBullet.Position = this.Position;
         ufoBullet.Chase(this.Screen.GetEntity(typeof(Fighter)), 150, true);
         this.Screen.AddEntity(ufoBullet);
         this.fireBulletStartTime = TimeManager.CurrentTime;
      }
   }
}
```

Figure 59 – Generated code for making a NPC to fire a bullet every second

The collision between entities and walls, and among entities themselves, is another sub-domain in which ArcadEx assets provide a useful level of abstraction. Figure 60 presents the Wall Collision Reactions editor, being used to define the reactions for Pong's *Ball* entity. Such an editor enables developers to specify what reactions are launched when a given entity collides with a specific wall of a screen. The amount of coding that the editor saves, in the

context of the Pong game, is presented in Figure 61 (the code for creating and initializing the ball is refactored into a new method to avoid duplication). Once again, a few modeling actions save developers a good amount of coding and time.

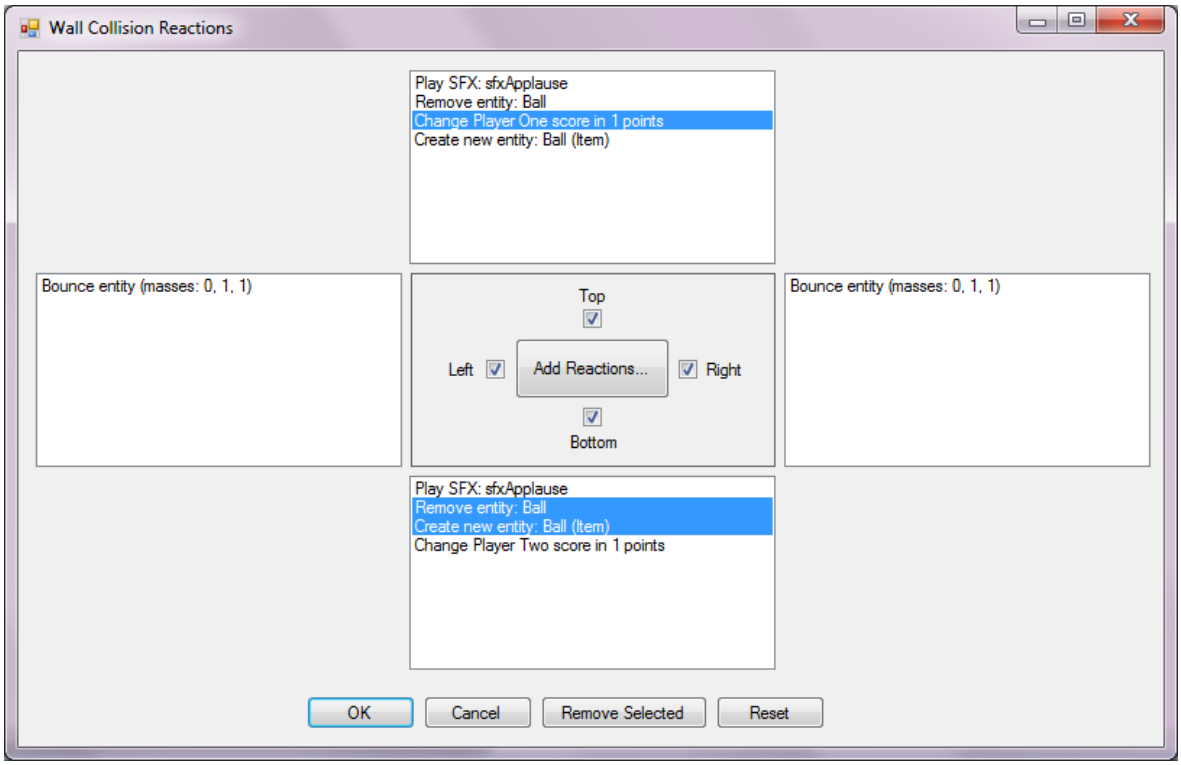


Figure 60 – Wall Collision Reactions editor

As a final, and perhaps most relevant example of ArcadEx's abstraction, Figure 62 presents an EntityDSL diagram in which collision interests are declared among the entities of the space shooter game "2942", such as the Fighter main character, the FighterBullet item, and the DumbUfo and SmartUfo NPCs. The amount of collision lines between such entities show that the game has a lot of collision rules, such as:

- If a Fighter collides against an UFO and neither is exploding, both explode.
- If an exploding UFO collides against a non-exploding UFO, the latter explodes.
- If a FighterBullet collides against a non-exploding UFO, the latter explodes.
- If a non-exploding SmartUfo collides against a non-exploding DumbUfo, the former bounces against the latter

Although we recognize that drawing collision lines multiple times is not a very exciting task, implementing such rules manually is a much more error-prone and tedious task, requiring multiple *for* loops to iterate through the entity instances and *if* branches to check their state. Figure 63 and Figure 64 list the code required to implement the collision rules, omitting

the actual code for the collision reaction handlers. If more entities and more states were part of the game specification, the code maintainability and readability would become worse.

```
public override void HandleWallCollision(Wall wall)
{
	base.HandleWallCollision(wall);

	if (wall.Orientation == WallOrientation.Top)
	{
		AudioManager.Play("sfxApplause");
		this.Screen.RemoveEntity(this);
		Arcade2DGame.GetPlayer(PlayerIndex.One).Score += 1;
		Ball ball = CreateAndInitializeBall();
		this.Screen.AddEntity(ball);
	}

	if (wall.Orientation == WallOrientation.Bottom)
	{
		AudioManager.Play("sfxApplause");
		this.Screen.RemoveEntity(this);
		Arcade2DGame.GetPlayer(PlayerIndex.Two).Score += 1;
		Ball ball = CreateAndInitializeBall();
		this.Screen.AddEntity(ball);
	}

	if (wall.Orientation == WallOrientation.Left)
	{
		this.BounceAgainstWall(wall, 0f, 1f, 1f);
	}

	if (wall.Orientation == WallOrientation.Right)
	{
		this.BounceAgainstWall(wall, 0f, 1f, 1f);
	}
}

private static Ball CreateAndInitializeBall()
{
	Ball ball = new Ball();
	ball.Position = new Vector3(
		Arcade2DGame.Resolution.Width / 2,
		Arcade2DGame.Resolution.Height / 2, 0);

	int velX = FlatRedBallServices.Random.Next(200, 351);
	int velY = FlatRedBallServices.Random.Next(200, 351);

	int xModifier = FlatRedBallServices.Random.Next(0, 2) == 0 ? 1 : -1;
	int yModifier = FlatRedBallServices.Random.Next(0, 2) == 0 ? 1 : -1;

	velX *= xModifier;
	velY *= yModifier;

	ball.Velocity = new Vector3(velX, velY, 0);
	return ball;
}
```

Figure 61 – Generated code for wall collision reactions

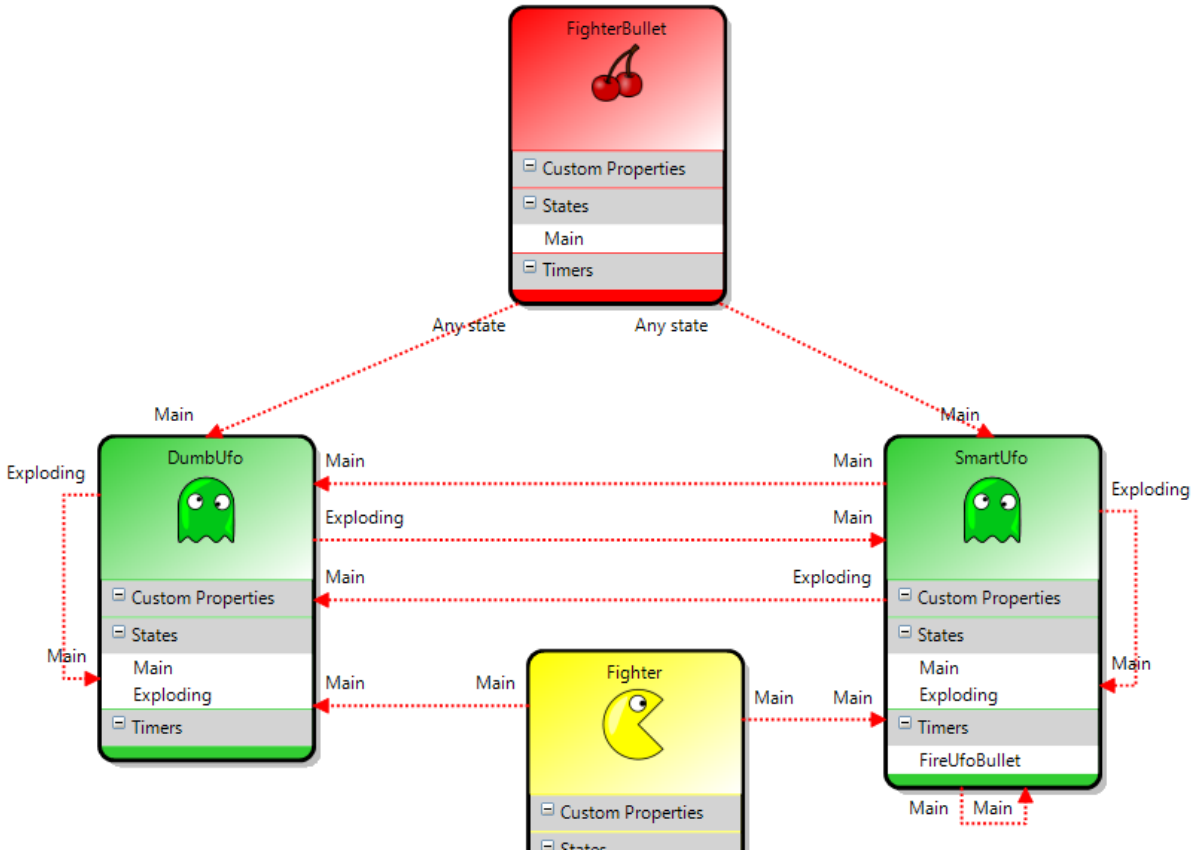


Figure 62 – Declaring collision interests that depend on the entities' states

Considering the *ratio between the generated code and the total game code* as the evaluation metric, games developed with the first version of ArcadEx had 75% of their code automatically generated by the SPL. We had to implement the remaining 25% as SPL extensions because some sub-domains were not initially automated, such as wall collisions, score-based events, and initialization of entity properties with random values. Once we retrofitted such extensions into subsequent versions of the game SPL, the number approached 100%. If new games have unanticipated variability, i.e., if they require behaviors not supported as built-in by the game SPL, then the number will drop again.

The reduced level of flexibility in the behavior of the generated games as is (with no extensions), due to increased abstraction levels, was observed as the approach's major drawback. As opposed to click-n-play tools though, extensibility hooks with full development-environment support and integration are provided for custom behaviors. That way, unpredicted behaviors can still be programmed by hand and integrated to the models as extensions. As previously mentioned, many of these extensions were then incorporated to the game SPL's built-in feature set in later iterations.

```
// Handling collision between entities of type Fighter and DumbUfo/SmartUfo.
for(int sourceCount = 0; sourceCount < fighters.Count; sourceCount++)
{
    Fighter fighter = fighters[sourceCount];
    for(int targetCount = 0; targetCount < dumbUfos.Count; targetCount++)
    {
        DumbUfo dumbUfo = dumbUfos[targetCount];
        if (fighter.CollideAgainst(dumbUfo))
        {
            if (fighter.CurrentState is FighterStates.Main
                && dumbUfo.CurrentState is DumbUfoStates.Main) { ... }
        }
    }
    for (int targetCount = 0; targetCount < smartUfos.Count; targetCount++)
    {
        SmartUfo smartUfo = smartUfos[targetCount];
        if (fighter.CollideAgainst(smartUfo))
        {
            if (fighter.CurrentState is FighterStates.Main
                && smartUfo.CurrentState is SmartUfoStates.Main) { ... }
        }
    }
}
// Handling collision between entities of type FighterBullet and DumbUfo/SmartUfo.
for(int sourceCount = 0; sourceCount < fighterBullets.Count; sourceCount++)
{
    FighterBullet fighterBullet = fighterBullets[sourceCount];
    for(int targetCount = 0; targetCount < dumbUfos.Count; targetCount++)
    {
        DumbUfo dumbUfo = dumbUfos[targetCount];
        if (fighterBullet.CollideAgainst(dumbUfo))
        {
            if (dumbUfo.CurrentState is DumbUfoStates.Main) { ... }
        }
    }
    for (int targetCount = 0; targetCount < smartUfos.Count; targetCount++)
    {
        SmartUfo smartUfo = smartUfos[targetCount];
        if (fighterBullet.CollideAgainst(smartUfo))
        {
            if (smartUfo.CurrentState is SmartUfoStates.Main) { ... }
        }
    }
}
// Handling collision between entities of type DumbUfo and SmartUfo.
for(int sourceCount = 0; sourceCount < dumbUfos.Count; sourceCount++)
{
    DumbUfo dumbUfo = dumbUfos[sourceCount];
    for(int targetCount = 0; targetCount < smartUfos.Count; targetCount++)
    {
        SmartUfo smartUfo = smartUfos[targetCount];
        if (dumbUfo.CollideAgainst(smartUfo))
        {
            if (dumbUfo.CurrentState is DumbUfoStates.Exploding
                && smartUfo.CurrentState is SmartUfoStates.Main) { ... }

            if (smartUfo.CurrentState is SmartUfoStates.Exploding
                && dumbUfo.CurrentState is DumbUfoStates.Main) { ... }

            if (smartUfo.CurrentState is SmartUfoStates.Main
                && dumbUfo.CurrentState is DumbUfoStates.Main) { ... }
        }
    }
}
```

Figure 63 – Implementing collision interests manually for the 2942 game (1/2)

```
// Handling collision between entities of type DumbUfo and SmartUfo.
for(int sourceCount = 0; sourceCount < dumbUfos.Count; sourceCount++)
{
    DumbUfo dumbUfo = dumbUfos[sourceCount];
    for(int targetCount = 0; targetCount < smartUfos.Count; targetCount++)
    {
        SmartUfo smartUfo = smartUfos[targetCount];
        if (dumbUfo.CollideAgainst(smartUfo))
        {
            if (dumbUfo.CurrentState is DumbUfoStates.Exploding
                && smartUfo.CurrentState is SmartUfoStates.Main) { ... }

            if (smartUfo.CurrentState is SmartUfoStates.Exploding
                && dumbUfo.CurrentState is DumbUfoStates.Main) { ... }

            if (smartUfo.CurrentState is SmartUfoStates.Main
                && dumbUfo.CurrentState is DumbUfoStates.Main) { ... }
        }
    }
}

// Handling collision between entities of type DumbUfo and DumbUfo.
for(int sourceCount = 0; sourceCount < dumbUfos.Count; sourceCount++)
{
    DumbUfo dumbUfo1 = dumbUfos[sourceCount];
    for(int targetCount = sourceCount + 1;
        targetCount < dumbUfos.Count; targetCount++)
    {
        DumbUfo dumbUfo2 = dumbUfos[targetCount];
        if (dumbUfo1 != dumbUfo2 && dumbUfo1.CollideAgainst(dumbUfo2))
        {
            if (dumbUfo1.CurrentState is DumbUfoStates.Exploding
                && dumbUfo2.CurrentState is DumbUfoStates.Main) { ... }
        }
    }
}

// Handling collision between entities of type SmartUfo and SmartUfo.
for(int sourceCount = 0; sourceCount < smartUfos.Count; sourceCount++)
{
    SmartUfo smartUfo1 = smartUfos[sourceCount];
    for(int targetCount = sourceCount + 1;
        targetCount < smartUfos.Count; targetCount++)
    {
        SmartUfo smartUfo2 = smartUfos[targetCount];
        if (smartUfo1 != smartUfo2 && smartUfo1.CollideAgainst(smartUfo2))
        {
            if (smartUfo1.CurrentState is SmartUfoStates.Exploding
                && smartUfo2.CurrentState is SmartUfoStates.Main) { ... }

            if (smartUfo1.CurrentState is SmartUfoStates.Main
                && smartUfo2.CurrentState is SmartUfoStates.Main) { ... }
        }
    }
}
```

Figure 64 – Implementing collision interests manually for the 2942 game (2/2)

Another challenge that we faced with the approach relates to backward compatibility. More than once, new versions of the DSLs (still under the same case study) broke existing model instances. Since we did not develop migration tools to update models to the new versions of the DSLs, we had to update them manually.

Considering *development effort improvement* as the evaluation metric, we took time measurements for the development of games belonging to the case study's domain, with two versions implemented for each game: one using the ArcadEx toolset and the other consuming the game engine alone. We observed that ArcadEx games are developed in one-fourth to one-fifth of the time required to develop them using the game engine. For example, the development of the Money Hunt game (Figure 45, top-left) using the ArcadEx SPL took 33 minutes. We also developed the same game without the ArcadEx SPL, by using only XNA and the FlatRedBall game engine. Although we were more used to the game rules and logic from the first (ArcadEx) version, the second one took **4.88** times more: 2 hours and 41 minutes.

Some of the implementation tasks took an order of magnitude less when the ArcadEx SPL was used (e.g., creating collision detection handlers and updating the state of entities), while others did not require any effort in ArcadEx since they are already provided as built-in, such as the creation of screen borders (walls) to contain entities. It is also worth noticing that standard code styling and documentation were left out of the scope of Money Hunt's manual implementation, although they are enforced by generators. The introduction of such additional requirements would lead to a bigger development time.

On the other hand, no improvement was evidently observed for tasks which are not supported by the ArcadEx SPL, such as a custom difficulty increase logic required by Money Hunt. In fact, our time measurements contrasting the two game versions revealed that such special tasks actually take longer with the game SPL, since hooks from the models to the custom code have to be established and an initial overhead is required to implement the extensibility mechanisms, such as adding partial classes and applying the double-derived design pattern (see Subsection 4.5.4).

The manual implementation of Money Hunt required a total of 596 lines of code, ignoring comments and whitespace. The implementation of the same game using the ArcadEx SPL resulted in 5 diagrams (1 GameDefinitionDSL diagram, 2 InputMappingDSL diagrams, 1 ScreenDSL diagram and 1 EntityDSL diagram) which generated 760 lines of code that had to be complemented by 25 lines of manually implemented code (developer-added extensions). Therefore, the custom code corresponds to 3.3% of the total code when considering the generated code, or to 4.2% when considering the code implemented in the manual-only version. Either option leads to the conclusion that **more than 95% of the code was automatically generated** by the game SPL.

Data obtained from implementing other game samples corroborate with the code generation ratio and development effort (time) numbers obtained for Money Hunt. For example, we also implemented Pong (Figure 45, top-right), which was defined as having a similar

screen flow when compared to Money Hunt, but less complex entities and events. The development of Pong using the ArcadEx SPL took 16 minutes. The manual implementation of the same game, by using only XNA and the FlatRedBall game engine, took **4.56** times more: 1 hour and 13 minutes.

Our previous experience with the manual Money Hunt implementation improved the effort to manually accomplish some tasks in Pong, such as initializing the game, creating screen walls and displaying texts. As opposed to Money Hunt though, Pong development with ArcadEx did not require any custom-added code. We were able to obtain a full code generation (100%) of Pong's source code with the game SPL assets.

A valid concern about using SPL techniques in the digital games domain is whether they threaten the generated games' creativity and distinctness. So far, our results actually indicate that automating the routine and error-prone activities in the game-development process (the "commonality") let us spend more time and resources on the domain's variability and extension points, contributing to the uniqueness of each title. In fact, game engines have already been responsible for myriad creative, unique industrial titles. Similarly, we do not suggest end-to-end game generators; rather, we recommend layering SPLs and DSLs on top of game engines so that the software reuse is more structured, effective, and intuitive.

5.4 The Experimental Study

Although the exploratory and confirmatory case studies provided relevant data about the proposed approach, the major weakness of case studies is that the data collection and analysis is open to interpretation and researcher bias [Easterbrook et al., 2007]. In order to overcome that, we used the methodology suggested by Wohlin et al. [1999] to also run a controlled experiment to evaluate the proposed Domain-Specific Game Development approach. The methodology suggests experiments to be broken down into five main activities. The *definition* activity defines the experiment in terms of problem, objective and goals. *Planning* determines the design of the experiment, considers instrumentation and evaluate its threats. The *operation* collects measurements, which are then analyzed and evaluated in the *analysis and interpretation* activity. Finally, the results are presented and packaged in the *presentation and package* activity.

5.4.1 Definition

According to Wohlin et al. [1999], there are two kinds of variables in an experiment: *independent* and *dependent*. The variables that are objects of the study, which are necessary to study to see the effect of the changes in the independent variables, are called **dependent variables**. Often there is only one dependent variable in an experiment. All variables in a

process that are manipulated and controlled are called **independent variables**. An experiment studies the effect of changing one or more independent variables. Those variables are called **factors**. The other independent variables are controlled at a fixed level during the experiment, or else it would not be possible to determine if the factor or another variable causes the effect. A **treatment** is one particular value of a factor.

In our experiment, the development effort (time in man-hours required to develop games) is the dependent variable. The independent variables are the development method, the order in which the development method treatments are used, the experience of the subjects and the target games. The development method is the factor of the experiment (independent variable that changes to measure effects on the dependent ones). We employed two possible treatments for it: development using the FlatRedBall game engine only, and development using the ArcadEx game SPL toolset.

Since our experiment also involves other metrics, we also used a Goal-Question-Metric (GQM) template [Basili et al., 1994] to more comprehensively define it. The **goal** of the experiment was to analyze the *Domain-Specific Game Development approach* for *evaluating it* with respect to the *efficiency and difficulties of game SPLs created from it versus using game engines only*, from the point of view of *game developers and designers* in the context of *the development of digital arcade games*.

To achieve this goal, the following **questions** were defined:

- Q_1. Does the approach enable the generation of arcade games faster than the state-of-the-art (game engines)?
- Q_2. Does the approach minimize the amount of manual code required to develop arcade games?
- Q_3. Do the subjects consider that the approach aids in game development and design?
- Q_4. Do the subjects have difficulties to use game SPLs created with the approach?

Four **metrics** were defined to support the questions above. The first two are objective and obtained by measurements performed during the experiment, while the last two are subjective and obtained via questionnaires.

- **Development effort improvement:** the difference between the time required to develop sample games without the approach (but still with the state-of-the-art, i.e., game engines), and the time required time to develop the same games with the approach.
- **Generated/total code ratio:** average % of the total game code that was able to be generated automatically from the DSLs.

- **Helpfulness:** average score for the helpfulness of game SPL assets, created with the approach, to the game development process, as evaluated by subjects, in a scale from 1-10, in which 1 means *not helpful* and 10 means *very helpful*.
- **Difficulty:** average score for the difficulty of employing game SPL assets, create with the approach, in the game development process, as evaluated by subjects, in a scale from 1-10, in which 1 means *very easy* and 10 means *very difficult*.

5.4.2 Planning

The experiment will be conducted with eight experienced software developers (five or more years in the industry), according to the plan outlined in the following subsections. Respecting their availability for the experiment, the subjects will be selected by convenience sampling [Wohlin et al., 1999], representing a non-random subset from the universe of software developers in the industry. Also due to availability constraints, the experiment will not require subjects to be game developers, but software developers in general with some game development experience.

Subjects will initially be briefed about the experiment workflow, what they are supposed to do in the experiment and how to consult the experiment resources (more details in Subsection 5.4.2.3, Instrumentation). The experiment sessions will start with a 2h training session, as described next.

5.4.2.1 Training

Subjects will be trained in two fronts:

- How to develop arcade games using the FlatRedBall game engine and Microsoft XNA, culminating with the creation of a simplified version of the Money Hunt game (Figure 45, top-left) with no models or code generation.
- How to develop arcade games with the ArcadEx game SPL, including its DSLs and the ArcadEngine layer that promoted FlatRedBall to ArcadEx's domain framework, culminating with the creation of a simplified version of the Money Hunt game via ArcadEx's DSLs and generators. All four DSLs will be explored as part of the training: GameDefinitionDSL, InputMappingDSL, ScreenTransitionDSL and EntityDSL.

5.4.2.2 Subject Groups and Target Games

Subjects will be randomly organized in two distinct groups, responsible for implementing a total of 32 versions of the same games, Pong and 2942 (Figure 45, top-right and bottom-right

respectively), which are representative samples of the domain yet not too complex, making the experiment's cost viable. Each subject group will use both approaches (manually and via ArcadEx), but in different order, as described in Table 8.

Table 8 – Subject groups

Group 1	Group 2
Pong (ArcadEx)	Pong (manually)
Pong (manually)	Pong (ArcadEx)
2942 (manually)	2942 (ArcadEx)
2942 (ArcadEx)	2942 (manually)

By randomly allocating the subjects according to the permutations planned in Table 8, some threats related to the experiment are mitigated. For instance, the plan ensures that all developers create the same games, using all approaches (with and without the game SPL). This avoids the productivity delta between exceptional developers and underperformers to influence the results. Likewise, by swapping the manual vs. ArcadEx order within each group, we minimize the chances that the familiarity acquired with a game after its first implementation will impact the overall results. More robust approaches to measure the toolset learning impact, such as the Latin Square [Dénes & Keedwell, 1974], were not used due to limitations in the number of the experiment resources, such as the number of subjects vs. the number of possible permutations.

5.4.2.3 Instrumentation

Development time will be measured for each version of the games. Subjects will receive a questionnaire inquiring about their development experience (Appendix C), which also has open questions as a means to collect quantitative data. During the experiment, subjects will be able to consult a XNA/FlatRedBall Cheat Sheet (Appendix D), which is a quick reference for common programming tasks using such an API and game engine. The cheat sheet's goal is to level the different familiarities of the subjects on XNA and FlatRedBall, i.e., it contributes to attenuate the impact of previous XNA/FlatRedBall experience on the experiment.

Game assets such as textures, sound effects and background music will be provided upfront to developers for each game, in a starter solution, in order to keep media creation efforts at a constant level (zero). Game specifications with playable demos will also be provided prior to the experiment, to familiarize developers with their target games.

Finally, checklists will also be provided to the subjects (Appendix E) for each game, listing the required development tasks they will have to complete. A game's checklist is applicable to both the manual implementation scenario (XNA/FlatRedBall only) as well as the

modeling one (with game SPL assets). We considered giving to the subjects sketches of diagrams that describe the games' requirements visually, instead of a checklist. However, such diagrams sketches are too close to the actual DSL models the subjects are supposed to come up with. Therefore, we discarded this option to avoid the diagrams to interfere with the experiment and unfairly benefit the game SPL scenario. An interesting observation is that the time spent to create checklist items for some game development requirements was virtually the same as modeling its solution using the game SPL assets, which is an indication that the game SPL approach is helpful as an "executable specification".

5.4.2.4 Experiment Hypotheses

The **null hypothesis** is the one that the experimenter wants to reject with as high significance as possible. In this study, the null hypothesis determines that the Domain-Specific Game Development approach do not provide benefits that justify its use and that the subjects have difficulties to use the game SPL assets created with it. Thus, according to the selected criteria, the following null hypotheses were defined:

- **H_{0a}: $\mu_{coding\ effort} - \mu_{modeling\ effort} \leq 0$**, i.e., developing sample games with the proposed approach requires the same or more time than developing sample games without the proposed approach.
- **H_{0b}: μ generated/total code ratio < 100%**, i.e., subjects are not able to fully generate code from models.
- **H_{0c}: μ helpfulness < 7**, i.e., the subjects give an average score of less than 7, in a scale from 1 to 10, for the helpfulness of the toolset and its encompassing process. This value is based on an experiment performed by Lisboa et al. [2007] to evaluate a Domain Analysis tool.
- **H_{0d}: μ difficulty >= 2**, i.e., the subjects give an average score equal to or greater than 2, in a scale from 1 to 10, for the difficulty of the toolset and its encompassing process. This value is based on an experiment performed by Almeida [2007] to evaluate a Domain Engineering process.

For the last two (subjective) metrics, the authors recognize that there is no well-known value for them in the literature. Therefore, they chose "arbitrary values based on practical experience and common sense". We decided to keep such values with the purpose of having a comparison baseline.

The **alternative hypothesis** is the one in favor of which the null hypothesis is rejected. In this study, the alternative hypotheses determine that the use of the Domain-Specific Game Development approach produce benefits that justify its use:

- **H_{1a}: $\mu_{coding\ effort} - \mu_{modeling\ effort} > 0$**, i.e., developing sample games with the proposed approach requires less time than developing sample games without the proposed approach.
- **H_{1b}: μ generated/total code ratio = 100%**, i.e., subjects are able to fully generate code from models.
- **H_{1c}: μ helpfulness >= 7**, i.e., the subjects give an average score equal to or greater than 7, in a scale from 1 to 10, for the helpfulness of the toolset and its encompassing process.
- **H_{1d}: μ difficulty < 2**, i.e., the subjects give an average score of less than 2, in a scale from 1 to 10, for the difficulty of the toolset and its encompassing process.

5.4.2.5 Threats to the Validity of the Experiment

To provide a set of valid results, Wohlin et al. [1999] defend the establishment of four types of threats to the validity of the experiment, presented next. This experiment's planning and design aimed at mitigating as many threats as possible. On the other hand, some were not mitigated, either due to being "by design" or due to resource constraints. Wohlin et al. [1999] mention that sometimes some threat to validity has to be accepted, that increasing one type of validity may decrease the other, and that it may even be impossible to carry out an experiment without certain threats.

The **internal validity** of the study is defined as the capacity of a new study to repeat the behavior of the current study, with the same subjects and objects with which it was executed. We identified the following internal validity threats to our experiment:

- History: evolution of subjects' experience with the toolset (models) and game engine (code). In order to mitigate that, subjects were organized in two different groups in which the order of the treatments is different (Table 8).
- Maturation: given the length of the experiment, subjects could get bored or tired over time. We employed multiple sessions with the subjects to avoid such a fatigue.
- Testing: bias caused by knowing the results of the first game implementation and results from other subjects. Subjects were not given such numbers until they completed their own tasks.
- Selection: the subjects have a natural variation in performance. Instead of comparing subjects among themselves, we designed the experiment so that a subject is compared with himself. This also mitigates the "compensatory rivalry" or its opposite "resentful demoralization" threats Wohlin et al. [1999].

The **external validity** measures the capability of the study to be affected by generalization, i.e., the capability to repeat the same study in other research groups. We identified the following external validity threats to our experiment:

- Interaction of selection and treatment: the chosen subjects were not game developers, but software engineers in general. This was a consequence of convenience sampling.
- The subjects belong to a homogeneous group (all belonging to the same culture and working for the same company).
- Subjects' motivation can change. For instance, developers in a game company might be more motivated to be part of the experiment than subjects belonging to an academic environment. Our experiment chose industry professionals with a genuine interest in Software Engineering and reuse.
- Interaction of setting and treatment: we assumed that consuming the FlatRedBall game engine (manual treatment) represents a typical manual development experience. Moreover, the experiment's domain takes into account games of reduced complexity, whose results we may not necessarily be able to generalize to more complex game domains. Finally, the game SPL toolset was developed as part of an academic research, not an industrial (real world) context.
- Since the subjects and the experimenter knew each other from past interactions in the software industry and consequently built some level of relationship, that could bias the experiment's results. In order to mitigate that, the subjects were explicitly guided on providing candid feedback and not taking at all any actions that could bias the experiment, such as entering a higher than actually perceived score in the evaluation form. Given the open and mixed feedback we received from the subjects in multiple areas of the experiment, we believe this guidance was followed properly.
- Environment configuration (machines, training rooms, etc.) can change. For instance, our experiment had a limited number of available machines, properly configured with the experiment environment.
- The knowledge of the instructor who is delivering the experiment training can change.

The **conclusion validity** is concerned with the relationship between the treatment and the outcome, and determines the capability of the study to generate conclusions. The following threats were identified for this validity:

- Limited number of subjects (eight).

- Limited number of implemented games: two (times two) per subject.
- Subjective nature of two of the experiment metrics (Helpfulness and Difficulty feedback from the subjects).
- Random interruptions of a subjects' work (e.g., required pauses during a long coding effort).
- Ensuring the modeling and coding treatments performed by a same subject for the same game resulted in the same end product. The experimenter performed manual inspection to ensure that indeed happened.
- Ensuring the games developed by different subjects given the same specification are the same. The experimenter performed manual inspection to ensure that indeed happened.

Finally, the **construct validity** refers to the relation between the theory that is to be proved and the instruments and subjects of the study, for which we identified the following threats:

- Evaluation apprehension and interaction of testing and treatment: the subjects knew their development time was being effort, and as a result that could make them to feel more receptive or sensitive to the treatment. We attempted to mitigate this threat by explicitly guiding them on working on their code and models as close as possible to what they perform as part of their daily work.
- Subject's inability to understand the game specifications. This was mitigated by providing game checklists and playable versions upfront, and also due to the fact that a relatively easily understandable set of game samples was chosen for the experiment.
- The experiment takes into account one single developer per game, however games are typically created by development teams.
- Convenience sampling may imply in subjects not representing a random subset of the developers' population.

5.4.3 Operation

The profiles of the 8 subjects that participated in the study are presented in Table 9. The subjects present a mix of bachelors' and masters' degrees and 5 to 15 years of software development experience in the industry (8.75 years in average). Half eventually use modeling as part of their daily work, while the other half are just aware of it or have used it only to a limited extent. All of them are proficient in C#, programming language used in the experiment. In average, each of them has developed more than 3 games in the past, and only one subject has never developed one. It is worth noticing that since the experiment and its metrics have a

software engineering focus, the subjects are all software engineers. Experimenting with game designers, artists and other profiles was not taken into account, and constitutes a future work of this research.

Table 9 – Subjects' Profiles

ID	Degree	Software Development Experience	Modeling Experience	# of Previous Games
1	B. Sc.	11 years	Just aware/limited experience	7
2	M. Sc.	7 years	Just aware/limited experience	3
3	B. Sc.	7 years	Eventually use it as part of daily work	0
4	M. Sc.	5 years	Eventually use it as part of daily work	2
5	M. Sc.	7 years	Eventually use it as part of daily work	4
6	B. Sc.	10 years	Eventually use it as part of daily work	2
7	M. Sc.	8 years	Just aware/limited experience	5
8	M. Sc.	15 years	Just aware/limited experience	2

The experiment was conducted during November and December of 2011, and had a total of 14 experiment sessions, each with one or two subjects at a time and adjusted to fit their availability. In the experiment briefing, subjects were trained and given the aforementioned help resources: game checklists (Appendix E) and the cheat sheet (Appendix D). Time was set aside for introducing the requirements of the games and allowing the subjects to play with them upfront. In the post-experiment survey (Appendix C), subjects were asked whether they had any difficulties in understanding the target games to be implemented (or modeled) and their requirements. None of them reported any issues in relation to such a regard.

The sessions totalized a cost of **196 man-hours**, as presented in Table 10. Of those, 24 man-hours were dispended on training, and 172 man-hours on actual modeling and coding, including buffer time for introducing and playing the target games as well as interrup-

tions. Such numbers also consider the time dispended by the experiment organizer who was delivering the trainings and tracking the progress of the modeling and coding activities. Otherwise, the total cost of the operations phase, considering only the subjects' time, comes down to 125 man-hours.

Table 10 – Time dispended during the experiment's operation phase

Activity	Subjects' time (man-hours)	Experimenter's time (man-hours)	Total
Training	16	8	24
Modeling and coding	109	63	172
Total	**125**	**71**	**196**

The experiment did not require any especial hardware beyond two machines and two Xbox 360 gamepads. Discarding other minor costs related to the experiment's logistics, the essence of the experiment's cost was basically related to the aforementioned operation costs plus the hours required from the experimenter for other phases such as definition, planning and analysis, including the development and preparation of resources such as training materials, sound and art assets, installers, packages and starter solutions, as well as dry runs prior to the actual experiments to ensure all resources were in place. Such additional tasks required about 40 man-hours, bringing the total man-hour cost of the experiment to approximately **236 hours**.

Each training session of the experiment resulted in the development of two simplified versions of the Money Hunt game (Figure 45, top-left): one version without the game SPL assets and hence with no models or code generation, the other with the provided DSLs and generators. Data was not collected for Money Hunt since the development of this game was for training purposes. Each subject then developed two versions of the Pong game (Figure 45, top-right) and the 2942 game (Figure 45, bottom-right), with and without the game SPL assets. That resulted in **32 (implementations of) games** created by the subjects. When the experiment was concluded, each subject was requested to complete the questionnaire in Appendix C with their feedback about the experience. The data collected from the time measurements, source code and feedback forms are analyzed and interpreted next.

5.4.4 Analysis and Interpretation

The following subsections describe the analysis and interpretation of the experimental study, according to each one of the metrics under evaluation.

5.4.4.1 Development Effort Improvement

Figure 65 presents the measured development efforts for the Pong game, per subject, in man-minutes, for both treatments (coding and modeling), ordering the subjects by the improvement (highest to lowest). The graph presented in Figure 66 provides a better visualization of the development efforts ratio: the manual implementation time is presented in the X-axis and the modeling time is presented in the Y-axis. The numbers inside the diamonds are the subject IDs.

As it can be noticed, the development times for all subjects are compliant to the hypothesis H_{1a} ($\mu_{coding\ effort} - \mu_{modeling\ effort} > 0$) and the majority of them are at or under the 4x development efforts ratio line. For the Pong game, the average development effort improvement was **136.75 man-minutes**, with a standard deviation of **42.41 man-minutes**. The average modeling time with the game SPL was 39.5 minutes, while its average manual implementation time was 176.25 minutes. The average development efforts ratio was **4.59**. In other words, the experiment suggests that developing games with the SPL assets is 4.59 times faster than implementing the same games without them. It is worth noticing that the calculation of the average development efforts ratio should not be done by dividing the average manual implementation time by the average modeling time, but by averaging the subjects' development efforts ratios. Otherwise, the smaller times from faster subjects become outweighed (shadowed) by the bigger times from slower subjects, both for coding and modeling.

Similarly, Figure 67 and Figure 68 present the development efforts required by the subjects for the 2942 game. For such a game, the average development effort improvement was **263.375 man-minutes**, with a standard deviation of **77.41 minutes**. The average development efforts ratio was **5.57**. The average modeling time with the game SPL for this game was 55.375 minutes, while the average manual implementation time for it was 318.75 minutes.

Despite the expressive values obtained for the averages and standard deviations, we employed hypothesis testing in order to be able to tell whether the obtained numbers are able to reject the null hypothesis ($\mathbf{H_{0a}: \mu_{coding\ effort} - \mu_{modeling\ effort} \leq 0}$) in favor of its alternative counterpart ($\mathbf{H_{1a}: \mu_{coding\ effort} - \mu_{modeling\ effort} > 0}$) with statistical significance. For example, in the Pong game, the measured mean for the development time effort differences is 136.75 man-minutes. We want to rule out chance as an explanation for such results. In other words, we want to know the probability of chance alone to produce a difference as large or larger than 136.75 man-minutes. If such a probability, also called p-value, is less than a significance level (traditionally, researchers use either the 5% or the 1% significance), we can then conclude that the experimental treatment (using the game SPL assets) has a real effect [Lane, 1999]. We will assume a significance value of 0.01 (1%).

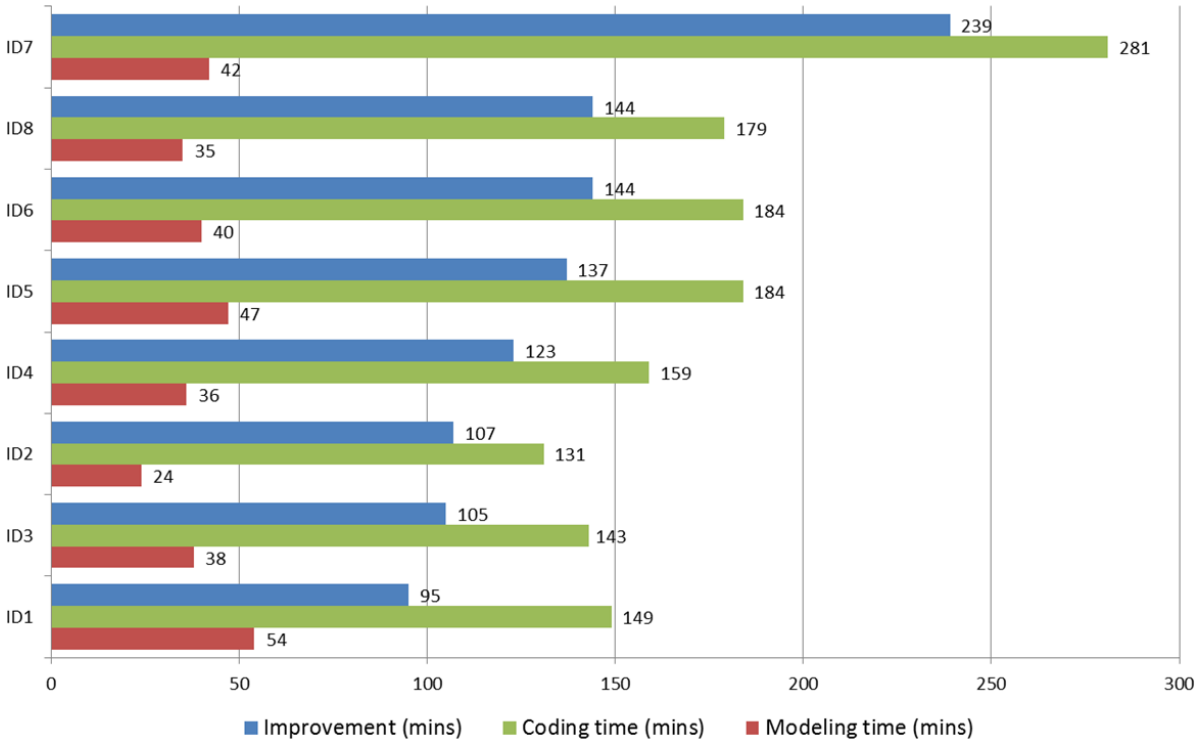


Figure 65 – Development efforts per subject (Pong)

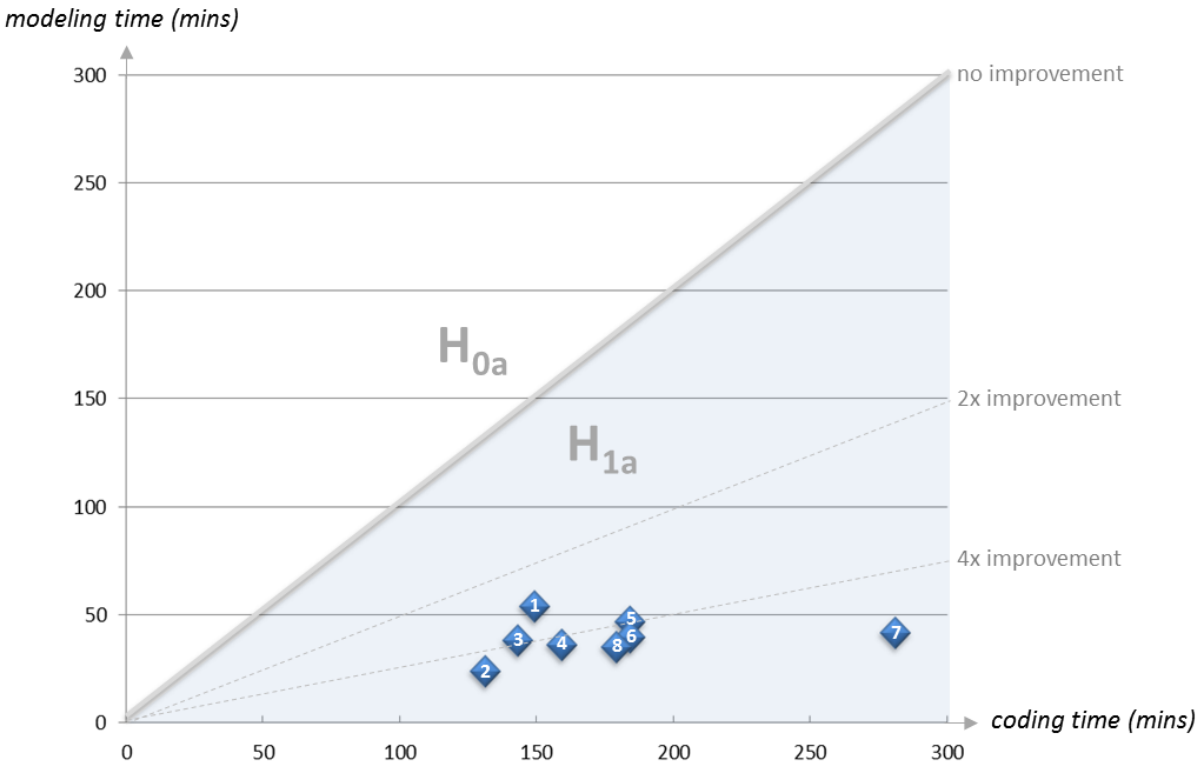


Figure 66 – Distribution of development efforts (Pong)

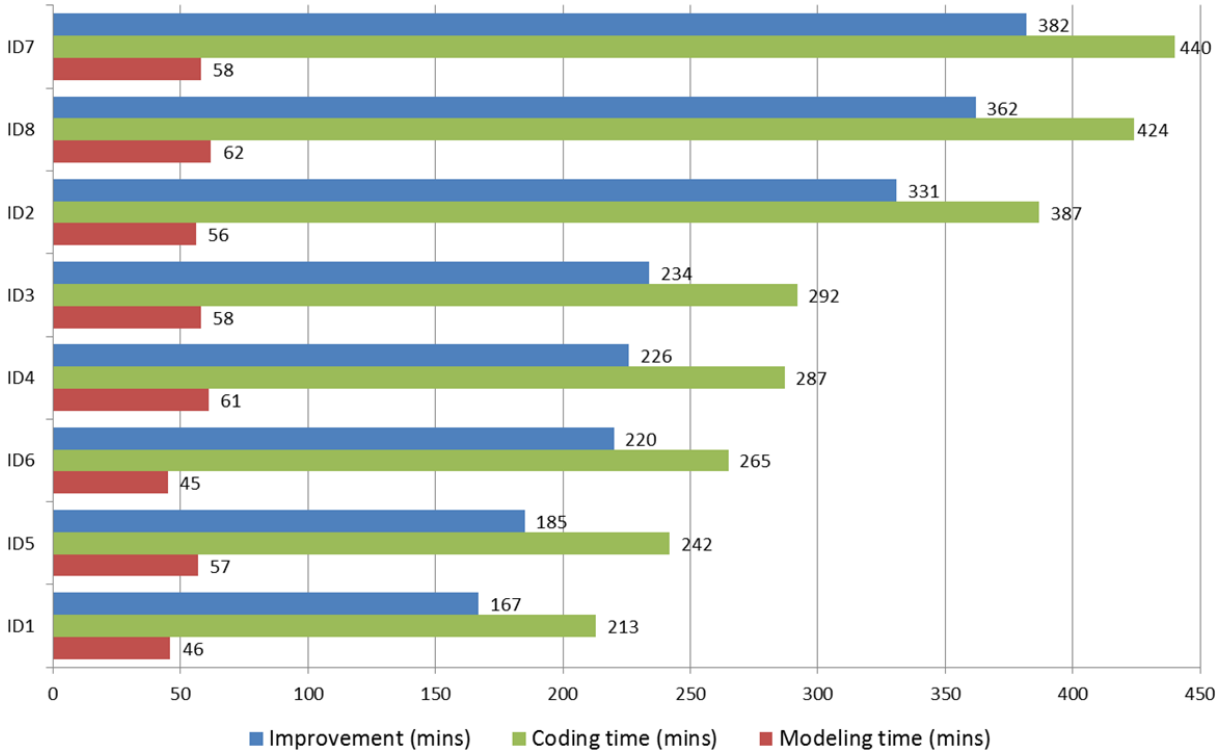


Figure 67 – Development efforts per subject (2942)

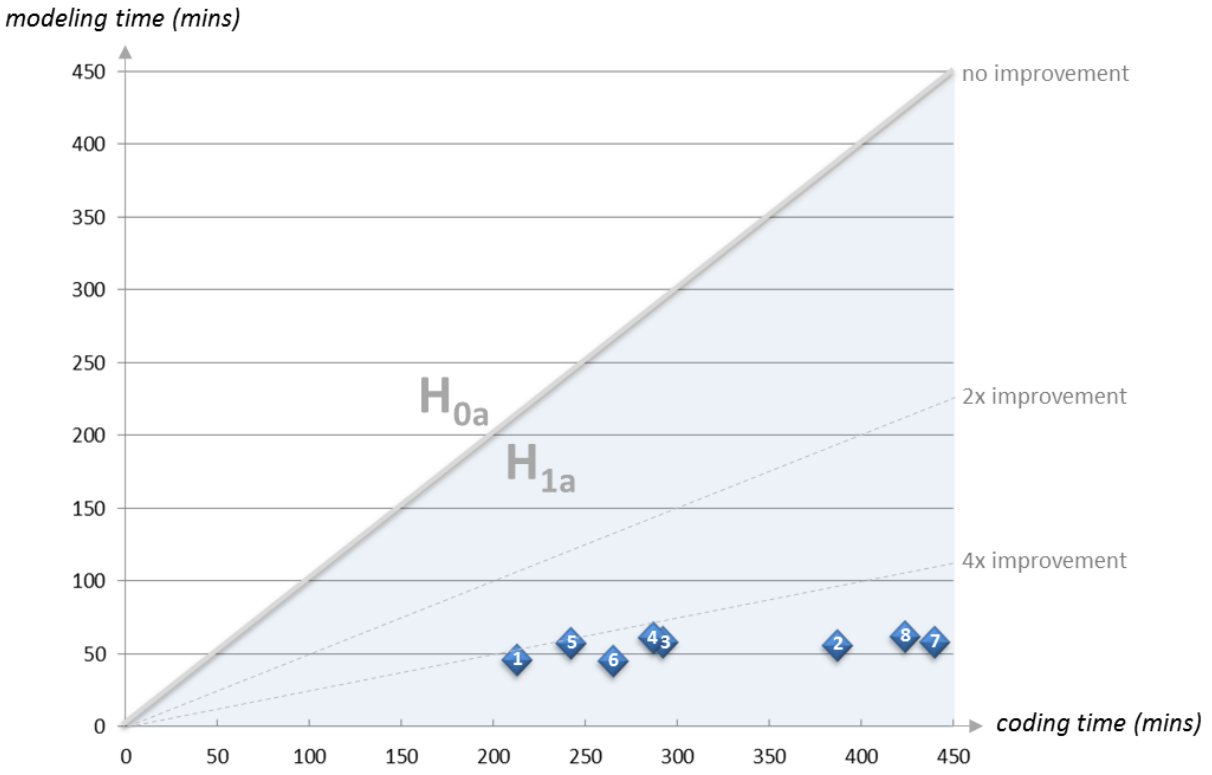


Figure 68 – Distribution of development efforts (2942)

Since the actual population's standard deviation is unknown, an *estimated* standard deviation has to be calculated from the sample size and the obtained data. Its formula gives a 45.33 value for the Pong game (implying in an estimated standard error of 16.03) and 82.76 for the 2942 game (implying in an estimated standard error of 29.26). Given the number of degrees of freedom is 7 (the number of measurements from subjects minus one) and applying the formula of the **t** distribution, which is typically used instead of the normal distribution if the standard error has to be estimated from the data [Lane, 1999], a p-value of less than the 0.01 significance level is obtained. The same result is obtained for the 2942 game. This **rejects** the null hypothesis (H_{0a}) in favor of its counterpart (H_{1a}).

Subjects whose ID is odd belong to Group 1 (see Table 8), i.e., when developing the Pong game, they were applied to the modeling treatment first and the coding treatment later, then the opposite happened for the 2942 game. Subjects whose ID is even belong to Group 2, i.e., when developing the Pong game they were applied to the coding treatment first and the modeling treatment later, then the opposite happened for the 2942 game. No conclusions can be drawn as for how the group allocation influenced the results, not only because each group contained different subjects, but also due to the measured data. Receiving the modeling treatment before the coding treatment, or vice-versa, did not imply in a consistent better performance when groups were compared.

An interesting observation from the measured times is that the results are more sparsely distributed in the manual implementation scenario (X-axis of Figure 66 and Figure 68) than the modeling scenario (Y-axis of the same figures). In other words, the delta between the biggest and smallest development times for the modeling scenario is smaller than the delta between the biggest and smallest development times for the manual implementation scenario. Restricted to the experiment constraints, we interpret that as an indication that Domain-Specific Game Development could improve the predictability of the game development effort, contributing to more reliable estimations.

Aligned with the results obtained by Hernandez & Ortega [2010] in their small experiment to evaluate a modeling language for 2D games, the data obtained from our experiment also make us believe that the approach becomes more effective as the target games become more complex, restricted to this experiment's constraints and assuming of course that the game SPL assets are able to automate the complex sub-domains. Figure 69 presents the evolution of the development efforts ratio from Pong, a simpler game, to 2942, a more complex game. In 2942, subjects had to manually implement more challenging game behaviors not present in Pong, such as a scrolling background and making one entity to chase another, which on the other hand are provided as built-in by the game SPL assets. As a result, *all* sub-

jects reported a better development efforts ratio in the second, more complex game (a 27.29% increase, in average).

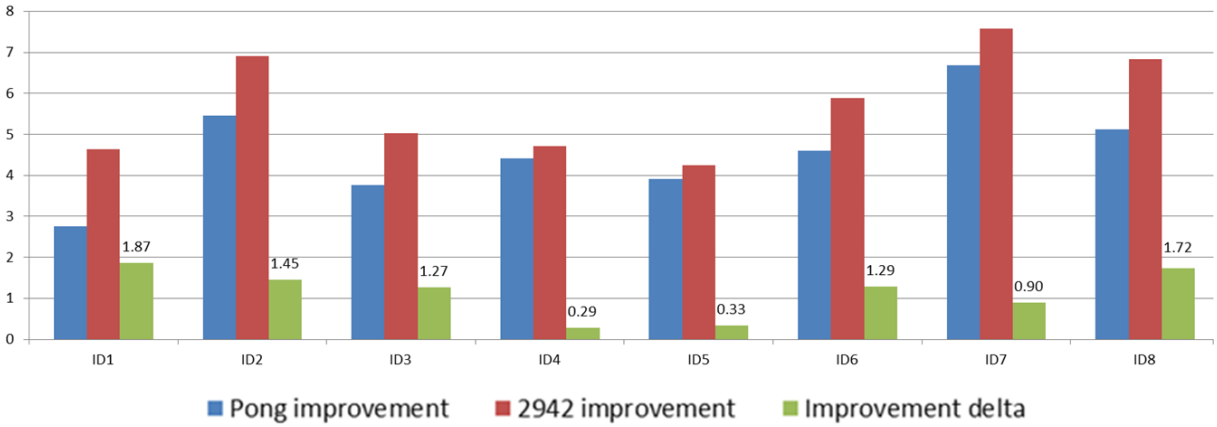


Figure 69 – Evolution of development efforts ratio

During the experiment, we were able to notice that the development effort improvement metric is quite influenced by another one: code quality. For the manual implementation scenarios, we guided developers on creating high quality and maintainable code, as they normally would, instead of rushing with “quick and dirty” code just to finish the experiment faster. As a result, we noticed that many developers employed good Software Engineering practices such as refactoring code in order to abstract, modularize and clean it up. On the other hand, developers still acknowledged that the experiment inhibited them from spending too much time on some quality tasks they might have performed in a real environment, such as unit testing, commenting and studying what design patterns they could apply to implement some aspects of the game more elegantly, such as screen transition triggers. For instance, one of the subjects mentioned that he would have probably coded more carefully if he was supposed to undergo a code review by his team at work. Such a trend was more evident when subjects moved from Pong to the more complex 2942 game: while Pong’s simplicity encouraged subjects to keep its code quality at best, 2942’s complexity required more quality vigilance (e.g., to avoid code duplication). As a result, some subjects explicitly recognized that as time went by, the manual implementation quality decreased: they eventually got more concerned with getting 2942 implemented than spending too much time on its code quality. We believe that if quality tasks were fully employed, for all games, more time would have been spent on the manual implementation, causing the development efforts ratio to be increased.

Considering both games, the average development efforts ratio was **5.16**. A question that naturally arises then is how such a result compares with measurements of domain-specific development effectiveness in other domains. Summarizing the Domain-Specific Development evaluations presented in Appendix B, Figure 70 compares the our measured ef-

fectiveness for the digital games domain with evaluations in the literature for other domains [Kärnä et al., 2009] [Safa, 2007] [Kelly & Tolvanen, 2008] [MetaCase, 2000] [Weiss & Lai, 1999].

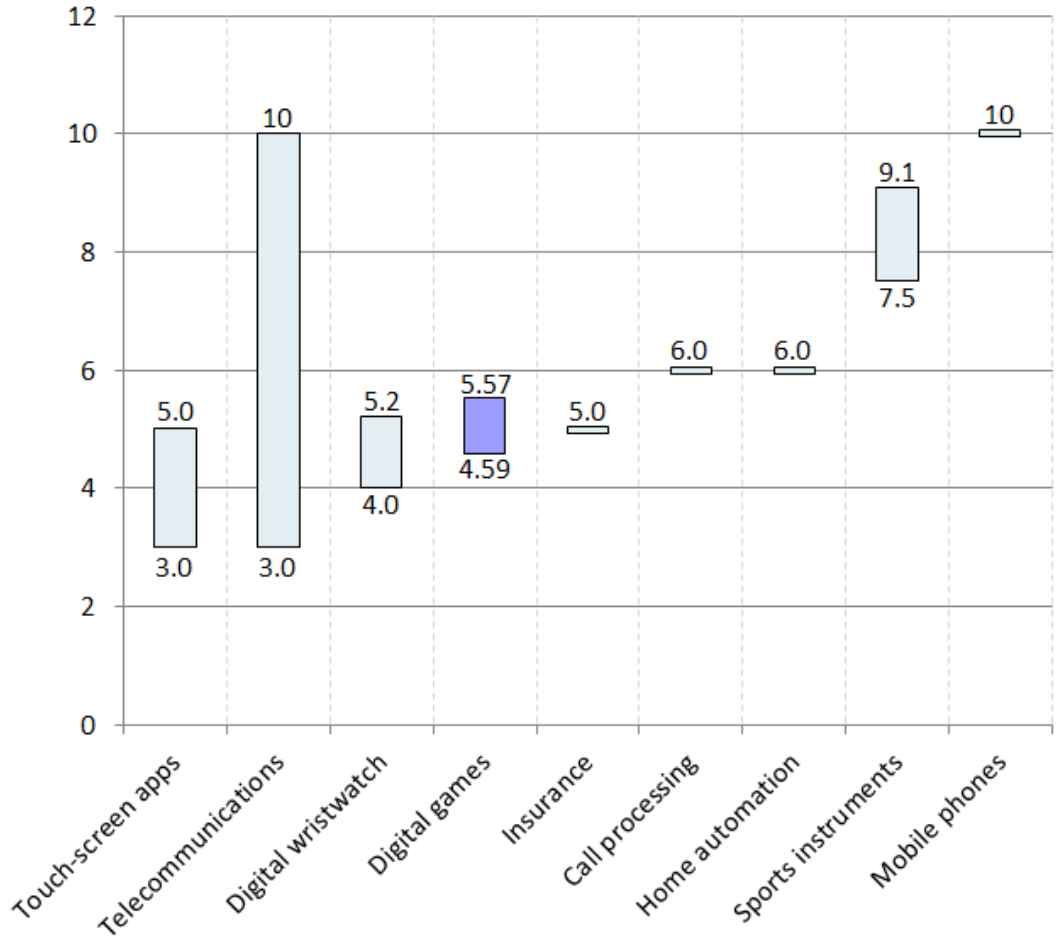


Figure 70 – Comparison of domain-specific development improvements per domain

It is worth noticing that many of the numbers above were not the result of controlled experiments, which are actually not easy to find in the SPL and DSL literatures. Hermans et al. [2009] mention that the DSL literature primarily provides *anecdotal* evidence for the claimed DSL usage benefits, often based on a handful of usage scenarios for the language in question. Kärnä et al. [2009], on the other hand, point out that many good scientific research methods are simply too expensive and time-consuming for practical use in a commercial setting, while the industry does not usually have the time and resources to conduct extensive analysis such as building the same system twice with different development approaches. In fact, one of the authors whose results are displayed in Figure 70 recognizes that their sample size is too small to be statistically significant. Other used estimates instead of measurements. Another inferred the development effort improvement from the percentage of the generated code. Finally, one of the results was solely based on the informal feedback

from the CTO of the company in which the domain-specific solution was employed. Nevertheless, in the lack of more rigorously measured data, we consider the numbers in Figure 70 to be a valid baseline for comparison purposes. More details on each of such numbers are presented in Appendix B.

Based on their experience as domain-specific modeling tool vendors, Kelly & Tolvanen [2008] conclude that the "normal" (expected) range of improvement for the use of domain-specific development in the industry is 5 to 10 times. That is somewhat aligned with previous findings from Weiss & Lai [1999], who reported productivity improvements by 3 to 10 times, depending on the product. The average improvement factor observed in the experimental study performed for Domain-Specific Game Development, **5.16**, falls under such an expected range. The fact that our numbers are in the bottom half is an indication that there might still be room for improvement. We believe that addressing the difficulties reported in Subsection 5.4.4.4 could contribute to improve the measured numbers. In fact, one of the greatest lessons of the experiment was that using domain experts and end-users to validate the DSLs and to obtain early user experience feedback is as relevant as using them as input to language design.

A final question related to the development effort improvement and subsequent savings is what the ROI is for employing our Domain-Specific Game Development approach. Reaching this number is not straightforward, since it depends not only on already known variables, but also on some unknown ones. From one side, we already know the improvement factor (4.59 to 5.57) and the time it took to implement the game SPL assets: about 8 days in our case, which is in line with a similar pedagogical effort performed by Kelly & Tolvanen [2008] for the digital wristwatches development domain and a couple of days less than some industrial cases compiled by Kelly [2010]. On the other hand, estimating the ROI also depends on how long it would take to manually implement the next domain instances (games) chosen by the game SPL users (game developers and designers). The longer it takes to manually implement domain instances, the higher are the absolute savings per instance, and therefore the sooner the investment will break even.

Using as the next domain instances hypothetical games whose complexity is similar to the 2942 game, which is a better representative of the domain than the simpler Pong game also used in the experiment, we can consider an improvement effort of 5.57 and an average manual implementation time of 318.75 minutes (both measured by the experiment). Such numbers mean that when each one of such hypothetical games is developed with the game SPL assets, there are absolute savings of 261.52 minutes (318.75 – 318.75/5.57). Given the game SPL assets were developed in about 8 days, or 64 hours, the investment break-evens **by the 15th instance**.

We ran the numbers again considering that the complexity of the domain instances (i.e. their manual implementation effort) is actually bigger than the one measured for the experiment's games. This is not a stretch given such games could not be made too much complex due to the experiment's constraints (see Subsection 5.4.2.5, *Threats to the Validity of the Experiment*). If we assume it takes a day (8 hours) for a domain instance to be implemented, then the absolute savings per instance come up to 393.82 minutes, making the investment to break-even **by the 10th instance**. This is illustrated in Figure 71.

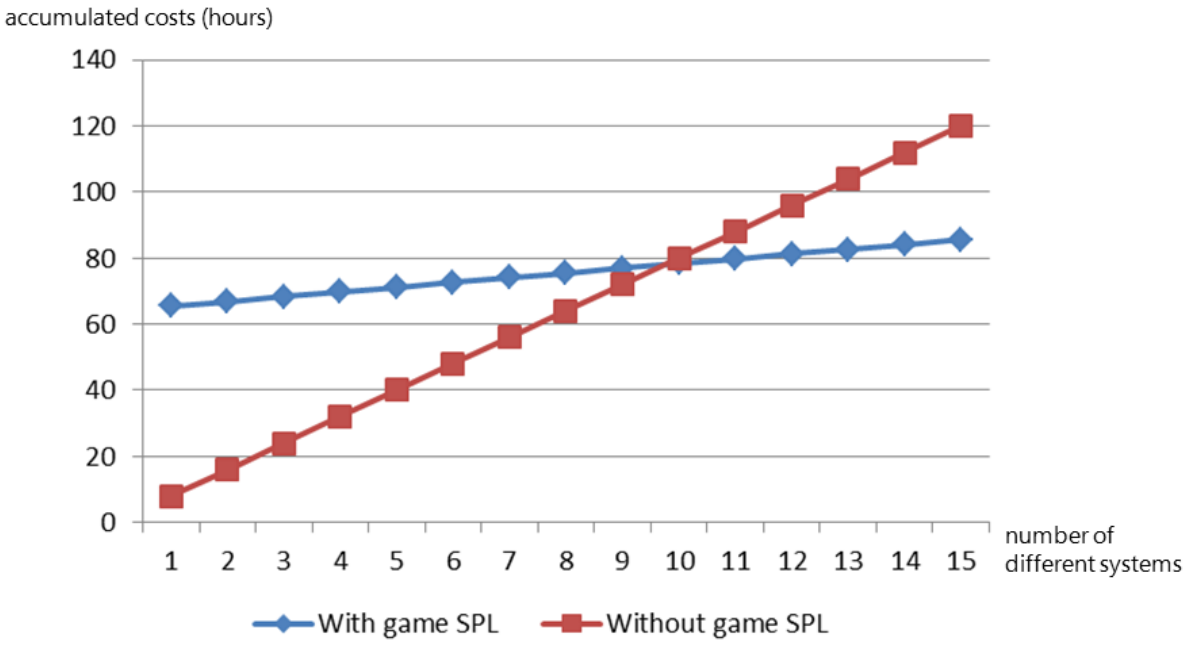


Figure 71 – Costs of developing single game instances vs. the use of game SPL assets

An interpretation of such a result, restricted to the experiment's constraints, is that Domain-Specific Game Development is not recommended for families of games targeted at less than 10-15 instances. On the other hand, other data indicates that the ROI numbers might improve for (more complex) domains approached by the actual game industry. As Figure 69 presents and also based on the survey feedback from the subjects, the development efforts ratio gets better as the complexity of the target games increase. Kelly & Tolvanen [2008], who have experience in deploying domain-specific solutions for multiple domains in the industry, also observed the same trend.

Likewise, the required time for manually implementing the game samples of our chosen domain is considerably small, which also contributes to bring the ROI down. While in our domain we estimated the worst manual implementation time to take a single day, in the actual game industry the development of a game can actually take several months, or even years [Reyno & Cubel, 2008]. Although such a timeframe includes a series of tasks that are out of the scope of our Domain-Specific Game Development approach, ranging from scripting the

game plot to fine-tuning the game with beta testers, actual Software Engineering tasks such as designing and coding still play a big role in it. Increased manual implementation times per domain instance might improve the ROI of a SPL-based approach, since such bigger efforts mean much more costs and impact to one-off development approaches than to SPL-based ones. On the other hand, the investment to create SPLs and DSLs for more complex domains will also be considerably higher.

Finally, organizations are not required to build their game SPL assets upfront to only then start harvesting the benefits. Extractive SPL approaches [Krueger, 2001], which reuse existing products toward the creation of a SPL, can be used to amortize upfront investment costs across time.

5.4.4.2 Generated/Total Code Ratio

For both games, subjects were able to generate 100% of the source game code from models. While this **rejects** the null hypothesis (H_{0b}) in favor of its counterpart (H_{1b}), it is important to emphasize that this metric is strongly dependent on the games chosen by the experiment. Domain games other than Pong and 2942, with higher variability and unpredicted SPL behaviors, will yield to lower numbers. Hence, although this metric was useful to confirm that the subjects were able to reach high levels of automation on their own, corroborating our own conclusions in the case studies, we acknowledge that more confidence on this metric can still be further achieved by broadening the scope of the experiment to more games, potentially chosen by the subjects themselves or by an actual game development company interested in running the experiment in loco.

5.4.4.3 Helpfulness

In a scale from 1 to 10, in which 1 means not helpful and 10 means very helpful, subjects provided an average score of **9.125** for the helpfulness of the game SPL assets. This is above the expected helpfulness score (7.0), therefore it also **rejects** the null hypothesis (H_{0c}) in favor of its counterpart (H_{1c}). The helpfulness feedback score per subject is presented in Figure 72.

Open feedback revealed some interesting data points about the helpfulness of the game SPL assets. Reflecting on the results obtained above for this metric, this subsection will detail the positive feedback and outcomes of the game SPL. On the other hand, the next subsection (which discusses the *Difficulty* metric) will focus on its challenges and areas of improvement.

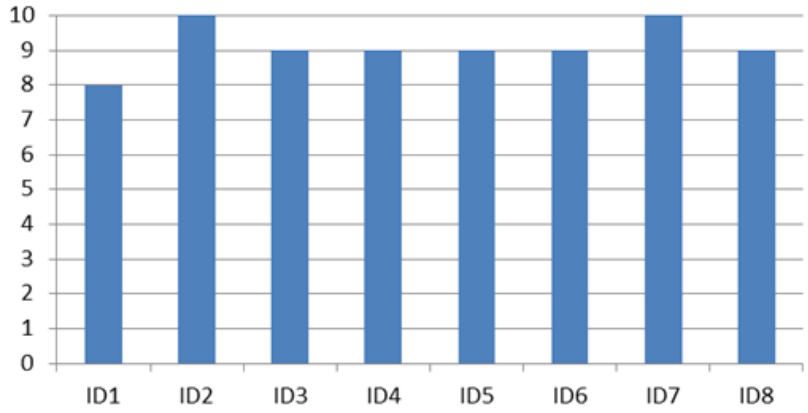


Figure 72 – Helpfulness evaluation per subject

Some open feedback from the subjects about the helpfulness of the game SPL assets, collected via the post-experiment survey, follows below:

- *"More repetitive tasks like setting backgrounds of a screen can be easily mastered."*
- *"The tool is especially useful on repetitive operations that typically happen in games (like creating several screens or several sprites)."*
- *"The diagrams are intuitive and the learning curve is really short for someone with some modeling experience."*
- *"Modeling is something that I do naturally, in order to understand what the problem is and to design a solution. Afterwards I have to code that solution manually, but the tool allows me to take even more advantage of a work that I already do."*
- *"The tool is very helpful to build the game without having to think a lot about the details of the code. However we still have to think about gaming concepts while developing the game."*
- *"Using the framework enables the developer to focus on the design and interaction of elements, instead of on the implementation details."*
- *"I didn't have to think at all about the implementation. My effort was entirely focused on making the game to comply with the specifications. During the 'manual' development using the game engine, coding specifics were often the main concern."*

The data collected from the implemented games and from observations during the experiment allowed us to find out evidence that supports the subjects' claims above. First of all, the manually implemented versions of the games had considerably more bugs upfront, which had to be fixed reactively, when compared to the versions created with the game SPL assets. Examples of bugs include:

- The background music of a previous screen keeps playing even though the game has transitioned to a new screen.
- A collision sound effect is played several times when a collision happens, instead of only once.
- Typos in screen and asset names caused the runtime to load wrong assets and even throw “file not found” exceptions.
- For Pong, which is a two-player game, concepts related to different players got recurrently swapped in the implementation. Examples include player-specific sounds, “player X wins” screens, scores, textual heads-up displays that show the scores, etc.
- Misplaced screen walls, which are manually implemented by specifying the coordinates of rectangles in relation to the screen dimensions, which is an error-prone activity. As an illustration, Figure 73 presents actual code implemented by one of the subjects during the experiment.

```
  topWall  = AddWall(600, 599,   0, 800);
  leftWall = AddWall(600,   0,   0,   1);
 rightWall = AddWall(600,   0, 799, 800);
bottomWall = AddWall(  1,   0,   0, 800);
```

Figure 73 – Wall initialization logic from one of the subjects’ code

- Bad random initialization of an entity velocity. In the Pong game, for instance, the initial **vertical** velocity of the ball is defined in the specs as a random value between 200 and 350 pixels per second, upwards or downwards. More than half of the developers introduced one of the following two bugs: (1) the ball only goes up (i.e., the random value considers the [+200, +350] interval for the Y speed but ignores the [-350, -200] one), or (2) the random initialization code enables the ball’s vertical velocity to be initialized to zero or very close to zero, making it to move only horizontally instead of toward the player’s paddles.
- An entity that was supposed to chase the main character actually runs away from it.
- The velocity of a chasing entity was wrongly initialized to a value that depends on the distance between the chasing entity and the target entity, instead of being constant.
- Bad sequencing of game tasks. For instance, the collision detection for entities which were already removed from the screen during that frame crashed the game.

- On the top of such game logic specific bugs, more generic programming errors were also observed, such as null reference exceptions and race conditions due to the introduction of parallel programming.

We believe such an increased number of bugs has at least four main source causes. First, many manual implementation tasks require repetitive, error-prone efforts, such as copying, pasting and slightly modifying code. The tedious wall initialization code above is an example.

Second, implementing tasks in a lower abstraction level typically imply in higher levels of code complexity, such as multiple conditional branches, nested loops, etc. For example, the logic of collision handlers and input handlers commonly depend on the current state of their target entities. Players should not be able to move an entity or fire a bullet (by using the gamepad or keyboard) if the entity is in an "exploding" state. Likewise, decreasing the main character's hit points when it collides with an enemy should not happen when the main character is in an "invincible" state. All of such N:M combinations between handlers and states can easily get out of control when low-level code is the only abstraction developers have available. That leads to game logic misbehavior and an excessive dependency on testing, or reactive bug fixing.

Third, while the modeling experience enabled players to grasp the big picture of the game more easily and anticipate or look for requirements more constantly, the manual implementation implied in a lot of code churn and refactoring as the requirements were assimilated. Quite commonly, such code churn introduced not only new bugs but regressions as well. For instance, when refactoring code from a specific class into a recently created base class, some developers forgot to replace specific, hardcoded values (that make sense only to the specific class) by parameters. Changing models caused less churn on already existing code and, apparently, less bugs and regressions.

Fourth and interestingly, the subjects consulted the games' checklists (Appendix E) more often when using the game SPL assets, when compared to the manual implementation. Such a better and more continuous connection with the problem domain led to a better fidelity to the requirements and hence less bugs. In fact, the subjects mentioned that when programming against the FlatRedBall engine and XNA only, their focus was on the implementation, but when dealing with the game SPL assets, their focus was on the problem. One of the subjects said that he focused on the implementation when creating games manually because his first priority at that moment was to understand and overcome technical challenges. When he moved to the game SPL assets, he got a feeling that the implementation "would be easy" and that made his first priority to be the game requirements.

Positive feedback also revealed that the ArcadEx SPL brings higher levels of 168racticetions to different fronts, providing intuitive DSL syntaxes and context-aware tool integration to speed up game development tasks. Examples follow below:

- The ArcadEx toolset automatically manages common domain requirements and some technical tasks, such as:
  - Defining screen walls. In ArcadEx, this is already part of ArcadEngine's initialization code.
  - Creating score variables for the players. In ArcadEx, the concepts of players and scores are implicit and integrated with game events.
  - Checking what bounding shape a given sprite has (box or circle), so that the right collision methods are called. In ArcadEx, generators handle all cases and the DSLs enable developers to simply tell what the colliding entities are, independent from their collision shapes.
- As mentioned in Chapter 2, the complexity and learning curve of game engines are one of their main challenges. We noticed that when developers used only FlatRedBall and XNA, without the game SPL assets, many questions related to the inner workings of the engine and the platform were asked, such as where to place the input mapping code (in the screens vs. in the game initialization), whether screen instances are reused or re-created by the engine, whether the provided elapsed game time is in seconds or milliseconds, whether screen class names should be the qualified or not when implementing screen transitions, etc. For instance, when implementing Pong, one of the subjects (ID7) made some assumptions about FlatRedBall and did not create the required empty constructors for screen classes. Since the subject tested his code too late, he had a considerable amount of code to fix when he found out that the engine would not run his game under the implemented assumptions. As a result, he ended up taking 75% more time to develop Pong when compared to his counterparts (Figure 65).
- Subjects reported that the models and their concepts are more cohesive than source code. For example, the wall collision behavior of an entity and its input handlers are declared as part of the entity itself, instead of belonging to a screen's "update" method.
- While developers had to solely rely on file names and paths to manually assemble animation frames, the IDE integration provided by the modeling experience displays the actual contents of such files, drawing textures in custom property editors and playing the animations in preview controls.

- The solution/project explorer tool window in the IDE does not provide enough semantics to a given game asset. For example, the IDE presents audio assets and texture assets to developers in the same way: as an item in the project hierarchy. ArcadEx improves that by making the IDE to distinguish between asset types, suggesting them only in the right context. For instance, when creating an entity animation, only texture assets are displayed. On the other hand, if the developer is adding a background music to a game screen, the SPL will provide a list of audio assets only. This can be contrasted with the IDE's project tree navigation in which all assets look alike.
- Declaring "collision interests" of entities in EntityDSL was considered a relevant improvement over coding loops and inner loops (or complex Linq queries) every time something should be done when two types of entities collide. Subjects found the directionality of the collision relationship (with "source" and "target" roles) to be helpful.
- Similar feedback was observed for the placement of entity instances in a game screen's canvas, which is more effective than specifying an entity's position by using "vector" objects and coordinates in source code. In other words, visually placing entities and HUDs in the screen canvas using ScreenDSL provides a WYSIWYG (what you see is what you get) experience not available in the manual implementation.
- Likewise, in the manual implementation experience, developers who consume the FlatRedBall game engine have to specify the color of a textual heads-up display by providing three values, from 0 to 1, representing the RGB (Red, Green and Blue) components of the desired color. In the modeling experience, developers can accomplish the same task more effectively by selecting colors from a palette and receiving real-time feedback from their choice, which automatically re-colors textual HUD elements in the ScreenDSL model.

With regards to evaluating the progress of their games under development, developers in the manual implementation scenario ran partial versions of their games much more often. Developers reported that such an increased need for progress verification happened due to the smaller functionality increments in the manual implementation scenario, the overall slower pace to accomplish tasks manually, an increased confidence on the game SPL assets and unfamiliarity with the FlatRedBall game engine. Blow [2004] mentions that in real world scenarios, the costs of continuously rebuilding and running interim versions of a game for progress verification can become prohibitive, especially due to the large amount of game assets and the steps required to reach a given game state. On the other hand, in the modeling

experience, developers were able to accomplish a more coarse-grained set of tasks at a time, which resulted in a reduced need for progress verifications and more confidence on the development progress.

Maintainability was another relevant point of feedback. Across the board, developers agreed that maintaining and updating classes require more work than doing the same for concepts in a model. They found that models provide a better grouping and visualization of the game concepts than source code, since the IDE by itself does not make any distinction among source code files representing different types of concepts. IDEs do allow developers to group related classes in the same project folder or package, but that does not seem to be enough. On the other hand, keeping some model concepts in sync was a challenge, as explained in the next subsection. Moreover, additional maintainability challenges are introduced by partitioning models, which was not required by the experiment due to the simplicity of the target games, as mentioned in the experiment's threats (Subsection 5.4.2.5).

Another intriguing difference between the modeling and the manual implementation experiences relates to the order in which the game features were implemented. When models were used, developers tended to model a group of related concepts together, apparently bringing more cohesion to the development process. For instance, all screens of the game were typically modeled at once, enabling developers to have a concise overview of the game flow and to promptly identify when adjustments were required to such aspect of the game. Similarly, developers virtually implemented all of the collision detection handlers in the same "round". That seemed to create a "development vicinity" for concept instances of the same type, enabling developers to immediately compare them and freshly transfer learning from one modeled concept to the next, in a product line fashion. In contrast, in the manual implementation scenario, developers typically mixed different concept instances in the same "round", mimicking the findings of players along the game. For example, in some cases developers implemented the initial game screens, then some screen transition code, then the first entities that habit such screens, then the collision detection for such entities, then another set of entities, which resulted in a new series of collision detection logic, then the final screens, followed new screen transition code. That introduces an undesired level of context switching to the game development process and avoids developers to focus on a single aspect of the game at a time, therefore making them to miss the opportunity to deal with the concepts of such aspect holistically.

As introduced in Subsection 5.4.4.1 (*Development Effort Improvement*), the quality and consistency of the manually implemented code is also a very interesting data point obtained from the experiment. Developers reported that keeping code quality was an impacting extra task for the manual implementation scenario. As for consistency, different developers

ended up with different approaches and coding styles. Some developers refactored duplicated code to base and helper classes, while others favored copying and pasting. Some were concerned with providing documentation for their methods and classes, but some were not. Some benefited from their previous knowledge on the target APIs, while others did not have any. On the other hand, generated code ensures consistency and quality. Best practices and non-functional concerns such as performance, which is quite vital for digital games, are taken into account by generators, which play an optimization role similar to compilers. ArcadEx generators create code that has a minimal documentation at least, and follows a common set of standards. Most importantly, generators encapsulate the knowledge of experienced developers, which gets applied even when the generator is used by less experienced ones.

The initialization of the ball's velocity in the Pong game is an interesting example to illustrate such a discussion about code consistency. The requirement was to initialize the ball's horizontal and vertical velocities with a random value between 200 and 350 pixels per second, for each axis. Figure 74 to Figure 77 present different implementations from four subjects. The implementations not only differ in style, but use different algorithms, are located in different classes (*Ball* entity class vs. *MainScreen* class), use different random APIs (.NET's *System.Random* class vs. FlatRedBall's *FlatRedBallServices.Random* class) and have different assumptions about the random range's upper value (inclusive or exclusive).

```
Public Ball()
{
    ...
    Random m = new Random(DateTime.Now.Millisecond);
    this.Velocity.X = m.Next(200, 351) * GenerateMultiplier(m);
    this.Velocity.Y = m.Next(200, 351) * GenerateMultiplier(m);
}

int GenerateMultiplier(Random m)
{
    int multiplier = 0;
    if (m.Next(1, 3) == 1)
    {
        multiplier = 1;
    }
    else
    {
        multiplier = -1;
    }

    return multiplier;
}
```

Figure 74 – Ball's velocity initialization code (Subject ID1)

```
private void SetSprite()
{
    ...
    this.Velocity = new Vector3(Random(), Random(), 0);
    ...
}

private static int Random()
{
    Random random = FlatRedBall.FlatRedBallServices.Random;
    return random.Next(0, 2) == 0
      ? random.Next(200, 350)
      : -random.Next(200, 350);
}
```

Figure 75 – Ball's velocity initialization code (Subject ID3)

```
int Xspeed = 0;
int Yspeed = 0;

while (Xspeed > -200 && Xspeed < 200)
{
    Xspeed = FlatRedBall.FlatRedBallServices.Random.Next(-350, 350);
}

while (Yspeed > -200 && Yspeed < 200)
{
    Yspeed = FlatRedBall.FlatRedBallServices.Random.Next(-350, 350);
}

ballNPC.Velocity.X = Xspeed;
ballNPC.Velocity.Y = Yspeed;
```

Figure 76 – Ball's velocity initialization code (Subject ID8)

```
private void CreateBall()
{
    ...
    ball.Velocity.X = GetRandomSpeed();
    ball.Velocity.Y = GetRandomSpeed();
    ...
}

private int GetRandomSpeed()
{
    int abs = FlatRedBall.FlatRedBallServices.Random.Next(200, 351);
    int sign = FlatRedBall.FlatRedBallServices.Random.Next(0, 100);

    if (sign < 50)
    {
        return abs * (-1);
    }
    else
    {
        return abs;
    }
}
```

Figure 77 – Ball's velocity initialization code (Subject ID6)

Likewise, one of the developers used multithreading to implement some game 173ractiiors, although that was not really required and not used by any other subject. Moreover, by consuming the FlatRedBall game engine as is, some subjects identified patterns and good implementation practices for the target domain that were missing from the engine. The extra effort in identifying and implementing such patterns also ended up with multiple implementation approaches and brought consistency issues. For example, although a FlatRedBall screen contains a list of its game entities, it is a good practice to group entities belonging to the same type in their own lists, so that parts of the game logic, such as collision detection, can be more easily implemented. Some subjects realized that, but others did not.

In contrast, the ArcadEngine domain framework complemented by the modeling experience saved developers from the effort required to come up with patterns or identify the best way to implement a given task, since such patterns and best practices are automatically generated. When ArcadEx's models and code generators are used, implementations will always have the same style and algorithm, comply to a consistent design, consume the same APIs and never introduce bugs due to assumptions (e.g., inclusive vs. exclusive boundaries for random number generators). Two comments collected from the subjects in the post-experiment survey illustrate this discussion:

- *"The toolset gives easy access to well implemented algorithms and solutions to the average developer, who would otherwise need to figure out by himself solutions for typical game development challenges."*
- *"One thing to note was that the game generated with the tool had similar or better performance than the game manually created. One can argue that auto-generated code is not as fast as a manually created one, but the reality is that developers can also introduce performance bugs."*

5.4.4.4 Difficulty

In a scale from 1 to 10, in which 1 means very easy and 10 means very difficult, subjects provided an average score of **2.125** for the difficulty of using the game SPL assets and its encompassing process. This means they had slightly more difficulty than expected by the target value for this metric (2.0). Therefore, we were **not able to reject** this null hypothesis (H_{0d}) in favor of its counterpart (H_{1d}). The difficulty feedback score per subject is presented in Figure 78.

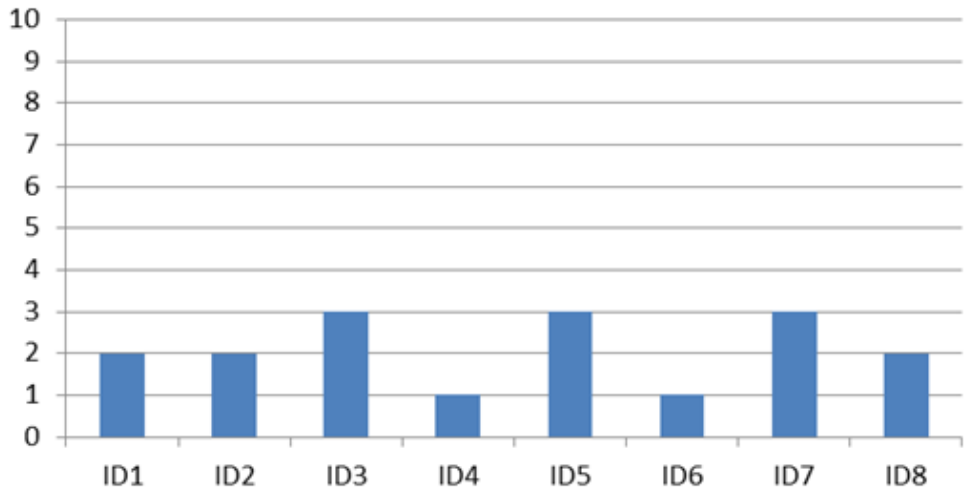


Figure 78 – Difficulty evaluation per subject

Open feedback was very valuable for identifying improvement areas of the game SPL assets. The majority of the comments in this space referred to improving the user experience, especially the user interface. Although at least two subjects reported in the feedback forms that once one gets used to the toolset, the experience flows well, one of our main lessons learned is that early investments on user experience and feedback might really pay off as a means to improve acceptance and productivity. Open feedback collected in the post-experiment survey includes:

- *"Some features of the tool were not very discoverable or I don't feel they were placed with the best experience in mind, for whatever reason."*
- *"The icons could be more meaningful; I had to hover over them many times to understand what action they would do."*
- *"The toolset still has some rough edges that would need to be addressed if it is intended to be used in the industry."*
- *"The toolset is very handy if you have previous game development experience and have a game development mindset. For newbies, I believe a wizard would make it easier for going through all game creation phases."*

The last comment is really interesting as it reveals the desire for contextualized, automated guidance, for which we advocate in Section 4.5.5 but were not able to introduce as part of the experiment's toolset due to resource constraints.

Similarly to the Helpfulness metric (Section 5.4.4.3), we compiled data from the subjects' code and models, as well as from observations taken during the experiment, to identify evidence that supports the subjects' claims above. First of all, we were able to conclude that some aspects of the DSLs' syntax and IDE integration, such as custom property editors, could be made more intuitive and allow developers to enter data in a more effective way. For

example, when creating a new entity as part of an event reaction, developers can set a random position for the new entity by means of specifying a screen area (rectangle) inside which the entity will be randomly placed. Such a rectangle is determined by its top-left point (X and Y coordinates), together with its width and height (Figure 79).

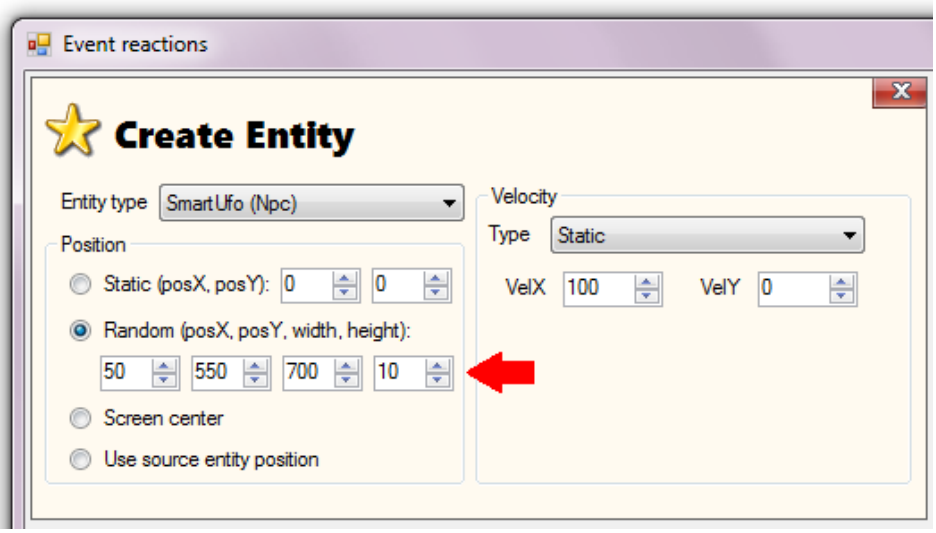


Figure 79 – Specifying a random position for a new entity

Since such coordinates and dimensions are just entered as numbers and cannot be visualized in the game screen, developers introduced errors in the random position initialization of new entities. We believe that this experience could be improved by letting developers to actually *draw* the rectangle in the screen canvas to tell the area in which the new entity will be randomly placed, which is then converted by the game SPL to the coordinates and dimensions required by the DSL meta-model.

In the same way, ArcadEx in its current state does not seem to be robust enough to backtrack all model changes (such as renaming entities) and keep the impacted modeling concepts in sync. Likewise, managing concepts and relations in the ArcadEx models can start to get cumbersome if the total number of concepts in the same diagram exceeds 5 and the majority of them are inter-related. Relationship lines eventually cross over others and the underlying DSL Tools engine is not able to keep the diagram clean enough. For instance, Figure 80 shows the first versions of diagrams modeled by the subjects for Pong. As it can be noticed, they are quite cluttered and that impacts the understanding of the overall game flow.

We conclude that improving readability by using a clean style is something applicable not only to source code, but to models as well. The screen inheritance relationship could be used to make the game flow more visually digestible, but none of the subjects employed such a relationship or asked about it, which gives some indications that more user education, such as training, tutorials, semantic validators and scaffolding techniques might be required

for a better awareness, discoverability and adoption of the SPL assets. On the other hand, while the diagram cluttering brings some indication that more robust modeling paradigms might be required to make the solution to scale (e.g., collapsible views or semantic zooming), it also emphasizes the relevance of enabling developers to partition models in multiple diagrams.

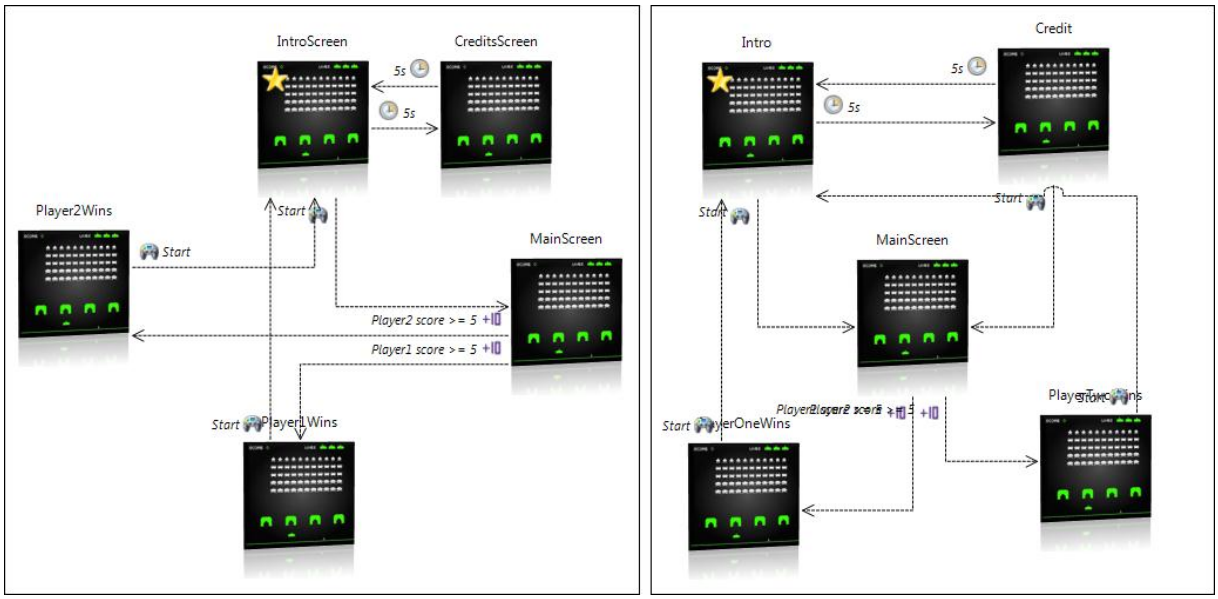


Figure 80 – Examples of cluttered GameDefinitionDSL diagrams for the Pong game

Developers also identified the introduction of logical operators (AND, OR and NOT) to manipulate lists of concepts as a desirable improvement. For example, GameDefinitionDSL does not let developers to specify multiple transition triggers for the same screen transition, such as *"Go from the intro screen to the main screen if the player 1 presses the Start button OR the player 2 presses the Start button"*. In such a case, two transitions have to be created, one for each trigger.

Developers also reported that they would welcome more semantic validators, to avoid them from introducing modeling errors or bugs that were only found in execution time. For example, ArcadEx does not warn developers when the position entered for a new entity is outside the screen boundaries. In fact, we suggest game SPL designers to document invalid values and states for all model concepts, including their properties and relationships, so that they can create a rich set of semantic validators that speed up the modeling experience.

Additional broadly applicable suggestions include:

- Support copy and paste of model elements.
- Make all custom property editors to load with the current property value.

- When a concept is double-clicked, launch its most used property editor. For example, double-clicking a collision link should open the *Collision Triggers* property editor.
- Present specific options before the generic ones, otherwise developers may settle for the sub-optimal, generic options. For example, consider Figure 79, which offers the “screen center” option for a new entity position. Such an option is very specific, since it enables developers to create new entities in a precise, specific location quite relevant to the game (the screen center). However, a more generic option unfortunately comes first in the list: the “static” position, in which X and Y coordinates can be freely entered for the new entity’s position, enabling developers to place new entities anywhere in the screen. As a result, even when developers wanted to create an entity in the screen center, they commonly settled for the “static” position option just because they found it first. They then hard-coded the screen center coordinates in the “static” position option, instead of using the more appropriate “screen center” option, which is independent from absolute values.
- Integrate the code generation step as a post-compilation event. Some subjects forgot quite often to re-generate code prior to compile their games.

On the other hand, more precise improvement areas were collected for each DSL. Similar to the broadly applicable suggestions above, the specific suggestions were either explicitly reported by the subjects or concluded as the result of observing the experiment.

5.4.5 Lessons Learned

The following lessons were learned from the experiment:

- One of the main challenges faced during the experiment was the limited number of available machines properly configured with the experiment environment. With only two machines available, only two subjects at most attended to each experiment session, leading to an increased number of sessions. That was a limitation to scale the experiment.
- Guidance should be provided upfront to the subjects as for the expectations on the quality of their manual code. Subjects should be guided to **not** compromise quality by implementing code as fast as possible, just to complete the experiment. On the contrary, they should be guided to write code as they normally would, taking into account good practices such as modularization, reuse, refactoring, etc. This increases the cost of the experiment but provide more accurate results.

- When defining ranges of values for the experiment's metrics, use a consistent approach for all metrics (e.g., bigger numbers mean better results). In our case, a big number for the *Helpfulness* metric was a good thing, while a big number for the *Difficulty* metric was a bad thing. This caused some confusion when the subjects were filling the post-experiment survey.
- The provided training and cheat sheet (Appendix D) were invaluable to the experiment. Some developers reported that they would probably take the double of the development time in the manual implementation scenario, looking for how to implement a given set of tasks, if a cheat sheet was not available. However, it would not be fair to run the experiment without the cheat sheet, since it accounts for the developers' eventual lack of experience in the FlatRedBall game engine and XNA.
- Likewise, the game development tasks checklist (Appendix E) was considered useful for different purposes: some developers used it as a specification, which helped to guide the implementation, while others preferred to consult the checklist only close to the end, for validation purposes.
- The measured experiment's cost should by no means be taken as an indication of the actual adoption costs that an organization would face when employing Domain-Specific Game Development. Adoption costs involve tasks which are out of the scope of this research and specific to the reality of each organization, such as costs related to changing from and integrating with legacy tools and processes, coming up with profiles for deploying the approach, and others.

5.5 Chapter Summary

This chapter presented the different techniques employed to evaluate the proposed Domain-Specific Game Development approach. First, an informal evaluation against desirable properties of domain-specific development assets was presented. Following, exploratory and confirmatory cases studies were described. Finally, an experimental study, carried out with actual software developers in the industry, was detailed and discussed. Although such evaluation techniques still represent a subset of what can be measured and analyzed from the outcomes of the proposed approach, it provides a valid starting point for assessing and somehow indicating its potential for improving digital games development.

The controlled experiment was the most costly and relevant evaluation approach. It brings a much better foundation to understand the benefits and shortcomings of Domain-Specific Game Development, especially given that the DSL literature primarily provides an-

ecdotal evidence for the claimed DSL usage benefits [Hermans et al., 2009]. We were able to measure a development efforts ratio of 4.59 to 5.57, to estimate a ROI after 10 to 15 instances, to collect positive feedback on the helpfulness of the approach and to identify opportunities to decrease the difficulties of its usage. The main lesson learned is that designing and refining the user experience as a whole, not only the DSLs' visual (concrete) syntax, can make a good use of early experiments and feedback. We believe that automated guidance (Section 4.5.5) might also play an important role to guide developers on accomplishing their tasks more effectively. In short, we consider the obtained results to be encouraging and an indication that the approach has potential and should be pursued.

6. CONCLUSIONS

This final chapter presents the main contributions of this research (Section 6.1), its limitations and suggestions for future work (Section 6.2), and our final remarks (Section 6.3).

6.1 Contributions

Given the peculiarities of digital games, automating game domains should not simply employ Software Engineering techniques as is, in special SPLs, DSLs, and Domain Engineering. On the contrary, by using a systematic domain-specific development approach streamlined to digital games, game developers and designers can envision and analyze target game domains and bridge the Domain Analysis to core assets in a game SPL, in a more effective, systematic and reproducible way. That constitutes the essence of the ideas presented in this thesis.

The main contributions of the Domain-Specific Game Development approach claimed by this research can be summarized as follows:

- **Executable game specifications.** By using expressive game DSLs that are really close to the domain of game developers and designers, the approach enables high-level game design discussions to not only be documented, but also promptly executed for evaluation and prototyping purposes. Some game companies in the industry have already been using click-and-play tools to create prototypes, which are later on discarded once the actual development phase beings, in which actual IDEs are employed for implementation. We believe the proposed approach reuses and extends the lifetime of prototypes, encouraging them to be refined toward the definitive game models and (generated) code, instead of being discarded.
- **Reduced complexity for consuming game engines.** As introduced in Chapters 2 and 3, one of the main challenges in consuming a game engine is its tough learning curve. Promoting game engines to domain frameworks, via domain-specific languages and tools streamlined for game sub-domains, as well as adaptation layers, increases the level of abstraction to consume many game engine functionalities. It also customizes game engines for specific game sub-domains.
- **Breakdown of game development tasks into more granular and automatable chunks.** Game development is not a monolithic task. While this is already clear for the multiple disciplines involved in the process (art, sound, design, engineering, etc.), Domain-Specific Game Development provides a bet-

ter separation of concerns specifically targeted at the engineering part of game development, using a divide-to-conquer approach in which game developers and designers can deal with specific sub-domains, one at a time, in a higher abstraction level. This aids with managing the complexity of tangled game architectures, a recurrent problem in the area [Blow, 2004].

- **Incremental delivery of value for prioritized game sub-domains.** Game SPLs are not an all-or-nothing automation approach. Even if the first version of a game SPL automates only one sub-domain, delivering a single language and generator, game designers and developers can already start harvesting the benefits from it. Future versions of such a SPL can deliver new assets or improve already existing ones, automating more sub-domains incrementally. Most importantly, Domain-Specific Game Development provides guidance to ensure the sub-domains with the best return on investment are the ones prioritized for automation.
- **Domain-specific assets tailored to the unique characteristics of the envisioned family of games.** Domain-Specific Game Development does not suggest a one-size-fits-all solution for automating digital games development. In contrast, it recognizes that game sub-domains are too peculiar to be handled by the same set of tools, languages and assets in general. As the result, game SPL assets are developed by taking into account the specific features of the target game sub-domains, as well as the specific tasks that need to be performed by game developers and designers.
- **Flexibility and extensibility for unforeseen behaviors.** Perhaps more than any other software macro-domain, digital games development is characterized by a strong creative process targeted at causing the rupture of standards and surprising end-users, attempting design and technical feats that may have been never experimented before. That said, Domain-Specific Game Development establishes that game SPL designers should have in mind, from day one, that it is virtually impossible to predict all possible variations of game sub-domains. As a result, the created game SPLs rely on extensibility hooks of multiple sorts, enabling built-in SPL behaviors to be complemented by game developers and designers. Its assets are developed with flexibility as a primary tenet. It also relies on a feedback process in which extensions and customizations can be incorporated as built-in assets in subsequent versions of the SPL. This contrasts the expressive but inflexible universe of click-and-play tools.

- **Increased confidence that the resulting games comply with the original vision and requirements.** As introduced in Chapter 2, mapping the requirements of each product variant onto a framework, such as a game engine, is a non-trivial problem that generally requires the expertise of an architect or senior developer. Leveraging SPL techniques, Domain-Specific Game Development encapsulates the knowledge of more experienced developers and domain experts to automate such mapping step.

From the problem domain point-of-view, this research contributes to **integrating more the digital games development and DSL/SPL areas**, by providing a comprehensive discussion on the motivation and needs for the former to leverage the latter, elaborating on their intersections and points of high and low applicability.

We also provide at least three contributions to the broader Domain-Specific Development field. First, we present a **compilation of the good characteristics of domain-specific assets**, which can be used for guidance and evaluation of domain-specific processes. Second, we also presented a **compilation of multiple approaches related to the evaluation of Domain-Specific Development** (see Appendix B). Finally, our **experimental study** follows a scientific methodology, comprehends different standpoints (development effort improvement, ROI, helpfulness, difficulties, etc.) and can be used as a baseline for assessing the effectiveness of other domain-specific solutions.

6.2 Limitations and Future Work

Perhaps the main limitation of the proposed Domain-Specific Game Development approach is that it cannot be considered a complete Domain Engineering process per se, covering in full all areas of Domain Engineering (Domain Analysis, Design and Implementation). As introduced in Chapter 4, some Domain Engineering tasks are out of the scope of the approach or were not detailed. Examples include market analysis, evaluation functions, the domain-specific software architecture design and representation, build processes and installation processes. An important next step in the research is to comprehend such tasks toward a complete game Domain Engineering process, exploring how digital games development peculiarities impact or are impacted by them. An initial step to evaluate the effort of such a future work is to identify what percentage of Domain Engineering activities are touched by the approach versus the ones left out (Figure 14).

Some guidance items that we suggest as part of the approach (Chapter 4) are based on our own experience, trends identified from the SPL and DSL literatures and good sense. As a result, more actual evidence is welcome to complement the justification of such guidance items and more strongly back them up from a scientific perspective.

On the other hand, we believe that at least some of the outcomes of this research can also be employed to domains other than game development, or even be used as best practices for generic Domain Engineering processes. Examples include domain envisioning 184ractiities, the use of core dimensions prior to the features breakdown and, most notably, the “edge center” approach for creating core domain assets. However, exploring the applicability of such tasks and guidelines to other domains was not pursued by this research. Such a new focus seems to be quite exciting as a means to investigate a more broad applicability of the contributions to the SPL community.

The proposed approach has a strong focus on the engineering pieces of game development. While boundaries with other areas are touched, especially game design, there is a lot of room to explore on how the proposed engineering practices integrate with other disciplines, such as art, sound, storytelling, artificial intelligence and so on. Likewise, the experimental study could be extended by collecting opinions not only from software engineers, but also from other profiles such as game designers, artists and others. Running new versions of the experiment in which more threats are addressed (e.g., using development teams instead of a single developer or a more heterogeneous group of subjects) would also be a valuable contribution.

Still in the evaluation front, one of the most natural next steps is to assess the 184ractitiveness of the approach against other digital game domains, such as adventure games, platform games, board games, god games or a mix of them. We would also be really interested in knowing the outcomes of applying and evaluating the approach to horizontal domains, such as educative games (edutainment), serious games, adult games, “adver-games” and programming games. In the latter domain, players have no direct influence on the course of the game. Instead, a computer program or script is written in some domain-specific programming language in order to control the actions of the game entities. Therefore, playing a “programming game” means creating a program, using a DSL, that will determine how entity instances behave. This adds a whole new meta-level to the proposed approach (creating DSLs for creating games that create DSLs) which might yield interesting outcomes. It also opens a new front of exploration related to the ability of machines to fully create (and why not play?) games. Some robots are already able to deliver artistic contributions (by painting and composing); creating end-to-end games would be a challenging next step. Such very interesting discussion, of course, is out of the scope of this research.

Our evaluation methodology compiled good properties (characteristics) of domain-specific development assets, and evaluated the proposed approach against them. A natural evolution in that line consists on improving the approach by making it to better address some of such characteristics, such as testability and model-to-model transformation.

While our evaluation methodology was focused on investigating how the proposed approach compares to the state-of-the-art techniques in game development employed by the industry (i.e., using game engines), we did not engage in comparing it with click-n-play tools, as they are more applicable for simpler domains. Likewise, more refined experiments could be performed to evaluate the approach in a more holistic way, with control groups aimed at developing DSLs and generators for a same game domain, but one trained on the Domain-Specific Game Development approach and another employing state-of-the-art DSL development approaches. Experiments for evaluating other deliverables of the approach, such as feature models and reference architectures, could also be employed in the future.

From an implementation perspective, our case study and experiment were limited to Microsoft technologies such as the Microsoft DSL Tools (as the language workbench) and XNA (as the target game platform), empowered by the FlatRedBall game engine. The implementation of the approach through a mix of other language workbenches (such as Intentional Software and JetBrains' MPS), game engines and target platforms (e.g., mobile devices such as the iPhone) would be an interesting addition to the evaluation. We also consider as promising future work to pursue the investigation in more details of how other Software Engineering techniques such as Aspect Oriented Programming (AOP) [Kiczales et al., 1997] and staged configuration [Czarnecki et al., 2004b] apply to game SPLs.

As developers move the focus from source code to models, it is natural to think about empowering the modeling experience with popular tools and concepts that are already in place for the programming experience. For instance, although the techniques explored in this thesis enable game developers to debug generated code, we do not touch the topic of model debugging. Such a technique provides a higher level of abstraction to the debugging experience, enabling developers to add breakpoints to some model elements and inspect their values in run time, during a debug session. Similarly, model tracing and other programming techniques that could be applied to models were left out of the scope of this research.

Blow [2004] points out that a way to avoid the long compile and load times when developing games is to write a significant amount of code in a higher-level extension language that can be dynamically reloaded by the game engine without restarting. That said, another possible future work consists in investigating whether MDD and DSLs could play a more important role in the design and use of such higher-level, dynamically loaded languages.

The ArcadEx SPL itself can still be further extended to automate more of the arcade games domain. Not all of the variability points identified in the feature models are covered by the ArcadEx built-in assets, so those are yet to be implemented. Examples include extending the ScreenDSL to enable game developers and designers to draw indoor walls, adding rotation support to entities and provide more cross-language integration support between the

heads-up displays (HUDs) of screens and entity properties. Such sub-domains and features, of course, should be prioritized by following the tasks and guidelines of the proposed approach, and also take the controlled experiment's feedback as input.

Finally, one of the pending items from this research is to realize the end-to-end experience of creating a new product instance by first configuring the domain's feature model associated to the SPL, then launching rules based on the configuration to customize the IDE and other SPL assets (such as the product architecture, frameworks, generated code and DSL diagrams). This would provide a better starting point for developers to consume SPLs and their assets.

6.3 Final Remarks

This research claims that there is a hiatus in game development (Figure 8), in which easy-to-use script languages and click-n-play tools are many times not flexible enough, contrasting to the powerful yet many times too complex world of game engines. As a confirmation of such a claim, we have recently observed game development toolsets moving into the "high expressive yet high flexible" quadrant, such as integrated graphical environments backed up by game engines, like Unity [Menard, 2011]. Another related, recent trend is the increasing popularity of casual games, especially for the mobile and tablet markets. A natural question that arises then is whether the proposed approach stays current in the light of such new developments.

We believe that the new trend of casual and mobile games is a perfect case for Domain-Specific Game Development. These games not only have domain-specific mechanics, but are also much simpler and require less development costs than typical AAA titles such as the Halo, Gears of War and Fable series. That implies in a higher flow of games being developed for a same domain, which could make a good use of the ROI provided by Domain-Specific Game Development. Such an increased amount of games in the market per year will also require developers to focus on what makes each of them unique and distinct, and again Domain-Specific Game Development can help by means of encapsulating commonality in domain frameworks and supporting variability through languages, generators, IDE integration and extensibility hooks. Domain-specific tools can be a good competitive advantage over casual and mobile game development toolsets which are still somewhat generic, such as Unit itself.

Tasks related to the Domain-Specific Game Architecture might need to be revisited and customized for such new domains. Game engines are still used by recent casual games development approaches such as integrated graphical environments like Unit, but in contrast to that and to AAA titles, casual and mobile games may not necessarily consume robust

game engines, whose complexity is one of the main motivations of Domain-Specific Game Development. On the other hand, reuse opportunities (common code, patterns, configurable components, etc.) are inevitable identified in any family of games belonging to a new domains. Domain-Specific Game Development can still help to integrate those reuse artifacts toward a domain framework that can be seamlessly consumed by variability assets, such as languages. Nevertheless, we foresee that Game Domain Envisioning and Analysis activities, such as establishing expectations for core dimensions and breaking down the target domain in sub-domains toward more effective reuse assets, will still be applicable as is.

For the future, we envision a **game development supply chain**, in which the many different components encompassed by a digital game, from its plot to pixel shaders, could be independently developed, customized and assembled with other components, toward composing a final game. That would act as a catalyzer to unfold the "long tail" possibilities of game development, enabling its mass customization, i.e., large-scale production of goods tailored to individual customers' needs [Pohl et al., 2005]. We consider the guidelines proposed in this research, which enable the effective creation of game SPLs, as a down payment to realize such a vision.

Nonetheless, all the guidelines and techniques presented as part of this research just improve **how** game developers and designers are able to express their intentions. The most important part, the **what**, still has to be figured out for each game. And that happens to be the most challenging part of any software, as mentioned by Linda Northrop during the *"No Silver Bullet: Software Engineering Reloaded"* panel held at the 22nd OOPSLA Conference: *"Software Engineering involves more than programming... the hardest thing about building software is figuring out **what to say**, not how to say it."* [Fraser & Mancl, 2008]. No matter how many abstraction layers, languages, tools and frameworks are provided atop the assembly language instructions that run when a game is played, the most deterministic factor for a successful game title is still a good game design, empowered by human-centric tasks such as a creative process, rapid prototyping, early and often feedback. Engineering 187racticees are just means that enable a successful game development journey, but should not be taken as goals by themselves. In short, Gabler et al. [2005] summarize it very well: *"Design is paramount: everything else from art to engineering exists only to serve the final design"*.

REFERENCES

[Aaen et al., 1997] Aaen, I.; Bøttcher, P.; Mathiassen, L. The Software Factory: Contributions and Illusion, in Proceedings of the Twentieth Information Systems Research Seminar in Scandinavia, Oslo, 1997.

[Aarseth et al., 2003] Aarseth, E.; Smedstad, S.; Sunnana, L. A multi-dimensional typology of games, in Copier M., Raessens J., Level Up, In Proceedings of Digital Games Research Conference 2003, Utrecht, The Netherlands, 2003.

[Albuquerque, 2005] Albuquerque, M. Revolution Engine: 3D Game Engine Architecture (in Portuguese), Computer Science BS conclusion paper, Federal University of Pernambuco, 2005.

[Almeida et al., 2006] Almeida, E. S., Mascena, J. C. C. P., Cavalcanti, A. P. C., Alvaro, A., Garcia, V. C., Meira, S. R. L. And Lucrédio, D. The Domain Analysis Concept Revisited: A Practical Approach. 9th International Conference on Software Reuse (ICSR), Turin, Italy, p. 43-57, 2006.

[Almeida, 2007] Almeida, E. S. RiDE: The RiSE Process for Domain Engineering, Ph.D. Thesis, Federal University of Pernambuco, 2007.

[Alves et al., 2005] Alves, V.; Matos, P.; Cole, L.; Borba, P.; Ramalho, G. Extracting and Evolving Mobile Games Product Lines, 9th International Software Product Line Conference (SPLC'05), Rennes, France, September, 2005, pp. 70-81.

[Ambrosine, 2009] Ambrosine.com, Game Creation Resources, available at http://www.ambrosine.com/resource.html, retrieved on April 3, 2009.

[America et al., 2000] America, P.; Obbink, H.; Ommering, R. V.; Linden, F. V. D. CoPAM: A Component-Oriented Platform Architecting Method Family for Product Family Engineering, 1st Software Product Line Conference (SPLC1), Denver, Colorado, USA, August, 2000, pp. 15.

[Anastasopoulos & Gacek, 2001] Anastasopoulos, M.; Gacek, C. "Implementing Product Line Variabilities," in Symposium on Software Reusability (SSR), Toronto, Canada, 2001, pp. 109–117.

[Araujo, 2006] Araujo, A. AGP: Agile Game Process (in Portuguese), Computer Science BS conclusion paper, Federal University of Pernambuco, 2006.

[Araujo, 2009] Araujo, A. R. S. Play4Fun: A Casual Digital Games Factory. (in Portuguese) MSc dissertation, Federal University of Pernambuco, 2009.

[Atkinson et al., 2000] Atkinson, C.; Bayer, J.; Muthig, D. Component-Based Product Line Development: The KobrA Approach, 1st Software Product Line Conference (SPLC1), Denver, Colorado, USA, August, 2000, pp.19.

[Azevedo et al., 2009] Azevedo, L. S. S. T.; Santos, A. L. M.; Furtado, A. W. B. Elegy: Applying the Game Factory Process to the Role-Playing Games Domain (in Portuguese), VIII Brazilian Symposium on Games and Digital Entertainment (SBGames2009), 2009.

[Baker et al. 2005] Baker, P.; Loh, S.; Weil, F. Model-driven engineering in a large industrial context – 190otorola case study. In Proceedings 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS), Vol. 3713 of Lect. Notes in Comp. Sc., pp. 476-491. Springer-Verlag, 2005.

[Barros et al., 2006] Barros, M. O.; Dantas, A. R.; Veronese, G. O.; Werner, C. M. L. Model-driven game development: experience and model enhancements, in Software Project Management Education. Software Process: Improvement and Practice, Vol. 11, No. 4, pp. 411-421, 2006.

[Basili et al., 1994] Basili, V. R.; Caldiera, G.; Rombach, H. D. The goal question metric approach. In Encyclopedia of Software Engineering, pp. 528–532. John Wiley & Sons, Inc. 5.1.2, 1994.

[Bass et al., 2003] Bass, L.; Clements, P.; Kazman, R. Software Architecture in Practice, 2nd Edition. Addison-Wesley, 2003.

[Basu et al., 1997] Basu, A.; Hayden, M.; Morrisett, G.; von Eicken, T. A language-based approach to protocol construction, in First ACM SIGPLAN Workshop on Domain-Specific Languages, pp. 1-15, 1997.

[Batory & O'Malley, 1992] Batory, D.; O'Malley, S. The Design and Implementation of Hierarchical Software Systems with Reusable Components, ACM Transactions on Software Engineering and Methodology (TOSEM), Vol. 01, No. 04, October, 1992, pp. 355-398.

[Batory et al., 2002] Batory. D.; Johnson, C.; MacDonald, B.; von Heeder, D. Achieving extensibility through product-lines and domain-specific languages: a case study, ACM Transactions on Software Engineering and Methodology (TOSEM), Vol. 11, No. 02, April, 2002, pp. 191-214.

[Bayer et al., 1999] Bayer, J.; Flege, O.; Knauber, P.; Laqua, R.; Muthig, D.; Schmid, K.; Widen, T.; DeBaud, J. PuLSE: A Methodology to Develop Software Product Lines, In: Symposium on Software Reusability (SSR), ACM Press, Los Angeles, USA, May, 1999, pp. 122-131.

[Bellis, 2009] Bellis, M. Computer and Video Game History, available at http://inventors.about.com/library/inventors/blcomputer_videogames.htm, retrieved on April 3, 2009.

[Beuche & Spinczyk, 2003] Beuche, D.; Spinczyk, O. Variant management for embedded Software Product Lines with pure::consul and AspectC++, in 18th Object-oriented programming, systems, languages, and applications (OOPSLA). ACM, 2003, pp. 108–109.

[Björk & Holopainen, 2005] Björk, S.; Holopainen, J. Patterns in Game Design, Charles River Media Inc., Hingham, Massachusetts, 2005.

[Björk et al., 2003] Björk, S.; Lundgren, S.; Holopainen, J. Game Design Patterns, in Copier M., Raessens J., Level Up — Proceedings of Digital Games Research Conference 2003, Utrecht, The Netherlands, 2003.

[Blow, 2004] Blow, J. 2004. Game Development: Harder Than You Think. ACM Queue, Volume 1, Issue 10, 2004, pp. 28-37.

[Bonnie, 2007] Bonnie, R. The Power of the Persona. The Pragmatic Marketer Magazine, vol. 5, no. 4, 2007.

[Bosch, 2000] Bosch, J. Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach, Pearson Education (Addison-Wesley & ACM Press), 2000.

[Bosch, 2002] Bosch, J. Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization, 2nd Software Product Line Conference (SPLC2), San Diego, California, August, 2002, pp. 257-271.

[Breyer, 2008] Breyer, F. Heuristic Evaluation for Digital Game Prototypes (in Portuguese), MSc dissertation, Department of Design, Federal University of Pernambuco, 2008.

[Bruce, 1997] Bruce, D. What makes a good domain-specific language? APOSTLE, and its approach to parallel discrete event simulation, in First ACM SIGPLAN Workshop on Domain-Specific Languages, pp. 17-35, 1997.

[Calheiros at al., 2007] Calheiros, F.; Borba, P.; Soares, S.; Nepomuceno, V.; Alves, V. Product line variability refactoring tool, 1st Workshop on Refactoring Tools (WRT'07), in conjunction with the 21st European Conference on Object-Oriented Programming (ECOOP'07), Berlin, Germany, July, 2007, pp. 33-34.

[Callele et al., 2005] Callele, D., Neufeld, E., Schneider, K., 2005. Requirements Engineering and the Creative Process in the Video Game Industry. In Proceedings of the 13th IEEE International Conference on Requirements Engineering. IEEE, 240-252, 2005.

[Carvalho, 2006] Carvalho, G. H. P. A Prescriptive Methodology for Computer Games Development (in Portuguese), Computer Science BS conclusion paper, Federal University of Pernambuco, 2006.

[Chandler, 2006] Chandler, H. M. Game Production Handbook. Charles River Media. 1st edition, 2006.

[Cheesman & Daniels, 2001] Cheesman, J.; Daniels, J. UML Components: A Simple Process for Specifying Component-Based Software, Addison-Wesley, 2001.

[Church, 1999] Church, D. Formal Abstract Design Tools, July 1999, available at http://www.gamasutra.com/view/feature/3357/formal_abstract_design_tools.php, retrieved on February 28, 2010.

[Clements & Northrop, 2001] Clements, P.; Northrop, L.M. Software Product Lines: Practices and Patterns, Addison Wesley, August, 2001.

[Cockburn, 2001] Cockburn, A. Agile Software Development, Addison-Wesley, 2001.

[Collins-Cope & Matthews, 2000] Collins-Cope, M.; Matthews H. A Reference Architecture for Component Based Development. Object Oriented Information Systems (OOIS) 2000, pp. 225-237.

[Cook et al., 2007] Cook, S.; Jones, G.; Kent, S.; Wills, A. C. Domain-Specific Development with Visual Studio DSL Tools, Addison-Wesley Professional, June 2007.

[Costikyan, 1994] Costikyan, G. I Have No Words & I Must Design, Interactive Fantasy journal, no. 2, 1994.

[Crawford, 1984] Crawford, C. The Art Of Computer Game Design: Reflections Of A Master Game Designer, Osborne/McGraw-Hill,U.S, May 1984.

[Czarnecki & Eisenecker, 2000] Czarnecki, K.; Eisenecker, U.W. Generative Programming: Methods, Tools, and Applications. Addison Wesley, 2000.

[Czarnecki et al, 2002] Czarnecki, K.; Bednasch, T.; Unger, P.; Eisenecker, U. Generative programming for embedded software: An industrial experience report, in GPCE, ser. LNCS, vol. 2487. Springer, 2002, pp. 156–172.

[Czarnecki et al., 2004] Czarnecki, K.; Helsen, S.; Eisenecker, U. Formalizing Cardinality-based Feature Models and their Staged Configuration, OOPSLA'04 Eclipse Technology eXchange Workshop, 2004.

[Czarnecki et al., 2004b] Czarnecki, K.; Helsen, S.; Eisenecker, U. Staged Configuration Using Feature Models, R.L. Nord (Ed.): SPLC 2004, LNCS 3154, pp. 266–283, Springer-Verlag, 2004.

[Czarnecki, 2005] Czarnecki, K. Overview of generative software development," in Intl Workshop on Unconventional Programming Paradigms, France, Sep 15-17, 2004, ser. LNCS, Vol. 3566. Springer, 2005, pp. 326–341.

[D'Souza & Wills, 2001] D'Souza, D.; Wills, A. C. Objects, Components, and Frameworks with UML – The Catalysis Approach, Addison-Wesley, 2001, pp. 816.

[Datamonitor, 2011] Datamonitor. Software: Global Industry Guide 2010, abstract available at http://bit.ly/zIhV0b, last accessed on March 11, 2012.

[Deelstra et al., 2005] Deelstra, M.; Sinnema, M.; Van Gurp, J.; Bosch, J. Model Driven Architecture as Approach to Manage Variability in Software Product Families. In: Aßmann, U., Aksit, M., Rensink, A. (eds.) MDAFA 2003. LNCS, vol. 3599, pp. 109–114. Springer, Heidelberg (2005).

[Demachy, 2003] Demachy, T. Extreme Game Development: Right on Time, Every Time, 2003. Available at www.gamasutra.com.br/resource_guide/20030714/demachy_01.shtml, retrieved on December 12, 2008.

[Dénes & Keedwell, 1974] Dénes, J.; Keedwell, A. D. Latin squares and their applications. New York-London: Academic Press. Pp. 547, 1974.

[Diehl & Stroebe, 1987] Diehl, M.; Stroebe W. Productivity Loss In Brainstorming Groups: Toward the Solution of a Riddle. Journal of Personality and Social Psychology, Vol. 53, No. 3, pp. 497-509, 1987.

[Discovery, 2007] Discovery Channel, Rise of the Videogame Documentary, released in November 2007.

[Dobbe, 2007] Dobbe, J. A Domain-Specific Language for Computer Games. MSc dissertation, Delft University of Technology, 2007.

[Easterbrook et al., 2007] Easterbrook, S. M.; Singer, J.; Storey, M.; Damian, D. Selecting Empirical Methods for Software Engineering Research, in F. Shull and J. Singer "Guide to Advanced Empirical Software Engineering", Springer, 2007.

[ESA, 2011] Entertainment Software Association, Essential Facts about the Computer and Video Game Industry, 2011.

[Eskelinen, 2001] Eskelinen, M. Towards Computer Game Studies, Proceedings of SIGGRAPH 2001, Art Gallery, Art and Culture Papers, pp. 83-87, 2001.

[Fabricatore et al., 2002] Fabricatore, C.; Nussbaum, M.; Rosas, R. Playability in Action Videogames: A Qualitative Design Model, Human Computer Interaction, Vol. 17, No. 4, pp. 311-368, 2002.

[Falstein, 2006] Falstein, N. The 400 Project, March 2006, available at http://www.theinspiracy.com/400_project.htm, retrieved on February 28, 2010.

[Fan et al., 1996] Fan, J. et al. Black Art of Java Game Programming, Waite Group Press, 1996.

[Fenton & Pfleeger, 1988] Fenton, N. E.; Pfleeger, S. L.; Software Metrics: A Rigorous and Practical Approach, Course Technology. 2 edition, 1998.

[Flood, 2003] Flood, K. Game Unified Process (GUP). 2003. Available at www.gamedev.net/references/articles/article1940.asp, retrieved on December 12, 2008.

[Flynt, 2005] Flynt, J. Software Engineering for Game Developers. Course Technology, 1st edition, 2005.

[Folmer, 2007] Folmer, E. Component Based Game Development: A Solution to Escalating Costs and Expanding Deadlines? 10th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE), 2007, pp. 66-73.

[Fowler, 2004] Fowler, M. Inversion of Control Containers and the Dependency Injection pattern, January 2004, available at http://martinfowler.com/articles/injection.html (May 10, 2009).

[Fowler, 2005] Fowler, M. Language Workbenches: The Killer-App for Domain Specific Languages?, available at martinfowler.com/articles/languageWorkbench.html, retrieved on March 23, 2009.

[Frakes & Isoda, 1994] Frakes, W. B.; Isoda, S. Success Factors of Systematic Software Reuse, IEEE Software, Vol. 12, No. 01, September, 1994, pp. 15-19.

[Frakes & Kang, 2005] Frakes, W. B.; Kang, K. C. Software Reuse Research: Status and Future, IEEE Transactions on Software Engineering, Vol. 31, No. 07, July, 2005, pp. 529-536.

[France & Rumpe, 2007] France, R.; Rumpe, B., "Model-driven development of complex software: A research roadmap," in 29th Intl Conference on Software Engineering — Future of Software Engineering. USA: IEEE/CS, 2007, pp. 37–54.

[Fraser & Mancl, 2008] Fraser, S., Mancl, D. No Silver Bullet: Software Engineering Reloaded. IEEE Software, vol. 25, no. 1, pp. 91-94, Jan./Feb. 2008

[Frauenheim, 2004] Frauenheim, E. No fun for game developers? CNET News, http://news.cnet.com/8301-10784_3-5449296-7.html, retrieved on November 11, 2010.

[Fullerton et al., 2004] Fullerton, T.; Swain, C.; Hoffman, S. 2004. Game Design Workshop: Designing Prototyping and Playtesting Games. CMP Books, pp. 157.

[Furtado & Santos, 2002] Furtado, A.; Santos A. FunGEn: A Game Engine for Haskell (in Portuguese), 1st Brazilian Workshop in Games and Digital Entertainment (Wjogos2002).

[Furtado, 2006] Furtado, A. SharpLudus: Improving Game Development Experience through Software Factories and Domain-Specific Languages. MSc dissertation, Federal University of Pernambuco, 2006.

[Gabler et al., 2005] Gabler, K.; Gray, K.; Kucic, M.; Shodhan, S. How to Prototype a Game in Under 7 Days: Tips and Tricks from 4 Grad Students Who Made Over 50 Games in 1 Semester, available at http://www.gamasutra.com/features/20051026/gabler_01.shtml, retrieved on May 1, 2008.

[Gamma et al., 1995] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[Genero et al., 2007] Genero, M. et al. Building measure-based prediction models for uml class diagram maintainability. Empirical Softw. Engg., Kluwer Academic Publishers, Hingham, MA, USA, Vol. 12, No. 5, pp. 517–549, 2007.

[Genero et al., 2008] Genero, M.; Poels, G.; Piattini, M. Defining and validating metrics for assessing the understandability of entity-relationship diagrams. Data Knowl. Eng., Elsevier Science Publishers B. V., Amsterdam, The Netherlands, Vol. 64, No. 3, pp. 534–557, 2008.

[Gillin, 2006] Gillin, P. Getting Into Gear. BizTech Magazine, June, 2006.

[Gomaa, 2005] Gomaa, H. Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures, Addison-Wesley, 2005, pp. 701.

[Gray et al., 2008] Gray, J.; Fisher, K.; Consel, C.; Karsai G.; Mernik, M.; Tolvanen, J.-P. "Domain-Specific Languages: The Good, the Bad, and the Ugly" Panel, International Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA) 2008.

[Greenfield et al. 2004] Greenfield, J. et al., Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools, Wiley & Sons, 2004.

[Greenfield, 2004] Greenfield, J. The Case for Software Factories, Microsoft Architect Journal, July 2004.

[Griss et al., 1998] Griss, M. L.; Favaro, J.; d' Alessandro, M. Integrating Feature Modeling with the RSEB, 5th International Conference on Software Reuse (ICSR), Victoria, Canada, June, 1998, pp. 76-85.

[Guckenheimer & Perez, 2006] Guckenheimer, S.; Perez, J. J. Software Engineering with Microsoft Visual Studio Team System, Addison-Wesley Professional. 1st edition, May 2006.

[Guerra et al., 2008] Guerra, E.; Lara, J. de.; Díaz, P. Visual specification of measurements and redesigns for domain specifi c visual languages. J. Vis. Lang. Comput., Academic Press, Inc., Orlando, FL, USA, Vol. 19, No. 3, pp. 399–425, 2008.

[Guizzardi et al., 2002] Guizzardi, G.; Pires, L. F.; Sinderen, M. J. V. On the role of domain ontologies in the design of domain-specific visual modeling languages," in 2nd OOPSLA Workshop on Domain-Specific Visual Languages, 2002.

[Haddad & Tesser, 2003] Haddad, H.; Tesser, H. Reusable Subsystems: Domain-Based Approach. In: 2002 ACM Symposium on Applied Computing (SAC 2002), pp. 971–975. ACM, New York (2003).

[Henderson, 2006] Henderson, J. The Paper Chase: Saving Money via Paper Prototyping. Gamasutra, available at http://www.gamasutra.com/features/20060508/henderson_01.shtml, retrieved on March 21, 2009.

[Hermans et al., 2009] Hermans, F.; Pinzger, M.; van Deursen, A. Domain-specific languages in practice: A user study on the success factors. In Proceedings ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MODELS), Empirical Track, vol. 5795 of Lecture Notes in Computer Science, pp. 423-437. Springer-Verlag, 2009.

[Hernandez & Ortega, 2010] Hernandez, F. E.; Ortega, R. R. Eberos GML2D: A Graphical Domain-Specific Language for Modeling 2D Video Games.In Proceedings of the 10th SPLASH Workshop on Domain-Specific Modeling (DSM'10), Aalto-Print, 2010.

[Herndon & Berzins, 1988] Herndon, R. M.; Berzins, V. A. The realizable benefits of a language 195rototypeing language, IEEE Transactions on Software Engineering, Volume 14, pp. 803-809, 1988.

[Hessellund et al., 2007] Hessellund, A.; Czarnecki, K.; Wasowski, A. Guided development with multiple domain-specific languages, in MoDELS, ser. LNCS, Vol. 4735. Springer, 2007, pp. 46–60.

[Jackson, 1995] Jackson, M.A. Software Requirement & Specifications: A lexicon of practice, principles and prejudices, Addison-Wesley, 1995.

[Jacobson et al., 1999] Jacobson, I.; Booch, G.; Rumbaugh, J. The Unified Software Development Process, Addison-Wesley Professional, 1st edition, 1999.

[Jenkins, 2003] Jenkins, H. Game Design as Narrative Architecture, in Wardrip-Fruin N., Harrigan P., First Person: New Media as Story, Performance and Game, The MIT Press, Cambridge MA, 2003.

[Juul, 2009] Juul, J. A history of the computer game, in A Clash between Game and Narrative, MSc dissertation, Institute or Nordic Language and Literature, University of Copenhagen, February 1999.

[Kang et al., 1990] Kang, K.; Cohe, S.; Hess, J.; Nowak, W.; Peterson, S. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90TR-21, Software Engineering In-stitute, Carnegie Mellon University, 1990.

[Kang et al., 2002] Kang, K. C.; Lee, J.; Donohoe, P. Feature-Oriented Product Line Engineering, IEEE Software, Vol. 19, No. 04, July/August, 2002, pp.58-65.

[Kärnä et al., 2009] Kärnä, J.; Tolvanen, J-P., Kelly, S. Evaluating the use of DSM in embedded UI application development, Proceedings of DSM'09 at OOPSLA, 2009.

[Keith, 2006] Keith, C., 2006. Get in the Game: Agile Lessons From Video Game Developers. Better Software Magazine, Nov. 2006.

[Kelly & Tolvanen, 2008] Kelly, S.; Tolvanen, J-P. Domain-Specific Modeling: Enabling full code generation. Wiley, 2008.

[Kelly, 2010] Kelly, S. Domain-Specific Modeling: MDD that Works, blog, 17 March 2010, available at http://bit.ly/g1KyWp, retrieved on December 28, 2011.

[Kent, 2006] Kent, L.S. Return of Arcade: Great '80s-style fun. Entertainment News Service, South Coast Today newspaper, March 03, 96.

[Kiczales et al., 1997] Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C. V.; Loingtier, J-M.; Irwin, J. Aspect-Oriented Programming. In Proceedings of the European Conference on Object-Oriented Programming, ECOOP'97, LNCS 1241, pp. 220–242, Finland, June 1997. Springer-Verlag.

[Kieburtz et al., 2006] Kieburtz, R. B.; McKinney, L.; Bell, M.; Hook, J.; Kotov, A.; Lewis, J.; Oliva, D. P.; Sheard, T.; Smith, I.; Walton, L. A software engineering experiment in software component generation, in Proceedings of the 18th International Conference on Software Engineering, pp. 542-553, 1996.

[Klotz, 2009] Klotz, M. Lunch session at the International Summit on the Law and Business of Video Games, 2009.

[Knodel et al., 2005] Knodel, J.; Anastasopolous, M.; Forster, T.; Muthig, D. An Efficient Migration to Model-driven Development (MDD). Electronic Notes in Theoretical Computer Science 137(3), 17–27 (2005).

[Koster, 2004] Koster, R. A Theory of Fun for Game Design, Paraglyph, 2004.

[Kreimeier, 2003] Kreimeier, B. Game Design Methods: A 2003 Survey, March 2003, available at http://www.gamasutra.com/features/20030303/kreimeier_01.shtml, retrieved on February 28, 2010.

[Kreimeier, 2003b] Kreimeier, B. The Case For Game Design Patterns, March 2002, available at http://www.gamasutra.com/features/20020313/kreimeier_01.htm, retrieved on December 14, 2007.

[Kruchten, 1995] Kruchten, P. The 4+1 view model of architecture. IEEE Softw., IEEE Computer Society Press, Los Alamitos, CA, USA, v. 12, n. 6, p. 42–50, 1995.

[Krueger, 1992] Krueger, C. W. Software Reuse, ACM Computing Surveys, Vol. 24, No. 02, June, 1992, pp. 131-183.

[Krueger, 2001] Krueger, C. Easing the transition to software mass customization. In Proceedings of the 4th International Workshop on Software Product-Family Engineering, pp. 282-293, 2001.

[Kücklich, 2003] Kücklich, J. Perspectives of Computer Game Philology. Games Studies: The International Journal of Computer Games Research, Vol. 3, No. 1, 2003.

[Lane, 1999] Lane, D. Hyperstat, Atomic Dog Publishing, 2nd edition, 1999.

[Lange & Chaudron, 2004] Lange, C.; Chaudron, M. An empirical assessment of completeness in uml designs. In: Proceedings of the 8th Conference on Empirical Assessment in Software Engineering (EASE04), 2004. Pp. 111-121.

[Lange & Chaudron, 2005] Lange, C. F. J.; Chaudron, M. R. V. Managing model quality in UML-based software development. In: STEP '05: Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice. Washington, DC, USA: IEEE Computer Society, 2005. Pp. 7-16.

[Ledeczi et al., 2001] Ledeczi, A.; Bakay, A.; Maroti, M.; Volgyesi, P.; Nordstrom, G.; Sprinkle, J.; Karsai, G. Composing Domain-Specific Design Environments. IEEE Computer, vol. 34, no. 11, pp 44-51, 2001.

[Lee & Kang, 2004] Lee, K.; Kang, K. C. Feature dependency analysis for product line component design, in 8th Intl Conference on Software Reuse (ICSR), Madrid, Spain, 2004, pp. 69–85.

[Lee at al., 2002] Lee K.; Kang, K. C.; Lee, J. Concepts and guidelines of feature modeling for product line software engineering, in 7th Intl Conference on Software Reuse (ICSR), Austin, Texas, 2002, pp. 62–77.

[Lennox et al., 2004] Lennox, J.; Wu, X.; Schulzrinne, H. Call Processing Language (CPL): A Language for User Control of Internet Telephony Services, October 2004, http://www.ietf.org/rfc/rfc3880.txt

[Lenz & Wienands, 2006] Lenz, G.; Wienands, C. Practical Software Factories in .NET. Apress, 2006.

[Lindley, 2003] Lindley, A. Game Taxonomies: A High Level Framework for Game Analysis and Design, GamaSutra.com, 2003, available at http://www.gamasutra.com/features/20031003/lindley_01.shtml, May 04, 2009.

[Lisboa et al., 2007] Lisboa, L. B.; Garcia, V. C.; Almeida, E. S.; Meira, S. R. L. ToolDAy A Process-Centered Domain Analysis Tool , 21st Brazilian Symposium on Software Engineering, Tools Session, João Pessoa, Paraíba, Brazil, 2007.

[Lisboa et al., 2010] Lisboa, L. B.; Garcia, V. C.; Lucrédio, D.; Almeida, E. S.; Meira, S. R. L.; Fortes, R. P. M. A systematic review of domain analysis tools. Information & Software Technology, vol. 52, no. 1, pp. 1-13, 2010.

[Lucrédio et al., 2008] Lucrédio, D.; Fortes, R. P. M.; Almeida, E. S.; Meira, S. R. L. Performing domain analysis for model-driven software reuse. In 10th International Conference on Software Reuse, Beijing, China, 2008.

[Lucrédio, 2009] Lucrédio, D. A Model-Driven Approach for Software Reuse, PhD thesis, São Paulo University (USP), 2009 (In Portuguese: Uma Abordagem Orientada a Modelos para Reutilização de Software)

[Lucrédio, Fortes, Furtado et al., 2009] Daniel, L.; Fortes, R. P. M.; Furtado, A. W. B.; Santos, L. M.; Almeida, E. S.; Meira, S. L. Systematic Domain Implementation Using Model-Driven Development, submitted to the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE).

[Madeira, 2003] Madeira, C. FORGE V8: A Computer Games and Multimedia Applications Development Framework (in Portuguese), MSc dissertation, Federal University of Pernambuco, 2003.

[Maiden & Sutcliffe, 1996] Maiden, N.; Sutcliffe, A. A computational mechanism for parallel problem decomposition during requirements engineering. In: 8th International Workshop on Software Specification and Design, Germany, pp. 159–163 (1996).

[Maier & Volk, 2008] Maier, S.; Volk, D. Facilitating Language-Oriented Game Development by the Help of Language Workbenches. In Proceedings. Of the 2008 Conference Future Play, pp.224-227, ACM, 2008.

[Marinho, 2010] Marinho, L. P. GAL: A Domain-Specifc Language for Multi-Touch Games (in Portuguese), MSc dissertation, Federal University of Pernambuco, 2010.

[Marques de Almeida, 2008] Marques de Almeida, E. W. Commander Assembler: A Game Factory for Tactical RPGs using Domain-Specific Languages (in Portuguese), Computer Science BS conclusion paper, Federal University of Pernambuco, 2008.

[Mascena et al., 2005] Mascena, J. C. C. P.; Almeida, E. S. de.; Meira, S. R. de L. A comparative study on software reuse metrics and economic models from a traceability perspective. In: IEEE International Conference on Information Reuse and Integration (IRI). Las Vegas, Nevada, USA: IEEE/CS Press, 2005. Pp. 72–77.

[McGonigal, 2010] McGonigal, G. Can Gaming Save the World? TED 2010 Conference Talk, available at www.ted.com/talks/jane_mcgonigal_gaming_can_make_a_better_world.html, retrieved on May 02, 2010.

[McGuire, 2006] McGuire, R. 2006. Paper Burns: Game Design with Agile Methodologies. Gamasutra, available at http://www.gamasutra.com/features/20060628/mcguire_01.shtml, retrieved on March 27, 2009.

[McIlroy, 1968] McIlroy, M. D. Mass Produced Software Components, NATO Software Engineering Conference Report, Garmisch, Germany, October, 1968, pp. 79-85.

[Menard, 2011] Menard, M. Game Development with Unity. Course Technology PTR; 1 edition, January 19, 2011.

[Menon & Pingali, 1999] Menon, V.; Pingali, K. A case for source-level transformations in MATLAB, in Proceedings of the second USENIX Conference on Domain-Specific Languages, pp. 53-66 1999.

[MetaCase, 2000] MetaCase, Nokia Case Study, available at http://metacase.com/papers/MetaEdit_in_Nokia.pdf, retrieved on December 28, 2011.

[Metacase, 2009] Metacase, Domain-Specific Modeling With Metaedit+: 10 Times Faster Than Uml, http://www.metacase.com/papers/Domain-specific_modeling_10X_faster_than_UML.pdf.

[Miller, 2008] Miller, P. 2008. Top 10 Pitfalls Using Scrum Methodology for Video Game Development. Gamasutra, available at http://www.gamasutra.com/view/feature/3724/top_10_pitfalls_using_scrum_.php, retrieved on March 27, 2009.

[Modelware, 2006] Modelware. Engineering Metrics Definition. 2006. Available at: http://www.modelware-ist.org. Retrieved on August 15, 2010.

[Mohagheghi & Aagedal, 2007] Mohagheghi, P.; Dehlen, V. Where is the proof? – a review of experiences from applying MDE in industry. In: ECMDA-FA '08: Proceedings of the 4th European conference on Model Driven Architecture. Berlin, Heidelberg: Springer-Verlag, 2008. P. 432–443.

[Mongan & Suojanen, 2000] Mongan, J.; Suojanen N. Programming Interviews Exposed, p. 177, John Wiley & Sons, 2000.

[Monperrus, 2008] Monperrus, M. et al. A model-driven measurement approach. In: MoDELS '08: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems. Berlin, Heidelberg: Springer-Verlag, 2008. Pp. 505–519.

[Moon et al., 2005] Moon, M.; Yeom, K.; Chae, H. S. An approach to developing domain requirements as core assets based on commonality and variability analysis in a product line. IEEE Transactions on Software Enginerring, vol. 31, no. 7, pp. 551-569, 2005.

[Moreno-Ger et al., 2008] Moreno-Ger, P.; Martínez-Ortiz, I.; Sierra, J. L.; Fernández-Manjón, B. A Content-Centric Development Process Model, Computer, Volume 41, Number 3. IEEE, pp. 24-30, 2008

[Murray, 1997] Murray, J.H. Hamlet on the Holodeck: The Future of Narrative in Cyberspace, The Free Press, New York, 1997.

[Muskens et al. 2004] Muskens, J.; Chaudron, M.; Lange, C. Investigations in applying metrics to multi-view architecture models. In: EUROMICRO '04: Proceedings of the 30th EUROMICRO Conference. Washington, DC, USA: IEEE Computer Society, 2004. Pp. 372–379.

[Muszynski, 2005] Muszynski, M. Implementing a domain-specific modeling environment for a family of thick-client GUI components, in 5th OOPSLA Workshop on Domain-Specific Modeling, San Diego USA, 2005.

[Nascimento, 2008] Nascimento, L. M. Core Assets Development in Software Product Lines: Towards a Practical Approach for the Mobile Game Domain. MSc dissertation, Federal University of Pernambuco, Recife, Pernambuco, Brazil, 2008.

[Neighbors, 1980] Neighbors, J. M. Software Construction Using Components, PhD Thesis, University of California, 1980.

[Neves et al., 2008] Neves, A.; Campos, F.; Campello, S. B.; Castillo, L.; Barros, S.; Aragão, S. eXtensible Design Methods (XDM) (in Portuguese), 8th Brazilian Congress of Research and Development in Design, 2008.

[Neward, 2008] Neward, T. Why the Next Five Years Will Be About Languages, keynote at the The ServerSide Java Symposium, March 27, 2008.

[Nobrega, 2008] Nobrega, J. An Integrated Cost Model for Product Line Engineering, MSc dissertation, Federal University of Pernambuco, 2008.

[Overmars, 2004] Overmars, M. Game Maker 6 Help, release on 2004.

[Oxland, 2004] Oxland, K. Gameplay and Design, London: Pearson Education, 2004.

[Parnas, 1972] Parnas, D. On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM, vol. 15, no. 12, pp. 1053–1058, 1972.

[Parnas, 1976] Parnas, D. On the Design and Development of Program Families, IEEE Transactions on Software Engineering, March 1976.

[Pessoa, 2003] Pessoa, C. wGEM: A Game Development Framework for Mobile Devices (in Portuguese), MSc dissertation, Federal University of Pernambuco, 2003.

[Pilgrim, 2008] Pilgrim, J. Measuring the level of abstraction and detail of models in the context of MDD. In: Models in Software Engineering: Workshops and Symposia at MoDELS 2007, Nashville, TN, USA, 2007, revised selected papers. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 105–114.

[Pohl et al., 2005] Pohl, K.; Bockle, G.; van der Linden, F. Software Product Line Engineering: Foundations, Principles and Techniques, Springer, 2005, pp. 468.

[Preece et al., 1994] Preece, J.; Rogers, Y.; Sharp, H.; Benyon, D.; Holland, S.; Ca, T. Human-Computer Interaction, Addison Wesley, Harlow, England, 1994.

[Prieto-Diaz, 1990] Prieto-Diaz, R. Domain Analysis: An Introduction. In: ACM SIGSOFT Software Engineering Notes, Vol. 15, No. 02, pp. 47-54, April, 1990.

[Reed, 2010] Reed, A. Learning XNA 4.0: Game Development for the PC, Xbox 360, and Windows Phone 7, O'Reilly Media, Inc., December 27, 2010.

[Reyno & Cubel, 2008] Reyno, E. M.; Cubel, G. A. C. Model-Driven Game Development: 2D Platform Game Prototyping. Game-On 2008, 9th International Conference on Intelligent Games and Simulation, 2008, pp 5-7.

[Riebisch et al. 2002] Riebisch, M.; Böllert, K.; Streitferdt, D., Philippow, I. Extending Feature Diagrams with UML Multiplicities. 6th World Conference on Integrated Design & Process Technology (IDPT2002), Pasadena, CA, USA. June 23 – 27, 2002.

[Riebisch, 2003] Riebisch, M. Towards a More Precise Definition of Feature Models. Modelling Variability for Object-Oriented Product Lines, pp. 64-76, BooksOnDemand Publ. Co. Norderstedt, 2003.

[Robbes & Lanza, 2008] Robbes, R.; Lanza, M. Example-based program transformation, in MoDELS, ser. LNCS, vol. 5301. Springer, 2008, pp. 174–188.

[Roberts & Johnson, 1996] Roberts, D.; Johnson, R. Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks, Proceedings of Pattern Languages of Programs, 1996.

[Rocha, 2003] Rocha, E. Forge 16V: An Isometric Game Development Framework (in Portuguese), MSc dissertation, Federal University of Pernambuco, 2003.

[Rollings & Adams, 2006] Rollings, A.; Adams, E. Fundamentals of Game Design. Prentice Hall, 2006.

[Rollings & Morris, 2000] Rollings, A.; Morris, D.; Game Architecture and Design, The Coriolis Group, 2000.

[Safa, 2009] Safa, L. The Making Of User-Interface Designer: A Proprietary DSM Tool, Proceedings of DSM'07 at OOPSLA, 2007.

[Sametinger, 1997] Sametinger, J. Software Engineering with Reusable Components, Springer- Verlag, 1997, pp.275.

[Sawyer, 1995] Sawyer, B. The Getting Started Guide to Game Development FAQ, GameDev.net, 1995, available at http://www.gamedev.net/reference/articles/article261.asp, May 04, 2009.

[Sayre, 2007] Sayre, C., 10 Questions for Shigeru Miyamoto, http://www.time.com/time/magazine/article/0,9171,1645158,00.html, retrieved on October 18, 2007.

[Schmidt, 2006] Schmidt, D. C. Guest editor's introduction: Model-driven engineering, IEEE Computer, vol. 39, no. 2, pp. 25–31, 2006.

[Shukla & Schmit, 2006] Shukla, D.; Schmidt, B. Essential Windows Workflow Foundation, Addison-Wesley Professional, October 13, 2006.

[Simos et al., 1996] Simos, M.; Creps, D.; Klingler, C.; Levine, L.; Allemang, D. Organization Domain Modeling (ODM) Guideboo, Version 2.0, Technical Report, June, 1996, pp. 509.

[Sinnema et al., 2007] Sinnema, M.; Deelstra, S. Classifying Variability Modeling Techniques. Information and Software Technology, 49, pp. 717-739, 2007.

[Sirer & Bershad, 1999] Sirer, E. G.; Bershad, B. N. Using production grammars in software testing, in Proceedings of the second USENIX Conference on Domain-Specific Languages, pp. 1-14, 1999.

[Slocombe, 2005] Slocombe, M. Men Spend More Money on Video Games Than Music: Nielsen Report.Available at http://digital-lifestyles.info/display_page.asp?section=cm&id=2091, retrieved on March 27, 2009.

[Sommerville & Flynt, 2007] Sommerville, I.; Flynt, J. Software Engineering. Addison-Wesley. 7th edition, 2007.

[Spinellis, 2001] Spinellis, D. Notable design patterns for domain-specific languages. Journal of Systems and Software, Issue 56, pp. 91–99, 2001.

[SPL Hall of Fame, 2012] Software Engineering Institute (SEI). Software Product Line Hall of Fame, available at http://www.sei.cmu.edu/productlines/plp_hof.html, retrieved on April 1st, 2009.

[Staron, 2006] Staron, M. Adopting model driven software development in industry: A case study at two companies. In Proceedings 9th Int. Conf. on Model-Driven Engineering Languages and Systems (MoDELS'06), volume 4199 of Lect. Notes. In Computer Science, pp. 57-72. Springer-Verlag, 2006.

[STARS, 1993] Software Technology for Adaptable, Reliable Systems (STARS), The Reuse-Oriented Software Evolution (ROSE) Process Model, Technical Report, July, 1993, pp. 143.

[Sutman & Schementi, 2005] Sutman, A. B.; Schementi, J. M. Video game language. MSc dissertation, Worcester Polytechnic Institute, 2005.

[Svahnberg et al., 2001] Svahnberg, M.; van Gurp, J.; Bosch, J. On the Notion of Variabilities in Software Product Lines, IEEE/IFIP WICSA, Amsterdam, Netherlands, August, 2001, pp. 45-54.

[Szyperski, 2002] Szyperski, C. Component Software: Beyond Object-Oriented Programming, Addison-Wesley, 2002, pp. 588.

[Tolvanen & Kelly, 2005] Tolvanen, J.-P.; Kelly, S. Defining domain-specific modeling languages to automate product derivation: Collected experiences," in SPLC-Europe, ser. LNCS, vol. 3714. Springer, 2005, pp. 198–209.

[Tolvanen, 1998] Tolvanen, J.-P. Incremental method engineering with modeling tools: Theoretical principles and empirical evidence. PhD thesis, Jyvskyl Studies in Computer Science, 1998.

[Tolvanen, 2005] Tolvanen, J.-P. Domain-specific Modeling: Welcome to the Next Generation of Software Modeling, October 28, 2005, available at http://www.devx.com/enterprise/Article/29619, retrieved on April 13th, 2008.

[Tracz, 1995] Tracz, W. Domain-Specific Software Architecture (DSSA) Pedagogical Example, ACM SIGSOFT Software Engineering Notes, vol. 20, no. 3, pp. 49–62, 1995.

[van Deursen & Klint, 1998] van Deursen, A.; Klint, P. Little languages: Little maintenance?, Journal of Software Maintenance, Vol. 10, pp. 75-92, 1998.

[van Deursen et al., 2000] van Deursen, A.; Klint, P.; Visser, J. Domain-Specific Languages: An Annotated Bibliography, SIGPLAN Notices, ACM Press, Vol. 35, No. 6, pp. 26-36, 2000.

[Varró, 2006] Varró, D. Model transformation by example, in MoDELS, ser. LNCS, vol. 4199. Springer, 2006, pp. 410–424.

[Villela, 2000] Villela, R. M. M. B. Search and Recovery of Components in Software Reuse Environments (in Portuguese), PhD Thesis, Federal University of Rio de Janeiro, December, 2000, pp. 264.

[Völter & Bettin, 2004] Völter, M.; Bettin, J. Patterns for model-driven software development, in 9th European Conference on Pattern Languages of Programs (EuroPLoP), Irsee, Germany, 2004.

[Völter & Groher, 2007] Völter, M.; Groher, I. Product line implementation using aspect-oriented and model-driven software development, in Software Product Line Conference. IEEE/CS, 2007, pp. 233–242.

[Völter, 2003] Völter, M. A catalog of patterns for program generation. In Eight European Conference on Pattern Languages of Programs (EuroPLoP 2003), Irsee, Germany, 2003.

[Warmer & Kleppe, 2006] Warmer, J.; Kleppe, A. Building a flexible software factory using partial domain specific models, in Proc. Of The 6th OOPSLA Workshop on Domain-Specific Modeling, 2006.

[Weiss & Lai, 1999] Weiss, D.M.; Lai. C. T. R. Software Product-Line Engineering: A Family-Based Software Development Process, Addison-Wesley, 1999, pp. 426.

[White et al., 2005] White, J.; Schmidt, D. C.; Gokhale, A. Simplifying autonomic enterprise java bean applications via model-driven development: A case study. In Proceedings 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS), Vol 3713 of Lect. Notes in Comp. Sc., pp. 601-615. Springer-Verlag, 2005.

[Wiegers, 1999] Wiegers, K. E. Software Requirements, Microsoft Press, 1999, Chapter 1.

[Wiering, 1999] Wiering, M. The Clean Game Library. MSc dissertation, University of Nijmegen, 1999.

[Wijers, 1991] Wijers, G. Modeling Support in Information Systems Development, Thesis Publishers Amsterdam, 1991.

[Wimmer et al., 2007] Wimmer, M.; Strommer, M.; Kargl, H.; Kramler, G. Towards Model Transformation Generation By-Example. In: 40th Hawaii International Conference on System Sciences (HICSS 2007), Hawaii, 2007.

[Winter et al., 2002] Winter, M.; Zeidler, C.; Stich, C. The PECOS Software Process, Workshop on Component-based Software Development, 7th International Conference on Software Reuse (ICSR), Austin, Texas, USA, April, 2002, pp.07.

[Wohlin et al., 1999] Wohlin, C.; Runeson, P.; Höst, M.; Experimentation in Software Engineering: An Introduction, Springer. 1st edition, 1999.

[Wolf, 2002] Wolf, M. J. P. Genre and the Video Game, in Wolf M.J.P, The Medium of the Video Game, University of Texas Press, 2002.

[Zagal et al., 2005] Zagal, J.; Mateas, M.; Fernandez-Vara, C.; Hochhalter, B.; Lichti, N. Towards an Ontological Language for Game Analysis, In Proceedings of the Digital Interactive Games Research Association Conference (DiGRA 2005), Vancouver B.C., June, 2005.

[Zwicky, 1948] Zwicky, F. Morphological Astronomy. The Observatory, Vol. 68, No. 845, pp 121-143, 1948.

APPENDIX A. ARCADEX FEATURE MODEL

This Appendix presents the feature model created for the ArcadEx case study, during its multiple Game Domain Analysis iterations.

A.1 Root ArcadEx Game Feature

The root feature of the ArcadEx game SPL is the *ArcadEx Game* (Figure 81), whose sub-features are majorly a direct implication of the core game dimensions defined in Chapter 4. An ArcadEx game contains at least one player, a set of entities, graphics configurations, a physics modeling, a flow of screens, a set of events, input devices and other miscellaneous features. An ArcadEx game can optionally have audio capabilities and can present some artificial intelligence as well.

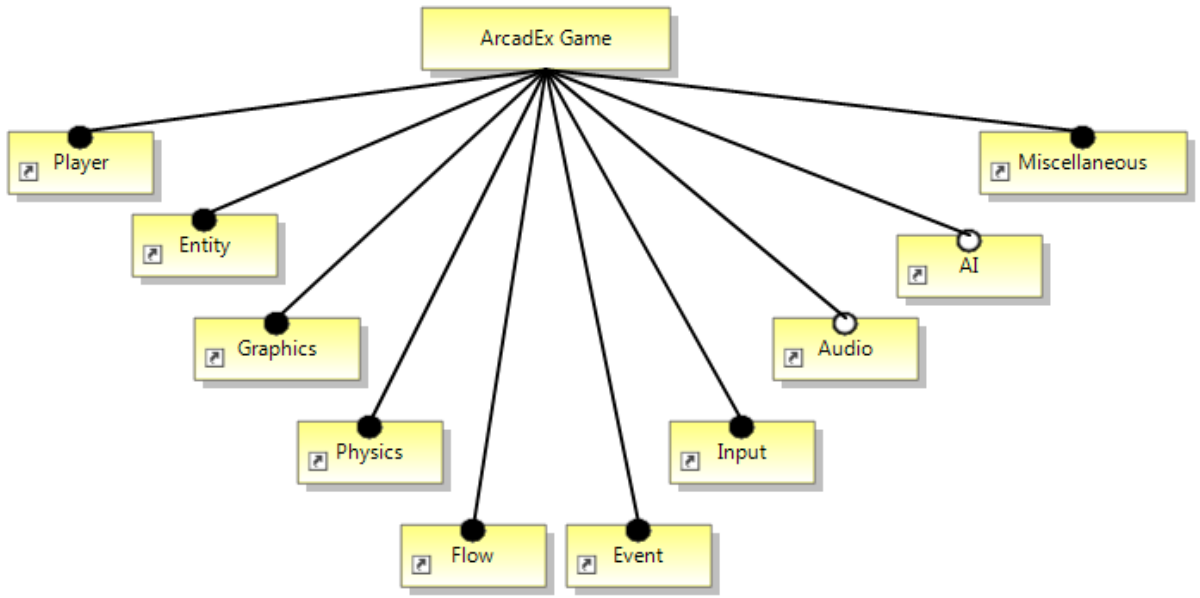


Figure 81 – Root ArcadEx Game feature model

A.2 Player Feature

Every ArcadEx game *Player* (Figure 82) has a score, a number of lives and at least one main character entity which is controlled by him or her. An ArcadEx game can have a single player mode and/or a multiplayer mode, in which up to four players play simultaneously and/or in a turn-based fashion. It is interesting to notice that a multi-player game can offer both modes to the players (obviously not at the same time): in turns **and** simultaneously.

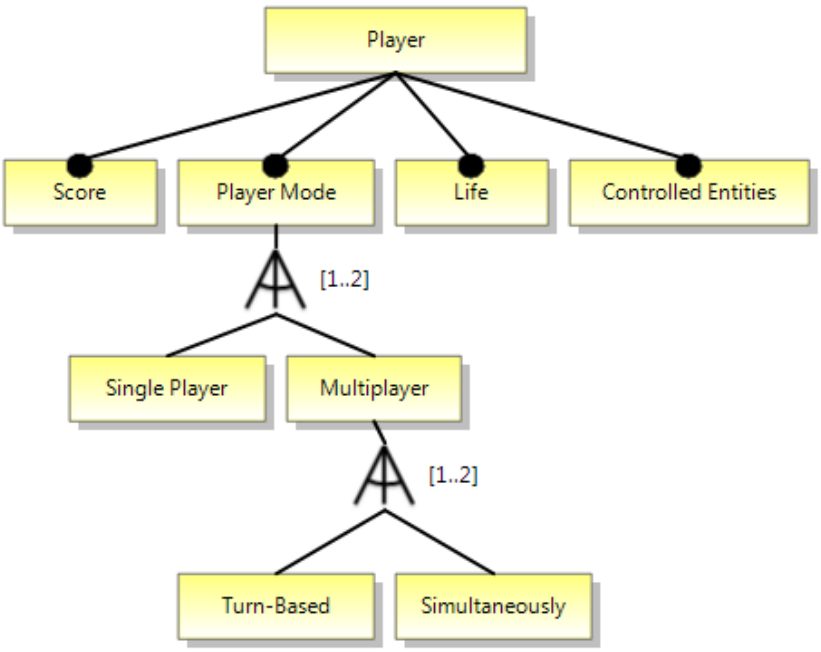


Figure 82 – Player feature model

A.3 Entity Feature

The *Entity* feature (Figure 83) is one of the most complex features of the ArcadEx domain. Entities represent the base classes of live beings and things of an ArcadEx game, while entity instances (next subsection) are their actual instances. The logical conclusion to create two distinct features (*Entity* and *Entity Instance*) was not straightforward and reveals the value of the Domain Analysis. If the ArcadEx game SPL was designed with only one Entity feature in mind, as it could be incorrectly considered the most intuitive way, automation opportunities would be missed and extra challenges would arise for creating the SPL assets. Of course, since this is an interactive process, it is always possible to adjust the feature model in later iterations.

Each entity has a set of states with rules describing when one state should change to another. Example of such states are *walking*, *jumping* or *dying*. A very common (but not mandatory) state is the *invincible* state, in which the entity cannot be killed or hurt by any event. For example, in the Berzerk game (Figure 84), the Evil Otto enemy is always in an invincible state. In the game 1942, on the other hand, the player's airplane becomes invincible for a short period of time when executing a looping move, but has other states as well.

An entity can have a special "role" in an ArcadEx game. Every game should have an entity belonging to the Main Character role, which has a player associated with it. Another role is the Non-Player Character (NPC), typically an enemy or other being who interacts with the main character. Common types of NPCs are level bosses, which must be defeated so

that a level can be finally completed, and “damager touch” NPCs, which may hurt or kill a main character if a collision between them happens. Finally, a game entity can also be an item (which generally awaits to be collected) or projectiles, which can be thrown at (and usually damage) enemies.

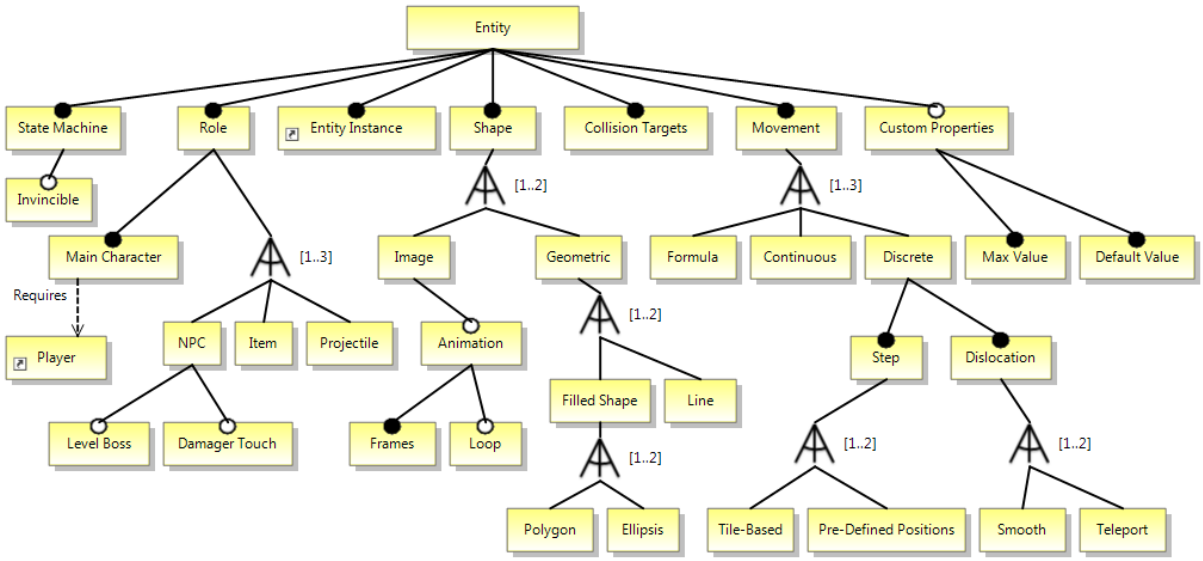


Figure 83 – Entity feature model

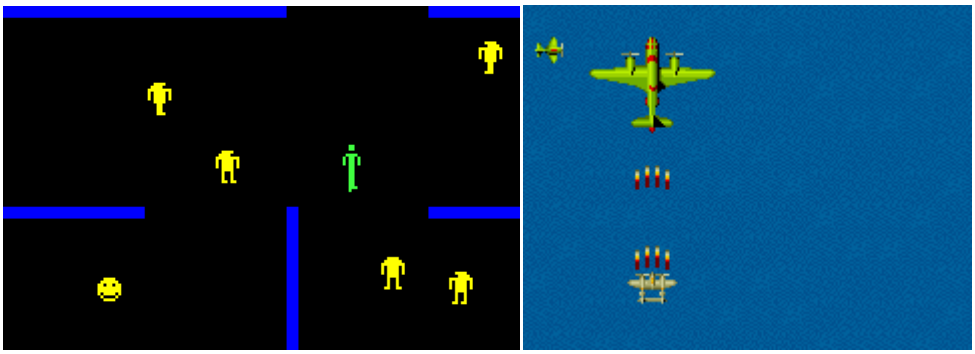


Figure 84 – Evil Otto (invincible, smiley face) and 1942 airplane (the bottom one)

A shape defines how an entity is displayed. It can be as simple as a geometric shape or images that can compose an animation. Regarding movement, an ArcadEx game entity can have a continuous movement or a discrete one. If the movement is step-based, the entity can simply teleport to the next position or smoothly move from the previous one to the next. Moreover, an entity movement, be it discrete or continuous, may be based in a formula. One example is a circular movement around an axis or center.

An entity may also cause something interesting to happen when it collides with other entities, but not all of them. Therefore, we say that an entity can have an “interest” to collide with some other entities of the game. Finally, entities may have a plethora or custom attributes, such as fuel, ammo, etc. Such attributes may have a maximum and a default value, and

their current value can be shown to the players via heads-up displays (HUDs), as described in the *Graphics* feature (Section A.5).

A.4 Entity Instance Feature

Entity instances (Figure 85) are actual instances of “beings and things” of an ArcadEx game. For example, while an entity called “Monster” defines how monsters look and behave, monster *instances* (containing a position, velocity, damage level and so on) are those actually seen by players in the screen. An object-oriented analogy also applies here, in the sense that entities are classes and entity instances are objects.

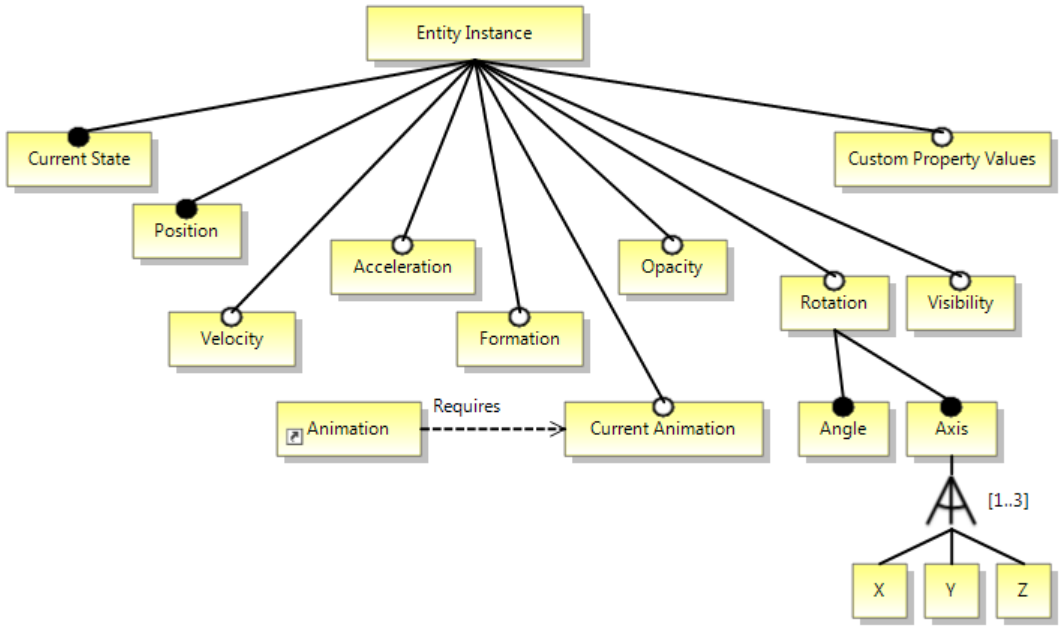


Figure 85 – Entity Instance feature model

Each entity instance has a current state related to the base Entity state machine, and a bi-dimensional position related to the game world (which should be converted into a screen position, in order to be rendered). Velocity and acceleration features are optional (albeit very common) and control the movement of specific instances. Rotation angle and axis can impact an entity instance display and movement as well. Although ArcadEx games are 2D games, there’s support for Z-axis rotation, which creates some interesting effects. Entity instances can also be visible or not to the player, or even something in between (defined by its opacity value). If the entity in which an instance is based has the animation feature enabled, the instance will have a current animation (and animation frame) in a given moment.

Entity instances may belong to a group or formation. For example, enemy space ships in the Galaga series attack in “waves” (Figure 86, left-hand side). Only when all enemies of a formation are destroyed, the player can proceed to the next level. An interesting

example of a behavior that relies on formations is when the instances of the formation decide to withdraw (or change their behavior somehow) when the player kills the majority of the formation instances, or the formation leader. Other example occurs in Feeding Frenzy (Figure 86, right-hand side): if the player is able to eat a complete “school of fishes” (an entity formation), he or she is eligible to win some bonuses.

Figure 86 – Galaga and Feeding Frenzy have “formations” of NPCs

Finally, for each custom attribute defined for the base entity on which an instance is based, the instance should have a value for it. For example, if the base entity defines a “hit points” positive integer attribute whose maximum value is 100, related entity instances will have, in a given moment, a number of hit points ranging from 0 to 100.

A.5 Graphics Feature

The *Graphics* feature (Figure 87) describes the graphical aspects of an ArcadEx game. With regard to its display, for example, the game can be windowed or presented in full screen. ArcadEx games should also have a screen resolution, consisting of a width and height, and can rely on particle systems to model visual effects such as explosions, fire, rain, etc.

The presence of at least a score Heads-up Display (HUD) is mandatory to ArcadEx games. HUDs can be used to show other relevant game information to the user, such as the high score, level data (current level number, for example) or even a radar containing the position of entities in the current game screen. Some HUDs can be specific to a numerical game or entity property, such as the number of lives, amount of ammo, fuel, etc. Finally, a HUD can be displayed in different ways: plain text, icons, progressive bars or any other custom representation provided by the SPL designers. Figure 88 shows a screenshot of the RallyX game, which is a good example to illustrate such different types of displays.

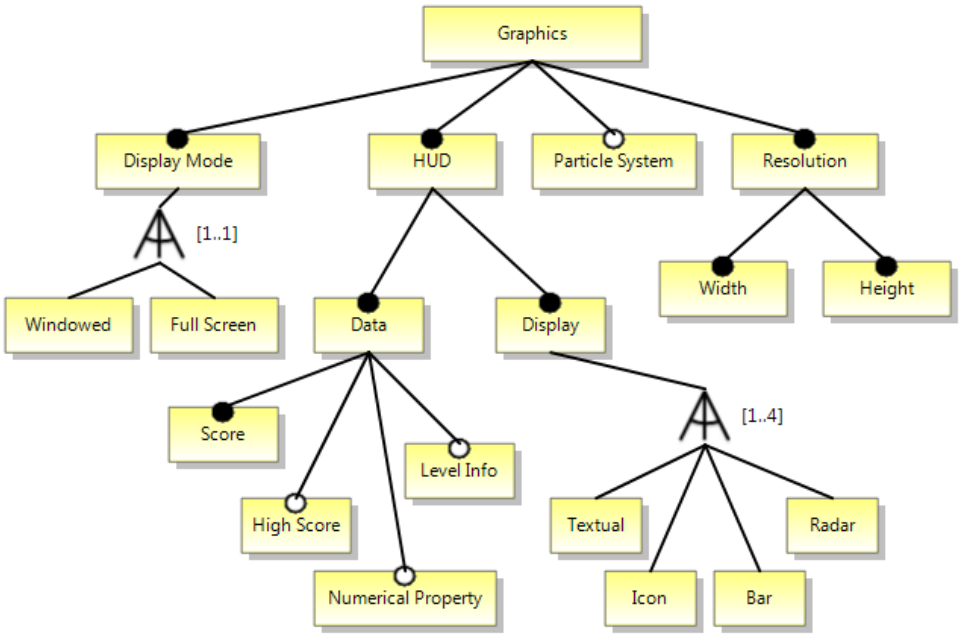


Figure 87 – Graphics feature model

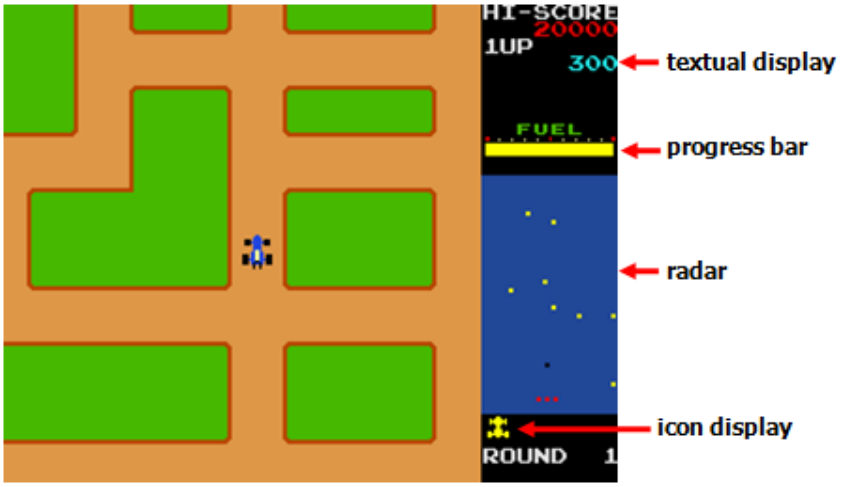


Figure 88 – RallyX and its different heads-up displays

A.6 Physics Feature

The *Physics* feature (Figure 89) of the ArcadEx domain reveals that every game should have a collision detection mechanism, be it based on regions (such as bounding-boxes techniques) or on pixels. Both techniques can also be simultaneously used. For example, a game can use pixel-based collision detection only on the overlapping area of two bounding boxes.

Entities of an ArcadEx game can also bounce against the walls and other entities, eventually reversing directions or even stopping after that. Finally, some attraction forces, such as gravity, can be part of the game physics as well.

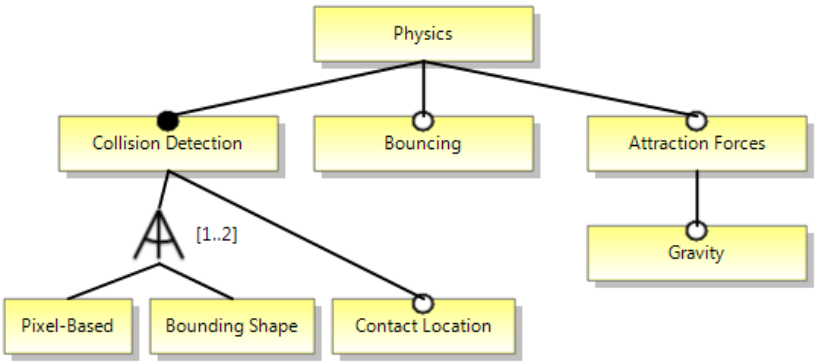


Figure 89 – Physics feature model

A.7 Flow Feature

The *Flow* feature (Figure 90) essentially describes how the game flows, i.e., the sequence of game screens presented to players. The purpose of a given game screen can be either to display information (such as an introduction screen, game over screen, instructions screen, etc.) or "host" actual game action (where the entities in fact behave and react to events). A special (optional) type of action screen is the "demo" screen, in which game action can be seen but there is no player interaction.

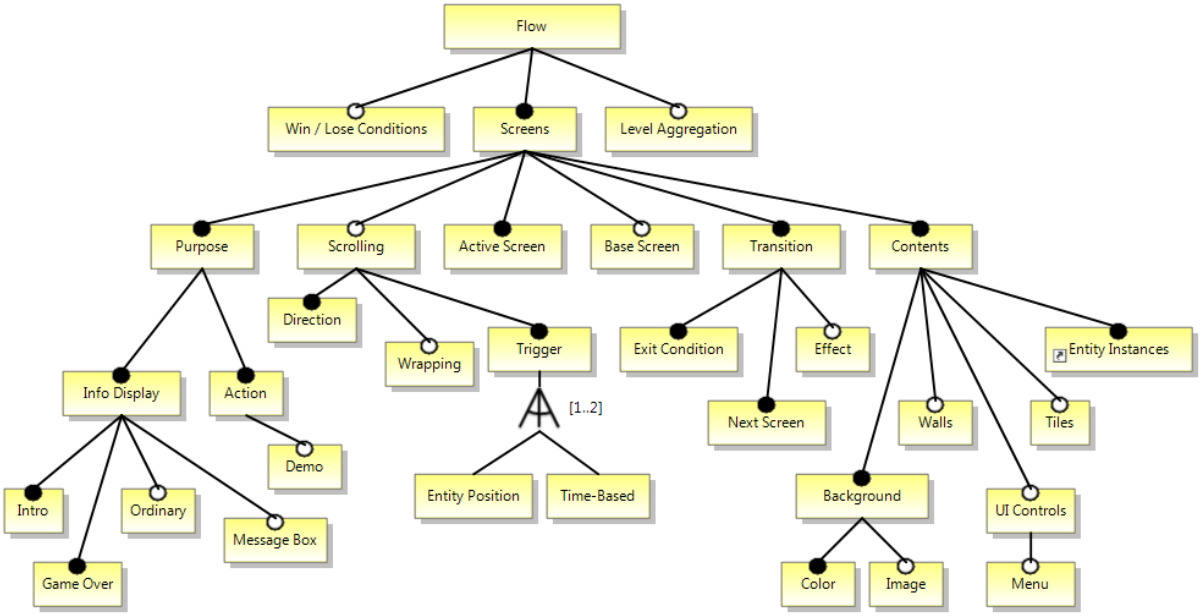


Figure 90 – Flow feature model

A screen can scroll in one or more directions, based on time or on the position of a given entity. If scrolling in a given direction can lead to the same area that the screen was when the scrolling originally started, the scroll is considered to be a "wrapping scroll".

ArcadEx game screens contain different types of contents. Entity instances can be present in a given screen, as well as walls, tiles and UI controls, such as heads-up displays. Every screen contains a background, which has a color and possibly an image.

Screens leads to other screens, therefore every screen has a set of exit conditions which trigger the transition to next screens. The transition can have an effect, such as fading out or in, for example. A (base) screen can also be inherited by other (child) screens, which reuse (or override) the parent screen behavior and contents. Finally, in a given game there should always be a screen which is the active (current) screen.

Still regarding the Flow feature, other interesting optional sub-features are level aggregation and win/lose conditions. Level aggregation refers to having a set of screens composing, together, a game level. Win/lose conditions are special conditions that leads the game flow to an end (the player wins the game or the game is over).

A.8 Event Feature

The *Event* feature (Figure 91) tells how an ArcadEx game behaves. Events are composed by triggers (a condition that fires the event) and reactions (game, player or entity behavior that is performed after the event is fired). Triggers can be of different types: input-based, timer-based or collision-based (entity-entity collision or entity-wall collision). Changes in the value of entities and players are also event triggers. Reactions, on the other hand, can do one or more game actions, such as setting the value of an entity/player property, playing a sound effect, removing or creating an entity instance, etc.

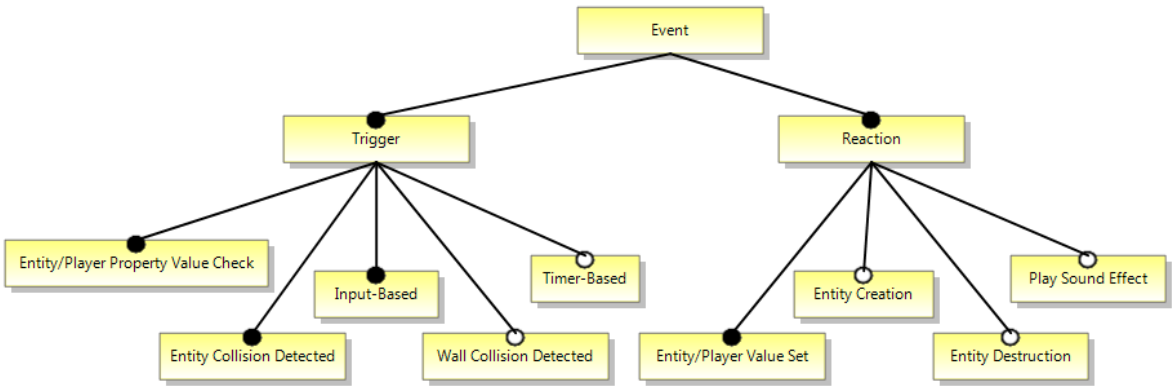


Figure 91 – Event feature model

A.9 Input Feature

The *Input* feature (Figure 92) is very straightforward and refers to how players provide input to the game. A combination of devices can be used, such as keyboard, joystick, mouse and dancing pads. An ArcadEx game can also have mappings from one input device to another

(for example, a set of keyboard keys that matches joystick buttons). That is quite useful to provide input alternatives to players when not all devices are present. Finally, input patterns refer to common bindings from input actions to the change of game/player/entity attributes. For example, joystick sticks or keyboard arrows can be bound to the velocity of an entity. In other common input pattern, a joystick stick is bound to entity rotation while other is bound to entity thrusting.

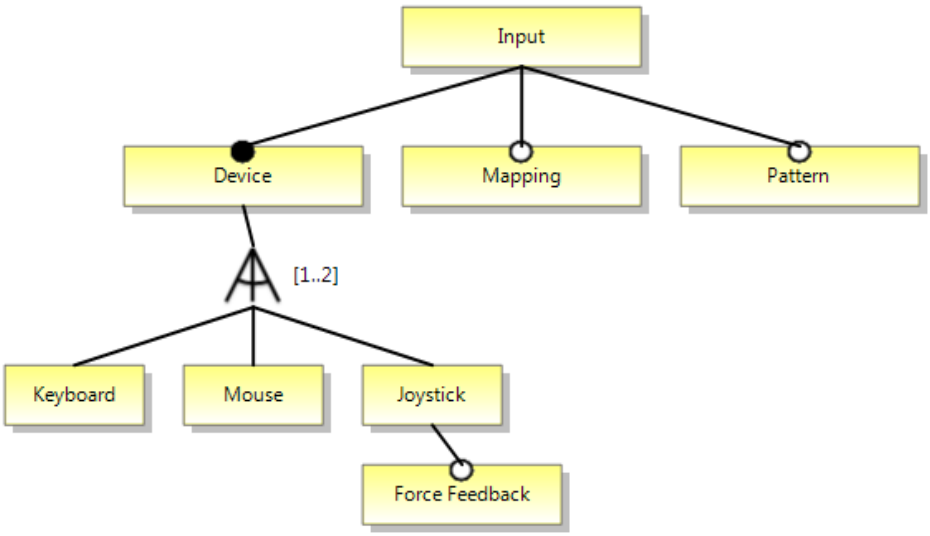


Figure 92 – Input feature model

A.10 Audio Feature

The *Audio* feature (Figure 93) is all about optionally adding sound effects and background music to an ArcadEx game. A sound effect can be created either from a resource (such as a .wav file which is transformed into a game "asset") or from a string using speech synthesizer. Background music is created from a resource and can optionally be played in loop.

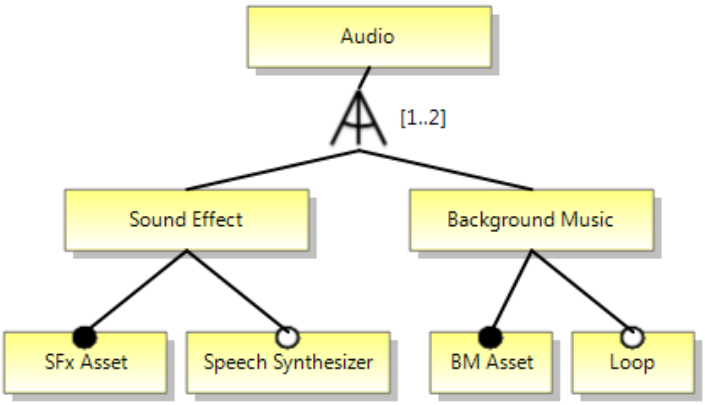


Figure 93 – Audio feature model

A.11 AI Feature

The *AI* feature (Figure 94) is composed by a series of optional sub-features related to the artificial intelligence of the game. An ArcadEx game, especially its entities, can be provided with intelligent logic to follow a path, find a path, chase an entity, run away from an entity and avoid a collision (evasion).

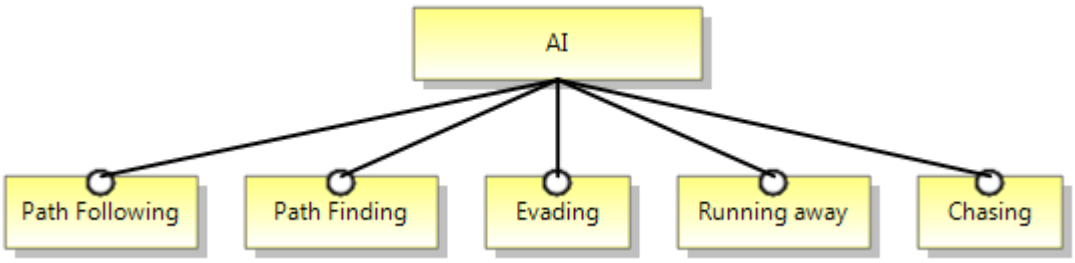


Figure 94 – AI feature model

A.12 Miscellaneous Feature

The Miscellaneous feature (Figure 95) groups together general metadata of an ArcadEx game, such as the game description (synopsis) and suggested rating, like those defined by the Entertainment Software Ratings Board (ESRB). It also comprises an optional runtime element of ArcadEx games: the generation of "randomness" to be consumed by other game elements (entities, events, AI, etc.).

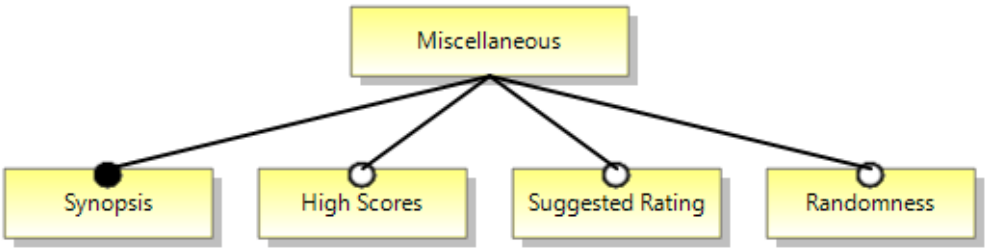


Figure 95 – Miscellaneous feature model

APPENDIX B. DOMAIN-SPECIFIC DEVELOPMENT EVALUATION

This appendix compiles the resources explored to bring more insights on how to evaluate Domain-Specific Development and MDD. Related work on metrics and evaluation approaches is presented.

Hermans et al. [2009] compiled a number of success factors for DSLs, which were used as metrics to evaluate a DSL (called ACA.NET) for the web services domain. Such factors are:

- Learnability: developers have to learn an extra language, which takes time and effort. Therefore a DSL should have a low learning curve ("high learnability"). Furthermore, as the domain changes the DSL has to evolve and developers need to stay up-to-date.
- Usability: tools and methods supporting the DSL should be easy and convenient to use.
- Expressiveness: using a DSL, domain-specific features should be implemented compactly; however, the language is specific to that domain and may limit the possible scenarios that can be expressed.
- Reusability: with a DSL, reuse is possible at model level, making it easier to reuse partial or even entire solutions, rather than pieces of source code.
- Development costs: a DSL helps developers to model domain concepts that otherwise are time-consuming to implement. The corresponding source code is generated automatically, lowering development costs and shortening time-to-market.
- Reliability: in addition to reducing development costs, automation of large parts of the development process leads to fewer errors.

A questionnaire composed of 20 questions was used by the authors in order to evaluate ACA.NET. Examples of such questions include *"Did the ACA.NET user interface help you modeling?"* and *"Do you agree ACA.NET makes implementing easier?"*.

While such an empirical study was useful to assess the specific language in question, at least two drawbacks can be identified. First, data collection was done through a questionnaire only, meaning that the study's final outcomes can be vulnerable to different interpretations from the subjects. Moreover, the answers to some questions rely on estimation from the subjects instead of measurements, which could lead to some lack of precision.

Kärnä et al. [2009] presents the evaluation of a domain-specific modeling (DSM) solution at Polar, a company in Finland focused on sports instruments and heart rate monitoring. Their evaluation approach combines developers' opinions with quantitative measurements of

the development process, focusing on three factors: developer productivity, product quality and the general usability of the tooling. They measured the modeling time for an application and *estimated* the manual implementation of the same application, resulting in a productivity increase of a factor of 7.5 to 9.1 times better with DSM. Developers also estimated the quality of the code and the quality of the design process to be significantly better with DSM. As for the return of investment, they estimate that the modeling investments are already worth even for one single product: while the product would take 23 days to build with their current development method, the modeling infra-structure development took 7.5 days and the development of a new product based on such an infra-structure takes 2.3 days.

Safa [2007] evaluates a domain-specific modeling solution for touch-screen UI applications. The evaluation was based on the generated code: since the models are able to generate 60% to 80% of the target sub-domains, the author claims a productivity increase of a factor of 3 to 5, although it seems no actual *time* measurements were taken to confirm that. The development of the modeling infra-structure is evaluated to take 3 man-days, which is 6 times more than developing a sub-domain instance from scratch. Therefore, the modeling investment break-evens after the sixth instance is developed using the modeling toolset. The experiments also demonstrated that a half-day training was enough to have a new hire to get used to the modeling toolset, and it took 3 days for the new hire to perform toolset extensions (creating a new code generator for a different target platform). However, such absolute data is not presented in a comparative context, i.e., the "learnability" *improvement* was not measured.

As it can be noticed, industry metrics tend to focus on two major factors: productivity and quality. Other metrics are also commonly applied (such as usability for Kärnä's case and target platform independency for Safa's) but their presence seems to depend on their importance and costs associated to their measurement. As Kärnä et al. [2009] point out, many good scientific research methods are simply too expensive and time-consuming for practical use in a commercial setting. Some of the characteristics of good empirical research, like a large number of participants to support generalization of the results, are not always even possible since there may only be a handful of developers using the particular language within the company. The authors also recognize that the industry does not usually have the time and resources to conduct other extensive analysis, such as building the same system twice with different development approaches, using parallel teams, analyzing large numbers of development tasks [Kieburtz et al., 2006] and focusing on development activities in detail with video recording, speaking while working or observing individual developers' actions [Wijers, 1991].

Kelly & Tolvanen [2008] created a DSL, generator and models to demonstrate the principles of DSM as part of the MetaEdit+ evaluation package, for the digital wristwatches development domain. The authors consider this as a pedagogical tool but large enough to provide realistic insights on into the multiple aspects of DSM. They concluded that once DSM is used, the generated code is shorter, simpler and has better quality over the "tangle of code that is normally found when similar embedded systems are hand coded". They ran a small experiment to identify whether the improvement happened due to DSM or only due to the greater attention spent on developing a good domain framework. A senior and a junior developer implemented two extensions, each, to an application in the domain. They first used models and code generation, then implemented the same extension manually. The authors report an improvement factor of 4 to 5.2 times when models are used. Nevertheless, the authors consider the sample size to be too small to be statistically significant – the number of subjects was limited to two developers, and the whole coding/modeling effort was only about 17 man-minutes. The authors report that the SPL investment took about 8 man-days, but the ROI was not calculated.

Kelly & Tolvanen [2008] also report the outcomes of industrial cases in which DSM was employed. One of them is the Call Processing Language (CPL), a DSL for IP telephone and call processing [Lennox et al., 2004]. They report a development effort improvement factor of 6 times. It is worth noticing that such a result was not obtained from a controlled experiment, but from comparing the modeling experience to earlier manual practices. They also believe that as the specification size and complexity of the end products become bigger, the improvement could be even better. The investment on the DSM approach was 11 man-days, but the ROI was not calculated because the domain platform was not ready yet: the DSL was built before any CPL servers were implemented.

Another industry case reported by Kelly & Tolvanen [2008] is a domain-specific solution for creating financial and insurance products. The CTO of the company in which the solution was applied reported improvements of "up to five times", although it is not clear whether controlled experiments were performed to measure that. The investment on the SPL approach was again 11 man-days, but since the case does not report how long it takes to create a product instance, the ROI was not calculated.

Home automation was another industry domain in which DSM was employed and reported by Kelly & Tolvanen [2008]. They mention that "applications which previously took a day could be made in an hour or two". That could be translated into an improvement factor of 4 to 8 times, or an average of 6 times. In fact, that is exactly the number reported by Kelly [2010] when referring to such a project. Assuming the products of such a domain take one day to be developed, the SPL investment break-evens by the fourth product instance.

A short white paper reporting a case study of the utilization of the MetaEdit+ DSM tool by Nokia is also available [MetaCase, 2000]. It reports that after a domain-specific solution was created, Nokia's productivity "increased by a factor of 10". The paper does not detail how that was measured, though. Other claimed benefits include a better focus on functionality instead of the implementation, full code generation from models, improved documentation and better support for learning and introducing new developers.

In the digital games development domain, Hernandez & Ortega [2010] ran a small experiment to evaluate their Eberos Game Modeling Language 2D, targeted at two-dimensional games. The first game, Pong, was implemented from scratch by the main author of the paper, who then modeled and implemented it using the DSL. The implications or mitigations for carrying knowledge from the first implementation effort (non-DSL) to the second one (with the DSL) were not discussed. For the second game, SpaceKatz (a Shoot'em up game), the non-DSL implementation was performed by a group of students, while the DSL version was implemented again by the main author of the paper. It is assumed that the final games are functionally equivalent. The evaluations measured, for the first game, a 29% of savings on programming effort (lines of code) and an 8.82% savings on programming time. For the second game, the reported savings were 86.4% on programming effort and 82.3% on programming time. According to the authors, the disparity among the results is justified by the argument that the more complex the target game is, the more applicable is the proposed DSL.

Kelly & Tolvanen [2008] conclude that the "normal" (expected) range of improvement for the use of domain-specific development in the industry is 5 to 10 times. That is somewhat aligned with the findings from Weiss & Lai [1999] at Lucent Technologies, focused on the telecommunications domain: they reported a productivity improvement factor of 3 to 10 times, depending on the product. The improvement factor observed in the experimental study performed for Domain-Specific Game Development (5.16 times, in average) falls under the expected range (see Chapter 5 for details). The fact our numbers are in the bottom half is an indication that there might still be room for improvement.

Hermans et al. [2009] compiled other studies about DSL evaluation in the literature but recognize that such studies are less common than model-driven engineering evaluation in general. According to them, although several papers can be found on advantages and disadvantages of using domain-specific languages [van Deursen & Klint, 1998], the DSL literature primarily provides anecdotal evidence for the claimed DSL usage benefits, often based on a handful of usage scenarios for the language in question. Therefore, more confidence can be gained from rigorous empirical studies in the area. Some of the DSL evaluation studies identified by Hermans et al. [2009] are:

- Batory et al. [2002] describe a case study where a DSL is used for simulations. They report improved extensibility and maintainability.
- Kieburtz et al. [2006] describe a series of experiments prototyping code generation using a DSL to generate code via templates.
- Herndon & Berzins [1988] report improvements, amongst which reduced time-to-market and improved maintainability due to the use of DSLs. Unfortunately, they lack to report how they come to their observations. Furthermore, their Kodiyak language has been used in only four cases.

Lucrédio [2009] mentions that the state-of-the art in Model-Driven Development evaluation has evidences that modeling is still considered a craftsmanship task. Developers in the area still depend on specialists' opinion to determine whether a model is good or not [France & Rumpe, 2007]. The author identified additional studies that investigate Model-Driven Development evaluation and the usage of metrics to increase confidence in the evaluation results. However, as it happens in the literature compiled by Hermans et al. [2009], the majority of the evaluation studies found by Lucrédio [2009] relate to Model-Driven Development in general, being only a few of them specifically targeted at DSLs:

- Guerra et al. [2008] created a DSL for defining metrics, with an additional focus on redesigning and refactoring models.
- Monperrus et al. [2008] argue that it is necessary to build specific tools to measure software each time a specific domain is implemented. They created a meta-tool that, given a set of metrics, generates tools that are able to measure such metrics.

With regards to Model-Driven Development in general, the following studies were identified as defining or discussing empirical work in MDD, including metrics and evaluation approaches [Lucrédio, 2009] [Hermans et al., 2009]. As it can be noticed, some aspects of the studies are still applicable to DSLs.

- Mohagheghi & Aagedal [2007] present aspects related to evaluating Model-Driven Development processes, such as complexity, tools and suitability for a particular domain.
- Pilgrim [2008] presents some metrics to determine the abstraction level of a model, based on properties such as number of attributes and the diagram size.
- The Modelware initiative [Modelware, 2006] also investigates metrics for domain-driven development, encompassing multiple engineering aspects such as the quality of models and generators.

- Genero et al. [2008] defined twelve metrics for the structural evaluation of entity-relationship (ER) models. According to them, the more attributes and relationships (1:1 and 1:N) a diagram has, the less comprehensive it is.
- Genero et al. [2007] concluded that the same structural properties of ER diagrams are also applicable to the maintainability of UML class diagrams.
- Muskens et al. [2004] and Kruchten [1995] defined UML metrics based on the "4 + 1" architectural view.
- Lange & Chaudron [2004] investigated the completeness and consistency of UML diagrams, pointing out issues such as nameless objects, classes without methods, interfaces without methods, abstract classes in sequence diagrams, classes that are not called in sequence diagrams, and messages between unrelated classes.
- Lange & Chaudron [2005] defined metrics to evaluate UML quality attributes, such as complexity, traceability, modularity, communication and esthetics.
- Baker et al. [2005] describe a large case study in which code and test cases were generated from models. They present numbers on increased productivity, quality and maintainability.
- White et al. [2005] also describe a case study in which code is generated. Their paper reports reduced effort on development and improved quality, but they only describe the results of one case.
- Staron [2006] used questionnaires to study the ideal situations for Model-Driven Development in an industry case study.

APPENDIX C. EXPERIMENT QUESTIONNAIRE

ID: ___

Date: ___/___/___

C.1 Personal Experience

What is your degree?

( ) B. Sc.

( ) M.Sc.

( ) Ph.D.

What is your software development experience in the industry (years)?

How do you evaluate your software modeling experience (MDD, UML, DSLs, etc.)?

( ) I don't know about it.

( ) I'm just aware of them and/or just had a couple of limited experiences.

( ) I eventually use it as part of my daily work.

( ) I constantly use it as part of my daily work.

How many games have you worked on previously?

C.2 Feedback

Did you have difficulties in understanding the specification of the game samples?

From a scale from 1-10, in which 1 means not helpful and 10 means very helpful, how do you evaluate the helpfulness of the toolset and its encompassing process for the development of games?

Please justify your answer for the helpfulness metric above.

From a scale from 1-10, in which 1 means very easy and 10 means very difficult, how do you evaluate the difficulty of using the toolset and its encompassing process for the development of games?

Please justify your answer for the difficulty metric above.

Please provide any additional comments.

APPENDIX D. EXPERIMENT CHEAT SHEET FOR XNA/FLATREDBALL

D.1 Game

D.1.1 How to define and initialize a game class

```
public class MyGame : Microsoft.Xna.Framework.Game
{
   public static GraphicsDeviceManager graphics;

   public MyGame()
   {
      graphics = new GraphicsDeviceManager(this);
      ...
   }

   protected override void Initialize()
   {
      base.Initialize();
      FlatRedBallServices.InitializeFlatRedBall(this, graphics);
      FlatRedBallServices.GraphicsOptions.TextureFilter = TextureFilter.None;
      SpriteManager.Camera.UsePixelCoordinates(true);
      Window.Title = "Game Title";
      ScreenManager.Start("QualifiedNameOfFirstScreen");
      ...
}
```

D.1.2 How to initialize game graphics in full screen mode

```
FlatRedBall.FlatRedBallServices.GraphicsOptions.SetFullScreen(width, height);
```

D.1.3 How to initialize a game graphics in windowed mode

```
FlatRedBall.FlatRedBallServices.GraphicsOptions.SetResolution(width, height);
```

D.1.4 How to run a game cycle

```
protected override void Update(GameTime gameTime)
{
    base.Update(gameTime);
    FlatRedBallServices.Update(gameTime);
    ScreenManager.Activity();
}
protected override void Draw(GameTime gameTime)
{
    base.Draw(gameTime);
    FlatRedBallServices.Draw();
}
```

D.1.5 How to start a game

```
static void Main(string[] args)
{
    using (MyGame game = new MyGame())
    {
        game.Run();
    }
}
```

D.2 Screens

D.2.1 How to define a screen class

```
using FlatRedBall.Math;
using FlatRedBall.Math.Geometry;
using FlatRedBall.Graphics;

public partial class IntroScreen : Screen
{
	// The base screen class already contains a list of sprites, but not walls.
	Protected PositionedObjectList<AxisAlignedRectangle> walls =
		new PositionedObjectList<AxisAlignedRectangle>();

	// An empty scene file (EmptyScreen.scnx) is provided.
	Public IntroScreen () : base("EmptyScreen.scnx", "IntroScreen") {...}
}
```

D.2.2 How to create a static background

```
using FlatRedBall;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework;

protected void CreateStaticBackground(string textureAssetName)
{
	Sprite backgroundSprite = new Sprite();
	backgroundSprite.Texture =
		FlatRedBallServices.Load<Texture2D>(textureAssetName);

	backgroundSprite.PixelSize = 0.5f;
	backgroundSprite.Position = new Vector3(
		game.Window.ClientBounds.Width / 2,
		game.Window.ClientBounds.Height / 2,
		-100);

	SpriteManager.AddSprite(backgroundSprite);
	mSprites.Add(backgroundSprite);
}
```

D.2.3 How to add walls to a screen

```
using FlatRedBall.Math.Geometry;

AxisAlignedRectangle wall = new AxisAlignedRectangle();
wall.SetFromAbsoluteEdges(top, bottom, left, right);
mAxisAlignedRectangles.Add(wall);
ShapeManager.AddAxisAlignedRectangle(wall).Visible = false;
```

D.2.4 How to transition to other screens

```
public override void Activity(bool firstTimeCalled)
{
	base.Activity(firstTimeCalled);
	...
	if (TransitionConditionIsMet())
	{
		NextScreen = "NextScreenQualifiedName";
		IsActivityFinished = true;
	}
}
```

D.3 Sprites

D.3.1 How to define a sprite class

```
public class MySprite : Sprite
{
	public MySprite()
	{
		SpriteManager.AddSprite(this);
		this.PixelSize = 0.5f;
		this.Position = new Vector3(posX, posY, 0);
		this.IgnoreAnimationChainTextureFlip = true;
		...
	}
}
```

D.3.2 How to define sprite animations

```
AnimationChain animationChain = new AnimationChain();
animationChain.Name = "AnimationChainName";

// animation 1, frame 1
Texture2D texture2D = FlatRedBallServices.Load<Texture2D>("texture1AssetPath");
AnimationFrame frame = new AnimationFrame(texture2D, frame1Length);
animationChain.Add(frame);

// animation 1, frame 2
texture2D = FlatRedBallServices.Load<Texture2D>("texture2AssetPath");
frame = new AnimationFrame(texture2D, frame2Length);
animationChain.Add(frame);

// optionally set more animations

this.AnimationChains.Add(animationChain);
this.SetAnimationChain(animationChain);
this.Animate = true;
```

D.3.3 How to flip a sprite texture

```
sprite.FlipHorizontal = true;
sprite.FlipVertical = true;
```

D.3.4 How to set a sprite's bounding circle based on its texture

```
using FlatRedBall.Math.Geometry;

Circle c = new Circle();
this.SetCollisionI;
this.CollisionCircle.Radius = texture.Width / 2;
```

D.3.5 How to set a sprite's bounding box based on its texture

```
using FlatRedBall.Math.Geometry;

float width = texture.Width;
float height = texture.Height;
AxisAlignedRectangle rect = new AxisAlignedRectangle();
rect.SetFromAbsoluteEdges(height / 2, -height/2, -width/2, width/2);
this.SetCollision(rect);
```

D.3.6 How to add sprites to a screen

```
MySprite sprite = new MySprite();
// optionally set the sprite's position, velocity, etc.
this.mSprites.Add(sprite); // "this" is the current screen
```

D.3.7 How to remove sprites from a screen

```
this.mSprites.Remove(sprite); // "this" is the current screen
SpriteManager.RemoveSprite(sprite);
```

D.3.8 How to bounce a sprite after a collision against a wall

```
foreach (AxisAlignedRectangle wall in mAxisAlignedRectangles)
{
   foreach (Sprite sprite in this.mSprites)
   {
      // sprite collision is based on a bounding circle
      if (sprite.CollisionCircle != null)
      {
         sprite.CollisionCircle.CollideAgainstBounce(wall, 0, 1, 1);
      }

      // sprite collision is based on a bounding box
      else if (sprite.CollisionAxisAlignedRectangle != null)
      {
         sprite.CollisionAxisAlignedRectangle.CollideAgainstBounce(wall, 0, 1, 1);
      }
   }
}
```

D.3.9 How to bounce a sprite after a collision against another sprite

```
// bounding circle case
if (sprite1.CollisionCircle.CollideAgainstBounce(
       sprite2.CollisionCircle, sprite1mass, sprite2mass, elasticity))
{
       ...
}

// bounding box case
if (sprite1.CollisionAxisAlignedRectangle.CollideAgainstBounce(
       sprite2.CollisionAxisAlignedRectangle, sprite1mass, sprite2mass, elasticity))
{
       ...
}
```

D.4 Audio

D.4.1 How to create and play a sound effect

```
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;

// the contentManager can be obtained from the "Content" property of a game
SoundEffect soundEffect = contentManager.Load<SoundEffect>(assetPath);
SoundEffectInstance soundEffectInstance = soundEffect.CreateInstance();
soundEffectInstance.IsLooped = isLoop;
soundEffectInstance.Play();
```

D.4.2 How to stop a sound effect

```
soundEffectInstance.Stop();
```

D.5 Input

D.5.1 How to verify whether a button was pushed

```
if (InputManager.Xbox360GamePads[playerIndex].ButtonPushed(
       Xbox360GamePad.Button.Start)) {...}
```

D.5.2 How to retrieve the position of an analog stick

```
Vector2 stickPosition =
       InputManager.Xbox360GamePads[playerIndex].LeftStick.Position;
```

D.5.3 How to apply input mapping

```
using Microsoft.Xna.Framework.Input;
using FlatRedBall.Input;

KeyboardButtonMap map = new KeyboardButtonMap();
map.LeftAnalogLeft = Keys.A;
map.LeftAnalogRight = Keys.D;
map.LeftAnalogUp = Keys.W;
map.LeftAnalogDown = Keys.S;
if (!InputManager.Xbox360GamePads[playerIndex].IsConnected)
{
       InputManager.Xbox360GamePads[playerIndex].ButtonMap = map;
}
```

D.6 Text

D.6.1 How to add display texts to a screen

```
using FlatRedBall.Graphics;
using Microsoft.Xna.Framework;

Text text = TextManager.AddText("hello");
text.Position = new Vector3(posX, posY, 0);
text.SetColor(red, green, blue);
text.AdjustPositionForPixelPerfectDrawing = true;
this.mTexts.Add(text); // "this" is the current screen
```

D.6.2 How to update display texts

```
text.DisplayText = "New Text";
```

D.7 Miscellaneous

D.7.1 How to retrieve a random number

```
int n = FlatRedBall.FlatRedBallServices.Random.Next(min, max);
```

D.7.2 How to get the elapsed time

```
double elapsedTime = FlatRedBall.TimeManager.CurrentTime;
```

APPENDIX E. EXPERIMENT CHECKLIST

E.1 Development tasks for the "Pong" game

☐	Intro screen (initial screen), with its own music and texture background
☐	Credits screen, with its own music and texture background
☐	Transition from Intro to Credits screen (and vice-versa) after 5 seconds
☐	Main screen, with its own music and texture background
☐	Transition from Intro screen to Main screen when player1 presses the Start button
☐	Player1 Wins and Player2 Wins screens, each with its own music and backgrounds
☐	Transition from Main to Player1 Wins screen when player1 score is 5 or more
☐	Transition from Main to Player2 Wins screen when player2 score is 5 or more
☐	Transition from any "Wins" screen to Intro screen, when player1 presses the Start button
☐	Input mapping of player1's Start button to the keyboard's Enter key.
☐	Input mapping of player1's left analog stick, left position, to the keyboard's left arrow key
☐	Input mapping of player1's left analog stick, right position, to the keyboard's right arrow key
☐	Input mapping of player2's left analog stick, left position, to the keyboard's A key
☐	Input mapping of player2's left analog stick, right position, to the keyboard's D key
☐	BluePaddle main character, representing the player1, bounding box collision shape, single texture
☐	BluePaddle's wall collision behavior: bounce
☐	BluePaddle's left/right movement controlled by the player1's left analog
☐	RedPaddle main character, representing the player2, bounding box collision shape, single texture
☐	RedPaddle's wall collision behavior: bounce
☐	RedPaddle's left/right movement controlled by the player2's left analog
☐	Ball NPC, bounding circle collision shape, animation composed by a sequence of ball textures
☐	Ball's default velocity is random: X = +/-[200,350], Y = +/-[200,350]
☐	Ball's wall collision behavior: if against left and right screen walls, bounce
☐	Ball's wall collision behavior: if against top and bottom screen walls, play goal sound, destroy the ball instance and create another in the middle of the screen
☐	Ball's wall collision behavior: if against top screen wall, also increment player1 score in 1
☐	Ball's wall collision behavior: if against bottom screen wall, also increment player2 score in 1
☐	If ball collide with BluePaddle, bounce the Ball and play the BluePaddle collision sound effect
☐	If ball collide with RedPaddle, bounce the Ball and play the RedPaddle collision sound effect
☐	Main screen contents: BluePaddle, RedPaddle, Ball and two textual HUDs with players' score

E.2 Development tasks for the “2942” game

	Intro screen (initial screen), with its own music and texture background
	Action screen, with its own music and vertically scrolling texture background
	Transition from Intro screen to Action screen when player1 presses the Start button
	Game Over screen, with its own music, texture background and player score in the bottom
	Transition from Action screen to Game Over screen when the main character dies
	Transition from Game Over screen to Intro screen when player1 presses the Start button
	Input mapping of player1’s Start button to the keyboard’s Enter key.
	Input mapping of player1’s left analog thumb stick to the keyboard’s arrow keys
	Input mapping of player1’s right trigger to the keyboard’s X key
	Input mapping of player1’s left trigger to the keyboard’s Z key
	Fighter main character, representation the player1, bounding box collision shape
	Fighter’s animations: main and exploding
	When the Fighter’s exploding animation ends, it is considered dead
	Fighter’s wall collision behavior: bounce
	Fighter’s movement controlled by the player1’s left analog thumb stick, not applicable to exploding state
	FighterBullet item, bounding circle collision shape, simple texture
	FighterBullet’s wall collision behavior: disappear
	Fighter fires a vertical FighterBullet when the right gamepad trigger is pressed, followed by a sound effect, not applicable to exploding state
	Fighter fires horizontal FighterBullets, one to each side, when the left gamepad trigger is pressed, followed by a sound effect, not applicable to exploding state
	DumbUfo NPC, bounding box collision shape
	DumbUfo’s animations: main and exploding
	When the DumbUfo’s exploding animation ends, it is considered dead
	DumbUfo’s wall collision behavior: disappear
	SmartUfo NPC, bounding box collision shape
	SmartUfo’s animations: main and exploding
	When the SmartUfo’s exploding animation ends, it is considered dead
	SmartUfo’s wall collision behavior: bounce against left and right walls, disappear against bottom
	UfoBullet item, bounding circle collision shape, animation composed by two textures
	UfoBullet’s wall collision behavior: disappear
	SmartUfos fire a UfoBullet every second; the bullet is targeted at the Fighter’s position
	When the Fighter collides against a UfoBullet, the Fighter explodes, the UfoBullet disappears and an explosion sound effect is played; not applicable if the fighter is already exploding
	When the Fighter collides against a UFO, both explode and an explosion sound effect is played; only applicable if the UFO and Fighter are not already exploding

	When a FighterBullet collides against a UFO, the bullet disappears, the UFO explodes, a sound effect is played and the player1 score increases in 50 points (DumbUfos) or 100 (SmartUfos)
	When an exploding UFO collides against a non-exploding UFO, the latter also explodes, and the explosion sound effect is played
	When a SmartUfo collides against another UFO and neither are exploding, the first one bounces
	Action screen initial contents: Fighter and yellow textual HUD with the player1 score
	In the Action screen, a DumbUfo is created every 2 seconds in the top of the screen, in a random horizontal position, with no horizontal speed a random negative vertical speed ([-400,-100]), followed by a sound effect
	In the Action screen, a SmartUfo is created every 5 seconds in the top of the screen, in a random horizontal position, with a random horizontal speed (+/- [300,400]), a random negative vertical speed ([-300,-100]), followed by a sound effect

“If life doesn't offer a game worth playing, then invent a new one.”

(Anthony J. D'Angelo)