Abstract— This paper presents an environment targeted at computer games development industrialization in the .NET Platform. A computer game product line definition and its architecture are specified and implemented by means of software factory assets, such as a visual designer based on a domain-specific language (DSL), semantic validators and code generators. The proposed approach is then illustrated and empirically validated by the creation of real world case studies. Finally, it is investigated how the proposed factory can be used as an edutainment platform for Computer Science 1 (CS1) and 2 (CS2) courses. The final intention is to empower game developers and designers to work more productively, with a higher level of abstraction and closer to their application domain.

Index Terms— domain-specific languages, software factories, Visual Studio, edutainment

1. INTRODUCTION

Digital games are one of the most profitable industries in the world. According to the ESA (Entertainment Software Association) [1], digital games (both computer and console games, along with the hardware required to play them) are responsible, annually, for more than ten billion dollars in sales. These impressive numbers are a match even for the movie industry, while studies show that more is spent in digital games than in musical entertainment [2].

The digital game industry, however, is as surrounded by success as it is continuously faced by challenges. Software development industrialization, an upcoming tendency entailed by the exponential growth of the total global demand for software, will present many new challenges to game development.

Studies reveal 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 [3]. For example, although game engines [4], state-of-the-art tools in game development, brought the benefits of Software Engineering and object-orientation towards game development automation, the abstraction level provided by them 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
According to the Microsoft Software Factories Initiative [3], a software factory can be defined as a product line that configures extensible development tools like Visual Studio [5] or Eclipse [6] with packaged content and guidance, carefully designed for building specific kinds of applications. This paper, therefore, addresses the need for computer games industrialization by proposing the implementation of a game software factory, called SharpLudus\(^1\), through the extension and customization of the Visual Studio IDE (Integrated Development Environment). The focus of the approach consists in embedding in the IDE a new visual designer, based on a visual domain-specific language, through which game designers and developers can specify a computer game with a higher level of abstraction and closer to their application domain.

Differently from traditional modeling approaches, however, such a specification is a live development process artifact. It can be used as input to other factory assets which customize the IDE, such as semantic validators and a code generator responsible for outputting computer games in the C# programming language [7] and, therefore, targeted at the .NET Platform [8].

The remainder of this paper is organized as follows. Section 2 presents a specification for the proposed game factory. Section 3 describes how a new modeling designer is embedded in Visual Studio .NET by means of a visual DSL. Section 4 details the code generator and framework (game engine) used by the factory. Section 5 presents a case study, named Ultimate Berzerk. Section 6 investigates how the proposed factory can be used as an edutainment platform. Section 7 concludes about the presented work and points out some future directions.

2. FACTORY SPECIFICATION

Before any factory assets (IDE extensions and customizations) can be described, it is important to detail what the software factory will be able to produce as well as how it will be done. This section covers such issues by defining a product line for the factory and a suggested low-level architecture for its products.

2.1 SharpLudus Product Line Definition

The great diversity of games created so far has turned the digital games universe into a very broad domain. Therefore, creating a software factory targeted at computer games development in general, ranging from 2D platform games to 3D flight simulators, constitutes a too broad and ineffective endeavor. In such a scenario, the production process and its tools would not be able to fully exploit factory benefits such as component reuse and assemblage. In other words, a narrower subset of games should be chosen. It is worth noticing, however, that assets common to all digital games do exist and could be made reusable, but such discussion is beyond the scope of the proposed project.

\(^1\) Resources and more information can be found in http://www.cin.ufpe.br/~sharpludus.
In the SharpLudus software factory, the adventure game genre was chosen for the factory product line. It can be described as a genre encompassing games which are set in a “world” usually made up of multiple, connected rooms or screens, involving a goal which is more complex than simply catching, shooting, capturing, or escaping, although completion of the objective may involve several or all of these. Additional information about the chosen factory domain, contemplating typical computer game features, is presented in Table 1.

As it can be observed, games that can be produced by the SharpLudus factory are somewhat simple. Although one may argue that such games do not present the characteristics of successful titles, this does not invalidate the research. Even though two-dimensional games are more and more being replaced by 3D games, they are still appreciated by several people [9]. Even today, the game industry still invests in 2D games. Many publishers, for example, have recently released nostalgic compilations of many successful 2D titles, such as Pac Man, Street Fighter, Dig Dug and so on. Examples of such compilations are the Capcom Classics Collection, Tecmo Arcade Classics, Taito Legends and Namco Museum 50th Anniversary [10].

Moreover, there are always opportunities and markets for “low-end games”, such as games for cell-phones or old arcade games. Low-budget, value-priced games (those selling in the U$6-U$15 range) seem to have a longer shelf-life than the big budget games [11]. The per-game profit is lower, of course, but the development and distribution costs are also much lower. Such a reality is already explored by some game industry initiatives, such as the Xbox Live Arcade [12], in which developers are able to create and sell simple Xbox games (traditional card games, nostalgia arcade games, puzzles and trivia quizzes). On the other hand, John Buchanan, Electronic Arts CTO, reveals in his keynote “Experiences, Stories and Video Games” [13] that current complex 3D games fail to provide immersion to game players, since the expectations around them are too high. He points out, for example, that users still prefer the first, two-dimensional version of NHL Hockey (a successful Electronic Arts title) in comparison with the latest 3D versions. In other words, the higher the graphics complexity introduced by a game, the harder its suspension of disbelief.

Finally, it is worth noticing that real commercial computer games demand a development effort which is out of the scope of this research. More than two years are usually required for completing such kind of games, and about 30 people are involved, which can equate to an investment of about U$9 million.

2.2 SharpLudus Product Line Definition

The software factory product line architecture is one of the most important assets produced by product line development. It describes the common high-level design features of the products that will be produced by the product line.

For the proposed game software factory, a top-level network topology is presented in Figure 1. It is actually very simple,
comprising only two components: the target client where the created game is supposed to be played and a web server that hosts a web service [14] which is responsible for storing and retrieving scores of games created through the factory.

The low-level architecture of games produced by the SharpLudus factory is split in the following three figures, due to space constraints. Figure 2 presents the main Game class, which contains collections of sound effect, game state and event objects. Figure 3 details the implementation of game states, their exit conditions and background music. Finally, Figure 4 provides an implementation overview for entities (basic game units such as non-playable characters and items) and sprites (animations).

3. EMBEDDING A NEW DESIGNER IN VISUAL STUDIO .NET

The most important extension that the SharpLudus game factory brings to Visual Studio .NET is the addition of a new visual designer to the IDE, through which the game developer can specify the main game configuration (resolution, screen mode, etc.), game states (rooms and information display screens) and their flow, exit conditions and properties. The designer includes the creation and manipulation of many game components (events, rooms, sprites, etc.), through enriched Visual Studio property windows.

In order to create such a modeling extension, however, a visual domain-specific language (DSL) [15] was conceived. Such a language, named SLGML (SharpLudus Game Modeling Language) is the foundation of the visual designer, which is actually a visual representation of the DSL. Its creation process is presented below.

3.1 Selecting a Language Workbench

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 [16]. 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.

The SharpLudus software factory uses Visual Studio 2005 Team System (VSTS) [5] language workbench technologies, called DSL Tools [17], to design and implement its visual DSLs. The DSL Tools provides a framework and toolset that enable partners to build custom visual designers and domain-specific language designers using Visual Studio. In other words, it is possible to extend VSTS by creating and plugging into it a new designer, based on a visual domain-specific language. Through the DSL Tools, one can create, edit and visualize metadata that is underpinned by a code framework, which makes it easier to define domain-specific schemas for metadata, and then to construct a custom graphical designer hosted in Visual Studio.

---

2 Suspension of disbelief is a willingness of a reader or viewer to suspend his or her critical faculties to the extent of ignoring inconsistencies and unreality so as to enjoy a work of fiction.
3.2 SLGML Design

A visual domain-specific language is composed by different elements: a graph of concepts (or “classes”), relationships (comprising roles, cardinality, etc.), attributes and so on. A common approach to specify a visual DSL is to use a meta-modeling language, such as the GOPRR (Graph Object Property Relationship and Role) language [18] or the Microsoft DSL Tools meta-modeling language. This last one was used by the SharpLudus software factory to specify its visual DSLs.

A meta-modeling language is used to build a meta-model, which describes a modeling language (such as SLGML), similarly to the way a model describes a system. The root concepts of the SLGML meta-model are presented in Figure 5 (deeper concepts and relationships are not shown due to space constraints).

The SharpLudusGame is the root domain concept of the SLGML DSL. It is related to six top-level elements:

- **AudioComponent**: an abstract concept representing every sound that can be reproduced in a SharpLudus game. It is specialized by SoundEffect and BackgroundMusic concepts.
- **Entity**: an abstract concept which is the base unit of a SharpLudus game design. It is anything that can react with anything else in any way. It is specialized by MainCharacter, NPC (non-playable character) and Item concepts.
- **EntityInstance**: represents an instance of an entity, containing information such as position, speed, number of remaining hit points, etc.
- **Sprite**: represents an animation that can be assigned to entities (such as “main character walking”, “main character jumping”, etc.). It is composed by a Frame collection and it may loop after it ends.
- **Event**: represents a special condition that occurs to a SharpLudus game, fired by one or more Triggers (such as “collision between the main character and a specific item”), and that cause one or more Reactions (such as “add item to main character inventory”). The CustomTrigger and CustomReaction concepts, which inherit from Trigger and Reaction respectively, make it possible to create custom-made events.
- **GameState**: abstract concept which represents the game flow. It is specialized by InfoDisplay and Room concepts. InfoDisplays contain a Purpose attribute, indicating if it is an introduction, game over or ordinary information display screen. Finally, each GameState contains an ExitCondition collection, which tells when the game should move from one state to another.

3.3 SLGML Syntax and Visual Editor

Language syntax defines how the language elements appear in a concrete, human-usable form. In the case of visual languages, the syntax is not only purely textual; it combines graphics, text and conventions by which users may interact with the graphics and the text under the auspices of tools. The visual syntax elements of SLGML are presented in Table 2.
The game designer can add *InfoDisplays*, *Rooms* and *Transitions* to a SLGML model by drag-and-drop operations from the IDE toolbox, which is presented in Figure 6. Other language concepts (such as events, sprites, transition exit conditions, information display purpose or the entities belonging to a room) can be manipulated by the game designer through the IDE Properties window (Figure 7), which is sensitive to the model element in focus. Depending on the element selected, the Properties window can launch custom property editors, such as the SharpLudus software factory designers (room designer, event designer, sprite designer, etc.), which will be better illustrated during the explanation of the case study (Section 5).

The game designer is also able to visualize the elements of its current SLGML diagram in a more organized, hierarchical way, by using a tool window called SLGML Explorer (Figure 8). It is possible to add and delete items from this window, as well as select one of its elements and edit its properties through the Properties window.

### 3.4 Semantic Validators

Besides aiding the game designer with visual edition features, SLGML modeling experience also ensures that the DSL semantics are respected by the game designer. This is done through semantic validators, which are associated to a domain concept and can output validation issues to the Visual Studio Error List, as presented in Figure 9. Double-clicking in an Error List error automatically makes Visual Studio to focus on the model element which is the error source.

In the specific case of the Microsoft DSL Tools language workbench, semantic validators can be created programmatically. Since each domain concept generates a C# partial class, all the language designer needs to do is to implement a method in the desired concept partial class, which checks for special error conditions and logs the error, as shown in Figure 10. C# method attributes [7] can be used to specify when the validators should run: after opening a SLGML model, when saving it or explicitly through a context menu command, for example.

Some examples of SLGML semantic rules which can be enforced through validators are presented below:

- A game state transition must have at least one exit condition;
- A SharpLudus game should contain one main character;
- A SharpLudus game should contain one introduction InfoDisplay;
- A SharpLudus game should contain one game over InfoDisplay;
- An entity should contain at least one sprite;
- All game states should be reachable.

Figure 11 gathers all features presented so far and illustrates a complete overview of the SLGML modeling experience, hosted in Visual Studio .NET.
4. CODE GENERATOR AND FRAMEWORK

As pointed out by Deursen, Klinten and Visser [15], a DSL can be made much more useful if it relies on a framework or library which implements the semantic notions and a compiler that translates DSL programs to a sequence of framework calls. In the SharpLudus software factory context, such framework is a game engine, the compiler is a code generator (which outputs code that consumes the game engine) and the DSL programs to be translated are actually SLGML models.

Considering the .NET Platform scenario for the SharpLudus software factory, this research reused a public game engine made available by the DigiPen Institute of Technology [19]. The engine is developed in C# and consumes the DirectX multimedia API [20]. Some of its features include:

- Creation and dynamic manipulation of game entities, including the assignment of sprites and movement;
- Keyboard interaction;
- Sound effects support;
- Text manipulation.

However, the engine was considerably modified and extended to make it compliant with the SharpLudus software factory product line domain. For example, the original version of the engine does not present any features related to game states or game events, and treats all entities equally. Besides that, its architecture was modified to become compliant with the one presented in the SharpLudus software factory product line (Section 2.2).

The extensions increased the amount of the original game engine code by 40%. Its final version, used by the SharpLudus software factory, contains 45 classes and more than 2340 lines of source code (excluding commented and empty lines), while the original version contains 20 classes and 1700 lines of source code.

The SLGML code generator receives a SLGML model as input and generates the following C# classes as output:

- **AudioComponents**, responsible for providing sound effect and background music objects via C# properties following the Singleton [21] design pattern.
- **Sprites**, responsible for providing sprite objects via C# properties. Differently from some other classes, the Singleton design pattern is not used in this case, since multiple sprite instances are required to store different animation information for each sprite, such as the current sprite frame.
- **One class for each Entity concept specified by the game designer.** Such a class inherits from the Item, MainCharacter or NPC game engine classes.
- **EntityInstances**, responsible for providing entity instance objects via C# properties following the Singleton design pattern.
• **States**, responsible for providing room and information display screen objects via C# properties following the Singleton design pattern. The change of game states is implemented internally by the game engine through the State design pattern [21].

• **Program**, which contains the Main method and is responsible for instantiating and running the game.

• The main game class, whose name corresponds to the *Name* property of the *SharpLudusGame* root concept. Such a class inherits from the *Game* game engine class. The code generator also creates a method in this class named *InitializeResources*, where the game configuration is set and game events are registered.

The developer is not allowed to make changes to the generated code. However, the generated code is based on C# partial classes [7] and, therefore, can be complemented by the developer. To such kind of deeper extensions, developers are not required to read nor understand the generated code, but they should be familiar with the game engine architecture.

Besides the generated classes, the IDE project additionally provides two initial classes which are not re-generated: *CustomTriggers* and *CustomReactions*. Developers can add their own methods to these classes in order to implement custom triggers and custom actions specified by the game designer in the SLGML model. A generated code example for a SharpLudus game is illustrated by the class diagram presented in Figure 12. The relationships between the classes are not presented due to simplicity purposes, but they are also generated.

In the Microsoft DSL Tools language workbench, artifact generators, including code generators, are implemented by a text template transformation toolkit. Such a toolkit provides a script language, whose syntax is similar to C#, to manipulate model concepts.

Figure 13 presents an excerpt of the SLGML code generator which is responsible for generating the *Sprites* class. Text between the `<#` and `#>` tags contains script commands, used to manipulate the visual DSL concepts. For example, a *foreach* loop is used to iterate through all sprites defined in the SLGML model by the game designer in order to create C# properties for them. Text between the tags `<#=` and `#>` are used to evaluate some script expressions. For example, the name of the C# property to be generated is the evaluation result of the expression *spriteName*, which is a string previously declared and assigned in the script. Finally, all text outside tags is just copied to the generated artifact.

Considering both script instructions and raw text to be copied to the generated classes, the SLGML code generator implemented with the DSL Tools text template transformation toolkit contains 410 non-empty lines of code. It is worth noticing that the generator used other toolkit features, whose explanation is beyond the scope of this paper. More information about the toolkit can be found in the MSDN DSL Tools workbench [17].
5. CASE STUDY: ULTIMATE BERZERK

This section presents the creation of a real world adventure game named Ultimate Berzerk, which illustrates the use of the SharpLudus software factory. In Ultimate Berzerk, the player controls a main character, using the arrow keys, to move around a maze composed by connected rooms. Once the player collects a special item (named Weapon), the spacebar can be used to shoot fireballs against enemies. Enemies may have special behaviors (not originally provided by the factory). The goal of the game is to collect the Diamond item and find the exit sign. A screenshot of the game is presented in Figure 14.

5.1 Designing the Game

By modeling a SLGML diagram and launching factory designers from the Properties window, the game designer is able to visually create the majority of the game: sprites, entities, events, audio components, etc. For example, Figure 15 presents one of the screens of the sprite designer. This designer is launched from the Sprites property of a SharpLudus game and makes it possible for the game designer to specify frames and information such as if the animation will loop or not.

Figure 16, on the other hand, presents the room designer, where previously created sprites can be assigned to room as tiles and entity instances (such as enemies and items) can be added to rooms based on previously created entities.

Finally, Figure 17 presents the event designer, through which the game designer can specify the rules that govern the Ultimate Berzerk world, by means of event triggers and event reactions.

5.2 Custom Developer Code

Although the SharpLudus factory provides many interesting predefined event triggers and reactions, there will always be some situations where the game designer needs more complex or non-usual behaviors. For example, suppose the game designer wants the Knight entity to shoot a fireball at every 3 seconds. The SharpLudus factory does not provide any built-in trigger for that. However, by adding a custom trigger in the event designer and specifying a method name, the game designer delegates the task of creating such a trigger to a developer.

As previously pointed out, the game project in the IDE contains two predefined classes, called CustomTriggers and CustomReactions. Supposing that the game designer specified in the SLGML model a custom trigger named KnightShotTrigger to define when the Knight entity should shoot a fireball, a developer would only have to add the code presented in Figure 18 to the CustomTriggers class.

While creating custom code, developers are provided with full IDE editor support, as shown in Figure 19. The model elements, once the code generator has run, become accessible in code as strongly-typed C# classes and properties.
The ability to customize the game under development with code is not restricted to custom triggers and custom reactions. The behavior of any entity, state or sprite can be extended or modified by the use of partial classes.

Suppose, for example, that the game designer wants to create a special movement type for Diamond Guardian NPC (non-playable character) instances, making them bounce as they touch the tiles of the room where they are placed (Figure 14). A developer can easily accomplish this task by adding to the IDE project a class named DiamondGuardian and assign to it the partial modifier. This will make the final DiamondGuardian class to be composed by both the factory generated code and the developer added code.

By overriding the Update method of the DiamondGuardian class, as shown in Figure 20, it is possible to define a custom movement type for this entity. The code shown in the figure moves the Diamond Guardian up, if it is stationary, and inverts its vertical direction as it reaches the desired upper and lower bounds of the room.

5.3 Discussion: Factory Effectiveness for Ultimate Berzerk

Although Ultimate Berzerk is a relatively simple game, with a few rooms to be investigated by the main character, its development explored many interesting SharpLudus software factories assets and features that illustrate how the factory can be used to create real world games. Extending Ultimate Berzerk to a game with a better gameplay and replay value is just a question of adding more model elements which reflect the creativity of the game designer.

The automation and productivity provided by the SLGML modeling experience, its code generator and consumed game engine is evident: in less than one hour of development effort, 16 classes and almost 3900 lines of source code were automatically generated for the development team. What is most important is that such lines of source code mainly present routine, boring and error-prone tasks, such as assigning pictures to frames, frames to sprites, sprites to entities, entities to rooms, rooms to the game, events to the game and so on.

By using the SharpLudus software factory, especially the visual designers, the authors empirically concluded that the development team experience was made more intuitive and accurate, although more formal experiments should be carried out. At the same time, when more complex behavior was required (such as specifying the Diamond Guardian movement or creating custom event triggers) the factory was flexible to allow developers to add their own code to the solution, using all of the benefits of an object-oriented programming language and being aided by IDE features such as editor support, debug support and so on. This contrasts the development experience of visual-only game development tools, where weak script languages should be used under an environment which was not originally conceived for codification.

Considering the generated code along with the consumed game engine, it can be concluded that the SharpLudus software factory is able to provide, in one hour, a development experience which would require, from scratch, the implementation of 61
classes and more than 6200 lines of source code.

5.4 Other Case Studies

In order to validate the SharpLudus software factory as capable of creating not only a single product, but a family of products, other case studies were developed as well. Detailing such case studies is out of the scope of this paper, but some screenshots are provided in Figure 21 (Stellar Quest game) and Figure 22 (Tank Brigade game).

In spite of belonging to the same product line definition, both games present unique variabilities (such as world rules, presentation style, influence of collected items and state flow), which made it possible to empirically conclude that the SharpLudus software factory is successful in creating distinct products belonging to the same domain. Once more, however, the next step towards measuring the factory effectiveness refers to carrying out more detailed experiments with more formal metrics.

6. SHARPLUDUS AS AN EDUTAINMENT PLATFORM

Once understood what the SharpLudus software factory is, what can be generated by it and its benefits, this section highlights an interesting field in which the authors of this research believe SharpLudus can be applied: the teaching of concepts related to Computer Science 1 (CS1) and Computer Science 2 (CS2) courses. In other words, this section investigates how the factory can be used to offer to educators and students an edutainment environment targeted at the creation of motivating game challenges as CS1/CS2 laboratory exercises.

Regarding the content of CS1 courses, the research considered the following topics: basic programming language concepts and fundamentals, variables, operators, qualifiers, scope, lifetime, primitive types, type conversion, expressions, flow control, arrays, basic oriented programming concepts, classes, objects, fields, methods, inheritance, dynamic binding, polymorphism, abstract classes and interfaces. Typical topics of CS2 courses are more advanced, contemplating reuse, layered architectures, files/streams, collections, concurrency, I/O, graphical user interfaces (GUIs), networking and event-driven programming.

6.1 CS Education through Games

The creation of a CS1/CS2 education environment based on digital games has some unique requirements. First of all, the environment should be expressive enough to allow students to explore and learn the fundamentals and benefits of object-oriented programming languages and programming concepts. Moreover, it should provide a rich user experience, which enables the intuitive usage of digital game concepts. Finally, the environment should be reusable, similar to a small software factory which exploits the concept of “economies of scope” [3], where multiple similar but distinct designs and products are produced collectively, rather than individually. We believe that current approaches available to CS1/CS2 educators lack to satisfy all of the above demands.
On one hand, although popular tools intended to help beginner and amateur game designers/programmers may offer visual aids for programming language syntax and constructs (variable declaration, variable assignment, instruction blocks, if-then-else branches, loops, etc.), as shown in Figure 23, they do not scale and more elaborated programming concepts, such as class inheritance and polymorphism, are overlooked.

In addition, languages embedded in visual game creation tools (Figure 24), such as the Game Maker Language (GML) [22] or the Ruby Game Scripting System (RGSS), used in RPG Maker [23], are actually script languages hosted in an environment which was not originally conceived for complex codification. In short, they do not provide to end-users a chance to learn true object-oriented programming concepts and languages, with the support of robust integrated development environments with full editor and debugging support.

On the other hand, although approaches such as the Microsoft XNA Game Studio Express [24] are built up on more robust development environments, the game modeling experience provided by such tools would be made richer and more intuitive if users were provided with distinct visual aids for different game concepts and relationships, which would be added to models through the Toolbox or enhanced property windows, depending on their nature. Moreover, since such tools are generic by design, capable of generating games belonging to different game genres (adventure, shooters, RPGs, etc.), they do not provide built-in domain-specific assets which would enhance development experience focused on a specific game genre.

Game engines, which are the state-of-the art solution for creating computer games, also present some negative issues to teach CS1/CS2 concepts. The learning curve for mastering such tools is somewhat high. The demands for understanding a game engine architecture, interaction paradigm and programming peculiarities can turn their use into an experience whose efforts are beyond the desired and available by CS1/CS2 educators and students. In fact, 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 [25].

Finally, some successful approaches towards motivating Computer Science education through computer games development, such as ImagineCup’s Project Hoshimi Programming Battle [26], are based on SDKs which require a not so straightforward learning curve. Moreover, they are targeted at end-users (programmers), not at game product line developers, therefore not exploring the opportunity to use such approaches in a large scale to create ready-to-consume “game challenges” as Computer Science exercises.

In summary, visual game creation tools are more user-friendly, while unprovided of more interesting CS1/CS2 learning capabilities. Game engines and more sophisticated game development tools, on the other hand, are more complex to consume, while representing a more solid foundation where a CS1/CS2 edutainment platform can be built. Our proposal, therefore, is
aimed at bridging the gap between such approaches, providing through SharpLudus an intuitive yet rich environment to allow CS educators to create their own laboratory exercises, integrating programming language concepts and computer games. Students will also be able to use the tool to learn basic object-orientation concepts while creating their own games.

6.2 A New Edutainment Platform

We intend to use the SharpLudus game factory as the technical and conceptual basis for creating a new edutainment environment. The purpose is to move the factory a stage beyond, encompassing the requirements and addressing the drawbacks presented in the previous subsection. Three questions that immediately arise from such purpose are:

• How CS1/CS2 teaching dynamics will look like with the new environment?
• How to evaluate the results?
• Which extensions should be done to the current factory?

Regarding the teaching dynamics, we argue that with the new proposed edutainment environment, educators will be able to visually create game challenges (motivating laboratory exercises), while students will solve such challenges by complementing factory-generated partial classes with their own code, as well as adding new game behavior (such as game entities, rooms, etc.) by visually extending the models created by the educators. Students are also allowed to create their own games in order to consolidate the learning of CS1/CS2 concepts.

In CS1 courses, educators will be able to create and guide students on the creation of game challenges targeted at: variables, primitive types, operators, type conversion, expressions, flow control (if-then-else branches, for loops, etc.), classes, objects, fields, methods, arrays. More advanced concepts such as inheritance, abstract classes, dynamic binding and polymorphism might be used as well.

In CS2 courses, some of the target subjects identified as capable of being assisted by the environment are: exception handling, collections, reuse, concurrency, networking and I/O (such as file/streams).

For instance, an educator intending to teach the concept of “if-then-else” expressions can visually design, in less than one hour, a turn-based version of the Blobbit Dash game (Figure 25). In such a 2D game, the main character needs to collect all “Blobbit babies” in the game map and then find the exit, before he runs out of time and avoiding monsters on his way.

Educators, however, might leave the implementation of the movement logic of monsters to half of the students, and the main character movement logic to the other half. Students may have access to functions/methods to determine what is on their surroundings (a baby, an enemy, the exit, etc.). Therefore, by using the results of such functions/methods in if-then-else statements, students can properly decide where to drive the entity under their responsibility (monsters or main character).

Finally, educators have many possibilities to motivate the learning: they can divide the students in pairs, where each pair will
be composed by a student responsible for the monsters and another for the main character. Next, competitions among students can be created and the code of some students can (anonymously) be presented to the whole class. Finally, educators could ask students to create similar or simpler versions of games with the same purpose as homework.

Regarding the evaluation of the project results, a mix of some human-computer interaction approaches can be used. First of all, a control group study can be carried out, where CS1/CS2 students are divided in control and experimental groups. While the “game challenge learning approach” is offered to the experimental group, the control group uses traditional approaches. Results by means of satisfaction and “learnability” variables can then be measured from both educators and students, through task analysis approaches [27] and qualitative/quantitative interviews.

Finally, in order to make the SharpLudus factory compliant to CS1/CS2 (more specifically object-orientation) educational purposes, the following extensions were identified:

- Review and eventually update the concepts and relationships of the visual domain-specific language used to model games, considering now educative purposes. For example, “hooks” between the modeling and the programming experiences should be made clearer and easier to introduce in game challenges.

- Enrich the user experience, considering now that the game factory will be used by educators and students with little or no game development experience. One possibility is to investigate how Microsoft’s Guidance Automation Toolkit [28] (GAT) could be used to provide wizards and automation to help such users to accomplish their goals.

- Enhance factory documentation to assist educators and students on the creation of game challenges, encompassing three different levels: formal documentation, tutorials and samples. The idea is to have such documentation levels integrated with the environment, as it happens with the Microsoft Visual Studio Team System [5] documentation.

- Integrate a more powerful game engine to the factory, in order to enable the productive creation of more interesting games, which will subsequently improve students’ satisfaction. The Torque X Game Engine [29] is being considered.

- Change the code generators of the factory to be compliant with the factory modifications (DSLs, target game engine, etc.).

- Provide out-of-the-box libraries of images and sound effects that could be used to the creation of games. This issue is very important since the appeal of the game is directly related to its graphics and sound.

7. CONCLUSION

This paper presented a study, illustrated with real examples, of how digital games development can better exploit an upcoming tendency: software industrialization. By implementing the factory with extensions and customizations of the Visual Studio .NET integrated development environment, different aspects were encompassed by such a study, being the addition of a new visual
designer to the IDE the most appealing subject.

Works related to SharpLudus correspond to currently used game development technologies: multimedia APIs (such as DirectX [20] and OpenGL [30]), visual game creation tools (such as RPG Maker [23]) and game engines (such as OGRE [31] and Crystal Space [32]). The SharpLudus game software factory, however, does not discard the use of such technologies and tools. On the contrary, it is built on their strengths to provide a higher abstraction level to game designers and developers.

One interesting future work is the creation, following the proposed approach, of other factories targeted at other game genres, such as racing games or first-person shooters. Extending the SharpLudus software factory architecture and code generator to support the creation of games targeted at mobile devices, such as cell phones, seems to be quite appealing, since a recognized issue is that porting the same game to different mobile phone platforms is a burdensome and error-prone task. In such a case, once a code generator is implemented for each platform, all platforms would be able to share a single game model (specified with the SLGML visual domain-specific language) and maintenance would be made much simpler. Another future possibility is to investigate how the use of Aspect-Oriented Programming can be used in the product line for crosscutting concerns.

While the results obtained so far empirically shows that the SharpLudus factory is indeed an interesting approach, it is important to notice that deploying a complete software factory is also associated with some costs. Return of investment may arise only after a certain amount of games are produced. Besides that, despite being easy to use, software factories are complex to develop. They will certainly require a mindset evolution of the game development industry.

A final remark is that the presented proposal alone will not ensure the success of a game development. In fact, no technology is substitute for creativity and a good game design. Game industrialization, languages, frameworks and tools are means, not goals, targeted at the final purpose of making people to have entertainment, fun and enjoyment. Players, not the game or its constituent technologies, should be the final focus of every new game development endeavor.
### TABLE 1: ADDITIONAL PRODUCT LINE INFORMATION

<table>
<thead>
<tr>
<th>Game Feature</th>
<th>SharpLudus Product Line Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dimensionality</td>
<td>Two-dimensional (2D). World rooms are viewed from above.</td>
</tr>
<tr>
<td>User Interface</td>
<td>Information display screens containing textual and/or graphical elements are supported. HUDs (heads-up display) can also be configured and displayed.</td>
</tr>
<tr>
<td>Game Flow</td>
<td>Each game should have, at least, a main character, an introduction screen, one room and a game over screen (this last one is reached when the number of lives of the main character becomes zero).</td>
</tr>
<tr>
<td>Sound/Music</td>
<td>Games will be able to reproduce sound effects (wav files) as event reactions. Background music (mp3 files) can be associated to game rooms or information display screens.</td>
</tr>
<tr>
<td>Input handling</td>
<td>Keyboard only.</td>
</tr>
<tr>
<td>Multiplayer</td>
<td>Online multiplayer is not supported by the factory. Event triggers and reactions can be combined, however, to allow two-player mode in a single computer.</td>
</tr>
<tr>
<td>Networking</td>
<td>High scores can be uploaded to and retrieved from a web server.</td>
</tr>
<tr>
<td>Artificial Intelligence</td>
<td>Enemies can be set to chase the player within a room. More elaborated behaviors can be created visually by combining pre-defined event triggers and event reactions, or programmatically by developers.</td>
</tr>
<tr>
<td>End-user editors</td>
<td>Not supported by the factory. Once created, a game cannot be customized by its players.</td>
</tr>
<tr>
<td>Target Platform</td>
<td>PCs running Microsoft Windows 2000 or higher.</td>
</tr>
<tr>
<td>Feature</td>
<td>Graphical Representation</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------------</td>
</tr>
<tr>
<td><strong>InfoDisplay</strong>: An information display screen is represented by a picture (shown in the right), and contains a textual decorator on its outer top, describing its name.</td>
<td></td>
</tr>
<tr>
<td><strong>Intro purpose decorator</strong>: This image decorator is applied to an info display, on its inner top, if the info display’s purpose is <em>Intro</em>.</td>
<td></td>
</tr>
<tr>
<td><strong>Game Over purpose decorator</strong>: This image decorator is applied to an info display, on its inner top, if the info display’s purpose is <em>GameOver</em>.</td>
<td></td>
</tr>
<tr>
<td><strong>Room</strong>: A game room is represented by a picture (shown in the right) and contains a textual decorator on its outer top, describing its name.</td>
<td></td>
</tr>
<tr>
<td><strong>Transition</strong>: State transitions are visually represented as black arrows.</td>
<td></td>
</tr>
</tbody>
</table>
Figures

Fig. 1. SharpLudus top-level network topology

Fig. 2. SharpLudus low-level game architecture (1): the Game class and its relations
Fig. 3. SharpLudus low-level game architecture (3): the `GameState` class and its relations
Fig. 4. SharpLudus low-level game architecture (3): the Entity class and its relations

Fig. 5. Top-level SLGML concepts
Fig. 6. Toolbox support for the game designer

Fig. 7. Properties window support for the game designer

Fig. 8. SLGML Explorer

Fig. 9. Semantic errors displayed in the Visual Studio .NET Error List
Fig. 10. Implementing a semantic validator in the Transition concept class

```java
private void ValidateExitConditions(ValidationContext context) {
    if (this.ExitConditions.Count == 0) {
        context.LogError('"A game state transition must have at least one exit condition specified",
         "NoExitCondition",
         this);
    }
}
```

Fig. 11. Complete SLGML modeling experience
Fig. 12. Class diagram of a generated code example

```csharp
public static class Sprites {
    <#
    foreach($sprite sprite in this.SharpLudusGame.GameSprites) { 
        string spriteName = sprite.Name.Trim().Replace(" ", "");
    <#>
    public static Sprite <#spriteName#> { 
        get { 
            Sprite result = new Sprite();
            result.Name = "<#spriteName#>";
            result.Loop = '<#sprite.Loop.ToString().ToLower()#>';
    <#>
            foreach(Frame frame in sprite.Frames) { 
    <#>                Picture <#frame.Picture.Name#> = new Picture(
                    "<#frame.Picture.FilePath#>",
                    Color.<#frame.Picture.TransparencyKey#>);
                    Game.Add(<#frame.Picture.Name#>);
                    Frame <#frame.Name#> = new Frame(
                        <#frame.Picture.Name#>,
                        <#frame.Delay#>);
                    result.Add(<#frame.Name#>);
            <# }>
    <#>                result.Play();
                return result;
        }
    <# }>
    <# }>
    <# }
```
Fig. 14. Ultimate Berzerk screenshot

Fig. 15. Sprite Designer

Fig. 16. Room Designer
Fig. 17. Event Designer

```csharp
namespace MyGameNamespace {
    public static class CustomTriggers {

        static DateTime shootingTime = DateTime.Now;

        public static bool KnightShotTrigger() {
            bool result = false;
            TimeSpan threeSeconds = new TimeSpan(30000000);

            if (Game.CurrentGameState == States.ExitRoom
                && DateTime.Now - shootingTime > threeSeconds) {
                shootingTime = DateTime.Now;
                result = true;
            }

            return result;
        }
    }
}
```

Fig. 18. Implementing a custom trigger

```csharp
if (Game.CurrentGameState == States.ExitRoom
    && DateTime.Now - shootingTime > threeSeconds) {
    shootingTime = DateTime.Now;
    result = true;
}
```

Fig. 19. IDE support (IntelliSense) and strongly-typed code
```csharp
namespace MyGameNamespace {
    public partial class diamondGuardian : NPC {
        public override void Update() {
            if (this.Velocity.Y == 0) {
                this.Velocity = new Vector2(0, 1);
            }
            if (this.Position.Y < 180) {
                this.Velocity = new Vector2(0, 1);
            } else if (this.Position.Y > 300) {
                this.Velocity = new Vector2(0, -1);
            }
        }
    }
}
```

Fig. 20. Custom entity behavior with partial classes

![Stellar Quest](image)

Fig. 21. Stellar Quest

![Tank Brigade](image)

Fig. 22. Tank Brigade
Fig. 23. Specifying flow control visually in Game Maker [19]

```c
// This script adapts the direction of the monster
// when it comes at a possible crossing
{
    if (haspeed == 0)
    {
        if (random(1) < 1 && place_free(x-4,y))
        {
            haspeed = -4; vspeed = 0;
        }
        if (random(1) < 1 && place_free(x+4,y))
        {
            haspeed = 4; vspeed = 0;
        }
    }
    else
    {
        if (random(1) < 1 && place_free(x,y-4))
        {
            haspeed = 0; vspeed = -4;
        }
        if (random(1) < 1 && place_free(x,y+4))
        {
            haspeed = 0; vspeed = 4;
        }
    }
    switch (ce)
```

Fig. 24. Script language embedded in visual game creation tool.

Fig. 25. Blobbit Dash game screenshot
REFERENCES


[10] Folha de São Paulo Newspaper, Collection packs bring arcade games back to videogames (in Portuguese), Informatics, October 12, 2005;


[19] DigiPen.edu, MSDN Webcast Archive - Video Game Development: Learn to Write C# the Fun Way, http://www.digipen.edu/webcast (last visited in June 13, 2005);


