A UML Profile for GUI Layout

Master's Thesis of Kai Blankenhorn

May 23, 2004

A UML Profile for GUI Layout

Master's Thesis
University of Applied Sciences Furtwangen
Department of Digital Media

Kai Blankenhorn

FACHHOCHSCHULE FURTWANGEN
HOCHSCHULE FÜR TECHNIK UND WIRTSCHAFT
UNIVERSITY OF APPLIED SCIENCES

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Master-Thesis selbständig und ohne unzulässige fremde Hilfe angefertigt habe.
Alle verwendeten (Online-)Quellen und Hilfsmittel sind angegeben.

Furtwangen, den 23.05.2004

I Abstract

The Unified Modeling Language (UML) is a visual language for modeling complex applications. Although most of today's applications have some sort of graphical user interface (GUI), the UML does not specify a diagram capable of creating a model of a GUI's visual appearance. However, coupling GUI design to software design more tightly is desirable to create a complete model of an application prior to implementation, thus to avoid as much errors as possible.

Hence, we have developed the UML 2.0 Profile for GUI layout to support modeling of GUIs in UML-based software development processes. In contrast to earlier approaches, our profile provides an easily comprehensible abstract representation of an actual screen based on designers' sketches. A GUI Layout Diagram consists of a Screen, which contains multiple ScreenAreas. Each may be decorated with one or more Stereotypes, representing performed functionalities like text, image or link. By nesting and arranging properly stereotyped ScreenAreas within each other, the developer is able to create an abstract version of a user interface. The Navigational Diagram provides UML-based support for common design artifacts like storyboards and sitemaps. The diagrams created can be linked to Use Case modeling using existing UML mechanisms to specify requirements and context of a particular screen. Additionally, GUI Layout Diagrams may be used with Activities to model workflows and interaction.

By aligning the architecture of our profile with UML's extension mechanisms and by using the inherent layout information added to UML 2.0 by Diagram Interchange, we create a standards-based solution that should be easy to adopt for tool vendors.

II Contents

1 Introduction ... 1

2 Basics ... 3
- 2.1 UML ... 3
- 2.2 Graphical User Interface Design ... 5

3 Analysis ... 9
- 3.1 Software Design ... 10
  - 3.1.1 Software Design Processes ... 10
  - 3.1.2 Domains ... 11
  - 3.1.3 Artifacts ... 13
  - 3.1.4 Tool Support ... 14
- 3.2 GUI Design ... 15
  - 3.2.1 GUI Design Process ... 15
  - 3.2.2 Domains ... 16
  - 3.2.3 Artifacts ... 17
  - 3.2.4 Tool Support ... 23
  - 3.2.5 Low- vs. High-Fidelity Prototypes ... 23
  - 3.2.6 Observations ... 26
- 3.3 Conclusion: Integrate Software and GUI Design ... 26
  - 3.3.1 Why is it needed? ... 26
  - 3.3.2 What will it do? ... 27
- 3.4 Approaches to Integration ... 29
  - 3.4.1 Using basic UML ... 30
  - 3.4.2 Other Approaches ... 35
  - 3.4.3 Diagram Interchange ... 37

4 Requirements and Goals ... 39
- 4.1 Requirements ... 40
  - 4.1.1 User Groups ... 40
  - 4.1.2 Workflows ... 42
  - 4.1.3 Summary of Requirements ... 42
- 4.2 Goals ... 43
- 4.3 Evaluation of Existing Approaches ... 43

5 Design ... 45
5.1 Classification of Stereotypes ... 46
5.2 General Design Principles ... 47
5.3 Architectural Overview ... 49
5.3.1 Connection to UML ... 49
5.3.2 Overview of Classes ... 49
5.4 Package GUILayout ... 50
5.4.1 ScreenArea ... 53
5.4.2 ContainerScreenArea ... 57
5.4.3 Screen ... 59
5.4.4 FunctionalScreenArea ... 60
5.4.5 UIFunctionality ... 61
5.4.6 StaticUIFunctionality ... 62
5.4.7 ActivatableUIFunctionality ... 63
5.4.8 Form ... 63
5.4.9 Link ... 64
5.4.10 Navigation ... 66
5.4.11 Workspace ... 67
5.4.12 Heading ... 68
5.4.13 Image ... 69
5.4.14 Logo ... 70
5.4.15 Text ... 71
5.5 Package GUILayout::References ... 72
5.5.1 Reference ... 72
5.5.2 LinkReference ... 74
5.5.3 ScreenFlow ... 75
5.5.4 Screen ... 76
5.5.5 ActivatableUIFunctionality ... 76
5.5.6 Form ... 77
5.5.7 Link ... 77
5.5.8 Navigation ... 77
5.5.9 Workspace ... 77
5.6 GUI Layout Diagram ... 78
5.7 Navigational Diagram ... 78
5.8 Links to Existing Diagrams ... 78
5.8.1 Use Case Diagrams ... 79
5.8.2 Activity Diagrams ... 81
5.9 Examples ... 82

6 Prototype 89
6.1 Requirements and Installation 90
6.2 Usage Instructions 90
6.3 Comments 92
6.4 Third Party Products Used 93

7 Testing 95
7.1 Expressive Power of the Extension 95
7.1.1 XHTML 1.0 Elements 95
7.1.2 CSS 2.0 Attributes 96
7.1.3 Java Swing Classes 98
7.2 Designers' Feedback 99

8 Results 101
8.1 Conclusion 101
8.2 Outlook 102

A UML 2.0 DI Metamodel 105

B References 109

III List of Tables and Figures

Figure 1. UML Metamodel Architecture ... 5

Figure 2. The Input and Result of Prototyping the User Interface ... 11

Figure 3. The Workflow for Capturing Requirements as Use Cases, Including the Participating Workers and Their Activities ... 12

Figure 4. Domains Involved in the Design of User Interfaces ... 16

Figure 5. A Sitemap ... 18

Figure 6. A Storyboard ... 19

Figure 7. A Hand-Drawn Schematic ... 20

Figure 8. A Computer-Drawn Wireframe ... 20

Figure 9. A Mockup of a Web Page ... 21

Figure 10. A Slide from a Presentation ... 22

Figure 11. Part of a Paper Prototype ... 24

Figure 12. A High-Fidelity Prototype ... 24

Figure 13. Taxonomy of UML Structure Diagrams ... 30

Figure 14. A Screen Modeled as a Plain Class Diagram ... 32

Figure 15. One Screen Modeled as Two Composite Structure Diagrams – Layout Information Bears No Semantic Meaning ... 32

Figure 16. GUI Layout Modeled Using the Notation Proposed by BAUMEISTER, KOCH & HENNICKER ... 34

Figure 17. UI Element Cluster by PHILLIPS & KEMP ... 35

Figure 18. Screen Modeled with OMMMA-L ... 36

Figure 19. Package Diagram of the UML Profile for GUI Layout 50

Figure 20. Architecture of the package GUILayout 51

Figure 21. Profile Metamodel Layers 52

Figure 22. Example of ScreenArea Notation and ScreenArea Inheritance 56

Figure 23. Nested ContainerScreenAreas as Concrete Syntax and Instance Specification 58

Figure 24. Notation of a Screen 60

Figure 25. Two FunctionalScreenAreas Owning Two UIFunctionalities Each 61

Figure 26. Stereotype Icon of Form and its Sketched Origin 64

Figure 27. Stereotype Icon of Link and its Sketched Origin 65

Figure 28. Stereotype Icon of Navigation 66

Figure 29. Stereotype Icon of Workspace 67

Figure 30. Stereotype Icon of Heading and its Sketched Origin 68

Figure 31. Stereotype Icon of Image and its Sketched Origin 69

Figure 32. Stereotype Icon of Logo and its Sketched Origin 70

Figure 33. Stereotype Icon of Text and its Sketched Origin 71

Figure 34. Architecture of the package GUILayout 72

Figure 35. Use Case Diagram Enriched with Isolated ScreenAreas 80

Figure 36. GUI Layout Diagram Enriched with Isolated UseCases 80

Figure 37. Two-Column View of UseCases and ScreenAreas 80

Figure 38. A GUI Layout Diagram and the Corresponding Web Page 82

Figure 39. Example of a GUI Layout Diagram with Associated UseCases 83

Figure 40. Example of a Screen with Abstract ScreenAreas Showing the Base Layout 83

Figure 41. Example of Precisely Sized ScreenAreas 84

Figure 42. Example of ScreenArea Inheritance 84

Figure 43. Example of an Activity with an ActivityPartition for Presentation........ 85

Figure 44. Example of a Navigational Diagram, LinkReferences and ScreenFlows 86

Figure 45. Example of a Storyboard Based on a Navigational Diagram and LinkReferences 87

Figure 46. Example of a Sitemap Based on a Navigational Diagram and ScreenFlows 87

Figure 47. Example of a GUI Layout Diagram of an Application Window........ 88

Figure 48. Screenshot of the Prototype with Annotations........ 89

Figure 49. Building a Screen Layout within Five Minutes 91

IV Abbreviations and Definitions

Artifact

“A piece of information that is used or produced by a software development process, such as an external document or a work product. An artifact can be a model, description, or software.”[1]

CASE

Computer Aided Software Engineering

DTD

Document Type Definition; a definition of how an XML file may be structured in order to be valid.

GUI

Graphical User Interface

GUI layout

Sizing and positioning GUI elements to form a functional, visually attractive screen.

High-fidelity

High level of detail, visually elaborate, looking like real

IDE

Integrated Development Environment; a software development tool that includes at least an editor, a compiler and a debugger.[2]

Low-fidelity

Abstract, low level of detail, visually imperfect

Mockup

A non-interactive, high-fidelity representation of a GUI

[1] Rumbaugh et al. 1998, 152
[2] Wikipedia 2004, “Integrated development environment”

MOF

Meta Object Facility[3]; the metalanguage that is used to define UML and other modeling languages (the Common Warehouse Model, CWM, for example).

OCL

Object Constraint Language; used to specify constraints and operations in UML models

OMG

Object Management Group, http://www.omg.org

OOA

"Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain."[4]

OOD

"Object-oriented design is a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design."[5]

OOP

"Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represent an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships."[6]

Reification

"[T]he act of making a data model for a previously abstract concept"[7]; for instance, in the UML the abstract concept of "usage" is reified into a usage dependency and displayed as an arrow in the diagram.

Round-trip engineering

A functionality of software development tools that provides generation of models from source code and generation of source code from models; this way, existing source code can be converted into a model, be subjected to software engineering methods and then be converted back.

---

[3] OMG 2003d
[4] Booch 1994, 39
[5] Booch 1994, 39
[6] Booch 1994, 38
[7] Wikipedia 2004, "reification"

RUP

Rational Unified Process[8]; a heavyweight, use case driven software development process that can be thought of as a manual on how to use UML in a project.

Schematic

Hand-drawn, low-fidelity version of a GUI that is used in the early stages of GUI design.

UI

User Interface

UML

Unified Modeling Language[9]

Wireframe

Computer-drawn, low-fidelity version of a GUI that is used in the early stages of GUI design.

XHTML

Extensible Hypertext Markup Language; successor of HTML that offers the same features but has a slightly different syntax so it is well formed XML.

XSLT

Extensible Stylesheet Language Transformations[10,11]; used to transform XML files from one XML dialect into another.

[8] Kruchten 1999
[9] OMG 2003b
[10] W3C 1999
[11] Kay 2000

Chapter 1

Introduction

Since its adoption by the OMG in November 1997, the Unified Modeling Language (UML) has emerged as the dominating language for modeling software systems. Via several minor revisions, the UML has evolved from the originally adopted 1.1 to its latest major revision, version 2.0. With its spreading use within the software development community, more and more issues with software development practice had been risen[12].

Numerous tendencies have formed the UML to what it is today, improving on many fields to model all important aspects of software systems. These aspects are reflected in the diagrams UML provides for modeling. Judging from these, aspects covered are classes, use cases, components, deployment, internal states of the software, activities, timing and sequences, and collaboration. Obviously, presentation is not one of them. Does that mean presentation is not an important part of software?

Of course it is[13]. User Interfaces are important to the users of a software system, as they are *per definitionem* the only part of the system that is visible to the users. They have a big impact on overall cost and productivity[14,15]. They are important in terms of realization, as they make up about 50% of application code[16,17]. The UML simply fails to take this into account by not providing the methods for modeling user interfaces. How can a model be complete if it ignores an aspect as important as user interfaces?

User interfaces consist of their visual representation (the layout) and the interaction they permit. UML does provide diagrams for modeling interaction, and they can be applied to the in-

---

[12] OMG 2003, 22
[13] Galitz 2002, 3-14
[14] Myers 1993
[15] Galitz 2002, 5-6
[16] Galitz 2002, 3
[17] van der Veer & van Vliet 2001

teraction within user interfaces nicely. But UML does not provide a diagram for modeling the layout of user interfaces, which is especially important for graphical user interfaces (GUIs), because their graphical nature allows for more diversified designs. UML provides extensibility mechanisms that can be used to extend UML to new domains, and we will analyze if they are sufficient for this task.

Extending a formal method from the field of computer science to the creative work of screen designers can be hazardous. Graphics artists could feel too restricted by the method and be unwilling to adopt it. Therefore, before integrating user interface design into a software development process we will account for the designer's needs in this.

The remainder of this thesis is structured as follows: First, we will introduce some basic facts for readers who are not familiar with one or both of our fields, GUI design and modeling. The concept of modeling, the diagrams of UML 2.0 as well as UML's metamodel are explained in brief, followed by a categorization of GUI design and its goals.

After the basics, we will perform a detailed analysis of these two domains in chapter 3, see how they are structured into subdomains and which artifacts are produced for what purpose. Software tools could be an interface between the two fields; therefore, we examine how work in each of the two is supported by tools. In the analysis of GUIs, we also attend to the issue of prototypes, as they are often used to test GUI design decisions. From the facts learned in this analysis, we will draw our conclusions on integrating the two fields into one solution and have a look at how other researchers have addressed the problem and which technologies and methods come in handy. Based on these findings, we develop requirements to satisfy both user groups and formulate our goals according to these requirements (chapter 4).

In chapter 5 (Design) we show how we bring together these goals into a sound solution. We explain the architecture of our extension and give exact definitions of each of its elements, including simple examples and usage guidelines. These definitions are followed by an extensive example of how our extension can be used in combination with existing UML diagrams.

The prototype that illustrates our ideas is presented in chapter 6. This section includes requirements, installation instructions, several screenshots and the limits of the prototype. To make sure our extension can be used to fulfill its purpose, we have performed some tests. First, we have checked the expressive power by assigning the language elements of two typical programming languages used to build GUIs to a model element of our extension. Additionally, we have presented our results to designers and state their responses in this chapter (7. Testing). In the last chapter, we summarize our results and give an outlook of future work in this area.

Chapter 2

Basics

2.1 UML

"The Unified Modeling Language is a visual language for specifying, constructing and documenting the artifacts of systems. It is a general-purpose modeling language that can be used with all major object and component methods, and that can be applied to all application domains [...] and implementation platforms"

UML 2.0 Infrastructure[18]

"Modeling refers to the process of generating a model as an abstract representation of some real world entity."[19] A modeling language defines the grammar of the model, i.e. how the model is defined.

The Unified Modeling Language allows developers to create a model of a software system similar to blueprints of a house. To keep an overview of the whole model, the UML provides several diagrams. Each diagram shows only a few aspects of the system and omits all that are irrelevant to this specific perspective. Together, these diagrams form a complete picture of the model with all its aspects. In UML 2.0, 13 diagrams are defined. They reach from a high level of abstraction – how users perceive and use the system – to a low level of abstraction – detailed timings of critical processes and fine-grained object structure. Table 1 gives an overview of the UML diagrams and their use.

[18] OMG 2003a
[19] Wikipedia 2004, entry for "model"

Diagram	**Description**
class diagram	Fine-grained internal object structure of the system, both real-world and abstract or implementation aspects
object diagram	Runtime structure of class instances
composite structure diagram	Internal structure and collaborations
component diagram	Dependencies among system components
deployment diagram	Physical arrangement of computer systems and the components being executed on them
package diagram	Logical groupings of classes and dependencies between groups
activity diagram	Computations, workflows, object flow and control flow
use case diagram	Behavior of the system to an outside user, describe what can be done with the system
state machine diagram	Various possible states of an object and their transitions
sequence diagram	Timing of messages between objects during an interaction
interaction overview diagram	Overview of the flow of control
timing diagram	Time-dynamic behavior and state changes
collaboration diagram	Objects and links within an interaction

Table 1. UML diagrams overview[20]

Three diagrams are of special importance for this thesis. The highest level is the use case diagram, which "captures the behavior of a system [...] as it appears to an outside user"[21]. It is designed to be understood by users not proficient in UML. In the diagram, a subsystem is depicted along with one or more persons called actors who use it. Each actor is associated with several use cases, each of which describes what the actor wants to do with the system, for instance "load a file", "type text" etc. On the lower levels of UML, there are the class diagrams and activities. Class diagrams show the structure of things, which are called classes in UML, within a software system. For instance, a document may have an author, a title, some keywords etc. In a class diagram, these properties are modeled as attributes, and operations that can be performed on the class are defined, e.g. saving a document. Activities describe what happens when such an operation is invoked and provide a fine-grained way of modeling algorithms.

[20] OMG 2003a
[21] Rumbaugh et al. 1998, 63

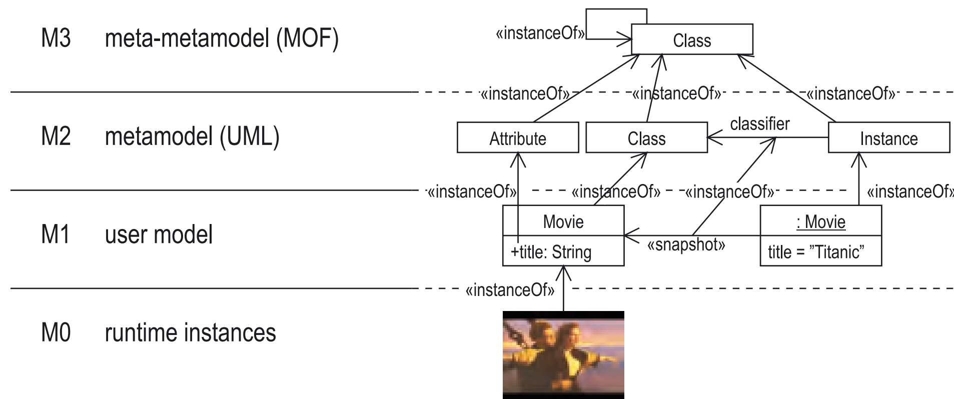


Figure 1. UML Metamodel Architecture[22]

UML is defined as a model itself, using a metamodeling language called Meta Object Facility (MOF)[23]. The prefix 'meta' is of Greek descent and means 'higher, beyond'.[24] A metamodel is a model on a higher level of abstraction, i.e. a model of a model; it is used to specify models. This implies that every model is an instance of a metamodel. Consequently, a meta-metamodel is used to specify metamodels.

The architecture of UML is divided into four layers, labeled M0 to M3 in order of their level of abstraction[25] (Figure 1 – for reasons of comprehension, not all instance relationships are modeled). Each layer, beginning with M0, is specified by its respective metamodel. Therefore, the topmost and most abstract layer is M3, which in case of the UML metamodel consists of MOF. MOF is reflective, i.e. it can be used to define itself. Therefore, there is no need for an M4 layer.

2.2 Graphical User Interface Design

The user interface is the part of software that is used to convey information to the user or take instructions from the user. Hence, it consists of two parts: An input language and an output language.[26] The user utilizes the input language to communicate with the computer by manipulating interaction devices like a computer mouse, a keyboard or a touch screen. The computer system uses the output language to communicate its state to the user. In a graphical user interface (GUI), the output language is a graphical language that is presented on a screen. Its base elements are pixels, which build more complex elements like lines, boxes, letters, etc.

[22] OMG 2003a, 31
[23] OMG 2003d
[24] Oxford 1995
[25] OMG 2003a, 28-31
[26] Foley et al. 1990, 394

User interfaces design can be separated into four components.[27] In the *conceptual design*, the principal application concepts like objects, relationships and metaphors are developed. In the *functional* or *semantic design*, all operations on all objects of the user interface are defined with their input and output information. *Sequencing* or *syntactic design* defines the ordering of inputs and outputs to the system. An example of input ordering is drag and drop: Click an object, drag the mouse, and release the mouse button. Syntactic design for output includes the layout of a display and its dynamic changes. *Lexical design*, also called hardware binding design, determines how syntactic elements are represented by actual hardware primitives.

GUI layout (also called screen design) is the action of methodically creating the visual presentation[28] of a user interface by spatially arranging elements like lines, boxes or text on a computer screen. Therefore, it is a part of syntactic output design. Its goals are:[29]

- *Increase speed of learning*. How long does it take a user to reach a certain level of proficiency of the system? This is crucial for situations where a system is used infrequently by its users.
- *Increase speed of use*. How long does it take an experienced user to perform a task with the system? This is crucial for situations where the system is used frequently or for a large amount of time.
- *Reduce error rate*. How many errors does a user produce during one interaction? Error rate affects the speed of learning and the speed of use of a system, and therefore is important for most systems.
- *Encourage rapid recall*. How long does it take infrequent users to remember how to use the system?
- *Increase attractiveness*. How does the user interface appeal to the shareholders of the project? Attractiveness can often be counterproductive.[30]

While web design is a subset of GUI design, it is different from it in several aspects.[31] GUI systems usually have a fixed set of widgets to draw from. Web pages have almost no presentational restrictions besides technical issues. As a result, many GUI applications share the same look and feel, which makes getting familiar with them much easier, while web pages tend to try to estab-

---

[27] Foley et al. 1990, 394
[28] Galitz 2002, 23
[29] Foley et al. 1990, 391
[30] Foley et al. 1990, 392
[31] Galitz 2002, 29-36

lish their own look and feel. Because of the lack of restrictions, web pages can be more artistic and individual. Hence, graphics design is more important for web pages than for GUI systems.

Doing innovative and effective page layout for a large website requires graphics designer expertise.[32] For websites, the GUI design is a critical factor for the success of the site.[33,34] Section 3.2 deals with the work of screen designers in detail.

[32] Shneiderman 1997, 577
[33] Shneiderman 1997, 580
[34] Lohse & Spiller 1998

Chapter 3

Analysis

After the basics have been made clear in the previous chapter, it is important to gain a detailed and complete image of the two fields of work, software design and GUI design. Therefore, in this chapter, we will analyze how they are structured into subdomains and how these fit together. We will compile a list of artifacts that are created in each field, and check if there are similarities that can be built upon for a unifying solution. While Analyzing GUI design, we will take special care of prototypes, as they are artifacts that belong to GUI design and software design simultaneously. Therefore, we will perform an analysis of the pros and cons of different ways of prototyping. All this is important to get an idea of the real life work of software and graphics designers, in order to maximize acceptance of a new method for GUI layout. The facts learned about their work are then laid down in the requirements chapter that follows.

Extending methods of formal modeling to artistic designer practice is a problem of integrating two worlds. Today, user interfaces are widely created by graphics designers or by dedicated interface designers, and not by software designers. In addition, software design tends to treat user interfaces as just another part of the software, neglecting all particularities. We will find out which approaches exist to change this, how they are doing what they are doing and whether this is of practical use to software and GUI designers.

3.1 Software Design

3.1.1 Software Design Processes

Software engineering has developed a multitude of processes to apply to software design. Software design processes are used to manage how a piece of software is being created. To put it simply, they try to answer the question "Who is doing what at which point of time in which way?"[35] Today, there are so many different software design processes that it is beyond the scope of this section to cover them all. Instead, we will focus on two widely used ones.

Generally, there are two classes of design processes: Heavyweight processes and lightweight/agile processes. Heavyweight processes offer a lot of features and control, at the cost of high overhead. They tend to make more use of modeling than their opposite, lightweight or agile processes. Agile processes focus on code instead of a rigid sequence of actions and produce less overhead, but constrain the set of features and offer much less control than heavyweight processes.

The Rational Unified Process (RUP)[36] is an example of an iterative heavyweight software development process. In RUP, much effort is made for developing and maintaining detailed models of every aspect of the system. RUP is tailored to the UML and uses it to specify its models.[37] It includes several activities, artifacts and roles related to GUI design. User interface design is part of the phase of capturing requirements as use cases at the beginning of the project (see Figure 2).

Extreme Programming (XP) is the most successful agile process. In XP, only the most important aspects are modeled, if any at all.[38] Working directly with code is preferred.[39] XP runs short (1 to 3 weeks) iteration cycles, during which new functionality and user interfaces are developed and tested.

---

[35] KRUCHTEN 1999, 35
[36] KRUCHTEN 1999
[37] KRUCHTEN 1999, 28
[38] FOWLER 2000
[39] BECK 1999, 112

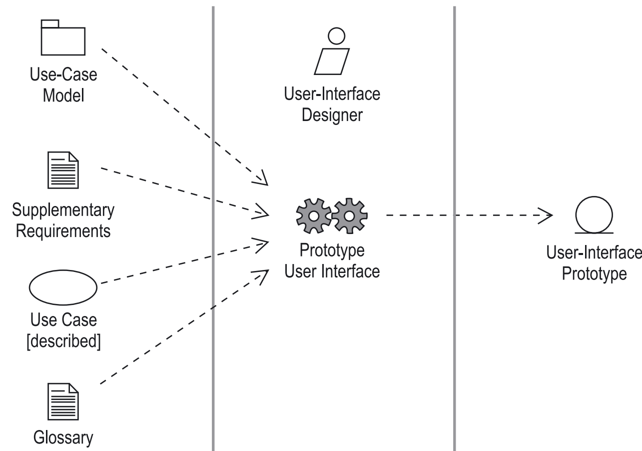


Figure 2. The Input and Result of Prototyping the User Interface[40]

Development processes differ greatly in their use of formal modeling. Most interesting for this thesis are those processes that:

- Do a lot of modeling
- Use the UML for specifying their models.

We will call these processes UML-based processes from now on. RUP is a UML-based process, and we will keep using it as an example.

3.1.2 Domains

The domain of software design has evolved into several sub domains. Because most of them are of little to no importance to user interface design, only relevant ones will be mentioned here. As every process creates its own naming convention, we will use one of these conventions, the one from RUP. RUP describes domains in the form of workers, who are the ones performing the tasks of a certain domain. Figure 3 shows how the workers' activities fit together.

System Analyst

The system analyst is responsible for the domain of requirements capture. He accomplishes this by talking to the customer and then identifying actors and use cases.[41] He only creates some (often the most important) use cases and is assisted by several use case specifiers for the rest. Often this role is filled by the account manager or project manager.

[40] Jacobson et al. 1998, 161
[41] Jacobson et al. 1998, 140, 144-153

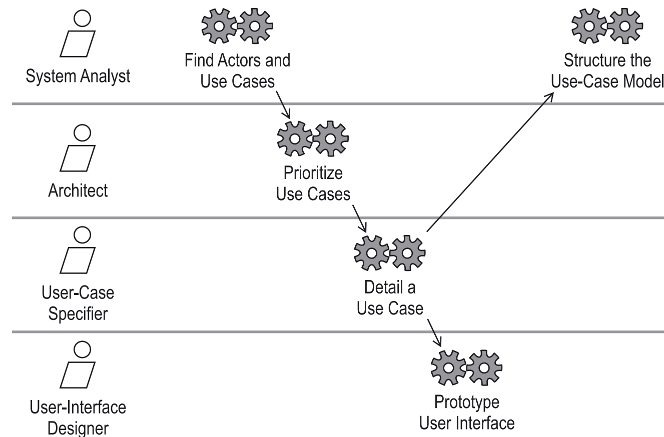


Figure 3. The Workflow for Capturing Requirements as Use Cases, Including the Participating Workers and Their Activities[42]

Architect

The architect is responsible for describing the architecture of the system during requirements capture and for prioritizing the use cases specified by the use case specifiers. Use cases that are required on multiple occasions or ones that are crucial for the success of the project are assigned a higher priority than unimportant, rarely used ones. The output the architect creates is used later on to determine which elements of the architecture are implemented first.[43]

Use Case Specifier

A use case specifier is responsible for detailing one or more use cases into their flow of events, including how it starts and ends, and how actors can interact. The use case specifiers create detailed use cases from the outlined use cases specified by the system analyst and include other information like the supplementary requirements. Use case specifiers have to work with the real users of the use cases frequently in order to tune the detailed use case to their situation.[44]

User Interface Designer

The user interface designer creates the layout of the user interface, but he does not implement it. However, he may create prototypes for the user interface of some use cases.[45]

The work of user interfaces designers is described in detail in section 3.2.

[42] Jacobson et al. 1998, 143
[43] Rumbaugh et al. 1998, 142
[44] Rumbaugh et al. 1998, 141
[45] Rumbaugh et al. 1998, 142

3.1.3 Artifacts

While working on a software project, a multitude of artifacts is created. RUP contains description of some common artifacts used during requirements capture and user interface design, which we will reproduce here.

3.1.3.1 Use Case Model

The use case model contains the use cases as identified by the use case specifier. Each use case represents one task or activity that is relevant to an actor.[46] Together, they form the requirements of the software from a user's point of view.[47] Thereby, customer and developer use it for communication, and it serves as a kind of contract between them.[48] The use case model is created by the system analyst and the use-case specifiers by first identifying all actors that use the system. Then, use cases for each actor are defined by interviewing the customer. The use case model is used to create user interface prototypes[49] and to generate tests to verify the system against after the development. The use case model as a whole defines the conceptual design (cf. 2.2) of the user interface. The detailed use cases, which may include operations and attributes[50], define its semantic design.

3.1.3.2 Supplementary Requirements

All nonfunctional requirements that cannot be modeled as a use case but instead affect all of them or none at all are represented as supplementary requirements. Typical supplementary requirements are interface requirements, physical requirements, design constraints and implementation constraints.[51] Table 2 gives examples for each of these types.

Supplementary requirement	Example
Interface requirements	New data will be available at 2 am and must be processed within 1 hour
Physical requirements	Server: Linux 2.6, IA64; Clients: Windows XP Professional, IA32
Design constraints	Use SAP/R3 data format; easy conversion to R4 format
Implementation constraints	Implementation in Java; valid XHTML output
Other requirements	Availability 99,5%; user data stored on a separate system

Table 2. Supplementary requirements examples

---

[46] Rumbaugh et al. 1998, 26
[47] Jacobson et al. 1998, 131-172
[48] Kruchten 1999, 100
[49] Kruchten 1999, 101
[50] Jacobson et al. 1998, 136
[51] Jacobson et al. 1999, 128

3.1.3.3 User Interface Prototype

The user interface prototype is created by the user interface designer (see 3.1.2), based on the use case model as a whole, the supplementary requirements, and the individual described use cases (Figure 2 on page 11). Also included is a glossary with common terms used by the actors. This artifact consists of sketches and interactive prototypes "that specify the look and feel of the user interfaces for the most important actors"[52]. Creating the prototype involves two steps of design: a logical and a physical user interface design.[53] Logical interface design corresponds to syntactic design of section 2.2, and physical user interface design corresponds to lexical design.

3.1.4 Tool Support

UML diagrams are intended to be drawn using a specialized software application. Any vector based graphics application would do fine for drawing diagrams, but for larger models it is important to provide a higher level of organization to help accessing information[54] and enforcing syntactic and semantic rules[55]. Since UML 1.1, numerous specialized applications offering advanced features for modelers have evolved, and existing CASE tools have adopted the UML syntax. Rational Software (and IBM respectively, which has acquired Rational Software in 2002), the company that employs the three amigos BOOCH, JACOBSON and RUMBAUGH, offers extensive support for the UML. Rational Rose™ is a standalone UML modeling software as well as part of the Rational Unified Process, which is not only the process itself, but also a software suite. There are also several open source applications for UML modeling, with ArgoUML[56] being the most widely used.

Most modern Integrated Development Environments (IDEs, e.g. Microsoft Visual Studio, Borland Delphi, the Eclipse project) provide a tool to create GUIs visually without much programming. These so-called GUI wizards can be used to create and modify high fidelity prototypes quickly. With an IDE that includes active support for GUI programming, software developers can do most of their work using a single application.

Many software projects use a version management system like CVS[57], Subversion[58] or Rational ClearCase. These systems are used to continually create backups and log files of all changes made to any files of the project. Every change can be undone individually to restore a previous

---

[52] JACOBSON et al. 1998, 161
[53] JACOBSON et a. 1998, 160-166
[54] RUMBAUGH et al. 1998, 108
[55] JACOBSON et al. 1998, 30
[56] ROBBINS 1999
[57] BERLINER 1990
[58] COLLINS-SUSSMANN et al. 2004

state. Thereby, these systems reduce the risk of making wrong implementation decisions and help a team work on the same set of files at a time without interfering with each other's work.

3.2 GUI Design

3.2.1 GUI Design Process

The usual methodology for GUI design is a top-down, user centric approach[59,60,61], beginning with specifying actors and use cases, and in the end creating sketches and prototypes for every use case or actor[62], or a prototype of the key screens[63]. The artifacts created during this phase often include a guidelines document, which contains detailed explanations of the design principles to be applied when implementing the design.[64]

It is well known that theory and reality often differ. A study among web designers showed that the everyday non-formalized design process comes in four phases:[65]

- During the *discovery* phase, the designers try to get an understanding of the client's and future users' desires and needs. If the system is a remake of an existing one, the latter is reviewed and evaluated. This phase often includes an analysis of the competition in the field.
- The goal of the *design exploration* phase is to quickly produce several design proposals for the client, based on the results of the discovery phase. The client then selects one of the proposals to be further developed. Designers create multiple rough design ideas, disregarding details like color or typography. Navigational design is often developed in this phase. The activities of the design exploration phase correspond to conceptual and semantic (navigational) design components (see 2.2). Syntactic (graphics) design is started.
- In the *design refinement* phase, the selected design is iteratively refined in more detail. Page classes are identified and individually laid out. Details added in this phase include specific images, fonts, texts, and the color palette used.

---

[59] Foley et al. 1990, 429
[60] Shneiderman 1997, 105
[61] Jacobson et al. 1998, 131-172
[62] Jacobson et al. 1998, 142
[63] Shneiderman 1997 104
[64] Shneiderman 1997, 100-102
[65] Newman & Landay 2000

- Impending deadlines usually start the *production phase*. Designers prepare the artifacts that are delivered to the client as the result of the project. These artifacts typically include interactive prototypes, design documents, and specifications.

During this process, the website is represented by various intermediate artifacts like site-maps and mockups (cf. 3.2.3) to facilitate communication within the project and with outsiders like stakeholders.

3.2.2 Domains

Designing a graphical user interface involves several domains of design. There is a lot of overlap between them, and in many cases, a single person might fulfill some of them, and others are not fulfilled at all. Figure 4 shows how the domains are related to each other.

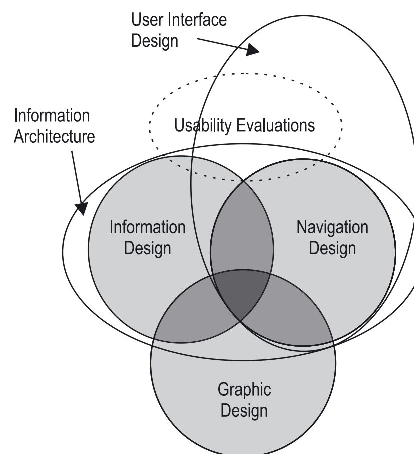


Figure 4. Domains Involved in the Design of User Interfaces[66]

Information design deals with structuring large amounts of information into sensible chunks, e.g. on web pages or individual dialogs: “Information Design addresses the organization and presentation of data: its transformation into valuable, meaningful information. […] Information Design doesn’t ignore aesthetic concerns but it doesn’t focus on them either. […] Information Design does not replace graphic design and other visual disciplines, but is the structure through which these capabilities are expressed.”[67] Information Design helps transforming data into information and finally into knowledge.

[66] Newman & Landay 2000
[67] Shedroff 2000

Navigation design is essentially done for web sites. It deals with creating the paths users can take to reach information or accomplish tasks. Navigation design is critical for hypertext media like websites or electronic encyclopedia, but can rarely be applied to other windowed applications.

Graphic design creates the visual part of the user interface, "using elements such as color, images, typography and layout"[68]. Graphics design is independent from user interface design and can be performed for other media as well, for instance print media. Therefore, we will refer to the graphics design part of user interface design as screen design, and to the layout part of screen design as GUI layout.

Though not part of the UI design team, the project and account manager plays an important role in the design process. He often is the only link of the design team to the customer, and thus must communicate design ideas as effectively as possible. He is also the one to capture user interface requirements.

3.2.3 Artifacts

> *„Some of the earliest prototypes are simple hand-drawn pictures. As they progress, they may take the form of wire frames, typically created with a drawing tool, such as Visio or Adobe Illustrator. Combined with use case specifications, these prototypes are good for storyboards describing specific scenarios of flow through the system's screens. The prototypes are also good for understanding the structure of compartmentalized screens and can give stakeholders an early and tangible view of the system."*
>
> *Jim Conallen*[69]

This quote outlines how the various artifacts that are involved in user interface design are employed during the development process. The following sections will explain the most important ones.

3.2.3.1 Sitemaps

Sitemaps are high-level diagrams of all the pages of a website. They are directed graphs that show the information structure of the website and part of the navigational structure.[70] The knots of the graph are individual pages, some of which may be grouped by additional boxes. The edges of the graph represent navigational paths within the website. Each knot may contain

[68] Newman & Landay 2000
[69] Conallen 2003, 188
[70] Newman & Landay 2000

a text as a label and description (Figure 5). Only the primary navigational paths of the website are actually shown in the sitemap. For example, while assuming that every page contains a link to the start page, these links are not shown as edges in the graph.

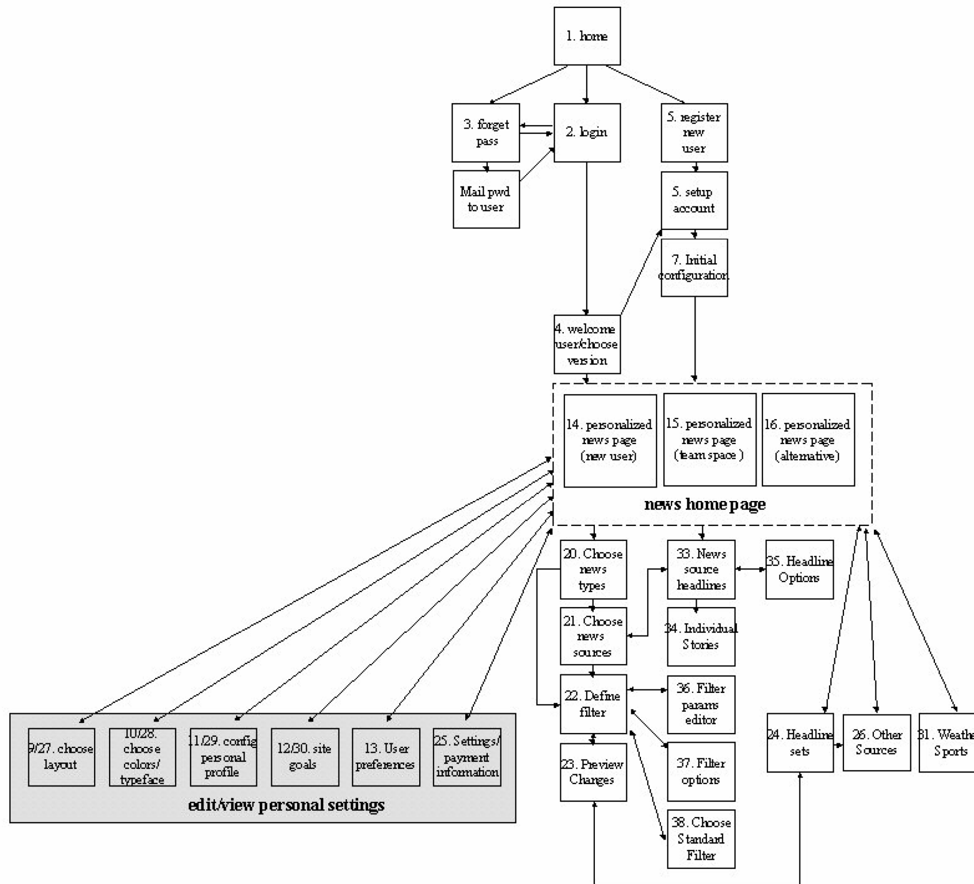


Figure 5. A Sitemap[71]

Sitemaps are used by information designers to evaluate and improve the site structure. They are also used by the whole design team to create a common idea of the whole website.

This notion of sitemaps differs from what website users may know as sitemaps; websites often have a publicly accessible sitemap. These sitemaps are merely a structured list of links to all parts of the website and lack most of the information of the sitemaps described above.

[71] Newman & Landay 2000

3.2.3.2 Storyboards

Storyboards show the individual steps of an interaction sequence.[72,73] The interaction is some task a user wishes to accomplish and involves multiple steps – in terms of UML: storyboards show the individual steps of a use case. A Storyboard consists of multiple simplified screens, which contain only the basic elements and links needed for the interaction. The links used are connected to the next screen in the interaction. Figure 6 shows how a user of a tutorial would interact with the system in order to find information on a specific topic.

Storyboards are used by information designers as well as the whole design team, to obtain an understanding of workflows and navigation.

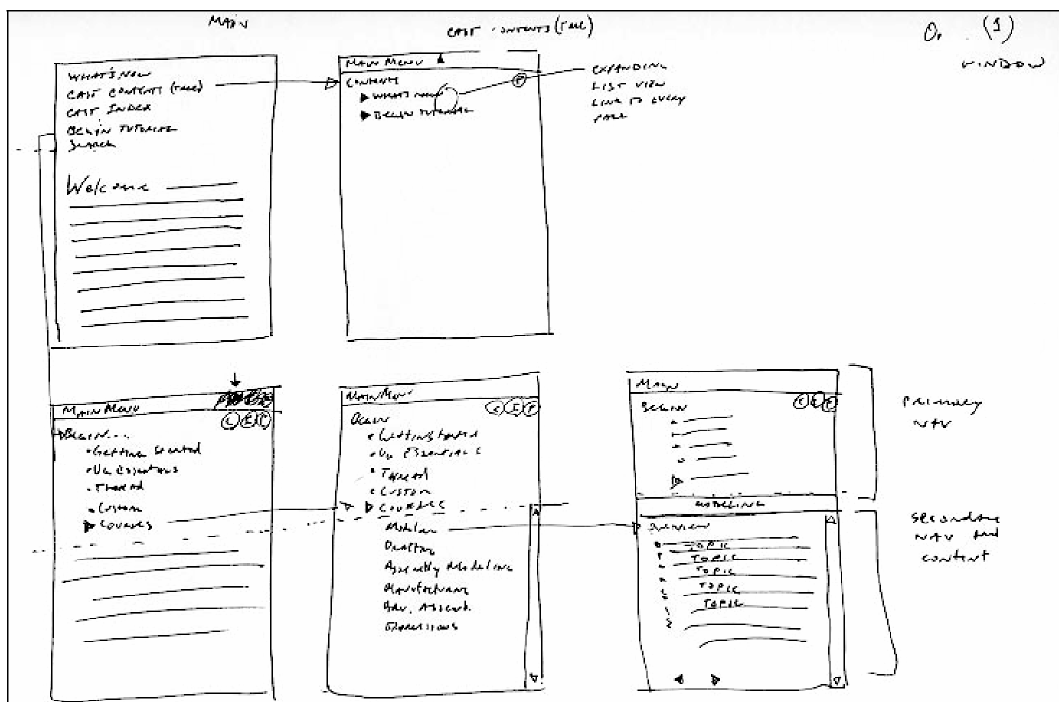


Figure 6. A Storyboard[74]

3.2.3.3 Schematics

Schematics show the actual contents of a screen and its layout.[75] Images and continuous text are represented by symbols or lines. Schematics can be drawn by hand (Figure 7) or using a

---

[72] Newman & Landay 2000
[73] Foley 1990, 430
[74] Newman & Landay 2000
[75] Newman & Landay 2000

vector drawing program like Microsoft Visio or Adobe Illustrator (Figure 8). Computer-drawn schematics are also called *wireframes*.[76]

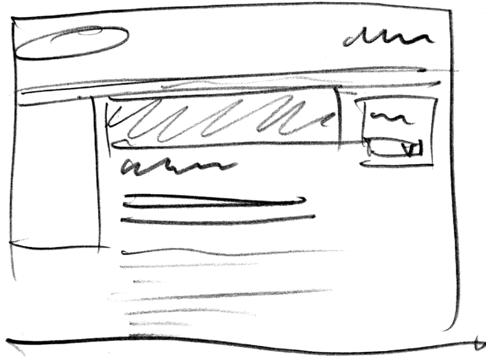


Figure 7. A Hand-Drawn Schematic[77]

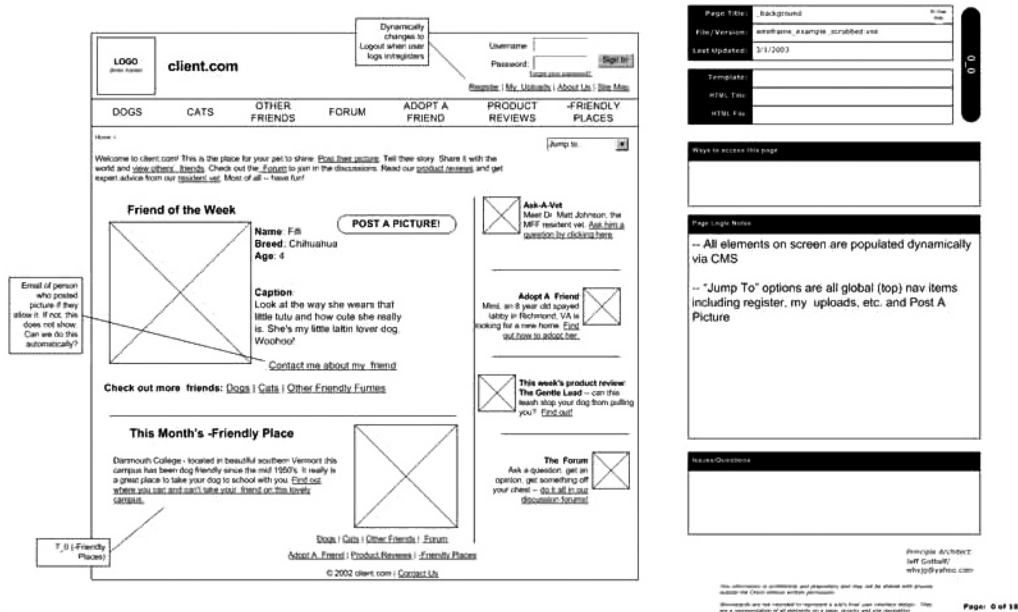


Figure 8. A Computer-Drawn Wireframe[78]

Although schematics represent individual pages, they are rarely put to real use by graphics designers. Our own studies have shown that instead they create some initial sketches that they usually not share with others and start working on the design in a graphics application soon.

[76] Conallen 2003, 188
[77] Used with permission of Virtual Identity AG
[78] Gotthelf (unknown year)

Usually, information designers use schematics to specify which elements are needed on a page and to coordinate their work with the graphics designers.[79] They are also used for communicating with the client, balancing the needs of focusing on basic issues and making a good impression on the client.

"Designers often sketch on paper early in the design process" to keep a high-level view of the site as long as possible.[80] This enables them to explore more design choices without getting lost in details.

Figure 9. A Mockup of a Web Page[81]

3.2.3.4 Mockups

Mockups are high-fidelity representations of screens.[82] They are created by graphics designers using a graphics application like Adobe Photoshop. Unlike schematics, they are to be taken literally and are usually indistinguishable from the final product. They may use dummy text instead of real texts. Mockups are used to create a visual specification for the programmer to adhere to.

---

[79] Newman & Landay 2000
[80] Newman & Landay 2000
[81] Used with permission of Virtual Identity AG
[82] Newman & Landay 2000

3.2.3.5 Prototypes

User interface prototypes differ slightly from the prototypes used in pure software development. They do not necessarily implement any or all the functions of the product; they do show the complete user interface of the system, though with varying levels of detail. The user interface prototype is iteratively evaluated by future users of the system and improved by the designers.[83] Section 3.2.5 provides more details on prototyping.

3.2.3.6 Specifications and Guidelines

The specifications document the intention the designers had while creating the prototype. They contain detailed explanations of all the design decisions in the system. Developers stick to the specifications created by the design team when implementing the system. While developers must adhere to specifications, adhering to guidelines is only a recommendation.[84]

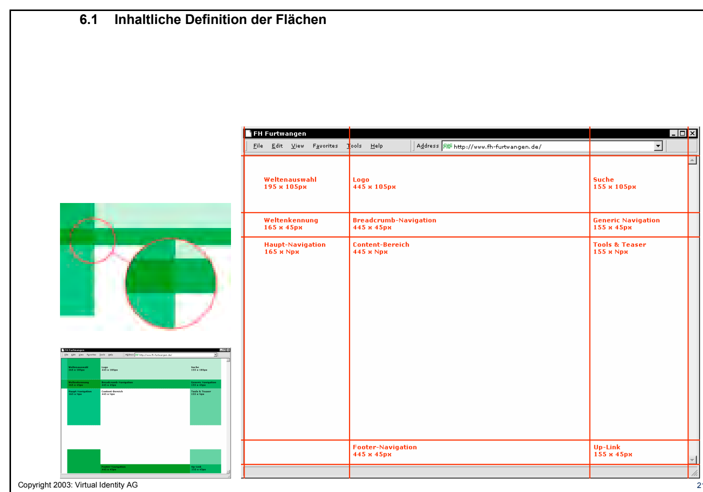


Figure 10. A Slide from a Presentation[85]

3.2.3.7 Presentations

Though not a part of the system itself, presentations are an important artifact because of the fact that their preparation requires a large amount of work. Additionally, they are often used in milestone meetings the get the customer's OK for the next step. They often include other artifacts like mockups or sitemaps. Presentations have to strike a balance between impressing the client, exactly representing project progress and finding the right level of detail.[86]

---

[83] Foley et al. 1990, 430
[84] Newman & Landay 2000
[85] Used with permission of Virtual Identity AG
[86] Newman & Landay 2000

3.2.4 Tool Support

Most applications that designers use focus on only one stage of the design process: Vector drawing and diagram drawing software like Freehand or Visio are used to create schematics and sitemaps, bitmap oriented graphics applications like Photoshop are used for mockups, and HTML-Editors or Director are used to create simple interactive prototypes.[87]

For web applications, user interfaces can be built using integrated web design tools like Macromedia Dreamweaver or Adobe GoLive. These tools have the look and feel of a drawing application, but directly produce HTML code. They are therefore very useful for creating interactive user interface prototypes.

DENIM[88] is a first approach to create an application to assist in more than one step of the design phase. It is an informal tool for sketching in early design phases. It permits designers to create multiple sketches within the same application and document, linking different levels of design to give a more complete overview of the system than is possible with distinct documents. DENIM does not connect user interface design to software design in general.

3.2.5 Low- vs. High-Fidelity Prototypes

When a UI design team decides to create a prototype, they have to decide whether to create a full-featured prototype using a user interface building or rapid prototyping tool or to draw all the screens and dialogs on individual sheets paper. These are called high- and low-fidelity prototypes, respectively.

Paper prototypes are the most widely used type of low-fidelity prototypes (Figure 11). The whole design team draws all the screens and dialogs on individual sheets of paper. They then ask a user to "interact" with the prototype by pointing on items to simulate mouse clicks. One team member then hands the next sheet of paper to the user, showing the result of the simulated click.[89] The user's actions on the interface are observed by a camera, a facilitator prompting for the user's thoughts, and member of the design team that takes notes about the problems the user encounters with the prototype.[90] After a session, the prototype is modified to eliminate the problems encountered by previous users. This sequence is repeated several times with different users, until the design team is satisfied with the quality of the prototype.

---

[87] Newman & Landay 2000
[88] Lin et al. 2000
[89] Snyder 2001
[90] Rettig 1994

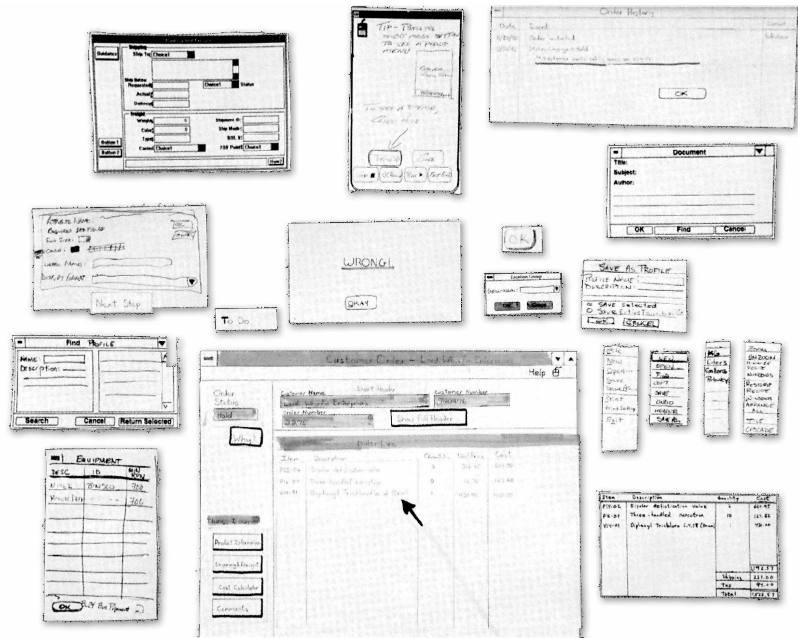


Figure 11. Part of a Paper Prototype[91]

Instead of using paper, low-fidelity prototypes can also be built and presented on a computer. However, evaluations using paper and computer-based prototypes created about the same number of user suggestions.[92]

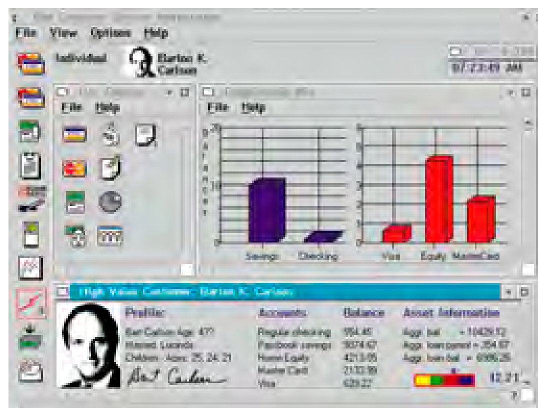


Figure 12. A High-Fidelity Prototype[93]

[91] Rettig 1994
[92] Sefelin et al. 2003
[93] Rudd et al. 1996

High-fidelity prototypes (Figure 12) are often built using graphical development tools like Macromedia Director or Microsoft Visual Studio.[94,95] They are fully interactive applications that may even have complete functionality. A typical user then operates the prototype, performing everyday actions and commenting on the responses he gets from the prototype. This is observed and recorded on video tape to be reviewed by usability experts, who identify usability problems in the prototype. After modifying the prototype, it is tested again, usually once.[96]

The advantages and disadvantages of each of the prototype classes are shown in Table 3.

	Advantages	**Disadvantages**
Low-fidelity prototype	Lower development cost Faster to create and iterate Evaluate multiple design concepts Useful communication device Address screen layout issues Useful for identifying market requirements Proof-of-concept Can be created by the whole design team	Limited error checking Poor detailed specification to code to Facilitator-driven Limited utility after requirements established Limited usefulness for usability tests Navigational and flow limitations
High-fidelity prototype	Complete functionality Fully interactive User-driven Clearly defines navigational scheme Use for exploration and test Look and feel of final product Serves as a living specification Marketing and sales tool	More expensive to develop Time-consuming to create Inefficient for proof-of-concept designs Not effective for requirements gathering Design team less willing to modify prototype

Table 3. Advantages and disadvantages of low- and high-fidelity prototypes[97,98,99]

[94] Thompson & Wishbow 1992
[95] Rosson & Carroll 2002, 213
[96] Rudd et al. 1996
[97] Rudd et al. 1996
[98] Snyder 1996
[99] Rosson & Carroll 2002, 206

3.2.6 Observations

We have interviewed user-interface designers on their individual methods in the design process. Our results indicate that sketches are also used for communicating with other team members.

According to one graphics designer, early in one project he and the project's account manager often had different ideas of how the UI might look like. This did not become obvious while talking over the design. So when he had prepared a high-fidelity mockup of the screen, the manager realized he had been misunderstood, and the graphics designer had to redo the screen, throwing away a few hours' work. In another project, the manager had prepared a wireframe model of the screen using Microsoft Visio to guide the graphics designer. This was seen as a major advance, and communication was perceived as much more efficient. As a result, the first design matched the manager's expectations.

Often designers do not create a whole screen, but only a smaller part of it. In this case, they sketch only this small part. This practice is well known and these sketches are widely perceived as being part of a screen that has already been described earlier.

Designers expressed their concern that formal modeling might give an expression more explicit and exact than intended by the modeler. They wished to be able to express rough design ideas that would be regarded just as that by others. One designer suggested keeping an informal nature in diagrams to reflect the incomplete and draft nature of the diagram.

These findings show that a language for describing graphical user interfaces on a high level of abstraction should be beneficial to the duration and the outcome of the design process.

3.3 Conclusion: Integrate Software and GUI Design

3.3.1 Why is it needed?

The GUI is an important part of most software systems and can be critical for the commercial success of a product. As can be seen from 3.1 and 3.2, software development cares little about GUI design. Issues of graphics design are handed over to user interface designers, who magically create a pleasant interface.

However, the software development processes drive a project. If the project manager of a software project neglects GUI design like the processes do, he runs the risk of failing to create a

user interface that meets the most basic requirement: being usable by its audience. Therefore, adding some control about graphical user interface design to processes will be beneficial.

The UML has grown into the most widely used modeling language, and it is used by many processes to specify models of the software system under development. It is already capable of modeling some of the aspects of GUIs. First of all, there is the use case diagram to capture user requirements before creating the user interface. From these requirements, designers can make conclusions for the GUI, and the GUI can later be tested against these. The UML can flex its muscles when it comes to modeling the details of every interaction with the system. This is the semantic design of the GUI (see 2.2), with the use case diagram, the class diagram and the activity diagram modeling inputs, outputs and structure of interactions. Designers can use the semantic GUI model to create a sound system and give exact instructions to the developers who will do the lexical design of the system and implement it.

But just like the processes, the UML does not provide a means to model all the aspects of the GUI. What is missing is support for the most obvious part of a GUI: The graphics layout. UML does not specify a diagram that is capable of modeling GUI layout (cf. 3.4.1). Adding support for this task to the UML is essential for the integration of software design and GUI design. We will therefore integrate the layout of graphical user interfaces into UML using a UML extension; extending the processes is left to others. With this extension as a foundation, the well-known UML diagrams can take up and specify all other aspects of the user interface.

3.3.2 What will it do?

Generally, the proposed extension improves communication of layout issues within the team and towards outsiders. Hand-drawn schematics often require artists to look more informative and impressive. A diagram with its non-artistic characteristics can be drawn and understood by anyone.

3.3.2.1 Graphics Design

Extensive modeling helps web designers to focus on the site's design and explore more design choices in a short time.[100] The more complete overview of the whole user interface points out inconsistencies and usability issues.

Designers have to manage multiple versions of one design idea. Sketching on paper can make this task quite troublesome.[101] By using models in electronic form instead of sketching on pa-

[100] Newman & Landay 2000
[101] Newman & Landay 2000

per, revisions and states of a user interface design can be controlled by a version control system and thus benefit from it.

If the system analyst creates wireframes of his ideas, the graphics designer will be able to create an interface that suits the analyst's ideas and meets all requirements more quickly.

On the other hand, graphics designers are artists. They might have objections against using formal diagrams from the domain of software engineering as a basis for their work because they feel restricted by them too much. This must be taken into consideration when developing an integration approach.

Another concern of graphics designers could be the fear of non-designers interfering with their work and dictating decisions that would be better left to a graphics design expert. This requires careful attention when extending software development processes; the UID should only be used for specifying layout *ideas*, not for dictating the work of graphics designers.

3.3.2.2 Software Engineering

The software developers are the ones who implement the user interface. Better models of the GUI will help them understand the application and its user interface and thus make fewer errors when specifications are inaccurate.

When ideas differ, rougher designs can be agreed upon more quickly than finished designs. Wireframes can be created earlier and much quicker than mockups. Thus, wireframes give a software developer clues on how to implement the foundations of a user interface earlier.

3.3.2.3 Information and Navigation Design

Mockups and prototypes take much more time to create than wireframes representing the same design do. "Because of the substantial amount of work they've put in, the team has an emotional investment in the status quo and will naturally tend to 'defend' their design."[102] Therefore, if mockups or prototypes have been created, the creators sometimes are reluctant to update and change them. Information and navigation designers often have to change the way an application is organized. This requires restructuring designs and prototypes. Our extension makes applying necessary design changes easier by minimizing the amount of work spent on creating the GUI model. Navigation designers get an artifact they can work with and that combines well with advanced UML modeling methods.

---

[102] Snyder 1996

3.3.2.4 Project Management

Software tools for formal modeling and prototyping can be used to provide a preview of an application at an early stage, ultimately reducing the project's risk. Presenting a model of an application to its future users can help avoiding many problems that arise when the application has to be changed late in the development process.[103,104] Formalized modeling methods help these users to understand models more quickly. They also help coordinating the various specialists involved when the design is carried out[105].

A diagram of the user interface that is linked to use cases may be used as an argument towards the customer why the user interface is laid out the way it is. Any design changes requested by the client can be incorporated quickly.

3.4 Approaches to Integration

Creating models of user interfaces is not a completely new idea. In recent years, research has emphasized modeling the interactional and navigational part of user interfaces.[106,107,108,109,110] These approaches disregard GUI layout and leave it to GUI designers. Some other researchers have strived to integrate the work of these GUI designers into software development. In the following sections, we will try to tackle the same problem using plain UML at first, then with UML's extension mechanisms, and finally using third-party extensions or languages created for that purpose. In the remainder of this chapter, Analysis, we will show how the different approaches try to solve the problem. After we have developed our requirements and goals in chapter 4, we evaluate them on how well they are suited for modeling GUI layout in section 4.3.

Describing user interface layout is a problem of describing structures that are "irrespective of time".[111] The main concepts of GUI layout are size, position, order, proportion, content, purpose. Typical relationships include "contains", "links to", "presents".

---

[103] Shneiderman 1997, 102
[104] Gould & Lewis 1985
[105] Shneiderman 1997, 157
[106] da Silva & Paton 2000
[107] Dolog & Bieliková 2002
[108] Baresi et al. 2001
[109] Gorshkova & Novikov 2002
[110] Lieberman 2001
[111] OMG 2003b, 590

3.4.1 Using basic UML

> *"The UML is a general-purpose modeling language. For specialized domains, such as GUI layout, [. . .] a more specialized tool with a special language might be appropriate."*
>
> *UML Reference Manual, p.4*

> *"This is consistent with a general problem of UML: the focus is on technically-oriented design, including architectural and implementation issues [. . .], not human-centred specification of functionality and behaviour."*
>
> *Hallvard Trætteberg*[112]

Although basic UML (i.e. without any extensions) does not seem to be very well-suited for GUI layout,[113] we will investigate how it performs in this domain. We will then review some extensions researchers have suggested to fill this gap.

3.4.1.1 Standard Diagrams

Due to the structural nature of GUI layout, if it can be modeled using plain UML, it must be done using one of the structure diagrams (Figure 13). We will therefore analyze which of these is best suited for GUI layout.

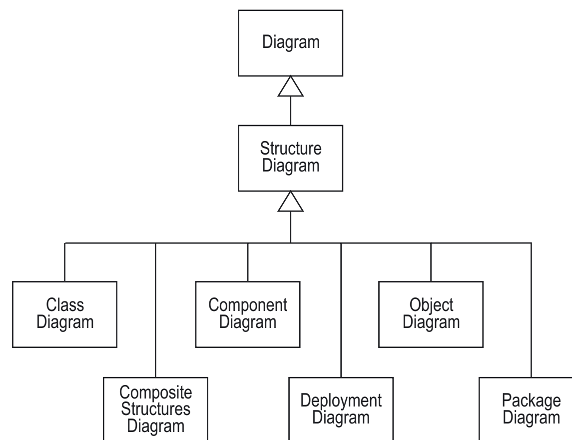


Figure 13. Taxonomy of UML Structure Diagrams[114]

[112] Trætteberg 2002, 18
[113] Palanque & Bastide 2003
[114] OMG 2003b, 590

The class diagram "shows a collection of declarative (static) model elements [...] and their contents and relationships".[115] Applied to the problem of GUI layout, the contents of the model elements correspond to the "contains" relationship of GUI layout. Accordingly, the "links to" and "presents" relationships can be regarded as general relationships. Concepts like size and position cannot be modeled implicitly.

The composite structure diagram is "a diagram that depicts the internal structure of a classifier, including the interaction points of the classifier to other parts of the system. It shows the configuration of parts that jointly perform the behavior of the containing classifier".[116] It is focused on collaborations and how they are formed. Although screen contents can be seen as the internal structure of a screen, they by no means perform its behavior. Additionally, concepts like size and position cannot be modeled implicitly.

The component diagram "shows the organizations and dependencies among component types"[117], with a component being "a physical, replaceable part of a system that packages implementation"[118] and providing a set of functions to other components. Screen layout can be seen as some sort of physical, replaceable part or even implementation. Components contain classes and include complex behavior that they provide as a service to other components. Components are complex pieces of software, while screen layouts are a flat view of one aspect of a system. Concepts like size and position cannot be modeled implicitly in the component diagram either.

The deployment diagram "shows the configuration of run-time processing nodes and the component instances and objects that live on them".[119] It is similar to the component diagram in that its elements are instances of components. Therefore, they are not suited for modeling GUI layout as well.

The package diagram shows how elements are grouped to create meaningful subpackets. Its main purpose is logical grouping, and it includes little semantics beyond this task. Once more, concepts like size and position cannot be modeled implicitly.

Judging from these observations, of the standard UML diagrams, class diagrams and composite structure diagrams are best suited for GUI layout. Figure 14 shows how a screen containing common elements can be modeled as a class diagram using standard notation. Obviously, the layout of the screen can not be expressed.

---

[115] RUMBAUGH et al. 1998, 190
[116] OMG 2003b, 7
[117] RUMBAUGH et al. 1998, 222
[118] RUMBAUGH et al. 1998, 216
[119] RUMBAUGH et al. 1998, 252

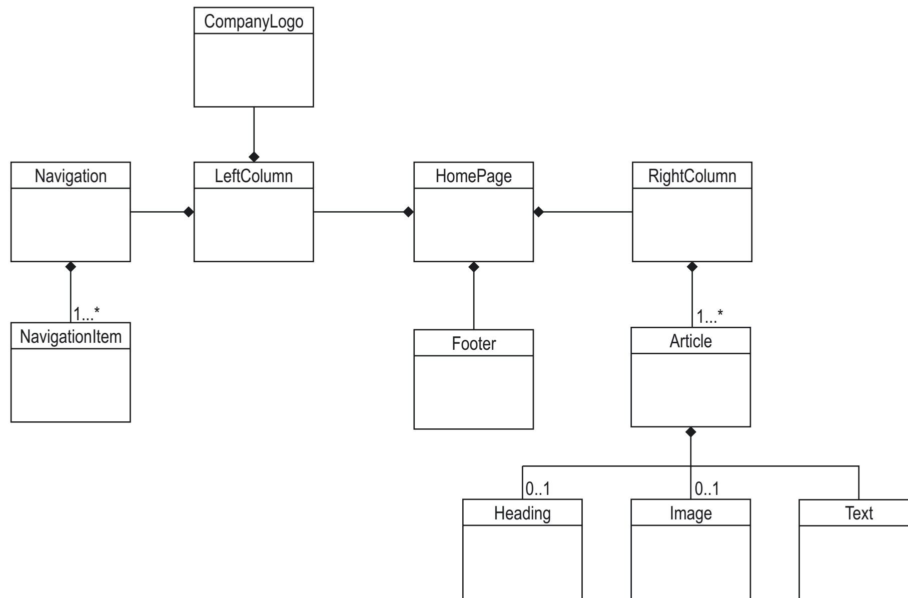


Figure 14. A Screen Modeled as a Plain Class Diagram

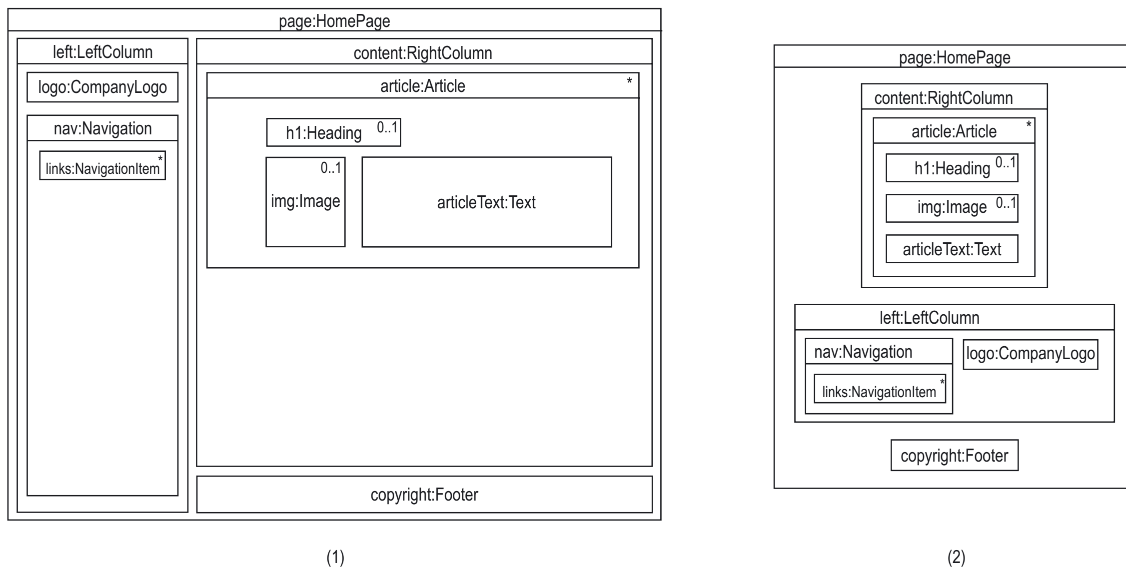


Figure 15. One Screen Modeled as Two Composite Structure Diagrams – Layout Information Bears No Semantic Meaning

Figure 15 shows two alternative ways of drawing the same diagram when compositions are implicitly modeled by graphically nesting them[120], seemingly creating a useable layout diagram in subfigure (1). However, as layout information, i.e. the way how elements are arranged, bears no semantic meaning in standard UML, subfigure (2), though it looks very different from (1),

[120] OMG 2003b, 172

bears the same semantic meaning. (2) is not even implausible; while in (1) the elements have been arranged and dimensioned according to the desired UI layout, in (2) they have been arranged in order of importance and dimensioned to occupy as little space as possible. This example illustrates the problem all standard UML diagrams have: Layout information is disregarded from a semantic point of view.

In addition to this general problem of UML, the default notation is the same for all classes or derivate elements. If the diagram for UI layout is to be accepted by designers, it should resemble existing practices as described in section 3.2.3 (see also section 4: Requirements and Goals). While the former cannot be tackled with existing UML elements, the latter can be improved using UML's extension mechanisms.

3.4.1.2 Extension Mechanisms

UML 2.0 can be extended through profiles.[121,122] A profile can tailor the UML metamodel for different platforms or domains. Profiles are "a straightforward mechanism for adapting an existing metamodel with constructs that are specific to a particular domain, platform, or method".[123] They may create additional constraints to the metamodel, but they may not take away any existing constraints. "There is nothing else that is intended to limit the way in which a metamodel is customized" using profiles.[124]

The most important elements of UML 2.0 profiles are stereotypes. A stereotype extends a class of the metamodel by adding additional semantics and possibly modifying its syntax. Usually, a stereotype is drawn as the class it extends with its name in guillemots (e.g. «stereotype»). The notation of a stereotype can be changed by attaching specific notation to it:[125] Instead of the default notation of a class, an arbitrary icon can be defined to represent the stereotype. Constraints can be used to change the semantics of the stereotype. They are Boolean expressions that cause the model to be ill-formed if they evaluate to false. They are often expressed in a formal language like OCL, but may also be formulated in natural language.[126] In addition to Constraints, stereotypes can have additional attributes, the meaning of which can be defined by the creator of the profile.

Conallen[127] describes a UML profile for web applications, focusing on architecture and the interaction between client and server pages. The stereotypes he uses are suitable for sketches on

---

[121] OMG 2003a, 164-178
[122] OMG 2003b, 569-584
[123] OMG 2003b, 569
[124] OMG 2003a, 164
[125] OMG 2003a, 176
[126] OMG 2003a, 50
[127] Conallen 2002

paper. The extension is not meant to be used for GUI layout, so it does not provide any benefit for this domain. Of course, it could be used in addition to any extension focusing on GUI layout, as UML extensions can be combined.

Koch et al.[128,129], Baumeister et al.[130] and Hennicker & Koch[131] use the extension mechanisms provided by the UML 1.x to create a UML profile for hypermedia design called UWE. This profile is to replace the initial sketches made by designers as described in section 3.2.3. The looks of the profile elements do not resemble designers' sketches created in that phase, though. The approach uses "specific stereotypes to model the navigational and presentational aspects of web applications"[132]. It is based on a conceptual model of the application, which is presented through the presentational model and interconnected through the navigational model. The presentational model contains frames, dynamic areas ("presentation classes") and primitives like text, images and audio/video. Each element is presented within a composite structure diagram through a stereotyped class (Figure 13). Semantics implicitly include rough information about relative size and position, though this is not reflected in the extension's metamodel. The diagram is meant to model "only the structural organization of the presentation, […] and not the layout characteristics […]. Such decisions are taken typically during the development of a user interface prototype or in the implementation phase." "The abstract user interface design may be considered as an optional step as the design decisions related to the user interface can also be taken during the realization of the user interface. However, the production of sketches of this kind is often helpful in early discussions with the customer."[133]

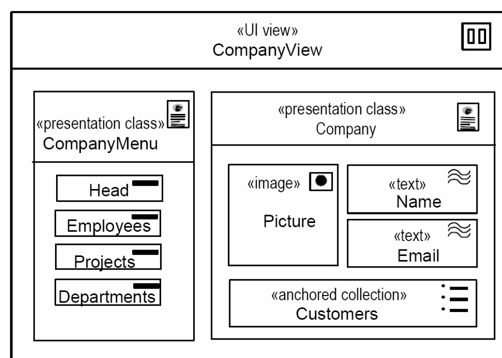


Figure 16. GUI Layout Modeled Using the Notation Proposed by Baumeister, Koch & Hennicker

[128] Koch 2001
[129] Koch et al. 2000
[130] Baumeister et al. 1999
[131] Hennicker & Koch 2001
[132] Koch et al. 2000
[133] Hennicker & Koch 2001, 6

3.4.2 Other Approaches

Several researchers have recognized the lack of support for layout information in UML and thus have taken different approaches to solve this.

Phillips and Kemp[134] propose two support artifacts, extended tabular use cases and UI element clusters, to fill the gap between user interface modeling and prototyping in the RUP. These artifacts can be used by a screen designer to prepare a user interface sketch in a following design step. In the first step, required user interface elements and workflows are identified and ordered in the extended tabular use case representation to form a "flow of events". Afterwards, the required functional elements are grouped into UI element clusters to form the visual interface (Figure 17). These clusters are then transformed into unformalized UI interface sketches by a designer. Layout information is stored implicitly, but bears only weak semantic meaning: The layout of the UI element clusters only reflects basic design choices like relative position and order. Switching to the design of real screens or to prototyping requires a switch of methods from diagramming to drawing or programming. This method is focused on identifying the tasks and UI elements needed for a use case. It provides a means for modeling simple interactions and only basic layout principles. Although it is based on UML use cases, it introduces its own notation, which is based neither on UML nor on designers' sketches.

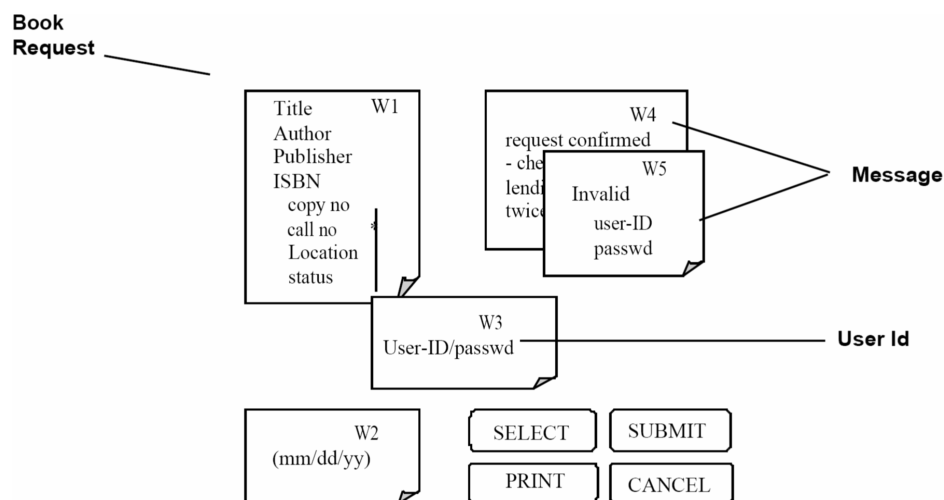


Figure 17. UI Element Cluster by Phillips & Kemp

[134] Phillips & Kemp 2002

OMMMA-L[135,136] is a UML-based language for modeling multimedia applications. It extends UML with elements and diagrams to model time-dynamic behavior and screen layout. The architecture of OMMMA-L goes beyond a profile, as it adds its own metaclasses instead of using stereotypes. In OMMMA-L, the logical structure of an application and its "interactive control" is modeled using plain UML. "To describe the temporal ensembling of different media objects" and for GUI layout, specialized and more advanced language constructs are introduced. GUI layout is modeled using the presentation diagram, which allows "an intuitive description of the layout, i.e. the spatial arrangement of visual objects at the user interface." The presentation diagram shows the layout of the user interface using bounding boxes. A bounding box is a virtual area on the screen that has size and position and serves a purpose, by being either interactional or visualizational. Interactional object may allow user interaction or trigger events, visualizational object passively present text, images etc. (Figure 18). OMMMA-L provides a solution to model the complete user interface of an application, including GUI layout. The presentation diagram uses a notation of its own, not resembling designers' sketches and using text to express the function of a bounding box. Size and position are not represented in the metamodel.[137]

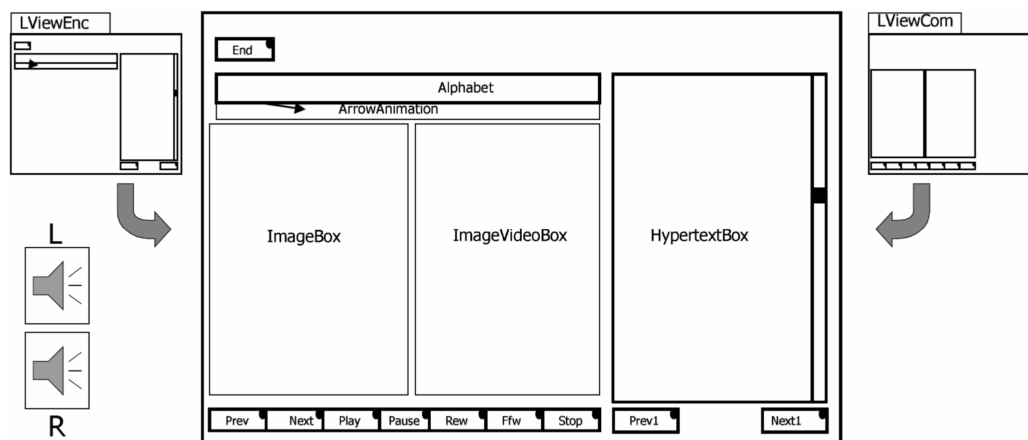


Figure 18. Screen Modeled with OMMMA-L

[135] SAUER & ENGELS 1999a
[136] SAUER & ENGELS 1999b
[137] SAUER & ENGELS 1999c

3.4.3 Diagram Interchange

The UML 2.0 Diagram Interchange[138, 139] (DI) is a part of the UML metamodel that has been introduced with UML 2.0 to store diagram layout information. It is MOF-compliant and adds a supplementary package to the UML 2.0 metamodel. DI is one of the four integral parts of UML 2.0[(140)] (infrastructure, superstructure, OCL, DI) and will be used as an enhancement of XMI.

DI is an evolution to the XML Metadata Interchange (XMI)[141], which has been developed to exchange diagrams between different modeling tools. However, diagram information in XMI only includes the semantics of a model, not its syntax, i.e. how the elements are arranged and dimensioned. For complex models, this adds a serious amount of information, as elements close to each other often belong to the same logical groups. Therefore, when a diagram is interchanged using XMI, much of the information included in the source diagram is lost.[142]

The principle used in DI is that all UML diagrams can be modeled as graphs. All visible elements of the UML notation are represented by either GraphNodes or GraphEdges (see appendix A for the DI metamodel). To model “the representation of more complex model elements”[143], the mathematical graph model has been extended with the concept of nesting. A diagram element can contain any number of graph elements, or an entire subgraph.

Every GraphEdge and every GraphNode stores its position relative to its surrounding container. GraphNodes also store their size; the size of GraphEdges is controlled using their waypoints. As every diagram element of UML 2.0 is represented either by a GraphEdge or by a GraphNode, element size and position of every UML 2.0 diagram element is automatically stored if DI is applied in the model[144]. The relevant metaclasses are Dimension and Point, which are subclasses of DataType. Please see Appendix A and the Diagram Interchange Specification for more details.

[138] Boger, Jeckle et al. 2002
[139] OMG 2003c
[140] OMG 2001, 1
[141] OMG 2002a
[142] Unisys 2000
[143] OMG 2003c, 9
[144] Jeckle 2004

Chapter 4

Requirements and Goals

In the previous chapter, we have learned that modeling is common practice among software designers as well as GUI designers. Software designers do it using special purpose modeling software, GUI designers do it by drawing by hand or using a graphics application.

In this chapter, we draw the conclusions from the facts learned during analysis, and develop requirements according to the work examined during software and GUI design. The many requirements thus captured are then condensed into few goals we aim to achieve in creating our solution. Additionally, we evaluate the approaches introduced in section 3.4 according to the requirements and state what is new in our approach compared to existing ones.

4.1 Requirements

4.1.1 User Groups

People are lazy. When they know how to do something satisfactorily, they will usually want to apply this knowledge over and over, resulting in methods of best practice. Generally, a new method will be accepted more quickly if it provides significant benefits over existing methods and if it introduces as little changes as possible for the users. This principle has to be applied to every user group, and will be in the following sections.

4.1.1.1 Graphics Designers

As explained in 3.2, graphics designers do a lot of initial sketching before creating elaborate designs. They share a common sketching style: how designers draw an image or heading often is similar between domains and companies. To increase acceptance among graphics designers, any approach to modeling GUI layout must use the symbolism of designers' sketches. Wireframes are an established practice and a good example of reusing the symbolism of schematics for computer-drawn versions.

"The modelling languages should be based on few and simple basic concepts and constructs, to make them easier to read and write."[145] This is important in order not to constrain creativity when drawing the diagram by offering a large number of elements to choose from, making modeling overly complex. If creativity is constrained seriously, designers will keep using their old method of sketching.

Modelers must have the possibility to first create informal and incomplete models and gradually refine them later on if required. "The visual notation should be flexible."[146]

Software tools for creating such models should be similar to traditional tools for drawing user interfaces and should support round-trip engineering.[147] Tools should be designed for productivity and speed of use.

If the diagram is based on UML, the concept of reification[148] should be used carefully and wisely. Reification is the explicit modeling of abstract concepts like an "is part of" relationship using special model elements. In the UML, reification is used in almost every diagram. For instance,

---

[145] Trætteberg 2002, 20
[146] Trætteberg 2002, 20
[147] Trætteberg 2002, 20
[148] Rumbaugh et al. 1998, 410-411

an "is part of" relationship (a composition) in a class diagram is drawn as a line between two classes with a solid diamond on the end of the containing element. Reification makes models more explicit, but also much more abstract and harder to read for unfamiliar users. In this case, this is necessary and makes sense; the gap between an abstract graphical representation of a program and its code is quite large. In the case of GUI layout, the step of abstraction between the graphical layout of a UI and its abstract graphical representation is much smaller. As both are graphics-based, all relationships that exist within one screen of the real user interface can be expressed in its abstraction implicitly without making use of reification. However, reification may be necessary to express the more complex aspects of user interfaces like inter-screen relationships, interactivity, and timing.

4.1.1.2 Software Developers

Software developers do most of their work in an IDE (see 3.1.4). If they are to make use of a diagram for GUI layout, their IDE should support it. Most IDEs support UML, so basing the diagram on UML will facilitate its integration into software developers' daily work.

As software developers are used to formal languages, diagram elements should be formal, unambiguous, and well-defined.

To create synergies, generating user interface stubs or prototypes from the GUI layout model without programming them manually should be possible.

4.1.1.3 Information and Navigation Designer

"There is a need for combining task models with models of the user interface's structure and behaviour, while still keeping them separate conceptually."[149] Therefore, the diagram should be easy to integrate into modeling interaction and activities. If the diagram is based on UML, task and behavior can be modeled using the appropriate UML diagrams.

4.1.1.4 Customers

The diagram must be easy to understand and create. For instance, reification should be used with care, and the elements of the diagram should be distinguishable easily.

Paper prototypes are an inexpensive and expressive way of testing software (see 3.2.5). The diagram should be able to be used for paper prototypes directly so they can be generated and modified by software from within the design process.

---

[149] Trætteberg 2002, 20

4.1.2 Workflows

To enable digital storage and archiving as well as version management, the diagram should be based on and built with software tools (contrary to being designed to be hand-drawn).

Many designers do sketching at the beginning of a project, and project managers express the requirements that they pass on to the graphics designers. To create as little additional overhead as possible, creating the diagram should integrate into these workflows. The same holds for software developers: "The formalisation of the modelling languages should allow for integration with existing languages used in software engineering."[150]

Creating a layout should be a matter of few minutes. The diagram itself and supporting tools should be designed to enable their users to create several similar diagrams quickly.

4.1.3 Summary of Requirements

Some of the requirements collected above are more important than others. The following is a prioritized and filtered list of requirements:

1. A diagram should model the layout of a screen.
2. Layout information should bear semantic meaning.
3. Layout should be modeled implicitly and without using reification.
4. The diagram language should be easy to learn, write and read.
5. The symbolism of the diagram should be derived from designers' sketches.
6. The diagram should be linked to and usable with an established modeling language like UML.
7. Creating a diagram with a specialized tool should only take a few minutes.
8. Creating a diagram should not impose much additional work.
9. It should be possible to extend the most important software development processes to support use of the new diagram.
10. Supporting software tools should have the look and feel of a diagramming application.
11. The diagram should support several levels of detail.
12. The diagram should be supported by IDEs.
13. Software tools should provide round-trip engineering of GUIs using the diagram.
14. The diagram should be designed for being created with a computer.

[150] Trætteberg 2002, 20

4.2 Goals

The goals for creating the UML extension for GUI layout are generally formed from the requirements in section 4.1. However, for this thesis, we will focus on creating only the core component, the diagram itself. Therefore, we will disregard wide tool support for the time being and only create a prototype of a diagramming application.

Our primary goals are to create a diagram that

- is an easily comprehensible abstract representation of a GUI's layout (thus meeting requirements no. 1, 3, 4 of section 4.1.3)
- extends the UML metamodel with layout information (requirements no. 2, 6 [partly], 14)
- will be accepted by its users (requirements no. 5, 8)

Secondary goals in designing our extension are:

- Creating a diagramming application prototype (requirements no. 7, 10)
- Creating links to other UML diagrams (requirement no. 6)

With a new diagram from our UML extension describing the layout of a GUI and providing classifiers for important screen elements, modeling activities and behavior using UML diagrams should be straightforward.

It is beyond the scope of this thesis to implement IDE support for our extension (requirement no. 12), to create software development process extensions (no. 9), or to provide capabilities for round-trip engineering (no. 13). Modeling in several levels of detail (no. 11) requires additional profiles to be created for other diagrams. This can be done independently from our approach.

4.3 Evaluation of Existing Approaches

Based on the requirements developed in the previous section, we will now evaluate the approaches described in section 3.4. For a detailed explanation of each method, please see the appropriate sections from page 30 on. Table 4 shows an analysis of methods for GUI layout according to the requirements developed in this chapter. The last column shows how our approach compares to the existing ones.

Reqmt. no.	UML class diagram	UWE	UI element clusters	OMMMA-L	Our approach
1	✓	✓	✓	✓	✓
2	×	(✓)[a]	(✓)[a]	(✓)[a]	✓
3	✓	✓	✓	✓	✓
4	✓	✓	✓	✓	✓
5	×	×	×	×	✓
6	✓	✓	(✓)[b]	✓	✓
7	✓	✓	✓	✓	✓
8	✓	✓	✓	✓	✓
9	✓	✓	✓	✓	✓
10	(✓)[c]	✓	–[d]	–[d]	✓
11	×	×	×	×	×
12	✓	(✓)[e]	×	✓	×
13	×	×	×	×	×
14	✓	✓	✓	✓	✓

[a] semantic meaning is not established in the metamodel
[b] notation is untypical for UML
[c] common UML modeling tools
[d] no drawing tools exist
[e] reference implementation for ArgoUML

Table 4. Evaluation of Existing Approaches

The essential difference between existing approaches and our own one is the degree of how deep layout information is established in the metamodel of the extension (requirement no. 2). As class diagrams are plain UML, they lack layout information completely. UWE, UI element clusters and OMMMA-L do not establish the semantic meaning of layout information within their metamodels. Although this has been difficult to realize formally with earlier versions of UML, it should at least be captured in a natural-language constraint on the respective metamodel elements. Instead, the three approaches specify the meaning of layout in their descriptions, which makes them a lot less expressive for the task of UI layout. Our approach will integrate layout information in its metamodel.

Another big difference is the look of the diagram (requirement no. 5). Earlier approaches established their own notation, requiring users to learn it. We will adopt common sketching practice, thus minimizing learning effort. The look of a diagram is very important to convince initially skeptic users of its benefits and get them to use it.

The syntax of UI element clusters is different from standard UML and can be confusing; e.g., workspaces resemble UML notes. Our solution will use UML notation as often as possible.

Chapter 5

Design

Now that we have specified requirements and goals, it is time to apply them and create our own solution.

Although the citations on page 30 suggest that GUI layout modeling is not possible with UML, this has changed with the advent of UML 2.0 including Diagram Interchange. We have therefore decided to design our solution as a UML 2.0 profile called *UML Profile for GUI Layout* that incorporates layout information by using the Diagram Interchange package of the UML 2.0 metamodel. It is designed to meet the requirements and goals elaborated in the previous chapter.

By choosing the lightweight solution of a profile – in contrast to a heavyweight approach using a MOF-based extended UML metamodel and creating a UML dialect – we ensure compatibility with existing tools and thus facilitate integration of the profile into tools by their respective vendors. In particular, any tool that is capable of using user-defined graphical stereotypes and creating XMI data with embedded DI data should already be capable of handling the elements proposed with this profile. It is the first solution in this field to incorporate the possibilities of UML 2.0 and Diagram Interchange, and one of the first applications of Diagram Interchange in general. In this respect it has an advantage over earlier approaches that could not establish layout information very well because the UML metamodel did not provide any means to do so prior to version 2.0 of the standard.

In this chapter, we will introduce the concepts of or profile in detail, explain how they can be used in diagrams and give some examples of the profile in action.

5.1 Classification of Stereotypes

Berner[151] et al. have developed a classification of stereotypes according to their expressive power. They discern between four kinds of stereotypes:

- "A *decorative stereotype* modifies the concrete syntax of a language element and nothing else."
- "A *descriptive stereotype* extends or modifies the abstract syntax of a language element and defines the pragmatics of the newly introduced element. The semantics of the base language remains unchanged. Additionally, a descriptive stereotype may modify the notation (the concrete syntax) of the stereotyped language element."
- "A *restrictive stereotype* is a descriptive stereotype that additionally defines the semantics of the newly introduced element."
- "A *redefining stereotype* redefines a language element, changing its original semantics. Concerning syntax, a redefining stereotype behaves in the same way as a restrictive one."

As we introduce stereotypes that are subject to constraints, most of our stereotypes are restrictive. For restrictive stereotypes, the authors state: "Restrictive stereotypes are first-class members in the language they are added to. They have the same expressive power and can be defined with the same degree of rigor as the elements of the base language themselves. Restrictive stereotypes are typically used to add missing features to some elements of a language, to strengthen weak features or to introduce a metalanguage on top of a given language."

For each class of stereotypes, they formulate several guidelines on how to create such stereotypes. These are their guidelines for restrictive stereotypes, followed by our comments on how we follow them:

- "Thoroughly investigate and discuss the need for the language extensions or modifications that shall be introduced with a restrictive stereotype.
- Never define restrictive stereotypes on the fly.
- Leave the definition of restrictive stereotypes to language and method specialists. For example, state in your stereotype policy [...] that restrictive stereotypes may only be defined

[151] Berner et al. 1999

by your software methods group and that a formal validation and approval process has to be employed.

- Take care that a restrictive stereotype is really a restrictive stereotype and not a redefining one. State the semantics of the stereotype as precisely and formally as possible."

We have considered and followed these guidelines in our design of the profile. In particular,

- the need for an extension of the UML metamodel with the concept of layout information has been discussed thoroughly in section 3;
- considering the amount of work spent in elaborating this thesis, we have not defined these stereotypes on the fly;
- all stereotypes have been syntactically well-defined using the UML 2.0 metamodel. The analysis chapter has brought up many issues we took care of, so we can regard ourselves somewhat competent in the modeling language and method;
- by sticking to the UML's extension mechanism, it is made sure that the stereotypes are not redefining, as UML prevents well-formed profiles from redefining UML language elements.

5.2 General Design Principles

The design restrictions imposed by the UML 2.0 profile mechanism have formed our architecture of the profile, as one of our goals was to stay conform to the UML standard. Particularly the following commandment influenced most of the constructs we have created:

> *"As part of a profile, it is not possible to have an association between two stereotypes or between a stereotype and a metaclass unless they are subsets of existing associations in the reference metamodel. However, it is possible to have associations between ordinary classes, and from stereotypes to ordinary classes. Likewise, properties of stereotypes may not be typed by metaclasses or stereotypes."*
>
> *UML 2.0 Superstructure*[152]

We have therefore designed all associations between our stereotypes as subsets of existing associations of the respective stereotype's superclass.

[152] OMG 2003b, 576

Our profile does not make use of color in diagrams, although it might seem appropriate due to the graphical nature of real life screens at M0. This is in accordance with UML: "UML avoids the use of graphic markers, such as color, that present challenges for certain persons (the color blind) and for important kinds of equipment (such as printers, copiers, and fax machines). None of the UML symbols *require* the use of such graphic markers. Users *may* use graphic markers freely in their personal work for their own purposes (such as for highlighting within a tool) but should be aware of their limitations for interchange and be prepared to use the canonical forms when necessary."[153]

The following sections contain explanations of the elements of the UML Profile for GUI Layout. They use a structure similar to the one defined in the UML 2.0 Infrastructure:[154] After the concept's name in the heading, a brief definition follows. It may simplify the concept for concise presentation, which is explained in detail in the description section. The following three sections cover attributes, associations and constraints that the element owns in additions to inherited ones. Any operations that are defined for this metamodel element are introduced in the next section. The semantics section that follows contains a detailed description of the meaning of the concept and an overview of subordinate structural properties. Notation is covered in the next section, followed by some examples. If there are alternate notations of an element (e.g. an actor can be represented by a box or by a stickman), they are explained in 'presentation options'. In the last section 'rationale', we will explain design decisions that may not be obvious and clarify possibly confusing points. If a section for an element contains no text, it is omitted.

Although stereotype icons are of special importance to our profile, they do not appear in the metamodel. This is because there is no way of defining them using UML constructs[155], and they are therefore described in the notation section of the respective stereotype.

The order of the elements in section 5.4 is not alphabetical; it is rather what we regarded as useful.

[153] RUMBAUGH et al. 1998, 451
[154] OMG 2003a, 33-35
[155] JECKLE 2004

5.3 Architectural Overview

5.3.1 Connection to UML

Profiles are the standard extension mechanism of UML 2.0 to allow for an easy and MOF-conformant extension of the UML metamodel. Profiles are regarded lightweight extensions and have limited possibilities to make changes to the metamodel. They may not change the semantics of existing metamodel classes, but they can introduce new constraints and semantics by the means of stereotypes.[156,157] Stereotypes are Classes that extend existing Classes with Attributes, Constraints, Notation, etc.

Our extensions are gathered into a UML 2.0 profile to create a lightweight and easily implementable extension of the UML 2.0 metamodel. It "extends UML's present modeling capabilities in a manner which is conformant to the official UML specification. In detail, UML's well-known class diagrams are extended to cover more meaning by adding a set of stereotypes defined by the GUI profile. The semantic information expressed by adding stereotypes to UML's predefined constructs is combined with spatial information added to the UML metamodel"[158] by the UML 2.0 Diagram Interchange (section 3.4.3). "Additionally, the set of stereotypes defined by the GUI profile which is offered for usage within UML models is accompanied by graphical symbols which may be used instead of the textual representation. Based on this, class diagrams which conform to the profile outlined below are well suited for describing both visual as well as semantic aspects of a GUI layout."[159]

5.3.2 Overview of Classes

The package diagram of the UML profile for GUI layout is shown in Figure 19, and its architecture is shown in Figure 20.

---

[156] OMG 2003a, 164-178
[157] OMG 2003b, 569-584
[158] Blankenhorn & Jeckle 2004
[159] Blankenhorn & Jeckle 2004

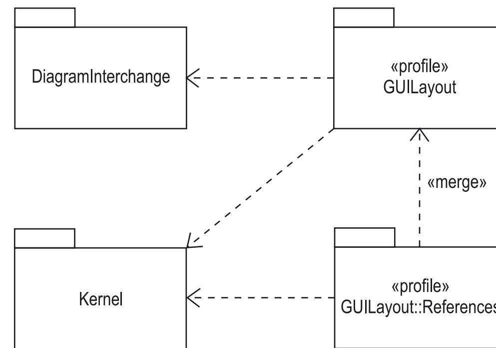


Figure 19. Package Diagram of the UML Profile for GUI Layout

The subpackage GUILayout::References is merged into the GUILayout package. Package merge is used "when elements of the same name are intended to represent the same concept, regardless of the package in which they are defined. A merging package will take elements of the same kind with the same name from one or more packages and merge them together into a single element using generalization and redefinitions."[160] The following sections 5.4 and 5.5 explain the two packages GUILayout and GUILayout::References, respectively. In the second package, only those aspects that are added to the concept of each element are specified. This means, for example, that the notation is only stated in the GUILayout package and is the same for the elements in GUILayout::References, because they are identical with the elements in GUILayout.

"Technically speaking the profile consists of a set of stereotypes applicable to UML's standard model element Class. The stereotypes defined by the GUI profile are organized in a hierarchical manner in order to emphasize their semantics. Furthermore, stereotypes which require the presence of another stereotype or type of stereotype (i.e. generalized stereotype) are interrelated by UML associations."[161]

5.4 Package GUILayout

The central class of the profile is ScreenArea, which represents a coherent area within the GUI. It is modeled as a stereotype of the standard metamodel class `Class` that stores information about its size and relative position using two DataTypes from the Diagram Interchange package, `Dimension` and `Point`, respectively. A ScreenArea can either be used as a container for other

[160] OMG 2003b, 101
[161] Blankenhorn & Jeckle 2004

ScreenAreas or provide a part of the functionality of the user interface. Accordingly, ScreenArea has two subclasses.

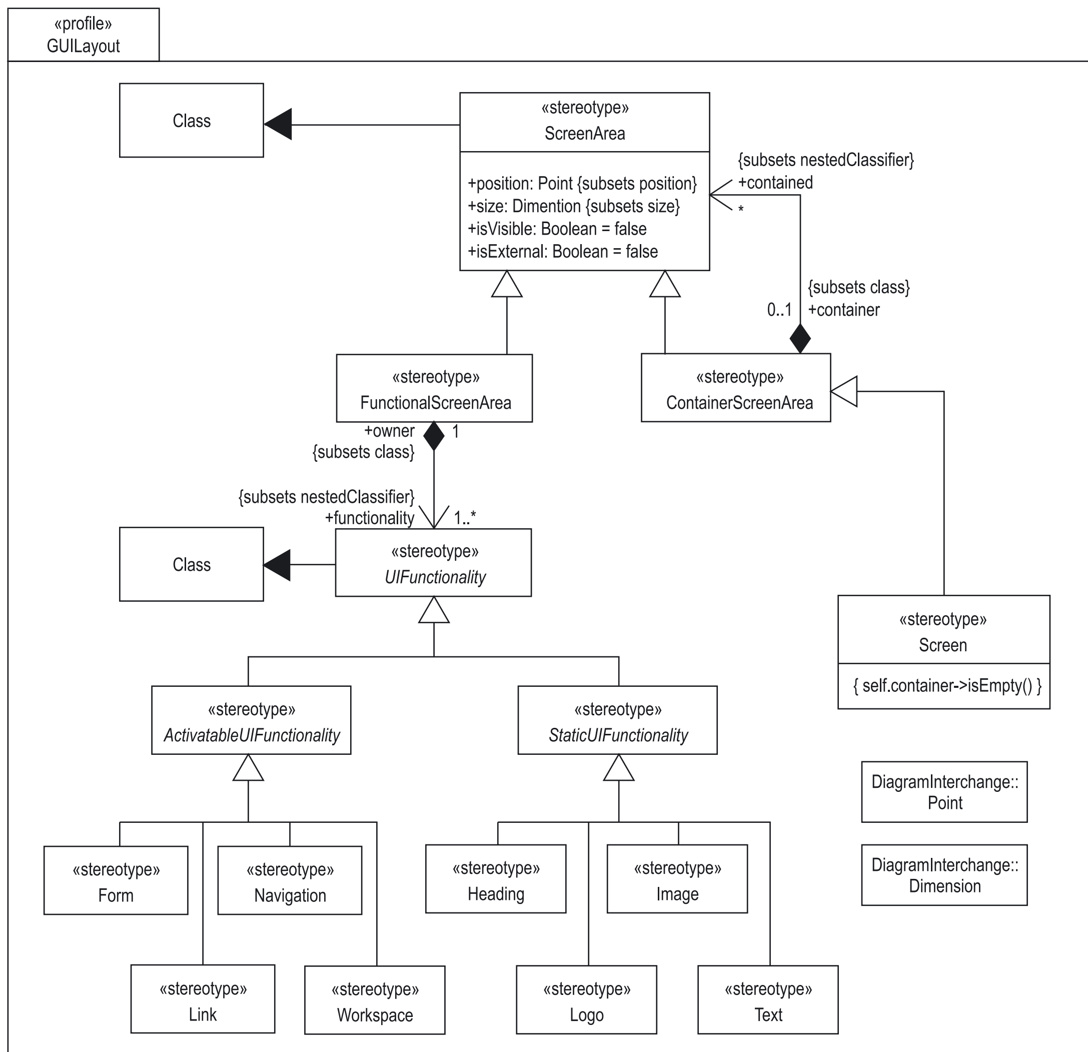


Figure 20. Architecture of the Package GUILayout

ContainerScreenArea is a concrete subclass of ScreenArea that can contain other ScreenAreas, forming a nested hierarchy of ScreenAreas. Screen is a special kind of ContainerScreenArea: It is the root of the ScreenArea hierarchy, thus it may not be contained in any Container-ScreenArea.

A FunctionalScreenArea is a ScreenArea that provides certain functionalities; therefore, it is associated with one or more UIFunctionalities. UIFunctionality is an abstract stereotype of Class that represents one functionality of the user interface. Concrete subclasses are classified

into ActivatableUIFunctionalities and StaticUIFunctionalities. ActivatableUIFunctionalities have behavior and can cause other ScreenAreas to be displayed; StaticUIFunctionalities statically display an UI element. Form, Link, Navigation and Workspace are the concrete subclasses of ActivatableUIFunctionality, Heading, Image, Logo and Text are concrete subclasses of StaticUIFunctionality. The notation of each of these concrete stereotypes is derived from the sketches graphics designers prepare while developing a screen design.

We have extended the informal modeling language observed in the work of screen designers with the two elements Workspace and Navigation in order improve expressiveness of the GUI profile. Their icons are based on simple metaphors and are easily comprehensible. A Navigation is substantially different semantically from a group of Links and thus has its own notation. The visual representation of an instance of Workspace often is a blank area and would otherwise have to be modeled using an empty ContainerScreenArea, omitting all of its semantics.

FunctionalScreenArea and ContainerScreenArea are specializations of ScreenArea. This implies that "an instance of the more specific element may be used where the more general element is allowed"[162], i.e. if a ContainerScreenArea may contain other ScreenAreas, this means that it may contain ContainerScreenAreas and FunctionalScreenAreas as well (but no Screen, which is constrained to have no container). This mechanism applies to all of the following, except when directly referring to specific classes.

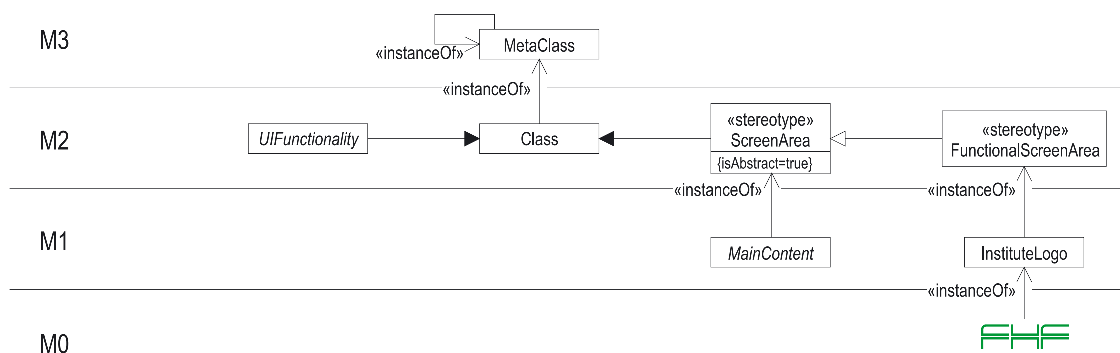


Figure 21. Profile Metamodel Layers

As ScreenArea is a *concrete stereotype* of `Class` in the M2 layer of the UML metamodel, it can be instantiated in M1 by adding it to a diagram, regardless of its `isAbstract` attribute. This attribute specifies that an M1 instance of ScreenArea may not be instantiated in M0, while an M1 instance of an M2 subclass of ScreenArea that has `isAbstract=true` may (e.g. an M1 instance of

[162] Rumbaugh et al. 1998, 287

ContainerScreenArea). This means that although ScreenArea has the constraint `isAbstract=true`, it can be added to a diagram, but it cannot be displayed on a real life screen due to the constraint. The subclasses of ScreenArea, ContainerScreenArea, Functional-ScreenArea and Screen, do not have this constraint, and thus can have instances in M0, i.e. be displayed on a real life screen.

However, *abstract stereotypes* of classes in the M2 layer (e.g. UIFunctionality) may not even be added to a diagram; they have been created to extract common characteristics from their subclasses into one superclass. Figure 21 shows examples of each type of abstract element and instances of concrete elements of the profile.

For brevity, we will therefore use the term 'a ScreenArea' as a short form of 'an M1 instance of the M2 stereotype ScreenArea or of an M2 subclass of ScreenArea'; in other words, a ScreenArea or a subclass in the model layer.

Accordingly, we will use 'an instance of ScreenArea' as a short form of 'an M0 instance of a concrete M1 instance of the M2 stereotype ScreenArea or one of its M2 subclasses'; in other words, an instance of a ScreenArea subclass on the real screen.

5.4.1 ScreenArea

A ScreenArea is a stereotype of `Class`[163] that represents an area within a screen of a graphical user interface.

Description

A ScreenArea is a contiguous area of a graphical user interface that serves a purpose. ScreenAreas are rectangular, but multiple independent ScreenAreas may be arranged to form arbitrarily shaped areas. Each ScreenArea is used to model an area on a screen mockup or piece of software.

Attributes

position : Point	The position of the top left corner of the ScreenArea. Subsets GraphElement::position.
size : Dimension	The size of the ScreenArea. Subsets GraphNode::size.

[163] Jeckle 2004

isVisible : Boolean = false	Specifies if M0 instances of the ScreenArea should have a graphical border. Default value is false.
/isExternal : Boolean = false	Specifies if the ScreenArea is specified outside of a Screen. Default value is false. This is a derived value.

Associations

container : ContainerScreenArea [0..1]	The ContainerScreenArea that contains the ScreenArea. Subsets class.

Constraints

[1] ScreenAreas that have a container must be an element of its contained attribute.

```
self.container->notEmpty() implies self.container.contained->includes(self)
```

[2] All non-subclassed ScreenAreas are abstract.

```
self.oclIsTypeOf(ScreenArea) implies self.isAbstract
```

[3] ScreenAreas that subclass another ScreenArea are external.

```
self.superclass->notEmpty() implies self.isExternal
```

[4] ScreenAreas that subclass another ScreenArea have the same size, position and container.

```
self.superclass->notEmpty() implies
       self.size=self.superclass.size and
       self.position=self.superclass.position and
       self.container=self.superclass.container
```

Semantics

A ScreenArea may be contained in a ContainerScreenArea.

In contrast to conventional UML elements, the positioning and sizing of ScreenAreas (the layout) bears semantic meaning. A GUI layout model is a model of a real life GUI, like a mockup or a prototype. The layout of the model determines the layout of its elements' instances in the M0 layer (cf. 2.1) that are created and arranged by a designer for that instance of the GUI. For this, "we rely on the inherent feature of UML 2.0 and re-use the existing layout information for describing dimensions and arrangement"[164] of ScreenAreas. "In essence, the spatial data which is part of every UML 2.0 compliant class diagram instance can be interpreted as information

[164] Blankenhorn & Jeckle 2004

describing the layout of the visual components of the GUI to develop."[165] The model is not intended to be taken by the pixel but may be if clearly marked so by the modeler (in an attached note, for example).

The coordinate system used for `position` and `size` is described in the Diagram Interchange Specification.[166] In essence, its axes are right-angled, with the x-axis pointing to the east and the y-axis pointing south.

The attributes `size` and `position` are measured in pixels along the axes of the coordinate system. `Position` is relative to the position of the ScreenArea's container. If the ScreenArea does not have a `container`, `position` is relative to the top left corner of the workspace of the tool used to create the diagram. Neither the width nor the height attribute of `size` may be negative. Scaling may be applied to any instance of ScreenArea to ensure it meets the size requirements at the actual user interface at M0. This implies that ScreenAreas within a diagram may be magnified to show their detailed layout.

As a ScreenArea may only represent a logical and spatial partitioning of screen space, an M0 instance of ScreenArea does not need to have a graphical border on a screen. This is specified by the `isVisible` attribute; it is true if instances of ScreenArea should have a visual border, and false otherwise.

All ScreenAreas are abstract, i.e. they must be subclassed by another ContainerScreenArea or FunctionalScreenArea to be displayed. A ScreenArea that subclasses another ScreenArea inherits its `position` and `size`, has the same `container` and is external. Abstract ScreenAreas are ones that are not fully specified yet, for example because they are used as placeholders for general purpose areas of the GUI that have different content depending on the particular screen that is displayed. Each of these different sets of content is specified in separate ScreenAreas (usually ContainerScreenAreas) that each subclass the abstract ScreenArea. Thereby they inherit its layout information and can be displayed instead of it. Section 5.9 has an example that shows the use of abstract ScreenAreas and ScreenArea inheritance.

A ScreenArea is called external when it is specified outside the bounds of a Screen in the model. In this case, it must subclass another ScreenArea and has its `isExternal` attribute set to true. ScreenAreas that are specified as nested ScreenAreas within their container have their `isExternal` attribute set to false, meaning they are defined at the position where they are displayed.

---

[165] Blankenhorn & Jeckle 2004
[166] OMG 2003c, 13

Instantiating a ScreenArea in M0 means providing it with all data it and any contained ScreenArea needs and then make the software system display it.

ScreenArea bounds may overlap.

Notation

The following rule applies to all subclasses of ScreenArea: Usually, the name of an instance of a subclass of ScreenArea is not drawn because the name string would be distracting within the diagram. However, it may be drawn if the name of the ScreenArea is of importance, e.g. if it is referred to in a separate diagram. If so, the name is written within the boundaries of the ScreenArea at the top left corner.

A ScreenArea is drawn as a box. The stroke of the box depends on the isVisible attribute. If `isVisible=true` the stroke is solid, if `isVisible=false` the stroke is dashed.

Abstract ScreenAreas and abstract instances of its subclasses have their name printed in italics in the upper left corner of their box when specified in a Screen. When specified outside a Screen, the name may be drawn outside the box.

Examples

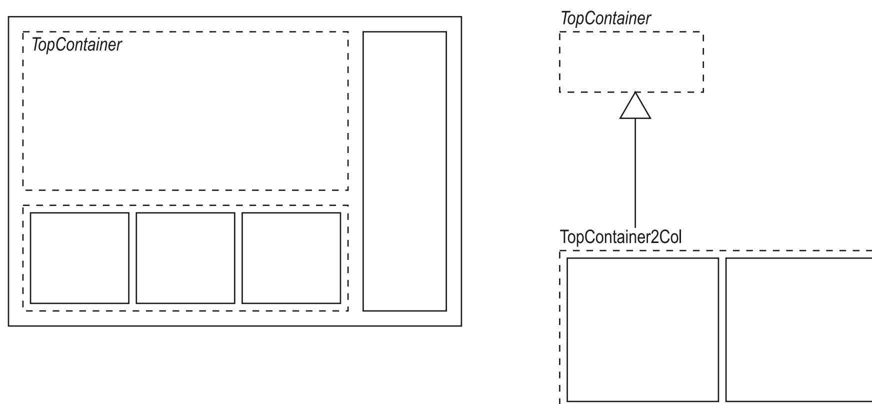


Figure 22. Example of ScreenArea Notation and ScreenArea Inheritance

Figure 22 shows an example of ScreenArea inheritance. The abstract ScreenArea named Top-Container is inherited by TopContainer2Col, which has the same size and position information as TopContainer.

Rationale

Please note that the alignment of ScreenAreas is not restricted to only horizontal or vertical. In fact, tool vendors may choose to allow any alignment to be used.

5.4.2 ContainerScreenArea

A ContainerScreenArea is a ScreenArea that can contain other ScreenAreas.

Description

A ContainerScreenArea is a concrete subclass of ScreenArea that serves as a virtual container for other ScreenAreas, creating a hierarchy of ScreenAreas.

Attributes

contained : ScreenArea [*]	The ScreenAreas that the ContainerScreenArea contains. Subsets nestedClassifier.

Constraints

[1] All contained ScreenAreas must have this ContainerScreenArea set as their container.

```
self.contained->forAll(sa | sa.container=self)
```

[2] A ContainerScreenArea may not contain itself.

```
self.contained->excludes(self)
```

[3] A ContainerScreenArea may not be contained in itself.

```
self.container<>self
```

Semantics

The container relationship expressed in the two attributes `container` and `contained` implies a spatial nesting of the `contained` ScreenAreas within the ContainerScreenArea. With ContainerScreenAreas, it is thus possible to create a nested hierarchy of ScreenAreas. This structure is often the same as the hierarchy of windows in windowed environments.

The top left corner of a ContainerScreenArea is used as the origin of the coordinate systems of the contained ScreenAreas. This implies that manipulations like moving or scaling a ContainerScreenArea also affect all ScreenAreas contained in it.

Several ScreenAreas performing similar functions might be contained in a Container-ScreenArea to show their direct spatial and semantic relationship. Whether this Container-

ScreenArea is graphically manifested (e.g. into a box surrounding the ScreenAreas) or not is determined by the `isVisible` attribute.

A ContainerScreenArea can be abstract to specify it as a general purpose ScreenArea that can have its layout changed, i.e. it is subclassed by one or more concrete ScreenAreas that can be put in its place. An abstract ContainerScreenArea must be subclassed by a concrete ScreenArea before the Screen it belongs to can be instantiated, and during instantiation one of these subclasses must be selected to be included in the display.

Notation

If a ContainerScreenArea does not contain any ScreenAreas, it is drawn as an empty box. If it does, all ScreenAreas that have this ScreenArea as their container and that are not external are laid out within the container according to their `size` and `position` attributes. Any contained ContainerScreenAreas and their contents are recursively drawn until `contained->isEmpty()`.

If the boundaries of a `contained` ScreenArea exceed the boundaries of its `container`, the parts of the ScreenArea that exceed the container's boundaries are not shown – in other words, they are clipped.

The user is encouraged to always use this notation instead of the default notation for stereotypes (stereotype name in angled brackets).

Examples

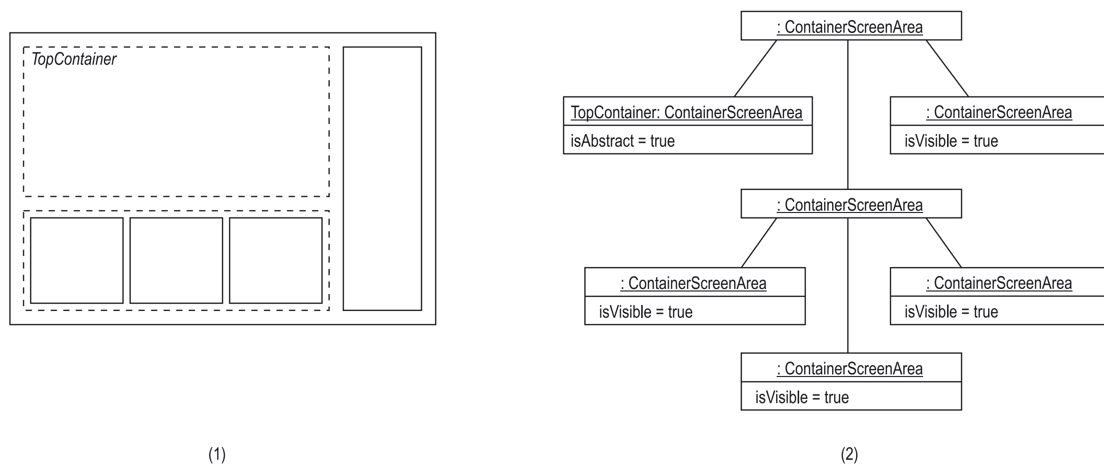


Figure 23. Nested ContainerScreenAreas as Concrete Syntax and Instance Specification

Figure 23 shows several nested ContainerScreenAreas. Subfigure (1) shows the model of a ContainerScreenArea and its contents. TopContainer is an abstract ScreenArea with no visual Bor-

der. Subfigure (2) shows the instance specification for this ContainerScreenArea. Size and position attributes have been omitted from the view.

Rationale

The containment relationship of ScreenAreas is modeled implicitly, i.e. the concept of reification does not apply here. The reason for this is that reification adds graphical elements that are only used in the model, not in instances of the model (i.e. on the real screens). As GUI layout exclusively relies on graphical information, these elements would be distracting. While software designers may understand and happily apply diagrams using reification, it is detrimental to the visual expressive power of the diagram for graphics designers.

5.4.3 Screen

A Screen contains all elements of a GUI that are displayed at one point of time. Its instances are for example application windows or a web page displayed in a web browser.

Description

A Screen is a ContainerScreenArea that may contain other ScreenAreas, but may not be contained in any ContainerScreenArea. If a model contains a Screen, it must be the root of a ContainerScreenArea hierarchy.

Constraints

[1] A Screen may not have a container.

```
self.container->isEmpty()
```

[2] A Screen is always visible.

```
self.isVisible=true
```

Semantics

A Screen contains everything that is displayed by the system on a single output device at one point of time. On window-oriented systems, a Screen usually corresponds to the application window, which contains all GUI elements of the system. Hence, only one instance of Screen can be displayed by the GUI at a time.

Notation

A Screen is drawn as a box with a bold top horizontal line. The user is encouraged to always use the stereotype icon instead of the default notation for stereotypes (stereotype name in angled brackets).

Examples

Figure 24 shows the notation of an empty Screen.

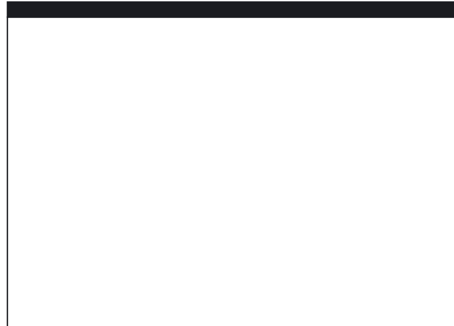


Figure 24. Notation of a Screen

5.4.4 FunctionalScreenArea

A FunctionalScreenArea is a ScreenArea that provides one or more functionalities.

Description

FunctionalScreenArea is a concrete subclass of ScreenArea that owns one or more UIFunctionalities to provide the functionalities associated with the owned UIFunctionalities.

Associations

functionality : UIFunctionality [1..*]	The functionalities that the FunctionalScreenArea provides. Subsets nestedClassifier.

Semantics

A FunctionalScreenArea provides a part of the functionality of the user interface. This means that the screen space that corresponds to the FunctionalScreenArea is primarily used to create this functionality. If only a minor part of the screen space is used for this functionality, the modeler should consider creating more fine-grained ScreenAreas.

One FunctionalScreenArea can provide multiple functionalities. For instance, it can be an image and serve as a link. In this case, `functionality` contains links to all the corresponding UI-Functionalities.

Notation

The notation of a FunctionalScreenArea is a simple box containing the stereotype icons of all owned UIFunctionalities. If a FunctionalScreenArea owns more than one UIFunctionality, the stereotype icons are drawn on top of each other transparently.

The user is encouraged to always use the stereotype icon instead of the default notation for stereotypes (stereotype name in angled brackets).

Presentation Options

A FunctionalScreenArea that has its `isVisible` attribute set to false may also be drawn without its box, so that only its UIFunctionalities are drawn.

Examples

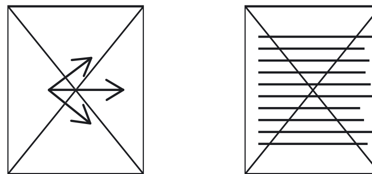


Figure 25. Two FunctionalScreenAreas Owning Two UIFunctionalities Each

Figure 25 shows the notation of two FunctionalScreenAreas that each have two UIFunctionalities. The left one owns the two stereotypes Image and Navigation, and its notation consists of the ScreenArea box plus the stereotype icons of Image and Navigation. The right one owns the two stereotypes Image and Text.

5.4.5 UIFunctionality

A UIFunctionality is a single functionality that the user interface provides to the user.

Description

UIFunctionality is an abstract stereotype of Class[167] that is associated with a FunctionalScreenArea to describe its purpose within the GUI.

[167] Jeckle 2004

Associations

owner : FunctionalScreenArea	The FunctionalScreenArea that the UIFunctionality describes. Subsets class.

Semantics

While the dimensioning, positioning and nesting of ScreenAreas can be seen as the syntax of a screen, UIFunctionalities are part of the semantics. They specify what the purpose of a ScreenArea is.

Notation

As an abstract stereotype, UIFunctionality has no notation.

5.4.6 StaticUIFunctionality

A StaticUIFunctionality displays a screen element without providing behavior or interaction.

Description

StaticUIFunctionality is an abstract subclass of UIFunctionality that describes a static functionality, i.e. a functionality that does not react on user input or other events.

Semantics

A StaticUIFunctionality is a functionality of the user interface whose purpose is to display something without providing interaction. A StaticUIFunctionality can never provide or trigger an interaction. However, it can be influenced by or present the result of an interaction.

Despite the word *static*, instances of StaticUIFunctionality may include time-based media like a video clip, as long as it does not react to user input (which should be modeled using an additional ActivatableUIFunctionality).

Notation

As an abstract stereotype, StaticUIFunctionality has no notation.

5.4.7 ActivatableUIFunctionality

An ActivatableUIFunctionality is a GUI element that can be activated.

Description
ActivatableUIFunctionality is an abstract subclass of UIFunctionality that describes a functionality that has interactive features, i.e. reacts to user input or other events, and can case the display of another ScreenArea.

Semantics
ActivatableUIFunctionalities can include a wide variety of GUI elements, which all have in common the fact that they exhibit behavior. For the semantics of this behavior, please see the descriptions of the GUILayout::References package in section 5.5.

Notation
As an abstract stereotype, ActivatableUIFunctionality itself has no notation of its own.

5.4.8 Form

A Form requests a given set of data from the user.

Description
Form is a concrete subclass of ActivatableUIFunctionality that describes a screen element that request information from the user in order to send it to a processor. The screen element usually contains a set of labeled input elements, one for each data element.

Semantics
A Form provides limited interaction like sending the form and error checking of inputs. The details of these interactions are modeled in separate activity diagrams.

Notation
The common notation of Form is its stereotype icon, which is an abstraction of a designers' sketch of a form. Due to the nature of an abstraction, the icon is not a representation of the actual form's input elements, but rather reflects a general form that consists of multiple form elements. The stereotype icon is drawn in the center of the owning FunctionalScreenArea. As the exact icon contents do not represent actual input elements, the icon can be arbitrarily sized, but should be scaled linearly to almost fill the FunctionalScreenArea's box.

The user is encouraged to always use the stereotype icon instead of the default notation for stereotypes (stereotype name in angled brackets).

Examples

Figure 26 shows the stereotype icon of Form.

Figure 26. Stereotype Icon of Form and its Sketched Origin

Examples of Form instances are web forms created by the `<form>`, `<input>` and `<select>` tags as well as input dialogs of windowed applications.

Rationale

We have decided against modeling all elements of forms as individual elements, because there already are comfortable tools for building forms interactively both for web and for windowed applications. Additionally, the level of detail required to create a complete model a form is too high for this profile, which is aligned to rough layouts. However, if form layout is desired, stereotypes for common form elements (one-line and multi-line input fields, dropdown and list select boxes, radio buttons, checkboxes etc.) can be added using a second profile without much effort. This is possible as multiple profiles can be applied at a time and even defined to depend on each other.[168] In this case, the second profile ("Profile for Form Layout") depends on the Profile for GUILayout.

5.4.9 Link

A Link describes a functionality that is activatable.

Description

Link is a concrete subclass of ActivatableUIFunctionality that is used to mark a ScreenArea that triggers something when activated.

[168] OMG 2003b, 579

Constraints

[1] A Link may only have one target.

```
self.target->size() = 0
```

Semantics

A Link can be activated, for instance by clicking it with the mouse. This may trigger various things, for instance an Activity may be started. Usually the Link's reference is activated when the Link is activated (cf. 5.5.6).

Notation

The common notation of Link is its stereotype icon, which is an abstraction of a designers' sketch of a link. The stereotype icon of a Link is a short arrow and is drawn in the center of the owning FunctionalScreenArea. The icon can be arbitrarily sized, but should be scaled linearly to almost fill the FunctionalScreenArea's box.

The user is encouraged to always use the stereotype icon instead of the default notation for stereotypes (stereotype name in angled brackets).

Examples

The stereotype icon of Link is shown in Figure 27.

Figure 27. Stereotype Icon of Link and its Sketched Origin

Examples of instances of Links are HTML links created using <a href=""> as well as links within windowed applications like a help button.

Rationale

Instances of Links are especially common on web-based GUIs, and when one instance is activated, the page is reloaded as a whole. This can be modeled by creating a LinkReference between the Link and the ScreenArea subclass Screen (cf. 5.5.6).

5.4.10 Navigation

A Navigation describes the functionality of providing the user with efficient means of reaching all major parts of the modeled system.

Description

Navigation is a concrete subclass of ActivatableUIFunctionality that links to the main ScreenAreas of all major parts of the software system under construction using a structured display.

Semantics

Visually, a Navigation is often a structured and sometimes hierarchical list of the names of the major parts of the system or of the classification of commands in a system.

Though similar, Navigation is substantially different from ContainerScreenArea that contains multiple FunctionalScreenAreas that each own a Link. Even when grouped into a ContainerScreenArea, the ensemble of Links bears no semantic meaning about the targets of the Links or their relation to each other. In a Navigation, the linked ScreenAreas cover all parts of the system and are unique within the navigation. In addition, the navigation is considerably more important for the layout of a Screen and thus must be treated with special care.

Notation

The common notation of Navigation is its stereotype icon. The stereotype icon is three arrows that have a common origin and point to different directions. It is drawn in the center of the owning FunctionalScreenArea. The icon can be arbitrarily sized, but should be scaled linearly to almost fill the FunctionalScreenArea's box.

The user is encouraged to always use the stereotype icon instead of the default notation for stereotypes (stereotype name in angled brackets).

Examples

Figure 28 shows the stereotype icon of Navigation.

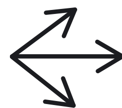


Figure 28. Stereotype Icon of Navigation

Examples of instances of Navigation include the navigation on web pages as well as the menu bar in windowed applications.

5.4.11 Workspace

A Workspace provides complex interactions in order to edit data.

Description

Workspace is a concrete subclass of ActivatableUIFunctionality that describes a GUI element that is used to interactively edit arbitrary data.

Semantics

Workspaces are used to model areas that provide the functionality of creating or editing data. The interactions involved therein can be modeled in a separate activity diagram.

Notation

The common notation of a Workspace is its stereotype icon. The stereotype icon is an abstraction of a pencil and is drawn in the center of the owning FunctionalScreenArea. Workspaces often are quite large compared to the Screen they are contained in. Therefore, the icon can be arbitrarily sized, but should be scaled so the icon does not dominate the whole Screen.

The user is encouraged to always use the stereotype icon instead of the default notation for stereotypes (stereotype name in angled brackets).

Examples

Figure 29 shows the stereotype icon of Workspace.

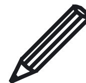


Figure 29. Stereotype Icon of Workspace

Examples of instances of Workspace include the editing area in a word processor or the viewports of a 3D CGI application.

5.4.12 Heading

A Heading displays a short text in a larger font that states the topic of the following screen elements.

Description

Heading is a concrete subclass of StaticUIFunctionality that displays a short text in a larger or otherwise distinct font to declare the topic of the following Text or screen elements.

Semantics

A Heading is a short, important text that gives the user an impression of what will follow on the Screen. Due to its importance, it should be eye-catching and therefore visually distinct from the surrounding text. Usually, a larger or bolder font type is used.

Notation

The common notation of Heading is its stereotype icon, which is an abstraction of a designers' sketch of a heading. The stereotype icon a sinuous line and is drawn in the center of the owning FunctionalScreenArea. The icon should be scaled to completely fill the FunctionalScreenArea's box.

The user is encouraged to always use the stereotype icon instead of the default notation for stereotypes (stereotype name in angled brackets).

Examples

Figure 30 shows the stereotype icon of Heading.

Figure 30. Stereotype Icon of Heading and its Sketched Origin

Examples of instances of Heading include news headlines on web pages or the window title of a windowed application.

5.4.13 Image

An Image is a screen element that displays a picture.

Description
Image is a concrete subclass of StaticUIFunctionality that displays a picture.

Semantics
Images are not limited to stills; its instances can also be video clips or non-interactive applications.

Notation
The common notation of Image is its stereotype icon, which is an abstraction of a designers' sketch of an image. The stereotype icon is a box crossed with two diagonal lines and is drawn in the center of the owning FunctionalScreenArea. The icon should be scaled to completely fill the FunctionalScreenArea's box.

The user is encouraged to always use the stereotype icon instead of the default notation for stereotypes (stereotype name in angled brackets).

Examples
Figure 31 shows the stereotype icon of Image.

Figure 31. Stereotype Icon of Image and its Sketched Origin

Images can be graphics, photographs, diagrams, or even videos. Examples of instances of Heading include images embedded in HTML using the `<img>` tag or icons within a windowed application.

Rationale
In regard to GUI layout, video clips and stills are the same; therefore, a video clip is also modeled as a FunctionalScreenArea with an associated Image stereotype.

5.4.14 Logo

A Logo is a textual or graphical screen element that is the key element of the corporate design of the client who ordered the system.

Description

Logo is a concrete subclass of StaticUIFunctionality that describes a GUI element that displays the main element of the client's corporate identity.

Semantics

The semantics of Logo are somewhat different from the semantics of Image. A Logo is not necessarily graphical; it can also be textual. In addition, it has a special role within the page layout and is more important than any other single image.

Notation

The common notation of Logo is its stereotype icon, which is an abstraction of a designers' sketch of a logo. The stereotype icon is an ellipse and is drawn in the center of the owning FunctionalScreenArea. The icon should be scaled to fill the FunctionalScreenArea's box completely.

The user is encouraged to always use the stereotype icon instead of the default notation for stereotypes (stereotype name in angled brackets).

Examples

Figure 32 shows the stereotype icon of Logo.

Figure 32. Stereotype Icon of Logo and its Sketched Origin

Examples of instances of Logo include a company logo on web pages or in the about box of windowed applications.

5.4.15 Text

A Text is a continuous text that includes several lines and possibly paragraphs.

Description

Text is a concrete subclass of StaticUIFunctionality that displays continuous text. It consists of multiple lines and paragraphs and is set in an easy-to-read font size.

Semantics

Text does not convey anything about the formatting of the displayed text. Actually, the text can have any alignment and may be arbitrarily structured, for instance with paragraphs, lists and bullets.

Notation

The common notation of Text is its stereotype icon, which is an abstraction of a designers' sketch of a paragraph of text. The stereotype icon is several horizontal lines. They may be drawn in bold to simulate line-height; however, this bears no semantic meaning. If they are drawn in bold, their color should be adjusted so their grey-value does not dominate the screen.

The stereotype's icon is drawn in the center of the owning FunctionalScreenArea. The icon should be scaled horizontally to completely fill the FunctionalScreenArea's box. In vertical dimension, new lines should be added instead of keeping the number of lines and increasing the space in between.

The user is encouraged to always use the stereotype icon instead of the default notation for stereotypes (stereotype name in angled brackets).

Examples

Figure 33 shows the stereotype icon of Text.

Figure 33. Stereotype Icon of Text and its Sketched Origin

Examples of instances of Text include articles on web pages or help texts windowed applications.

5.5 Package GUILayout::References

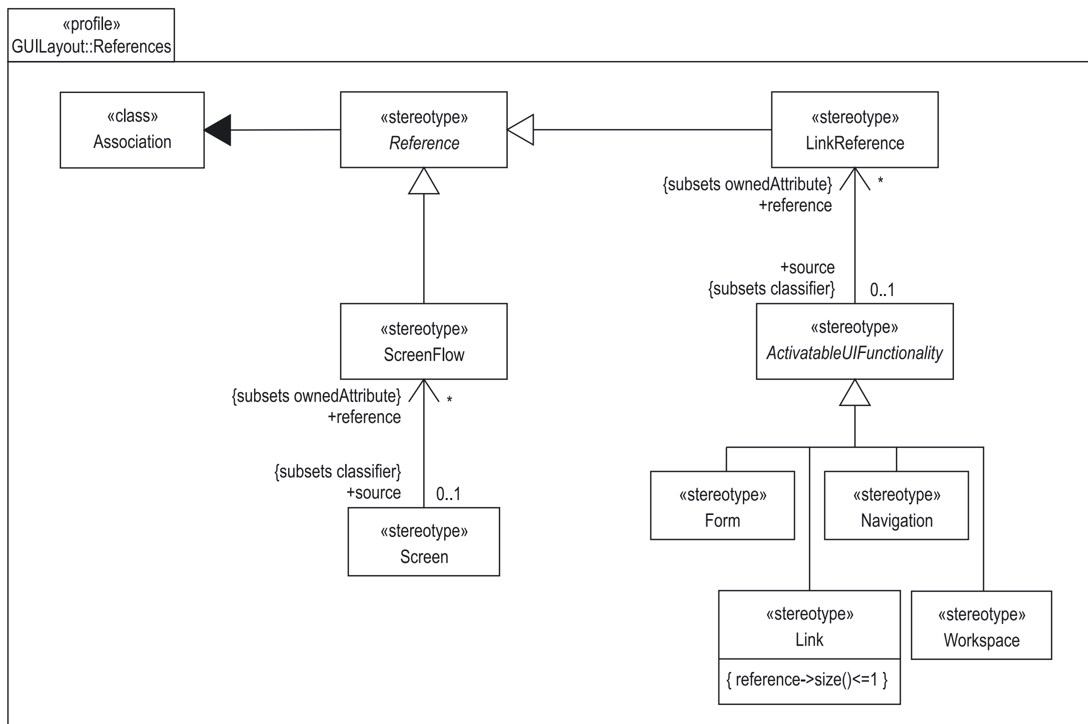


Figure 34. Architecture of the Package GUILayout::References

The References package contains mechanisms to create activatable transitions between GUI elements. The base element is Reference, which is an abstract stereotype of Association that is restricted to be binary and navigable in one direction. It is specialized by two concrete stereotypes, LinkReference and ScreenFlow. LinkReference can be used to create a transition between an ActivatableUIFunctionality and a ScreenArea that is activated when the associated ActivatableUIFunctionality triggers it. ScreenFlow is used to model the flow of Screens, i.e. Screens that follow each other in a sequence of actions.

5.5.1 Reference

A Reference is a link between a GUI element and a ScreenArea.

Description

Reference is an abstract stereotype of Association that is restricted to be binary and navigable in one direction. The `target` is always a ScreenArea.

Constraints

[1] A Reference must have exactly two memberEnds.

```
self.memberEnd->size() = 2
```

[2] A Reference must have exactly one ownedEnd.

```
self.ownedEnd->size() = 1
```

[3] The owned end must be a ScreenArea.

```
self.ownedEnd->forAll(c | c.type.oclIsKindOf(ScreenArea))
```

[4] The memberEnd that is not the ownedEnd must be the owning Classifier.

```
(self.memberEnd-self.ownedEnd)->includes(self.classifer)
```

Operations

[1] The query target() gives the ScreenArea at the owned end.

```
Reference::target(): ScreenArea;
target = self.ownedEnd->first()
```

[2] The query source() gives the memberEnd that is not the target.

```
Reference::source(): Classifier;
source = memberEnd->excludes(self.target())->asOrderedSet()->first()
```

Semantics

A Reference is an abstraction of a link between a `source` GUI element and a `target` ScreenArea that can be activated during runtime.

The `target` ScreenArea is displayed when this Reference is activated i.e. it is displayed. If the `target` ScreenArea is a subclass of another ScreenArea, an instance of it is displayed at the same size and position as its superclass in the Screen instance that owns the more general ScreenArea. If the owning Screen instance is not currently displayed, the currently displayed Screen instance is hidden and the owning Screen instance is displayed instead with the target ScreenArea in place.

If the `target` ScreenArea is a Screen, the currently displayed Screen instance is replaced by an instance of the `target` Screen as a whole.

Notation

The notation of a Reference is the same as for an association.

5.5.2 LinkReference

Description

LinkReference is concrete subclass of Reference that is restricted to linking from an ActivatableUIFunctionality to a ScreenArea.

Associations

+source : ActivatableUIFunctionality [0..1]	The source of the LinkReference that may activate it. Subsets classifier.

Constraints

[1] One of the memberEnds must be an ActivatableUIFunctionality, the other must be a ScreenArea.

```
self.memberEnd->one(c | c.type.oclIsKindOf(ActivatableUIFunctionality))
 and self.memberEnd->one(c | c.type.oclIsKindOf(ScreenArea))
```

[2] The memberEnd that is not the ownedEnd must be an ActivatableUIFunctionality.

```
(self.memberEnd-self.ownedEnd)->oclIsKindOf(ActivatableUIFunctionality)
```

Semantics

A LinkReference is an activatable link between a functionality of the GUI that can be activated and a ScreenArea that is displayed as the result of the owning element's activation.

Notation

The notation of a LinkReference is the same as for a Reference.

Rationale

The reason why LinkReferences do not generally target Screens is that they are employed in windowed applications as well, where they may only reload a small part of a Screen; even on web GUIs it is possible to reload only part of a Screen, for instance using frames or JavaScript.

A problem that would arise when LinkReferences targeted ScreenAreas instead of Screens is where the target ScreenArea would be displayed. If the target is a Screen, there is no question about where it is displayed: A Screen always contains all elements of the user interface that are displayed at a time and thus replaces all others when dis. But an anonymous ScreenArea *per se* has no location, and the only way of locating a ScreenArea within a Screen is by nesting it in a ContainerScreenArea. However, by nesting it, we prescribe the contents of the ContainerScreenArea and there is no way for it to contain a ScreenArea with a different layout. It might

be necessary, though, for a ScreenArea in the model to contain several ScreenArea instances during the runtime of the application. The profile must be able to model situations like these.

We solve this using ScreenArea inheritance. In this example, the ContainerScreenArea to contain the desired ScreenArea is made abstract. Then all ScreenAreas that need to be displayed in its place are subclassed from it, inheriting its position and size attributes.

5.5.3 ScreenFlow

A ScreenFlow models the sequence of two Screens that follow each other in time.

Description
A ScreenFlow is a concrete subclass of Reference that has Screens as its source and target.

Associations

+source : Screen [0..1]	The source of the ScreenFlow. Subsets classifier.

Constraints

[1] All memberEnds must be Screens.

```
self.memberEnd->forAll(c | c.type.oclIsKindOf(Screen))
```

[2] The owned end must be a Screen.

```
self.ownedEnd->forAll(c | c.type.oclIsKindOf(Screen))
```

Semantics
ScreenFlows are used to model the order in which Screens are displayed during a sequence of actions by the user. The `source` Screen is replaced by the `target` Screen of the Reference. Often, the transitions between the two Screens associated via a ScreenFlow are based on the activation of an ActivatableUIFunctionality. However, for the ScreenFlow it is of no importance how the transition has been initiated.

Notation
The notation of a ScreenFlow is the same as for a Reference.

5.5.4 Screen

Description

A Screen can be followed by another Screen via a ScreenFlow.

Associations

+reference : ScreenFlow [*]	The ScreenFlows that point to following Screens. Subsets ownedAttribute.

Semantics

Reference does not need to be specified if the order of Screens is unimportant.

5.5.5 ActivatableUIFunctionality

An ActivatableUIFunctionality is an abstract subclass of UIFunctionality that can be the `source` of a LinkReference.

Associations

+reference : LinkReference [*]	The LinkReferences that may be activated as a result from the activation of the ActivatableUIFunctionality. Subsets ownedAttribute.

Operations

[1] The query targets() gives all targets of all references.

```
ActivatableUIFunctionality::targets(): Set(ScreenArea);
targets = self.references->collectNested(r | r->target())
```

Semantics

When an ActivatableUIFunctionality is activated, one or more of its `reference`s may be activated. The details on how and when a `reference` is activated are up to the concrete subclasses of ActivatableUIFunctionality.

An ActivatableUIFunctionality can trigger an interaction that is specified in an activity diagram, or provide interaction itself, which again is specified in an activity diagram.

A `reference` can be left unspecified if it is unimportant or is to be specified later.

5.5.6 Form

Semantics

A Form can activate a Reference after it has been sent. The `reference` association inherited from ActivatableUIFunctionality contains any References that may be activated after the form has been sent. This includes, for example, confirmations, error pages etc. Which `reference` is activated in which case can be modeled by placing constraints on the Reference.

5.5.7 Link

A Link describes a functionality that provides a navigable pointer that causes the system to display another ScreenArea.

Description

A Link is a concrete subclass of ActivatableUIFunctionality that has zero or one `references`. It provides a one-way navigable pointer to another ScreenArea.

Constraints

[1] A Link must have at most one reference.

```
self.reference->size() <= 1
```

Semantics

A Link established a rigid connection between its owning ScreenArea and its target: When the Link is activated, for instance by clicking it with the mouse, the target is activated as well.

5.5.8 Navigation

Semantics

The `reference` association inherited from ActivatableUIFunctionality contains LinkReferences to the most important ScreenAreas of the software system. Usually, they are all Screens, but they do not need to be.

5.5.9 Workspace

Semantics

The `reference` association inherited from ActivatableUIFunctionality contains any LinkReferences that are activated as a result of the interaction within the Workspace. This might include

context-sensitive help or additional sub windows, for instance. Which `reference` is activated in which case can be modeled by placing constraints on the Reference.

5.6 GUI Layout Diagram

A class diagram in which all classes are stereotypes defined in this profile is called a GUI Layout Diagram. One such diagram can contain multiple Screens and ScreenAreas, each one containing more ScreenAreas. This means that there may be ScreenAreas outside of Screens subclassing ScreenAreas which exist in a Screen that has previously been modeled. This relationship is expressed by making the superclass that is specified in a Screen abstract and subclassing it outside the Screen by a concrete ScreenArea.

Links and Navigations can be graphically associated with the Screens they link to using association arrows.

Examples of GUI Layout Diagrams can be found in section 5.9.

5.7 Navigational Diagram

A class diagram which consists of Screens and ScreenAreas that are interrelated using References is called a Navigational Diagram. The detailed contents of ScreenAreas may be omitted for clarity. The most important elements of Navigational Diagrams are References, which model the paths that can be taken to traverse the system. This includes LinkReferences pointing from ActivatableUIFunctionalities to ScreenAreas, as well as ScreenFlows modeling a sequence of Screens.

Examples of Navigational Diagrams can be found in 5.9.

5.8 Links to Existing Diagrams

As stated in the goals section on page 43, interactivity and behavior of user interfaces is modeled using UML behavioral diagrams, especially the activity diagram. To model interaction that involves GUI elements, it is necessary to refer to these elements from the behavioral diagram. In this section, we will outline how this could be accomplished. However, this is not completely elaborated, as it requires additional profiles to be created and would thus be beyond the scope of this thesis. The profiles required are needed to stereotype the links between ScreenAreas and conventional UML diagram elements.

Linking GUI layout to other diagrams and UML in general is beneficial, as it creates a more complete model of the system and enables GUI layout to be included in software engineering.

5.8.1 Use Case Diagrams

Every ScreenArea serves a certain purpose. This purpose is expressed by choosing between the ContainerScreenArea and FunctionalScreenArea subclasses and by associating one or more UIFunctionalities with a FunctionalScreenArea. However, this purpose is derived from a higher goal that the whole Screen is to achieve. As each Screen is part of the user interface, the goals that have to be achieved are to provide the user with the means to fulfill his tasks, which are modeled in use case diagrams. Therefore, to associate a Screen or ScreenArea with the purpose it fulfills, it should be associated with the appropriate UseCase.

Every UseCase can be associated with its subject based on the existing UML metamodel [169] to establish the relationship between UseCase and realizing Classifier semantically. As ScreenArea is a stereotype of Class which is derived from Classifier, this mechanism can be utilized to associate a ScreenArea with a UseCase without changing the metamodel. This association is drawn as a normal association arrow from the UseCase to the Classifier, in this case the ScreenArea. This is in conformance with the Rational Unified Process, which assumes each UseCase has a user interface.[170]

There are three possible ways of modeling the relationship between a ScreenArea and a UseCase:

First, a use case diagram can be enriched by isolated representation of ScreenAreas that are identified using their unique name (path name if necessary). This way, an overview can be given how the realization of UseCases is spread over the various Screens of the GUI (Figure 35).

Second, a GUI Layout Diagram can be enriched by isolated representations of UseCases to give an overview of which UseCases are realized by a particular Screen. Figure 36 shows a simple example of this method, and section 5.9 contains a more extensive example.

Third, UseCases and ScreenAreas can be arranged in two columns, with the UseCases in the left column and the ScreenAreas in the right. Then a mapping of UseCases to ScreenAreas can be performed by inserting the associations between them. This can also be used to create a list of ScreenAreas that are necessary to realize a UseCase before creating the GUI layout. After the

[169] OMG 2003b, 519
[170] Jacobson et al. 1998, 142

necessary ScreenAreas have been identified, they can be grouped into Screens and arranged within them (Figure 37), similar to the tabular use cases approach described in section 3.4.2.

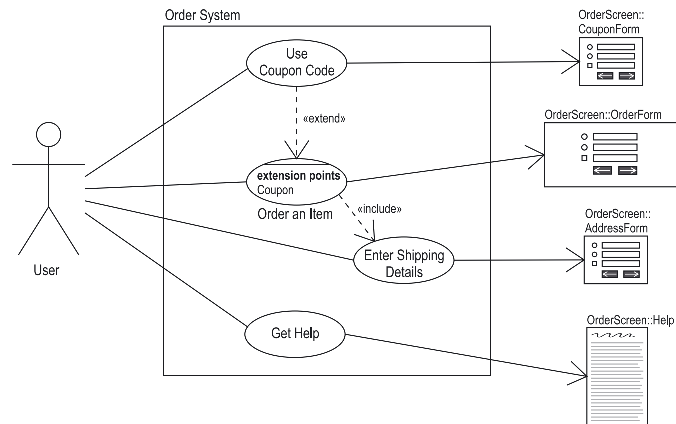


Figure 35. Use Case Diagram Enriched with Isolated ScreenAreas

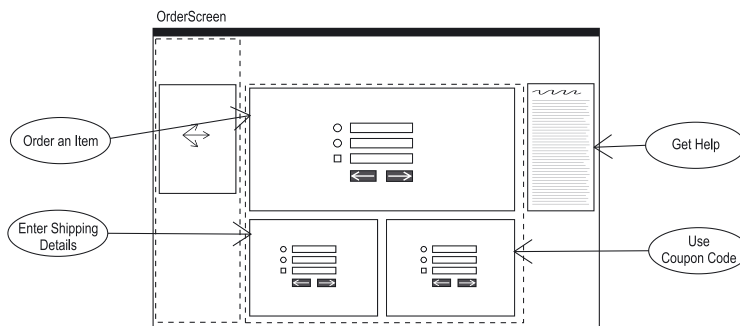


Figure 36. GUI Layout Diagram Enriched with Isolated UseCases

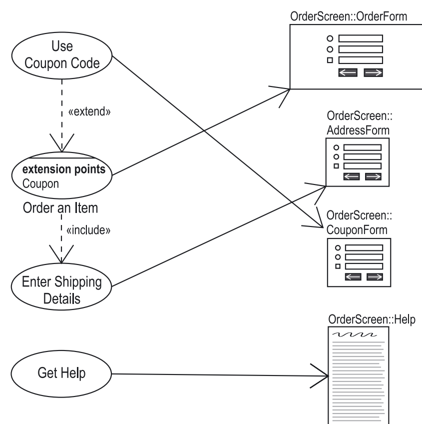


Figure 37. Two-Column View of UseCases and ScreenAreas

5.8.2 Activity Diagrams

Activity Diagrams are the new standard in UML 2.0 of modeling detailed behavior. An Activity can be partitioned using an ActivityPartition to reflect common behavior or responsibility for Actions within the Activity. This is widely known as 'swimlane notation' and can be used to mark the responsibility of the GUI. All elements that 'swim' in the Presentation swimlane are presented by the GUI. These elements are ObjectNodes that each directly correspond to a Screen or a ScreenArea with the same name.

This is possible because ObjectNode "is an activity node that indicates an instance of a particular classifier, possibly in a particular state, may be available at a particular point in the activity"[171]. Usually, ObjectNodes are used to model a flow of information from one activity node to another. They represent a set of data that is produced by the source node and then used by the target node. In our case, this set of data can be thought of as the information presented at the user interface. As the target node is always an Action in the user partition, the user uses the information by receiving and intellectually processing them. Presenting the data on the user interface is a necessary step for the user to receive the data.

As ScreenArea is a Stereotype of Class, which is a subclass of Classifier, it is therefore possible to use ObjectNodes to indicate the availability of an instance of a ScreenArea containing context-specific data. By grouping the ObjectNodes that actually represent ScreenArea instances into the presentation ActivityPartition, they are made distinct from ObjectNodes performing other functions

FunctionalScreenAreas that own an ActivatableUIFunctionality can trigger activities or have a complex behavior of their own.

Class is a BehavioredClassifier, i.e. it is a Classifier that can have a Behavior. "Behavior is a specification of how its context classifier changes state over time."[172] As ScreenArea is a stereotype of Class, every ScreenArea is also a BehavioredClassifier and thus can have a Behavior. The commonly used subclass of Behavior is an Activity. In short: It is possible to model the behavior of a ScreenArea using plain UML.

[171] OMG 2003b, 349
[172] OMG 2003b, 379

5.9 Examples

Figure 38 shows a GUI Layout Diagram including ContainerScreenAreas, FunctionalScreenAreas and UIFunctionalities. The diagram shows the model of a real website, the home page of the department of Digital Media at the University of Applied Sciences, Furtwangen. We have created the model by abstracting from the existing design to illustrate the similarity between a model of GUI Layout at M1 and its instance at M0.

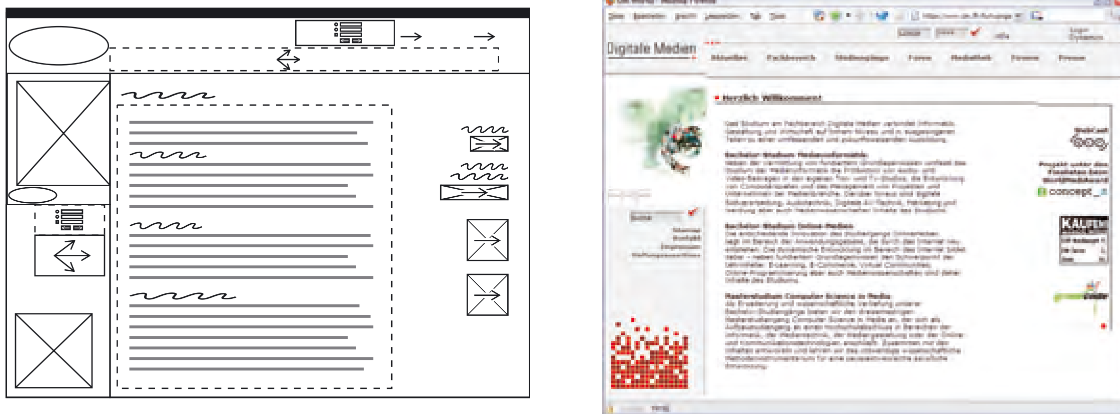


Figure 38. A GUI Layout Diagram and the Corresponding Web Page

Now we will model parts of a fictive online bookstore application. Our examples start after use case modeling has been finished; as a result, we have several UseCases that need an interface to the user.

Figure 39 shows the homepage of the bookstore, named Welcome, with all ScreenAreas fully modeled. The Screen is marked by a bold top line and is divided into four parts: Navigation and company logo on the left, another navigation at the top, two ScreenAreas on the right and a big main content ScreenArea at the center of the Screen. Some ScreenAreas are associated with a UseCase, for instance the Navigation ScreenArea on the left is associated with the UseCase “Browse Items”. It is therefore a special kind of Navigation, whose links have been selected to allow browsing. This Screen might be the initial artifact a designer created. It reflects the most important layout choices and gives examples on how to fill general-purpose ScreenAreas like the ContentMain ScreenArea. However, this diagram is very specific in its contents, and the large amount of ScreenAreas create make it look a little cluttered even when the final Screen might not.

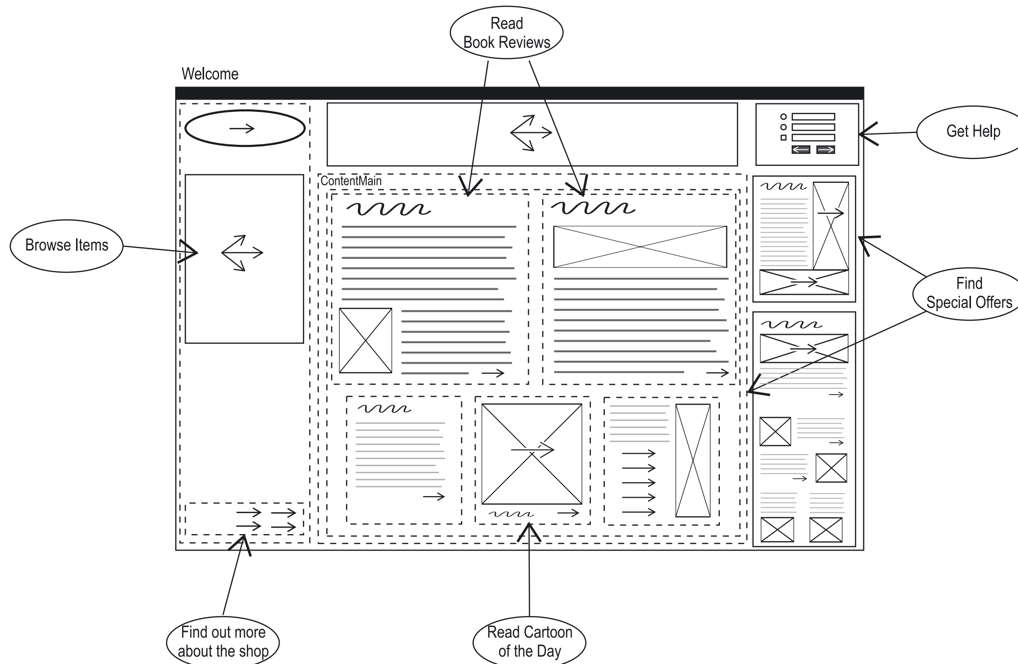


Figure 39. Example of a GUI Layout Diagram with Associated UseCases

To reduce the complexity of the diagram, we have extracted most of the contents from the Screen (Figure 40). The layout of the Screen is clearly visible in this diagram, and we can see there are three abstract ScreenAreas in it: ContentMain, which we have seen in the other diagram as well, FeaturedItem and Recommendations. They are all empty now, which means they need to be specified somewhere else. The concrete ScreenAreas subclassed from the two abstract ScreenAreas named FeaturedItem and Recommendations on the right are shown on Figure 41.

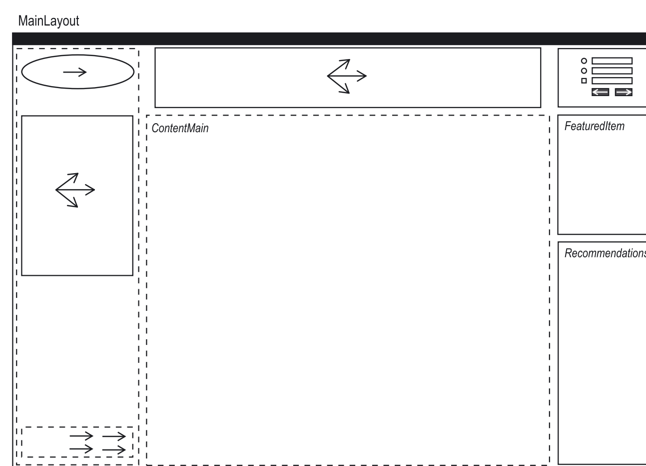


Figure 40. Example of a Screen with Abstract ScreenAreas Showing the Base Layout

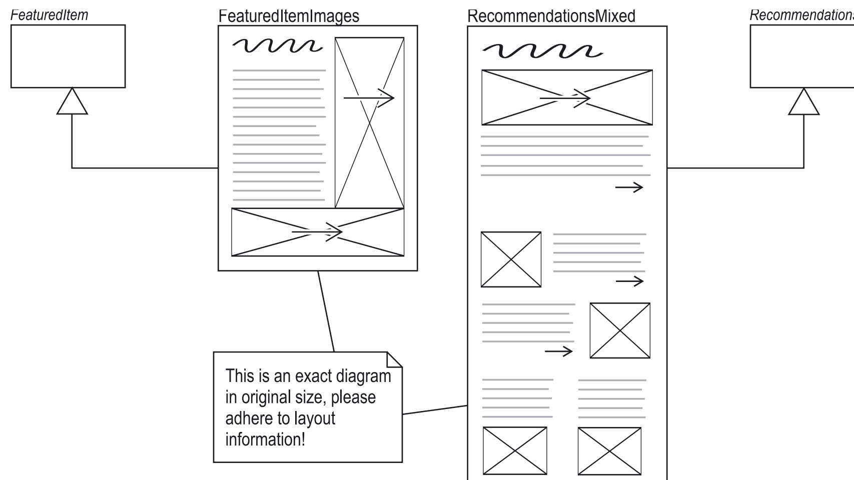


Figure 41. Example of Precisely Sized ScreenAreas

The two concrete ScreenAreas of Figure 41 inherit the two abstract ScreenAreas FeaturedItem and Recommendations. This means that their instances will be inserted at the same size and position as their respective superclasses in the Screen that own their respective superclasses. They have both been magnified to show their detailed layout. If they were put back in the Screen, they and their contents would be linearly scaled down to fit into the gap. The reason why they have been magnified in this view is given in the note; their model is exactly the same size as their instances should be.

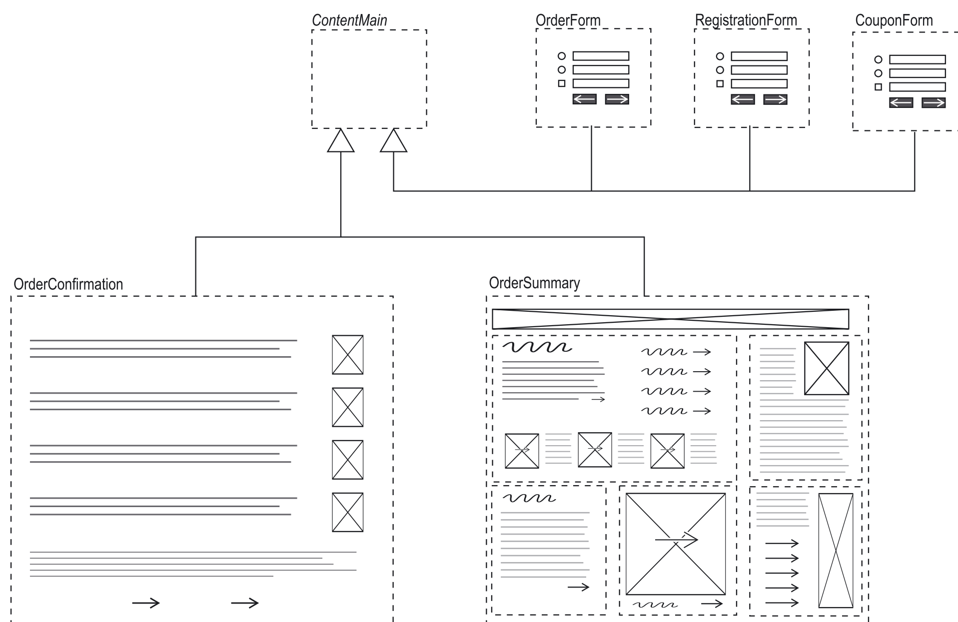


Figure 42. Example of ScreenArea Inheritance

Figure 42 shows the inheritance of the ContentMain ScreenArea. As we can see, it has several subclasses that have completely different layouts. Three of them, OrderForm, Registration-Form and CouponForm have not been modeled in detail, as they only contain more or less big forms. The other two, OrderConfirmation and OrderSummary are modeled in detail and are examples of inherited ScreenAreas that have a complex layout.

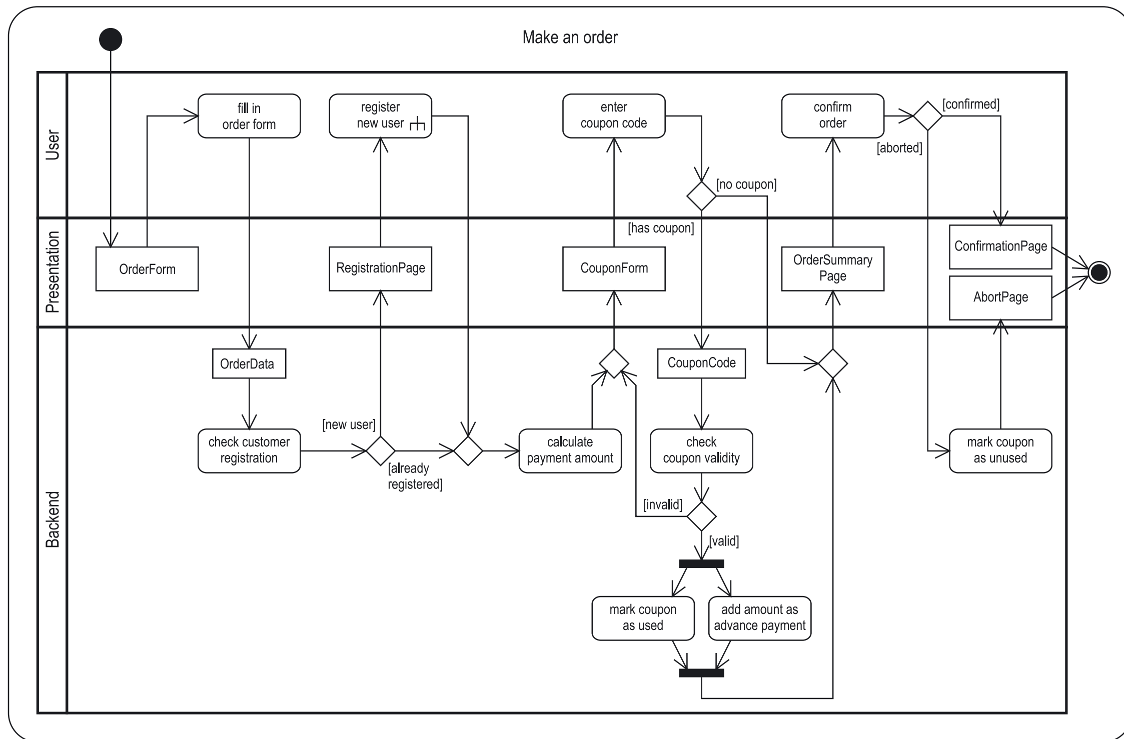


Figure 43. Example of an Activity with an ActivityPartition for Presentation

Figure 43 shows how Screens and ScreenAreas can be used with an Activity Diagram. The example is a model of the ordering process of our bookstore application. It is partitioned using three ActivityPartitions into backend, presentation and user partitions. All elements in the backend partition are executed on the server. Every time output is created using a web page, an ObjectNode representing an instance of ScreenArea is activated in the presentation partition. The user partition contains all Actions that have to be executed by the user, e.g. form input. The resulting form data are sent back to the backend partition using an ObjectNode containing the data entered in the form. The ScreenAreas represented by the ObjectNodes in this diagram have been modeled including their layout on the previous pages, creating a link between behavioral and structural modeling.

The following three diagrams illustrate our ideas how the profile could be used to help designers in their work by supporting some of the artifacts described in section 3.2.3.

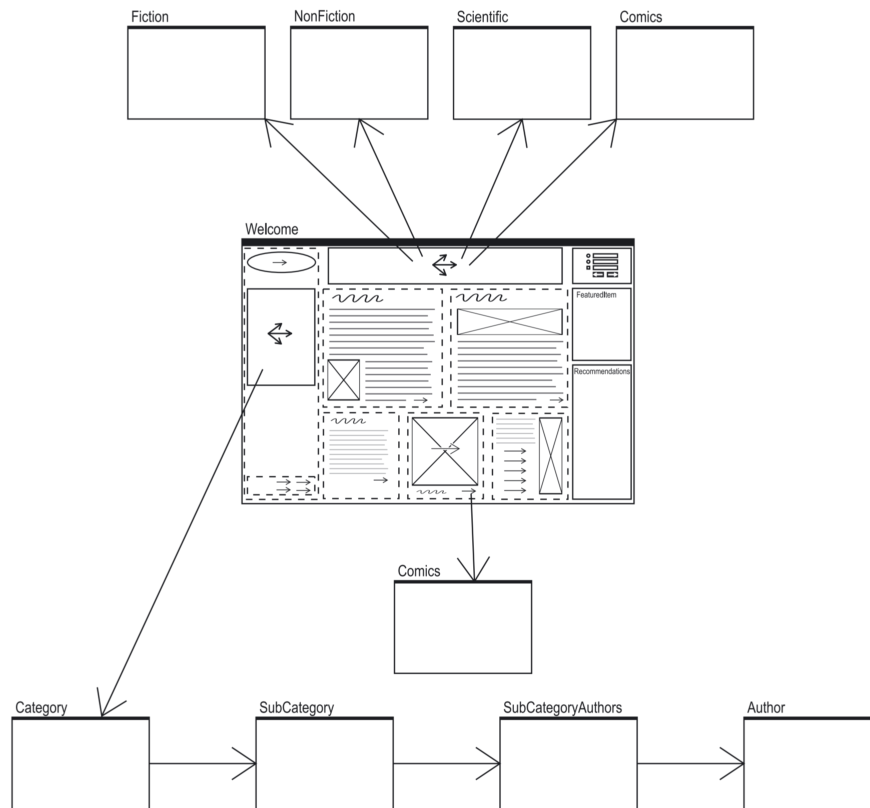


Figure 44. Example of a Navigational Diagram, LinkReferences and ScreenFlows

Figure 44 shows some Links within the bookstore application. The Navigation on top of the page has LinkReferences to the four main areas of the site, represented by the four Screens Fiction, NonFiction, Scientific and Comics. The last one, Comics, is also referred to from the daily Comic at the bottom of the page. The Navigation on the left offers four levels of hierarchy down to a Screen containing all books of an individual author. The deeper levels of the navigation hierarchy have been modeled using ScreenFlows.

Figure 45 is a storyboard of an order based on a recommendation. In every Screen, one or two ActivatableUIFunctionalities own a LinkReference to the Screen that is displayed when the functionality is activated. This storyboard shows the sequence of ScreenAreas when a user follows a link in a recommendation list on the front page and decides to order the item.

Figure 46 illustrates the whole site's structure and the most important links between the Screens using ScreenFlows, creating a sitemap. It is obvious from the sitemap that the whole site is organized around the Screen named Book, in order to guide users to the Order Screen. If this can be seen from the diagram, it is to be considered useful and has fulfilled one of its purposes.

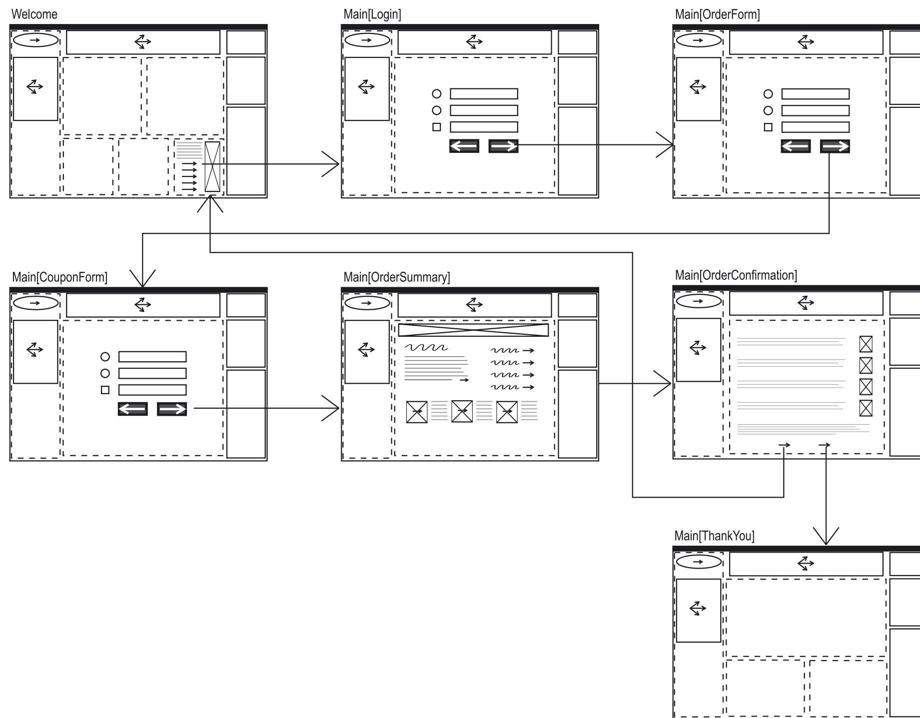


Figure 45. Example of a Storyboard Based on a Navigational Diagram and LinkReferences

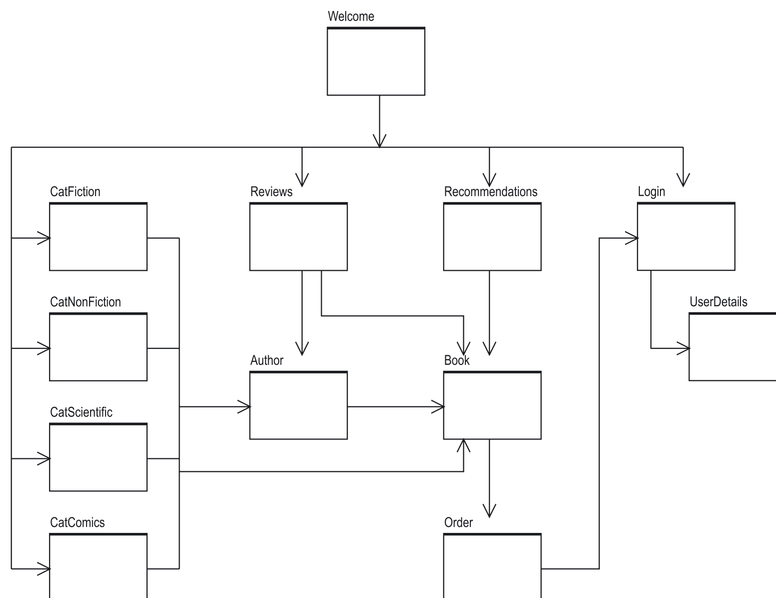


Figure 46. Example of a Sitemap Based on a Navigational Diagram and ScreenFlows

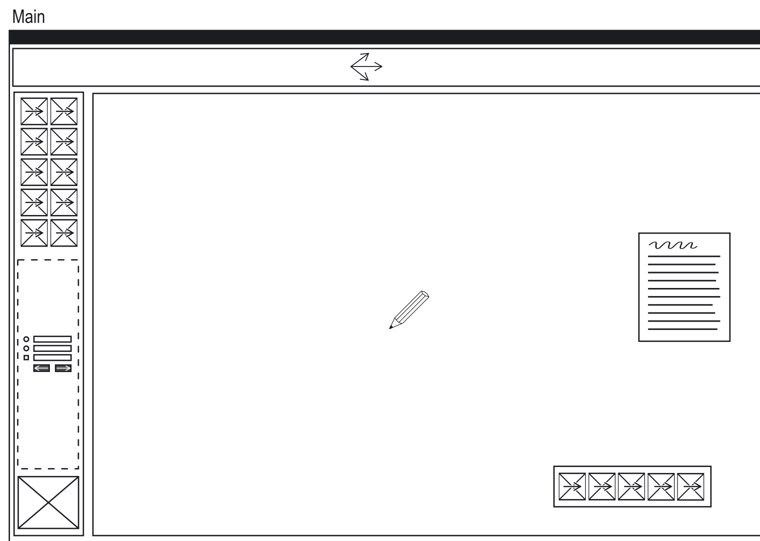


Figure 47. Example of a GUI Layout Diagram of an Application Window

The last example in Figure 47 breaks with the bookstore example. It shows the model of a Screen of a windowed application like a graphics editing application. The restrictions imposed by the widgets toolkits are reflected in this Screen; it has few different elements. The large Workspace is also typical for windowed applications. The two ContainerScreenAreas on the right are floating windows.

Chapter 6

Prototype

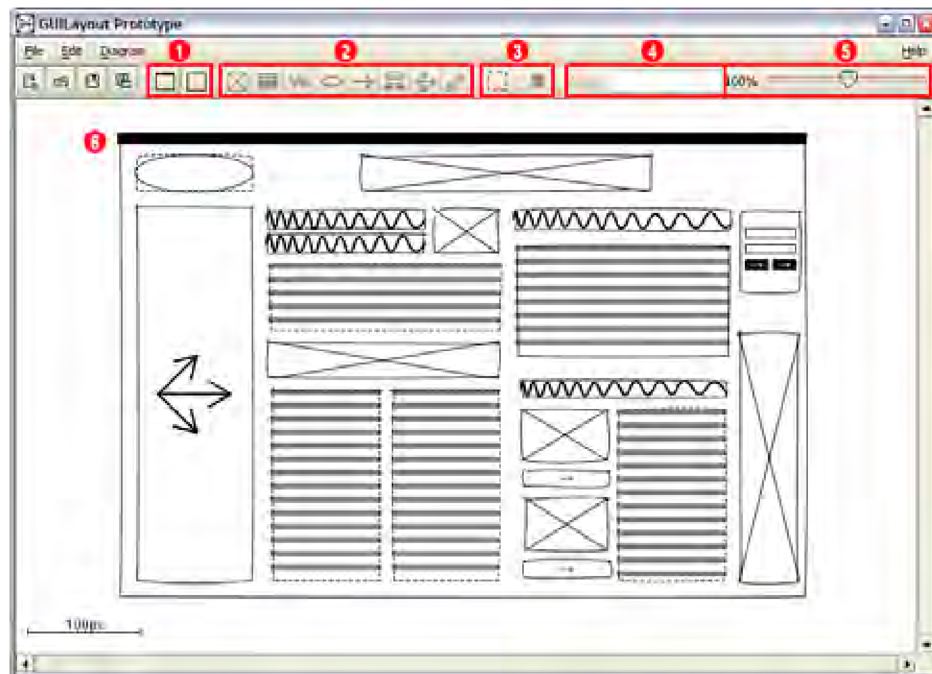


Figure 48. Screenshot of the Prototype with Annotations

1 Create new Screens and ScreenAreas

2 Toggle UIFunctionalities of selected ScreenArea

3 Toggle visible border or delete selected ScreenArea

4 Set the name of the selected ScreenArea

5 Set the zoom level of the whole diagram

6 Diagram view

We have created a prototype to demonstrate the use of the main concepts introduced in this thesis. Figure 48 shows its main screen along with some annotations on the most important functionalities.

6.1 Requirements and Installation

The prototype requires the Sun Java Runtime Environment (JRE) 1.4 to be installed on the system to run. It is also provided for Windows and Linux machines in the `\JRE1.4` directory of the disc, and the latest version can always be obtained from `http://java.sun.com/j2se/downloads/index.html`. On the CD, there is version 1.4 of the Sun JRE as a Windows installer (`\JRE1.4\j2re-1_4_2_04-windows-i586-p.exe`) and as a self-extracting RPM file for Linux (`/JRE1.4/j2re-1_4_2_04-linux-i586-rpm.bin`). To install, run the file appropriate to your operating system and follow the instructions that appear. On Mac OS X, the Apple JRE 1.4 is preinstalled and no additional installation is needed.

After the JRE has been installed, the prototype can be started using the program `\prototype\GUILayout.exe` on Windows computers or by running `java -jar GUILayout.jar` from a command prompt in the `\prototype` directory.

6.2 Usage Instructions

As stated in the requirements section, the program can be used like a drawing application. In particular, it uses a document and a drawing metaphor to build basic GUI layouts. Of course, diagrams can be saved and loaded; in the directory \prototype on the CD, there are some example diagram files. With a new diagram (File/New), Screens and ScreenAreas can be inserted by clicking on the appropriate toolbar button (see Figure 48) or menu item, then clicking at the position of one corner of the desired ScreenArea and dragging to the opposite corner until the ScreenArea has the desired right size. Exact layouts are supported by displaying the ScreenArea pixel size while creating or resizing it. Position and size of ScreenAreas can be tuned by dragging the ScreenArea to a new position and by dragging its borders to a new size. Dragging is supported by automatically docking the ScreenArea to adjacent ScreenAreas in order to align them more easily. This behavior can be disabled temporarily by holding down the Shift key while dragging. ScreenAreas can be nudged by single pixels using the arrow keys, or by ten pixels when additionally holding down the Ctrl key.

When a ScreenArea is dragged on to another ScreenArea, the latter is automatically converted into a ContainerScreenArea if possible and used as a container for the former. Accordingly,

newly inserted ScreenAreas whose top left corner lies inside a Screen or another empty ScreenArea are automatically nested inside this ScreenArea. Nested ScreenAreas are subordinate to their container when the latter is moved, copied or deleted.

When a ScreenArea is selected by clicking it once, its outline is drawn in blue color. Now, several actions can be applied to this ScreenArea (no. 2-4 in Figure 48). If the ScreenArea is empty, i.e. if it can be converted into a FunctionalScreenArea, UIFunctionalities can be added to and removed from it by the toggle buttons in the tool bar. Its visual outline can be toggled by the next button, it can be deleted from the diagram, and its name can be set by entering it in the following text field.

Figure 49 shows how a layout evolves from an empty screen to a Photoshop template of the full screen within few minutes.

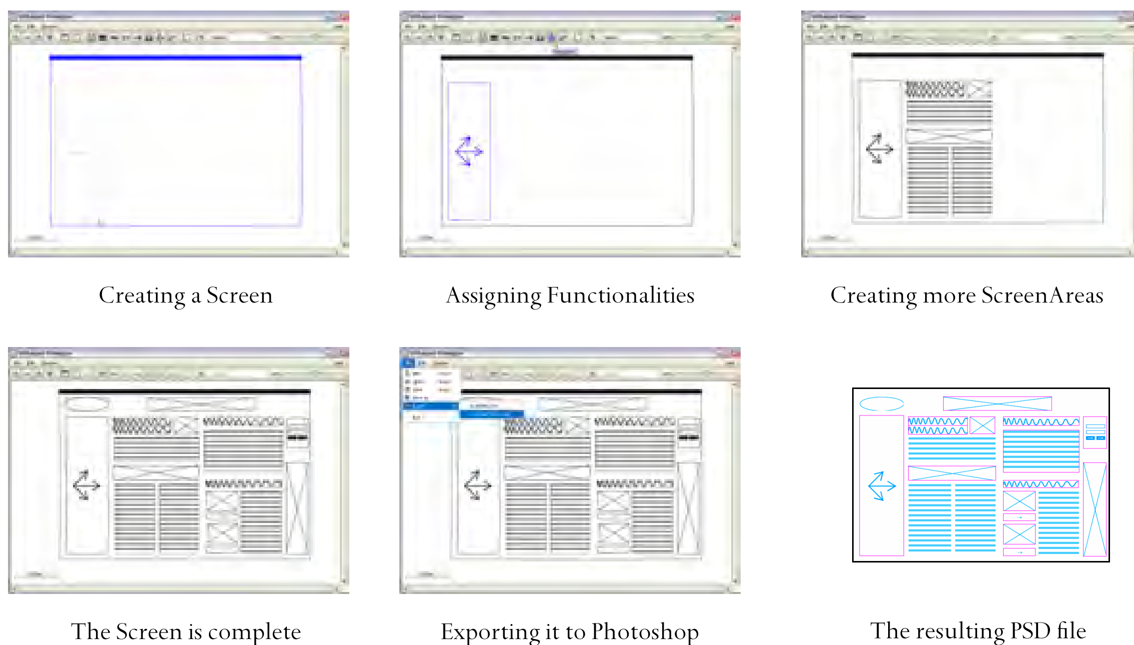


Figure 49. Building a Screen Layout within Five Minutes

The last control on the tool bar is the zoom slider, which controls the zoom level of the while diagram. Zoom reaches from 25% to 400%, making it possible to use the application both for an overview of multiple screens and for layout details on a part of one screen. The current scale can be seen in percent next to the slider and, more concretely, in a scale that is displayed at the lower left corner of the diagram view and always shows the displayed size of 100 pixels at the current zoom level.

Often, several ScreenAreas in a Screen look the same; therefore, we have added copy and paste to the application. The currently selected ScreenArea and its contents can be copied, cut and pasted using the Edit menu or the keyboard shortcuts Ctrl-C, Ctrl-X and Ctrl-V, respectively.

Screens can be exported into XHTML/CSS and Adobe Photoshop formats using File/Export. Before exporting, please select a Screen in the diagram, as because of logical reasons only Screens can be exported. The XHTML output inserts dummy text as headlines and texts, and uses the stereotype icons for the rest. This output can be used to create a semi-interactive HTML prototype of a web application quickly.

Generated Photoshop files contain the outlines of all ScreenAreas that have a visible border. Outlines and stereotype icons are drawn in signal colors to make them clearly distinguishable from any content that may lie beneath them later in the process: outlines are magenta, and stereotype icons are green. This output can be used to switch to the creation of mockup screens directly from the GUI layout diagram, reusing the results of this stage.

6.3 Comments

The whole application has been designed for professionals who use keyboard shortcuts a lot. Keyboard shortcuts can improve efficiency for trained users, which is crucial for tools that should produce results very quickly (see Requirements, section 4.1). The menu and most functions can be accessed using mnemonics or accelerator keys, especially functionalities can be toggled using the numeric keys from 1 to 8.

The look of the diagram is similar to sketches, because not only the stereotype icons have been designed according to sketches, but also because a specific Java Swing Look and Feel is applied to the diagram elements making them look like sketches. This amplifies the unfinished nature of the layout and should make designers more willing to use the program.

Photoshop export is not perfect, as all image data is stored in a single layer. For a production level application, the outlines would have to be drawn on their own layer of the Photoshop file. In addition, output file size is quite large with the prototype, as image data is written in uncompressed format.

Exported HTML files will only look nice when saved and displayed in the same directory as the prototype, because the referenced image files are located in the `\prototype\resources` subdirectory of the CD. This means that in order to create your own HTML exports, you will have to copy the `\prototype` directory to a directory on your hard disc drive.

The package GUILayout::References as well as abstract ScreenAreas have not been implemented for this proof-of-concept prototype. However, with the zoom functionality the prototype is supplied with all necessary elements for creating and displaying larger and more complex diagrams as would result from implementing and employing References.

6.4 Third Party Products Used

The EXE launcher that can be used to launch the prototype on Windows computers conveniently is created by DevWizard[173] and is distributed as freeware.

The Swing Look&Feel used for the main application is the Plastic XP Look and Feel by JGoodies[174] and is distributed under the BSD license (see `License - JGoodies.txt`).

The Swing Look&Feel used for the diagrams that gives them their sketchy look is the Napkin Look and Feel[175], which is distributed under the BSD license, but contains other packages that are distributed under their respective licenses (see `License - Napkin L&F.txt`).

[173] DevWizard 2002
[174] JGoodies 2004
[175] Arnold 2004

Chapter 7

Testing

7.1 Expressive Power of the Extension

A special-purpose modeling language that is not capable of expressing the elements of its domain is useless. To make sure our profile is capable of modeling the most common GUI elements, we have chosen to evaluate it against the elements of two important GUI building technologies.

7.1.1 XHTML 1.0 Elements

As has been explained in section 2.2, web design offers more freedom to the designers than the design of windowed applications, resulting in a larger design space that has to be explored using diagrams. Therefore, the GUI elements that can be created with web technologies are the most important ones.

The web is a hypertext medium whose documents are written in HTML. The latest incarnation and successor of HTML is XHTML[176], which has reformulated HTML to well formed XML. We have extracted the names of all tags of XHTML 1.0 from its DTD[177] and created a mapping from every tag to one of the elements of our profile. XHTML 1.0 consists of the same tags as the latest version, XHTML 1.1, which differs mainly in how the tags are organized.[178]

Table 5 shows these mappings. The tags mentioned in the first table row provide meta information that is of no importance to the layout out the page and thus is usually not modeled. If

---

[176] W3C 2002
[177] Available at http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd [19.11.03]
[178] W3C 2001

it is desired, they can of course be added using an UML note. The second row contains tags used for font formatting, which is below the threshold of being modeled. Text is always modeled as a Text stereotype, no matter its formatting.

Most web pages not only rely on (X)HTML to create their graphical representation but use an additional style sheet language like Cascading Style Sheets (CSS)[179] to format the (X)HTML output.

Type	Elements	Equivalent	Comments
Meta information	html, head, title, base, meta, link, noframes, style, script, noscript, isindex	– (UML note)	no need to be modeled
Font formatting	center, br, em, strong, q, sub, sup, tt, i, b, big, small, u, s, strike, basefont, font, pre, blockquote, span, address, ins, del, bdo, dfn, code, samp, kbd, var, cite, abbr, acronym	–	too fine-grained
Page	body	Screen	
Headings	h1, h2, h3, h4, h5, h6	Heading	
Lists	ul, ol, li, dl, dt, dd, dir, menu	Text, Link	
Text structure	p, div	Text	
Tables	table, caption, thead, tfoot, tbody, colgroup, col, tr, th, td	Text, multiple ScreenAreas	
Links	a, map, area	Link, Image+Link	
Forms	form, label, input, select, optgroup, option, textarea, button	Form	
Buttons	button, input type="button"	Link	
Logical Partitions	hr, fieldset, legend, table	ScreenArea	
Images	img	Image, Logo	
Objects	object, param, applet	Image, Image+Link	
Embedding	frame, iframe	ScreenArea	

Table 5. Mapping of XHTML Elements to Elements of Our UML Profile

7.1.2 CSS 2.0 Attributes

CSS adds the ability to web pages to create complex formatting rules for various output media. For example, there are attributes dedicated to visual, aural, or tactile representation. However, as out profile aims at visual layout, we have only examined attributes of the visual media

[179] W3C 1998

group. We created a list of attributes, extracted only those of the visual media group and put them in a mapping table as explained in the previous section (Table 6).

Because detailed formatting is not the aim of our extension, but that of CSS, most attributes have no equivalent in our profile. Nevertheless, CSS adds layering and visibility, both of which can be modeled using our profile easily.

Type	Elements	Equivalent	Comments
Background	background, background-attachment, background-color, background-image, background-position, background-repeat	–	too fine-grained
Margin and padding	margin, margin-top, margin-right, margin-bottom, margin-left, padding-top, padding-right, padding-bottom, padding-left, padding	–	implicitly
Color	color	–	deliberately omitted
Tables	table-layout, empty-cells	–	implicitly
Borders	border, border-collapse, border-color, border-spacing, border-style, border-top, border-right, border-bottom, border-left, border-top-color, border-right-color, border-bottom-color, border-left-color, border-top-style, border-right-style, border-bottom-style, border-left-style, border-top-width, border-right-width, border-bottom-width, border-left-width, border-width, outline, outline-color, outline-style, outline-width	–	too fine-grained
Font formatting	font, font-family, font-size, font-size-adjust, font-stretch, font-style, font-variant, font-weight, text-decoration	–	too fine-grained
Text formatting and converting	letter-spacing, text-shadow, text-transform, quotes, direction, unicode-bidi, white-space, word-spacing, content, display	–	too fine-grained
Text structure	line-height, text-align, text-indent, vertical-align	Text	
Positioning and dimensioning	position, top , bottom, left, right, caption-side, clear size, height, width, max-height, max-width, min-height, min-width	ScreenArea	implicitly
Visibility	clip, overflow, visibility	ScreenArea	implicitly
Layers	float, z-index	ScreenArea	implicitly
Lists	list-style, list-style-image, list-style-position, list-style-type, marker-offset, counter-increment, counter-offset	Text	
Cursor	cursor	UML note	only if essential

Table 6. Mapping of CSS 2.0 attributes to Elements of Our UML Profile

7.1.3 Java Swing Classes

Second important after web design is GUI design for windowed applications. As there is a plentitude of widget toolkits on the market, we have decided to examine one of them as an example. We chose Java Swing because it is a widespread, platform independent toolkit that is well documented. At the time this thesis was written, the latest version of the Java 2 SDK was 1.4, and this is what our analysis is based on.

Type	Elements	Equivalent	Comments
Meta classes	JApplet, JComponent, JDesktopPane, JLayeredPane, JRootPane, JViewport	–	no need to be modeled
Forms	JCheckBox, JCheckBoxMenuItem, JComboBox, JFormattedTextField, JFormattedTextField.AbstractFormatter, JFormattedTextField.AbstractFormatterFactory, JPasswordField, JRadioButton, JRadioButtonMenuItem, JSlider, JSpinner, JSpinner.DateEditor, JSpinner.DefaultEditor, JSpinner.ListEditor, JSpinner.NumberEditor, JTextArea, JTextField, JToggleButton, JToggleButton.ToggleButtonModel	Form	multiple classes are displayed as one form
Editing	JEditorPane, JTextPane	Workspace	
Button	JButton	Link	
Grouping and layout	JPanel, JSeparator, JSplitPane JTabbedPane	ContainerScreenArea, ScreenArea	
Windows and Dialogs	JFrame, JInternalFrame, JInternalFrame.JDesktopIcon, JWindow JDialog, JFileChooser, JOptionPane	Screen, ScreenArea	
Texts and Images	JLabel	Text, Image	
Table	JList, JTable	Text, multiple ScreenAreas	
Menu	JMenu, JMenuBar, JMenuItem, JPopupMenu, JPopupMenu.Separator, JToolBar, JToolBar.Separator	Navigation, Links	
Tree	JTree, JTree.DynamicUtilTreeNode, JTree.EmptySelectionModel	Links	
Scrollbar	JScrollBar, JScrollPane	–	too fine-grained
Progress bar	JProgressBar	–	too fine-grained
Tooltip	JToolTip	–	too fine-grained

Table 7. Mapping of Java Swing Classes to Elements of Our UML Profile

Again, we have extracted the relevant classes and put them into a mapping table. The list of classes is from the SDK documentation[180] and includes all classes of the package javax.swing.

One characteristic that has been mentioned before (cf. 2.2 and 5.4.8) is reflected in Table 7: Layout for windowed application resolves mainly to form layout. In the second row of the table, many Java classes are grouped for only one equivalent in the profile, the Form stereotype.

7.2 Designers' Feedback

We have presented our results to some of the designers we had interviewed during analysis. They were quite pleased with general look and feel of the diagrams produced with the profile. They stated that the meaning of the diagram icons could be grasped by anybody because of their similarity to sketches. While not that important for GUI layout itself, understanding the navigational structure of a website was important to the designers because they often create the site structure themselves if the project manager has not done it. A tool that integrates navigational und layout issues as described in our profile could help them avoid navigational deadlocks they sometimes encounter. They regarded the tool especially helpful in situations where layout and navigational specifications made by project manager and customer are spread over several large documents, leading to a lot of searching in order to find a specific piece of information.

Due to the complete-view nature of the diagrams, the designers said that a model of a website could constitute a whole briefing meeting with a project manager or even substitute it.

Export functionality was appreciated, especially the Photoshop file format. One designer demonstrated how he would fill the generated wireframe with dummy content in order to produce a low-fidelity mockup within few minutes. They objected to the color used in the Photoshop file for drawing the stereotype icons; we have therefore altered it according to their wishes.

[180] The source is available at http://java.sun.com/j2se/1.4.2/docs/api/javax/swing/package-frame.html [19.11.03].

Chapter 8

Results

8.1 Conclusion

We have presented a standards-conformant extension to UML 2.0 to integrate GUI layout into software engineering. First, we have performed an in-depth analysis of the situation both in software engineering and in GUI design and found that modeling is a concept common to both of them. While software engineers create rigid, geometrical models of software systems, GUI designers do quick sketches of screen layouts using paper and pencil. We have found a way to unify both approaches by creating geometrical abstractions of designers' sketches of GUI elements and combining them into models of complex screen layouts. Our selection of elements has proven powerful enough to mapping all GUI elements that can be created by popular GUI technologies to a construct of our model.

Our extension is a UML 2.0 profile that incorporates layout information by using metaclasses added to UML by the recent specification of the UML 2.0 Diagram Interchange. Being based completely on the UML metamodel and using its designated extension mechanisms, it is a lightweight addition that can be implemented in any modeling tool that offers complete support for UML 2.0.

The profile is based on stereotypes of Class that are used to describe GUI layout and functionality. The base unit of our profile is the ScreenArea, which is a stereotype of Class. It represents a piece of screen space that has a purpose. This purpose can be made clearer by choosing between one of the two subclasses of ScreenArea, one of which may contain other ScreenAreas, the other one carrying specific functionality. The former allows for a hierarchical nesting to be created, the latter allows complex arrangements of functionalities on the GUI to be modeled. Functionalities are represented by several stereotypes, the icons of which have been derived from designer sketches.

These models can be combined with the existing UML diagrams in various ways. We have outlined a combination with use case diagrams. Each UseCase may be associated to a ScreenArea and its contents, creating a link between the use case and the screen areas used to present its user interface, and a link between behavioral and structural modeling of GUIs. These two can be further interrelated using Activities that are partitioned in a way to reflect the user interface. In our example, we have shown how an Activity in a web application can be structured to support GUI layout. Therefore we have created three partitions, backend, presentation and user, and arranged all Actions in the backend and user partitions. Screens are represented by object flows that flow from the lowermost partition, backend, to the uppermost partition, user, passing through the intermediate presentation partition. We have shown that this approach is compliant to UML and can adequately model the flow of Action in GUI applications.

Presented with our results, designers stated that the profile we have developed would be useful for creating better navigational structures and for creating low-fidelity mockups very quickly. They said that the diagrams created could be interpreted correctly by everybody. Based on their assessment that GUI layout and navigational diagrams could be the only means of communication in a designer briefing, we believe that using this type of artifact would be especially useful for remote designers; for instance, freelancers could quickly be briefed in detail and without ambiguities.

8.2 Outlook

As a next step, support in a general-purpose UML modeling tool is planned. We have already been exploring the capabilities ArgoUML, a Java-based open-source modeling tool, and consider adapting it to our profile

The possibilities of combining GUI Layout Diagrams with other UML diagrams can still be improved. Activity diagrams look promising, especially in invoking behavior explicitly via ScreenAreas.

We have explicitly excluded extending software development processes with support for GUI layout. The Rational Unified Process can be extended to use the Profile for GUI Layout in order to create a guideline on how to use the profile in software projects.

As our profile incorporates UML's Diagram Interchange, the diagrams including all layout information can be stored as XML data using XMI. This data can be transformed with XSLT

into an XML dialect for GUIs like the User Interface Markup Language (UIML)[181] or the XML User Interface Language (XUL)[182] to generate prototypical applications from the modeled Screens.[183] Of particular interest is the embedding of code into models in order to build fully functional prototypes. We are eager to see synergies from our work and the efforts towards executable UML.

[181] OASIS 2004
[182] Mozilla.org 2001
[183] Blankenhorn & Jeckle 2004

Appendix A

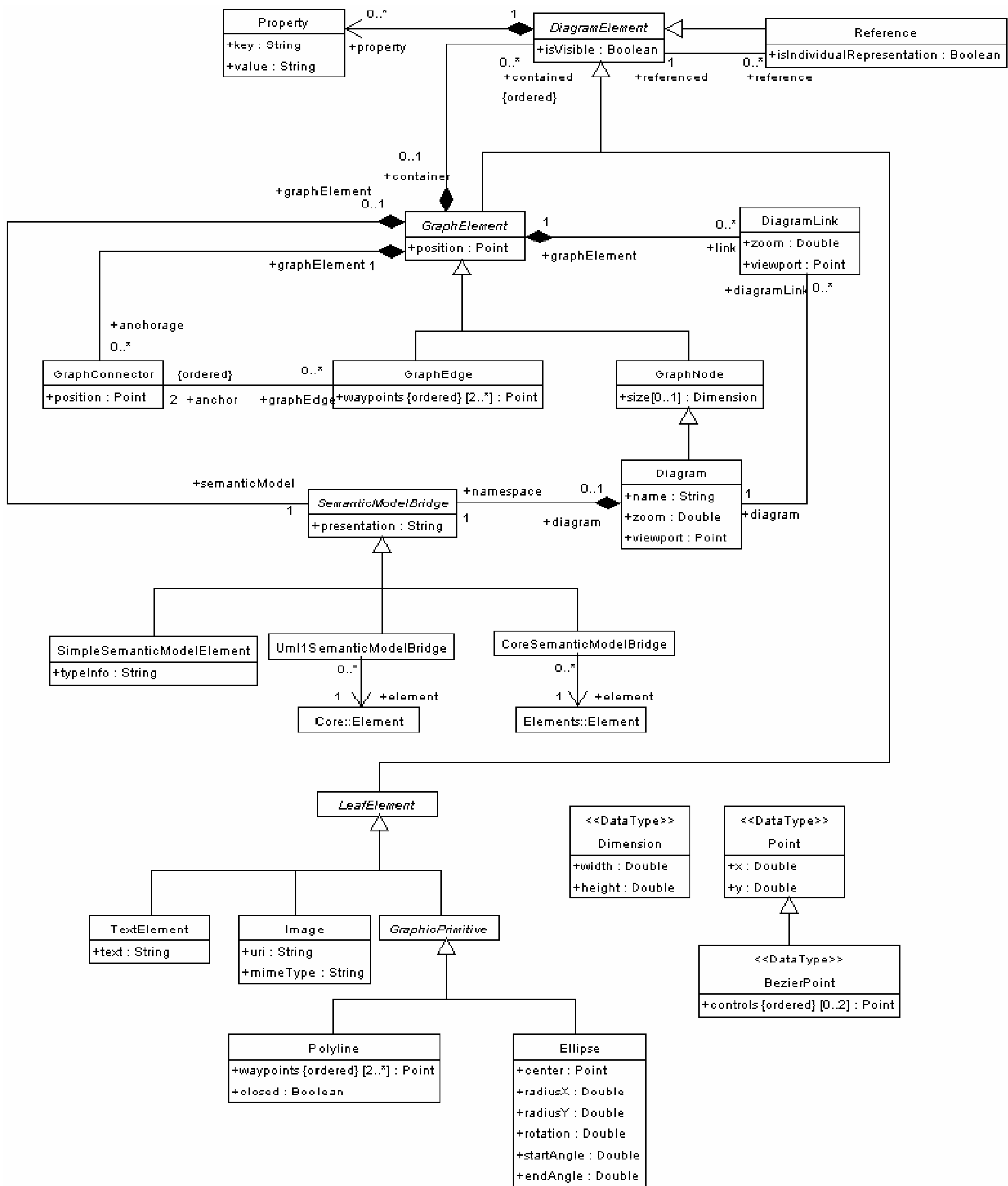
UML 2.0 DI Metamodel

"This extension adds a new package to the current UML metamodel packages. Yet the existing standard is not changed in any way. Also, changes to the UML metamodel due to version updates should not affect this model as long as the highlevel notions of Core::Element (as used in UML 1.x) and Elements::Element (as used in UML 2.0 as well as all metamodels based on the Common Core) are maintained. The extension and the UML metamodel are kept largely independent such that solely links from the extension to the UML metamodel are included. Thus, graphical and model information are cleanly separated. [...] The proposed package contains elements to reflect the diagram information of any diagram element of the standard UML."[184]

"The underlying concept [...] is based on the idea of modeling the contents of the UML diagrams as graphs. The core classes are GraphNode and GraphEdge. Every visible model element is represented either by a GraphNode or by a GraphEdge. The base class of the graph elements is GraphElement. Graph elements are linked via a class called GraphConnector. This allows linking of a GraphEdge with a GraphNode or another GraphEdge. The latter case is an extension to the concept of a pure mathematical graph."[185]

---

[184] OMG 2003c, 7
[185] OMG 2003c, 9



Property
+key : String
+value : String
0..*
+property
1
DiagramElement
+isVisible : Boolean
Reference
+isIndividualRepresentation : Boolean
0..*
+contained
{ordered}
1
+referenced
0..*
+reference
0..1
+container
+graphElement
0..1
GraphElement
+position : Point
1
+graphElement
0..*
+link
DiagramLink
+zoom : Double
+viewport : Point
+diagramLink
0..*
+graphElement 1
+anchorage
0..*
GraphConnector
+position : Point
{ordered}
2 +anchor
0..*
+graphEdge
GraphEdge
+waypoints {ordered} [2..*] : Point
GraphNode
+size[0..1] : Dimension
Diagram
+name : String
+zoom : Double
+viewport : Point
+semanticModel
1
SemanticModelBridge
+presentation : String
+namespace
1
0..1
+diagram
1
+diagram
SimpleSemanticModelElement
+typeInfo : String
Uml1SemanticModelBridge
0..*
1
+element
Core::Element
CoreSemanticModelBridge
0..*
1
+element
Elements::Element
LeafElement
<<DataType>>
Dimension
+width : Double
+height : Double
<<DataType>>
Point
+x : Double
+y : Double
TextElement
+text : String
Image
+uri : String
+mimeType : String
GraphicPrimitive
<<DataType>>
BezierPoint
+controls {ordered} [0..2] : Point
Polyline
+waypoints {ordered} [2..*] : Point
+closed : Boolean
Ellipse
+center : Point
+radiusX : Double
+radiusY : Double
+rotation : Double
+startAngle : Double
+endAngle : Double

Appendix B

References

ARNOLD, Ken (2004): Napkin Look & Feel. Available at http://napkinlaf.sourceforge.net/ [18.05.04]

BARESI, Luciano et al. (2001): Extending UML for Modeling Web Applications. In Proceedings of the 34th Annual Hawaii International Conference on System Sciences, pages 1285 -1294, Maui (USA), January 2001.

BAUMEISTER, Hubert et al. (1999): Towards a UML Extension for Hypermedia Design. In: FRANCE, R.; RUMPE, B. (eds.): Proceedings of the UML '99 Conference, LNCS, Vol. 1723, Springer-Verlag, pp. 614-629.

BECK, Kent (1999): Extreme Programming Explained. Reading: Addison Wesley, 1999.

BERLINER, Brian (1990): CVS II: Parallelizing Software Development. In: Proceedings of the USENIX Winter 1990 Technical Conference, USENIX Association, Berkeley, pp. 341-352.

BERNER, Stefan et al. (1999): A Classification of Stereotypes for Object-Oriented Modeling Languages. In: FRANCE, Robert; RUMPE, Bernhard (1999): UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30, 1999, pp. 249-264.

BLANKENHORN, Kai; JECKLE, Mario (2004): A UML Profile for GUI Layout. Submitted for publication at net.objectdays 2004.

BOGER, Marko; JECKLE, Mario et al. (2002): Diagram Interchange for UML. In: JÉZÉQUEL, J.-M. ; HUSSMANN, H.; COOK, S. (Eds.): UML 2002 – The Unified Modeling Language 5th International Conference, Dresden, Germany, September 30–October 4, 2002, Springer LNCS vol. 2460.

Booch, Grady (1994): Object-Oriented Analysis and Design with Applications, 2nd Edition. Reading: Addison Wesley, 1994.

Collins-Sussman, Ben et al. (2004): Version Control with Subversion. To be published by O'Reilly. Available at http://svnbook.red-bean.com/ [04.03.04].

Conallen, Jim (2003): Building Web Applications with UML, Second Edition. Boston: Addison Wesley, 2003.

da Silva, Paolo Pinheiro & Paton, Norman W. (2000): UMLi: The Unified Modeling Language for Interactive Applications. In: <<UML>> 2000 - The Unified Modeling Language: Advancing the Standard. LNCS Vol. 1939. Springer, pp. 117-132.

DevWizard (2002): JavaEXE. Available at http://devwizard.free.fr/html/en/JavaExe.html [18.05.04]

Dolog, Peter; Bieliková, Mária (2002): Hypermedia Modeling Using UML. In: Hanacek, Petr: Proc. of ISM'2002, April 2002.

Foley, James D. et al. (1990): Computer Graphics: Principles and Practice, Second Edition. Reading: Addison Wesley, 1990.

Fowler, Martin (2000): Is Design Dead? In: Proceedings of the XP2000 conference. Cagliari, Sardinia, Italy, 2000. Available at http://martinfowler.com/articles/designDead.html [20.03.04].

Galitz, Wilbert O. (2002): The Essential Guide to User Interface Design, Second Edition. New York: John Wiley & Sons.

Gorshkova, Ekaterina; Novikov, Boris (2002): Exploiting UML Extensibility in the Design of Web Information Systems. Proc. of the DB\&IS'2002, Tallinn, Estonia, June 2002, pp. 49-64

Gotthelf, Jeff (unknown year): Personal homepage. Available at http://www.jeffgothelf.com/examples.html [03.05.04].

Gould, John D.; Lewis, Clayton (1985): Designing for Usability: Key Principles and what designers think. Communications of the ACM, 28, 3 (March 1985), pp. 300-311. Available at http://doi.acm.org/10.1145/3166.3170 [01.03.04].

Hennicker, Rolf; Koch, Nora (2001): Modeling the User Interface of Web Applications with UML. In: Evans, A.; France, R.; Moreira, A. (eds.) (2001): Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists, Workshop of the pUML-Group at the UML 2001. Gesellschaft für Informatik, Köllen Druck+Verlag, October 2001, pp. 158-172.

Jacobson, Ivar et al. (1998): The Unified Software Development Process. Reading: Addison Wesley, 1998.

Jeckle, Mario (2004): Personal Conversation on 04-04-06.

JGoodies (2004): JGoodies Looks Freeware Library. Available at http://www.jgoodies.com↩
/freeware/looks/index.html [18.05.04]

Kay, Michael (2000): XSLT Programmer's Reference. Birmingham: Wrox Press, 2000.

Koch, Nora (2001): Software Engineering for Adaptive Hypermedia Systems: Reference Model, Modeling Techniques and Development Process. PhD. Thesis, Ludwig-Maximilians-Universität München, UNIDRUCK Verlag.

Koch, Nora et al. (2000): Extending UML to Model Navigation and Presentation in Web Applications. In: Winters, Geri; Winters, Jason (eds.) (2000): Modeling Web Applications in the UML Workshop, UML2000, York, England, October 2000.

Kruchten, Philippe (1999): The Rational Unified Process, German language version. Munich: Addison Wesley Longman.

Lieberman, Ben (2001): UML Activity Diagrams: Detailing User Interface Navigation. The Rational Edge Oct. 2001. Available at http://www.ibm.com/developerworks/rational/library↩
/content/RationalEdge/oct01/UMLActivityDiagramsOct01.pdf [17.04.04].

Lin, James et al. (2000): DENIM: Finding a Tighter Fit Between Tools and Practice For Web Site Design. In: CHI Letters: Human Factors in Computing Systems, CHI 2000, pp. 510-517.

Lohse, Gerald L.; Spiller, Peter (1998): Quantifying the effect of user interface design features on cyberstore traffic and sales. In: Proceedings of CHI'98, New York: ACM, 1998, pp. 211-218.

Mozilla.org: XML User Interface Language (XUL) 1.0, Available at http://www.mozilla.org↩
/projects/xul/xul.html [01.05.2004].

Myers, Brad A. (1993): Why are Human-Computer Interfaces Difficult to Design and Implement? Carnegie Mellon University Technical Report CMU-CS-93-183, July 1993. Available at http://citeseer.nj.nec.com/myers93why.html [19.02.04].

Newman, Mark W.; Landay, James A. (2000): Sitemaps, Storyboards, and Specifications: A Sketch of Web Site Design Practice. In: Boyarski, D.; Kellogg, W. A. (Eds.): Proceedings of the Conference on Designing Interactive Systems: Processes, Practices, Methods, Techniques. New York: ACM Press, pp. 263-274.

OASIS: User Interface Markup Language (UIML) Specification, Available at http://www.oasis-open.org/committees/documents.php?wg_abbrev=uiml [01.05.2004].

OMG (2001): UML 2.0 Diagram Interchange RFP. Available at http://www.omg.org↩ /cgi-bin/apps/doc?ad/01-02-39.pdf [18.04.04].

OMG (2002a): XML Metadata Interchange (XMI) Specification v1.2, Framingham, USA, January 2002. Available at http://cgi.omg.org/docs/formal/02-01-01.pdf [07.05.04].

OMG (2002b): UML Profile for CORBA. Available at http://www.omg.org/docs/formal/02-04-01.pdf [03.05.04].

OMG (2003a): UML 2.0 Infrastructure Final Adopted Specification (ptc/03-09-15), Available at http://www.omg.org/docs/ptc/03-09-15.pdf [03.05.04].

OMG (2003b): UML 2.0 Superstructure Final Adopted Specification (ptc/03-08-02), Available at http://www.omg.org/docs/ptc/03-08-02. [03.05.04].

OMG (2003c): Unified Modeling Language: Diagram Interchange, version 2.0. Final Adopted Specification (ptc/03-09-01), Available at http://www.omg.org/docs/ptc/03-09-01.pdf [03.05.04].

OMG (2003d): Meta Object Facility (MOF) 2.0 Core Specification. OMG Final Adopted Specification (ptc/03-10-04). Available at http://www.omg.org/docs/ptc/03-10-04.pdf [03.05.04].

OMG (2003e): UML 2.0 OCL Draft Adopted Specification (ptc/03-08-08) Available at http://www.omg.org/docs/ptc/03-08-08.pdf [03.05.04].

Oxford (1995): Oxford Advanced Learner's Dictionary. Oxford University Press, Oxford, England.

Palanque, Philippe; Bastide, Rémi (2003): UML for Interactive Systems: What is Missing. Proceedings of the INTERACT 2003 workshop, September 1-2, 2003, Zurich, Switzerland. Available at http://www.se-hci.org/bridging/interact/p96-99.pdf [03.05.04].

Phillips, Chris; Kemp, Elizabeth (2002): In Support of User Interface Design in the Rational Unified Process. In: Grundy, John; Calder, Paul (eds.): Third Australasian User Interface Conference (AUIC 2002), Melbourne, Australia, Conferences in Research and Practice in Information Technology, Vol. 7.

Rettig, Marc (1994): Prototyping for Tiny Fingers. Communications of the ACM, 37 (4).

ROBBINS, Jason Elliot (1999): Cognitive Support Features for Software Development Tools. PhD thesis, University of California, Irvine, 1999. Available at http://argouml.tigris.org/docs↩
/robbins_dissertation/ [04.03.04].

ROSSON, Mary Beth; CARROLL, John M. (2002): Usability Engineering – Scenario-Based Development of Human-Computer Interaction. San Francisco: Morgan Kaufmann, 2002.

RUDD, Jim et al. (1996): Low vs. High-Fidelity Prototyping Debate. Interactions of the ACM, vol. 3, issue 1, pp. 76-85.

RUMBAUGH, James et al. (1998): The Unified Modeling Language Reference Manual. Addison Wesley, Reading, 1998.

SAUER, Stefan; ENGELS, Gregor (1999a): Extending UML for modeling of multimedia applications, *Proc. IEEE Symposium on Visual Languages*, Tokyo, September 13-16, 1999.

SAUER, Stefan; ENGELS, Gregor (1999b): OMMMA: An Object-Oriented Approach for Modeling Multimedia Information Systems. In: GOLUBCHIK, L.; TSOTRAS V.J. (eds.): Proc. Fifth International Workshop on Multimedia Information Systems - MIS'99, October 21-23, 1999, Miramonte Resort, Palm Springs Desert, California, USA, 1999, pp. 64-71.

SAUER, Stefan; ENGELS, Gregor (1999c): UML-based modeling of multimedia applications, Proc. Modellierung'99, Karlsruhe, March 10-12, 1999, Teubner, Stuttgart, 1999 (in German).

SEFELIN, Reinhard et al. (2003): Paper Prototyping – What is it good for? A Comparison of Paper- and Computer-based Low-fidelity Prototyping. CHI 2003, New York: ACM, 2003, pp. 778-779.

SHEDROFF, Nathan (2000): Information Interaction Design: A Unified Field Theory of Design. In: JACOBSON, Bob (ed.) (2000): Information Design. Cambridge, Massachusetts: MIT Press, 2000, pp. 267-292.

SHNEIDERMAN, Ben (1997): Designing the User Interface, Third Edition. Addison Wesley, Reading, 1997.

SNYDER, Carolyn (1996): Using Paper Prototypes to Manage Risk, Software Design and Publisher Magazine, October 1996.

SNYDER, Carolyn (2001): Paper prototyping. IBM developerWorks article, 2/27/2002. Available at http://www-106.ibm.com/developerworks/library/us-paper [11.03.04].

Thompson, Michael; Wishbow, Nina (1992): Prototyping: Tools and Techniques: Improving Software and Documentation Quality Through Rapid Prototyping. Proceedings of the 10th annual international conference on Systems documentation, Ottawa, Canada, pp. 191-199.

Trætteberg, Hallvard (2002): Model-based User Interface Design. Doctoral Thesis, Norwegian University of Science and Technology, Trondheim, Norway.

Unisys (2000): UML 2.0 Diagram Interchange RFP Presentation. Available at http://www.omg.org/docs/ad/00-12-06.pdf [03.05.04].

van der Veer, Gerrit; van Vliet, Hans (2001): A Plea for a Poor Man's HCI Component in Software Engineering and Computer Science Curricula. In: Computer Science Education, Vol. 13, no 3 (Special Issue on Human-Computer Interaction), pp 207-226. Available at http://www.cs.vu.nl/~hans/publications/y2003/plea/vliet.pdf [19.02.04].

W3C (1998): Cascading Style Sheets, level 2: CSS2 Specification. Available at http://www.w3.org/TR/REC-CSS2/ [07.05.04].

W3C (1999): XSL Transformations (XSLT) Version 1.0. Available at http://www.w3.org/TR/xslt [06.05.04].

W3C (2001): XHTML™ 1.1 – Module-based XHTML, Appendix A. Available at http://www.w3.org/TR/xhtml11 [19.05.04].

W3C (2002): XHTML™ 1.0 The Extensible HyperText Markup Language (Second Edition). Available at http://www.w3.org/TR/xhtml1/ [07.05.04].

Wikipedia (2004): Wikipedia: The Free Encyclopedia. Available at http://www.wikipedia.org [20.04.04].