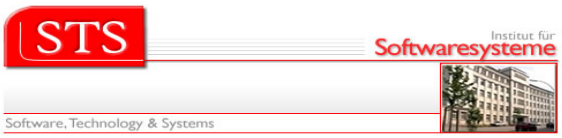


Modeling of User Interfaces for Conceptual Content Management Systems

PROJECT WORK

by

Gerald N. Mofor

Information and Communications Systems
Hamburg University of Technology

Supervised by

Prof. Dr. Joachim W. Schmidt

Institute of Software, Technology & Systems

Dr. Hans-Werner Sehring

Institute of Software, Technology & Systems

Hamburg, 17th August 2006

Acknowledgments

I would first of all like to thank Professor Joachim.W Schmidt for boosting my moral and giving me once more the chance to develop my theoretical skills in this project work.

Special thanks goes to my supervisor, Dr Hans-Werner Sehring, for being very patient and acting as a source for advice in almost every domain. I lack the right expressions for this, is it always an honour for me to work with the COCoMa project team. The idea is wonderful. I always have the feeling I converse with a super computer.

Last but not least, I would like to thank Sebastian Boßung, a research assistant of Professor J.W. Schmidt, for his mental support and realistic tips to go around solving problems in general. This has really helped me moving first straight to the point. I must confess, I am lucky to have seen your days at STS and to have witnessed your abilities.

I feel elated to have been part of a dream team.

Abstract

Conceptual Content Management Systems (CCMSs) were conceived with the intention of superseding some of the limitations of information systems, such as either failing to stay *open* to constantly changing domain environments or not being able to *dynamically* integrate evolution steps. In as much as this target has been achieved CCMSs, however, still need to take responsibility to offer at least the same services as conventional information systems. This study examines one of the paramount services, namely to enable interaction with the user. For this, CCMSs need to offer *visualization* and the visualization has to be flexible to accommodate changes and support on-the-fly evolution by adopting CCM system properties of *openness* and *dynamics*. This primarily necessitates the design of *flexible* and *scalable* conceptual User Interface (UI) *component* and *technology* models that can be mapped onto concrete ones by *generative* means. This study furthermore argues that *understandability* and hence *usability* can be fostered on the UI by simulating *Object-Oriented (OO)* concepts, such as inheritance, association and composition. This is done by *formally* binding every application specific information type to be visualized with conceptual UI components. On request for displaying information, valid[1] formal bindings are evaluated, followed by a view update with respective *actual* bindings for these formal bindings. Together with the above-mentioned advantages, the end result of modeling UIs for CCMSs is that *complex UIs are easily created*, since an interface designer need not worry about their real implementations on target UI technologies.

Content

Chapter 1: Introduction…….…….. 1
1.1 Motivation……. 1
1.2 A Comparison of Realization Approaches……. 3
1.2.1 Scripting Languages……... 3
1.2.2 Generic UI Technologies..... 3
1.2.3 Asset Language and its Compiler Framework……4
1.3 Organizational Structure of the Study…… 4

Chapter 2: Analysis of Visualization for CCMMs…..……5
2.1 State of the Art and Scope of Work.……... 6
2.2 Terminology Definitions…….……... 6
2.3 Problems Faced by Conventional UIs...…….. 8
2.4 Rational of the Study…….. 9
2.5 Project Development Process Model…….. 10
2.6 Requirements Analysis……..… 10
2.7 The Conceptual Content Management System Environment……13
2.7.1 Properties…… 13
2.7.2 ADL Compiler Based System Construction……..…13
2.8 Naïve Solution Proposals……....14
2.8.1 Expected Visualization Behavior…….....… 14
2.8.2 The Need For Models and Separation of Concerns……... 15

Chapter 3: Conceptual UI Model Design…… 17
3.1 Modeling the UI Space…….. 18
3.2 Visualization Construction Scenario……. 18
3.2.1 Logical UI Component Domain Model…….. 19
3.2.1.1 Model-View-Controller……... 19
3.2.1.2 UI Component Model……...……... 21
3.2.1.3 UI Container Layout Modeling…..……...….. 22

3.2.2 Logical Technology Domain Model…… 24
3.2.2.1 Object-based Technology Model…….…….. 25
3.2.2.2 Class-based Technology Model...…… 25
3.3 Linkage Patterns between UI Components and UI Technologies……..…….. 26
3.4 Asset Binding……..…….. 27
3.4.1 Formal Binding..…….. 27
3.4.2 Actual Binding…...……. 28
3.5 UI Generation Process..…… 29
3.5.1 UI Openness….……29
3.5.2 UI Dynamics.…… 29
3.5.3 Visualization Modality……. 30

Chapter 4: Conceptual Model Implementation…….. 31
4.1 UI Modeler's Code……...……. 31
4.1.1 Component Model Description…...……31
4.1.2 Technology Description Model……... 34
4.1.3 Component Implementation Description Model……...…….. 35
4.1.4 Swing Implementation Abstraction Description Model…….. 36
4.2 Application Domain Model…….. 37
4.3 Display Constraints Implementation……. 37
4.4 User Interface Description Implementation…….. 38

Chapter 5: Prototype Experiment……41
5.1 Experiment Environment…..…….. 41
5.2 Visualization Decision.....…….. 45
5.3 Generator Design Structure…….…….. 46
5.4 Generated Code Design Structure….…… 48

Chapter 6: Evaluation and Outlook……...…….……51
6.1 Evaluation…….……. 51
6.2 Outlook…….. 52

Appendix A: Detailed Component Model……...……..... 54
Appendix B: Detailed Swing Implementation Model…….....58
Appendix C: User Model.....…..……...65
Appendix D: Modality Code...…….. 69

Bibliography……...…….. 71

Chapter 1

Introduction

Content Management Systems (CMSs) are considered a critical success factor in many E-Commerce and E-business scenarios in that their primary role is to ease the process of creating, managing the life cycle, discovering, archiving and publishing corporate information. Like most information systems, CMSs suffer a great deal from being inefficient in either allowing flexibility on the schema level or effectuating these changes on- the-fly when evolving the system. In view of solving these inefficiencies *Conceptual Content Management Systems (CCMSs)* were conceived with the properties of *openness* towards system modification and *dynamics* to support on-the-fly system evolution.

1.1 Motivation

Although CCMSs have been successful in achieving the additional requirement for system responsiveness[1] based on the modular architecture shown in figure 1.1, they, however, still have to take responsibility to offer at least the same services as conventional CMSs. This study claims that one of such services is to allow users to interact and exploit the functionality of the software system. This entails publishing of information and giving room for system control.

[1] openness and dynamics

Therefore, to enable interaction and publication of information, CCMSs need *visualization*[2]. Furthermore CCMSs evolve dynamically due to their open and dynamic properties. This imposes both open and dynamic behavior on the visualization.

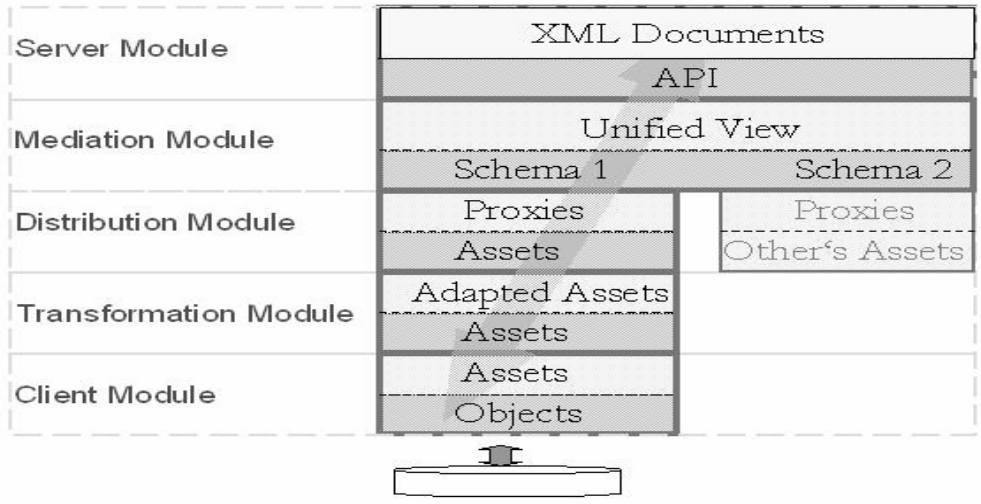


Figure 1.1 *Modular architecture of CCMSs [Sehr04]*

Figure 1.1 shows the present structure of the modular architecture of CCMSs systems. CCMSs are characterized by layered modules, specialized for individual tasks. Of paramount interest for this study is the `Server Module`. It is responsible for enabling communication with the external world. At the moment, requests are transmitted in XML format which are then translated into API calls to the underlying system.

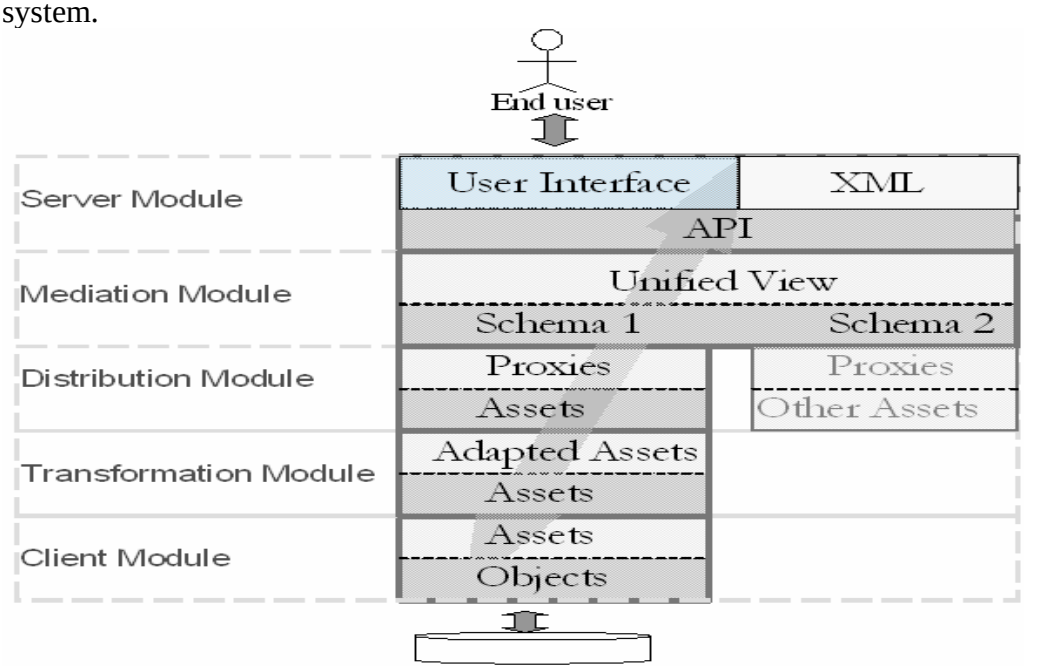


Figure 1.2 *Modular architecture of CCMSs with Visualization*

[2] also known as *User Interface (UI)*

What is missing in figure1.1 is human interaction and for this reason figure 1.2 proposes an enhancement of the present CCMS architecture to incorporate human interaction. This is done through the introduction of a *User Interface (UI)*.

1.2 A Comparison of Realization Approaches

There are several ways to implement visualizations for CCMSs. This section will consider the three main ones:

- Scripting languages
- Generic UI technologies, and the
- Asset language and its compiler framework

1.2.1 Scripting Languages

One option for implementing visualizations for CCMSs is to make use of scripting languages. They are characterized by being interpreted, memory-managed and exhibit dynamic behavior. They accomplish new tasks by combining existing components and can control any GUI-based application by executing a series of commands that might have otherwise been entered at the command prompt. Their main advantages lie in the following:

- Fast to program,
- Much smaller program length of the script files.

These great advantages come at a heavy price, namely that of the following:

- Poor performance arising from frequent interpretation,
- Significantly slower program execution and higher memory consumption. This situation is aggravated when dealing with a large, complex UI,
- With a change of the underlying system due to openness and dynamics, scripting-based visualizations for CCMSs are not able to meet up in reflecting these changes dynamically. One will have to resort to manual coding, and depending on how much needs to be modified it can be cumbersome.

1.2.2 Generic UI Technologies

Another approach to implement visualizations for CCMS is to use generic means. An example of a way in which user interfaces can be created in a generic fashion is by imploring the technology User Interface Markup Language (UIML) [APBW+99]. UIML is an XML based language for describing user interfaces, which can be implemented on any platform (appliance independent). The main advantage is that it can be used to span a variety of platform paradigms such as desktop, handheld PC, palm and cellular phone. Abstract User Interface Markup Language (AUIML) [MWK04] is a further example of a technology used for describing generic user interfaces. AUIML assists in the development of graphical user interfaces running either as Swing or Web applications. Other examples of XML-based languages used for defining generic user interfaces are eXtensible Application Markup Language

(XAML) [DRDY05] and XML User Interface Language (XUL) [BSD01]. The main drawback of implementing UIs by generic means is that the above-mentioned technologies are inflexible when coping with system evolution.

1.2.3 Asset Language and its Compiler Framework

The next alternative is to implement the visualization based on a similar pattern in which CCMSs are constructed. That is, using the Asset Description Language (refer to section 2.9.2 for definition) in combination with a compiler framework. This alternative offers the following advantages:

- Visualizations are easy to implement. This emanates from the fact that the asset language is user friendly and it is designed in the way users view entities in the real world (i.e. according to characteristics and relationships [Sehr04]),
- Openness and dynamics are given for free,
- Fast execution[3], and
- A homogeneous[4] system as a whole due to a match with the underlying system. This way maintenance is easy.

1.3 Organizational Structure of the Study

In this chapter so far, a brief overview of the context and the problem was given. The preliminary conclusions are:

- CCMSs need visualization to support interaction with the user.
- In order to meet-up with the dynamic evolution of CCMSs, the visualization will be generated based on asset definitions compiled by the compiler framework.

The next chapters will delve in-depth into the problems that arise with the implementation option and offer some solutions. For now, a brief overview of what each chapter will deal with is presented.

Chapter 2 analyses the problem for modeling visualizations by first of all stating the state-of-the-art of other studies prior to this study. It goes on further to offer some requirements on the visualization for CCMSs and finally delimits the scope of the work.

Chapter 3 discusses some design issues aimed at solving requirements on the visualization system. Along side the design decisions will be stated.

Chapter 4 goes a step closer to the implementations of the chosen design alternatives in the asset language.

Chapter 5 will verify the models by implementing a prototype. The scope of implementation will be stated and the results will be interpreted.

Chapter 6 closes up the study with a brief evaluation of the study and an outlook.

[3] due to the asset compiler framework

[4] everything is one language: asset language

Chapter 2

Analysis of Visualization for CCMSs

The last chapter briefly introduced the motivation of this study and it was made clear that a user interface needs to be provided by CCMSs in order to be considered as part of the family of interactive systems. In recent years the role of UIs in highly interactive software systems has become very important because they act as an intermediary between the user – understanding the user's language – and the system – translating the user's request into the system's language. This property is especially important in the business world in the sense that it is seen as one of the most import source of a corporate's core competence – attracting customers, offering services, maintaining customer relationships, and also making the employees of the business to get acquainted to the software in order to easily fulfill their tasks. UIs for CCMSs also have to follow this trend and also contribute more by offering better advantages compared to UIs on conventional systems. The next section will shortly describe some prior works done closely related to this study.

2.1 State of the Art and Scope of Work

This study is inspired by two main literatures [Sehr04][Xu04], which although being concerned with to topic visualization for CCMSs are yet different in some aspects.

[Sehr04] mentions the basic need for visualization in CCMSs and makes mention of some heuristics that good quality UIs need to follow in order to be more effective. This work can be stated as merely providing the general guidelines of what needs to be considered without dealing with the real implementation.

[Xu04] goes a step closer into the realization by discussing in a general note the classification scheme of UI components and how they can be modeled. Some general design considerations are given, together with their pro and contra arguments. It could be considered as more of a theoretical research work.

Based on the above mentioned prior work done in this area, this study will contribute further by re-examining some of the models proposed by [Xu04] and combine some of the ideas from [Sehr04] to come up with implementation feasible design models for the realization of the visualization for CCMSs.

2.2 Terminology Definitions

So far this study has been working on some terminology assumptions, which need some formal definition in this section.

a) Content Management System

A content management system is a software system for organizing and facilitating the processes of creation, organizing, managing, storing, searching and publishing of (complex) multimedia content such as text, video, audio, images, and maps.

b) Usability

Usability refers to the extent in which a UI takes human psychological and physiological factors into account. It serves as a measure of how effective, efficient and satisfying the usage of the system is, form the user's perspective [DFAB03].

c) Visualization: ***dynamic visualization***

Visualization refers to that part of a program which provides a display for the user – includes the screen, look and feel, the character encoding scheme and the font size – as well as giving room for the user to interact and control the system. In order words, it is a simplified view on the application, enabling and supporting users to adequately carry out their tasks.

Dynamic visualization refers to some form of advanced user interface that restricts, maintains and dynamically alters the viewed interface components in response to

some user action. This dynamically restricted view is based on object oriented some paradigms – inheritance, composition, associations – See figure 2.1.

d) Asset Model

The asset model is a new entity description scheme that merges two conventional ways of describing an entity, namely the conceptual modeling paradigm (model view) and the content modeling paradigm (media view). The idea behind this new model is that both conventional patterns need not exist in isolation, but go along together [SeSc04]. The conceptual model serves in describing properties, relationships and rules acting upon entities in the real world, while the content model serves as an existential proof of validity of the concepts.

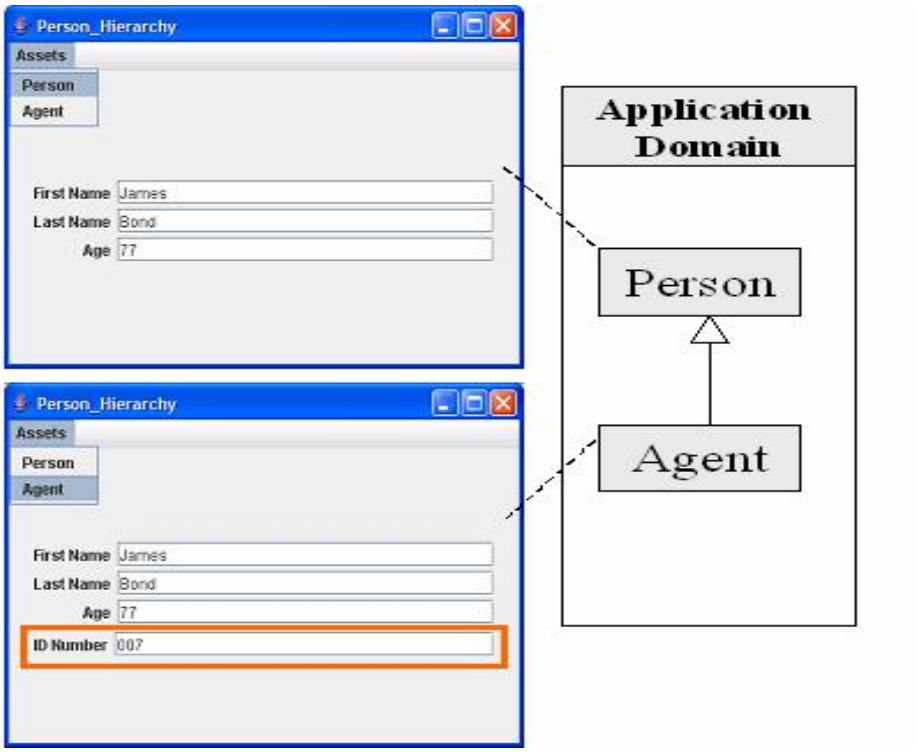


Figure 2.1 *Dynamic Visualization*

e) Conceptual Content Management Systems (*CCMSs*)

CCMSs are management systems that abstract the complex heterogeneous mix of media content – texts, images, maps, audio, video – and presenting them in a homogenous manner with the aid of domain specific conceptual models. They are geared at overcoming the difficulties faced by conventional CMS by redirecting and dealing with the problems at the meta-level. They are identified across two main properties [Sehr04]:

- Expressiveness
- Responsiveness

Expressiveness refers to the system's ability to allow for the representation of an entity in the real world to the maximum extend. This is achieved through usage on the asset model.

Responsiveness refers to the ability of the system to give room for modifications and for the effectuation of these modifications on-the-fly. This property is supported through *openness* and *dynamics* properties. Being conceptual in nature and acting like virtual machines, they are concretized (made real) by automation, using methods known as generative programming.

f) Openness

Openness is a property allowing asset models to be adapted according to the entity requirements at hand, and not being based on predefined ontology for concepts and categories.

g) Dynamics

Dynamics is a property allowing aspects of an asset model to be subject to inspection and adap-tation at any time. This means that changes made as a result of the openness property have to be effectuated on-the-fly.

h) Generative Programming

Generative programming is about incorporating the advantages of automation into system development. It tries to integrate object technology and domain engineering in order to provide an approach for systems generation. It heavily relies on the similarity or isomorphism – in the strict sense of the word – between domain models, which forms a basis for mapping from one system to another. It is a better approach to system development than OO in that it exploits all the advantages of OO, and at the same time resolves some of OO deficiencies by offering the advantage of robust system reuse and sets a good nice position to easily achieve scalability on software systems [CzEi00].

i) Virtual Machine

A virtual machine refers to an abstract, self-contained computing machine that behaves as if it was standalone. An evaluation function is implemented based on an instruction set. This normally requires the aid of a compiler [ABDM03].

2.3 Problems Faced by Conventional UIs

As businesses begin to grow, the demands on the application software increase causing the system to become more and more complex, less performing, and less scalable. This inefficiency at the level of the application domain has severe repercussions on the UI, namely the risks of:

- Inconsistency in data representation,
- Inflexibility to adapt to the changing environment,
- Low usability,
- Logical mismatch between the user's cognitive model[4] and the UI model making it difficult for the user to understand.

In order to find out what the possible causes of these problems are, it makes sense to understand how conventional systems are developed. Most systems are built following the principles of Object-Oriented (OO) Paradigm, introduced in the last decade. The main contributions [LiLa05] of this paradigm were to introduce the notion of:

- Classes (conceptual, and serves like a blue-print) and Objects (concrete instances of a class)
- Inheritance: allows for the reuse and extension of a class.
- Encapsulation: serves as a means to hide the inner structure of an object by just describing interfaces. This allows for the replacement of an object of one class by another object of a different class, both satisfying the same interface.
- Polymorphism and Dynamic binding: Polymorphism refers to the way in which objects respond to the same message in a different manner. Postponing the choice of the object type to be executed till runtime is called dynamic binding.

In as much as there are some advantages to be drawn out of OO, it is however limited in providing an efficient response to requirements like [Webs95]:

- Robust system reuse
- Dynamic adaptation to changing environment
- Scalability and increasing complexity

Due to the inefficiency in the OO paradigm, Generative Programming (GP) (see section 2.2) will be implored with the goal of superseding the limitations of the OO paradigm.

2.4 Rationale of the Study

The rationale of this study is not only to provide a UI for CCMSs but also to claim that UIs built on open, dynamic content management systems are better off than UIs for conventional systems. This claim is supported by the fact that CCMSs are open and dynamic, and thus can dynamically adapt to a changing environment. Furthermore, this claim becomes even more credible due to the fact that the visualization will be constructed using the ideology of GP paradigm [CzEi00], and therefore the inefficiencies faced by UIs on conventional systems will no longer exist. How this would be modeled is demonstrated in the next chapter.

2.5 Project Development Process Model

In order to move towards achieving the target, this section describes the process model chosen to guide the organization of objectives, activities of this study.
It furthermore provides a clear context and constraints the UI design process.

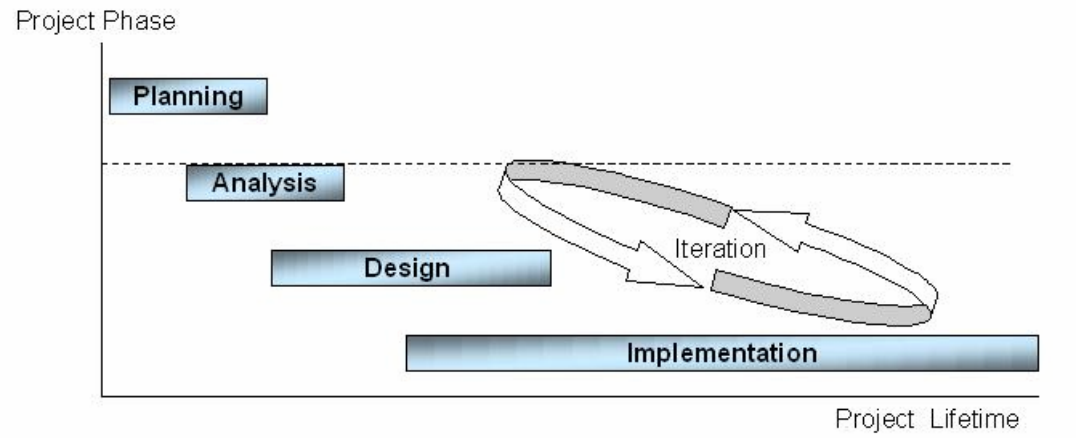


Figure 2.2 *Concurrent, iterative development process Model [Coll95]*

The concurrent, iterative design process model shown in figure 2.2 describes the evolutionary development of a software product by initially starting from a small portion to ever increasing lengths. Its greatest advantage is to uncover problems early enough and helps in avoiding to make faulty assumptions, which could lead to disastrous consequences in at the project end. It is suitable for this study because it is anticipated that certain steps may be done more than once, especially between the analysis phase and the implementation. This overlap in the phases is suitable for validating the deliverables of one phase, before it ends, in the next phase, hence avoiding any serious surprises [Coll95]. This process model is furthermore suitable for this study because it creates the chance reach the specification goals of the user who has difficulty expressing what he or she wants.

The planning phase has been dealt with up till now. Subsequent sections deal in greater depth with the analysis phase. The design and implementation phases follow suit in the next two chapters.

2.6 Requirements Analysis

Added to the requirements for:

- scalability to accommodate new UI Technologies
- easy-to-use,
- flexibility to changes,
- understandability,
- UI technology platform independence,

- flexibility in representing whatever component the UI-designer wants (*universality* in the Component domain)

emanating from the limitations of UIs for conventional systems, further requirements can be gotten by surveying of the environment and assessing the necessities of the various actors of the system. As a result of generative programming, there are two sets of actors that come into play.

Those acting at the *meta-system* level (generating code level), namely:

- UI domain modelers,
- Application domain experts, and
- Interface designers,

and those acting on the *generated visualization system*, namely the end users.

UI Domain Modelers have the goal of laying down building blocks for constructing UIs. Their role can be subdivided into more specialized roles like UI Component modeler and UI Technology modeler. Their main task on the meta-system is to define the asset-based UI-Model and to change it as the need arises. See figure 2.3 for UI domain modelers' use cases.

Application domain experts are the application developers. They have full knowledge and understanding of how the system works. They concerned with the so-called "black box" modeling i.e. everything, which end users and UI designers cannot see. This encompasses the entire framework set aside to offer all possible useful service to the user. They have a similar task on the meta-system as the UI designers since they define an asset-based application model and can change it as the need arises. See figure 2.3 for domain experts' use cases.

Interface designers are visualization experts. They posses ergonomic and graphic design skills and are responsible for defining the UI layout for the end users, based on models defined by the UI domain modeler and the application domain modeler (particularly specifying how an application asset should be visualized). See figure 2.3 for interface designer' use case. Their main task on the meta-system is to define the asset-based UI-Model and to change it as the need arises.

End users are those who interact and use the system to exploit its functionality. They are the targeted ones, whose needs and taste need to be satisfied. In addition, they want to display assets with the further constraint that the view on the assets are dynamically adaptable i.e. the view should vary depending on the asset being displayed. See figure 2.4 for end user's use case.

From figure 2.4, in order to fulfill the use case `Display Assets` two additional functionalities are required. One of these functionalities refers to the use case `Dynamically Adapt Asset View` with respect to the asset to be displayed (see definition for dynamic visualization in section 2.1). The second functionality refers to the use case `Controller Action` whose purpose is to evaluate constraints set on assets before they are displayed.

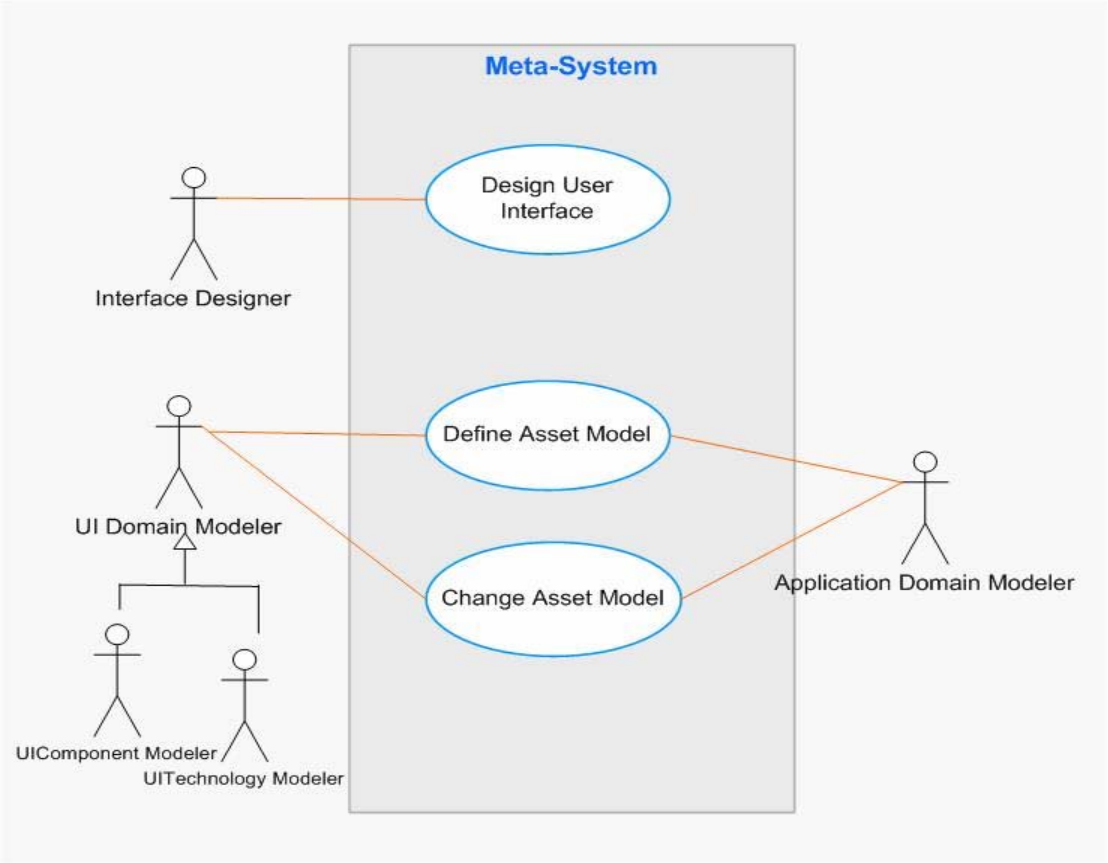


Figure 2.3 *Use Case diagram for Meta-System*

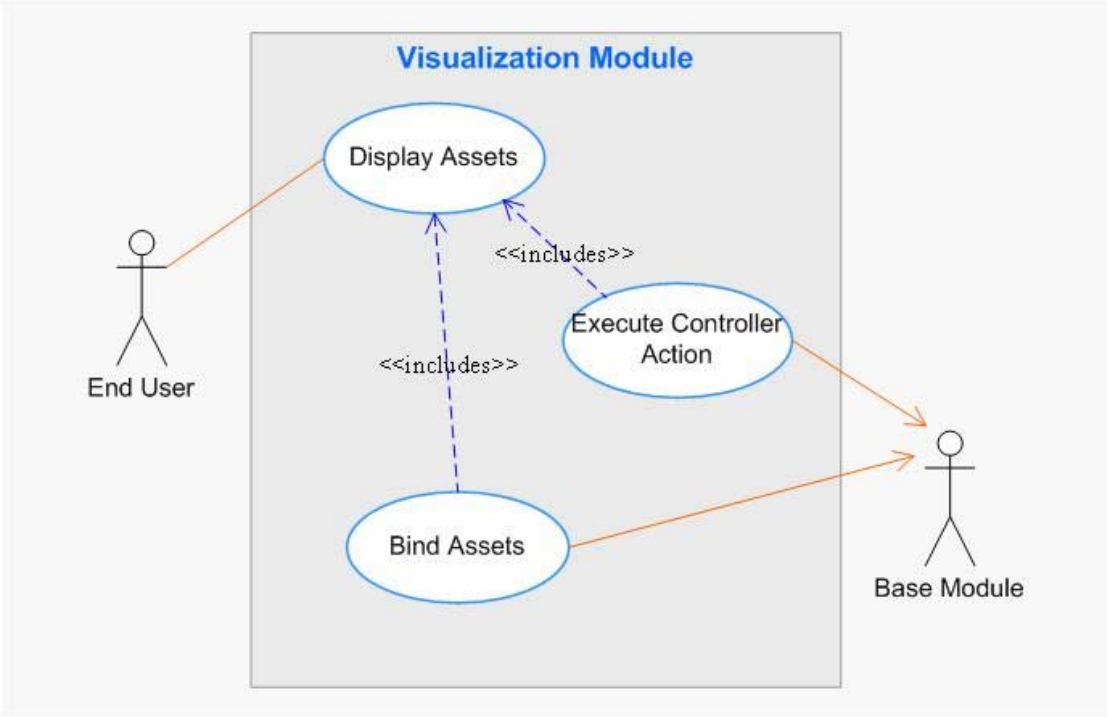


Figure 2.4 *Use Case diagram for generated Visualization Module*

Both of these use cases complete their tasks by forwarding their requests to underlying base modules (refers to all modules below the user interface in figure 1.2). Before moving on into processing ideas for the solving the above-mentioned requirements, the next section will give a formal overview of the CCM system.

2.7 The Conceptual Content Management System Environment

[SeSc04] supports the idea that the two classical ways of modeling entities, *content* wise and *concept*-wise, cannot exist in isolation but that they depend on each other, since they refer to the same entity. This idea gave birth to a new concept for entity description, called *assets*, geared at closely coupling content and concept. The process of managing these asset objects is called Conceptual Content Management, and a system based on the asset model is called a conceptual content management system (see figure 1.1).

2.7.1 Properties

These systems lay strong emphasizes on *openness* – meaning that a user can describe an entity as close as possible – and *dynamics* – meaning that model changes or adaptation can occur at any time, to reflect these changing scenarios. For example, entity descriptions are not static and cannot be valid all the time. The reasons for these are [Sehr04]:

- Entities keep on changing, and the context of the author changes as well.
- Entity descriptions are exchanged between users, which normally do not find themselves within the same context.

The asset language, Asset Description Language (ADL), is geared at satisfying requirements for *expressivity* [Peir31] and *responsiveness* [Cass02]. These requirements are being fulfilled by means of the system being open and dynamic.

2.7.2 ADL Compiler Based System Construction

CCMSs are created based on a model compiled by the ADL compiler, taking as input some asset definitions [Sehr04]. The compiler is designed as a framework and its basic structure follows the classical compiler architecture consisting of a front-end and a back-end, which communicate by exchanging some intermediate model. The front-end is in charge of lexing and parsing the asset definitions, as well as creating and checking the intermediate model. The backend on the other hand consists of an API generator and module generators. The compiler itself controls the order in which generators are run and the data flow between them. Domain experts formulate asset models using ADL.

This therefore means that for the generation of GUIs an asset based UI component description is needed, and based on these component descriptions an interface can be defined. The next section deals with some naïve design proposals.

2.8 Naïve Solution Proposals

In this section some pre-design analysis will be made towards solving the above requirements. This will serve as a stepping-stone on which the next chapter will build.

2.8.1 Expected Visualization Behavior

In order to achieve *understandability* it makes sense for the appearance of the view to reflect the underlying application domain model. The reason for this is that the underlying application domain is modeled following the object-oriented paradigm. Bearing in mind the OO paradigm is an approach that closely represents the relationship between entities in the real world, it therefore makes sense to have this concept reflected in the UI realm. This will mean that the model of the UI realm has to be isomorphic to the application domain model.

Figure 2.5 illustrates the correspondence relationship between the application domain and the UI realm. The arrows on the diagram do not conform to any particular annotation. They meant to emphasize the matching of view components in the UI realm to assets in the application domain. The starting point on this figure is the blue arrow showing a correspondence relationship between `Base Asset` (representing some current asset to be visualized) and the `Base View`. `Base View` represents a UI component that visualizes `Base Asset`. Due to the fact that the OO paradigm is used to model the application domain, other kinds of relationship with `Base Asset` are possible. For instance, `Inheritance Asset` refers to any asset from which `Base Asset` derives (in figure 2.1 the asset `Agent` can be seen as `Base Asset`, meanwhile the asset `Person` can be seen as `Inheritance Asset`).

Apart from the inheritance relationship other OO concepts like association and aggregation are reflected in `Association Asset` and `Aggregation Asset` respectively. These assets are visualized by the UI components `Association View` and `Aggregation View` respectively. That is, the additional view components (`Inheritance View`, `Association View` and `Aggregation View`) complement `Base View` while visualizing `Base Asset`.

Although the UI and the application realms look similar in structure, the relationships between the assets in the UI realm are not explicitly modeled as in the case of those in the application realm. That is, there is no real inheritance relationship between `Inheritance View` and `Base View` for example. The relationships in the UI realm are mere illusions that simulate existing relationships in the application model.

The advantages of this representation are the following:

- Ensures consistency of data representation in the visualization
- Dynamic displays can be carried out
- UI realm is open (not constrained), and flexible to display exactly any form of relationships an asset in the application domain may have.
- Easy to maintain due to the isomorphic match between both worlds
- Easy for the user to understand, because it conforms to his or her cognitive model

- Ensures scalability, in case the model becomes more and more complex

One brief remark is that figure 2.5 only shows *what* should be done to solve the understandability requirement. *How* this can be carried out will be treated under *Display Constraints* in the next chapter.

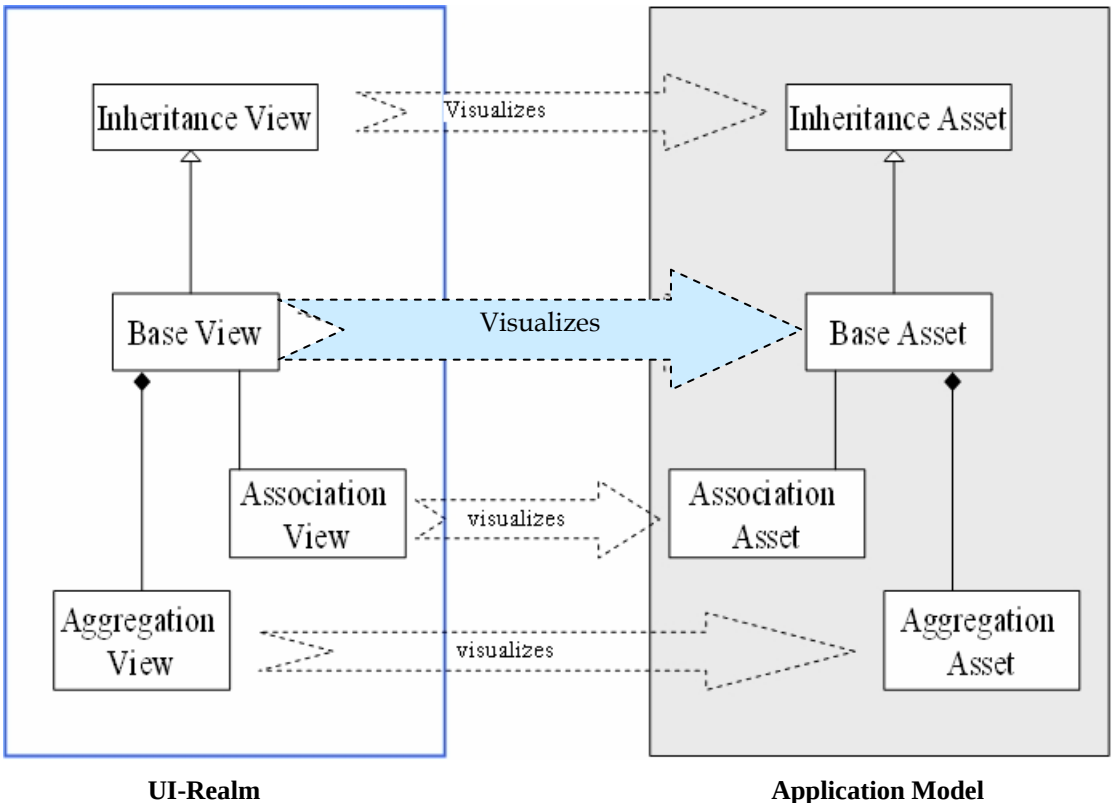


Figure 2.5 *Semantic relationship between the UI and Application Model*

2.8.2 The Need for Models and Separation of Concerns

The biggest hurdle relies on coming up with a suitable design for the UI realm, satisfying the stated requirements. The best way to go about it is to have things modeled separately. This way a modification arising from one model does not ripple off to the others. This will mean having separate models for the UI component realm, UI technology realm, and application domain. Furthermore these separate models need to be somehow glued together through some construct so as to have a unified model. It should furthermore be noted that for the purpose of modeling, there are three dimensions (spaces) involved:

- The user defines an asset based *interface (UI model Space),*
- The *generator* interprets this asset definition (Generator Space) and
- An equivalent *technology dependent code (Generated Code Space)* is generated, say in Java Swing. See figure 2.6

This role separation in the runtime process equally calls for separate thinking and modeling steps. This means that modeling will have to take place at the three different levels, as shown in Figure 2.6. The mode of thinking is reflected in chapter four and five (combines the generator and generated space).

A model is needed to describe all that the user needs to be able to define an asset-based interface. This will mean designing *UI component* model (for users to specify components) and a *UI technology* model (to reference an implementation technology). These models could be complex, but they however have to be constrained on making life easy for the user.

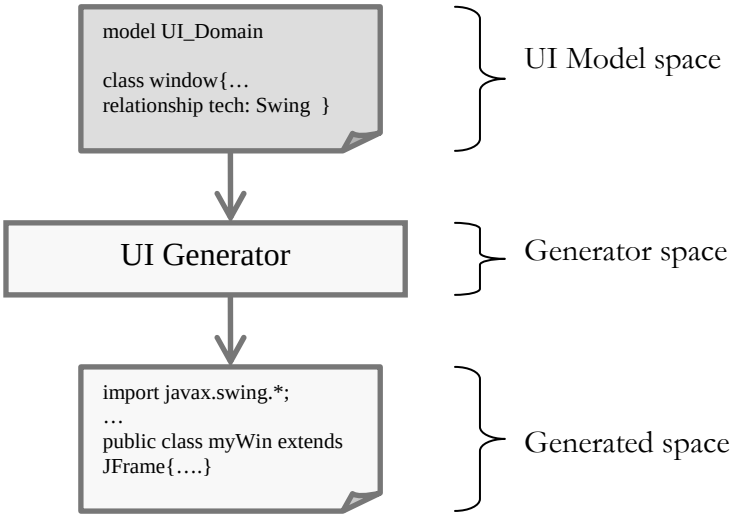


Figure 2.6 *Three-level modeling*

Chapter 3

Conceptual UI Model Design

The previous chapter explored some of the requirements that visualization for open and dynamic content management systems need to accomplish. These requirements came from various angles, especially from the various actors of the system. A survey of the development environment, conceptual content management system, was formally made and the advantages of using this environment were disclosed. The chapter ended by anticipating and making some naïve projections towards the solution. Some useful ideas regarding the technical designs were mentioned.

In this chapter, the ideas and requirements set by the previous chapter will serve as a foundation for coming up with concrete designs, satisfying all those requirements and serving as a basis on which visualization dynamics can be achieved. The way in which assets have to be presented has to be user-definable. The formal goals for the model designs are therefore for them to be easy-to-use and to provide room for scalability.

3.1 Modeling the UI Space

This section will strive at coming up with designs for the UI model space portion of figure 2.6. Full dynamics can be achieved when the UI-realm (from figure 2.5) is not limited in representing the model expressed in the application realm. This means that the argument in favor of isomorphism between the expressivity of both domains should hold. This furthermore means that the UI component model has to be as generic as possible such that full expressivity is attained.

The second open issue is to conceptually define existing UI-Technologies (concrete ones like *AWT*, *SWING*, *HTML*, *XHTML* etc). One of the requirements was to have the model independent from these technologies. By employing generative programming [CzEi00] technology platform independence can be easily supported. One major consequence of using generative programming is that families with similar characteristics can be being abstracted and treated in the same manner. This therefore necessitates a careful modeling of the presentation technologies.

3.2 Visualization Construction Scenario

Appealing back to the use case diagrams of figures 2.3 and 2.4, this section will examine in detail what the individual actors of the system do. From figure 3.1, the ultimate goal of the modeling is for the interface designer to be able to specify a layout model, which an end user can visualize. For this to happen, the interface designer needs to know how widgets are defined, which technologies are available and how the application domain is modeled. This last piece of information is necessary for the interface designer to be able to associate an asset from the application domain with widgets specified by the UI modeler.

Of special interest is the case of the UI modeler because his duty forms the core of the whole visualization process. This role faces the problems of:

- Modeling the components which can be visualized
- Conceptually representing concrete UI technologies, and
- Providing a conceptual implementation for the displayable components.

The design of his model is inspired by Albert Einstein's idea, quoted in [CoB05], stating that everything should be made as simple as possible but no simpler. This idea is vital for the abstraction of the concrete UI Technologies shown in the bottom-most layer. A further rationale behind this design is to employ the benefits behind layered architectural design [SJT05], whereby every layer is seen as an abstract machine satisfying the principles of locality and information hiding, and furthermore relies on the underlying layers for its implementation.

The outcome of this design concept is visible on figure 3.1. The idea of separation of concerns is exploited. The UI modeler provides three separate files (1,2,3) with the intention that modifications made on one may not ripple off to the others. This design structure already fulfils some of the requirements from section 2.6 in that:

- Flexibility towards changes is offered
- Scalability is supported from the use of the layered design structure
- UI technology independence is obtained because the interface designer can switch at any moment from one technology implementation to the other using the same UI components

The next section will delve into the design models for `UIComponent Model` (see number 1 in figure 3.1) and `UITechnology Model` (see number 2 in figure 3.1). The `ComponentImplementation Model` (number 3) will be discussed in the next chapter.

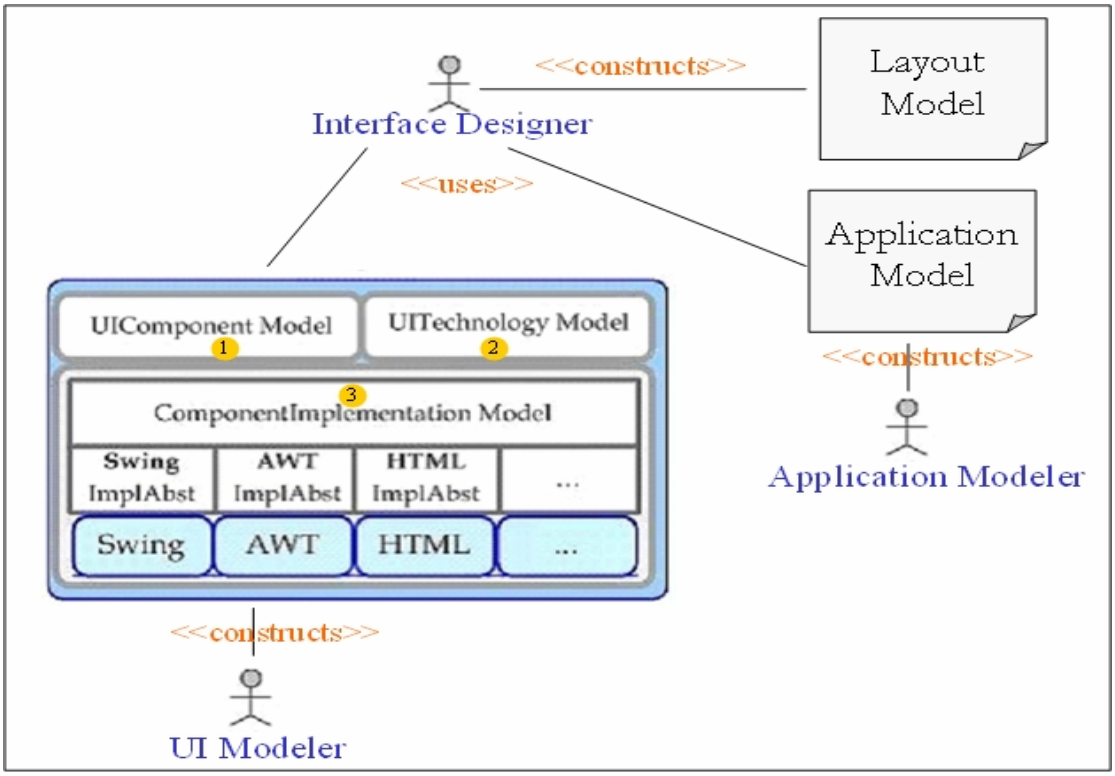


Figure 3.1 *Visualization Construction Scene*

3.2.1 Logical UI Component Domain Model

A generally employed pattern when designing and implementing user interfaces is the Model-View-Controller (MVC) [GoF95]. This section discusses the ideology behind the pattern and subsequently moves on to apply the MVC concept to modeling of conceptual UIs.

3.2.1.1 Model-View-Controller

This domain model follows the standard pattern for modeling GUIs, namely the MVC [GoF95]. This modeling pattern emphasizes on the separation of concerns stating three different parts [BMRS+98]:

i. Model

The model maintains core functionality and data. It represents an abstraction of the process of the real world and functions as a computational representation.

ii. View

The view displays information to the user, in the form of graphics, on a device. This view is usually linked to one display surface and knows how to render it. The view knows its model and renders its content on the display.

iii. Controller

The controller translates user input actions, say from the mouse or keyboard, into commands sent either to the model or the view to carryout some changes.

The modeling of the basic interaction between the three units is shown in figure 3.2. This interaction can be much more complex, depending on the scenario. The figure states that the *view* has a tight coupling to the model – it knows the exact type – and a weak coupling to the controller, giving room for polymorphism.

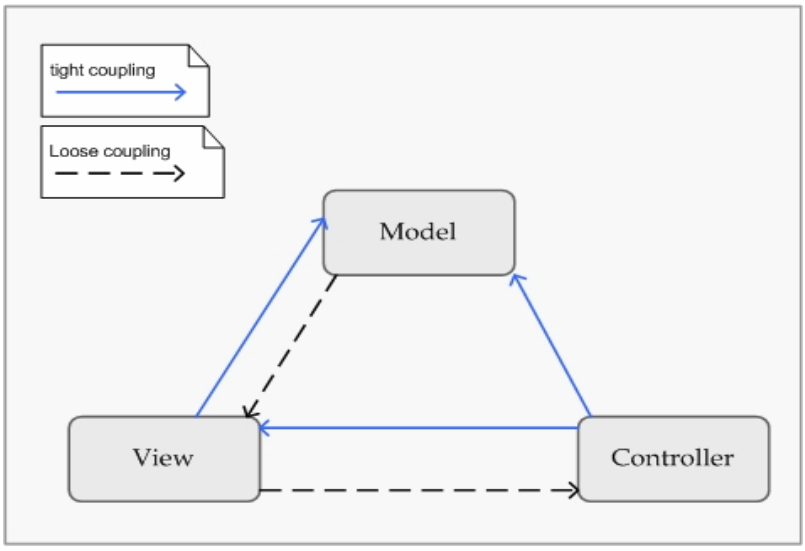


Figure 3.2 *Model-View Controller*

The *model* on the other hand has no direct communication with the controller, but has a weak coupling to the view, emanating from the fact that a model can have several views. Finally, the controller – acting as an intermediary – has a strong coupling to model and the view. The precision is necessary for the controller to be able to carryout specialized actions.

For the sake of simplicity, the modeling of the controller will not be part of the implementation scope of this work. The main focus will be on achieving the *view-model* relationship. The next section deals with how this can be done conceptually.

3.2.1.2 UI Component Model

Geared at satisfying the goals of the MVC, with particular attention on the view-model relationship, figure 3.3 has been conceived. This figure is a revised version of that proposed by [Xu04]. The main goal behind the revision was to achieve a maximum of flexibility as possible, by avoiding constraining the model. At the same time, care is taken not to give too much room for expressivity, which may lead to a lot of ambiguity.

At the root of the component hierarchy is the `UIComponent` class. This class has the role of modeling every basic property that a UI component can have. These properties include:

a) Model

Model here refers to the data abstraction to be displayed. Not every UI component has to be linked to a model, hence the reason for the cardinality *0..1*. Every model has a type, `ModelType`, associated to it.

b) Controller

The controller models the role of the controller in the MVC pattern (see figure 3.2). The diagram shows how this unit is aware of the model and the view(s).

c) Layout Parameter

Layout parameter represents the idea of geometrically positioning a component on the display screen. There are several ways to go about this, which can be narrowed down to two broad types: *absolute positioning* or *relative positioning* (usually with respect to some fixed component). Though introducing this component is necessary from a modeling point of view, its implementation is beyond the scope of this work. Also, the modeling of this component is an integral part of the functionality of the *Layout Manager*. For modeling simplicity, every `UIComponent` can be subdivided into three broad categories:

i. AssetViewComponent

The asset class `AssetViewComponent` models all those `UIComponent` the play the role of the view in the MVC pattern. Their role is only to display the contents of the model, and therefore strongly make use of the navigation link to the model at the `UIComponent` level. Examples are *textfields, labels* and *tables*, just to name a few.

ii. ActiveComponent

These are components whose primary role is to serve as a medium executing actions on the display. They have a strong use of the navigation link to the controller, because they initiate and exhibit controller behavior. They have the special property of not being bound to any model, hence making a very weak use of the model navigation link at the `UIComponent` level. Should `ActiveComponents`, however, require a

partial reference the model (e.g. buttons or labels that display text coming from the model), then they are modeled as a `PsuedoViewComponent` (see figure 3.3). That is, a `PsuedoViewComponent` is an active component that exhibits view behavior at the same time.

d) UI Container

This models the set of `UIComponents` that hold other `UIComponents`. Example, a window holding a label and a textfield. For the arrangement of these components within itself, the `UIContainer` may have a `LayoutManager`.

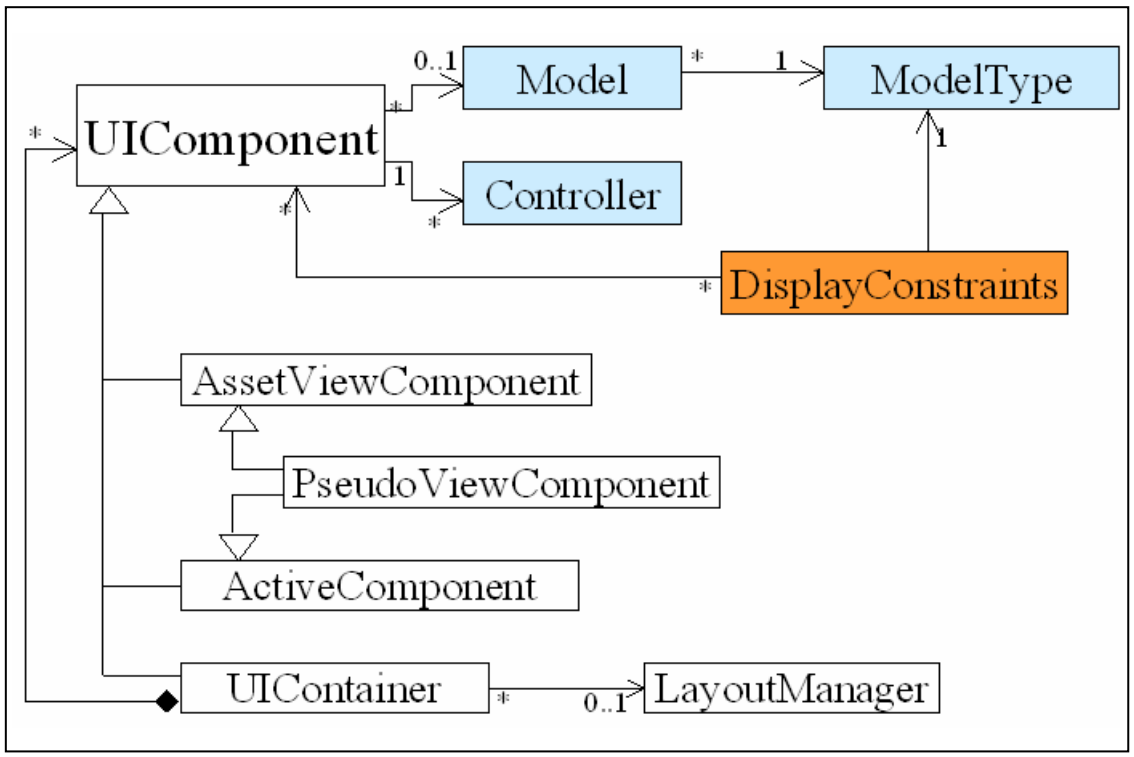


Figure 3.3 *UI Component Model*

The `LayoutManager` is delegated the responsibility of optimally arranging the components within the container. It gets as input some `LayoutParameter` information, which it is interpreted in a relative positioning fashion. How this correspondence is achieved, will be discussed in section 3.2.1.3.

3.2.1.3 UI Container Layout Modeling

As introduced in the previous section, it does not only suffice to be able to add `UIComponents` to a `UIContainer`, special considerations need to be taken to ensure where these added components have to be placed. For this to happen, a special way of relating the `UIComponents` to a `LayoutParameter` is needed. Several of these modeling alternatives are proposed below in figure 3.3.

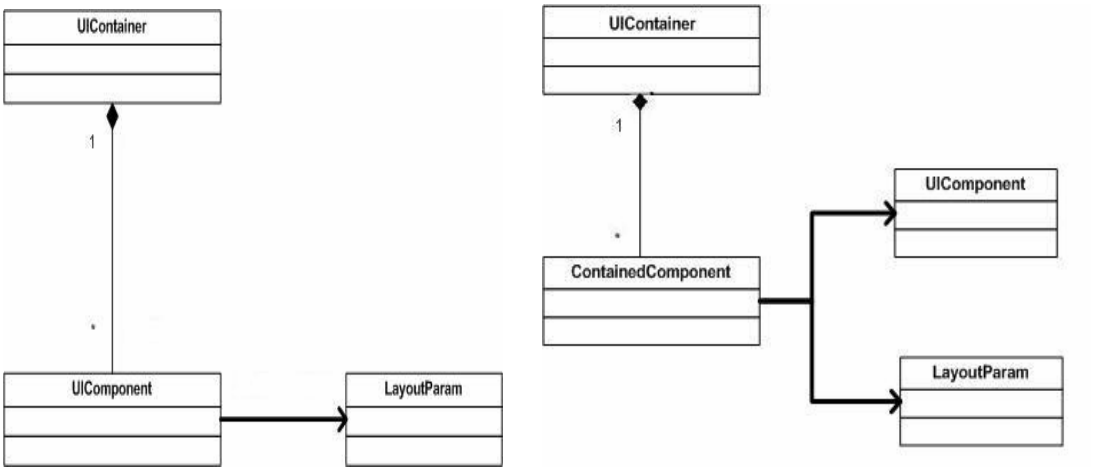
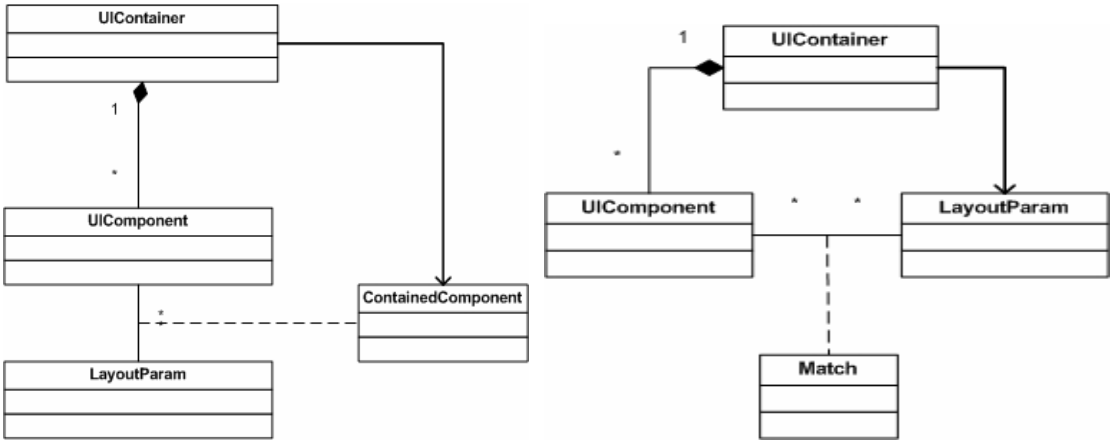


Figure 3.4 a) *Aggregate UIComponent* b) *Aggregate ContainedComponent*



c) *Aggregate UIComponent with ContainedComponent reference*

d) *Aggregate UIComponent with Layout Parameter reference*

a) Aggregate UIComponent

In the aggregate UIComponent style**,** the `UIContainer` knows all its Children. And the children are modeled with a Layout Parameter at definition time. Its main advantage is that it is easy for the user to use and apply. However, this model implies that every time a `UIComponent` is defined, a layout parameter has to be given, even without being in the context of being within a `UIContainer`. From a pragmatic point of view, therefore, it is weak as the user is forced from the onset to give the layout parameter without being in the context of the container. This may be prone to errors, as these components could be later on assigned to the wrong `UIContainer`.

b) Aggregate ContainedComponent

The aggregate ContainedComponent style tries to overcome the limitation of the previous style by enforcing that the thought about the layout parameter should only be

within the context of the UIContainer. This is done by introducing an intermediary class, `ContainedComponent`, which is delegated the responsibility of relating every `UIcomponent` child to its layout parameter. The advantage here is that the user's thought is confined to a particular `UIContainer`. On the other hand, this way of modeling is not in line with the user's way of thinking, since for every UIComponent to be modeled the user always has to construct a `ContainedComponent` asset class. This could be perceived as incomprehensive to the user.

c) Aggregate UIComponent with ContainedComponent reference

The aggregate UIComponent with ContainedComponent reference style tries to reduce the drawback in b), by making the `UIComponents` visible within the `UIContainer` (good for the user's understanding), as well as making sure that the thought on the layout parameter is confined only to the context of the `UIContainer`, by introducing the `ContainedComponent` class, which serves as a point of linkage between the `UIComponent` and the `LayoutParameter`. This model brings in some comprehension to the user's way of thinking but still poses some obstruction in the user's thought because the `ContainedComponent` concept is not familiar for users.

d) Aggregate UIComponent with Layout Parameter reference

The Aggregate UIComponent with Layout Parameter reference style serves as solution to both problems faced in b) and c). The modeling of layout parameter is confined to the context of the `UIContainer` and the user can relate the `UIComponents` to the `UIContainer`. Since both layout parameter and `UIComponent` live in isolation, they however need to be related together. This is taken care of by a class called `Match` class. In other words, what the asset class `Match` does is to associate UIComponents with their corresponding layout parameters. Although this model falls in line with the user's way of thinking and can be considered the most convenient from a design and pragmatic perspective, it however gives the user additional work related to defining the match class.

Bearing in mind that one of the main goals is to make he UI definition user-friendly, choice of figure 3.4a) is the best alternative, because it is simple for the user to use. Although the other models are suitable design alternatives, they either come at the cost of making life difficult for the user to understand (models b) and c) in figure 3.4) or by increasing complexity (model d) in figure 3.4).

3.2.2 Logical Technology Domain Model

The diagram in figure 3.5 shows a classification of existing UI Technologies. It has mainly a hierarchical structure with a wide range of application ranging from internet technologies to standalone ones.

The purpose of this work will be to be able to conceptualize this general hierarchical structure in an asset model. Furthermore, this conceptualization will be limited to the family of *ToolkitsAndUILibraries* , and in particular targeting the Swing technology.

The goal of this section is to come up with an approach to describe the inheritance scheme shown in figure 3.5.

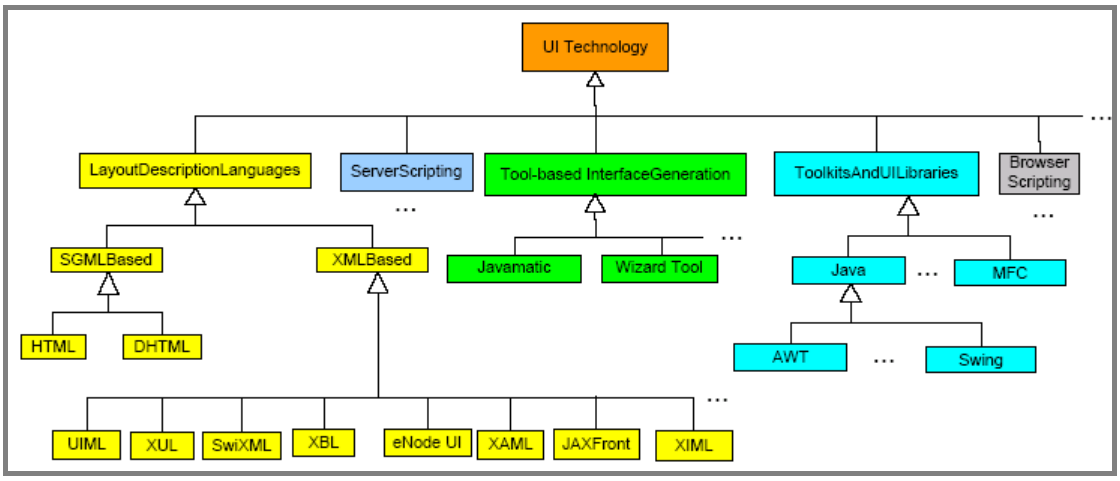


Figure 3.5 *Visualization Technologies Diagram [Xu04]*

For this purpose two models (see figure 3.6 a & b) are proposed. The methods on the class diagram of figure 3.6 make reference methods generated based on a code generation toolkit [Sehr06].

3.2.2.1 Object-based Technology Model

In this model, every technology element in figure 3.5 is assumed to have the same basic structure. The model can be seen in the light of an object system in which there are no classes (except the original prototype class) but instead data and code are encapsulated inside objects.

The advantage that this model offers is that it is easier and faster to implement in the sense that one only focuses on the behavior of some small set of technologies and only worrying about classifying them later on.

On the other hand, the major disadvantages are its inflexibility to control behavior from a single point (since every object encapsulates its own data and code), the absence of behavior reuse (inheritance can be carried out explicitly by delegation [Wegn87]), and the difficulty in exhibiting polymorphic behavior.

3.2.2.2 Class-based Technology Model

Figure 3.6b shows the second approach towards modeling the technology structure of figure 3.5. This approach works in line with the class-based paradigm, whereby the classes provide the basic structure and behavior on the one hand, and the instances maintain state information on the other hand. The advantages of this paradigm are:

- Flexibility in introducing a new UI technology family, since it primarily focuses on the taxonomy and relationships between classes [Wegn87].
- Behavior reuse is given for free
- Polymorphism can be employed, and
- Instance control is easy (from the class)

Considering the differences between both models, this study chooses to implement the class-based paradigm in order to reap its advantages.

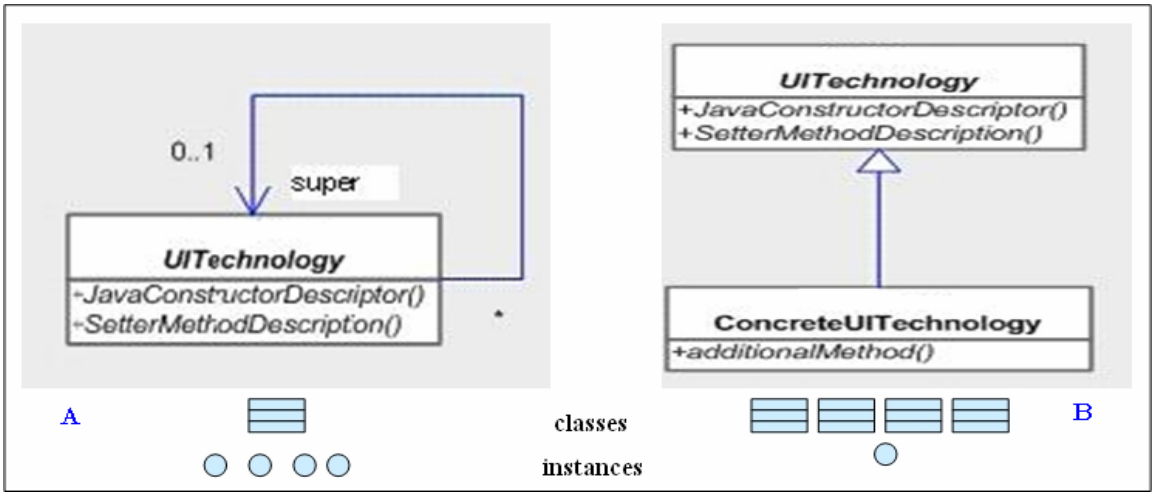


Figure 3.6 a) *Object-based Technology model* b) *Class-based Technology model*

3.6 Linkage patterns between UI Components and UI Technologies

The remaining issue is to allow the user to specify a component and an implementation technology. For this, there are three alternatives shown in figure 3.7.

a) Diagram A

The generator is modeled to receive a single input. This model is based on the fact that all components are implemented using a single input technology reference.

b) Diagram B

This diagram gives more flexibility in the choice of the technologies per component. Every component has the flexibility to specify it own implementation technology, thus giving room for mixing of technologies.

c) Diagram C

This model is similar to that on diagram 3.7A, the only difference being that the generator receives two separate inputs, one specifying the interface definition and the other specifying the implementation technology of the interface definition.

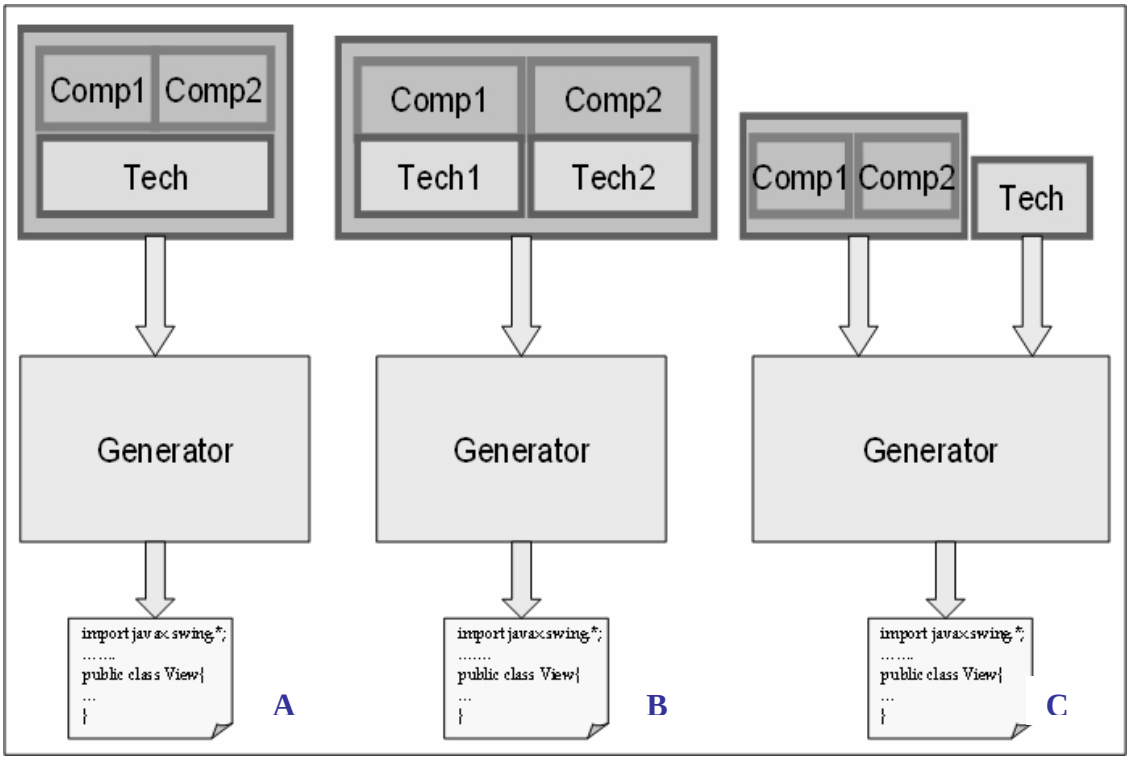


Figure 3.7 *Variations of Component and Technology Linkages*

Based on the flexibility criteria, the model proposed in diagram 3.7B is better that 3.7C, since it gives room for exploiting the advantages of other technology, when mixing them together.

3.4 Asset Binding

In order to fulfill the final requirement for dynamic view adaptation, a new kind of relationship is introduced. The idea behind this new relationship is to be able to formally specify which components should be used to view an application asset. This formal specification is followed at runtime by a concrete evaluation of instances satisfying the formal specification. The concept is similar the notion of formal and actual parameters in function definition. Refer to figure 3.8 for asset binding representation. The figure uses the conceptual graph notation [Sowa76] for representation. The main reason for choosing this form of annotation is because it offers logical preciseness and it is humanly readable.

3.4.1 Formal Binding

Formal binding is represented to the part above the dotted line in figure 3.8. There are two paths interpretation section following the arrows. The topmost path (refer to number 1 in figure 3.8) relates an asset from the displayable Component model to an `AssetClass` in the application domain, meaning that an instance of the referred `AssetClass` as a whole can be visualized by a set of UI-components.

Since the concept-part of an asset class is defined in terms of characteristics and relationships (see [Sehr04] for asset language specification), one could have a separate view for each `Characteristic` and `relationship` and have the entire view of the asset class tailored by the individual `Characteristic` and `relationship` view representations. This more refined representation is depicted on the path number 2 in figure 3.8.

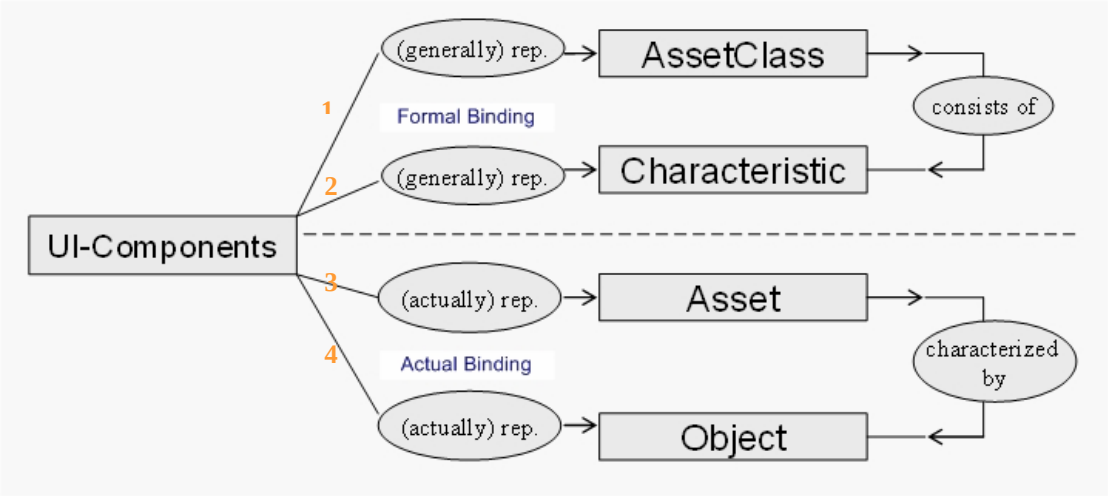


Figure 3.8 *Asset Binding* [Sehr04]

3.4.2 Actual Binding

Actual binding is represented in the part below the dotted line in figure 3.8. Two paths can also be identified here (number 3 and 4 in figure 3.8), respective to those in the formal binding. The only difference here is that one evaluates instances of the type specified in the actual bindings (compared to value passing).

The way of implementing this asset binding is by introducing the class called `DisplayContraints` in figure 3.3. This class acts like an association class linking a `UIComponent` to a `ModelType`. The advantage of this design, as opposed to one in which every `Model` by definition refers to UI-components for its view, is that both sides are decoupled. This enhances reuse of UI-components.

`DisplayContraints` (see figure 3.9 for display constraint example) exhibit some kind of modality behavior in the sense that based on their role in acting as an association class linking UI-components and assets from the application domain, they can constrain behavior by displaying only those UI-components for which an actual binding exists. This type checking evaluates relationships the asset to be visualized has with others based on the OO concepts of inheritance, association and aggregation between asset (solution to figure 2.5).

Figure 3.9 shows an example of the application of `DisplayConstraints`. To the right of the figure one sees assets from the application domain being visualized by a subset of UI components, on the left, across `DisplayContraints`. The

application `AssetClass Person` is visualized by a `Label` and a `TextField`. Likewise the `AssetClass Student` is visualized by a `Label` and a `TextField` (representing different information from that of `Person`).

From the UI-components point of view, `Person` and `Student` are two independent classes but the DisplayConstraints class, however, has information about the inheritance dependency between both. This permits that during the display of `Student` the `Label` and `TextField` pertaining to `Person` be displayed as well, since a `Student` is fully described with the inherited representation of a `Person`. It is the same idea behind figure 2.1.

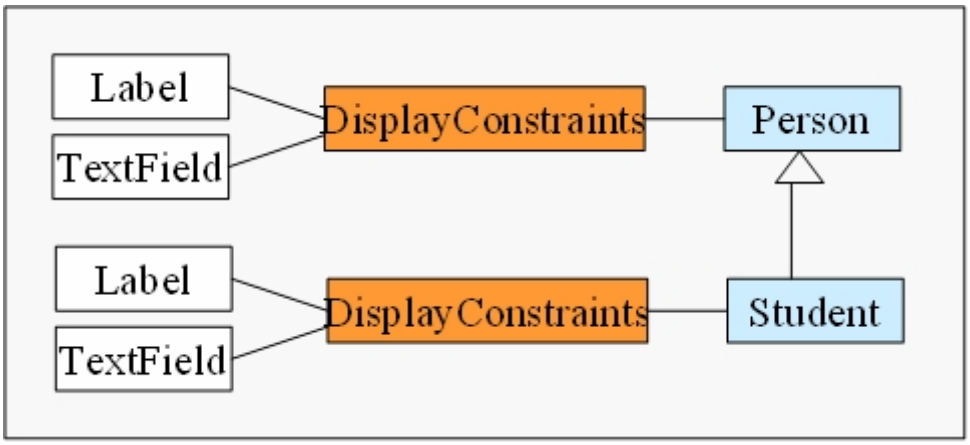


Figure 3.9 *Display Constraints Example*

3.5 UI Generation Process

This section is to summarize the entire design goal for the visualization. Figure 3.10 shows that in order to achieve the ultimate goal of view adaptation, two steps need to be superposed.

3.5.1 UI Openness

UI openness is the first step in figure 3.10 and its main task is to define the interface and to encode the way in which information (assets) has to be visualized [Chan04]. The background of this encoding follows the semiotics of Charles Peirce [Peir31], where he claims that a set of signs and symbols create meaning. Meaning in this case is won by associating every application asset with UI-components which visualize it. This association is implored through usage of `DisplayConstraints`.

3.5.2 UI Dynamics

After the UI openness property comes UI dynamics. The changes made by the user have to go into effect on-the-fly, and this accounts for the system dynamics. With the aid of a compiler the changes are integrated as shown on the diagram in figure 3.10.

3.5.3 Visualization Modality

Visualization modality refers to the way in which information structured and displayed to an end user. This structuring into modes is encoded in the first stage.
This phase supports three sub-processes:

- Sub-processes 1 and 2: UI evolution
- Sub-process 3: View adaptation

Sub-processes 1 and 2 account for the evolution of the user interface on model changes resulting from the openness and dynamics property of the system.

Sub-processes 3 depicts the intended view dynamics in that there is a change in the view state on instance selection. This is the effect of imploring `DisplayConstraints.`

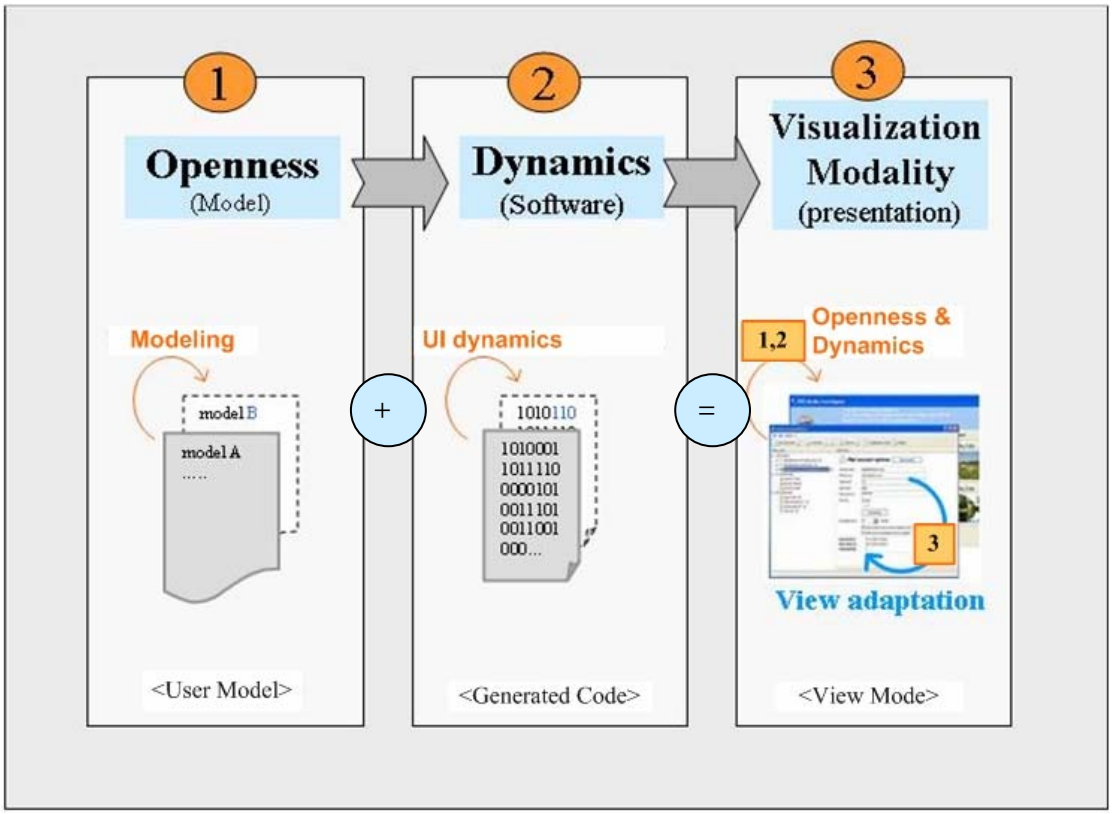


Figure 3.10 *UI generation Process*

Chapter 4

Conceptual Model Implementation

The previous chapter went through some design issues relevant for the implementation of a user interface for Conceptual Content Management Systems. A system process model was introduced on how the system should behave during runtime (figure 3.10). This chapter will delve into the implementation of the design decisions made in the previous chapter. Special attention is be paid to:

- How the interface definition given by the user would look like
- How the view adaptation based on assets can be realized.

4.1 UI Modeler's Code

This section gives a brief overview on sample implementations of the task of the UI modeler in the visualization construction scene presented in figure 3.1. The specification language used here is the ADL.

4.1.1 Component Model Description

This section refers to the implementation of number 1 in the UI modeler's design (figure 3.1). It is one of the two units visible to the UI designer and deals with the description of displayable components by the user. The implementation of this component model follows exactly the design presented in figure 3.3. So far, the elements set of the `Component` model consists of the union of all widgets present in the underlying visualization technologies. This means that the description to any one of these `UIComponents` is justified by the fact that it can be mapped on at least one existing target technology. See code example 4.1 for a sample description of the root components `AssetViewComponent`, `ActiveComponent`, and `UIContainer`. How these components are mapped onto concrete technologies is discussed in more detail in section 4.1.3.

```
model CompenentModel
from AssetModel import Attribute

class UIComponent{
    concept
     characteristic name       : String
     characteristic visibility : boolean
     characteristic height     : int
     characteristic width      : int
     relationship   modelattr  : Attribute
};UIComponent
class AssetViewComponent refines UIComponent

class Action{
     concept
          characteristic name             : String
          characteristic mnemonickey      : String
          characteristic actioncmdkey     : String
}
class ActiveComponent refines UIComponent{
    concept
     relationship a : Action
}; ActiveComponent

class UIContainer refines UIComponent{
     concept
          relationship lytMgr  : java.awt.LayoutManager
};UIContainer
```

Code Example 4.1 *Component Model*

An interesting part of the code example 4.1 is the relationship attribute `modelattr` of `UIComponent`. In order to avoid static binding of values on `UIComponents`, this attribute is used to indicate that there is an attribute which requires value update on display. In other words this attribute offers support for dynamic display by giving room for dynamic binding of instance values of application assets to `UIcomponents`, according to the

`DisplayConstraints` definition. That is, different values can be inserted on view components at runtime, given the fact that instances carry these values of a particular type (based on `DisplayConstraints`). Section 4.4 demonstrates the usage of the `modelattr` in an example.

For the sake of simplicity, only a few components were modeled in this study. See code example 4.2 and 4.3 for a sample implementation of a `Label` and `Button` for the categories `AssetViewComponent` and `ActiveComponent` respectively. One special property of `ActiveComponents` is that they initiate an action, hence their strong relationship with the `Action` class.

```
;;;;;;AssetViewComponents
class Icon
class Label refines AssetViewComponent{
     concept
          characteristic text  : String
          characteristic image :Icon
}
class TextField refines AssetViewComponent{
     concept
          characteristic text    : String
          characteristic columns : int
}
```

Code Example 4.2 *AssetViewComponents*

```
;;;;;;;;;;;;;;;;;;ActiveComponents

class Button refines ActiveComponent{
     concept
          characteristic label     : String
          relationship   i         : Icon
}
class MenuElement refines UIComponent;
class MenuItem refines MenuElement{
     concept
          characteristic mnemonic  : int
          relationship   i         : Icon
}
class Menu refines ActiveComponent, MenuElement{
     concept
          relationship   menuelms  : MenuElement*
}
```

Code Example 4.3 *ActiveComponents*

A bit tricky is the implementation for `UIContainer` components because they are characterized by the fact that they are usually structured into several different sub-sections. These different subsections contain different elements and it would be nice to have an ordered

way of distinguishing in which subsection a UIcomponent belongs. Code example 4.4 gives a sample implementation for the `Window` component. More on this can be view in appendix A.

```
;;;;;;;;;;;;Container Classes

class MenuBar refines UIContainer, MenuElement{
     concept
          characteristic comps     : UIComponent*
          relationship   menus     : Menu*
}

class ContentPane refines UIContainer{
     concept
          characteristic comps     : UIComponent*
}
class Window refines UIContainer{
     concept
          characteristic title          : String
          relationship   mb             : MenuBar
          relationship   contentpane    : ContentPane
};Window

class Panel refines UIContainer{
     concept
          characteristic comps     : UIComponent*
}
```

Code Example 4.4 *Container Implementation*

4.1.2 Technology Description Model

This section will refer to the implementation of number 2 in figure 3.1 referring to the UI modeler's design. The technology description model is the second part which is visible to the user. The technology model has the task of dealing with the description of the concrete visualization technologies. The implementation of this technology model follows the choice made in the previous chapter (figure 3.6 b). This way the user can choose in which technology the components will be implemented. Code example 4.5 gives a sample implementation for `Java technologies.`

```
model TechnologyModel

class UITechnology{
     concept
          characteristic name : String
};UITechnology
class Java refines UITechnology
class Swing refines Java
class Awt refines Java
```

Code Example 4.5 *Technology Model implementation*

4.1.3 Component Implementation Description Model

This section refers to number 3 in figure 3.1. It is invisible from the UI designer's perspective and serves as an abstraction of the component implementation in the concrete technologies. The reason for this invisibility is to decouple the implementation from the component definition. This allows for modification of the implementation without affecting the design of the interface. Its special focuses are on:

- Expressing a way to specify the mapping of one component from the `UIComponent` model too an equivalent one on the target platform (through the combined reference of `jClass` and `technology` in code example 4.6),
- Specifying a general description for constructors in the underlying technologies, based on the fact that not all components have a default constructor, and also on
- Describing *setter methods* (refer to `setterMethods` in code example 4.6) for extra attributes supplied by the user that cannot be initialized by the constructor definition.

```
model ComponentImplModel

from AssetModel import Member
from ComponentModel import UIComponent
from TechnologyModel import UITechnology


class UIComponentImpl{
  concept
     characteristic jClass     : java.lang.Class
            relationship   component  : UIComponent
     relationship   technology : UITechnology
     relationship constructors : JavaConstructorDescription*
     relationship setterMethods: JavaMethodDescription*
};UIComponentImpl


class JavaConstructorDescription{
     concept
          characteristic paramTypes   : java.lang.Class[]
          relationship componentAttributes : Member*
};JavaConstructorDescription

class JavaMethodDescription{
     concept
          characteristic methodName   : java.lang.String
          characteristic paramTypes   : java.lang.Class[]
          relationship   componentAttributes : Member*
};JavaMethodDescription
```

Code Example 4.6 *ComponentImplementation Abstraction*

4.1.4 Swing Implementation Abstraction Description Model

This unit is situated one level below the Component Implementation Model of figure 3.1and describes a focused implementation of the components defined in the Component Model for specific target technologies, in this case for the Swing technology. This entails specifying which constructors and setter methods to use for a particular component. Only those components from the Component model having a corresponding target technology representation are being considered here. For example, the code example 4.7 shows how the component Label is mapped onto `JLabel` for a Swing implementation, and that the constructor with setting the *text* attribute has to be used, and finally that, in case an icon is specified, the `setIcon` method should be used. Reflecting back on the attribute `modelattr` in code example 4.1, the implementation should specify exactly which attribute needs to be constantly updated by referring to its setter method. From code example 4.7, the value of any attribute associated with `modelattr` updates the `text` attribute of `TextField.` This way, the generator program can draw a correlation between the attribute on the component that needs its value constantly updated with the present application asset instance at hand and must also ensure that this value is of the required parameter type for the setter method. In the case of code example 4.7 the type of the attribute referred to by `modelattr` must be a String, if not the generator throws an exception.

```
model SwingTechImplModel
from ComponentModel import Label, TextField, SplitPane,
                       ScrollBar, Menu, Button, Window, Panel
from ComponentImplModel import UIComponentImpl,
          JavaConstructorDescription, JavaMethodDescription
from TechnologyModel    import Swing

;;;;;;;;;;TextFieldIMPL
let swing := create Swing{name="Swing"}
let swingTextFieldImpl := create UIComponentImpl{
     jClass := javax.swing.JTextField.class
                     component := TextField
     tech := swing
     constructors := {create JavaConstructorDescription{
               parameterTypes:= new Class[]{ String.class }
               componentAttributes := { TextField.text}
                       }
                     };constructors
     setterMethods := { create JavaMethodDescription{
                methodName:= "setColumns"
                parameterTypes := new Class[] { Integer.type}
                componentAttributes := {TextField.columns}
                       } create JavaMethodDescription{
                methodName:= "setText"
                parameterTypes := new Class[] { String.class}
                componentAttributes := {TextField.modelattr}
           }}};setterMethods
};c. swingUIComponentImpl
```

Code Example 4.7 *Swing Technology Implementation Model*

4.2 Application Domain Model

The application domain modeler is responsible for implementing this model. Present here are the asset classes to be visualized, together with the relationships between one another. Code example 4.8 shows a sample implementation.

```
model ApplicationModel

class Person {
                              concept
          characteristic name : String
}
class Student refines Person{
                              concept
          characteristic matrikel: int
}
```

Code Example 4.8 *Application Domain Model*

4.3 Display Constraints Implementation

In view of the request for users to be able to dynamically update the presentation depending on the asset class being presented, an asset binding was proposed in section 3.5. This asset binding was designed to be realized through the `DisplayConstraints` class of figure 3.2. With a model implementation for this class, the user can then define the formal bindings. A sample implementation is shown in code example 4.9. The contribution of this class is that at runtime its instances are evaluated and only those UIs will be displayed for which there is an actual binding. It also serves as a starting point for the generator to display additional relationship information due to inheritance or associations to other classes, as proposed in (figure 2.4). In this case the generator goes on further to evaluate all those `DisplayConstraints` instances, which have associations with the type of the additional information. Section 4.4 illustrates this point in more detail.

```
model DisplayConstraintModel

from AssetModel import AssetClass
from ComponentModel import UIComponent


class DisplayConstraints{
                              concept
          relationship comps    : UIComponent*
          relationship applmodel: AssetClass
}
```

Code Example 4.9 *Display Constraints Model*

4.4 User Interface Description Implementation

Based on the `ApplicationDomain` model, `Component` model, `Technology model` and the `DisplayConstraints` model this section specifies the format in which the interface designer may define an interface. The format described in code example 4.10 below relates all four aspects.

```
model InterfaceDescriptionModel
from ComponentModel import UIComponent
from TechnologyModel import UITechnology
from DisplayConstraintModel import DisplayContraints

class InterfaceDescription{
     concept
          characteristic interfacename : String
          characteristic technology    : UITechnology
          characteristic interfacecomps: UIComponent*
          characteristic disconsts     : DisplayConstraints*
}
```

Code Example 4.10 *Interface Description Model*

For a simple interface implementation consider figure 4.1 below. On the right-hand side of the figure the interface designer chooses to visualize an application domain consisting of the asset classes `Person` and `Student` in the window on the left-hand side of the figure. The designer, however, wishes to have each application asset class visualized panel-wise. That is, the `PersonPanel` focuses on displaying `Person` instances, by visualizing only those attributes unique to `Person` (i.e. `name`). Similarly the `StudentPanel` visualizes only those attributes unique to `Student` (i.e. `matrikel`).

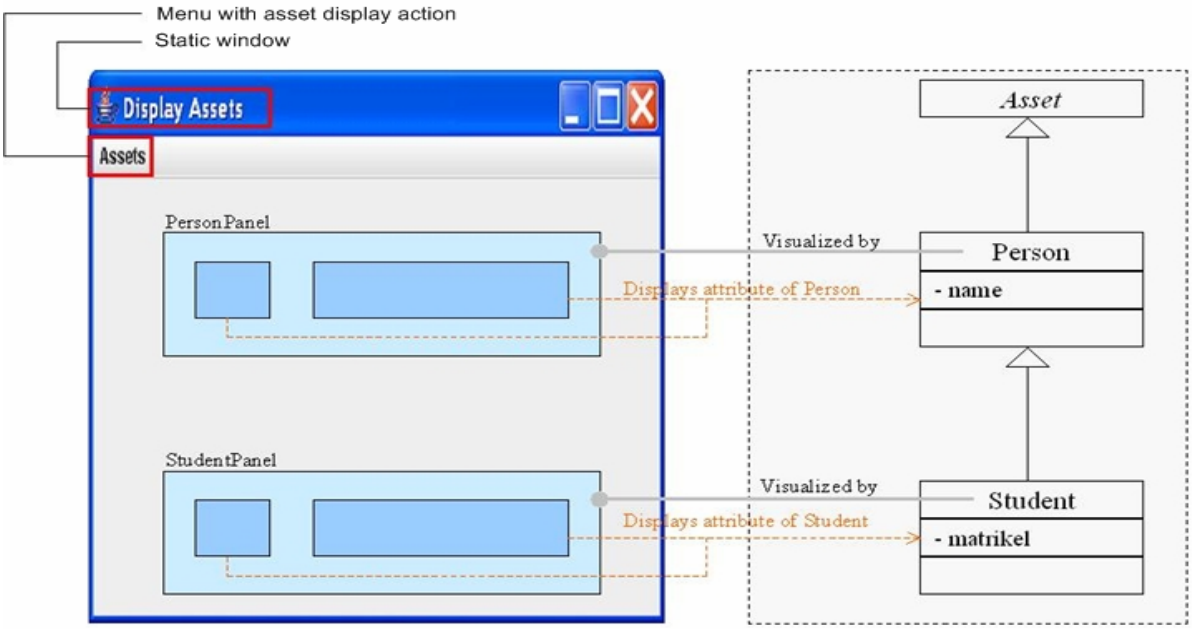


Figure 4.1 *Visualization Dynamics Scenario*

In this scenario a `Label` and a `TextField` are chosen to visualize the name attribute of `Person`. Given more attributes, their individual representations will be modeled within the panel. A similar argument holds for the visualization of `Student`. The corresponding user code for this scenario looks as shown in code example 4.11.

```
model UserModel
from ComponentModel import Window, Label, TextField,
                           ContentPane, Panel
from ApplicationModel import Person, Student
from DisplayConstraintModel import InterfaceDescription
from TechnologyModel import Swing

;;variable initializations
let swing     := create Swing{name:=“Swing”}
let plabel    :=create Label{name := “Name”}
let pTextField:=create TextField{
                              modelattr:=Person.name}
let slabel    :=create Label{name := “Matrikel”}
let sTextField:=create TextField{
                              modelattr:=Student.matrikel}
let pPanel    := create Panel{comps:={plabel, pTextField}}
let sPanel    := create Panel{comps:={slabel, sTextField}}
let pDisCons  := create DisplayConstraints{comps:= {pPanel}
                                        applmodel:=Person}
let sDisCons  := create DisplayConstraints{comps:= {sPanel}
                                      applmodel:= Student}
;;;Interface definition
let myInterfaceDescription : create InterfaceDescription{
     interfacename :=“MyAssetInterface”
     technology    := swing
     interfacecomps:= {create Window{
                    title:=“DisplayAssets”
                    contentpane:= create ContentPane{
                                 comps:={pPanel, sPanel}
                               };contentpane
                    };window
                    };interfacecomps
     disconts     :={pDisCons, sDisCons}
};create InterfaceDescription
```

Code Example 4.11 *User Interface Definition*

This is a simplified version of how an interface could be defined. What is missing is of course information concerning the geometric positioning of the components and their format (what size, color etc). Appendix C (user model) illustrates in greater detail a sample implementation of the design presented in figure 4.1.

A special remark on code example 4.11 a goes to the attribute `modelattr` on the `Person`’ TextField (`pTextField`). What this indicates is that the text attribute of `TextField`

(inferring from the setter method for this attribute on `TextField)` is dynamically bound to the `name` attribute of a `Person` instance. The generator uses this information to create code for the dynamic visualization (see figure 3.9 for better illustration).

```
;;instance initializations

let p     :=create Person {name    :="Joe Doe"}
let s     :=create Student{name    :="Gerald Mofor"
                           matrikel:="12345"}
```

Code Example 4.12 *Runtime Application Domain Instances*

For example, given the runtime instances shown above in code example 4.12, when visualizing the `Student` instance `'s'` the `sPanel` is initially painted (satisfying `sDisCons`), but the generator realized that `pDisCons` is also satisfied making also possible for `pPanel` to be painted. `Text` in `pTextField` gets the String value `"Gerald Mofor"` meanwhile the `text` in `sTextField` gets the String value `"12345".` In the case of the `Person` instance `'p'`, the evaluation is passed for `pDisCons` but it fails when evaluated against `sDisCons` because a `Person` is not an instance of `Student`. The value text in `pTextField` gets is then `"Joe Doe".`

Appendix D (modality code) shows a sample generated code showing how the values are inserted into the `TextFields` while exhibiting modality behavior. First and foremost, a reference is made to the current asset the needs to be visualized (refer to by `currentAsset.getType()` on the first line of the first grayed area). In accordance with figure 4.1 stipulating that an asset class is visualized by the cumulative visualization of its of its individual attributes, the next step is to load all the attributes describing the current asset class (refer to SECTION ONE in appendix C).

Once the attributes have been loaded, the next issue it to search for all view components constrained by each individual attribute (refer to SECTION THREE in appendix C). But before the search is carried out, the value of the current attribute (`attr`) in the current asset class has to be known (refer to SECTION TWO in appendix C). The local variable `modelattrValue` is dedicated for storing the value of the current attribute. The grayed area in section two shows how the attribute value is determined. Notice how for every class, there is a check against its corresponding attribute names. The class `Person` has only one attribute, hence only one check is made (`if("name".equals(attrName))).` Meanwhile the class Student has two attributes, and therefore two corresponding checks are made(`if("matrikel".equals(attrName))` and `if("name".equals(attrName))`).

After the current attribute value is determined, it can be inserted into the view components associated with the current attribute (refer to SECTION FOUR in appendix C). Two details are important here. The first remark is that an attribute may constrain more than one view component. The number of constrained view components is reflected in the number of if-statements in the grayed area of section four. In this case `JTextField` and `Jlabel` are the only two constrained view components (see also dotted lines leaving from panels to the application model in figure 4.1). The second remark is that these separate view components may require different setter methods for inserting values. In this case it is just a coincidence that both view components make use of the `setText` method. For the `Jlabel` component the method `setIcon` could have been used if one had to insert an image.

Chapter 5

Prototype Experiment

In this chapter the models designed so far will be tested in a live scenario. Worth mentioning is the fact that while imploring generative programming there are always two levels to be distinguished, namely the *generating* and the *generated* code. Both of these levels were referred to in section 2.8.2 as generator and generated space respectively. They involve two separate ways of thinking and these will be demonstrated in this chapter. For a brief outline, this chapter will deal with:

- A brief description of the experiment environment
- The structure of generator code, and
- The structure of generated code.

5.1 Experiment Environment

By the time of writing the compiler did not permit one to process instances at the meta level (work is still under construction). However, for illustrative purposes displaying the

expected behavior, a prototype version for the realization of the interface definition from chapter 4 section 4.4 has been conceived. The simplifications for this experiment involve the following:

- The implementation focuses on the Swing technology.
- Instances are not constructed the normal way in which one would expect in the meta level (refer to appendix C), but instead the generator implicitly creates an instance once it encounters a class definition. The values of these implicitly created instances are passed as parameters to the generator during runtime.
- Some naming conventions at the `DisplayConstraints` level making clear that the visualization for an asset is achieved through the visualization of its attributes. This naming convention ensures consistency between the UI for and attribute and the value passed to for the attribute.
- A static window painting all asset instances, as shown in figure 4.1. In this case, the `ContentPane` of the static window is used for dynamic display. This static window is implicitly created by the generator. That is, there is exist no clear definition for a window.
- The layout manager is the `GridBagLayout.`

```
model userinterfacedef

;;;; Displayable UI-Components

class UIComponent

class Frame refines UIComponent{
     concept
          characteristic tech  : javax.swing.JFrame
          characteristic title : String
          relationship   comps : UIComp*
}
class TextField refines UIComponent{
                              concept
          characteristic tech : javax.swing.JTextField
          characteristic text : String
}
class Label refines UIComponent{
     concept
          characteristic tech : javax.swing.JLabel
          characteristic label: String
}
```

Code Example 5.1 *Displayable UI-Components*

The `Displayable UIComponent` model (code example 5.1 above) contains only two displayable components at the moment (`Label, TextField`). As a consequence,

the user is limited to only these components when choosing a way to visualize the `Person` and `Student` assets from the Application model.
The `tech` attribute of the `Displayable UIComponents` combines into one unit the role of the `UITechnology Model` (number 2) and the `Component-Implementation Model` (number 3) shown in figure 3.1, by specifying directly which technology is involved and the corresponding the implementation component. For example the asset class `Label` will be implemented using the Swing technology (inferred from `javax.swing`) and the corresponding component is 's `JLabel`.

A bit tricky is the `DisplayConstraint` class. It plays a redirection role of just giving a type cover to all its subclasses. The reason for this is to enable generator to identify `DisplayConstraints` classes. All its subclasses need to conform to the format of having a mandatory attribute called `applAsset` which refers to a particular application asset (`Person or Student` in this case) to be visualized, and all other relationships make reference to `Displayable UIComponents` used for the application asset's visualization.

```
;;;DisplayConstraints Assets

class DisplayConstraints

class PersonDisCond refines DisplayConstraints{
                            concept
          relationship applAsset        : Person
          relationship nameUI           : Label
          relationship nametextvalueUI  : TextField
}; PersonDisCond

class StudentDisCond refines DisplayConstraints{
  concept
                    relationship applAsset
     relationship matrikelUI            : Label
     relationship matrikeltextvalueUI   : TextField
}; StudentDisCond
```

Code Example 5.2 *Prototype Display Constraints*

As one might notice from the code example 5.2, the referred component names are not chosen randomly. Looking at the `PersonDisCond`, what both relationships, `nameUI` and `nametextvalueUI`, have in common is that they both begin with a prefix referring to the name of the model's attribute to be visualized and end with a `UI` suffix, meaning that they are UI components for a particular attribute of a particular model.

A further distinction is made for those components expecting values coming from the model. These components are identified by the some middle term starting with the name of the UI component's attribute to be modified and ending with the word `value`. For

example the middle term in `nametextvalueUI` will be interpreted by the generator to mean that when inserting the value for the text property of a `Person's TextField` it needs to fetch values from the model (application asset) rather than displaying the default values. The reason for this walk around is simply to simulate the existence of conceptual instances.

Other information concerning the formatting of the UI components - like the height, width, visibility etc - are taken for granted. Default values are set by the generator. Of course, this tempers with the users right to make personal choices. Figure 3.9 shows the formal connection of all the classed in this experiment. It shows how the model (application asset) is only committed to the `DisplayConstraint` and not the `UIComponents`, and that the knowledge about the relationships a model has with another can be deduced from the `ModelType` side of `DisplayConstraints`.

However, the essence of this section is not to come up with a full-fletched implementation but rather to visually demonstrate through prototypical means how dynamic view adaptation can be realized. Nevertheless, the user is still given the right to decide upon the geometrical positioning of the components, by passing on GridBagLayout parameters referring to specific components. Code example 5.3 illustrates how the user may feed in geometric positioning information about view components.

```
<configuration name="VisualizationGenerator">
 <generator name="visualizationGenerator"
            class="de.sts.tuhh.myguipkg.VisualizationGenerator">

   <param name="PersonnameUI">width=1/height=1/posx=0/posy=0/weightx=0/
       weighty=0/fill=NONE/insets=(3,10,0,0)/anchor=EAST</param>
   <param name="PersonnametextvalueUI">width=2/height=1/posx=2/posy=0/weightx=1/
       weighty=0/fill=HORIZONTAL/insets=(3,5,0,20)/anchor=WEST</param>
   <param name="StudentmatrikelUI">width=1/height=1/posx=0/posy=0/weightx=0/
       weighty=0/fill=NONE/insets=(3,10,0,0)/anchor=EAST</param>
   <param name="StudentmatrikeltextvalueUI">width=2/height=1/posx=2/posy=0/
       weightx=1/weighty=0/fill=HORIZONTAL/insets=(3,5,0,20)/anchor=WEST</param>

   <param name="PersonPanel">width=3/height=1/posx=0/posy=0/weightx=1/
       weighty=0/fill=BOTH/insets=(3,10,0,0)/anchor=WEST</param>
   <param name="StudentPanel">width=3/height=1/posx=0/posy=2/weightx=1/
       weighty=0/fill=BOTH/insets=(3,10,0,0)/anchor=WEST</param>

 </generator>
 <generator name="apigen"
            class="de.tuhh.sts.cocoma.compiler.generators.api.APIGenerator">
   <param name="outputDir">H:\temp2\</param>
   <param name="targetPackage">de.sts.tuhh.myguipkg.apigenerated</param>
 </generator>
</configuration>
```

Code Example 5.3 *Prototype Input Configuration*

Code example 5.3 is part of and XML configuration file used by the compiler to run backend generators. This file carries information on how to run a generator by indicating

to the compiler the name of the configuration to be used. In this case the compiler is assigned to run the configuration with name `VisualizationGenerator`. This way the compiler is informed to run two generators, namely `visualizationGenerator` and `apigen` (refer to the orange colored code in code example 5.3). For the proper functioning of these generators they require some parameters referred to by the element tag `param`. More important is the configuration related to `visualization-Generator`. This parameter section is exploited to feed in the geometric positioning of view components in accordance with some naming convention. The naming convention followed here is that all names for the UI relationships under `DisplayConstraints` are prefixed by the type name of the referred application asset. For example, in order to refer to `Label` in the asset class `PersonDisCond` of code example 5.2 its relationship name (`nameUI`) is prefixed with `Person` (referenced by the attribute `applAsset`), shown in code example 5.3 (refer to the blue colored code).

The parameter value specification for the geometric positioning follows yet another convention (refer to the red colored code). This value specification convention is designed to be in line with the parameters requested by a GridBagLayout manager. Each individual in parameter information is separated by a `'//'`. This input parameter value is read and transformed into an appropriate type by a reader class type called `GridBagParameterReader` (see figure 5.1).

5.2 Visualization Decision

A major design decision was taken here by having the visualization of all application assets displayed panel-wise as shown on figure 4.1. The implication of this, especially from an inheritance point of view, is that every panel is related to only one asset type and carries the visualizations of *new* information. What *new* here means can be explained from the point of view of a Student (refer to figure 4.1 in the previous chapter) as referring to visualizations for the attribute `matrikel` and not the attribute `name`, since the `name` is only new with `Person` and not `Student`. The advantage of this design decision is that one need not search for all the individual view components (they could be many) but instead one needs to search for only the panel(s) that visualizes the application asset. Nevertheless, the generator still has to somehow trace the relationships between all separate panels at display time and visualize all those for which the model asset to be visualized is fully described. For example, when displaying a `Person` instance, only the `Personpanel` should be displayed whereas when visualizing a `Student` instance, both the `Personpanel` and the `Studentpanel` have to be visualized at the same time. Therefore in case there is more than one panel to be visualized, the user then has to provide extra information on how the geometrically place the panels relative to one another. For this purpose, GridBagLayout parameters for panels have to be specified (see `PersonPanel` parameter of code sample 5.3).

5.3 Generator Design Structure

In order to minimize the visualization generator's complexity the structure displayed in figure 4.3 is designated. The idea behind is to have separate specialized classes focused on the major activities of the generator. The generator then delegates its tasks to the appropriate instances.

a) GridBagParameterReader

As the name suggests, this interface is in charge of reading the parameters values stated in the input configuration file (see code example 5.3), and transforming them into an appropriate format. This translation is useful when calling the GridBagConstraints helper method (refer to `makeGBConstraints()` in the panel classes of figure 5.2). The method `processInput` first of all checks the validity of the input parameter value. An exception is thrown (`ReaderException`) in case the input value does not conform to the parameter naming convention stated at the end of section 5.1. Once the check is over, the input value is then processed and stored into appropriate variables using setter methods (refer to the orange section of `GridBagParameterReader` in figure 5.1). These values can then be retrieved on request by corresponding getter methods (refer to the blue section of `GridBagParameterReader` in figure 5.1).

b) PanelWriter

This interface focuses on the creation initialization of the asset UI-panels shown figure 5.1 above. In collaboration with values parsed by the `GridBagParameterReader`, the panel writer is able to set the internal positioning of UI components within the panels (refer to the blue colored code of code example 5.3). In this scenario, `PanelWriter` uses the `writePanelFile` method to create two files, namely `PersonPanel` and `StudentPanel` (see figure 5.2 for the generated code).

c) GenerateRoutineHelper

This interface takes care of all "odd" jobs needed by the generator. This includes creating field references, making assignments, creating code for adding components to containers, configuring the format for the UI components. These odd tasks are grouped and shown as different colors on figure 5.1. The blue colored methods are helpful for setting the configuration of the view components. The orange colored methods are mainly for supporting container components while adding elements. Most important are the gray colored methods. They are responsible for generating code exhibiting dynamic view adaptation. For instance the method `addActionLis` creates code responsible for listening to and initiating an action upon a view update request. Meanwhile the other methods interact by creating code updating the value of the view components. Finally, the green colored methods are responsible for keeping track of relationships between the application asset classes, storing attribute names that require view update and storing methods to be generated in different output files.

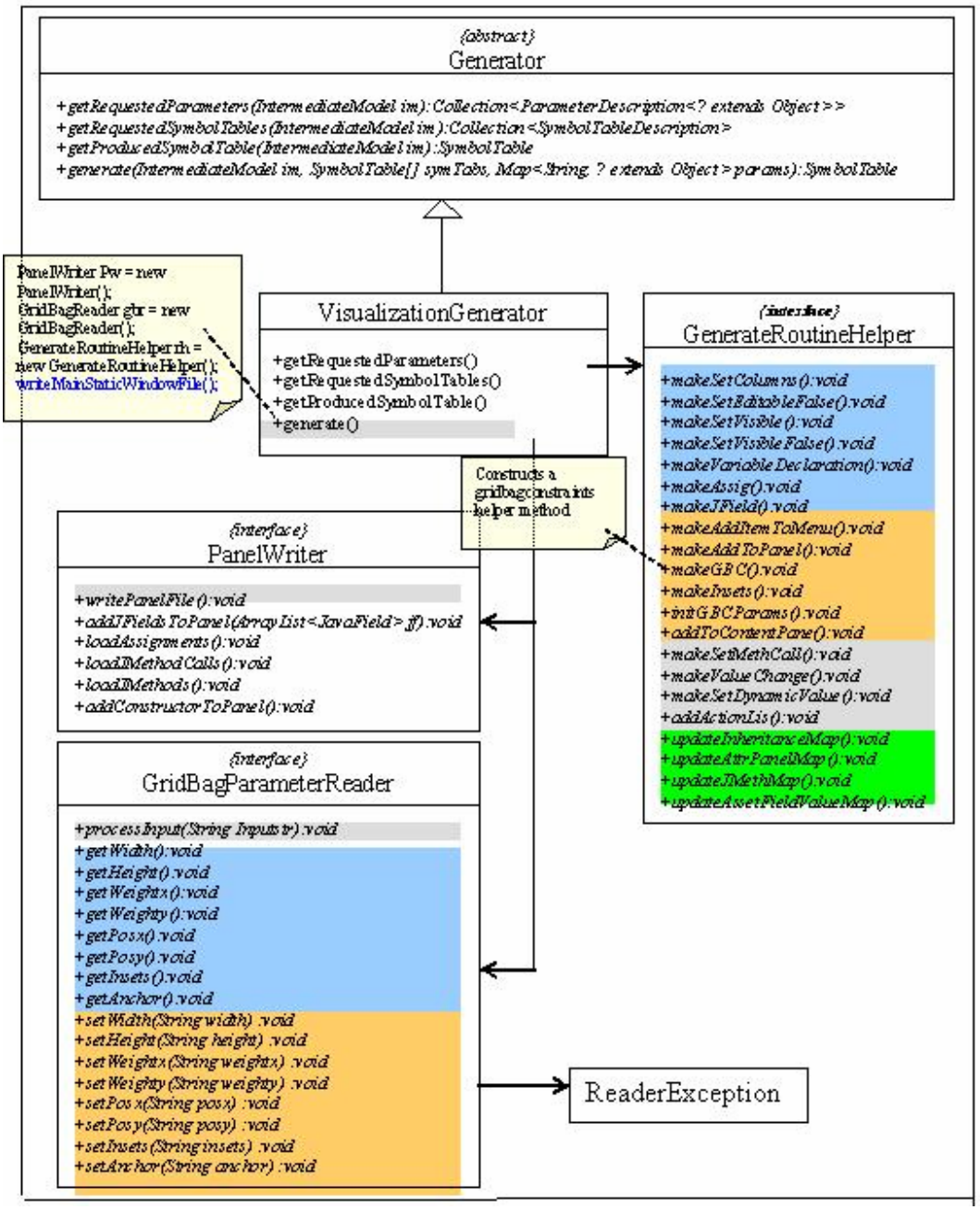


Figure 5.1 *Generator Structure*

d) VisualizationGenerator

The `VisualizationGenerator` is the main class that controls the interaction of the `GridBagParameterReader`, `PanelWriter`, and `GenerateRoutineHelper`. Important is the fact that this class recognizes the distinction between intrapanel and interpanel geometric parameter information and feeds the PanelWriter with intrapanel parameter information. With the aid of the method writeMainStatic-WindowFile the generator generates one file, MainStaticWindow (see figure 5.2), responsible for visualizing the application assets.

5.3 Generated Code Design Structure

The previous section hovered around the structure of the generator code. The conclusion was that three files would be generated, two of which come from the PanelWriter (`PersonPanel` and `StudentPanel`) and one from the VisualizationGenerator (`MainStaticPanel`). In this section the structure of the generated code is illustrated and the way in which the generated files communicate with one another is demonstrated.

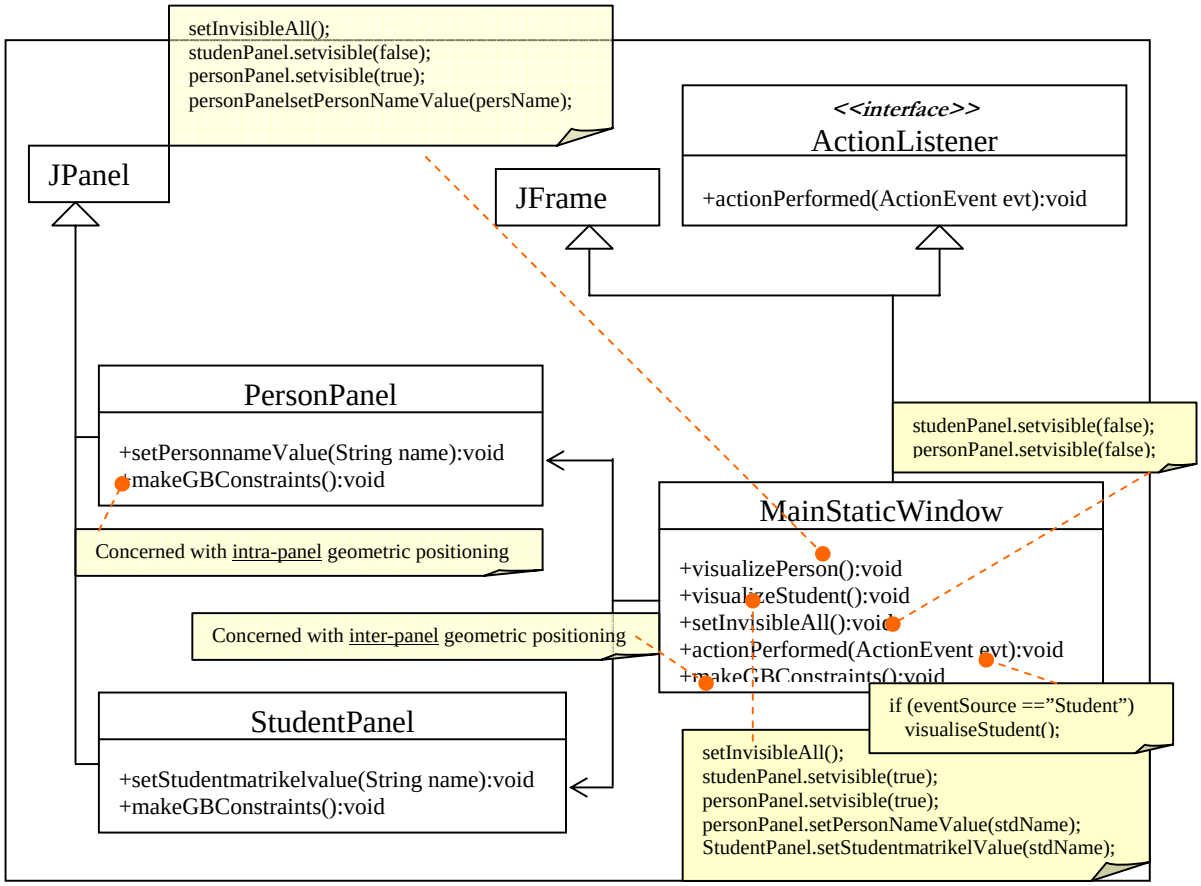


Figure 5.2 *Generated Code Structure*

Figure 5.2 shows the generated code design structure. This code structure is responsible for creating the intended effect shown on figure 5.3. On the diagram (figure 5.2) one can recognize the three classes mentioned at the beginning of this section. The idea here is that the `MainStaticWindow` class acts as the main class for the visualization. It has access to all the generated panel classes (see the navigation symbol leaving from `MainStaticWindow`). As earlier mentioned in section 5.2 (under the heading `GenerateRoutine Helper`) the helper method `makeGBConstraints` (generated by the method `makeGBC` in the `GenerateRoutineHelper` class of figure 5.1) takes care of adequately positioning the panels according to the input values given code example 5.3.

The other methods are concerned with view adaptation. Depending on which asset is to be visualized (upon an action event), the method `actionPerformed` is triggered, wherein the corresponding visualization method is called. For instance, in order to visualize a student, the menu action button on figure 5.3 is clicked to release an action event. This action event is intercepted by the actionPerformed method. The source of this action is analyzed (See note checking if the event source is from student in figure 5.2) and then method `visualizeStudent()` is called. Notice from the note linked to method `visualizeStudent()`in figure 5.2 that the person and student panels are set visible (compare with the `visualizePerson()` note).

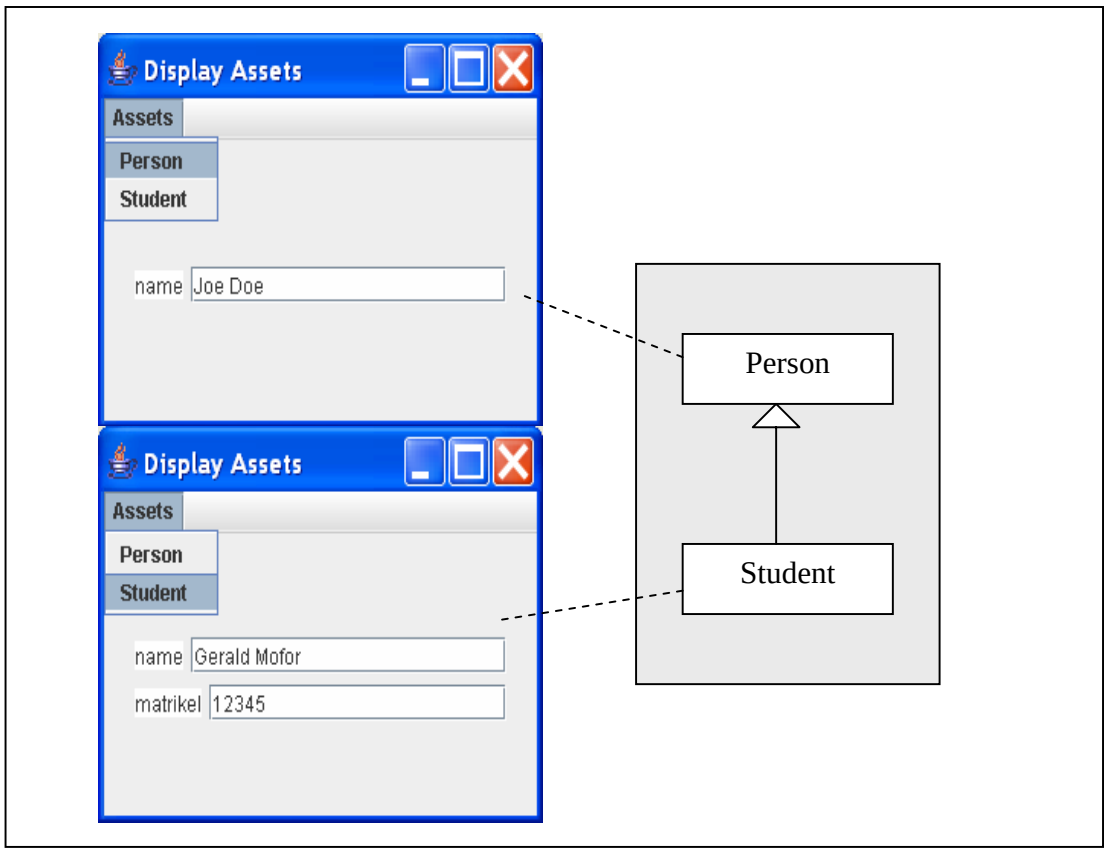


Figure 5.3 *Prototype Visualization Display*

An interesting remark is the application of a structural design style called *Façade* pattern [GoF95]. The Façade pattern provides a simplified interface to a larger body of code. It reduces dependencies of external code, hence allowing for flexibility in evolving a system. It is on the basis of these advantages that the façade pattern is applied. On figure 5.2, `MainStaticWindow` is a façade class the façade methods `actionPerformed`, `visualizePerson`, `visualizeStudent` and `setInvisibleAll`. These methods hide complexity by redirecting action to other methods.

Chapter 6

Evaluation and Outlook

In this chapter, a brief assessment of the entire study will be made (section 6.1) including a summary of the entire project work. Section 6.2 then closes the study by examining some amelioration possibilities.

6.1 Evaluation

The original intent of this study was to come up with suitable ideas to model user interfaces for conceptual content management systems. The study was motivated by the fact that Conceptual Content Management Systems belong to the family of interactive systems and as a result needs visualisations to interact with a user. The realization of this claim was supported by the idea that the visualisation had to be generated in order to match changes in the underlying system. Nevertheless one could not talk about implementation without discussing some design models.

The formal goal of designing models for CCMS was fully achieved following the thought process of an environment analysis to gather the main requirements of such a system

(chapter 2). This was followed by inspecting various designs alternatives that could sooth the requirements (chapter 3), with a final decision made on the basis of keeping things simple for the user, and at the same time avoiding implementation complexity. The advantages about the design models is the following:

- They give room for extension,
- They are scalable, and
- They offer flexibility.

Particular reference is made to the following core design diagrams: figures 3.2, 3.5 and 3.6. After the design procedure, the next step was to conceptually implement the chosen model diagrams (chapter 4), with a concluding step of presenting how code for the user would look like. The previous chapter took a step into the concrete realization of the conceptual implementation models, by implementing a prototype. The goal behind the prototype experiment, keeping all implementation limitations aside, was to validate the fact that models designed so far are actually realizable.

6.2 Outlook

This section provides a brief outlook on improvement proposals for the realization of future projects.

- The foremost proposal will be to have an improvement at the CCMS' meta-level to support instances. This is very vital for the full rollout of concrete user interfaces.

- A concise study of the on role of a controller for visualization and its implications for CCMSs would be very beneficial since every development hovering around linking the presentation layer and its application model makes use of the Model-View-Controller architectural pattern.

- Close implications of the previous point are concerns about modeling and embedding a suitable event-based system into the visualization construction. The main task would be to design how events are triggered and handled.

- Another improvement proposal will be to enable the implementation of a mix of UI technologies within an interface. The idea behind is of course to gain from the benefits of each individual technology.

- It would also formally be nice to improve on the visualization scheme like displaying other assets having a relationship with the present asset being visualized. See figure 6.1. The way to depict associations depends on the user, for example the figure below uses the right side of the `JSplitpane` to display all associations

- More investigation has to be done for displaying a collection of instances.

- There is also room for improvement at the level of including action states. Like for instance having buttons executing some query action and displaying the results maybe on a separate window.

- Open areas still remain in providing sound mechanisms for event management, application flows, and widget control.

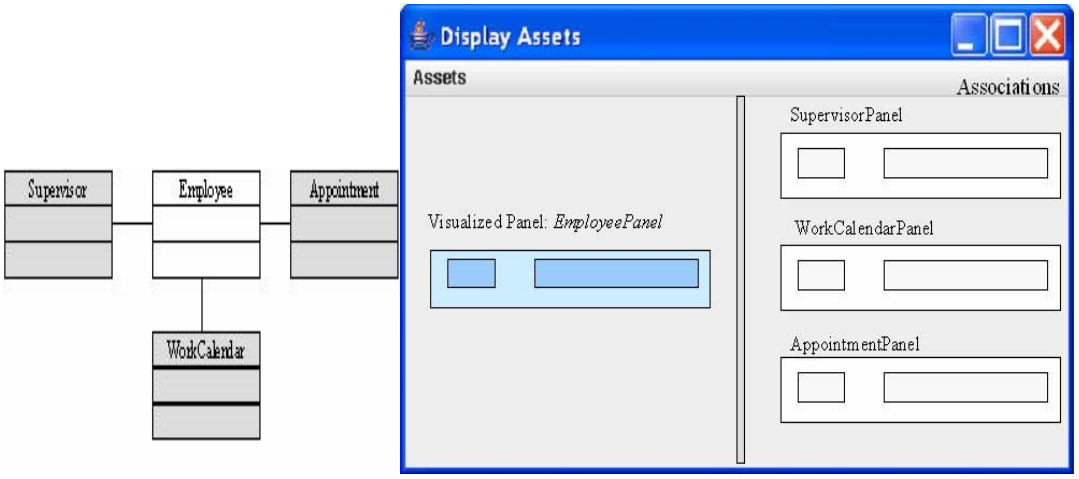


Figure 6.1 *Association relationships*

- Generators for conceptual interfaces could become extremely very complex with robust UIs. It would be nice to have a generator performance test, analyzing the feasibility of code when things get too complex.

- Last but not the least, multi-user authoring is a big issue with CMSs since there could be many simultaneous users involved. Features such as record locking to ensure that clashing changes are prevented still remains an area for further investigation.

APPENDIX A

Detailed Component Model

```
================================================================================
model CompenentModel

class LayoutParam;
class GridBagLayoutParam refines LayoutParam{
      concept
            characteristic width  : int
            characteristic height : int
            characteristic posx   : int
            characteristic posy   : int
            characteristic weightx: double
            characteristic weighty: double
            characteristic fill   : String
            characteristic insets : java.awt.Insets
            characteristic anchor : String
}
```

```
class UIcomponent{
      concept
            characteristic name             : String
            characteristic preferredsize    : java.awt.Dimension
            characteristic visibility       : boolean
            characteristic font             : java.awt.Font
            characteristic bgcolor          : java.awt.Color
            characteristic fgcolor          : java.awt.Color
            characteristic height           : int
            characteristic width            : int
            characteristic lookandfeel      : javax.swing.plaf.ComponentUI
            relationship   layoutmgrparam   : LayoutParam
            relationship   modelattr        : Attribute
};UIComponent

class AssetViewComponent refines UIComponent           ;AssetViewComponent

class Action{
      concept
            characteristic name                 : String
            characteristic mnemonickey          : String
            characteristic actioncommandkey     : String
}



class ActiveComponent refines UIComponent{
      concept
            relationship a  : Action
}

class UIContainer refines UIComponent{                ;UIContainer
      concept
            characteristic   lytMgr : java.awt.LayoutManager
};UIContainer

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;AssetViewComponents
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

class Label refines AssetViewComponent{
      concept
            characteristic text : String
            characteristic image: javax.swing.Icon
}

class TextField refines AssetViewComponent{
      concept
            characteristic text    : String
            characteristic columns : int
}
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;ActiveComponents
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
class Icon
class Button refines ActiveComponent{
      concept
            characteristic label     : String
            characteristic i         : Icon
}
class MenuElement refines UIComponent;
class MenuItem refines MenuElement{
      concept
            characteristic mnemonic : int
            characteristic i        : Icon
}
class Menu refines ActiveComponent, MenuElement{
      concept
            relationship   menuelms : MenuElement*
}
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;Container Classes
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
class MenuBar refines UIContainer, MenuElement{
      concept
            relationship menus : Menu*
}

class ContentPane refines UIContainer{
      concept
            relationship comps : UIComponent*
};ContentPane

class Window refines UIContainer{
      concept
            characteristic title      : String
            relationship   mb         : MenuBar
            relationship   contentpane: ContentPane
};Window

class Panel refines UIContainer{
      concept
            relationship comps : UIComponent*
};Panel

class SplitPane refines UIContainer{
      concept
            characteristic newOrientation  : int
            relationship newLeftComponent  : UIComponent
            relationship newRightComponent : UIComponent
}
class ViewPort{
      concept
            characteristic newsize : java.awt.Dimension
```

```
            relationship   view    : UIComponent
}
class ScrollPane refines UIContainer{
      concept
            characteristic vsbPolicy : int
            characteristic hsbPolicy : int
            relationship   view      : UIComponent
            relationship   vp        : ViewPort
}
```

APPENDIX B

Detailed Swing Implementation Model

```
================================================================================
model SwingTechImplModel

from ComponentModel     import Label, TextField, SplitPane, ScrollBar, Menu,
                               Button, Window, Panel
from ComponentImplModel import UIComponentImpl, JavaConstructorDescription,
                               JavaMethodDescription
from TechnologyModel    import Swing

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;    Label
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

let swing := create Swing{name= "Swing"}
let swingLabelImpl := create UIComponentImpl{
      jClass := javax.swing.JLabel.class
      component := Label
      tech := swing
      constructors := {create JavaConstructorDescription{
                               parameterTypes      := new Class[]{ String.class
}
```

```
                                        componentAttributes := { Label.text}
                                                                }
                              };constructors
      setterMethods := { create JavaMethodDescription{
                                                methodName     := "setIcon"
                                    parameterTypes := new Class[] { Icon.class}
                                          componentAttributes := {Label.image}
                                    }
                  create JavaMethodDescription{
                                          methodName     := "setText"
                                    parameterTypes := new Class[]{String.class}
                                    componentAttributes := {Label.modelattr}
                                    }
                        };setterMethods
};c. swingUIComponentImpl


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;    TextField
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

let swingTextFieldImpl := create UIComponentImpl{
      jClass := javax.swing.JTextField.class
      component := TextField
      tech := swing
      constructors := {create JavaConstructorDescription{
                              parameterTypes      := new Class[]{ String.class
}
                              componentAttributes := { TextField.text}
                                                                }
                              };constructors
      setterMethods := { create JavaMethodDescription{
                                          methodName     := "setColumns"
                                          parameterTypes := new Class[] {
                                                Integer.type}
                                    componentAttributes := {TextField.columns}
                                    }
                         create JavaMethodDescription{
                                                methodName     := "setText"
                              parameterTypes := new Class[]{String.class}
                              componentAttributes := {TextField.modelattr}
                                    }
                        };setterMethods
};c. swingUIComponentImpl

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;    MenuItem
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

let swingMenuItemImpl := create UIComponentImpl{
      jClass := javax.swing.JMenuItem.class
      component := MenuItem
      tech := swing
      constructors := {create JavaConstructorDescription{
                         parameterTypes      := new Class[]{ String.class }
```

```
                              componentAttributes := { MenuItem.text}
                                                                  }
                              }
      setterMethods := { create JavaMethodDescription{
                                    methodName     := "setAction"
                              parameterTypes :=new Class[] {Action.class}
                                    componentAttributes := {MenuItem.a}
                                    },
                           create JavaMethodDescription{
                                    methodName     := "setMnemonic"
                              parameterTypes:=new Class[]{Character.class}
                              componentAttributes := {MenuItem.mnemonic}
                                    },
                           create JavaMethodDescription{
                                    methodName     := "setIcon"
                              parameterTypes := new Class[] {Icon.class}
                                    componentAttributes := {MenuItem.i}
                                    },
                                 create JavaMethodDescription{
                                                methodName     := "add"
                                    parameterTypes:=new
Class[]{Component.class}
                                                componentAttributes :=
{MenuItem.comp}
                                    }
                        };setterMethods

};c. swingUIComponentImpl
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;Menu
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
let swingMenuImpl := create UIComponentImpl{
      jClass := javax.swing.JMenu.class
      component := Menu
      tech := swing
      constructors := {create JavaConstructorDescription{
                              parameterTypes      := new Class[]{ String.class
}
                                    componentAttributes := { Menu.text}
                                                                  }
                              };constructors
      setAction := { create JavaMethodDescription{
                                                methodName     := "add"
                              parameterTypes := new Class[] {Action.class}
                              componentAttributes := {MenuItem.a}
                                    },
                     create JavaMethodDescription{
                                    methodName     := "add"
                              parameterTypes:=new Class[]{MenuElement.class}
                              componentAttributes := {Menu.menuelms}
                                    },
                     create JavaMethodDescription{
                                                methodName     := "add"
                     parameterTypes:=new Class[]{JMenuItem.class}
                     componentAttributes := {Menu.mi}
```

```
                                    }
                };setterMethods
};c. swingUIComponentImpl
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;Button
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
let swingButtonImpl := create UIComponentImpl{
      jClass := javax.swing.JButton.class
      component := Button
      tech := swing
      ;; constructors  (use default constructor)

      setterMethods := { create JavaMethodDescription{
                                    methodName     := "setLabel"
                              parameterTypes := new Class[] { String.class}
                                    componentAttributes := {Button.label}
                                    },
                        create JavaMethodDescription{
                                    methodName     := "setActionCommand"
                              parameterTypes := new Class[] { String.class}
                              componentAttributes := {Button.actioncmd}
                                    }
                         };setterMethods
};c. swingUIComponentImpl
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;MenuBar
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
let swingMenuBarImpl := create UIComponentImpl{
      jClass := javax.swing.JMenuBar.class
      component := MenuBar
      tech := swing
      ;; constructors  (use default constructor)

      setterMethods := { create JavaMethodDescription{
                                    methodName     := "add"
                              parameterTypes := new Class[] { JMenu.class}
                                    componentAttributes := {MenuBar.menu}
                                    };c.JavaMethDescription
                        };setterMethods
};c. swingUIComponentImpl
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;ContentPane
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
let swingContentPaneImpl := create UIComponentImpl{
      component := ContentPane
      tech := swing
      setterMethods := { create JavaMethodDescription{
                                                methodName     := "add"
                              parameterTypes := new Class[] { Component.class}
```

```
                              componentAttributes := {ContentPane.comps}
                                    }
                        };setterMethods
};c. swingUIComponentImpl
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;Window
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
let swingWindowImpl := create UIComponentImpl{
      jClass := javax.swing.JFrame.class
      component := Window
      tech := swing
      constructors := {create JavaConstructorDescription{
                              parameterTypes      := new Class[]{ String.class
}
                              componentAttributes := { Window.title}
                                                    };c.Javaconstructor
                              };constructors

      setterMethods := { create JavaMethodDescription{
                                    methodName     := "setLayout"
            parameterTypes := new Class[] {
                                    java.awt.LayoutManager.class}
                                    componentAttributes := {Window.lytMgr}
                                    }
                        };setterMethods
};c. swingUIComponentImpl
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;    SplitPane
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
let swingSplitPaneImpl := create UIComponentImpl{
      jClass := javax.swing.JSplitPane.class
      component := SplitPane
      tech := swing
      constructors := {create JavaConstructorDescription{
                  parameterTypes      := new Class[]{
                        Integer.type, Component.class, Component.class}
                        componentAttributes := { SplitPane.newOrientation,
                                                 SplitPane.newLeftComponent,
                                                 SplitPane.newLeftComponent
                                                                        }
                                                    };c.Javaconstructor
                              };constructors
          setterMethods:= { create JavaMethodDescription{
                                       methodName     := "setLayout"
                        parameterTypes :=new Class[] {
                              java.awt.LayoutManager.class}
                                    componentAttributes := {SplitPane.lytMgr}
                                    }
                        };setterMethods
};c. swingUIComponentImpl
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;    ViewPort
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
let swingViewPortImpl := create UIComponentImpl{
      jClass := javax.swing.JViewPort.class
      component := ViewPort
      tech := swing

      setterMethods := { create JavaMethodDescription{
                                                methodName     := "add"
                                    parameterTypes := new Class[] {
                                                java.awt.Component.class}
                                    componentAttributes := {ViewPort.view}
                                    },
                             create JavaMethodDescription{
                  methodName     := "add"
                  parameterTypes := new Class[] {java.awt.Dimension.class}
                  componentAttributes := {ViewPort.newsize}
                                    };c.JavaMethDescription
                     },
                        create JavaMethodDescription{
                                          methodName     := "add"
            parameterTypes := new Class[] {java.awt.LayoutManager.class}
            componentAttributes := {ViewPort.lytMgr}
                                    };c.JavaMethDescription
                     },
                        create JavaMethodDescription{
                                    methodName     := "add"
                              parameterTypes:=new Class[]{Component.class}
                              componentAttributes := {ViewPort.comp}
                                    };c.JavaMethDescription
                     };setterMethods
};c. swingUIComponentImpl
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;    ScrollPane
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
let swingScrollPaneImpl := create UIComponentImpl{
      jClass := javax.swing.JScrollPane.class
      component := ScrollPane
      tech := swing
      constructors := {create JavaConstructorDescription{
                        parameterTypes      := new Class[]{Component.class}
                        componentAttributes := { ScrollPane.view }
                                                ;c.Javaconstructor
                              };constructors
      setterMethods := { create JavaMethodDescription{
                              methodName    :="setVerticalScrollBarPolicy"
                              parameterTypes:=new Class[] { Integer.type}
                              componentAttributes:={ScrollPane.vsbPolicy}
                                    },
                     create JavaMethodDescription{
                              methodName :="setHorizontalScrollBarPolicy"
                              parameterTypes:= new Class[] { Integer.type}
                              componentAttributes:= {ScrollPane.hsbPolicy}
```

```
                                    },
                   create JavaMethodDescription{
                        methodName     := "setViewport"
                        parameterTypes := new Class[] {javax.swing.class}
                        componentAttributes := {ScrollPane.vp}
                                    },
                   create JavaMethodDescription{
                        methodName     := "setLayout"
                  parameterTypes := new Class[]{java.awt.LayoutManager.class}
             componentAttributes := {ScrollPane.lytMgr}
                                    }
                     };setterMethods
};c. swingUIComponentImpl
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;    Panel
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
let swingPanelImpl := create UIComponentImpl{
      jClass := javax.swing.JPanel.class
      component := Panel
      tech := swing
      setterMethods := { create JavaMethodDescription{
                              methodName    :="add"
                              parameterTypes:=new Class[] { Component.class}
                              componentAttributes:={Panel.comps}
                                    },
                   create JavaMethodDescription{
                        methodName     := "setLayout"
                  parameterTypes := new Class[]{java.awt.LayoutManager.class}
             componentAttributes := {Panel.lytMgr}
                                    }
                        };setterMethods
};c. swingPanelImpl
```

APPENDIX C

User Model

```
================================================================================
model UserModel

from ComponentModel import Window, Label, TextField, MenuBar, Menu, Action,
                           MenuItem,MenuElement,ContentPane,Panel,LayoutParam,
                           GridBagLayoutParam,UIComponent

from ApplicationModel import Person, Student
from DisplayConstraintModel import DisplayConstraint

 ;*********************************************************************
 ;** Person Panel
 ;*********************************************************************

let pPanel:= create Panel{
              name :="PersonPanel"
              comps:={ create Label{text:="Name"
              layoutmgrparam := create GridBagLayoutParam{
                                  width  =1
                                  height =1
```

```
                                        posx = 0
                                        posy = 0
                                        weightx =0.0
                                        weighty =0.0
                                        fill= "NONE"
                                        insets= new java.awt.Insets(3,10,0,0)
                                        anchor="EAST"
                            };LayoutParam
                                                                  },;Label
              create TextField{
               modelattr := Person.name
               columns := 5
               layoutmgrparam := create GridBagLayoutParam{
                                        width  =2
                                        height =1
                                        posx = 2
                                        posy = 0
                                        weightx =1.0
                                        weighty =0.0
                                        fill= "HORIZONTAL"
                                        insets=new java.awt.Insets(3,5,0,20)
                                        anchor="WEST"
                            };LayoutParam
           };TextField
      };panel.comps
      lytMgr:= new GridBagLayout() ; layout of PersonPanel
      layoutmgrparam :=create GridBagLayoutParam{
                        width  =3
                        height =1
                        posx = 0
                        posy = 0
                        weightx =1.0
                        weighty =0.0
                        fill= "BOTH"
                        insets= new java.awt.Insets(3,10,0,0)
                        anchor="WEST"
                    };LayoutParam
        };Panel
```

```
;*********************************************************************
;** Student Panel
;*********************************************************************
```

```
let sPanel:= create Panel{
              name:="StudentPanel"
              comps:={
               create Label{text:="Matrikel"
                 layoutmgrparam :=
                   create GridBagLayoutParam{
                     width  =1
                     height =1
                     posx = 0
                     posy = 0
                     weightx =0.0
                     weighty =0.0
                     fill= "NONE"
                     insets=new java.awt.Insets(3,10,0,0)
```

```
                    anchor="EAST"
                };LayoutParam within panel
             },
            create TextField{
              modelattr := Student.matrikel
            columns := 5
            layoutmgrparam :=
              create GridBagLayoutParam{
              width  =2
              height =1
              posx = 2
              posy = 0
              weightx =1.0
              weighty =0.0
              fill= "HORIZONTAL"
              insets= new java.awt.Insets(3,5,0,20)
              anchor="WEST"
            };LayoutParam within panel
       };TextField
     };panel.comps
   lytMgr:= new GridBagLayout()
   layoutmgrparam :=create GridBagLayoutParam{
                           width  =3
                           height =1
                           posx = 0
                           posy = 2
                           weightx =1.0
                           weighty =0.0
                           fill= "BOTH"
                           insets= new java.awt.Insets(3,10,0,0)
                           anchor="WEST"
                  };LayoutParam
};Panel

.....................................................................................
;*********************************************************************
;UI definition
;*********************************************************************

let assetWindow := create Window{
     title:= "DisplayAssets"
     preferredsize := new java.awt.Dimension(500,300)
     mb:= create MenuBar{
             menus:={
              create Menu{name:="Assets"
                              menuelms:={
                              create MenuItem{
                               name:="Person"
                               action:=create Action{name:="Visualize Person"
                                             mnemonickey:="P"
                                            };Action
                                       },;MenuItem
                              create MenuItem{
                               name:="Student"
                               action := create Action{name:="Visualize Student"
                                                 mnemonickey:="S"
                                                };Action
```

```
                                                    };MenuItem

                                 };menuelms
                                };Menu
                             };menus
                            };MenuBar
      contentpane := create ContentPane{
                               comps := {pPanel, sPanel}
                             lytMgr:= new GridBagLayout()
                            };ContentPane
};Window
```

```
;*****************************************************************
;***Application instances
.*****************************************************************
```

```
      let p := create Person{name:= "Joe Doe"}
      let s := create Student{name := "Gerald Mofor"
                              matrikel:= 12345
                              }
```

```
;*****************************************************************
;****DisplayConst instances
.*****************************************************************
```

```
      let   pDisCons := create DisplayConstraint {
                                    applmodel:= Person
                                    comps:={pPanel}
                         }
      let   sDisCons := create DisplayConstraint {
                                    applmodel:= Student
                                    comps:={sPanel}
                              }
```

APPENDIX D

Modality Code

```
==================================================================
 AssetClass cls = currentAsset.getType();
 String clsName = cls.getName();

//***********SECTION ONE: Load all attributes of present AssetClass

HashSet<Attribute> attribs =
                        new HashSet<AttributeDescription>();
// load all the attributes of this current class
while(cls != null){
      for(Attribute attr: cls.getAttributes())
            attribs.add(attr);

      cls = cls.getSuperClass();
}//while


for(Attribute attr: attribs){
      String attrName = attr.getName();
```

```
//************ SECTION TWO: Load value for modality attribute
String modalattrValue;

if("Person".equals(clsName)){
      if("name".equals(attrName))
      modalattrValue = ((AbstractPerson)currentAsset).getName();
}//if Person
else if("Student".equals(clsName)){
       if("matrikel".equals(attrName))
          modalattrValue =((AbstractStudent)currentAsset).getMatrikel();
         else if("name".equals(attrName))
          modalattrValue =((AbstractPerson)currentAsset).getName();
      }//Student

//************* SECTION THREE: Look for views constrained by this
//particular attribute

        AssetViewComponentQuery q =
      (AssetViewComponentQuery)module.getClass(
                              "AssetViewComponent").startQuery();
      q.constrainModelattrEqual(attr);

      AssetViewComponentIterator it =
                              q.executeForAssetViewComponent();


//******* SECTION FOUR: Insert modality value on all constrained views
       for(AssetViewComponent view: it){
             if(view instanceof JTextField)
                   ((JTextField)view).setText(modalattrValue);
             if(view instanceof JLabel)
                   ((JLabel)view).setText(attrName);
         }//for it

}//for views constrained by the present attribute
```

Bibliography

[ABDM03] AGER, Mads S.; BIERNACKI, Dariusz; DANVY, Olivier; MIDTGAARD, Jan: *From Interpreter to Compiler and Virtual Machine: A Functional Derivation*, BRICS Report Series, RS-03-14, March 2003.

[APBW+99] ABRAMS, Marc; PHANOURIOU, Constantinos; BATONGBACAL, Alan; WILLIAMS Stephen and SHUSTER, Jonathan: *UIML: An Appliance Independent XML User Interface Language*. WWW8/Computer Networks 31(11-16), page 1695-1708, 1999.

[BMRS+98] BUSCGMANN, Frank; MEUNIER, Regine; ROHNERT, Hans; SOMMERLAD, Peter and STAL, Michael: *A System of Patterns. Pattern-Oriented Software Architecture*, John Wiley& Sons, 1998.

[BSD01] BULLGARD, Vaughn; SMITH, Kevin and DACONTA, Micheal: *Essential XUL Programming*, Wiley, July 2001.

[Cass02] CASSIRER, Ernst: *Die Sprache, Band 11 Philophie der symbolischen Formen der Reihe Gesammelte Werke*, Felix Meiner Verlag GmbH, Hamburger Ausgabe Auflage, 2002.

[Chan04] CHANDLER, Daniel: *Semiotics: The Basics*, pages 17-78, Taylor & Francis Books Ltd, 2004.

[Coll95] COLLINS, Dave: *Designing Object Oriented User Interfaces*, Benjamin/Cummings INC, 1995.

[Colw05] COLWELL, Bob: *At Random: Complexity in Design*, pages 10-12, Paper, IEEE Computer Society, October 2005, http://csdl2.computer.org/comp/mags/co/2005/10/rx010.pdf

[CzEi00] CZARNECKI, Krzysztof and EISENECKER Ulrich. *Generative Programming – Methods, Tools, and Applications*, pages 131-568, Addison-Wesley Professional, 2000.

[DFAB03] DIX, Alan; FINLEY, Janet; ABOWD, Gregory D.; BEALE, Russell: *Human-Computer Interaction, 3rd Edition*, pages 141-259, Prentice Hall, December 2003.

[DRDY05] DAVID, Jean-Luc; RYAN, Bill; DESERRANNO, Ron; YOUNG, Alexandra: *Professional WinFX Beta: Covers "Avalon" Windows Presentation Foundation and "Indigo" Windows Communication Foundation*, Pages 3-56, Wrox, September 2005.

[EEES03] EMRICH, Marco; EISENECKER, Ulrich; ENDLER, Christian; SCHLEE, Max: *Emerging Product Line Implementation Technologies: C++, Frames, and Generating User Interfaces*, University of Applied Sciences, Kaiserslautern, September 2003.

[GoF95] GAMMA, Erich; HELM, Richard; JOHNSON, Ralph and VLISSIDES, John: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1995.

[LiLa05] LITHBRIDGE, Timothy and LAGANIERE, Robert. *Object-oriented software engineering: practical software development using UML and Java*, McGraw Hill, 2005.

[MWK04] MERRICK, Roland; WOOD, Brain and KREBS, William: *Abstract User Interface Markup Language*. In Kris Luyten, Marc Abrams, Jean Vanderdonckt, and Quentin Limbourg, editors, Proceedings of the Workshop on Developing User Interfaces with XML: Advances on User Interface Description Languages, Gallipoli, Italy, May 2004.

[Peir31] PEIRCE, C.S.: *Collected Papers of Charles Sanders Peirce*, Harvard University Press, Cambridge, 1931.

[Sehr04] SEHRING, Hans-Werner. *Konzeptorientierte Inhaltsverwaltung Model, Systemarchitektur und Prototypen*, Dissertation, pages 146-158, 2004.

[Sehr06] SEHRING, Hans-Werner: *Code Generation Toolkit*, Website, http://www.sts.tu-harburg.de/~hw.sehring/codegentk/doc/index.html, last visited, May 2006.

[SeSc04] SEHRING, Hans-Werner and SCHMIDT, Joachim W. *Beyond Databases: An Asset Language for Conceptual Content Management*, In: András Benczúr, János Demetrovics, and Georg Gottlob (editors), Proceedings of the 8th East European Conference on Advances in Databases and Information Systems, volume 3255 of LNCS, pages 99–112. Springer-Verlag, 2004.

[SJT05] SHARMA, Vibhu S.; JALOTE, Pankaj and TRIVEDI, Kishor S.: *Evaluating Performance Attributes of Layered Software Architecture*. Proceedings of the 8th International SIGSOFT Symposium on Component-based Software Engineering (CBSE), refer LNCS 3489, pages 66-81. St Louis, Missouri, USA, May 2005.

[Sowa76] SOWA, John F.: *Conceptual Graphs for a Data Base Interface*, pages 336-357, IBM Journal of Research and Development 20(4), July 1976.

[Webs95] WEBSTER, Bruce F. *Pitfalls of Object Oriented Development*, M&T Books, 1995.

[Wegn87] WEGNER, Peter: *Dimensions of Object-Based Language Design*, In Conference on Object Oriented Programming Systems Languages and Applications Conference proceedings on Object-oriented programming systems, languages and applications ’87 Proceedings. ACM Press, October 1987.

[Xu04] XU, Fenfang: *Asset Presentation in Open Dynamic Content Management Systems: A Model of User Interface Components and Design Considerations for a Visualization Engine or Generator*, Project Work, 2004.

Declaration

I declare that

this work has been prepared by me, all the literal or content-based quotations are clearly pointed out, and no other sources or aids than the declared ones have been used.

Hamburg, 17th August 2006
Gerald Mofor