Technical University Hamburg-Harburg

Project Study Report

Fenfang Xu

Asset Presentation in Open Dynamic Content Management Systems: A Model of User Interface Components and Design Considerations for a Visualization Engine or Generator

Degree course: Information and Media Technologies
Supervising examiner: Prof. Dr. Joachim W. Schmidt
Second examiner: Dr. Hans-Werner Sehring
Delivery day 30th August 2004

Index

List of Figures

List of Tables

Appendix

1 Introduction

This chapter will describe a motivation and the structure of the project report.

1.1 Motivation

A new approach called conceptual content management [1] is based on a new language called asset language. This approach uses assets to model application domain in an innovative way. The conceptual content management is open and dynamic. Open means that users can change a domain model on the fly and any time. Dynamic means that the system implementation changes dynamically following any on the fly modification of a domain model. A user interface (UI) of a conceptual content management system has to be adapted dynamically because domain models change constantly. However, dynamic adaptation of a UI is usually not addressed by the existing UI technologies. Openness and dynamics together allow conceptual content management systems to be constantly adapted, refined and personalized according to the requirements as demanded by its users' tasks.

Since a good layout of a UI cannot be decided by a machine and a UI cannot be automatically constructed, a UI has to be user definable. Some inputs such as a layout from a user or a screen designer are required. After researching the existing UI technologies, no UI technologies which are open and dynamic could be found and there is no suitable UI technology for dynamic adaptation of a UI. A UI engine is needed to render a UI or a UI generator is required to generate code that creates a UI. There are several approaches to realize the dynamic adaptation of a UI. One approach is that a UI engine is given and user input such as a layout is required. The disadvantage of this approach is that a user has to use more than one language such as one language for application domain and one language for Layout. Alternative is that a UI engine is provided and user input such as a layout is defined by using assets. The user interface is implemented by describing UIs through the Asset Definition Language (ADL) by using assets to model the UI realm. The advantage of this approach is that the ADL allows three contributions: the evolution, personalization, and adaptability of a user interface. A user is required to use only one language which he already knows. A UI engine or generator is designed in order to realize an open dynamic visualization. The visualization is realized by a combination of the application domain and the UI realm. There are two domains in the UI realm that are orthogonal: one for logical UI components and one for presentation technologies. The UI engine or generator works based on a UI components model, a UI technologies model, and an application domain model.

This project study does a preparation for the development of an engine or generator for dynamic UIs and defines a UI components model and a UI technologies model logically as well as its formalization in terms of the asset language and analyses design considerations for a visualization engine or generator which realize dynamic visualization.

1.2 The Structure of this Project Report

After a short discussion of requirements for the visualization of a UI (chapter 2) we design a possible UI components model (chapter 3). Chapter 4 illustrates a UI technologies model. Chapter 5 analyses several possibilities for implementing a UI engine or generator and proposes one solution. Finally, chapter 6 concludes with a short summary and a look at future development of the UI visualization engine or generator for UIs.

2 Requirements

As briefly mentioned in chapter 1, the existing UI technologies have limitations. They are not open and dynamic as mentioned in chapter 1. In this chapter, first, we will discuss problems of existing user interface visualization approaches. Then we will analyse the requirements of a visualization engine or generator. Finally, we will look at possible approaches.

2.1 Problems of Existing User Interface Visualization Approaches

Existing user interface visualization approaches are not open and dynamic. For example, HTML is the language for writing hypertext. It is inadequate for a large and complex application due to its fixed set of tags and limited graphic capabilities. UIML (User Interface Markup Language) [2] allows designers to describe the user interface in generic terms, and then uses a style description to map the interface to various operating systems and appliances. SwiXML [3] is a small GUI generating engine for Java applications and applets. Graphical User Interfaces are described in XML documents that are parsed at runtime and rendered into `javax.swing` objects. XUL (XML User Interface Language) is a markup language created for the `Mozilla` application and is used to define its user interface [4]. WebML (Web Modelling Language) is a notation for specifying complex Web sites at the conceptual level [6], and WML (Wireless Markup Language) [7] is a markup language based on XML whose goal is to deliver content and user interface to devices with small displays and limited bandwidth, including cellular phones and pagers, etc. all have their limitations. These interface languages only partially cover the evolution, personalization and variants aspects of a UI, not all, such as WebML supports personalization by using User Modelling.

The existing visualization technologies are not suitable. Our innovative approach, open dynamic conceptual content management (see Figure 2-1: Assets represent entities by [content | concept]-pairs) is designed to cover advanced aspects of entity modelling, including three essential advantages: the evolution, personalization, and variants of a user interface, and including also other additional characteristics, such as logical organization, scalability, flexibility and efficiency.

In this report several different terms are used in describing the new approach and the UI engine or generator, but actually they express the same essential ideas. Here are some of the names used when describing our new approach, for instance, a content-concept based asset management, an asset language, ADL (Asset Definition Language), expressiveness and responsiveness, open and dynamic, conceptual content management, and concept-oriented content management, open and dynamic content management, Open dynamic conceptual content management. The UI engine may be referred to as a GUI (Graphic User Interface) engine, a GUI generator, a compiler, a visualization engine, a UI engine, and a UI visualization engine. We would like to introduce these terms here before discussing them in detail to avoid causing confusion while reading.

Application contents and application concepts are closely linked and represented by a single notation called an asset. An asset language has two properties [8]: expressiveness, which means the entity modelling has to cover three different perspectives, namely an entity's

inherent characteristics, its relationships to other entities and systematise behind the first two perspectives and responsiveness, which means entity modelling processes have to be open and dynamic.

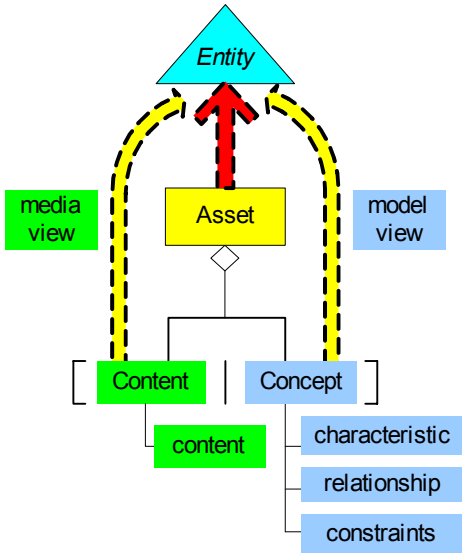


Figure 2-1: Assets represent entities by [content | concept]-pairs

Openness and dynamics together allow an asset management system to be constantly adapted, refined and personalized in a process which converges towards the requirements as demanded by its users' tasks. Since domain models change constantly, a UI of content concept management systems has also to be adapted dynamically.

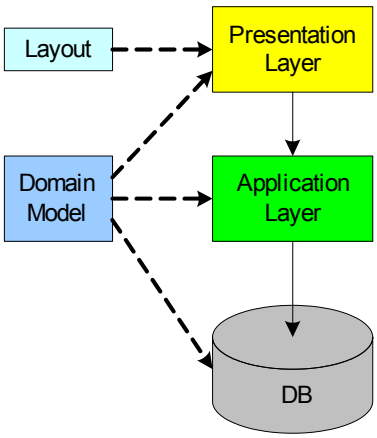


Figure 2-2: Three-tier architecture

An open dynamic conceptual content management system has a three-layer architecture or even more layers (see Figure 2-2: Three-tier architecture). A model compiler has done data layer and application layer. Normally a lower layer change will affect the upper layers. In a traditional implementation of information systems, data layers change constantly, but the presentation layer is not dynamically adaptable. However, an open dynamic content management requires dynamically adaptable UIs. Like the domain model, the presentation of Assets has to be user-definable. In concept-oriented content management systems, the

presentation changes constantly follow dynamic schema changes. Dynamically adaptable UIs cannot be created from a domain model or a compiler only. Some hints or inputs, such as a layout from a user or a screen designer are required.

In contrast to the drawbacks of the classic waterfall model (see Figure 2-3: The waterfall model, source [9]), which are difficult to rework and changes can be expensive, a conceptual content management system uses an IID (Iterative Incremental Development) model (see Figure 2-4: Iterative incremental development model, source [10]). The advantages of an IID model are that it has a better risk management, has no development cycle, delivers complete functionality per slice, and complete testing is done at the end of every slice.

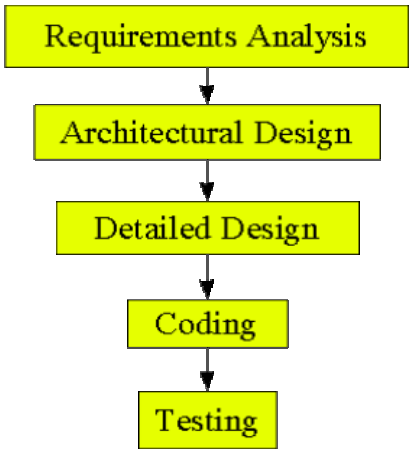


Figure 2-3: The waterfall model

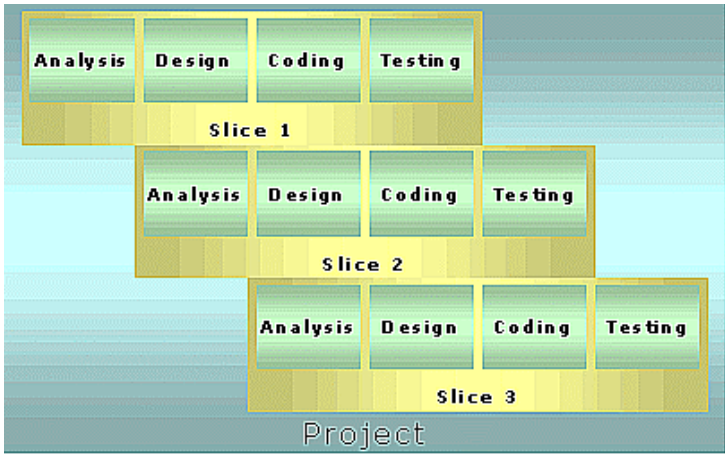


Figure 2-4: Iterative incremental development model

ADL (Asset Definition Language) allows the evolution, personalization and variants of a UI [11]. The following presents a detailed description of these three essential advantages. In order to explain them in an understandable way, one example class `Person` is given here:

```
class Person {
  content name: java.lang.String
          address: Address
  concept relationship visualizedComponentClass : UIComponent
                       visualizedtechnologyClass : UITechnology
}; Person

class Student refines Person {content Id: StudentId}; Student

class Professor refines Person {content dept: Department}; Professor

class Person2 {
  content name: java.lang.String
          address: Address
          age: java.lang.Integer
  concept relationship visualizedComponentClass : UIComponent
                       visualizedtechnologyClass : UITechnology
}; Person2
```

This example will also be used in chapter 5 while analyses different possibilities for the implementation of a UI engine or generator.

First, the evolution of a UI, where the different classes are presented by their corresponding user interfaces. With a changing domain model, the user interface changes too. The evolution is done during modelling time. For example, class `Person` has no content `age`, but class `Person2` does. The user interface `UIPerson`, which corresponds to class `Person`, does not show age information. The user interface `UIPerson2`, which corresponds to class `Person2`, does show age information. When class `Person` is modified to class `Person2`, the user interface `UIPerson1` will also be modified dynamically to the user interface `UIPerson2` (see Figure 2-5: The evolution of a user interface).

Second, the personalization of a UI, which is the definition of content or a presentation style based on user profile data, is the customisation feature for one-to-one content delivery. One class is presented by different user interfaces for several users or one user in the different contexts. Usually users are not willing to explicitly provide data that they do not consider interesting. Personalization is done during modelling time.

There are various possibilities for personalization. First, a user group provides a general UI. Then users adapt it, which means that users do not have to specify the whole UI, for example, users can use a provided frame, but adapt a label. The second alternative is that a user can design his own individual UI for different situations. The third alternative is that various UIs are already provided, so a user simply selects the one that matches his needs. For example, class `Person` can be depicted by two different user interfaces `UIWithStreet`, which shows information for name, street and city, and `UIWithAddress`, which shows information for name and address. A user can select either `UIWithStreet` or `UIWithAddress` according to his requirement (see Figure 2-6: The personalization of a user interface), so using personalization a user is able to define a UI that matches his needs.

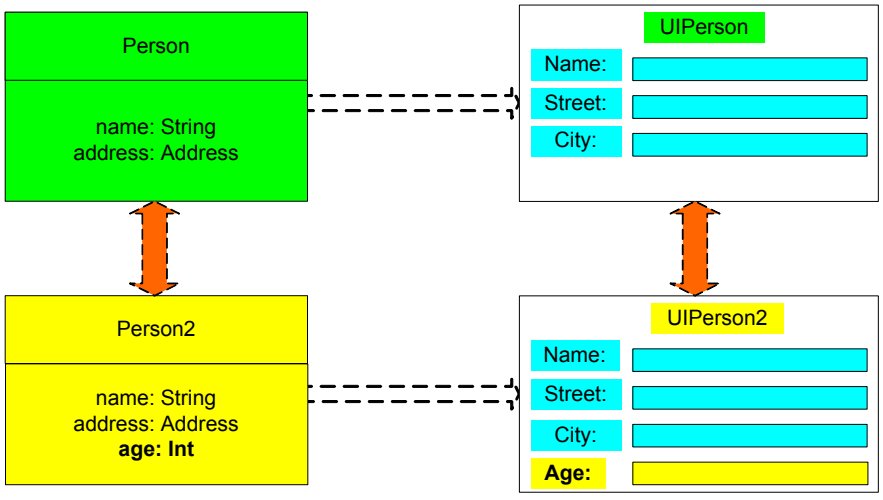


Figure 2-5: The evolution of a user interface

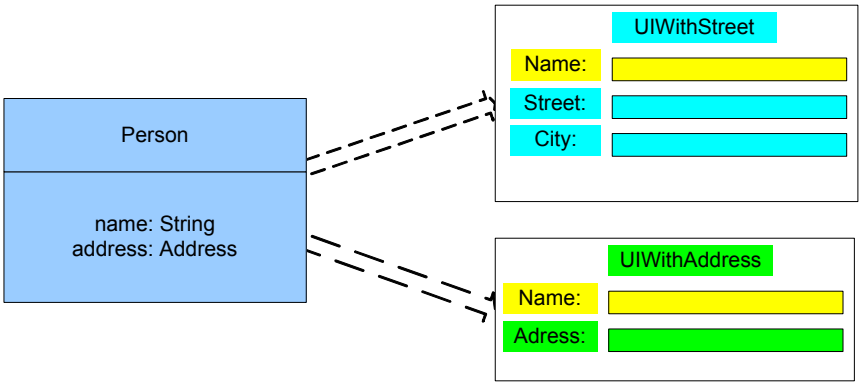


Figure 2-6: The personalization of a user interface

Third, the variants of a UI permit one UI to be adapted to super class and all its subclasses according to their attributes. The variants of a UI mean that a user interface can be adapted according to the class of an asset instance at run time. For example, a class `Person` has two subclasses `Student` and `Professor`. If at run time an object belongs to subclass `Student` or `Professor` that has an attribute `ID`, then the user interface called `UIPerson` will show an Id label and its text field; If an asset to be visualized belongs to the super class `Person` that has no attribute `ID`, then the Id label and its text field will disappear from the user interface `UIPerson` (see Figure 2-7: The variants of a user interface).

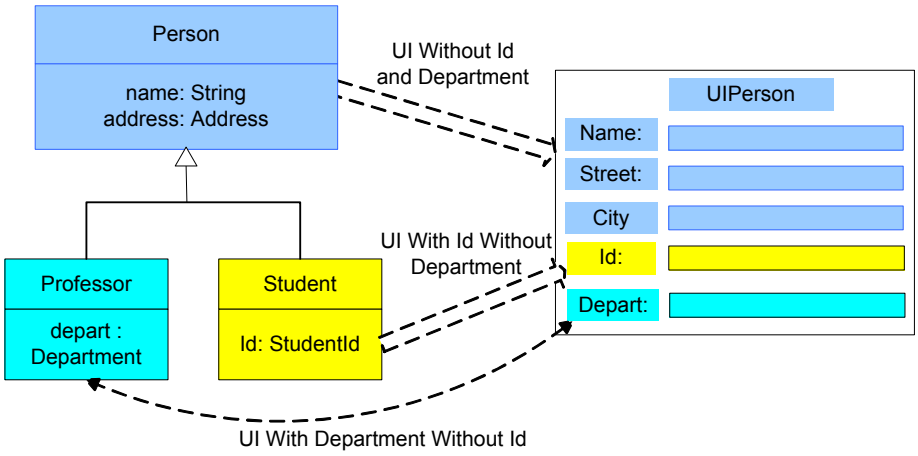


Figure 2-7: The variants of a user interface

2.2 Requirements of a Visualization Engine or Generator

As mentioned earlier, there are three essential contributions of a content-concept based asset language. Content is always associated with its concept and represented by assets. Asset schemata are open so that users can change asset attributes on-the-fly and at any time, thus guaranteeing best correspondence with the entity-at-hand. Asset management systems are dynamic, i.e., the system implementation changes dynamically following any on-the-fly modification of an asset schema.

To achieve dynamically adaptable UIs, a GUI engine or generator requires the following features [12]:

(1) Portability: applications must be portable across many machines and compilers.
(2) Evolution: A user interface has to be adjusted as the observed entities change.
(3) Personalization: The user's expertise influences the user interface needs. A user interface needs to be tailored to the user's needs.
(4) Variants: A user must be able to view a user interface in different contexts. Different user interfaces may be needed in order to adapt to a changing context.
(5) Dynamics: The user interface implementation changes dynamically according to any on-the-fly modification of classes.
(6) Extensibility: A UI visualization engine or generator must be able to work with new appliances and interface technologies. A user interface is extended with extra functionality with the advent of a new visualization technology.
(7) Reusability: when a family of products is evolving, the design for the old devices can be reused in an optimal way.
(8) Scalability: The UI engine or generator has to be designed for environments with large numbers of assets and users.
(9) Consistency: The UI engine or generator must offer consistency of the user interface among different environments and systems.
(10) Richer user interface: meet the needs of different users.
(11) Usability for end users, administrators, and implementers.
(12) Integration: allows the integration of software components from different sources.
(13) Ease to learn and use: The UI engine or generator has to be designed to be easy to learn and use by end users.
(14) Simplicity: The implementation of a UI engine or generator should be simple, not complex.
(15) Accessibility support: A UI engine or generator facilitates interface design for people with disabilities in a natural way [13]. Accessibility for disabled persons may require alternate interface technology, for example, using voice synthesis or Braille. This mandates that a user interface designer create not one, but multiple user interfaces. Thus a platform independent UI engine or generator must allow management of multiple interfaces naturally.

2.3 Possible Approaches

A special UI visualization engine or generator must be designed in order to realize an open dynamic visualization. This UI engine or generator is different from other GUI engines, such as the SwiXML engine that relies strongly on Swing and not open and dynamic. The UI engine or generator has the following essential properties: it is an innovative way that cannot be done only by one existing technologies such as Java or XML. The UI engine or generator is built on an asset-based technology, and use a presentation logic that associates assets from the application domain and the GUI realm. The UI engine or generator generates a user interface and exploits the dynamic openness of an asset management system for user interface adaptation. End users can define a user interface themselves by defining a domain model and a user interface layout (see Figure 2-8: Use case diagram of a user).

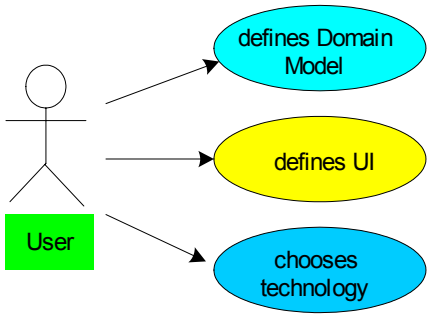


Figure 2-8: Use case diagram of a user

Because the UI engine or generator is built on an asset-based technology and the visualization is realized by a combination of the application domain and the UI realm, the UI engine or generator should work based on a UI components model, a UI technologies model, and a domain model (see Figure 2-9: The working mechanism of the UI visualization engine or generator). Finally, the UI engine or UI generator will generate a user interface to for an end user. The whole working process of the UI engine or generator can be divided into three stages:

(1) First a designer defines the UI components model and UI technologies model based on an asset language. The UI components model defines all components, such as container, window, and view, of the user interface. The UI technology model defines visualization technologies, for instances HTML, Java, AWT and Swing (see Figure 2-10: Case diagram of a domain designer).

(2) Then a user can define an individual domain model and user interface layout, and choose the UI technologies, which are inputs to the UI engine or UI generator at run time (see Figure 2-8: Use case diagram of a user).

(3) Finally the UI engine dynamically renders or a compiler generates a user interface for the end user (see Figure 2-11: Case diagram of a GUI engine or generator).

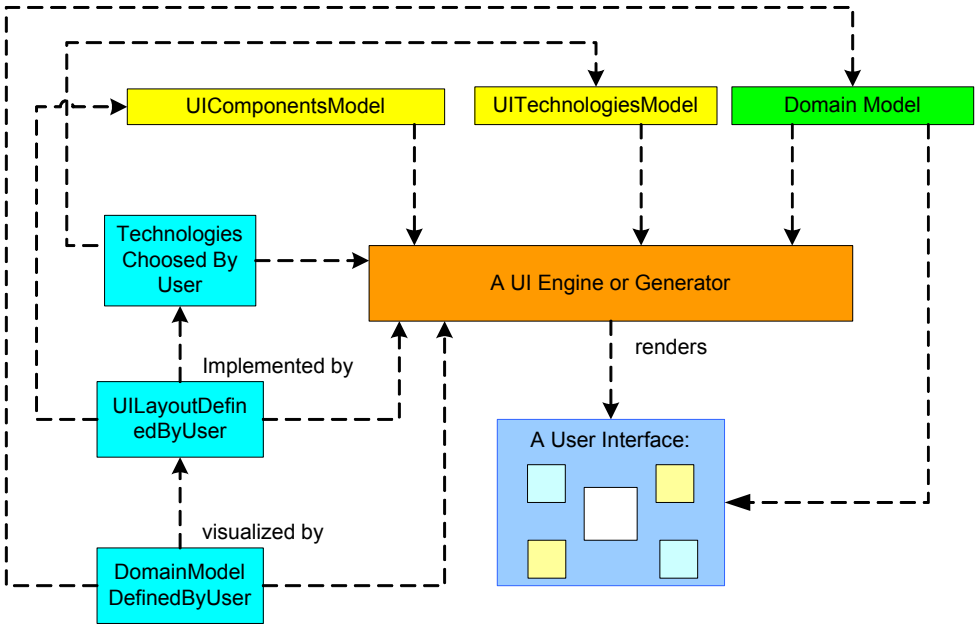


Figure 2-9: The working mechanism of the UI visualization engine or generator

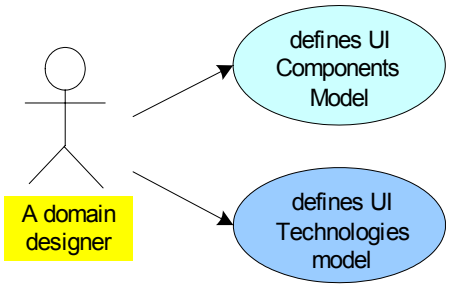


Figure 2-10: Case diagram of a domain designer

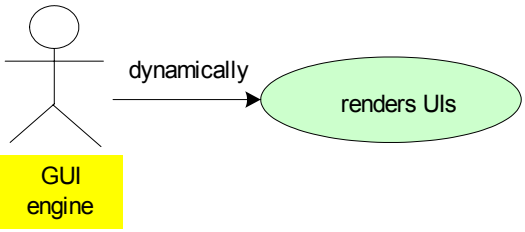


Figure 2-11: Case diagram of a GUI engine or generator

The working mechanism of the UI engine or UI generator will be designed according to the above requirements and properties. All possibilities to implement this UI engine or generator will be found and analysed.

There are several possibilities to implement the GUI engine or GUI generator. The essences of the implementation are as follows:

(1) Class-based implementation of a GUI engine or generator. The following code, for example, shows that `UITechnology` is a class and `Java` is a subclass of `UITechnology`.

```
a. class UITechnology {…}
b. class Java refines UITechnology {…}
```

(2) Instance-based implementation of a GUI engine or generator. The following code, for instance, shows that `UITechnology` is a class and `java` is an instance of `UITechnology`.

```
a. class UITechnology {… }
b. let java := create UITechnology {}
```

(3) Various combinations to implement a GUI engine or UI generator:
   a) UI components represented by classes, UI technologies represented by instances
   b) UI components represented by instances, UI technologies represented by classes
   c) Both UI components and UI technologies represented by classes

Chapter 5 analyses seven selected possibilities to implement the UI visualization engine or UI generator. They are described as follows:

(1) A UI engine or generator creates a UI component based on the type of an asset's content reference.
(2) A Java class is the value of a characteristic of an asset.
(3) An instance of a `Component` is the value of a characteristic of an asset.

(4) An instance of a UI component is the value of a content of an asset.
(5) A combination of technologies represented by instances and components represented by classes.
(6) A different combination of technologies represented by instances and components represented by classes.
(7) Another alternative of combination of technologies represented by instances and components represented by classes.

The following different aspects will be discussed for each alternative in chapter 5. How can a visualization engine or generator work? How can a user define a user interface? What are the advantages and disadvantages of each alternative? How complex would a UI engine or generator be? What are the numbers of asset classes and / or instances of both UI component and UI technology which have to be defined? Finally, the best solution to implement the UI engine or UI generator will be sought.

Before the detailed analysis of the different possibilities to implement a UI engine or generator, it is necessary to first design and define the representations of UI components and UI technologies. The following chapter discusses the visualization components and chapter 4 describes the visualization technologies.

3 Selected Visualization Components

Chapter 2 has already described the contributions, requirements, and possible implementation approaches of a UI engine or generator. Because such a UI engine or generator is based on UI components, UI technologies and a domain model, this chapter discusses the selected visualization components. The appearance and behaviour of the components can generate look and feel of the visualization. After we have researched the existing user interface technologies, such as `javax.swing` [14], `javax.faces, JavaServer Faces` [15], `java.awt` [19], a possible UI components model will be designed and discussed in the following chapters (see Figure 3-1: Components class diagram).

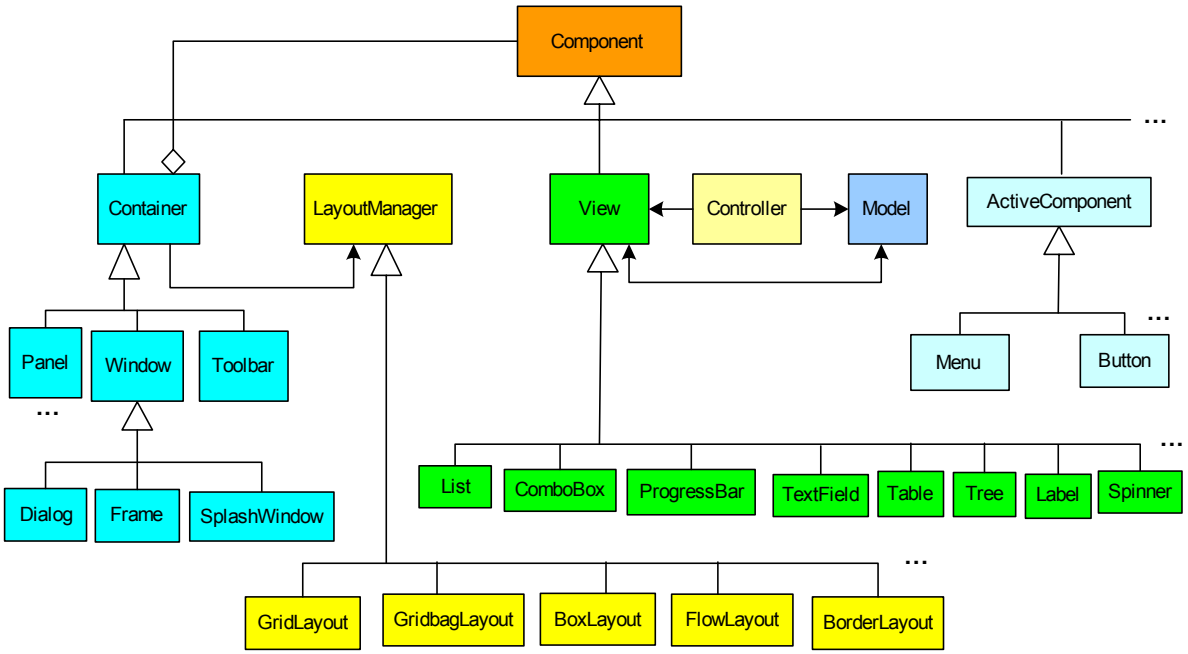


Figure 3-1: Components class diagram

3.1 Container Component

The `Container` is a component class at the top of any containment hierarchy, which holds other components [17]. For example window and panel are containers that can be used under several circumstances (see Figure 3-2: Container class diagram). The `Container` is also associated with the layout manager. The class `LayoutManager` will be discussed in section 3.2 Layout Management.

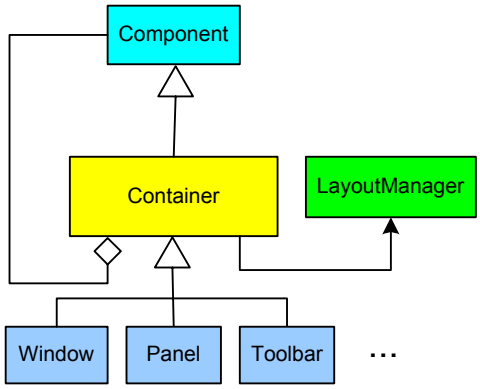


Figure 3-2: Container class diagram

3.1.1 Window Component

A `Window` is a container that is a user interface element that organizes and contains the information, which users see in an application. `Dialog`, `Frame` and `SplashWindow` are direct subclasses of `Window`.

A `Window` can contain a `Frame` component, which has a subclass `InternalFrame` (see Figure 3-3: Frame and Figure 3-4: Internal frame).

Figure 3-3: Frame

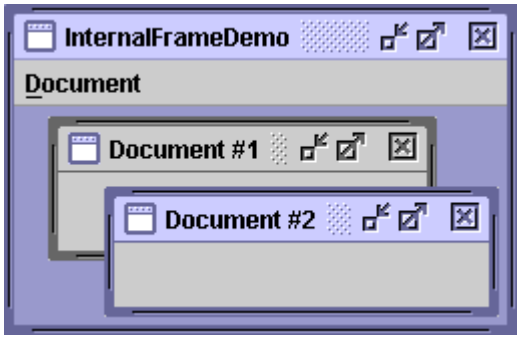


Figure 3-4: Internal frame

A dialog component is a window displayed by an application to gather information from users. Examples of the dialog component include windows that set properties of objects, set parameters for commands, and set preferences for use by the application. A dialog component can also present information, such as displaying a progress bar. A dialog component can contain panes, lists, buttons, and other components (see Figure 3-5: Dialog).

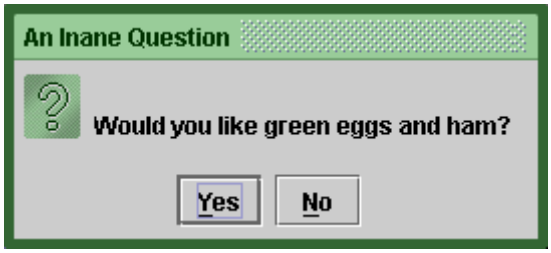


Figure 3-5: Dialog

3.1.2 Panel Component

A direct subclass of `Container` called `Panel` provides general-purpose containers. A panel can be a container for organizing the contents of other components like a `Label`, but a panel component cannot contain a `Window` component. Figure 3-7 shows a label on a panel.

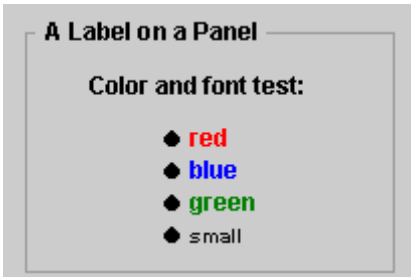


Figure 3-6: Panel

A `ScrollPanel` is a direct subclass of `Panel`. A `ScrollPanel` manages a view point. The following picture demonstrates that a `ScrollPanel` provides a scrollable view of a component.

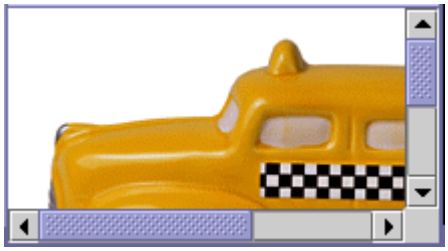


Figure 3-7: Scroll panel

3.1.3 Toolbar Component

A `ToolBar` is a container that groups several components into a row or column. It is a collection of frequently used commands or options. Toolbars typically contain buttons with icons (see Figure 3-8: Tool bar), like a tool bar button, but other components (such as text fields and combo boxes) can be placed in toolbars as well. However, a `ToolBar` cannot contain a `Window` component.

Figure 3-8: Tool bar

3.2 Layout Management

Layout management provides several layout managers (see Figure 3-9: LayoutManager class diagram) [18]. Layout manager is used to determine the size and position of components within a container, which is associated with a layout manager. Each container type has a default layout manager [19].

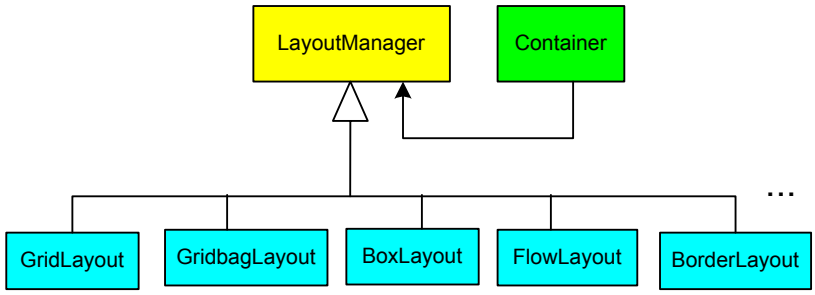


Figure 3-9: LayoutManager class diagram

A strategy pattern is the design pattern for layout management. The reason why a strategy pattern is used will be discussed in subsection 3.2.1.

3.2.1 Design of Layout Managers According to the Strategy Pattern

A Strategy Pattern is a design pattern to encapsulate variants of algorithms. According to Erich Gamma Erich Gamma [20], a Strategy Pattern is intended to define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients who use it. The Strategy Pattern has three participants that include `Strategy`, `Concrete Strategy` and `Context` (see Figure 3-10: The strategy pattern).

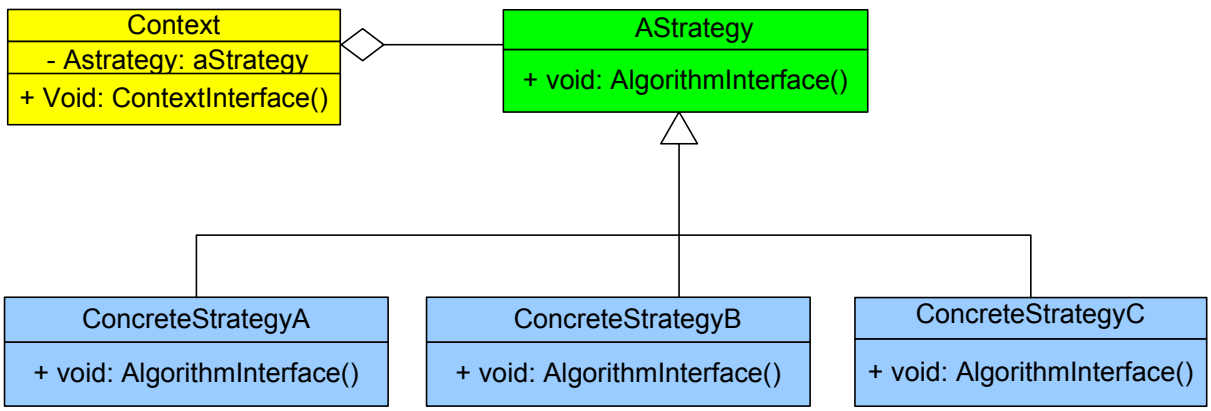


Figure 3-10: The strategy pattern

In the layout management (see Figure 3-9: LayoutManager class diagram), the abstract class called `LayoutManager` is referred to as the `Strategy`, the concrete classes called `GridLayout, GridbagLayout, BoxLayout, FlowLayout, and BorderLayout` are referred to as `Concrete Strategies` and the `Container` is referred to as the `Context` using Strategy.

Benefits of using Strategy Pattern to implement layout management are:

(1) A family of layout management algorithms can be defined as a class hierarchy and can be used interchangeably to alter application behaviour.
(2) By encapsulating the algorithm, new layout management algorithms complying with the same interface can be easily introduced.
(3) A user can switch layout management strategies at run time.
(4) Strategy pattern enables the domain designer to choose the required layout management algorithm without using a "switch" statement or a series of "if-else" statements.
(5) Data structures used for implementing the layout management algorithm are completely encapsulated in `Strategy` class `LayoutManager`. Therefore, the implementation of a layout management algorithm can be changed without affecting the `Context` class `Container`.

Drawbacks of using the strategy pattern to implement the layout management are that a user must be aware of all the strategies to select the right one for the right situation. Strategy base class `LayoutManager` must expose interface for all the required layout management behaviours, which some concrete Strategy classes might not implement.

Five Layout managers, which are called `GridLayout, GridbagLayout, BoxLayout, FlowLayout, and BorderLayout,` are depicted in the sequel.

3.2.2 GridLayout Manager

A `GridLayout` simply makes a bunch of components equal in size and displays them in rows and columns. The following picture shows that a `GridLayout` places components in a grid of cells.

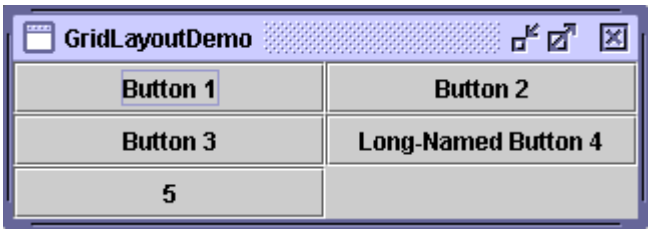


Figure 3-11: A grid layout example

3.2.3 GridBagLayout Manager

A `GridBagLayout` is a sophisticated, flexible layout manager. The following picture demonstrates that a `GridBagLayout` manager aligns components by placing them within a grid of cells, allowing some components to span more than one cell.

Figure 3-12: A grid bag layout example

3.2.4 BoxLayout Manager

A `BoxLayout` manager puts components in a single row or column as shown in the following picture.

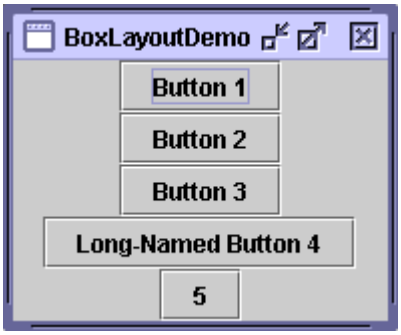


Figure 3-13: A box layout example

3.2.5 FlowLayout Manager

A `FlowLayout` manager simply lays out components in a single row as demonstrated in the following picture. If the horizontal space in the container is too small to put all the components in one row, a `FlowLayout` uses multiple rows. Within each row, components are centered (the default), left aligned, or right aligned as specified when the `FlowLayout` is created.

Figure 3-14: A flow layout example

3.2.6 BorderLayout Manager

A `BorderLayout` manager places components in up to five areas: top, bottom, left, right, and centre. The following picture shows how five different components are put in these five areas.

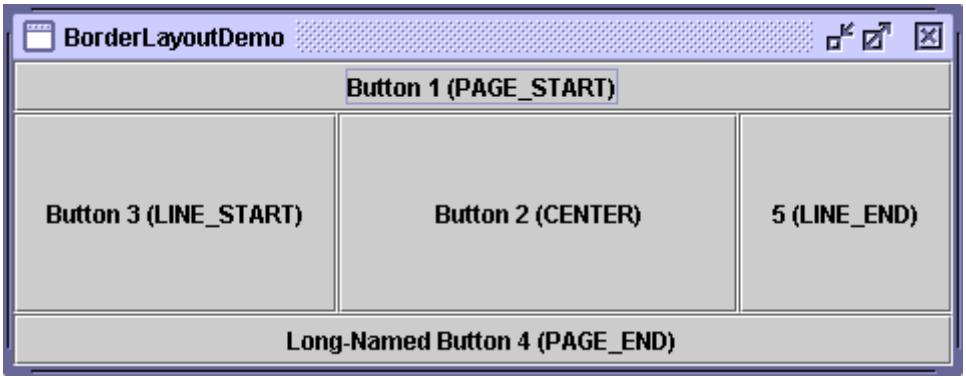


Figure 3-15: A border layout example

3.3 The MVC Design Pattern to Design View Component and Model and Controller

This section discusses `View` component and `ActiveComponent`. The Model-View-Controller (MVC) pattern is used to design `View` and `ActiveComponent`. First we will discuss the reason why MVC pattern is used, then illustrate `View` component and finally explain `ActiveComponent`.

3.3.1 The Model-View-Controller Design Pattern

The Model-View-Controller (MVC) [21] design pattern separates design concerns, decreasing code duplication, centralizing control, and making the application more easily modifiable.

The MVC pattern hinges on a clean separation of objects into one of three categories — models for maintaining data, views for displaying all or a portion of the data, and controllers for handling events that affect the model or view(s).

Because of this separation, multiple views and controllers can interface with the same model. Even new types of views and controllers that never existed before can interface with a model without forcing a change in the model design. The MVC abstraction can be graphically represented as follows (see Figure 3-16: The Model-View-Controller pattern).

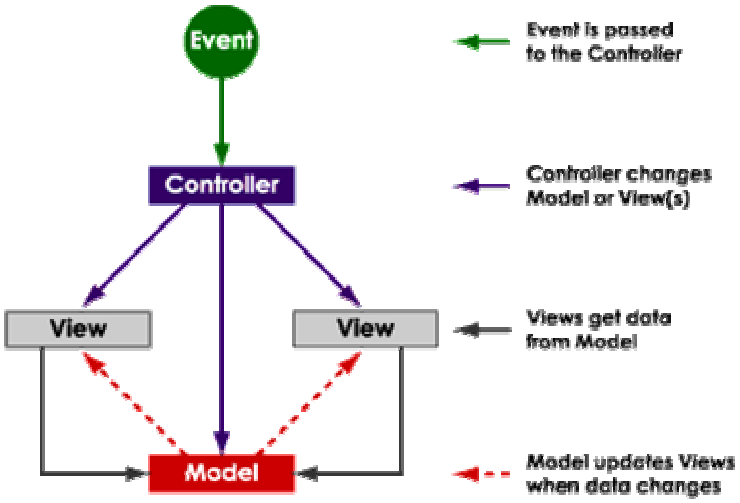


Figure 3-16: The Model-View-Controller pattern

In our case, a user interacts with instances of `ActiveComponent`. The UI engine or generator passes events to the controller. The controller changes the state of a model or view. A model contains assets. A model updates view when assets change. The view accesses the data from the model and draws them on the screen. The controller and model are associated with the attribute `called visualizedAsset` in a `View` component, because `UIComponents` are associated with assets.

The reason for using the MVC design pattern here is that the MVC divides the responsibilities for a user interface into three components thus allowing greater flexibility and possibility for re-use. The MVC also provides a powerful way to organise systems that support multiple presentations of the same information. Consequently, we represent arbitrary assets that are from domain model by generic views that are from component model and control it in a AML (Asset Manipulation Language) way.

However at the abstract level MVC provides a convenient division of the user interface. In practice it is difficult to implement and the result is a highly coupled model, view, and controller components. Coupling decreases the reusability and complicates making interchangeable software components for the user interface. Also each MVC component includes the code to display it, which makes it difficult to display it in more than one way or make global changes in the implementation.

3.3.2 View Component

A `View` component is a specific visual representation of information. in a window (see Figure 3-17: View class diagram). Direct subclasses of `View` have `ComboBox`,`ProgressBar`, `List`, `Spinner`, `TextField`, `Table and Tree`, which are atomic components that exist solely to give the user information.

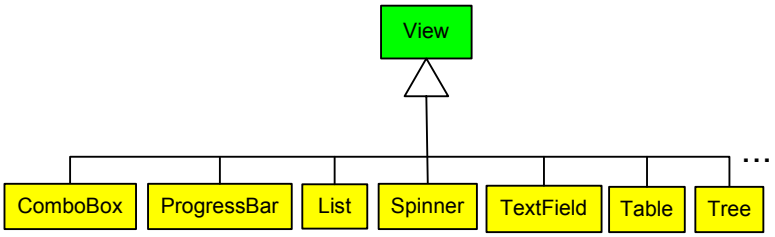


Figure 3-17: View class diagram

Combo box is a class of components with a drop-down arrow that the user clicks to display a list of options. There are two alternatives to implement a combo box component. One alternative is an `ActiveComponent`. The other alternative is a `View` component. As an `ActiveComponent`, combo box, which is called the editable combo box, offers a text field as well as a list of options features. The user can make a choice by typing a value in the text field or by choosing an item from the list. In our choice, combo box is a `View` component. A Combo box lets the user choose one of several choices (see Figure 3-18: Combo box) and is uneditable.

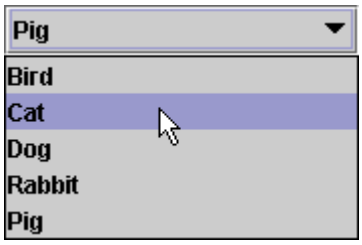


Figure 3-18: Combo box

A `ProgressBar` is a component element that indicates that one or more operations are in progress and show the user what proportion of the operations has been completed (see Figure 3-19: Progress bar).

Figure 3-19: Progress bar

A `List` is a component that presents a user a group of items, displayed in one or more columns, to choose from. Lists can have many items, so they are often put in scroll panel (see Figure 3-20: List). Items in a list can be text, graphics, or both. A List can be used as an alternative to radio buttons and checkboxes.

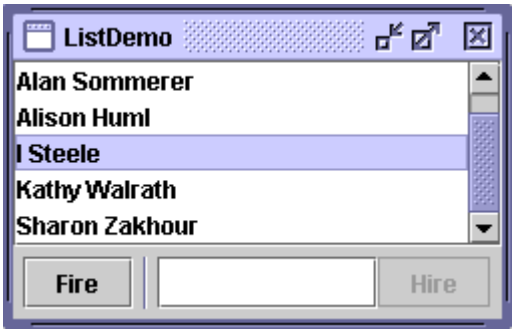


Figure 3-20: List

`Spinners` let the user choose one from a range of values, and generally allow the user to type in a value. Spinners typically provide a pair of tiny arrow buttons for stepping through the elements of the sequence. Here's a picture of an application named `SpinnerDemo` that has three spinners used to specify dates:

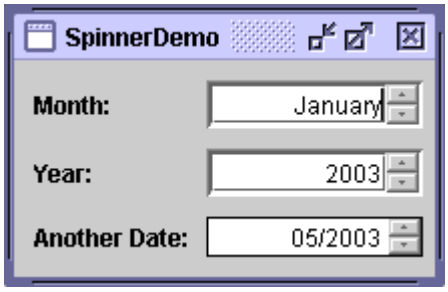


Figure 3-21: Spinner

`TextField`, `Table` and `Tree` can also be designed either as an active component or as a view component. In our design, they are view components that illustrate the information to a user and noneditable.

A `TextField` is a basic text control that lets the user enter a small amount of text (see Figure 3-22: Text field). In a noneditable text field, a user can copy, but not change, the text.

Figure 3-22: Text field and a label

A `Table` can display data. Here's a picture of a typical table displayed within a scroll panel:

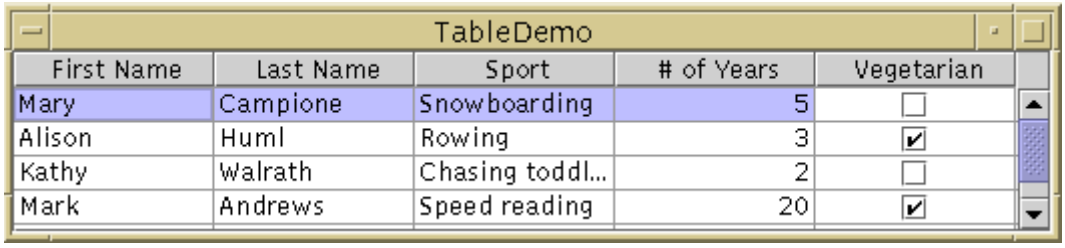


Figure 3-23: Table

A `Tree` is a component that can display hierarchical data. Here's a picture of a tree:

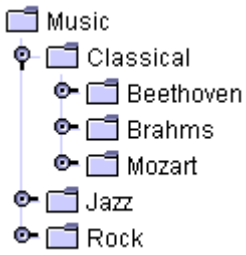


Figure 3-24: Tree

As the preceding Figure shows, each row displayed by the tree contains exactly one item of data, which is called a node. Every tree has a root node from which all nodes descend. By default, the tree displays the root node. A node can either have children or not. Nodes that can have children are branch nodes. Nodes that can't have children are leaf nodes. Branch nodes can have any number of children. Typically, the user can expand and collapse branch nodes (making their children visible or invisible) by clicking them.

3.3.3 Active Components

Active components are components that a user can manipulate to perform an action, choose an option, or set a value (see Figure 3-25: Active component class diagram). Direct subclasses are `Button` and `Menu`. They are atomic components that exist primarily to get input from the user. In our design active component takes a controller role.

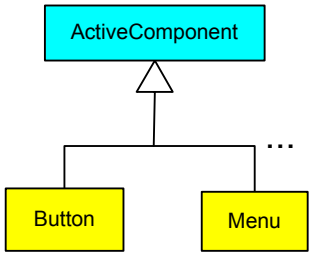


Figure 3-25: Active component class diagram

A `Button` is an interactive component, which can display both text and an image. When a button is disabled, it is shown in a disabled appearance (see Figure 3-26: Button).

Figure 3-26: Button

A `Menu` is a component that provides a space-saving way to let the user choose one of several options (see Figure 3-27: Menu). A list of menu items are logically grouped and displayed by an application so that a user needs not memorize all available commands or options. A menu usually appears either in a menu bar or as a popup menu. A menu bar contains one or more menus and has a customary, platform-dependent location — usually along the top of a window. A popup menu is a menu that is invisible until the user makes a platform-specific mouse action, such as pressing the right mouse button, over a popup-enabled component. The popup menu then appears under the cursor.

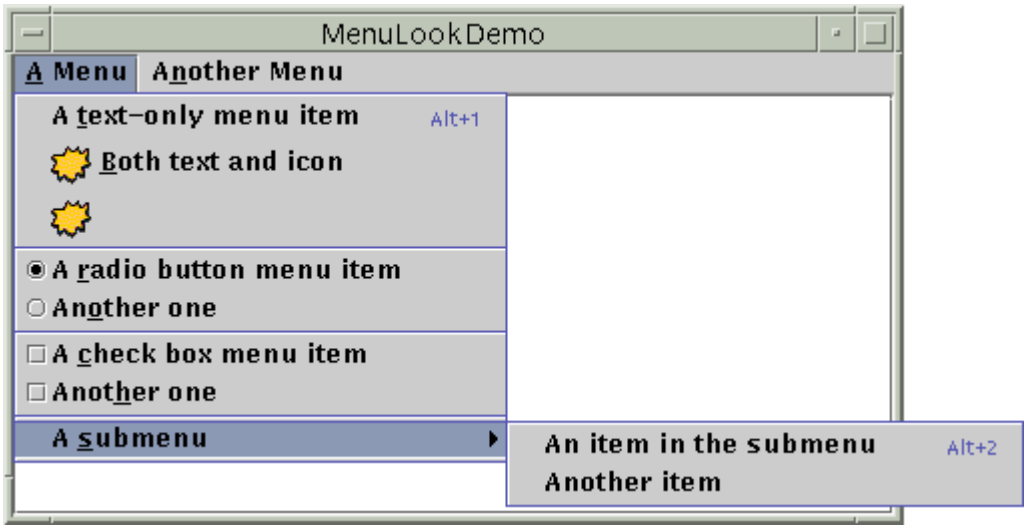


Figure 3-27: Menu

3.4 UI Components

As mentioned earlier, Figure 3-1, “Components class diagram”, shows some examples of UI components. This is one possible model, which demonstrates selected UI components. UI components are implementation dependent. However, they do not rely on any existing user interface technologies. The UI components model is extendable because it is convenient to add a new UI component into this model and / or delete a UI component from the model.

4 Visualization Technologies

As described in chapter 2, the UI engine or generator works based on a UI components model, a UI technologies model and a domain model. The UI components have already been discussed in chapter 3. This chapter discusses the UI technologies. First, we will look at the existing visualization technologies, then analyse their advantages and disadvantages. Afterwards we will discuss the technologies that a UI engine or generator supports.

There are several ways to create user interfaces (UIs) for Web and network applications. Initially there were markup languages: Dynamic HTML, or DHTML, and XML-based User Interface Language (XUL) [4] for traditional desktop applications; then Wireless Markup Language (WML) [7] for mobile devices such as cell phones with display. The growing popularity of the Extensible Markup Language (XML) [22] promises even more languages. In addition, there are traditional programming and scripting languages (e.g., Java, JavaScript, and Visual Basic and C++ through Active-X). These visualization technologies can be classified into four categories (see Figure 4-1: Existing visualization technologies diagram): Layout Description Languages, Tool-based Programming, UI Libraries for Programming Languages, Server Script and Browser Script. The following is a short description of each type of UI technology.

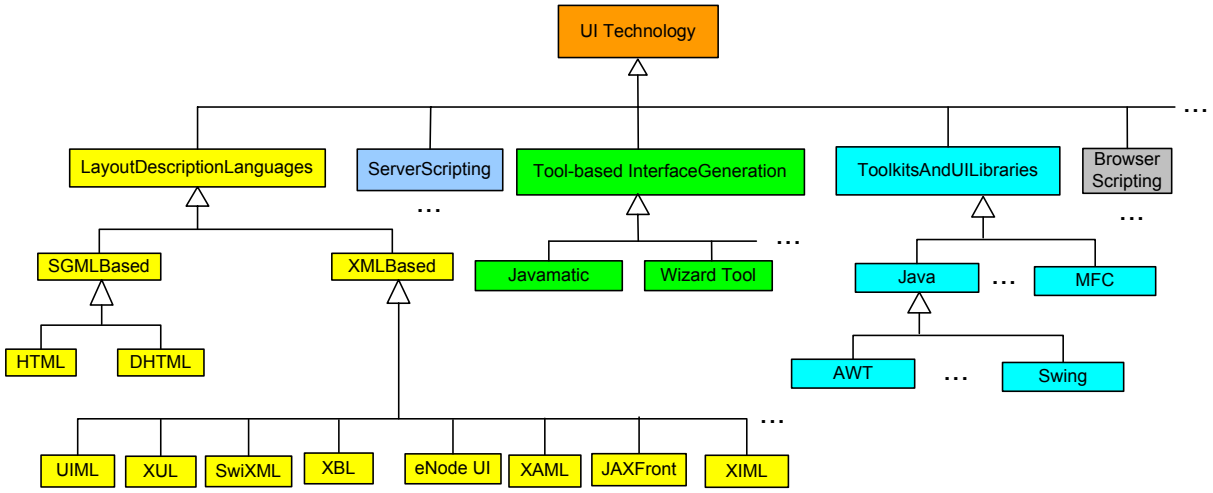


Figure 4-1: Existing visualization technologies diagram

4.1 Layout Description Languages

Layout description languages can be divided into two main categories: SGML-based and XML-based layout description languages. SGML and XML are the two most popular syntax standards for markup languages. First, we will discuss the SGML-based layout description languages, especially the advantages and disadvantages of HTML. Then we will discuss XML-based layout description languages, such as SwiXML, UIML (User Interface Markup Language), XUL (XML-based User Interface Language) and XBL (XML Binding Language).

4.1.1 SGML-based Layout Description Languages

SGML (Standard Generalized Markup Language) is a language for describing markup languages, particularly those used in electronic document exchange, document management, and document publishing. SGML has been in existence since the mid-80s but never received acceptance beyond the information retrieval community mainly due to its complexity. SGML-based layout description languages include HTML and DHTML (see Figure 4-1: Existing visualization technologies diagram).

The HyperText Markup Language (HTML) [23] is an example of a language defined in SGML. HTML is a language based on a document composition style known as "markup." HTML outlines a hypertext structure, which is the publishing language of the World Wide Web. HTML 4.0 is an SGML application conforming the International Standard ISO 8879. HTML 4.0 introduced Cascading Style Sheets (CSS) and the Document Object Model (DOM). CSS gives a style and layout model for HTML documents. The DOM gives a document content model for HTML documents.

Dynamic HTML or DHTML [25] is a combination of technologies to make Web pages dynamic by interaction of HTML, CSS and XSL (XML Style sheets Language) style sheets, the Document Object Model, and scripting. With DHTML, a Web developer can control how to display HTML elements in a browser window.

The advantages of HTML are that it is simple and easy to learn. HTML is portable, especially over networks. HTML pages that are textual files written in HTML are the most popular resources requested on the Web. The disadvantages of HTML are that Portability is limited in reality because of vendor-specific dialects. In HTML, the structure and the content are mixed together. HTML is insufficient for large and complex applications, due to its fixed set of tags and limited graphic capabilities.

4.1.2 XML-based Layout Description Languages

The Extensible Markup Language (XML) [26] describes a class of data objects called XML documents. XML is so-called application profile or restricted form of SGML. By construction, XML documents are conforming SGML documents. Its goal is to enable generic SGML to be served, received, and processed on the Web in the way that is now possible with HTML. XML has been designed for ease of implementation and for interoperability with both SGML and HTML.

Many of the new declarative languages obtain their syntax from XML. XML facilitates the creation of new vocabularies that describe domain-specific content and context, organized into hierarchical information structures. XML has become the official meta-language for information on the Internet. It is a meta-language because it can be used to define other languages that are relevant to various application domains by providing a common syntax.

There are some XML-based user interface definition languages (see Figure 4-1: Existing visualization technologies diagram), such as SwiXML, UIML, XUL, XBL, XAML, eNode UI, and XHTML. The following is a description of the advantages and disadvantages of several selected XML-based user interface definition languages.

4.1.2.1 SwiXml

SwiXml [3] is a small GUI generating engine for Java applications and applets. Graphical User Interfaces are described in XML documents that are parsed at runtime and rendered into `javax.swing` objects. The `SwingEngine` class is the rendering engine, which is able to convert an XML descriptor into a `javax.swing` UI.

The following is one example of SwiXML (see Figure 4-2: One SwiXML example):

```
<?xml version="1.0" encoding="UTF-8"?>
<frame size="640,480" title="Hello SWIXML World"
     DefaultCloseOperation="JFrame.EXIT_ON_CLOSE">
  <panel constraints="BorderLayout.CENTER">
    <label LabelFor="tf" Font="Comic Sans MS-BOLD-12" Foreground="blue"
       text="Hello World!"/>
    <textfield id="tf" Columns="20" text="Swixml"/>
    <button text="Click Here" Action="submit"/>
  </panel>
</frame>
```

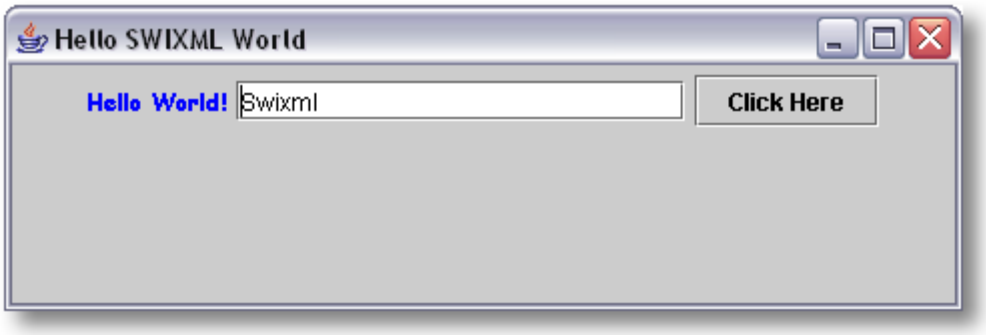


Figure 4-2: One SwiXML example

The advantages of SwiXML are that SwiXml allows developers to generate graphical user interfaces by writing XML documents defining the layout and content of the interfaces. These XML documents are parsed at runtime and rendered into `javax.swing` objects by a rendering engine. SwiXML frees the developer from programming by using the `javax.swing`. Programmers who know `Swing` already can immediately start writing descriptors: Class names are translated into tag names and method names into attribute names.

The disadvantages of SwiXML are that SwiXml relies completely on `javax.swing`. It doesn't free the developer from knowing the `javax.swing` package. The dynamic behaviour of the user interface has to be coded in Java.

4.1.2.2 UIML (User Interface Markup Language)

The User Interface Mark-up Language (UIML) [2] is a language for describing user interfaces in a device-independent manner. However, the UI designer must still design separate UIs for each device, and then represent those designs in UIML. UIML does not magically create multiple UIs from a single description. Instead it is a language in which those multiple UIs

can be recorded. UIML describes the appearance of a UI, the user interaction with the UI, and how the UI is connected to the application logic.

UIML is an interface meta-language that is based on the MIM model (Meta-Interface Model). The MIM model (see Figure 4-3: Meta-Interface model diagram, source [2]) is designed to describe generic interfaces that map to multiple devices and can connect to a wide range of application technologies.

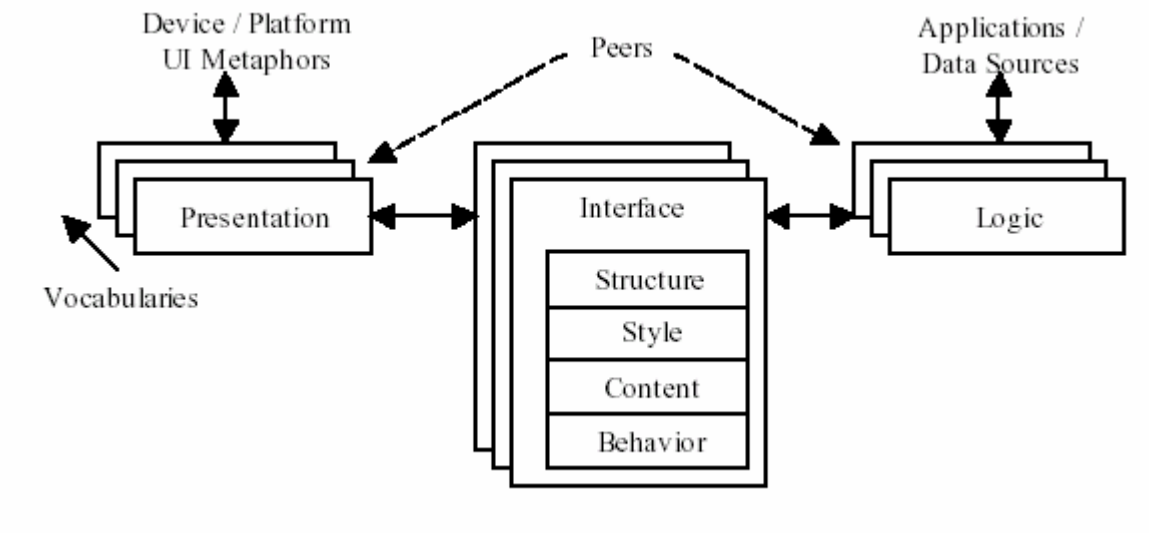


Figure 4-3: Meta-Interface model diagram

MIM divides the interface into three major components: presentation, logic, and interface. The logic component provides a canonical way for the user interface to communicate with an application while hiding information about the underlying protocols, data translation, method names, or location of the server machine. The presentation component provides a canonical way for the user interface to render itself while hiding information about the widgets and their properties and event handling. The interface component describes the dialogue between the user and the application using a set of abstract parts, events, and method calls that are device and application independent.

MIM subdivides the interface component into four additional subcomponents: structure, style, content, and behaviour. The structure describes the organization of the parts in the interface, the style describes the presentation specific properties of each part, the content describes the information that is presented to the user, and the behaviour describes the runtime interaction (including events and application method calls).

UIML factors the interface into the following five components: structure, style, content, behaviour, and peers according to the MIM model. The first four describe the interface and are grouped under the interface component. The last one describes the connections to the presentation and to the application logic. Here is a skeleton of a UIML document [27]:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE uiml PUBLIC "-//Harmonia//DTD UIML 2.0 Draft//EN" "UIML2_0e.dtd">

<uiml xmlns='http://uiml.org/dtds/UIML2_0e.dtd'>
      <head> ... </head>
```

```
    <interface>
        <structure>… </structure>
        <style>… </style>
        <content>… </content>
        <behaviour>… </behaviour>
    </interface>
    <peers> ... </peers>
    <template> ... </template>
</uiml>
```

The four elements head, interface, peers, and template may appear in any order. A UIML document must contain at least an interface element to be rendered.

A UIML interface element may contain multiple structure, style, content, or behaviour elements, provided that each one can be uniquely identified by name. Multiple structure, style, and behaviour elements allow reuse of the interface across different families of devices. Multiple content elements allow reuse across different applications.

UIML includes a `peers` element that specifies what widgets in the target platform and what methods or functions in scripts, programs, or objects in the application logic are associated with the user interface. In UIML, all the device and toolkit information is isolated in the `peers` element. This information is used by a UIML rendering engine to resolve all the names from the property, call, and event elements into actual widgets, methods, and events.

The following is one UIML example (see Figure 4-4: One UIML example, source [2]). The example displays a single window that represents a possible login screen for an application on a web site. The screen contains a header, two input fields (for the name and PID), and three buttons (to accept, to clear the input, and get help). The UIML document is rendered using the Java AWT toolkit.

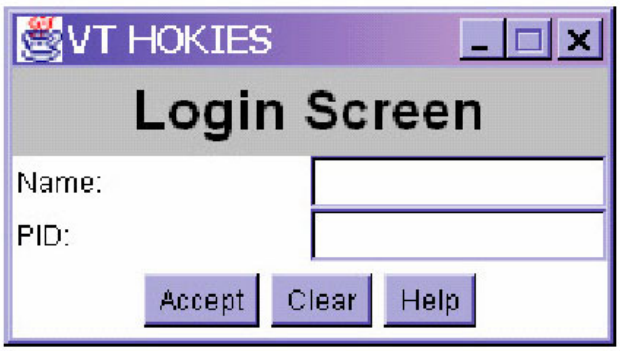


Figure 4-4: One UIML example

In this example, the main container is rendered as a frame (`java.awt.Frame`). We specify the title, the layout manager, and whether the user can resize the frame or not. For the label "Title" (`java.awt.Label`) we specify the font, the text alignment, the background and foreground colors, the text inside the label, and the alignment within the parent frame. For the center panel "`CenterPanel`" (`java.awt.Panel`) we specify the layout manager and its properties, and the alignment within the parent frame. For the input fields and their labels ("`NameL`", "`NameT`", "`PIDL`", "`PIDT`") (`java.awt.TextField`) we specify the label text and the number of characters allowed in each textfield. Finally, for the three buttons ("`Accept`",

"Reset", "Help") (java.awt.Button) we specify the text on them. The following is the complete UIML source code for the example.

```
<uiml>
  <interface name="Simple">
    <structure>
      <part name="Top_TimeTable" class="Frame">
        <part name="Title" class="Label"/>
        <part name="CenterPanel" class="Panel">
          <part name="NameL" class="Label"/>
          <part name="NameT" class="TextField"/>
          <part name="PIDL" class="Label"/>
          <part name="PIDT" class="TextField"/>
        </part>
        <part name="Actions" class="Panel">
          <part name="Accept" class="Button"/>
          <part name="Reset" class="Button"/>
          <part name="Help" class="Button"/>
        </part>
      </part>
    </structure>

    <style>
     <property part-name="Top_TimeTable"
       name="title">VT HOKIES</property>
     <property part-name="Top_TimeTable"
       name="layout">java.awt.BorderLayout</property>
     <property part-name="Top_TimeTable"
       name="resizable">false</property>

     <property part-name="Title"
       name="borderAlignment">North</property>
     <property part-name="Title"
       name="font">Dialog-Bold-24</property>
     <property part-name="Title"
       name="text">Login Screen</property>
     <property part-name="Title"
       name="alignment">CENTER</property>
     <property part-name="Title"
       name="background">lightGray</property>
     <property part-name="Title"
       name="foreground">black</property>

     <property part-name="CenterPanel"
       name="borderAlignment">Center</property>
     <property part-name="CenterPanel"
       name="layout">java.awt.GridLayout</property>
     <property part-name="CenterPanel"
       name="layout_columns">2</property>
     <property part-name="CenterPanel"
       name="layout_rows">0</property>

     <property part-name="Actions"
       name="borderAlignment">South</property>

     <property part-name="NameL" name="text">Name:</property>
     <property part-name="PIDL" name="text">PID:</property>
     <property part-name="NameT" name="columns">15</property>
     <property part-name="PIDT" name="columns">15</property>
```

```
      <property part-name="Accept" name="label">Accept</property>
      <property part-name="Reset" name="label">Clear</property>
      <property part-name="Help" name="label">Help</property>
    </style>

    <behaviour>
      <rule>
       <condition>
          <event class="actionPerformed" part-name="Accept"/>
       </condition>
       <action>
         <property part-name="Top_TimeTable"
           name="exists">false</property>
       </action>
      </rule>
    </behaviour>
  </interface>
</uiml>
```

Now let us look at the advantages and disadvantages of UIML. UIML is a declarative language. It describes user interfaces in an appliance-independent manner. For example, the interfaces for different appliances can be generated from a single UIML description with different style sheets. UIML is not claimed that it covers the evolution, personalization and variants aspects of a user interface. It may realize the personalization of a UI based on the MIM model. However, It can be seen that the UIML language structure decides that a user interface cannot be adapted dynamically according to class of an instance at run time.

4.1.2.3 XUL (XML-based User Interface Language) and XBL (XML Binding Language)

XUL is glossed alternately as "XML-based User Interface Language," "XML User Interface Language," and "Extensible User Interface Language." XUL is an interface definition language associated with the `Mozilla XPToolkit` Project [5]. XUL is an XML-based language for describing the contents of windows and dialogs. XUL was created by the `Mozilla` community to simplify the user interface development for new applications running under the Netscape Web browser. XUL separates the user interface into four parts: content (structure and description of UI elements), appearance (look, feel, skin, and themes), behaviour and locale (localization information for internationalisation).

XUL has built-in user interface widgets, but creating additional custom widgets needs a related language called the Extensible Bindings Language (XBL).

XBL (XML Binding Language) is used for declaring the behaviour of XUL widgets. XBL is a markup language for describing bindings that can be attached to elements in other documents. Bindings can be attached to elements using either cascading style sheets (CSS) or the document object model (DOM). The element that the binding is attached to, called the bound element, acquires the new behaviours specified by the binding. Bindings can contain event handlers that are registered on the bound element, an implementation of new methods and properties that become accessible from the bound element, and anonymous content that is inserted underneath the bound element [28].

The following shows the basic skeleton of an XBL file:

```
<!ENTITY % bindings-content "(binding|script|stylesheet)*">
```

```
<!ELEMENT bindings %bindings-content;>
<!ATTLIST bindings
   id               ID           #IMPLIED
   type             CDATA        #IMPLIED
>
```

```
<?xml version="1.0"?>
<bindings xmlns="http://www.mozilla.org/xbl">
  <binding id="binding1">
    <!-- content, property, method and event descriptions go here -->
  </binding>
  <binding id="binding2">
    <!-- content, property, method and event descriptions go here -->
  </binding>
</bindings>
```

The bindings element is the root element of an XBL document. It contains zero or more binding elements as children. Each binding child element defines a unique binding that can be attached to elements in other documents. The bindings element can also contain script and style sheet elements as children. These specify scripts and style sheets that are used by the bindings.

The id attribute is a document-unique identifier. The value of this identifier is often used to manipulate the element through a DOM interface (e.g., using `document.getElementById`).

The type attribute specifies the scripting language used by all bindings in the document. Bindings can selectively override this default by specifying type attributes of their own.

An XBL file contains a set of bindings. Each binding describes the behaviour of a XUL widget. For example, a binding might be attached to a scroll bar. The behaviour describes the properties and methods of the scroll bar in addition to describing the XUL elements that make up a scroll bar.

The disadvantages are that XUL is an integral part of the `Mozilla` browser. XBL and XUL rely strongly on each other. XUL and XBL are not claimed that they cover the evolution, personalization, and variants characteristics of a user interface. Let us look at the structure of the XUL and XBL; it seems difficult to realize the variants of a user interface, which means that a UI is adapted dynamically according to class of instances at run time.

4.1.2.4 XAML (Microsoft Extensible Application Markup Language)

XAML (Extensible Application Markup Language) [29] is a code name for the Microsoft “Longhorn” Markup Language. It is a new scripting language based on XML and enables developers to specify a hierarchy of objects with a set of properties and logic. The main purpose of XAML is to bring both Windows and Web programming worlds together. Here one example, create a file named `HelloWorld.xaml` with the following content:

```
<?xml version="1.0" standalone="yes"?>
<Window>
   <Button>Hello World</Button>
</Window>
```

Open this XAML file in Windows Longhorn browser as follows (see Figure 4-5: One XAML Example, source [29]):

Figure 4-5: One XAML Example

XAML uses .NET - C# as a script language. There are two ways to attach code to XAML events: either write C# directly into the XAML within a CDATA tag, or write the code in a separate file. The following is one example to create a button.

```
<Canvas ID="root"
xmlns="http://schemas.microsoft.com/2003/xaml"
xmlns:def="Definition">
      <Button>Click Me!</Button>
</Canvas>
```

However, if we want some event to occur when users click the button we must use code behind or within the markup to handle the click event. The following example shows a code block inside an "XAML" file. When the button is clicked its background becomes red.

```
<Canvas ID="root"
xmlns="http://schemas.microsoft.com/2003/xaml"
xmlns:def="Definition">
      <Button Click="Button_Click">Click Me!</Button>

<def:Code>
  <![CDATA[

    void Button_Click(object sender, ClickEventArgs e)
    {
        btn1.Background = Brushes.Red;
    }
  ]]>
</def:Code>
</Canvas>
```

We can also place event-handling code in a file separate from the "XAML" file, called a "code-behind" file. The following example creates the same application as the previous example but the code is in two files—an "XAML" file and a C# code-behind file.

"XAML" file

```
<Canvas ID="root"
xmlns="http://schemas.microsoft.com/2003/xaml"
xmlns:def="Definition">
      <Button Click="Button_Click">Click Me!</Button>
</Canvas>
```

C# code-behind file

```
using System;
using System.Windows;
using System.Windows.Controls;
```

```
using System.Windows.Media;
namespace Button
{
      public class Default : Panel
      {
    // Event handler
      void Button_Click(object sender,
System.Windows.Controls.ClickEventArgs e)
         {
          btn1.Background = System.Windows.Media.Brushes.Red;
         }
      }
}
```

Comparison of Figure 4-2: One SwiXML example with Figure 4-5: One XAML Example, it can be seen that both UIs are similar. However, they are based on different languages. SwiXML relies on `javax.swing`. Graphical User Interfaces are described in XML documents that are parsed at runtime and rendered into `javax.swing` objects. XAML uses .NET - C# as a script language. SwiXML is simpler than XAML.

The disadvantages of XAML are that XAML complies with only Microsoft Windows platform. It does not free users from knowing .NET - C# languages. As mentioned earlier, the main goal of XAML is to bring both Windows and Web programming worlds together. XAML is not claimed to focus on the evolution, personalization, and variants of a UI, which have been already discussed in chapter 2 Requirements.

4.1.2.5 XIML (Extensible Interface Markup Language)

XIML (Extensible Interface Markup Language) [30] is an XML-based interface representation language for universal support of functionality across the entire lifecycle of a user interface: design, development, operation, management, organization, and evaluation.

XIML is an XML-based language that provides a framework for the definition and interrelation of interaction data items. Figure 4-8 (source [30]) shows the basic representational structure of the XIML language. The XIML language includes the following representational units:

Components: XIML is an organized collection of interface elements that are categorized into one or more major interface components. These components are those typically found in an interface model: user tasks, domain objects, user types, presentation elements, and dialog elements.
Relations: A relation in XIML is a definition or a statement that links any two or more XIML elements either within one component or across components. By capturing relations in an explicit manner, XIML creates a body of knowledge that can support knowledge-based design, operation, and evaluation functions for user interfaces.

Attributes: In XIML, attributes are features or properties of elements that can be assigned a value. The value of an attribute can be one of a basic set of data types or it can be an instance of another existing element.

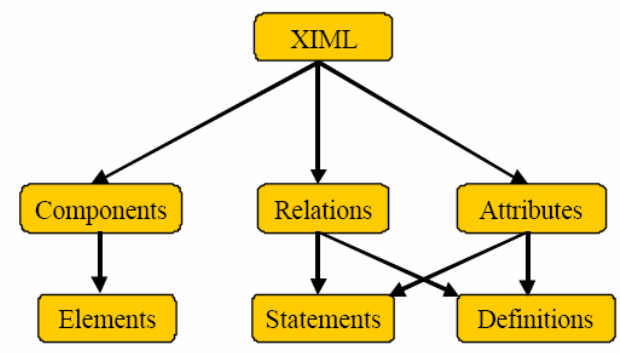


Figure 4-6: The Structure of XIML

XIML provides a framework for the development of user interfaces that have multiple target displays. There are various widgets available for personalization in an XIML specification. However, XIML is not claimed that it covers the evolution and variants of a UI that have been already discussed in chapter 2 Requirements.

4.1.2.6 eNode UI Markup Language

The eNode UI Markup Language [31] is used to describe user interfaces that may be difficult or impossible to describe using HTML and JavaScript; User interfaces can be reconstructed from markup data using a process called object realization. An eNode Object Realizer can realize objects from resource descriptions. By default, the realized form of a frame element is an instance of `javax.swing.JFrame`, and that of a label element is an instance of `javax.swing.JLabel`. eNode UI Markup Language defines the default mapping between an element type and the class used to realize an element of that type. It is easy to override this default mapping and substitute a different class, perhaps one that is user-defined, to realize an element. This can be done directly in the markup, on a per-element basis.

The eNode UI Markup Language is simple. The disadvantage of eNode UI Markup Language is that it relies on `javax.swing`. It is not claimed that the eNode UI Markup Language provides the properties such as the evolution, personalization, and variants of a user interface that have been described earlier.

4.1.2.7 JAXFront

JAXFront [32] generates the graphic user surface on the basis of an XML Schema. Its business model consists of XML Schema as well as a XML instance. The XML Schema describes the syntactic requirements of the business model, while the XML instance represents the described model. JAXFront analyses the business data structures from the XML Schema and provides a generic graphical user interface at run-time.

The presentation logic is partitioned in layout and behaviour ranges. The layout determines the appearance and the arrangement of the graphic elements, whereby the behaviour describes rules and conditions of the graphical front end.

There are two renderers that generate the graphical user components. The Java Renderer uses the Java Swing graphic toolkit for the creation of graphical user interfaces. These visual components are all subclasses of JComponent. The HTML Renderer creates HTML forms based on a XML Schema.

Using a Java Client, JAXFront is integrated in an existing Client Framework running in the presentation layer. Providing a client in terms of HTML user interfaces, JAXFront needs to be embedded in an existing server infrastructure.

JAXFront relies on `Swing` and HTML. It may cover the personalization of a user interface according to the system structure of JAXFront. However, it is not claimed that JAXFront focuses on the evolution and variants of a UI that have been discussed in chapter 2.

4.2 Tool-based Interface Generation

This section discusses tool-based interface generation. First, we will look at model-based tools. Afterwards we will discuss some direct manipulation tools.

Model-based tools reported in the literature include: Mickey [37], DON [38], UIDE [39], HUMANOID [40], ITS [41], Javamatic [42]. The following is a brief description of some of these tools.

UIDE [39] is a system with similar features. UIDE places its emphasis on describing the effects of commands and the application supports, and not the interface. The user interface description includes pre- and post- conditions of the operations that the system uses to automatically generate the interface.

Humanoid [40] uses the following dimensions in the model of how an interface should look and behave: application semantics, presentation templates (style), behaviour, dialog sequencing, and action side effects. The applications semantics refer to the objects and operations of the application domain. The presentation templates refer to the visual appearance of the interface (as defined by widgets). The behaviour refers how the user interacts with the presentation objects. The dialog sequencing refers to how commands are organized (usually with ordering constraints). The action side effects refer to what actions are executed automatically after a command.

ITS [41] is a system that uses design rules to generate an interface. The ITS architecture separates the application into four layers. The action layer implements backend application functions, the dialog layers defines the content of the user interface independent of its style, the style rule layer defines the presentation and behaviour of a family of interaction techniques, and the style program layers implements primitive toolkit objects that are composed by the rule layer into complete interaction techniques. ITS considers content as the objects that are included in each frame of the interface, the flow of control among frames, and the actions associated with each object. Example style programs include routines to format text, render images, and arrange units in rectangular layouts.

Javamatic [42] is an automated generation tool. Javamatic implements a method that allows programmers to add a Web-based graphical interface to command-line driven applications

without programming. Javamatic uses a high level description of an application to automatically generate a user interface, and then invokes commands in the legacy application transparently. Javamatic does not require any changes to the application code, nor does it require application recompilation with special toolkits. The application can be written in any programming language (compiled or interpreted) as long as the needed functionality is accessible from the command-line. Javamatic is written entirely in the Java language. Javamatic can add a modern GUI to legacy applications, can make them accessible on platforms to which the code has not been ported (e.g., scientific codes on supercomputers can be run from personal computers), can make them Web accessible through regular Web pages, and can permit collaboration between geographically separate users, because they share a single program and its associated data.

Direct manipulation tools can be subdivided into four categories: Prototyping tools, Wizard (sequence of cards) tools, Interface builders, and Graphical editors.

The prototyping tools allow the designer to quickly mock up how the interface looks for certain scenarios but cannot create the real user interface. These tools are different from "rapid prototyping" tools that can create workable user interfaces.

The wizard tools are tools for developing user interfaces that exhibit sequential behaviour. The user traverses a sequence of screens and the final screen shows the result. Each screen contains a set of widgets, which can be static (fixed set of widgets) or dynamic (set of widgets depends on previous responses from the user). The wizard tools usually allow the designer to create both static screens (each screen individually) and dynamic screens (using a template with embedded scripts).

Interface builders allow the designer to build the interface using direct manipulation. The user selects a widget from the list of available widgets (associated with a particular toolkit) and places them on a drawing area using a pointing device. The system then generates code that is compiled with the rest of the application. An example of an interface builder is "Visual Studio" from Microsoft, which provides a graphical tool to generate a user interface and then compile it with the actual application (written in C++, Visual Basic, or Java).

Finally, graphical editors are specialized tools for data visualization applications. Although similar to interface builders, they include custom widgets for sophisticated operations (such as simulations, process control, system monitoring, network management, and data analysis).

All interface generation tools are faced with a trade-off between giving designers control over an interface design and providing a high level of automation. Given extensive control forces designers program by hand all the details of the design. In this case, the designer must be an expert in interface design and the interface is costly to build. Automating significant portions of the interface design, on the other hand, removes the power from the designers, allowing them to control only a few details. This is preferred for applications where few resources are available for building and maintaining the interface code. Automation can generate cheap yet complete and consistent user interfaces. The goal is to achieve a balance between detailed control of the design and automation.

4.3 Toolkits and User Interface Libraries for Programming Languages

In this section we will look at some toolkits and user interface libraries for programming languages.

There are many different toolkits that render user interfaces. Some of the most popular toolkits are Microsoft Foundation Classes (or MFC), Motif, Interviews, Open View, and Smalltalk libraries.
Sun designed a toolkit—Java Foundation Classes or JFC, which provides that same look-and-feel on any platform that has a Java Virtual Machine (or JVM) implementation. JFC goes one step further in that it provides `javax.swing` to separate the look and feel from the implementation. Thus, you can create a custom look-and-feel and enforce it for all applications on all platforms.

Apple created MacApp, a software system that guides programmers in the development of user interfaces by providing an application framework. MacApp provides the classes for the most common parts, such as windows, buttons, etc., and the programmer specializes these classes to provide application-specific details. This ensured that the resulting user interface conforms to the Apples style guidelines and simplifies the writing of Macintosh applications.

Each toolkit is trying to solve a different problem: portability, easy of use, looks, more features, and so on. Tradeoffs between these problems makes it is very difficult to strike a balance and this has motivated development of multiple toolkits. The problem with too many toolkits is that programmers must support different toolkits for different platforms, thus defeating the original goal, which is portability.

One of the most popular ways to build user interfaces for applications is with a high-level language or with a visual designer, such as C++ or Visual Basic. High-level programming is powerful and provides the programmer with a lot of control over details in the design, while encapsulating the low-level assembly programming. However, it also requires significant programming experience and knowledge about the specific toolkit and usability principles. The most popular high-level languages are C/C++, Java, and Visual Basic (see Figure 4-7: UI Libraries for programming languages diagram).

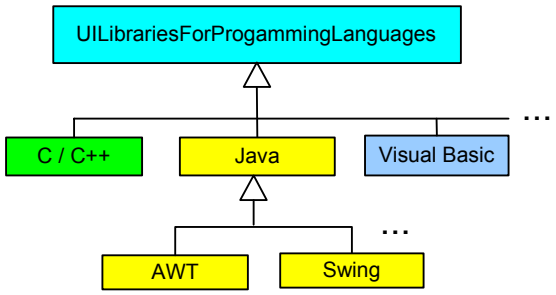


Figure 4-7: UI Libraries for programming languages diagram

The following is a description of AWT and Swing. We will also discuss their advantages and disadvantages.

AWT stands for Abstract Window Toolkit. The Abstract Window Toolkit supports Java GUI programming. It is a portable GUI library for stand-alone applications and/or applets. The AWT provides the connection between the application and the native GUI. The AWT is composed of a package of classes and it supports everything from creating buttons, menus, and dialog boxes to complete GUI applications. The AWT is platform independent, supports GUI Java programming.

Swing implements a set of GUI components and provides a pluggable look and feel. Swing is implemented entirely in the Java and AWT programming language.

AWT features include a rich set of user interface components, a robust event-handling model, graphics and imaging tools (including shape, colour, and font classes), layout managers which are for flexible window layouts that don't depend on a particular window size or screen resolution and data transfer classes which are for cut-and-paste through the native platform clipboard.

Swing features include all the features of AWT, a rich set of higher-level components (such as tree view, list box, and tabbed panes) and pluggable look and feel.

Comparison of AWT with Swing, they both have the advantages and disadvantages. The advantages of AWT are that use of native peers speeds component performance. AWT components more closely reflect the look and feel of the OS they run on. The disadvantages of AWT are that use of native peers creates platform specific limitations. Some components may not function at all on some platforms, e.g. J2SE versus J2ME. The majority of component makers, including Borland and Sun, base new component development on Swing components. There is a much smaller set of AWT components available, thus placing the burden on the programmer to create his or her own AWT-based components.

The advantages of Swing are that pure Java design provides for fewer platform specific limitations. Pure Java design allows for a greater range of behaviour for Swing components since they are not limited by the native peers that AWT uses. The pluggable look and feel lets you design a single set of GUI components that can automatically have the look and feel of any OS platform (Microsoft Windows, Solaris, Macintosh, etc.). It also makes it easier to make global changes to your Java programs that provide greater accessibility (like picking a hi-contrast colour scheme or changing all the fonts in all dialogs, etc.). However, the drawbacks of Swing are that Swing components are generally slower than AWT. Moreover, Swing applications consume too much memory, which is not suitable for small devices such as mobile phones.

Since AWT and Swing have disadvantages as mentioned earlier, many people advocate Eclipse's SWT now. SWT (The Standard Widget Toolkit) is a cross platform GUI developed by IBM. SWT solves the problems seen with the AWT and the Swing frameworks. The SWT framework accesses native widgets through JNI (Java Native Interface). If a widget is not available on the host platform, SWT emulates the unavailable widget [43].

4.4 Scripting Languages

This section briefly describes scripting languages. There are server scripting language and browser scripting language.

A scripting language is a programming language that performs tasks within a host environment. The host environment provides an interface to the user and a system of objects and facilities within which the scripting language performs its tasks. The combination of the scripting language and its host environment makes a complete programming environment.

Server scripting language, such as ASP, PHP, ADO, are executed on the server. ASP (Active Server Pages) is a program that runs inside IIS (Internet Information Services), which comes as a free component with Windows 2000. An ASP file can contain text, HTML, XML, and scripts and have the file extension "`.asp`". ADO (ActiveX Data Objects) is a Microsoft Active-X component that is automatically installed with Microsoft IIS, and is a programming interface to access data in a database by using SQL (Structured Query Language). PHP (Hypertext Preprocessor) is a server-side scripting language, like ASP. A PHP file may contain text, HTML tags and scripts.

The Velocity Template Language (VTL) [44] is scripting language. Velocity is a Java-based template engine. It permits web page designers to reference methods defined in Java code. Web designers can work in parallel with Java programmers to develop web sites according to the Model-View-Controller (MVC) model, meaning that web page designers can focus solely on creating a well-designed site, and programmers can focus solely on writing code. Velocity separates Java code from the web pages. It can be used to generate web pages, SQL, PostScript and other output from templates. It can be used either as a standalone utility for generating source code and reports, or as an integrated component of other systems.

Browser scripting language such as JavaScript and VBScript allow user to write scripting code and embed it in a HTML page.

VBScript [45] is short for Visual Basic Scripting Edition, a scripting language developed by Microsoft and supported by Microsoft's Internet Explorer Web browser. VBScript is based on the Visual Basic programming language, but is much simpler. In many ways, it is similar to JavaScript. It enables Web authors to include interactive controls, such as buttons and scrollbars, on their Web pages.

JavaScript [46] is a scripting language that is interpreted by the browser. It is included in the HTML page using the <SCRIPT> tag. JavaScript code is executed at load and unload time of a page and during or before actions that the browser user takes. The invocation of JavaScript code follows a trigger / event – action mechanism. An action can be the invocation of a JavaScript function. Functions are registered to events using `on-Conditions`.

Scripting languages are portable, simple to use, and do not require compilation. However it is difficult to reuse and extend the code that is embedded in a HTML page.

4.5 UI Technologies Supported by a Visualization Engine or Generator

After discussing so many existing UI technologies, we conclude that a visualization engine or generator is proposed to support HTML, Java, AWT, and Swing UI technologies (see Figure 4-8: UI technologies supported by a GUI engine or generator spectral diagram and Figure 4-9: UI technologies supported by a GUI engine or generator class diagram).

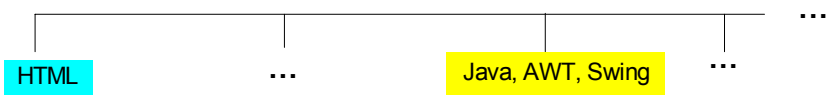


Figure 4-8: UI technologies supported by a GUI engine or generator spectral diagram

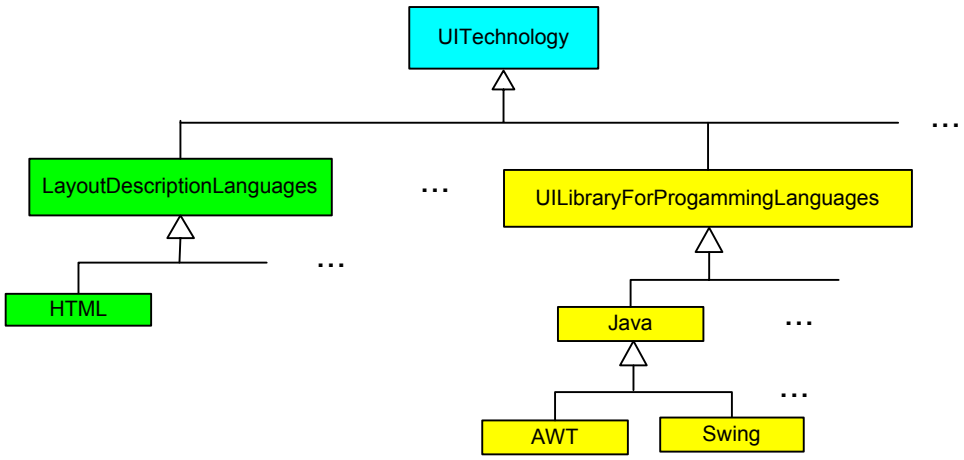


Figure 4-9: UI technologies supported by a GUI engine or generator class diagram

The advantages and disadvantages of HTML have been discussed in subsection 4.1.1 “SGML-based Layout Description Languages”. The advantages and disadvantages of the Java libraries have been already described in section 4.3 “Toolkits and User Interface Library for Programming Languages”.

HTML is simple, easy to learn, and very popular. A GUI engine or generator is proposed to support Java AWT and Swing, and then an end user can flexibly choose either AWT or Swing technology.

Here we compare a UI rendering of our new approach called conceptual content management with some existing visualization technologies such as HTML, SwiXML and UIML (see Table 4-1 Comparison of the new approach with some existing visualization technologies).

It can be seen that the existing visualization technologies such as HTML, SwiXML and UIML have their advantages and disadvantages. However, as mentioned earlier, it is not claimed that these visualization technologies cover the evolution, personalization and variants of a UI. A UI rendering of a conceptual content management system covers the evolution, personalization and variants of a UI that are three essential advantages of our new approach and have been discussed in chapter 2 “Requirements”.

Approaches / Advantages	A UI Rendering of a Conceptual Content Management	HTML	SwiXML	UIML
Portability	+	+-	+	+
Evolution	+	-	-	-
Personalization	+	-	-	-
Variants	+	-	-	-
Dynamic	+	-	-	+
Extensibility	+	-	-	+
Reusability	+	+	-	+
Usability	+	+	+	+
Ease to learn and use	+	++	+	+
Simplicity	+	++	+	+
Platform independence	+	+-	+	++

Table 4-1 Comparison of the new approach with some existing visualization technologies

5 A Visualization Engine or Generator

The requirements for a UI engine or generator are discussed in chapter 2. The visualization engine or generator works based on the UI component, the UI technology, and the domain model, and then visualizes the user interface to an end user (see Figure 2-9: The working mechanism of the UI visualization engine or generator on page 15). Selected visualization components have been designed in chapter 3 and visualization technologies have been described in chapter 4. This chapter discusses and designs the visualization engine or generator, and describes seven possibilities to implement the visualization engine or generator.

As mentioned in chapter 1, since domain models change constantly, open dynamic content management requires dynamically adaptable user interfaces. In order to realize dynamic UIs, the following discusses the implementation of the GUI engine or generator.

5.1 Analysis of Possibilities to Implement a Visualization Engine or Generator

There are several possibilities to implement a GUI engine or generator. As described in chapter 2, the GUI engine or generator needs input in terms of assets. The essence of the implementation of the GUI engine or generator is class-based approach, instance-based approach, and various combinations of class-based and instance-based approach to implement the GUI engine or generator. The following are seven selected possibilities to implement the GUI engine or generator:

(1) Class-based approach: a visualization engine or generator creates a UI component based on the type of an asset's content reference. The first approach will be discussed in subsection 5.4.1.

(2) Instance-based approach: a Java class is the value of a characteristic of an asset. A programmer defines the mapping between asset components and implementation classes. `java.lang.String` such as "menu" is the value of a characteristic called `name` of an asset. `java.lang.Class` such as `javax.swing.JMenu` is the value of a characteristic called `peerClass` of an asset. The default mapping between a component type like `swingMenu` and the class used to realize a component of that type such as `javax.swing.JMenu` is defined. The second alternative will be discussed in subsection 5.4.2.

(3) Class-based approach: an instance of a `Component` is the value of a characteristic called `peer` of an asset. When a user specifies an instance of the type such as `java.awt.Frame` for `peer`, a visualization engine or generator dynamically creates an instance for `peer`. The third alternative will be discussed in subsection 5.4.3.

(4) Class-based approach: an instance of a UI component is the value of a content called `prototype` of an asset. A programmer defines the combination between model `UIComponents` and model `UITechnologies` such as class `AWTWindow` that is a subclass of both `Window` and `AWT` and assigns an instance of a UI component to

`prototype`. The prototype pattern which means that it creates objects by cloning [20] is used in this alternative. This approach needs the multiple inheritances. When a user creates an instance of class `AWTWindow`, a visualization engine or generator dynamically creates an instance according to `prototype`. The fourth alternative will be discussed in subsection 5.4.4.

(5) In this approach technologies are instances and components are classes. An instance of a `Component` is a content called `prototype` of an asset in `model UITechnologies`. The fifth alternative will be discussed in subsection 5.4.5.

(6) Technologies are instances and components are classes. An instance of a `Component` is a content called `prototype` of an asset in `model UIComponents`. The sixth alternative will be discussed in subsection 5.4.6.

(7) Technologies are instances and components are classes. An instance of a `UIComponent` is a content called `prototype` of an asset in both model `UIComponents` and model `UITechnologies`. The seventh alternative will be discussed in subsection 5.4.7.

The following aspects will be analysed for each approach: How can a visualization engine or generator work? How can a user define a user interface? What are the advantages and disadvantages of each alternative? How complex would a visualization engine or generator be? What are the numbers of asset classes and instances of both UI components and UI technologies that have to be defined?

The example class `Person` that has been given in section 2.1 will be used here again while discussing. Classes in both the GUI domain and the technology domain have to be defined, before each alternative is discussed. Section 5.2 and section 5.3 will describe these two domains.

5.2 GUI Domain

Selected UI components have been discussed in chapter 3 (see Figure 3-1: Components class diagram on page 21). This section will define some UI components based on the asset language. The definitions are given in a simple and general way. Detailed definitions will be described in section 5.4 while discussing the different possibilities for an implementation of the GUI engine or generator.

The class `Component` has two characteristics called `visible` and `size`. The characteristic `visible` decides whether a `Component` is visible or invisible. The characteristic `size` decides the size of a `Component`. The class `Component` has two relationships called `visualizedAsset` that is an instance of `Asset` and `visualizedAssetClass` that is an instance of `AssetClass`. The connection between application domain and layout assets is done by these two relationships `visualizedAsset` and `visualizedAssetClass`. The `View` class is a subclass of `Component` as described in subsection 3.3.2. As mentioned in subsection 3.3.1, the Model-View-Controller Design Pattern is used. The `List` class and `Label` class are subclasses of `View` that correspond to the design in chapter 3.

```
class Component {
    concept characteristic visible : boolean
            characteristic size : java.awt.Dimension
    concept relationship visualizedAsset : Asset
             relationship visualizedAssetClass : AssetClass
} ; Component

class View refines Component {…} ; View
class List refines View {…} ; List
class Label refines View {…} ; Label
```

The class `Container` is a subclass of `Component` as described in section 3.1. A `Container` contains other components and is also associated with a layout manager. The classes `Panel` and `Window` are subclasses of `Container` and are described in subsection 3.1.1 and 3.1.2. The classes `FlowLayout` and `GridLayout` are subclasses of `LayoutManager and` are described in section 3.2.

```
class Container refines Component {
    concept relationship components : Component*
            relationship layout : LayoutManager
} ; Container

class Panel refines Container {…} ; Panel

class Window refines Container {
  concept relationship contentPane : container
          relationship menuBar : Menu*
} ; Window

class LayoutManager{…} ; LayoutManager
class FlowLayout refines LayoutManager {…} ; FlowLayout

class GridLayout refines LayoutManager {
    concept characteristic rows : int > 0
            characteristic cols : int > 0
} ; GridLayout
```

The class `ActiveComponent` is a subclass of `Component`. The class `Button` and `Menu` are subclasses of `ActiveComponent` that correspond to the design considerations mentioned in subsection 3.3.3.

```
class ActiveComponent refines Component {…} ; ActiveComponent

class Button refines ActiveComponent {
  concept characteristic label : Label …
} ; Button

class Menu refines ActiveComponent {
  concept relationship label : Label
          relationship menuItem: MenuItem*
} ; Menu
…
```

5.3 Technology Domain

The UI technologies have been discussed in chapter 4. This section will define some UI technologies based on the asset language. The definitions are given in a simple and general way. Detail definitions will be described in section 5.4 while discussing the different possibilities for implementation of the GUI engine or generator.

The class `UITechnology` has subclasses such as `HTML`, `Java`, `Swing` and `Awt` that correspond to the design considerations described in section 4.5.

```
class UITechnology {
      content prototype : Component
      concept relationship superType : UITechnology
      …
}; UITechnology

class HTML refines UITechnology {…}; HTML

class Java refines UITechnology {…};Java

class Swing refines UITechnology {
      prototype : javax.swing.JComponent
      superType : Java
      …
}; Swing

class Awt refines UITechnology {
      prototype : java.awt.Component
      superType : Java
      …
}; Awt
...
```

5.4 Different Possibilities for Implementation of a GUI Engine or Generator

Some classes of both the GUI domain and the technology domain have been defined in section 5.2 and 5.3. In this section seven selected possibilities to develop a GUI engine or generator will be described and their advantages and disadvantages will be discussed.

5.4.1 Creating a UI Component Based on the Type of an Asset's Content Reference

This subsection discusses the first approach—a UI engine or generator creates a UI component based on the type of an asset's content reference as mentioned in section 5.1.

The class `Component` has four attributes that are called `peer`, `visualizedAsset`, `visualizedAssetClass` and `visualizedAttribute`. The content called `peer` of an asset is a type of `java.awt.Component`. The relationship `visualizedAsset` is a type of `Asset`. The relationship `visualizedAssetClass` is a type of `AssetClass`. The connection between application domain and layout assets is done by two relationships `visualizedAsset` and

visualizedAssetClass. The characteristic visualizedAttribute is a type of Attribute that is associated with the application domain when a user defines a UI.

```
class Component {
  content peer : java.awt.Component
  concept relationship visualizedAsset : Asset
          relationship visualizedAssetClass : AssetClass
          characteristic visualizedAttribute : Attribute
} ; Component
```

The class Container is a subclass of Component as already mentioned in section 5.2.

```
class Container refines Component {
   concept relationship components : Component*
           relationship layout : LayoutManager
} ; Container
```

The class Window is a subclass of Container that corresponds to the design considerations described in subsection 3.1.1. The value of the attribute peer of the class Window is an instance of java.awt.Window. The attribute size defines the size of a window.

```
class Window refines Container {
  content peer : java.awt.Window
  concept characteristic size : java.awt.Dimension
          constraint peer.getSize ().equals (size)
          onviolation peer.setSize (size)
} ; Window
```

The class AWTWindow is a subclass of Window. The value of the attribute peer of the class AWTWindow is an instance of java.awt.Frame. The attribute size defines the size of an AWTWindow. The class SwingWindow is also a subclass of Window. The value of the attribute peer of the class SwingWindow is an insatnce of javax.swing.JFrame. The attribute peer has different types in different classes Component, Window, AWTWindow and SwingWindow. The GUI engine or generator creates an instance of a given class for peer.

```
class AWTWindow refines Window {
  content peer : java.awt.Frame
  concept characteristic size : java.awt.Dimension
} ; AWTWindow

class SwingWindow refines Window {
  content peer :javax.swing.JFrame
} ; SwingWindow
…
```

A user can define a user interface in a very simple way as follows:

```
let fenfang : Person := create Person {}

let myPersonWindow : Window := create AWTWindow {
    visualizedAsset := fenfang
    visualizedAssetClass := Person
    components := {
      create Label { text = "Name:" }
      create TextField { visualizedAttribute := Person.name
             text := fenfang.name}
```

```
      create Label { text := "Street:" }
      …
    }
    layoutManager := create GridLayout { width :=2 height :=3 }
 …
}
```

A user first creates an instance of class `Person` called `fenfang`, then creates an instance of class `AWTWindow` named `myPersonWindow` and gives `fenfang` as the value of the relationship `visualizedAsset`, `Person` as the value of the relationship `visualizedAssetClass`. As mentioned earlier, two relationships `visualizedAsset` and `visualizedAssetClass` connect application domain and layout assets. Finally, a UI engine or generator generates an `AWTWindow` to show the information about person `fenfang`.

A user can also select `SwingWindow` as the class to create an instance of `SwingWindow` named `myPersonWindow` as follows:

```
let myPersonWindow : SwingWindow := create SwingWindow {
    visualizedAsset := fenfang
    visualizedAssetClass := Person
    components := {
      create Label { text := "Name:" }
      create TextField {
               visualizedAttribute := Person.name
               text := fenfang.name
             }
      create Label { text := "Street:" }
      …
    }
    layoutManager := create GridLayout { width := 2 height := 3 }
 …
}
```

The number of classes that have to be defined is:
(the number of components) * [(the number of technologies) * (the number of components)]
A programmer has to define one class for each component and one class for each combination of the technology and the component.

It can be seen that the advantages of this approach are that it is simple and easy to learn for a user. The drawback is that the class definition is not portable because the type of `peer` is given and fixed. This leads to difficulty of reusing the code.

5.4.2 A Java Class is the Value of a Characteristic of an Asset

The following is a description of the second alternative— a Java class is the value of a characteristic of an asset as mentioned in section 5.1. A programmer defines the default mapping between a component type like `swingMenu` and the class used to realize a component of that type such as `javax.swing.JMenu`.

The class `Component` has four attributes `visualizedAsset`, `visualizedAssetClass`, `name` and `peerClass`. The value of the attribute `visualizedAsset` is an instance of `Asset` and the value of the attribute `visualizedAssetClass` is an instance of `AssetClass`. These two

attributes create the connection between application domain and layout assets. The value of the attribute `name` is an instance of `java.lang.String`. The value of the attribute `peerClass` is an instance of `java.lang.Class`. A programmer defines the default mapping between a component type `name` and the class used to realize a component of the type `peerClass` as follows:

```
class Component {
  concept relationship visualizedAsset : Asset
          relationship visualizedAssetClass : AssetClass
  concept characteristic name : java.lang.String
          characteristic peerClass : java.lang.Class
} ; Component

let awtWindow := create Component {
    name := "window"
    peerClass := java.awt.Window.class
    ; create an instance called awtWindow of class java.awt.Window.class
}

let swingWindow := create Component {
    name := "window"
    peerClass := javax.swing.JWindow.class
}

let swingMenu := create Component {
    name := "Menu"
    peerClass := javax.swing.JMenu.class
}

let awtMenu := create Component {
    name := "Menu"
    peerClass := java.awt.Menu.class
}

let awtMenuBar := create Component {
    name := "Menu-bar"
    peerClass := java.awt.MenuBar.class
}

let swingMenuBar := create Component {
    name := "Menu-bar"
    peerClass := javax.swing.JMenuBar.class
}

let swingMenuItem := create Component {
    name := "Menu-item"
    peerClass := javax.swing.JMenuItem.class
}

let swingMenuItemSeperator := create Component {
    name := "Menu-item-seperator"
    peerClass := javax.swing.JSeperator.class
}
…
```

Each component such as `swingMenu` and `awtMenu` must have attributes called `name` and `peerClass`. The name must be unique within the mapping definitions. The notion of

peerClass specifies an object type; the component's name uniquely identifies an instance of that type.

A visualization engine or generator can create objects from resource descriptions, for example, the realized form of a frame element is an instance of javax.swing.JFrame, and that of a label element is an instance of javax.swing.JLabel.

A user can define a user interface as follows:

```
class VisualizedPerson refines Person {
  concept relationship visualizedBy : Component
} ; a subclass called VisualizedPerson of class Person

let fenfang := create VisualizedPerson {
  visualizedBy := awtWindow
}

let myPersonWindow := create Component awtWindow
  modify myPersonWindow {
    visualizedAsset := fenfang
    visualizedAssetClass := Person
}
```

A user first creates a subclass called VisualizedPerson of class Person, and then creates an instance of class VisualizedPerson called fenfang. The user creates an instance awtWindow of class Component named myPersonWindow that is a prototype, and then the user modifies the prototype according to the value of the attribute visualizedAsset called fenfang and the value of the attribute visualizedAssetClass called Person. As mentioned earlier, two relationships visualizedAsset and visualizedAssetClass connect application domain and layout assets.

The number of classes and asset instances that are defined in a mapping file is:
(1 class for Component) + (the number of technologies) * (the number of components)
A programmer has to define one class for Component and create one instance for each combination of technology and component.

The advantages of this approach are that it allows the language to be extensible. It is easy to override this default mapping and substitute a different class, perhaps one that is user-defined, to realize a component.

5.4.3 An Instance of a Component is the Value of a Characteristic of an Asset

The third alternative—an instance of a Component is the value of a characteristic of an asset as mentioned in section 5.1. Definitions of both the model Components and the model Technologies are given as follows:

```
model Components
class ComponentType {
  concept relationship superType : ComponentType
}; ComponentType
```

The class `ComponentType` has an attribute `superType` whose value is an instance of `ComponentType`. The class `Component` is a subclass of `ComponentType`. It has five attributes `visualizedAsset, visualizedAssetClass, type, technology` and `peer`. The attributes `visualizedAsset` and `visualizedAssetClass` create the connection between application domain and layout assets. The value of the attribute `type` is an instance of `ComponentType`. The value of the attribute `technology` is an instance of `UITechnology` and the value of the attribute `peer` is an instance of `Component`.

```
class Component refines ComponentType {
  concept relationship visualizedAsset : Asset
          relationship visualizedAssetClass : AssetClass
          relationship type : ComponentType
          relationship technology : UITechnology
          characteristic peer : Component
}; Component

let component := create ComponentType {}
let window := create ComponentType { superType := component }
…

model Technologies

class UITechnology { concept relationship superType : UITechnology }
let java := create UITechnology {}
let awt := create UITechnology { superType := java }
…
```

Then a user can use this definition to create an `awt` window as follows:

```
let fenfang : Person := create Person {}
create Component {
       peer := new java.awt.Frame ()
       type := window
       technology := awt
       visualizedAsset := fenfang
       visualizedAssetClass := Person
}
```

A user can also create a `Swing` window by changing the values of attributes `peer, type` and `technology` as follows:

```
create Component {
       peer := new javax.swing.JFrame ()
       type := window
       technology := swing
       visualizedAsset := fenfang
       visualizedAssetClass := Person
}
```

The number of classes and instances that have to be defined is:
(one class for `ComponentType`) + (one class for `Component`) + (one class for `UITechnology`) + (The number of components) + (The number of technologies)
A programmer has to define one class for `ComponentType, Component, UITechnology` respectively, create one instance for each component and create one instance for each technology.

The advantages of this alternative are that it is based on instances of model `Components` and model `Technologies`. It is dynamic. A visualization engine or generator searches the prototype of the instance according to the given value of the attributes `type` and `technology`. The disadvantages of this approach are that it is complex to implement the GUI engine or generator. It will be a problem if a user first creates a `Component` that has the attribute peer whose value is an instance of `java.awt.Frame`, and then the user modifies the `Component` that has the attribute `peer` whose value is an instance of `javax.swing.JFrame`. The following code describes this scenario:

```
let myFrame := create Component {
    peer := new java.awt.Frame ()
    type := window
    technology := awt
    visualizedAsset := fenfang
    visualizedAssetClass := Person
}

modify myFrame {peer := new javax.swing.JFrame ()}

modify myFrame {peer := new javax.swing.XFrame ()}
```

The worst-case scenario is that the user modifies the `Component` that has the attribute `peer` whose value is an instance of `javax.swing.`***`XFrame`***. It is a run time error because `javax.swing.`***`XFrame`*** does not exist in the technology model.

5.4.4 An Instance of a UI Component is the Value of a Content of an Asset

In this subsection the fourth alternative will be analysed— an instance of a UI component is the value of a content of an asset as mentioned in section 5.1.

The class `Component` has two attributes `visualizedAsset` and `visualizedAssetClass`. The attributes `visualizedAsset` and `visualizedAssetClass` create the connection between application domain and layout assets. The class `Container` and the class `Window` are subclasses of `Component`.

```
model UIComponents:

class Component {
  concept relationship visualizedAsset : Asset
          relationship visualizedAssetClass : AssetClass
}; Component

class Container refines Component {
  concept relationship components : Component*
          relationship layout : LayoutManager
}; Container

class Window refines Container{…}; Window
…
```

The class `UITechnology` has a content called `prototype` of an asset, whose value is an instance of `java.awt.Component`. The class `Java` is a subclass of `UITechnology` and the class `AWT` is a subclass of `Java`.

```
model UITechnologies:

class UITechnology {
      content prototype : java.awt.Component
}; UITechnology

class Java refines UITechnology {…}; Java

class AWT refines Java {…}; AWT
class Swing refines Java {
      content prototype : javax.swing.JComponent
      …
}; Swing

...
```

The class `AWTWindow` is a subclass of both `Window` and `AWT`. It has an attribute `prototype`, which is an instance of `java.awt.Frame`. The class `SwingWindow` is a subclass of both `Window` and `Swing`. It has an attribute, which is an instance of `javax.swing.JFrame`. A UI engine or generator creates an instance based on `prototype` according to prototype pattern as mentioned in section 5.1.

```
class AWTWindow refines Window, AWT {
  content prototype : java.awt.Frame := new java.awt.Frame ()
}; AWTWindow

class SwingWindow refines Window, Swing {
  content prototype : javax.swing.JFrame := new javax.swing.JFrame ()
}; SwingWindow
…
```

A user can define a user interface as follows:

```
let fenfang : Person := create Person {}

;create myPersonWindow as an instance of AWTWindow
let myPersonWindow : Window := create AWTWindow {
    visualizedAsset := fenfang
    visualizedAssetClass := Person
    …
}

; Or create myPersonWindow as an instance of SwingWindow
let myPersonWindow : Window := create SwingWindow {
    visualizedAsset := fenfang
    visualizedAssetClass := Person
    …
}
```

A user first creates an instance of class `Person` called `fenfang`, then creates an instance of class `AWTWindow` or `SwingWindow` named `myPersonWindow` and gives `fenfang` as the value of the relationship `visualizedAsset` and `Person` as the value of the relationship

visualizedAssetClass. As mentioned earlier, two relationships visualizedAsset and visualizedAssetClass create the connection between application domain and layout assets.

Then the UI engine or generator creates an instance of AWTWindow called myPersonWindow based on the value of prototype given in class AWTWindow. As mentioned earlier, this approach uses the prototype pattern which creates instances by cloning. The UI engine or generator can also clone an instance of SwingWindow called myPersonWindow according to the value of prototype given in class SwingWindow.

The number of classes, which have to be defined, is:
(the number of components) + (the number of technologies) + (the number of components) * (the number of technologies)
A programmer has to define one class for each Component, each UITechnology and each combination between Component and UITechnology.

The advantage of this alternative is that it is portable. It is simple and easy to use. The drawback is that this approach has to support the multiple inheritances such as **class** AWTWindow **refines** Window, AWT {}. The requirement of the multiple inheritance leads to complex implementation of a UI engine or generator.

5.4.5 A Combination of Technologies as Instances and Components as Classes

The following describes the fifth alternative— a combination of technologies represented by instances and components represented by classes as mentioned in section 5.1.

In this approach technologies are instances and components are classes. The class UIComponent has an attribute technology, whose value is an instance of UITechnology. The class Component is a subclass of UIComponent. It has two attributes visualizedAsset and visualizedAssetClass that create the connection between application domain and layout assets.

```
model UIComponents

class UIComponent {concept relationship technology : UITechnology*}

class Component refines UIComponent {
  concept relationship visualizedAsset : Asset
          relationship visualizedAssetClass : AssetClass
}; Component

class Container refines Component {
   concept relationship components : Component*
           relationship layout : LayoutManager
}; Container

class Window refines Container{…}; Window
…

model UITechnologies

class UITechnology {
  content prototype : Component
```

```
  concept relationship superType : UITechnology
}

let java := create UITechnology {…}

let awt := create UITechnology { superType := java }

let swing := create UITechnology { superType := java }
...
```

The class `UITechnolgy` has two attributes `prototype` whose value is an instance of `Component` and `superType` whose value is an instance of `UITechnology`. The technologies such as `java`, `awt` and `swing` are instances of `UITechnology`. In this approach the instance `awtWindow` is created as follows: First, an instance of `Window` is created. Second, the attribute `technology` of `Window` is assigned the value `awt`, which is an instance of `UITechnology`. Finally, the value of the attribute `prototype` of `awt` is modified from an instance `Component` to an instance of `java.awt.Frame`. The instance `swingWindow` can be created in the same way.

```
let awtWindow := create Window {
    technology := modify create UITechnology awt {
      prototype := new java.awt.Frame()
    }
}

let swingWindow := create Window {
    technology :=  modify create UITechnology swing {
      prototype := new javax.swing.JFrame()
    }
}
…
```

A user can define a user interface as follows:

```
class VisualizedPerson refines Person {
      concept relationship visualizedComponent : UIComponent
}; Define subclass VisualizedPerson of super class Person

let fenfang : Person := create VisualizedPerson {
    visualizedComponent := awtWindow
}; select awtWindow as visualizedComponent

let fenfang : Person := create VisualizedPerson {
    visualizedComponent := swingWindow
}; Or select swingWindow as visualizedComponent

let myPersonWindow : Window := create UIComponent {
    visualizedAsset := fenfang
    visualizedAssetClass := Person
}
```

First, a user creates a subclass `VisualizedPerson` of class `Person`. It has an attribute `visualizedComponent` whose value is an instance of `UIComponent`. Second, the user creates an instance of class `VisualizedPerson` called `fenfang` whose attribute `visualizedComponent` has the value `awtWindow` or `swingWindow`. Third, the user creates an instance `myPersonWindow` of class `Window` and gives `fenfang` as the value of the attribute

visualizedAsset and Person as the value of the attribute visualizedAssetClass. As mentioned earlier, two relationships visualizedAsset and visualizedAssetClass connect application domain and layout assets.

Then a GUI engine or generator generates an awtWindow according to the value of the attribute visualizedComponent in the object fenfang. In the same way a UI engine or generator generates a swingWindow according to the value of the attribute visualizedComponent in the object fenfang.

The number of classes and instances, which have to be defined, is:
(the number of components) + (one class for UITechnology) + (the number of technologies) + (the number of components) * (the number of technologies)
It means that a domain designer has to define one class for each component, one class for the class UITechnology, create one instance for each technology and each combination between components and technologies.

The advantage of this alternative is that the value of the attribute prototype of UITechnology is given directly such as prototype: = new javax.swing.JFrame() by a programmer. The disadvantages of this approach are that it is complex, for example "technology: = modify **create** UITechnology awt {prototype: = **new** java.awt.Frame()}". It is not easy to use for a user. The number of classes and instances that have to be defined is quite larger compared to the other possibilities.

5.4.6 A Different Combination of Technologies as Instances and Components as Classes

The sixth alternative— a different combination of technologies represented by instances and components represented by classes as mentioned in section 5.1.

Technologies are instances and components are classes in this approach. The class UIComponent has an attribute technology whose value is an instance of UITechnology. The class Component is a subclass of the class UIComponent. It has three attributes prototype, visualizedAsset and visualizedAssetClass. The value of the attribute prototype is an instance of Component. The attributes visualizedAsset and visualizedAssetClass create the connection between application domain and layout assets.

```
model UIComponents

class UIComponent {concept relationship technology : UITechnology*}

class Component refines UIComponent {
      content prototype : Component
      concept relationship visualizedAsset : Asset
              relationship visualizedAssetClass : AssetClass
}; Component

class Container refines Component {
      concept relationship components : Component*
              relationship layout : LayoutManager
}; Container
```

```
class Window refines Container{…}; Window
…
```

The class `UITechnology` has an attribute `superType` whose value is an instance of `UITechnology`. The technologies such as `java`, `awt` and `swing` are instances of the class `UITechnolgoy`. The instance `awtWindow` is created as follows: First, an instance of the class `Window` is created. Second, an instance of `java.awt.Frame` is assigned as the value of the attribute `prototype` of the class `Window`. Third, the attribute `technology` of the class `Window` is assigned the value `awt`, which is an instance of `UITechnology`. The instance `swingWindow` of the class `Window` can be created in the same way.

```
model UITechnologies

class UITechnology {
      concept relationship superType : UITechnology
}

let java := create UITechnology {}

let awt := create UITechnology { superType := java }

let swing := create UITechnology { superType := java }
...

let awtWindow := create Window {
    prototype := new java.awt.Frame()
    technology :=  awt
}
let swingWindow := create Window {
    prototype := new javax.swing.JFrame()
    technology :=  swing
}
…
```

A user can define a user interface as follows:

```
class VisualizedPerson refines Person {
      concept relationship visualizedComponent : UIComponent
}; define subclass VisualizedPerson of super class Person

let fenfang : Person := create VisualizedPerson {
    visualizedComponent := awtWindow
}; select awtWindow as visualizedComponent

let fenfang : Person := create VisualizedPerson {
    visualizedComponent := swingWindow
}; Or select swingWindow as visualizedComponent

let myPersonWindow : Window := create UIComponent {
      visualizedAsset := fenfang
      visualizedAssetClass := Person
      components := {
        create Label { text := "Name:" }
        create TextField {
            visualizedAttribute := Person.name
            text:=fenfang.name
        }
        create Label { text := "Street:" }
```

```
        …
      }
      layoutManager := create GridLayout { width := 2 height := 3 }
      …
}
```

First, a user creates a subclass `VisualizedPerson` of class `Person`. It has an attribute `visualizedComponent` whose value is an instance of `UIComponent`. Second, the user creates an instance of class `VisualizedPerson` called `fenfang` whose attribute `visualizedComponent` has the value `awtWindow` or `swingWindow`. Third, the user creates an instance `myPersonWindow` of class `Window`, gives `fenfang` as the value of the attribute `visualizedAsset` and `Person` as the value of the attribute `visualizedAssetClass`, and creates components `Label` and `TextField` to show the name and street information of the person `fenfang`. As mentioned earlier, two relationships `visualizedAsset` and `visualizedAssetClass` connect application domain and layout assets.

Then a GUI engine or generator generates an instance `awtWindow` called `myPersonWindow` of class `Window` according to the value of the attribute `visualizedComponent` in the object `fenfang`. In the same way a UI engine or generator can also generates an instance `swingWindow myPersonWindow` of class `Window` according to the value of the attribute `visualizedComponent` in the object `fenfang`.

The number of classes and instances that have to be defined is:
(the number of components) + (one class for `UITechnology`) + (the number of technologies) + (the number of components) * (the number of technologies)
It means that a domain designer has to define one class for each component, one class for the class `UITechnology`, create one instance for each technology and each combination between components and technologies.

The advantages of this approach are that the value of the attribute `prototype` of the class `UITechnology` is given directly such as `prototype := new javax.swing.JFrame()` by the domain designer. It is simpler compared to the fifth solution by moving the attribute `prototype` from the class `UITechnology` to the class `UIComponent`. The disadvantages of this approach are that the number of classes and instances, which have to be defined, is quite huge. It is not easy to use for a user.

5.4.7 Another Combination of Technologies as Instances and Components as Classes

The following discusses the last approach— another combination of technologies represented by instances and components represented by classes as mentioned in section 5.1.

Technologies are instances and components are classes. The class `UIComponent` has three attributes `prototype`, `visualizedAsset` and `visualizedAssetClass`. The value of the attribute `prototype` is an instance of `UIComponent`. The attributes `visualizedAsset` and `visualizedAssetClass` create the connection between application domain and layout assets. The class `Component` is a subclass of the class `UIComponent`. The class `Container` is a subclass of the class `Component` and the class `Window` is a subclass of `Container` that

corresponds to the design consideration described in section 3.1. The `awtWindow` is an instance of `Window` whose attribute `prototype` is an instance of `java.awt.Frame`. The instances such as `swingWindow` and `htmlWindow` can be created by the same way.

```
model UIComponents

class UIComponent {
      content prototype : UIComponent
      concept relationship visualizedAsset : Asset
              relationship visualizedAssetClass : AssetClass
}
class Component refines UIComponent {…}; Component

class Container refines Component {
      concept relationship components : Component*
              relationship layout : LayoutManager
}; Container

class Window refines Container{…}; Window
…
let awtWindow := create Window { prototype := new java.awt.Frame () }

let swingWindow := create Window { prototype := new javax.swing.JFrame () }

let htmlWindow := create Window { prototype :=
"<script>window.open(...)</script>" }
...
```

The class `UITechnolgy` has an attribute `prototype` whose value is an instance of `UIComponent`. The class `Java` is a subclass of the class `UITechnolgy` and the class `AWR` is a subclass of `Java`. The `awtWindow` is an instance of `AWT` whose attribute `prototype` is an instance of `java.awt.Frame`. The instances such as `awtButton` and `awtTextField` can be created by the same way.

```
model UITechnologies

class UITechnology {content prototype : UIComponent }
class Java refines UITechnology { ... }
class AWT refines Java { ... }
...

let awtWindow := create AWT { prototype := new java.awt.Frame () }
let awtButton := create AWT { prototype := new java.awt.Button () }
let awtTextField := create AWT { prototype := new java.awt.TextField ()}
...
```

A user can define a user interface as follows:

```
class VisualizedPerson refines Person {
      concept relationship visualizedComponentClass : UIComponent
              relationship visualizedTechnologyClass : UITechnology
}; define subclass VisualizedPerson of class Person

let fenfang : Person := create VisualizedPerson {
    visualizedComponentClass := Window
    visualizedTechnologyClass := AWT
}; select Window as visualizedComponentClass and AWT as
visualizedTechnologyClass
```

```
let fenfang : Person := create VisualizedPerson {
    visualizedComponentClass := Window
    visualizedTechnologyClass := Swing
}; Or select Window as visualizedComponent and Swing as
visualizedTechnologyClass

let myPersonWindow : Window := create Window {
    visualizedAsset := fenfang
    visualizedAssetClass := Person
}
```

First, a user creates a subclass `VisualizedPerson` of class `Person`. It has an attribute `visualizedComponentClass` whose value is an instance of `UIComponent` and an attribute `visualizedTechnologyClass` whose value is an instance of `UITechnology`. Second, the user creates an instance of the class `VisualizedPerson` called `fenfang` whose attributes `visualizedComponentClass` and `visualizedTechnologyClass` have the value `Window` and `AWT` respectively. Third, the user creates an instance `myPersonWindow` of class `Window`, gives `fenfang` as the value of the attribute `visualizedAsset` and `Person` as the value of the attribute `visualizedAssetClass`.

Then the GUI engine or generator starts to search for the parameter with the value `Window` and the parameter with the value `AWT`, and then find the intersection of class extensions, here the intersection of class extensions is `awtWindow`. Finally, the GUI engine or generator clones an instance of `java.awt.Frame` as the value of the attribute `prototype`.

The number of classes and instances that have to be defined is:
(the number of components) + [(the number of components) * (the number of technologies)] + (the number of technologies) + [(the number of components) * (the number of technologies)]

A programmer has to define one class for each component and create one instance for each combination between components and technologies in the `UIComponents` model. In the same way, one class for each technology has to be defined and one instance for each combination between components and technologies has to be defined in the `UITechonologies` model.

The advantage of this alternative is that it is dynamic, portable, reusable, and extensible. The disadvantages of this approach are that it is complex because a UI engine or generator has to search for the parameter with the value of the attribute `visualizedComponentClass` and the attribute `visualizedTechnologyClass`, and then decide the type of the attribute `prototype` and clone it. The number of classes and instances that have to be defined is the largest compared to the other possibilities.

5.5 Comparison of Different Possibilities to Implement a GUI Engine or Generator

Based on the above detailed analyses of seven different alternatives to implement a GUI engine or generator, now their advantages and disadvantages will be further compared from the following different aspects: portability, personality, dynamic, etc. for details see the following table (Table 5-1: Analyses the alternatives for implementation of a GUI engine or generator). The advantages and disadvantages of each approach will be considered, and then

the best solution will be found. The following recommends one solution to implement a GUI engine or generator.

What we have done in this project is to define a model of UI components, UI technologies by assets, and analyse design considerations for a visualization engine or generator. Now come to the point to decide which alternative is the best to implement the GUI engine or generator. Before make a decision, let us look at table 5-1 in detail.

The first approach as mentioned in subsection 5.4.1 is very simple and easy to learn, but the drawback is that the class definition is not portable and not dynamic. We think that portable and dynamic properties are very important for a GUI engine or generator, so this is not a good solution.

The second alternative as mentioned in subsection 5.4.2 is very extensible, also simple and easy to learn, but not dynamic, so it is not the optimal way to implement a GUI engine or generator.

The third alternative as mentioned in subsection 5.4.3 is more complex than the other alternatives such as the solution (1), (2) and (5). As mentioned in subsection 5.4.3, it will be a problem if a user first creates a `Component` that has the attribute peer whose value is an instance of `java.awt.Frame`, and then the user modifies the `Component` that has the attribute `peer` whose value is an instance of `javax.swing.JFrame`. Simplicity is a very important characteristic of a GUI engine or generator, so we do not think this is a good solution.

The fourth alternative as mentioned in subsection 5.4.4 needs the multiple inheritance such as **`class`** `AWTWindow` **`refines`** `Window, AWT {}`. The requirement of the multiple inheritance leads to complex implementation for a UI engine or generator, so this is not a good solution.

Now let us look at three different combinations of technology represented by instances and component represented by classes. The seventh alternative as mentioned in subsection 5.4.7 requires the largest number of classes and instances that have to be defined. The fifth alternative as mentioned in subsection 5.4.5 and the seventh alternative as mentioned in subsection 5.4.7 are much more complex than the sixth alternative as mentioned in subsection 5.4.6. This matters learning difficult.

Finally, we conclude that the sixth alternative as mentioned in subsection 5.4.6 is the best solution to implement a GUI engine or generator because it is more dynamic, extensible, portable, has lower engine complexity and the lower number of defined classes and /or instances compared to the other solutions.

Solutions / Advantages	(1)	(2)	(3)	(4)	(5)	(6)	(7)
Portability	-	+	+	+	+	+	+
Evolution	+	+	+	+	+	+	+
Personalization	+	+	+	+	+	+	+
Adaptability	+	+	+	+	+	+	+
Dynamic	-	-	+++	+	+++	+++	+++
Extensibility	+	+++	++	+++	+++	+++	+++
Reusability	-	+	+	+	+	+	+
Usability	+	+	+	+	+	+	+
Ease to learn and use	+++	++	+	+	-	+	-
Simplicity	+++	++	+	+	-	+	-
Platform independence	+	+	+	+	+	+	+
Engine complexity	+	+	++	++	+++	+	++

Table 5-1: Analyses the alternatives for implementation of a GUI engine or generator

6 Summary and Outlook

This chapter will conclude with a short summary and a look at the further development of a UI visualization engine or generator for UIs.

6.1 Summary

Openness and dynamics as introduced in chapter 1 allow conceptual content management systems to be constantly adapted, refined and personalized according to the requirements as demanded by its users' tasks. Since domain models change constantly, open dynamic conceptual content management requires dynamically adaptable user interfaces. However, UI technologies are not open and dynamic as described in chapter 4.

Open dynamic conceptual content management that is based on a new language called asset language is an innovative way to implement information systems. Like the application domain model the presentation of assets has to be user-definable because a UI cannot be automatically constructed. The user interface of our approach is implemented by describing UIs through the ADL (Asset Definition Language) by using assets to model the UI realm. The advantages of this approach are that the ADL allows three essential contributions: the evolution, personalization, and adaptability of a user interface. A special UI visualization engine or generator must be designed in order to realize open dynamic visualization. The visualization is realized by a combination of the application domain and the UI realm that consists of two domains: one for logical UI components and one for presentation technologies. These two domains are orthogonal. A UI engine or generator as presented in this report works based on a UI components model, a UI technologies model, and an application domain model.

Consequently, this project study has defined models for UI components (chapter 3) and UI technologies (chapter 4) logically as well as the implementation by assets (section 5.2 and 5.3). Design considerations for a visualization engine or generator which realizes dynamic visualization are discussed (chapter 2 and chapter 5). There are several different approaches to design the input format of a GUI engine or generator. The advantages and disadvantages of seven selected possibilities have been analysed (section 5.4). Finally, a combination of technology represented by instances and component represented by classes was found that is an expected solution to implement a GUI engine or generator. This is because it is more dynamic, extensible and portable. Moreover, the engine complexity and the number of defined classes and instances are lower compared to the other solutions (section 5.5).

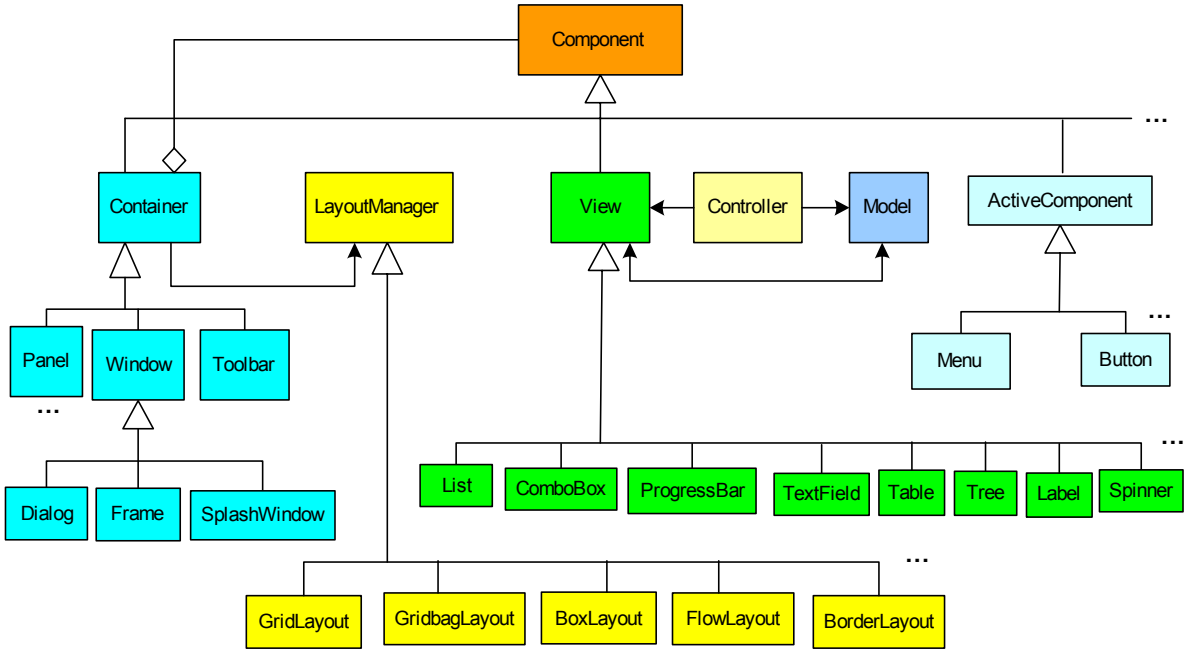
6.2 Outlook

As described in section 5.4, there are several alternatives to develop a GUI engine or generator. According to the recommendation in section 5.5, a combination of technology represented by instances and component represented by classes is an expected solution to implement a GUI engine or generator. For the next phase, the following is a description of what should be done in order to implement the GUI engine or generator that is based on assets

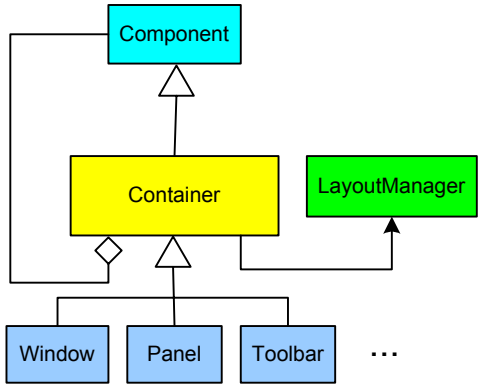
technology and realizes dynamic visualization. First, classes of all components in the UI components domain as described in chapter 3 have to be defined. Second, instances of all technologies in the UI technologies domain as described in chapter 4 have to be defined. Third, all instances that relate the UI components domain with the UI technologies domain have to be defined. Finally, a UI engine or generator has to be designed and implemented according to the chosen domain models.

A UI engine or generator has to be applied so that end users can use it in order to verify the suitability of the chosen approach, which includes ease of learning, user acceptance and maintainability, etc. The performance of a UI engine or generated code must be checked. It is also necessary to validate the methodology for refining UIs.

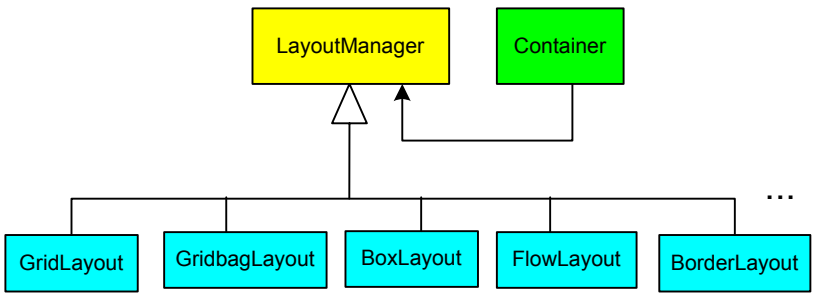
Appendix A: Visualization Components Class Diagrams



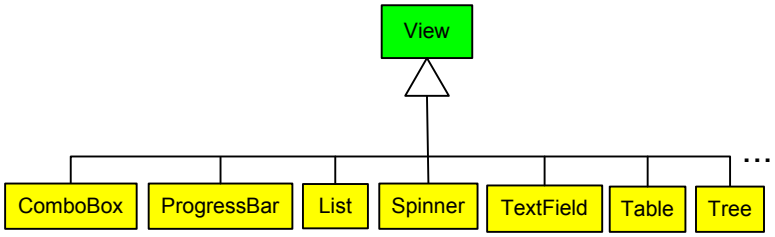
Appendix A 1 Component class diagram



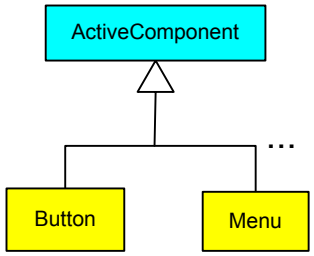
Appendix A 2 Container class diagram



Appendix A 3 LayoutManager class diagram

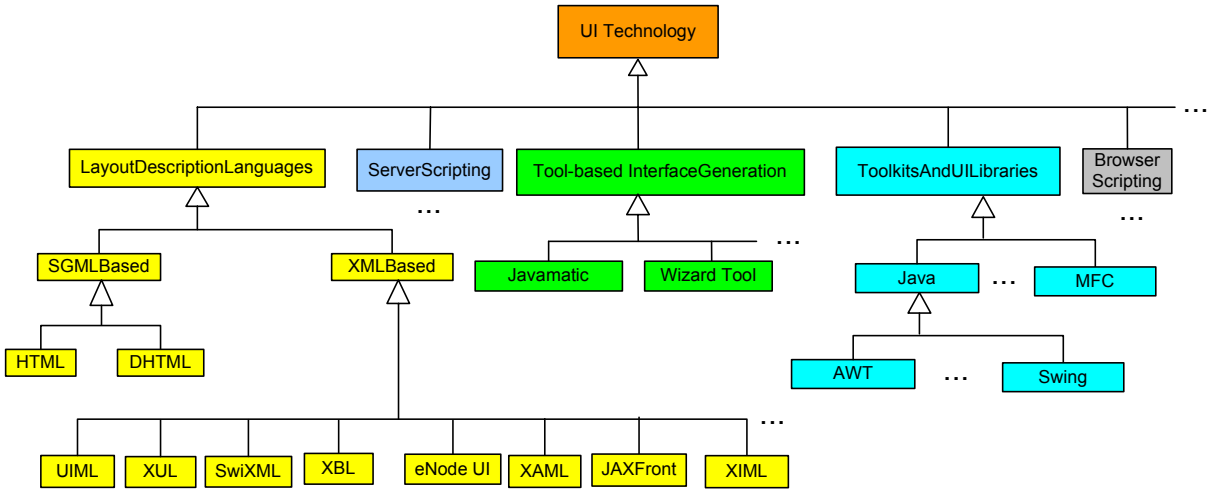


Appendix A 4 View class diagram

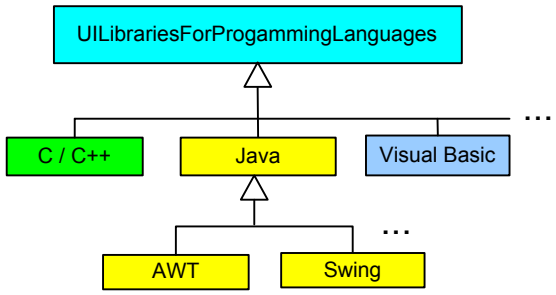


Appendix A 5 ActiveComponent class diagram

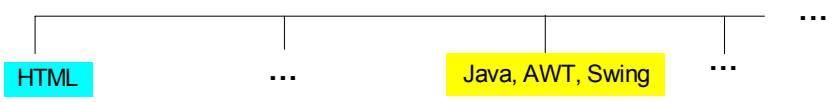
Appendix B: Visualization Technologies Diagrams



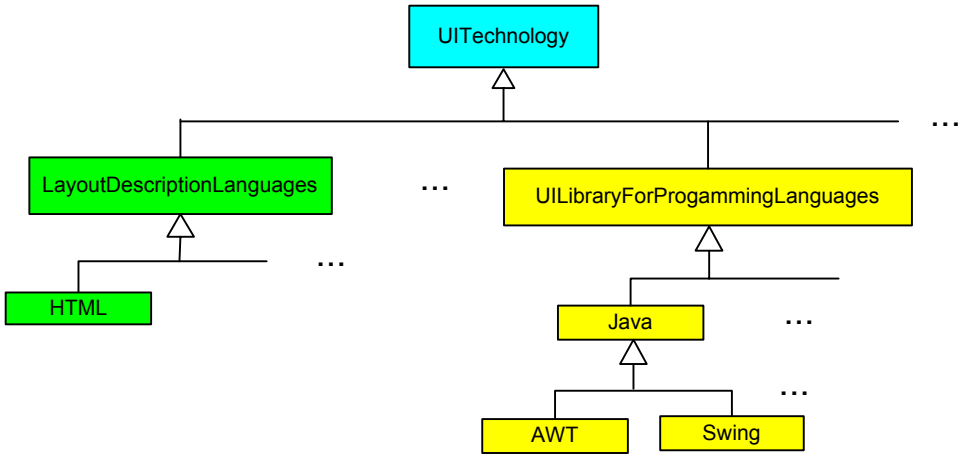
Appendix B 1 Existing visualization technologies diagram



Appendix B 2 UI libraries for programming languages diagram



Appendix B 3 UI technologies supported by a GUI engine or generator spectral diagram



Appendix B 4 UI technologies supported by a GUI engine or generator class diagram

Appendix C: Glossary

Abstraction	A description of something that omits some details that are not relevant to the purpose of the abstraction. It is the converse of refinement. Abstraction in programming is the process of identifying common patterns that have systematic variations; an abstraction represents the common pattern and provides a means for specifying which variation to use.
Application	A program that combines all the functions necessary for a user to accomplish a particular set of tasks
Active component	Components that a user can manipulate to perform an action, choose an option, or set a value. Examples include buttons, sliders, list, and combo boxes.
Available	Able to be interacted with. When a component is unavailable, it is dimmed and is unable to receive keyboard focus.
Behaviour	Refers to how applications interact with users.
Checkbox	A control, consisting of a graphic and associated text, which a user clicks to turn an option on or off. A check mark in the checkbox graphic indicates that the option is turned on.
Combo box	A component with a drop-down arrow that the user clicks to display a list of options. Noneditable combo boxes have a list from which the user can choose one item. Editable combo boxes offer a text field as well as a list of options. The user can make a choice by typing a value in the text field or by choosing an item from the list.
Component	A super class, Most components—for example, menus and toolbars—enable a user to control an application.
Container	A component (such as an applet, window, pane, or internal window) that holds other components.
DHTML	Dynamic HTML is a combination of technologies to make Web pages dynamic by interaction of HTML, CSS and XSL style sheets, the Document Object Model, and scripting.
Dialog	A secondary window displayed by an application to gather information from users. Examples of dialog component include windows that set properties of objects, set parameters for commands, and set preferences for use of the application. Dialog component can also present information, such as displaying a progress bar. A dialog component can contain panes, lists, buttons, and other components.
eNode UI Markup	The eNode UI Markup Language is used to describe sophisticated user interfaces that may be difficult or impossible to describe using HTML

Language	and JavaScript; User interfaces can be reconstructed from markup data using a process called object realization.
HTML	The HyperText Markup Language (HTML) is an example of a language defined in SGML. HTML is a language based on a document composition style known as "markup." HTML outlines hypertext structure, is the publishing language of the World Wide Web.
JAXFront	JAXFront generates the graphic user surface on the basis of an XML Schema. Its business model consists of a standardized model (XML Schema) as well as a concrete development of it (XML instance). The XML Schema describes the syntactic requirements to the business model, while the XML instance represents a concretising of the described model.
Label	Static text that appears in the interface.
Layout manager	Software that assists the designer in determining the size and position of components within a container. Each container type has a default layout manager.
List	A set of choices from which a user can choose one or more items. Items in a list can be text, graphics, or both. List can be used as an alternative to radio buttons and checkboxes. The choices that users make last as long as the list is displayed.
Look and feel	The appearance and behaviour of a complete set of GUI components.
Menu	A list of choices (menu items) logically grouped and displayed by an application so that a user need not memorize all available commands or options.
Menu bar	The horizontal strip at the top of a window that contains the titles of the application's drop-down menus.
Menu item	A choice in a menu. Menu items (text or graphics) are typically commands or other options that a user can select.
Panel	A container for organizing the contents of a window, dialog box, or applet.
Progress bar	An interface element that indicates one or more operations are in progress and shows the user what proportion of the operations has been completed.
Properties	For user interface objects, characteristics whose values users can view or change.
Scrollbar	A component that enables a user to control what portion of a document or list (or similar information) is visible on screen. A scrollbar consists of a vertical or horizontal channel, a scroll box that moves through the channel of the scrollbar, and two scroll arrows.
Separator	A line graphic that is used to divide components into logical groupings.
SGML	SGML (Standard Generalized Markup Language) is a language for

	describing markup languages, particularly those used in electronic document exchange, document management, and document publishing.
Slider	A control that enables the user to set a value in a range—for example, the RGB values for a colour.
Status bar	An area at the bottom of a primary window. A status bar is used to display status messages and read-only information about the object that the window represents.
Submenu	A menu that is displayed when a user chooses an associated menu item in a higher-level menu.
SwiXML	SwiXML is a small GUI generating engine for Java applications and applets, graphical User Interfaces are described in XML documents that are parsed at runtime and rendered into `javax.swing` objects.
Table	A two-dimensional arrangement of data in rows and columns.
Text field	An area that displays a single line of text. In a noneditable text field, a user can copy, but not change, the text. In an editable text field, a user can type new text or edit the existing text.
Title bar	The strip at the top of a window that contains its title and window controls.
Toolbar	A collection of frequently used commands or options. Toolbars typically contain buttons, but other components (such as text fields and combo boxes) can be placed in toolbars as well.
UIML	UIML is User Interface Markup Language that allows designers to describe the user interface in generic terms, and then use a style description to map the interface to various operating systems and appliances;
View	A specific visual representation of information in a window or pane.
Window	A user interface element that organizes and contains the information that users see in an application.
XAML	XAML (Extensible Application Markup Language) is a new scripting language based on XML produced by Microsoft. The main purpose of XAML is to bring both Windows and Web programming worlds together.
XBL	XML Binding Language (XBL) is a markup language for describing bindings that can be attached to elements in other documents.
XHTML	XHTML (eXtensible Hypertext Markup Language) is the combination of HTML and XML. It has taken the vocabulary of HTML and merged that with the syntax of XML.
XIML	XIML (Extensible Interface Markup Language) is an XML-based language that enables a framework for the definition and interrelation of interaction data items.

XML	Extensible Markup Language (XML) is a simple, very flexible text format derived from SGML (ISO 8879). Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere.
XUL	XUL (XML User Interface Language) is a markup language that was created for the `Mozilla` application and is used to define its user interface.

Appendix D: References

[1] Joachim W. Schmidt, Hans-Werner Sehring. *Conceptual Content Modeling and Management: the Rationale of an Asset Lanuage* Proc. Perspectives of System Informatics (PSI'03), 9-12 July 2003, Novosibirsk, Akademgorodok, Russia, LNCS, Springer-Verlag, 2003.

[2] Constantinos Phanouriou. *UIML: A Device-Independent User Interface Markup Language.* Dissertation, Virginia Polytechnic Institute and State University, 2000.

[3] Homepage of SwiXML. http://www.swixml.org, 2004.

[4] Homepage of xulplanet. http://www.xulplanet.com/, 2004.

[5] Mozilla, *XUL Language Spec, second draft.* www.mozilla.org/xpfe/languageSpec.html, 2004.

[6] Stefano Ceri, Piero Tranternail, Aldo Bongio, Marco Brambilla, etc. *Designing Data-Intensive Web Applications.* Morgan Kaufmann Publishers, 2003. ISBN: 1-55860-843-5.

[7] Ben Forta. *WAP, WML und WMLScript : developer's guide.* München, Markt+Technik Verl., 2001. ISBN: 3-8272-5995-9.

[8] Hans-Werner Sehring. *Konzeptorientiertes Content Management: Modell, Systemarchitektur und Prototypen.* PhD thesis, the Software Systems Department of the Technical University Hamburg-Harburg, Germany, 2003.

[9] Martin Fowler. *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley, 2004. ISBN: 0-321-19368-7.

[10] Ivar Jacobson, Grady Booch, James Rumbaugh. *The unified software development process: UML; the complete guide to the Unified Process from the original designers*. Addison-Wesley, 2003. ISBN: 0-201-57169-2.

[11] Hans-Werner Sehring, Joachim W. Schmidt. *Beyond Databases: an Asset Language for Conceptual Content Management.* To be published in the proceedings of Eighth East-European Conference on Advances in Databases and Information Systems, 2004.

[12] B. Shneiderman. *Designing the User Interface.* Addison-Wesley, 1998. ISBN: 0-201-69497-2.

[13] John M. Slatin, Sharron Rush. *Maximum accessibility: making your web site more usable for everyone.* Addison-Wesley, 2003. ISBN: 0-201-77422-4.

[14] Sun Microsystems. *Java Look and Feel Design Guidelines.* Addison-Wesley, 2001. ISBN: 0201725886.

[15] David M. Geary, Cay S. Horstmann. *Core JavaServer Faces.* Sun Microsystems Press, 2004. ISBN 0-13-146305-5.

[16] Miles O'Neal, Tom Stewart. *AWT programming for Java.* New York, M & T Books, 1997. ISBN: 1-558-51494-5.

[17] Douglas Bell, Mike Parr. *Java for students: Java 2 with swing.* Prentice Hall, 2002. ISBN 0-13-032377-2.

[18] Ralf Jesse. *Swing: Swing-Komponenten, Layout-Manager, Ereignisse, Threads.* Kaarst : bhv, 2003. ISBN: 3-8287-2055-2.

[19] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, 1999. ISBN 0-201-31009-0.

[20] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1994. ISBN: 3-89319-950-0.

[21] Sherman R. Alpert, Kyle Brown, Bobby Woolf. *The Design Patterns Smalltalk Companion (Software Patterns Series).* Addison-Wesley, 1998. ISBN: 0201184621.

[22] Aaron Skonnard, Martin Gudgin. *Essential XML quick reference: a programmer's reference to XML, Xpath, XSLT, XML Schema, SOAP, and more.* Addison-Wesley, 2003. ISBN: 0-201-74095-8.

[23] Elizabeth Castro. *HTML for the World Wide Web*. Peachpit Press, 2003. ISBN: 0-321-13007-3.

[24] Deborah S. Ray, Eric J. Ray. *Mastering HTML and XHTML*. SYBEX, 2002. ISBN: 0-7821-4141-2.

[25] Danny Goodman. *Dynamic HTML: the definitive reference*. O'Reilly, 2002. ISBN: 0-596-00316-1.

[26] Neil Bradley. *The XML schema companion*. Addison-Wesley, 2004. ISBN: 0-321-13617-9.

[27] Marc Abrams, Constantinos Phanouriou, Alan L. Batongbacal, Stephen Williams, Jonathan E. Shuster Harmonia, Inc. *UIML: An Appliance-Independent XML User Interface Language* http://www.harmonia.com/resources/papers/www8_0599/index.htm, 2004.

[28] David Hyatt. *XBL - XML Binding Language*. http://www.w3.org/TR/xbl/, 2004.

[29] Homepage of XAML. http://www.xaml.net/, 2004.

[30] Angel Puerta, Jacob Eisenstein. *XIML: A Universal Language for User Interfaces.* RedWhale Software. http://xml.coverpages.org/, 2004.

[31] Homepage of eNode-powerful XML infrastructure. http://www.enode.com/x/index.html, 2004.

[32] Homepage of JAXFront. http://www.jaxfront.com, 2004.

[33] Homepage of Open Dynamic Conceptual Content Management. http://www.sts.tu-harburg.de/~hw.sehring/cocoma/, 2004.

[34] Christian Pesch. *CobWeb-Ein Prototyp für ein personalisiertes, intranet-orientiertes Literaturinformationssystem zur kooperativen Bearbeitung von Bibliographien.* Project report, the Software Systems Department of the Technical University Hamburg-Harburg, Germany, 1997.

[35] Homepage of Warburg Electronic Library. http://www.welib.de/, 2004.

[36] Homepage of coremedia. *CoreMedia Smart Content Solutions.* http://www.coremedia.com/, 2004.

[37] D. R. Olsen. *A Programming Language Basis for User Interface Management. In Human Factors in Computing Systems. Proceedings SIGCHI'89.* Austin, TX, 1989.

[38] W. C. Kim, J. D. Foley. *DON: User Interface Presentation Design Assistant. In Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology.* ACM, New York, 1990.

[39] J. Foley, W. Kim, K. Murray, S. Kovacevic. *UIDE – An Intelligent User Interface Design Environment. In Sullivan, J. and S. Tyler (eds.), Intelligent User Interfaces.* Addison-Wesley, Reading, MA, 1991.

[40] P. Szekely, P. Luo, R. Neches. *Beyond Interface Builders: Model-Based Interface Tools. In Human Factors in Computing Systems.* Proceedings INTERCHI'93, Amsterdam, the Netherlands, 1993.

[41] C. Wiecha, W. Bennett, S. Boies, J. Gould, S. Greene. *ITS: A Tool for Rapidly Developing Interactive Applications.* In ACM Transactions on Information Systems, 8 (1990).

[42] C. Phanouriou, M. Abrams. *Transforming Command-Line Driven Systems to Web Applications.* In Computer Networks and ISDN Systems, 29 (1997).

[43] Homepage of developer. *SWT Programming with Eclipse.* http://www.developer.com/java/other/article.php/3330861, 2004.

[44] Jim Cole, Joseph D. Gradecki. *Mastering Apache Velocity (Java Open Source Library).* Wiley Computer Publishing, 2003. ISBN: 0471457949.

[45] Wolfgang Dehnhardt. *Scriptsprachen für dynamische Webauftritte: JavaScript, VBScript, ASP, Perl, PHP, XML.* Hanser, 2001. ISBN: 3-446-21413-5.

[46] John Pollock. *JavaScript: a beginner's guide.* McGraw-Hill/Osborne, 2004. ISBN: 0-07-222790-7.

Declaration

I declare within the meaning of the examination and study regulations of the international master program course Information and Media Technologies: this project report has been completed by myself independently without outside help and only defined sources and study aids were be used. Sections that reflect the thoughts of other works are made known through the definition of sources.

City	Date	Signature