THESIS / THÈSE

MASTER IN COMPUTER SCIENCE

Design pattern-oriented Software subsystems

Cornet, Louis; Mathieu, Benoît

Award date:
2003

Link to publication

University of Namur - FUNDP

Computer Science Department

Rue Grandgagnage, 21

5000 Namur (Belgium)

Design Pattern-Oriented
Software Subsystems

Louis CORNET & Benoît MATHIEU

Mémoire présenté en vue de l'obtention

du grade de Maître en Informatique

Année académique 2002–2003

Abstract

This thesis suggests a new approach to Software Development using Software Design Patterns: "Design Pattern-oriented Subsystems". Software Design Patterns are proven and generic design solutions to recurring object-oriented development problems. "Design Pattern-oriented Subsystems" are a subtle aggregation of three Design Patterns in one entity. The new concept is to be used as a subsystem foundation for easing the creation of new subsystems in software applications.

To check on its pertinence, this document confronts "Design Pattern-oriented Subsystems" with a range of typical and unavoidable subsystems.

This paper subsequently inspects the requirements one could have from a Design Pattern-capable CASE tool and verifies that tools existing on the market meet these expectations.

Finally, this work puts in perspective the notions introduced such as "Design Pattern-oriented Subsystems" and Design Patterns automation.

Keywords: Design Patterns, Software subsystems, Layered architecture, Software development, Software architecture, Design phase, Business subsystems, GUI subsystems, Preferences subsystems, Persistence subsystems, Design Patterns Automation, Design Pattern-capable CASE tools, Together ControlCenter, XML, XML databases, BML

Acknowledgements

We are very grateful to our advisor, Vincent Englebert, for his valuable opinions and durable support. His comments and proposals have greatly contributed to improving the quality of this document. In addition, this work being the completion of our internship, we also want to thank him for offering us the opportunity to live a wonderful and enriching work experience abroad.

A very special acknowledgement goes to Didier Burton. Didier introduced us to Design Patterns and showed us the beauty of them. During the internship, his attentive and day-to-day coaching allowed us to fully profit of such an experience. We also have to mention his unfailing enthusiasm and vast knowledge of computer science. Thank you for everything, Didier!

There are also many "victims" that deserve our gratitude. Their suggestions on the reading of our document helped us tremendously. We think of Sybille Henry, for her judicious corrections regarding the English writing and the style of the text, as well as Siân Collins and Peter Kelly. Paul Cornet and Dominique Snyers greatly contributed to the quality of the document too.

Next, we need to thank Jean Baltus and Nicolas Gilson which gave us useful advices to prepare our internship abroad, as well as useful tips for mastering LaTeX 2_ε.

We want also to cite everyone that supported us during the elaboration of this study, especially Marie-Noëlle Berlier and our families.

Our last words will be for Misters Harrap and Babylon.

Contents

List of Figures

Introduction

The importance of **software architecture** is common knowledge, since it constitutes the necessary foundation of every software application. From an architectural point of view, an application can be compared with a building. Without good foundations, a building simply collapses. The same applies for a software application: without a solid architecture reflecting a relevant analysis of the problem, it is nearly impossible to build a robust application[1].

The necessity to have a robust architecture has already been exposed by Jean Baltus and Nicolas Gilson in their master's thesis [BG02] and will not, therefore, be discussed further in this study. One can say that building an architecture at the very beginning of the development process avoids bad surprises later on. If the architecture is of high quality, the resulting application will be robust: it will smoothly accommodate changes and addition of new functionalities will become easier.

Some concepts of software engineering are very profitable to the development process. **Subsystems** are semantically useful grouping of classes or other subsystems. It is an application of the well-known "divide and conquer" principle: dividing a problem into smaller problems makes it easier to solve. The main advantage of subsystems is that they tend to make architecture more reusable and robust. Principles of **layered architecture**[2] may help to structure a subsystem.

This document will also present software **Design Patterns**. A software

---

[1]The comparison with building architecture stops there: in building construction, the processes and requirements are well-known and established, whereas software development suffers from changing requirements and technologies.

[2]See Appendix B

Design Pattern is an abstract design solution - in terms of communicating objects and classes - to a particular and recurrent design problem. They offer easy, proven, powerful and high-level solutions in software design. In addition, they capture the experience of many skilled software engineers and make it accessible to non-experts. As a matter of fact, they encourage the reuse of good software architecture practices, particularly significant for successful software development. Among other sources, the book from Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides[3] [GHJV95] has provided interesting material for this study. Each Design Pattern used in this thesis has been defined in [GHJV95].

The **subject** of this thesis is tightly linked to Design Patterns. This work first introduces the concept of Design Patterns and their contribution to software development. Subsequently the document defines a new concept: **"Design Pattern-oriented Subsystems"**. These subsystems hold an astute arrangement of Design Patterns. This new concept will be confronted with a range of typical subsystems of software applications such as an application subsystem, a presentation subsystem, a preferences subsystem, and a persistence subsystem. Other types of subsystems obviously exist, such as a subsystem managing communications with other applications or a security subsystem. These types of subsystems will not be covered in this study. Afterwards, this thesis looks into automation of Design Patterns and "Design Pattern-oriented Subsystems". Finally, this work puts in perspective the introduced concepts. Limits and flaws of Design Patterns having been pertinently exposed in [BG02], one is referred to this document for details on this topic. Criticisms will be focused on the relevance of patterns automation and on the "Design Pattern-oriented Subsystems".

Throughout this study, one example is used in order to illustrate theoretical concepts. This example is a real application: **The Equipment Manager**. It was our responsibility to develop this software application during our internship, from analysis to implementation. The purpose of this thesis is not to describe and explain in detail the Equipment Manager, but only to use it as an illustration. The Equipment Manager offered quite an original standpoint of Design Patterns. Indeed, there are many books about

---

[3]Often referred to as the Gang of Four, or GoF

patterns; however, almost all of them offer quite a theoretical study of patterns. Even if they often give concrete applications, they are rarely inspired from a whole, concrete and "real-world" application.
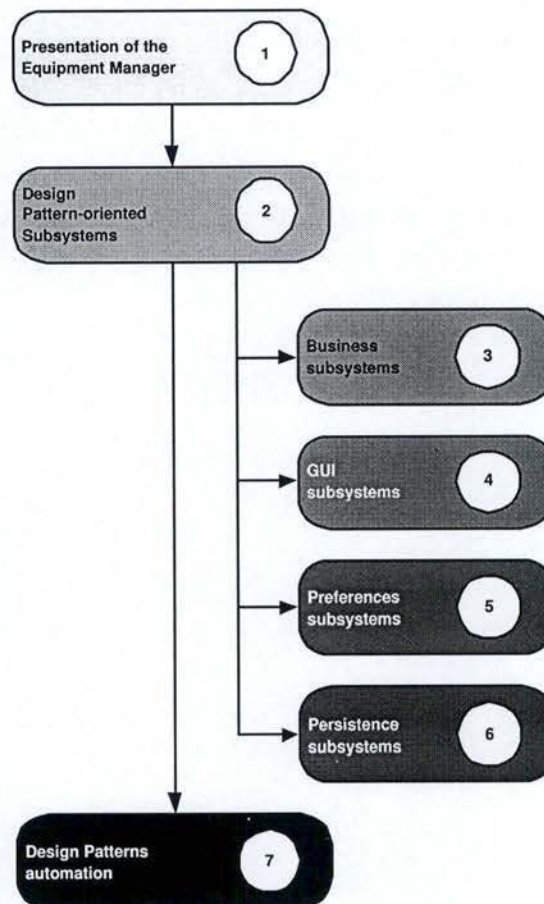
Figure 1: Structure of the thesis

The **structure** of this document is illustrated by Figure 1. Chapter 1 presents the illustrating software: the Equipment Manager. Main concepts, such as subsystems, Design Patterns and "Design Pattern-oriented subsystems", are exposed in Chapter 2. The following four chapters will confront "Design Pattern-oriented subsystems" with typical software application subsystems. Chapter 3 is about application subsystems. In addition, it discusses other Design Patterns used in such subsystems. Chapter 4 focuses on presentation subsystems and studies different technologies to define

graphical user interfaces[4]. Chapter 5 treats of preferences subsystems, while
Chapter 6 handles persistence subsystems. This chapter reviews the most
popular persistence paradigms and presents several Design Patterns useful
in a persistence module. Finally, Chapter 7 covers patterns automation and
a study on Design Patterns-capable CASE tools. This chapter lays down re-
quirements for Design Patterns-capable CASE tools and offers a case study
on the subject. Moreover, this chapter puts "Design Pattern-oriented Sub-
systems" in perspective and analyzes the pertinence of patterns automation.

At last, a piece of advice to the hurried reader must be provided. He
should rather spend reading time on chapters 2 and 7, as they constitute
the core of this thesis.

---

[4]GUI

Chapter 1

The Equipment Manager software

1.1 Purpose

The Equipment Manager software is a product database editor intended to be used in the sound industry. It was our privilege to develop this application for Acme Corporation[1] during our internship in the United States of America.

This piece of software is part of the A-8 suite. The A-8 suite is composed of four parts: the A-8 amplifier and three softwares. These are Acme Configuration Manager, Acme Layout Manager, and the Equipment Manager. The data flow amongst the suite is shown in Figure 1.1. On this illustration, squared objects represent the four elements of the A-8 suite. They are the entities processing data. A logical set of data is represented by the half-curved rectangles. They typically are a file or a database. More accurately, the Equipment Manager creates and edits a product database that is to be used by both Acme Configuration Manager and Acme Layout Manager. Based on the product database, Acme Layout Manager outputs a design file. Using both the design file and the product database, Acme Configuration Manager produces a configuration for the amplifier. The interaction between the participants of the suite is detailed in the following paragraphs.

---

[1]For proprietary reasons, the true name of this company will be undisclosed throughout this text.
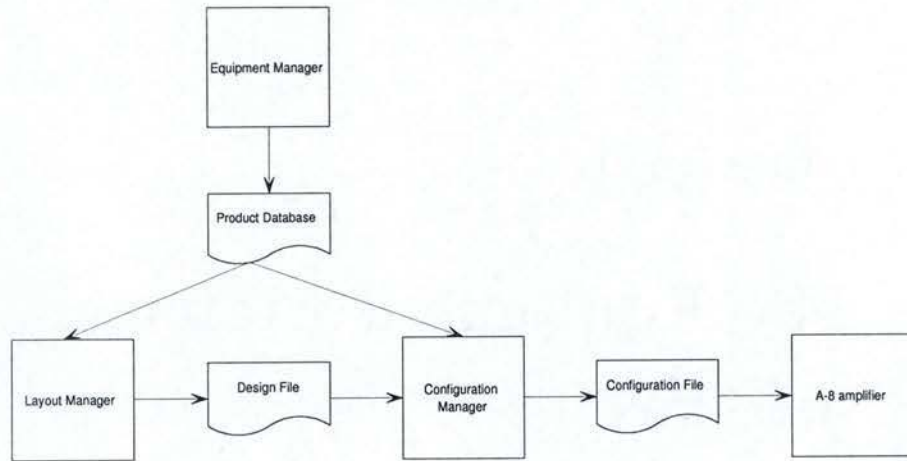
Figure 1.1: The A-8 Suite

1.1.1   The A-8 amplifier

The revolutionary A-8 amplifier is an amplifier that is entirely configurable by software. It is designed for business use, and can mostly be seen in restaurants, pubs, hotels, stores and so forth. It offers original features such as auto volume[2], scheduling, source levelling, signal routing, various equalization possibilities, remote control connections, and more. More details about these features can be found in [BG02].

The amplifier's configuration process is done by Acme Configuration Manager.

1.1.2   Acme Configuration Manager

Acme Configuration Manager communicates with the A-8 amplifier in order to flash a user-defined configuration inside it.

The amplifier can treat up to four input sources and four different playing zones or areas. The software assigns input sources to playing zones. The

---

[2]The amplifier is connected to several sense microphones in order to dynamically adjust the music level in each output zone. This allows people to always hear the music, regardless of the background noise.

user interface also allows different ways of processing sound signals with features like in and out gains, equalizers, auto volume, etc.

More than that, Acme Configuration Manager can verify the performance of systems designed by Acme Layout Manager. The necessary information about these systems is stored in both a design file outputted by Acme Layout Manager, and the product database created by Acme Equipment Manager.

1.1.3 Acme Layout Manager

Acme Layout Manager is needed to describe the layout of a facility, the desired audio components per room, and other requirements. It is used by sales representatives and their customers to define an audio system that suits their needs.

An audio system solution is computed by the software according to all defined requirements. It contains a summary of device types and quantities, their costs, interconnections, and locations inside the facility. The sum of these computations gives birth to the design file, which later will be used by Acme Configuration Manager in order to optimize sound.

All information about every product (loudspeakers, amplifiers, sources, and so forth) handled by Acme Layout Manager come from an external product database. This is where the Equipment Manager comes in. It is its responsibility to produce this database.

1.1.4 Acme Equipment Manager

The Acme Equipment Manager is intended to create and edit a product database. This database can be seen as a "shared resource". Resource sharing occurs when several applications or platforms agree to communicate through a third-party resource, such as a database or file.

As a matter of fact, a set of three software shares access to the resource. The Equipment Manager produces and edits a product database for two other applications. Acme Configuration Manager needs the database for equalizer information regarding the various loudspeaker families, in order

to optimize sound in each playing zone. For Acme Layout Manager, this database is the pool of products considered when creating a design. It is also used for computing the system's price.

The following sections of this chapter will attempt to cover in depth Acme's Equipment Manager.

1.2   Overview

1.2.1   The graphical user interface

Figure 1.2 shows the graphical user interface (GUI) of the Equipment Manager. It illustrates the edition of one selected product. The user first needs to select a product in the database tree (left side) and then may change the product properties at will.

To add a new product to the database, one may merely click on the "Add" button (right below the products tree). As shown in Figure 1.3, a new dialog appears. It asks the user to choose the category of the new product, and to enter its vendor name, model name, and product code. As will be explained further, these four values are key information. The Equipment Manager relies on the four of them to build its product keys. This is why a product cannot be created without having these four values set.

The purpose of the "Edit" button[3] is to set up how the marked-up cost will be computed: by mark-up or by margin. The pop-up dialog box is illustrated by Figure 1.4. Section 1.2.4 explains in detail how this works.

A "System Pricing" panel appears in the very bottom of the database tree (Figure 1.5). This tab permits database-level[4] properties to be edited, such as the currency for all prices stored in the database, lease terms, labor rates, the default labor rate, or the miscellaneous hardware charge[5]. The set of supported languages of the database can also be defined in this panel.

---

[3]Located on the right of the mark-up and marked-up cost fields
[4]By opposition with product-level and application-level
[5]See section 1.2.5 for a definition of database properties
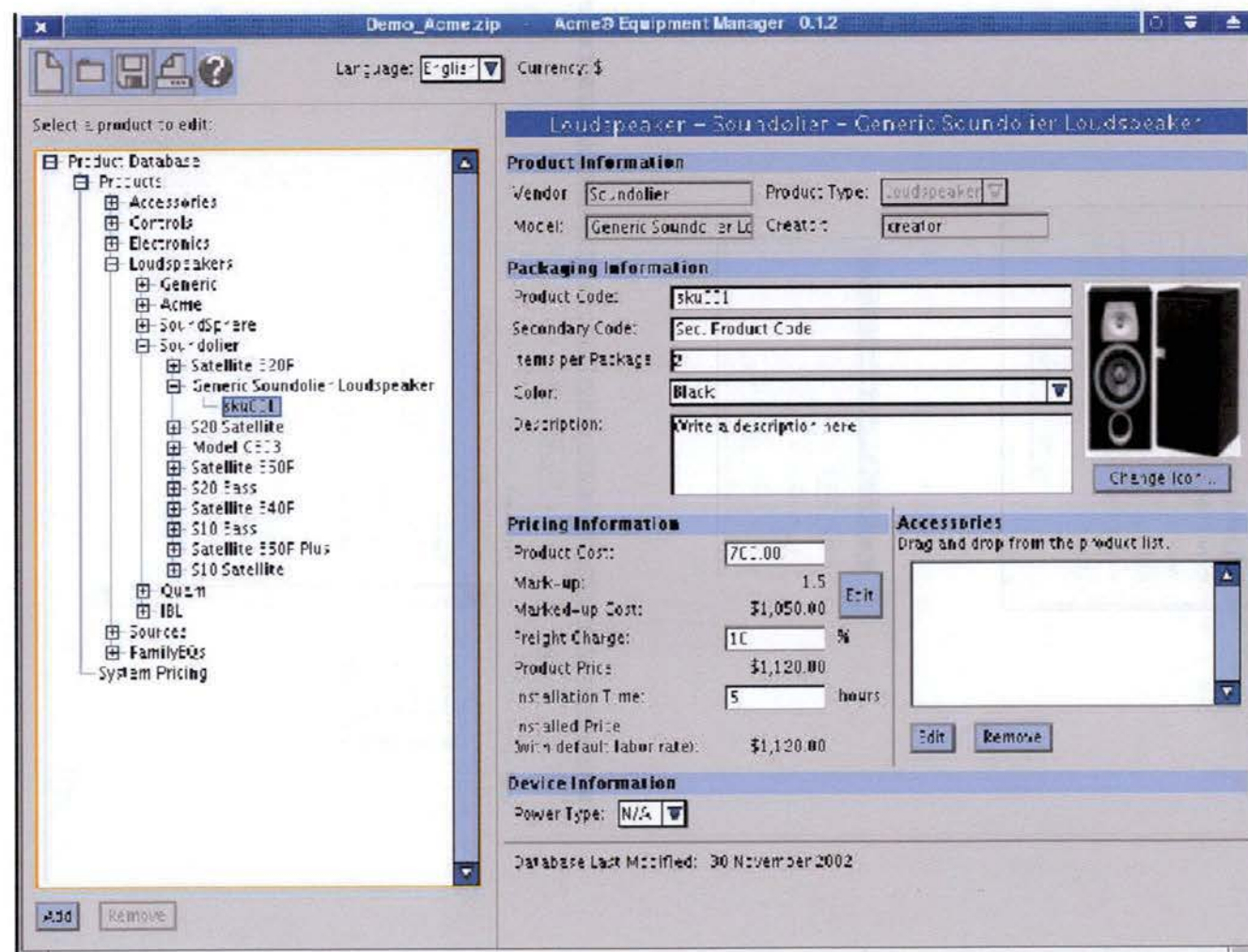
Figure 1.2: The Equipment Manager GUI

Figure 1.3: Addition of a product to the database
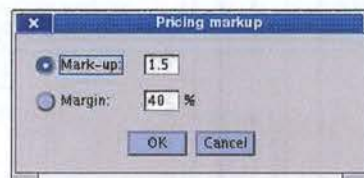
Figure 1.4: Pricing Markup
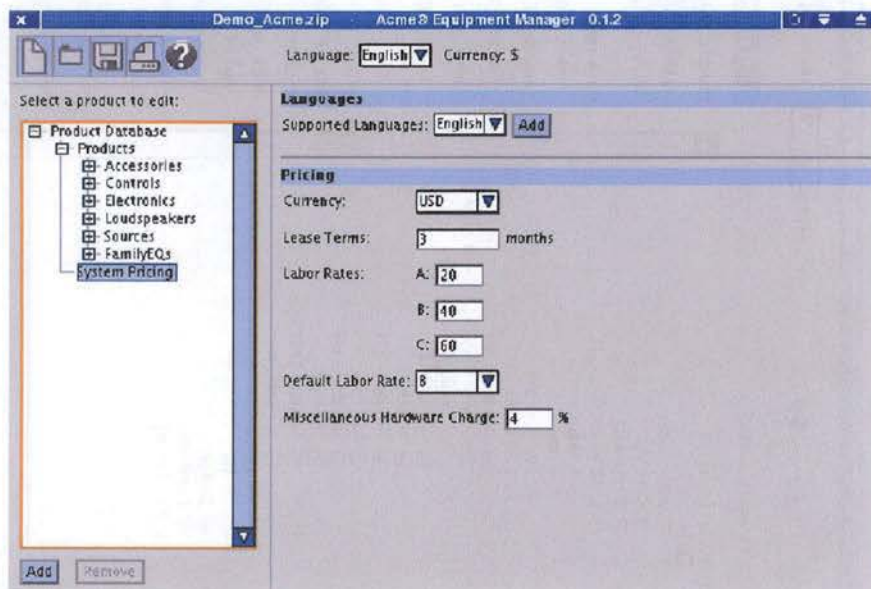
Figure 1.5: The System Pricing panel

1.2.2   The product categories

As stated earlier, the Equipment Manager is a product database editor. Its main purpose is to edit properties of all kinds of audio components. Every component is part of a category. There are six categories, described here below:

Loudspeakers The main category holds all loudspeakers. Loudspeakers are devices that change electrical signals into sounds loud enough to be heard at a distance.

Electronics This category consists of Signal Processors as well as Amplifiers (which are often signal processors that also have an amplification feature).

Sources This category lists any type of audio source: CD player, DVD, tuner, microphone.

Controls Controls are devices like Remote Volume Control, Remote Source Select (items that are mounted on a wall, using a wall plate, for instance, away from the electronics; not items such as a volume knob on the amplifier itself).

Family EQs The idea behind the concept of *Family EQs* is quite simple. There is an equalization curve that improves the sound quality for a given speaker. An electronics device applies this processing to alter the signal as it passes through the device. A "Family EQ" refers to the fact that one EQ may be suitable for more than one speaker model. For example, three different models might all use the same EQ. Because of this, it is possible to add different loudspeaker models to an output channel of an amplifier if they are of the same Family, and all will sound good. If incompatible speakers were put on the same output channel, the signal would not be processed properly for some of the loudspeakers. So each loudspeaker has a best EQ, but each EQ (FamilyEQ) may have a list of compatible loudspeakers.

Accessories This category holds accessories for all other categories.

1.2.3 The product hierarchy

As shown on the left side of Figure 1.2, the tree hierarchy is quite similar to the database structure. Within each category, products are sorted by vendor names, then by model names, and finally by SKU[6] numbers:

Vendor name The vendor of the product is the manufacturing company.

Model name The model is the reference of each product, independently of the way this product is packaged.

SKU number The SKU number is the identifier of the package (an SKU number is unique per vendor only). A package contains one or more pieces of a model, with or without accessories.

The word "product" in this text systematically refers to a package or SKU number, which is the lowest level of the tree structure in the GUI.

One must keep in mind that there is no coordination between the vendors to make SKU numbers unique across vendors. This means that an SKU number can identify different products for different vendors. The proper key to identify a product becomes a combination of the vendor name and the SKU number.

1.2.4 The product properties

Regardless of the category it belongs to, every product is specified by a set of properties. Among them, the vendor name, the model name, and the SKU number. See above for the definition of these three key attributes.

A product can also be characterized by the following properties.

Category The product is either a loudspeaker, an electronics, a source, a control, an accessory, or a family EQ.

Creator The creator is the name of the person who entered the entry in the database.

Secondary code A secondary code is used by a dealer or sales representative to identify the product according to his specifications.

---

[6]Stock Keeping Unit

Items per package The number of items per package represents the number of items of the *model* inside the package.

Color The color of the product.

Description A brief description of the product.

Image An image illustrating the product.

Accessories A list of accessories enumerating the accessories included in the package.

Power type The power type of the device (70 Volts, 100 Volts, Not Applicable).

Pricing information is divided according to several attributes:

Product cost The product cost is the cost of the product dealer or sales representative. It is also the price at which the vendor sells the product (to the dealer).

Marked-up cost The marked-up cost can be computed in two different ways: either with a mark-up or with a margin. The mark-up is a multiplier to be applied on the product cost. The margin, instead, is no multiplier but a percentage. In a more formal way, here is how the marked-up cost is computed:

$$markedup\_cost = product\_cost * markup$$

$$markedup\_cost = product\_cost * (1 + margin/100)$$

The mark-up/margin value is *not a product property*, it is set by a dealer for all pools of products (databases) he/she is dealing with. That is what is called an application-level property (cf. Section 1.2.6). This value can be set by means of the "Edit" button (on its right on the GUI) which pops up the dialog box illustrated in Figure 1.4.

Freight charge The freight charge is a percentage of the marked-up cost to be added to the marked-up cost. The result is the product price.

Product price The product price is the price at which the dealer will sell the product, regardless of installation fees:

$$product\_price = markedup\_cost * (1 + freight\_charge/100)$$

Installation time The installation time is the amount of time, in hours, that it takes to install this type of product.

Labor class The (installation) labor class allows a choice between three labor rates (price/hour). The price for each class is defined in the "System Pricing" panel.

Installed price The installed price is computed by the following formula:

$$installed\_price = product\_price + (installation\_time * labor\_rate)$$

The last product properties are the **technical attributes**. They need to be specified for every product in order to allow the Acme Layout Manager to optimize system solutions. For example, a loudspeaker's technical attributes are taps[7], loudspeaker directivity[8], other band data[9], and so on. Since the requirements on that topic are not stable yet, nothing has been implemented to handle these attributes.

1.2.5 The database properties

System Pricing System Pricing defines a set of database-level pricing information. Among them:

- the **currency** used for all product prices and costs,

---

[7]Some loudspeakers have a built in transformer device with a switchable power setting. For example, a loudspeaker may have 1, 2, 4 and 8 Watt taps. This means that the loudspeaker will be roughly 8 times more powerful when set to the 8 Watt tap than the 1 Watt tap. Taps are used when loudspeakers in an audio system need to play sound at different power levels.

[8]Loudspeaker directivity is an indication of how directional the loudspeaker is, or to look at it another way, how effective the speaker is at taking the sound it produces and sending it in one particular direction instead of all directions.

[9]The set of band data (sensitivity, efficiency, power, etc.) determines the contribution of a loudspeaker at a given location and orientation in space to a given listener location. One can accumulate the contributions of all loudspeakers to get an idea of the quality of sound for a listener.

- the **lease terms**: If a customer does not want to pay the purchase price for the system, he/she may be given the option to lease the system at a monthly fee. The fee is a fraction of the system price, paid monthly, for a set number of months.

- a set of three **labor rates** (A, B, C) for the installation of the device,

- the **default labor rate**,

- and a **miscellaneous hardware charge** to be added to the product price. It includes wiring costs among others.

Available languages   The user may choose a list of supported languages for the database in use. This list is a subset of the list of languages supported by the application.

Version number   A database version number is useful to handle structure changes and to develop backward-compatible[10] applications.

1.2.6   Application-specific properties

Application-specific properties, or user settings, will be managed by a preferences subsystem. This type of subsystem is covered by Chapter 5. Preferences that need to be defined in the Equipment Manager are the following:

Available languages   The application holds a list of supported languages at the application level. The user selects languages within this list that need to be supported by the edited database.

Default values   The Equipment Manager needs to offer to save default values for each product-related properties. The purpose of saving default values is to speed up the product mass addition process.

Mark-up type   As mentioned in Section 1.2.4, the mark-up type can be of two forms: a mark-up or a margin. The mark-up is a multiplier to be applied on the product cost in order to compute the resulting marked-up

---

[10]An application is backward-compatible if it can read and handle previous/obsolete versions of documents it has produced.

cost. The margin, instead, is no multiplier but a percentage. Application settings store both the mark-up type and its value.

1.2.7   Features to be implemented

Multi-language capabilities   The edited database will need to be shipped to dealers and sales representatives all over the world. Data like model names or product description are language dependent. The Equipment Manager needs to allow users to edit the products in their native language.

Application preferences panel   An application preferences panel[11] is required to define each application-specific property (cf. section 1.2.6).

1.3   Constraints

Database readability by the A-8 software suite   The most important restriction that the Equipment Manager needs to consider is to be fully compatible with the other tools of the A-8 suite. Typically, the format of the produced database has to be known by any tool willing to read it. Acme Configuration Manager and Acme Layout Manager are the first targets. The choice of the database format will be discussed in Chapter 6.

Backward-compatibility   The application is required to handle database structure changes. It should be able to read any version of the database, whether the structure is up-to-date or not. The database version number identifies the structure and helps the application to deal with different structure definitions.

1.4   Used technologies and methodologies

During the development of the Equipment Manager, several choices about the right technology to use or the most efficient methodology have been made. As will be shown later on, these choices are tremendously important for the success of a product. Here is a detailed list of the choices that have been made.

---

[11] Not to confound with database-level preferences

1.4.1   Technologies

XML and XML Schemas   For many reasons[12], XML has been used in almost every subsystem of the application, from the GUI to the persistence, passing by storage of application settings. This file format provides great flexibility and reusability at every level. XML Schemas[13] were used to define the database structure, as well as generating objects with the help of the Castor Source Generator.

Castor XML Source Code Generator   Castor is an open-source project of the Exolab organization. Its Source Generator creates a set of Java classes which constitutes an object model for an XML Schema, as well as the necessary Class Descriptors used by the marshaling framework to obtain information about the generated classes. This process is fully covered by Chapter 5. In the case of the Equipment Manager, objects generated by the Castor Source Generator are business objects for the business model (See Chapter 3), preferences serializer (See Chapter 5), or database serializers objects (See Chapter 6).

BML   The Bean Markup Language (BML) is an XML-based component configuration customized for the JavaBean component model[14]. This tool is very handy to define and generate graphical user interfaces. (See Chapter 4)

J2SE   The Equipment Manager is implemented in Java from A to Z. The reason for this is to make it available on several platforms.

Together ControlCenter   Together ControlCenter is a CASE tool which supports several programming languages such as Java, C++, C#, CORBA IDL, Visual Basic, and Visual Basic .NET. It also provides support for common software design tasks. The modeling tool always keeps its source and model diagrams in sync. It is a true architectural guide, revealing the physical and logical layout of a project. Dozens of Sequence Diagrams, Use

---

[12]These reasons will be discussed in Chapters 3 through 6.

[13]See http://www.w3.org/XML/Schema for more information about XML Schemas

[14]See http://www.alphaworks.ibm.com/tech/bml

Cases, Class Diagrams and Robustness Diagrams[15] have been produced by us with this tool during the design phase of the development.

CVS   CVS is the Concurrent Versions System, a tool to manage versioning in any kind of project, from individual developers to large and distributed teams. More than keeping track of the history of every file, CVS also helps developers avoid overwriting each other's changes in the same files. Finally, CVS stores all files of a project in one centralized repository, which promotes good organization and makes back-ups easier.

1.4.2   Methodologies

Throughout the development of the Equipment Manager, business analysts endeavoured to follow the **"Unified Software Development Process"** phases as defined by [JBR99]. As a reminder, these are

1. Use Cases

2. Analysis

3. Design

4. Deployment

5. Implementation

6. Testing

Along with the Unified Process, many concepts have been analyzed and formalized according to the **UML**[16] **representation**, using **Sequence Diagrams**, **Class Diagrams** or **Robustness Diagrams**.

As for the design and implementation phases, business analysts have relied heavily on **Design Patterns** [17] as generic solutions to recurrent problems. Solutions provided by this key methodology will be exposed throughout this text.

---

[15]See Appendix A for more details on UML

[16]Unified Modeling Language, see Appendix A for more details on UML

[17]See Chapter 2 to know more about the motivations of this choice

1.5 Global Architecture

This section will give a global view of the architecture of the Equipment Manager. There is no intention to motivate any choice made by the development team. The situation will be presented as such, and will be motivated and discussed in the following chapters.

The Equipment Manager project is divided into six modules. The **EquipmentOverview** and **EquipmentEdition** modules are the GUI modules. EquipmentOverview handles everything that is related to the tree representing the structure of the database. EquipmentEdition takes care of the edition of product properties. Another subsystem, the **Product Data** subsystem, contains the business model of the application. It holds all the information and manages it with a cache system. The **Preferences** subsystem is in charge of storing user settings. The **ApplicationFramework** is a reusable framework, common to every application of Acme Corporation, that controls the frame of the GUI and other generic GUI components. Last but not least comes the **Persistence** module. This subsystem is responsible for all interactions with the database.

Figure 1.6 shows a Robustness Diagram representing the splitting up of the application into subsystems. Every column pictures a subsystem. Subsystems are organized in four logical layers (View, Application, Domain, Persistence), the horizontal divisions. This diagram shows all the objects, while showing to what subsystem and logical layer they belong.

Subsystems are wanted to be independent of each other. There should be no link (coupling) between one another. Basically, they should not know about the existence of any other subsystem. All the coupling lies in the **Mediators**. The Mediator is just one of the numerous **Design Patterns** used in the described architecture. "Design Pattern-oriented Subsystems" will be studied carefully in the following chapters.

1.6 Summary

This chapter presents the Equipment Manager software, for which both of us were responsible during our internship at Acme Corporation. The ap-

plication has been situated in its context: the set of four products of the
A-8 Suite. An overview in depth of the software and the major concepts
that it is based on has been provided. The chapter then exposes the cho-
sen technologies and methodologies that were used to achieve such a goal.
Finally, one will find an overview of the global architecture of the software
and a brief explanation as to how this system was split into six independent
software subsystems.
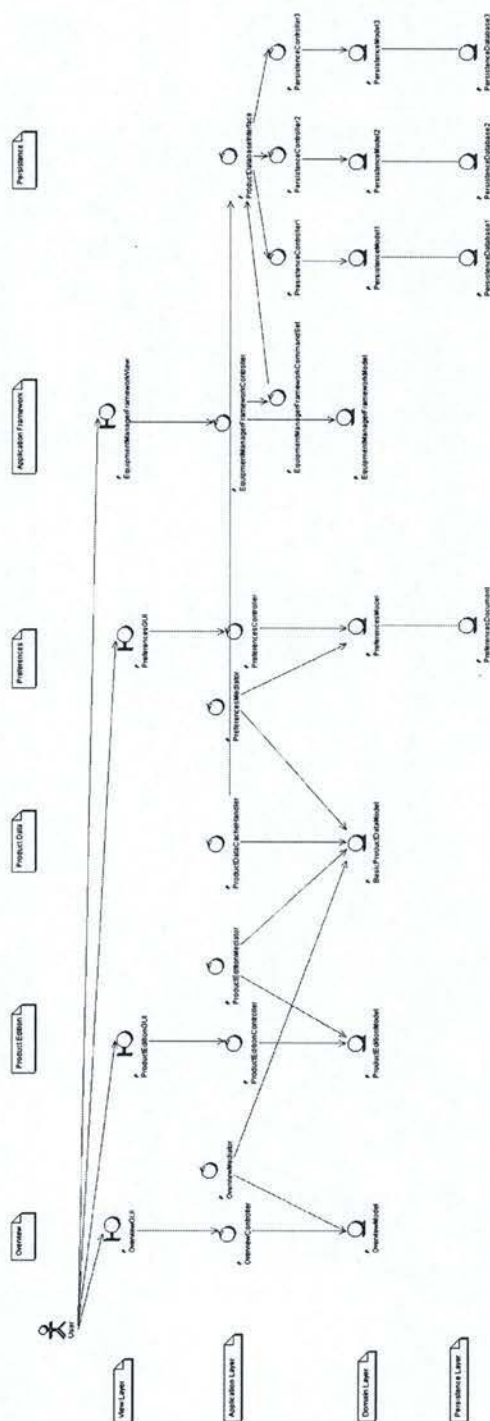
Figure 1.6: Architecture of the Equipment Manager

Chapter 2

Design Pattern-oriented Subsystems

The purpose of this chapter is to define what is hidden behind fundamental concepts of software engineering such as subsystem and Design Patterns and how using Design Patterns can improve the conception of subsystems. First the general ideas about subsystems will be exposed: what they are, and why they are useful. Then, this chapter will offer a glance at the separation into subsystems of the Equipment Manager. Each subsystem will be briefly described.

The main concept of this chapter will then be studied: Design Patterns. After a short definition, the reason for their existence will be discussed. The link between these two concepts will be explained afterwards, when the structure of a general subsystem has been thoroughly covered.

2.1 Subsystems

2.1.1 Motivations

At first, it must be shortly explained what subsystems are. A subsystem is a semantically useful grouping of classes or other subsystems. It is therefore a set of objects working together, that can be considered as a separate entity. Subsystems are an application of the well-known "divide and conquer" principle: it is much easier to divide a big problem into smaller in order to handle it.

For this reason, subsystems are used a lot in software architecture. One of their main advantages is to make architecture more reusable. In an application, a subsystem that has been built in an appropriate way can be used by any another application that needs the same operations to be done. The only thing left to do is to link the subsystem to the application, so that it is known by the application. This point will be discussed later.

In addition, using subsystems can lead to a more robust architecture, that accommodates better changes. For example, when an application needs a connection to a device, all the work and the specific operations necessary to deal with this connection can be put in a separate subsystem, which can be called the Connection subsystem. If the device radically changes or if another kind of connection to the device has to be supported, the only thing has to be changed is the Connection subsystem, and not the rest of the code! This is always true if another device is added.

Dividing a system into subsystems increase the modularity of that system. It is decomposed into smaller entities, easier to handle. It also facilitates the team work: each developer does not work on the same module and the tasks distribution becomes easier. In addition, modularity makes testing easier : each module can be tested separately from the others.

Besides reusability, modularity and robustness, other advantages can be underlined, such as a better legibility and clarity, which can be useful in case of debugging.

2.1.2   Description of the Equipment Manager subsystems

Now a short overview of the different subsystems of the Equipment Manager will be given. The Equipment Manager counts six of them: two GUI subsystems (Overview and Product Edition), one Business subsystem, the Preferences subsystem, the Framework subsystem and the Persistence subsystem.

A) The GUI subsystems

Three subsystems are in charge of the GUI: the Product Edition subsystem, the Overview subsystem and the Framework subsystem. The GUI subsystems will be treated in Chapter 4.

The Product Edition subsystem   The purpose of the Product Edition subsystem is to edit all the properties of a product. In other words, it shows on the screen all information about a product, an object, etc. and allows the user to modify them. It is obviously highly reusable. It can in fact be used in every application in which some properties of an object have to be edited. The edited product corresponds to the selected product in the Overview subsystem.

In the case of the Equipment Manager, the edited properties are information about products, like the product code, the model name or the colour.

The Overview subsystem   This subsystem is responsible for showing all the products of the database opened by the Equipment Manager depicted as a tree. Only one product can be selected at a time. The user selects a product to edit and all its properties are edited in the Product Edition subsystem. The information displayed in the Product Edition depends on the type of product selected.

Division into two separate subsystems instead of one   An important question can be raised at this point. Why is the graphical interface handled by two subsystems? Don't they depend on each other? They actually do, even if they are totally separated. The edited product in the Product Edition is actually the selected product in the Overview and what is displayed depends on the type of product selected. The design of the Equipment Manager has been thought of in terms of reusability and resistance to changes, whatever they might be. So, both of the GUI subsystems can be reused separately, because they are not coupled in the way they are conceived. Each time some information about an object needs to be edited, the Product Edition can be reused, with only a few modifications to integrate it into the application. And each time a tree overview is needed, the Overview

meets the expectations too. Moreover, what if it is decided to change the presentation of the overview of the database in the Equipment Manager? One can imagine, for example, that a tree becomes totally old-fashioned. The change is quite easy to make: only a part of the Overview has to be updated, but that is all! No need to change anything in the Equipment Edition. Of course, there must be some interaction between the Product Edition and the Overview. The question of the communication between two independent subsystems will be covered in Section 2.3.2).

B) The Product Data subsystem

This subsystem is the true business of the application, since it deals with the actual products coming from the database. It is also called the Business subsystem. The Product Edition does not need to know the type of product, it only knows a set of properties, and the Overview knows a sort of generic type. It plays the role of cache for the database. It is the only one to know the persistence subsystem, through an interface. This way, the rest of the application does not know the persistence but only the Product Data subsystem. Chapter 3 will look into Business subsystems.

C) The Framework subsystem

Its only purpose is to extend some classes of a framework, in order to deal with the behaviour of the main frame, the way files are opened, checked and saved. This subsystem will not be exposed in this study since it is beyond the scope of this study. Indeed, this framework is common to other Acme applications, such as the Acme Installer, the Acme Layout Manager (cf. Chapter 1), which must complete this framework so that they have common policies and behaviour for their main windows, etc. It is thus totally Acme-dependent.

D) The Preferences subsystem

The Preference subsystem allows the user to fix some application-dependent variables. Application-dependent means that, once fixed, these values will not change, whatever database is opened. It has to be distinguished from database-dependent. These values are, for example, the different languages supported by the Equipment Manager, the information about the markup

in the pricing system, etc. It offers also the possibility to define some default values for some properties of products: if present, all the products will have the same default values. This will be studied in Chapter 5.

E) The Persistence subsystem

The persistence subsystem is responsible for storing data in a permanent way. Moreover, such a module is required to be able to retrieve, query, and update information. To bring these requirements to life, a persistence subsystem can rely on a database, but also on a custom-made system.

A persistence module should be used as a service. Clients will request to store or retrieve data to an independent service, working as a black box. This ensures that the subsystem is fully reusable for any other application.

Chapter 6 covers in depth this type of module. It also explains the specific implementation used for the Equipment Manager.

2.1.3 Illustration

Figure 2.1 shows the robustness diagram of the Equipment Manager. In this diagram, the division into subsystems (see the vertical lines clearly distinguishing them) and layers is illustrated. Moreover, all components are shown. These components will be detailed below. Principles of layered architecture can be found in Appendix B. Even if it cannot describe the whole design of the application, this diagram is still a very good and useful overview of the architecture of the application. It also enables to see at first glance "who knows who", meaning which component knows about which other.
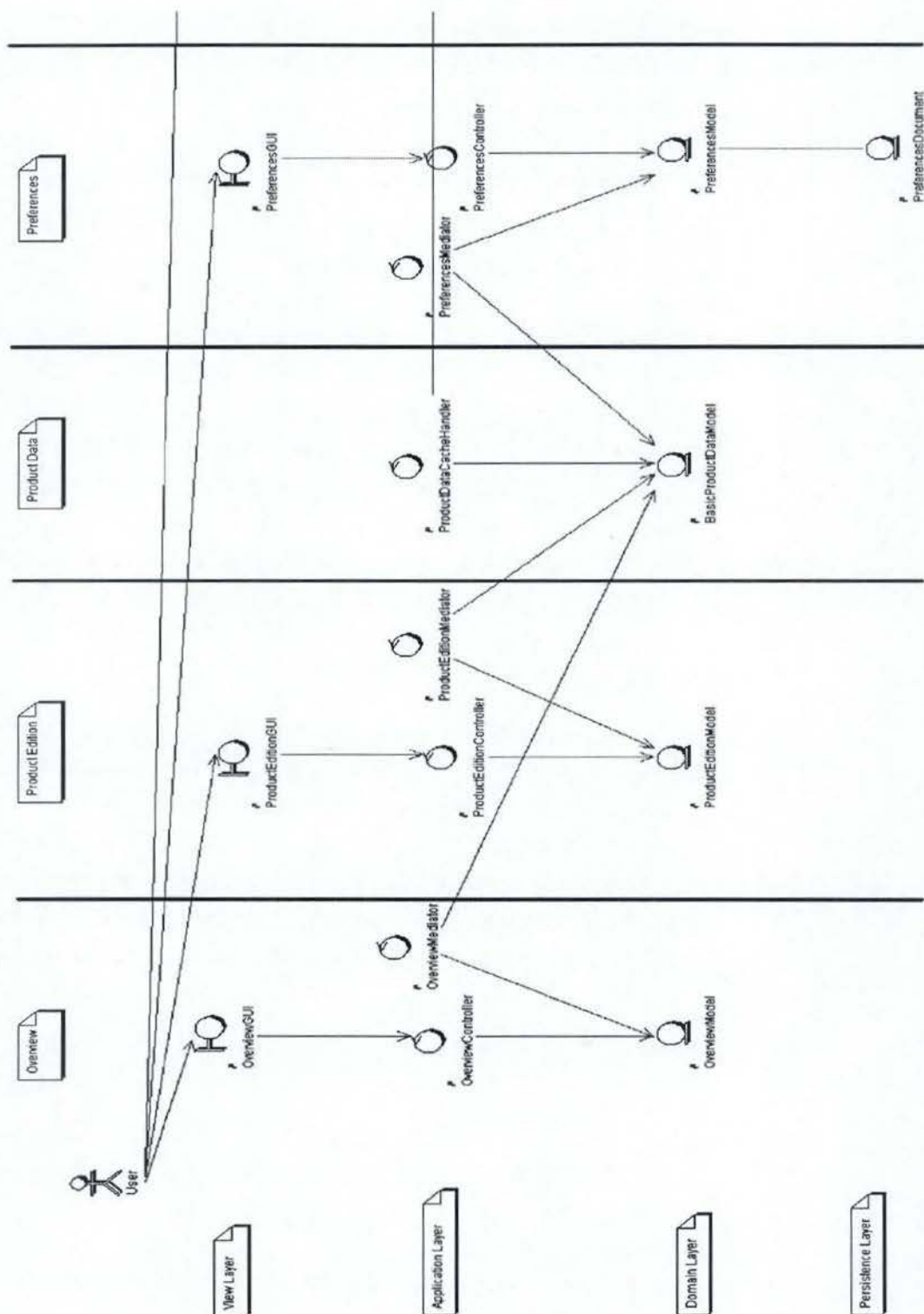
Figure 2.1: Architecture of the Equipment Manager

Figure 2.1 (suite)
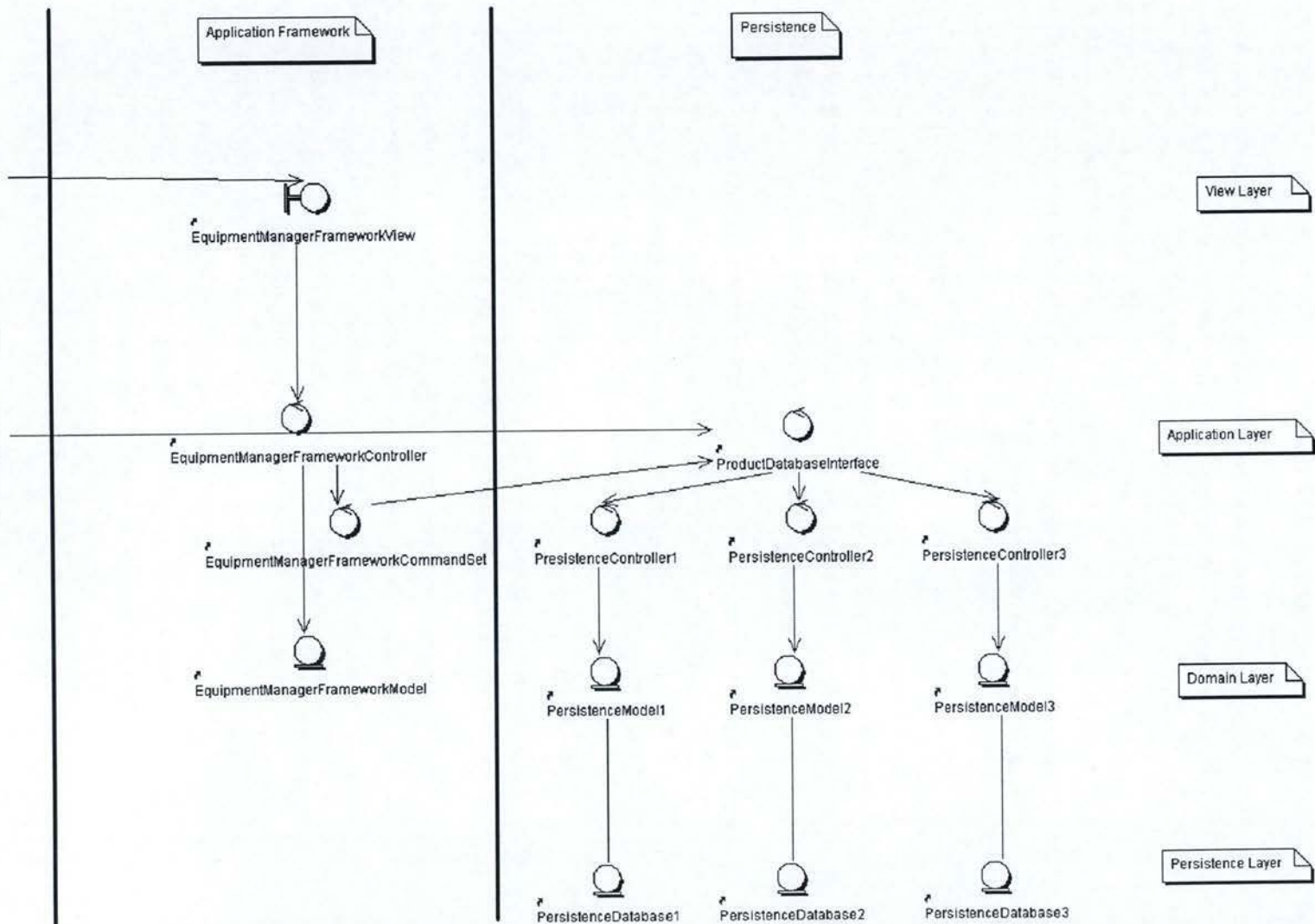
2.2 Design Patterns

The term "pattern" was first use by an architect: Christopher Alexander. *"Each pattern describes problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in a such way that you can use this solution a million times over, without ever doing it the same way twice."* [AIS+77] Even if he was talking of an environment of buildings and town, what he said about patterns stays true in software development.

Design Patterns have been designed in order to prevent the wheel being invented over and over again every day. They capture the collective experience of many skilled software engineers. Every Design Pattern describes a recurrent problem and the core of the solution. A Design Pattern is made up of four fundamental elements: [GHJV95]

Name Naming each Design Pattern makes communication much easier, between people, in documentations, etc. It is always easier to talk about something that has a name everyone agrees on. Each Design Pattern must thus be named for future reference and use.

Problem It explains briefly the problem and its context and thus points when to apply the pattern. In some cases, there is a list of conditions that must be satisfied before applying the pattern.

Solution As expected, the solution is made of the elements of the design, the relationship they have with each other, their responsibilities and collaborations, etc. It is important to underline the fact that the suggested solution is never a concrete design or implementation. Otherwise, it would not be generic and reusable! The solution consists in an abstract design - a configuration of collaborating objects that have to be adapted to the real situation - and how the elements it is made of collaborate to solve the initial problem. It means that it is possible to use the given solution a million times, but never exactly in the same way. The given solutions are the result of years of experience, and are therefore well-proven. They are furthermore presented in a short, easy and understandable form.

Consequences definition In this section, the consequences of the applica-

tion of the described pattern are explained. All the advantages, but also the drawbacks of using the patterns are discussed (flexibility, extensibility, portability, etc.). It is a kind of benefits/costs analysis.

In addition, with the description of a Design Pattern, a short example, with a sample of code is almost always given .

The kind of problems captured in Design Pattern are very frequent for someone designing architectures for software applications. This way these problems have been treated many times, leading to the elaboration of elegant and effective solutions, which are Design Patterns. Each pattern focuses on a specific object-oriented design problem.

Because they offer an easy, well-proven and powerful solution to recurrent problems in object-oriented design, because they capture expertise and make it accessible to non-experts, because they are well defined and contain their own advantages and drawbacks, because their solutions are "high-level" enough (at the object composition level) and can be used a thousand times but never exactly in the same way, for all these reasons, Design Patterns have proven that they are more than relevant and that no object-oriented designer can afford to ignore them...

The use of Design Patterns is not restricted to the object-oriented domain. They are present in solutions for distributed systems, as well as in security issues (security Design Patterns).

2.3   Using Design Patterns in the conception of subsystems

How the different subsystems are built, and how it is possible to use Design Patterns to improve their architecture will be now covered. One will take a look here on how the components of a subsystems are created. This subsystem will be called S, to make it simple (cf. Figure 2.2). The components are represented under the shape of a tree. Each component initiates the components placed below itself. The most important elements of a "classical" subsystem are described, but the View. Actually, the View is created using the BML compiler, but this is deeply covered in Chapter 4. Most of the

subsystems in the Equipment Manager are built this way. The application creates the Façade of the subsystem. After that, the Façade is in charge of creating everything the subsystem needs to work properly. The Mediator is used for the communication with the application or with other subsystems. These Design Patterns (the Observer, Mediator and Façade) will be tackled one by one in the following sections, as well as the benefits their use brings.
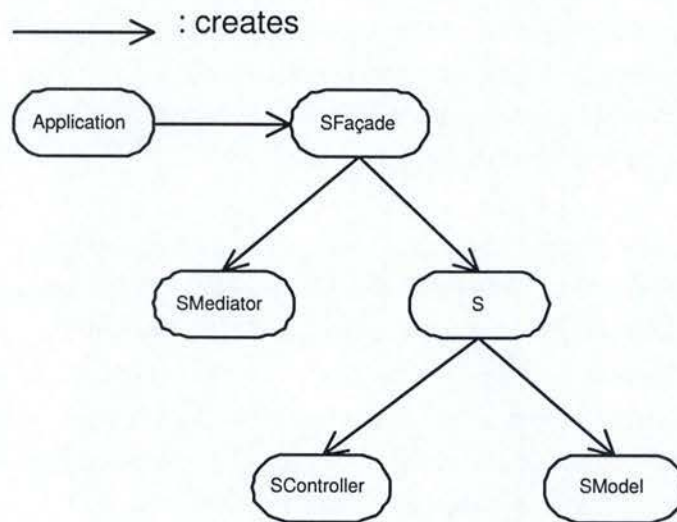
Figure 2.2: Creations of the subsystem's components

2.3.1 The Observer pattern

Motivation

It would be wise to not build a whole system or a subsystem as a single and monolithic object that does everything on its own. That would not respect the philosophy of object-oriented programming. If one imagines now that it is possible to write such a system composed of only one single class. This class would be so huge, and thus impossible to understand and maintain. Every system is actually a set of objects which collaborate with each other. Partitioning a system (or a subsystem, it does not make any difference) into a collection of cooperating classes has an obvious side effect: the consistency of the system has to be maintained all the time. Using strong coupling between objects in order to ensure consistency is not a good idea: it goes

in the opposite direction of reusability - objects that are coupled together cannot evolve on their own without affecting other objects - which is a very important criterion to validate an architecture.

Participants

The Observer pattern has been introduced for this purpose. It defines two types of components: subjects and observers. A subject can have several observers. All the observers of a subject are notified when the subject undergoes a change of state. Each of these observers will afterwards query the subject to synchronize its state with the state of the subject. This kind of interaction is also well-known as publish-subscribe. The reason for this name is quite simple: the subject publishes a change of state in order to notify its observers without having to know a single thing about them. And the observers have to subscribe, so they receive the notifications from the subject. If an object is able to notify other objects without making any assumptions about who these objects are, it means that they are not coupled. There is only an "abstract coupling" between the subject and the observers: the subject keeps a list of observers which respect the Observer interface.

The most famous application of this pattern is what is called the Model-View-Controller (MVC). Generally, the Controller, or a subclass of it, plays the role of the Observer, listening to the changes of the Model or the View (the Subjects), updating the other accordingly. The Model can be considered as a "picture" of the View. The View is constituted by what the user is presented on the screen. The Model is part of the domain layer, the View part of the presentation layer and the Controller, part of the application logic layer. The name Model-View-Controller is sometimes even used, in an abusive way, for the Observer appellation.

Consequences

This pattern (and its application in a MVC) allows multiple Views of the same Model to be presented. [BMR+96] An observer can indeed listen to several subjects or propagate updates to several objects. In the case of different Views sharing the same Model, the Controller listens to the unique Model and can update all the Views or listen to the different Views and modify the Model and the other Views if a View has been changed. It is

important to realize that these Views are synchronized. This process might
be very useful when various users have different expectations or opinions
about the presentation of the same data.

Applying the Observer pattern implies of course some drawbacks. Here
is the most bothering. Some unexpected updates might happen. Observers
have indeed no knowledge of each other's presence and a little change to
the subject - even if it looks totally harmless - can lead to a real cascade of
updates to observers and their objects.
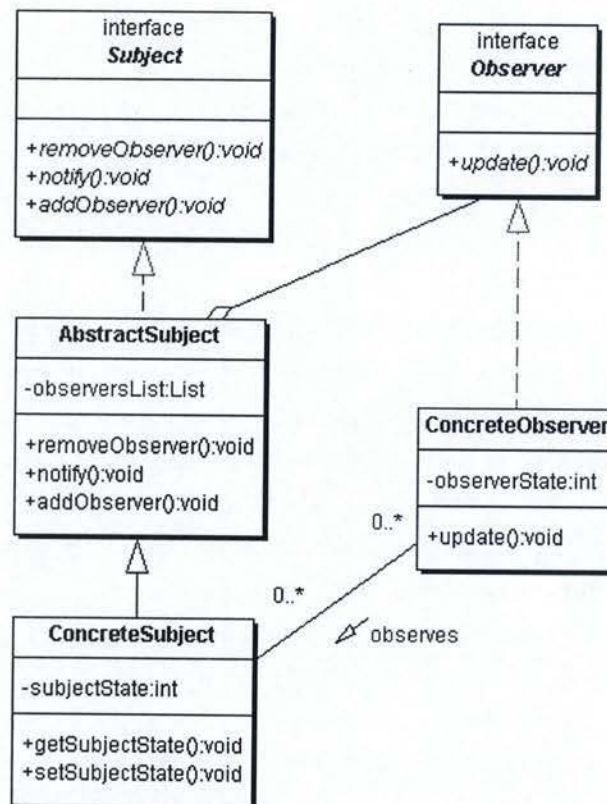
Figure 2.3: Observer's class diagram

Figure 2.3 shows the classes taking part in the Observer pattern. Now
that the observer knows that its subject has changed, it can update its own

state or reflect this change on another object that it knows, like a Mediator (see Section 2.3.2) would do. Doing this ensures the consistency of a system. Every change of state of an object has to be useful. An object that has nothing to do, that nobody knows about or needs does not have its place in the (sub)system. On the other hand, each modification to a useful object cannot be lost, otherwise the application would not be consistent. As stated previously, the Observer pattern is thus a way to ensure consistency with low coupling between objects. The observer is warned in case of the change of one of its subjects, knows exactly what has changed and can reflect it on its own state or other objects. The following sample of code shows how an observer reflects a change coming from a subject to another object.

```
1 private class ProductEditionModelEventHandler
2   implements PropertyChangeListener {
3
4   public void propertyChange(PropertyChangeEvent e) {
5
6   ...
7
8     if (propertyName.equals(ProductEditionModel.
                            VENDOR_NAME_PROPERTY)){
9       String newValue = productEditionModel.getVendorName();
10      String oldValue = (String)e.getOldValue();
11      if (oldValue!=null && !oldValue.equals(newValue)){
12        vendorName.setText(newValue);
13        vendor.setText(newValue);
14      }
15
16    ...
17
18    }
19  }
```

The class ProductEditionModelEventHandler is an inner class of a class named ProductEditionController. This Controller acts as an observer: it adds one of its inner class to the listeners of an object, the ProductEditionModel in this case. When the vendor name property in the Model is

changed, the Controller, as an observer, sees it and knows the new value of
the property. The Controller updates another object by setting its vendor
name property to its new value. This is how every GUI is updated in the
Equipment Manager. The vendor name field comes from the interface and
is a text field.
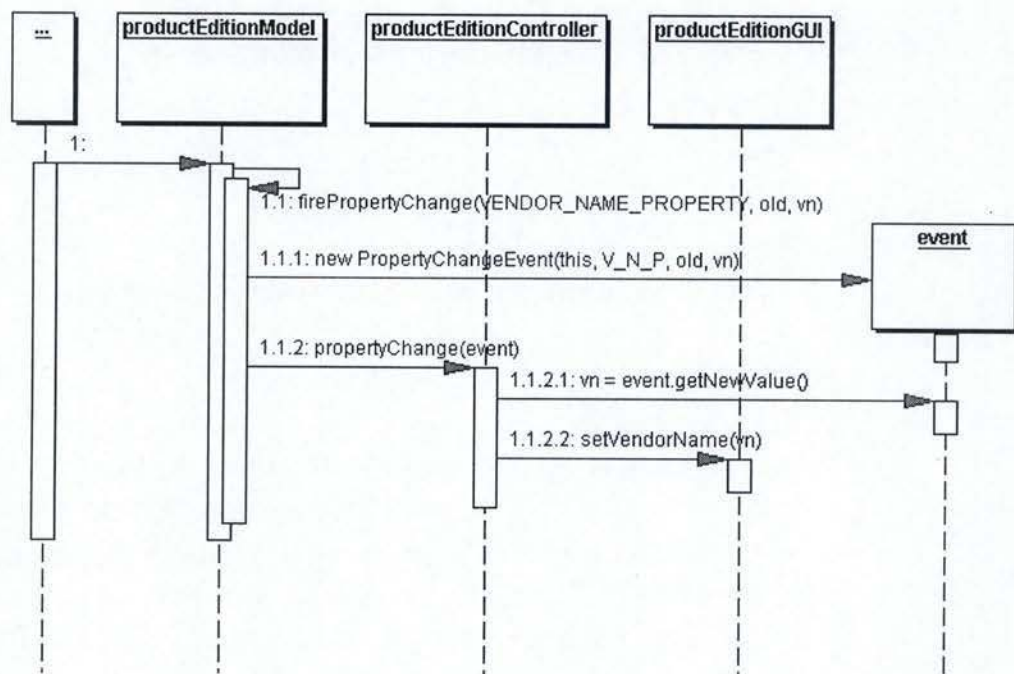
Figure 2.4: Model-View-Controller: Sequence Diagram

Figure 2.4 illustrates the change of the vendor name property in the Pro-
ductEditionModel, reflected on the GUI by the ProductEditionController.

The Model-View-Controller thus maintains consistency within a (sub)system,
through the different layers. The Controller, part of the application logic
layer, is responsible for maintaining both the Model (domain layer) and the
View (presentation layer) up to date.

2.3.2   The Mediator pattern

Presentation

Motivation   A Mediator is an object that encapsulates how a set of objects interact. The purpose of this object is to avoid that each object knows his neighbour; this way, objects do not refer to each other explicitly. Mediators go in the direction of loose coupling between objects.

Participants   Figure 2.5 presents the classes that are part of the Mediator Pattern. Each colleague knows its Mediator and communicates with it whenever it would have otherwise done it with one of its colleagues. The Mediator knows each of its colleagues and is in charge of maintaining them. It implements cooperative behaviour by coordinating colleagues.
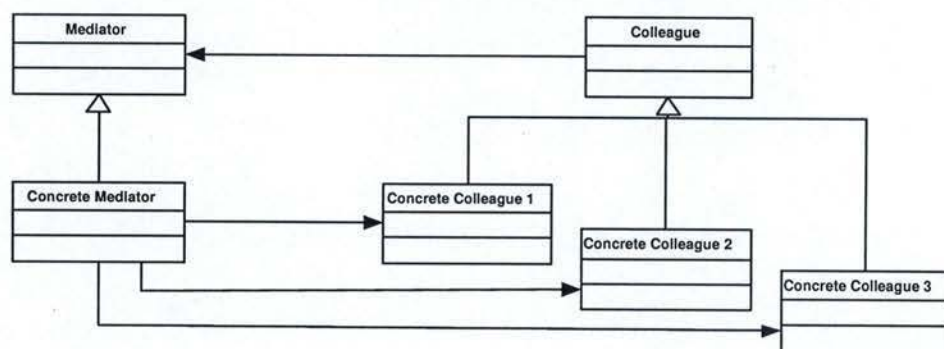
Figure 2.5: Simplified view of the Mediator

"Application history"

In the S subsystem, the only purpose of the SMediator is to interact, to communicate with other subsystems. It does not make sense to have it if the S subsystem is alone. Communication is the real purpose of this object.

This pattern is not applied directly as explained in [GHJV95] (cf. Section 2.3.2). It has been adapted according to the circumstances it is used in. It absorbs the coupling between the Models in the domain layer that belong to separate subsystems.

a) First draft of architecture This example will illustrate at the same time the benefits it is possible to obtain by separating programs in adapted subsystems and the way a Mediator works. This example will be based on the Equipment Manager. The utility of the Observer pattern and its application (Model-View-Controller) has already been discussed in Section 2.3.1. Figure 2.6 shows the very first possible architecture of the Equipment Manager. The Application View contains everything that is shown on the screen
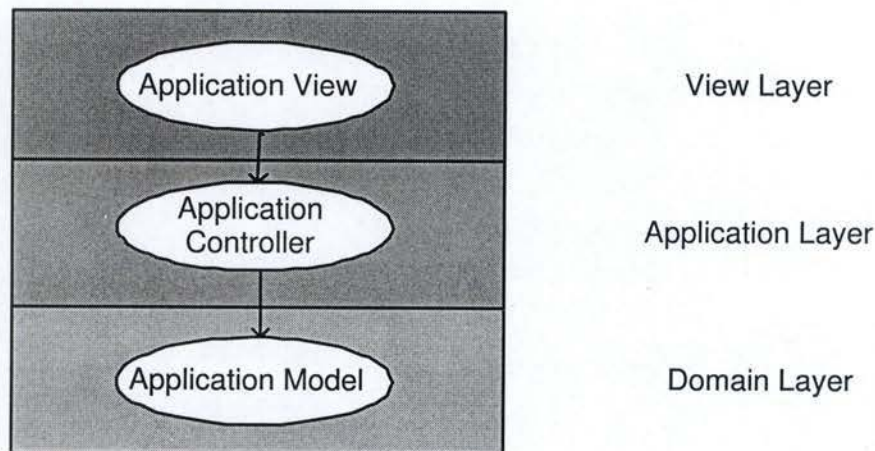
Figure 2.6: First draft of a part of the Equipment Manager's architecture

(every window), the Application Model is a sort of "picture" of the application, meaning that it is a record of the state of the Equipment Manager. The Controller links the View to the Model and makes these two interact.

What happens now if it is decided to make some modifications to the application? The code of a product, called product code or SKU, can be represented either by a string or by an integer. Just for this small change, the Model and the Controller need to be updated. Worse: is there any way to reuse the GUI of the Equipment Manager in a totally different application that also needs to edit the properties of a set of products or so on? That definitely sounds very hard to do. It could be feasible if the View had its own Model (separate from the application Model) reflecting the state of the GUI. This way, every single change made by the user in the GUI would be instantly reflected in the View Model. A separate subsystem (a GUI subsystem) is slowly appearing. (see Figure 2.7)

The GUI subsystem is clearly independent from the application; this means that it can be reused. Each time a set of products, clients or other having different properties has to be shown on the screen, the GUI subsystem can be used with only a few modifications.

There is, however, a primordial difference between the two Models, called Application Model and View Model (or GUI Model). If the GUI Model contains every single piece of information that is displayed on the screen, the application Model itself only holds what is relevant at the application level. The example of the information about the pricing system of the Equipment Manager is a perfect example to illustrate this principle. The user cannot only see the cost of the selected product, but also some additional costs, such as the freight charge. It is trivial that the installed price comes from the addition of the initial cost and all the other costs. This price can thus be deducted from the cost and all the extra charges. The GUI Model will contain all information, such as the cost and all the charges, but also the installed price. On the other hand, there is no trace of the installed price in the application Model because it is irrelevant at the application level, this price being deductible from other properties.

GUI SUBSYSTEM   APPLICATION

View

View Controller

View Model

Application Controller

Application Model

View Layer

Application Layer

Domain Layer
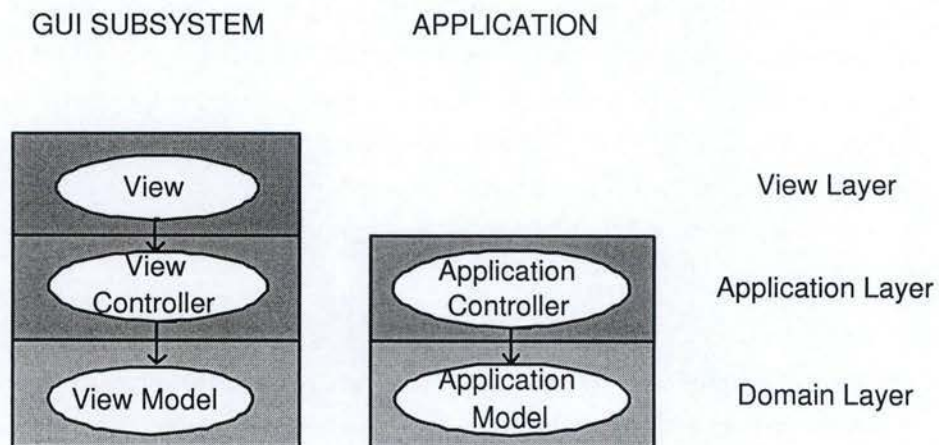
Figure 2.7: Part of Equipment Manager's architecture: second version

b) Division in two GUI subsystems   Obviously there is something missing in this architecture! There is absolutely no way for the two subsystems to communicate, implying that none of the modifications made by the user through the GUI will be reflected in the Application Model. This lack of communication causes incoherence between the Application Model and the View Model. That is why a Mediator will be introduced.

But first it is possible to make a subtler division in subsystems. As explained in Section 2.1.2, we can separate the GUI subsystem into two independent subsystems, each of them with a different aim. This division is made vertically: instead of one single GUI subsystem, there are two of them, each being respectively divided into layers. Horizontally division refers to layers.

The first subsystem, called Overview, presents and displays all the products of the database, in the shape of a tree in this case. The second one, named Product Edition, has the responsibility to show on the screen all the properties of a product. This division is needed for reusability and resistance to change. Each time an application will have to enumerate elements of a set, the Overview can be taken and adapted. And when a list of properties of an object, a product, etc. needs to be displayed, the Product Edition is a good candidate. Each of the GUI subsystems can be easily reused in a large set of different applications. Furthermore, in terms of resistance to change, we can decide to change the way either the Overview or the Product Edition work. For the Overview, it can be decided that a tree is no longer appropriate. The Product Edition will know nothing of the changes in the Overview, and reciprocally. It is in fact possible to add or remove properties of a product without having to make a single change in the Overview. (cf. Figure 2.8)

c) Communication   At this point one can resolve the problem of communication between subsystems by introducing Mediators. The principal difference with the GoF pattern is that in the subsystem S, none of the components knows the SMediator. In the GoF pattern [GHJV95], each object collaborating with the Mediator holds a reference to it. At first, the communication between the Overview and the application will be discussed
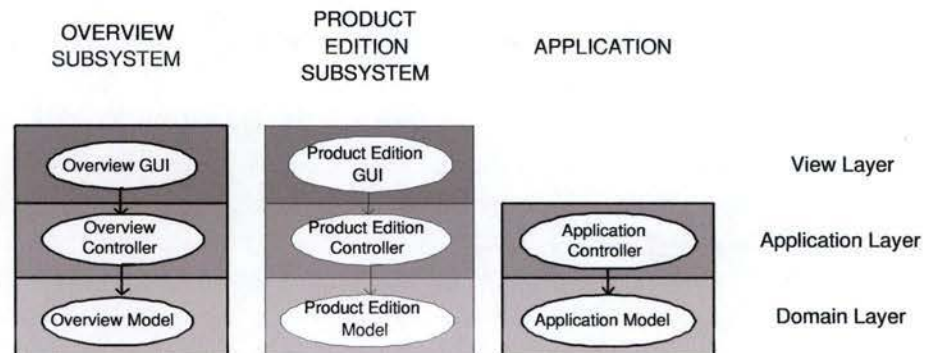
Figure 2.8: Part of Equipment Manager's architecture: third version

and illustrated by Figure 2.9. The problem will be examined in more details subsequently.
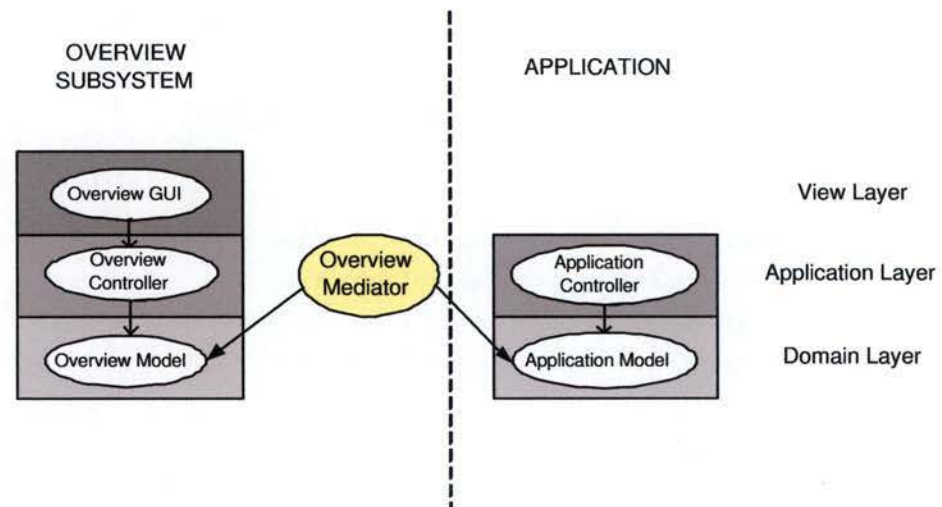
Figure 2.9: Equipment Manager's architecture: introduction of Mediators

The Overview Mediator is part of the Overview subsystem. It is the only component of the subsystem that cannot be reused in other applications. It is actually application-dependent because it is responsible for the communication between the subsystem and the application. So, if the Overview is reused, the Overview Mediator will have to be totally rewritten.

The Overview Mediator listens to the Overview Model and the Application Model, with the mechanism of listeners, exposed in Section 2.3.1. Here is a sample of code, showing how the Overview Mediator handles a change of property (in this case, the user of the Equipment Manager selects another product in the tree or reciprocally the Application Model has changed and the Overview Model has to be updated).

```
1   public class OverviewProductDataModelMediator {
2
3   ...
4
5   //----------------------------------------------------------------+
6   // Private inner classes
7   //----------------------------------------------------------------+
8
9   /**
10  * This class is a listener to a OverviewModel.
11  *
12  **/
13  private class OverviewModelEventHandler implements
                PropertyChangeListener {
14    public void propertyChange(PropertyChangeEvent event) {
15    String propertyName = event.getPropertyName();
16
17    if (propertyName.equals(OverviewModel.
                        SELECTED_OBJECT_PROPERTY)) {
18        Object o = event.getNewValue();
19        if(o != null) {
20          if(o instanceof OverviewController.ObjectWrapper) {
21            OverviewController.ObjectWrapper value =
22              (OverviewController.ObjectWrapper)event.
                    getNewValue();
23            List edited = value.list;
24            int size = edited.size();
25            ProductKey[] edt = new ProductKey[size];
```

```
26              for(int i = 0 ; i < size ; i++) {
27                ProductKey p = (ProductKey)edited.get(i);
28                edt[i] = p;
29              }
30              if(edited != null) {
31                table.put(edt,value);
32              }
33              productDataModel.setEditedProductKeys(edt);
34            }
35          }
36        }
37      }
38 }
39
40 //---------------------------------------------------------------+
41 //---------------------------------------------------------------+
```

The inner class declared in line 13 is a listener to the Overview Model.
When the selected object in the Overview Model changes, the Overview Me-
diator knows it and reflects this change on the Application Model (called
here ProductDataModel, line 33) after a few operations (line 18 to 32).

```
42
43 /**
44  * This class is a listener to a ProductDataModel.
45  *
46  **/
47 private class ProductDataModelEventHandler
48    implements ProductDataModelListener {
49
50    public void propertyChange(PropertyChangeEvent event) {
51      String propertyName = event.getPropertyName();
52
53      if(propertyName.equals(ProductDataModel.
                              EDITED_PRODUCT_KEYS_PROPERTY)) {
54        ProductKey[] selected = (ProductKey[])event.getNewValue();
55        if(selected != null) {
```

```
56              if(table.containsKey(selected)) {
57                OverviewController.ObjectWrapper objectWrapper =
58                  (OverviewController.ObjectWrapper)table.
                                            get(selected);
59                overviewModel.setSelectedObject(objectWrapper);
60              }
61            }
62          }
63        }
64
65      ...
76 }
```

The inner class declared in line 47 is a listener to the Application Model (named here ProductDataModel). When the edited object in the Application Model changes, the Overview Mediator knows it and reflects this change on the Overview Model (line 59) after a few operations (line 54 to 58).

The Mediator is thus responsible for reflecting the changes of one Model on the other. With the mechanism of listener and properties, when a property changes in a Model, an event is raised (propertyChangeEvent) and is treated by the Mediator. It identifies the source of the event and reacts accordingly. The Mediator is a kind of link between properties of Models. It knows which property to change (and how) in which Model when a certain property of a given Model is modified. A simplified sequence diagram illustrates the use case when the user selects another product in the tree (cf. Figure 2.10).

The solution can be extended to as many subsystems as necessary. Figure 2.11 shows how it looks with the application, the Overview subsystem and the Product Edition subsystem. The way the Mediators are placed is important. There is no Mediator between the Overview subsystem and the Product Edition subsystem. This is not an omission. The communication will still be possible between the two GUI subsystems: each property that is changed in the Overview is transmitted to the Application Model and conversely. This is also true for the Product Edition subsystem. The Application Model is updated by the Product Edition Mediator as soon as the
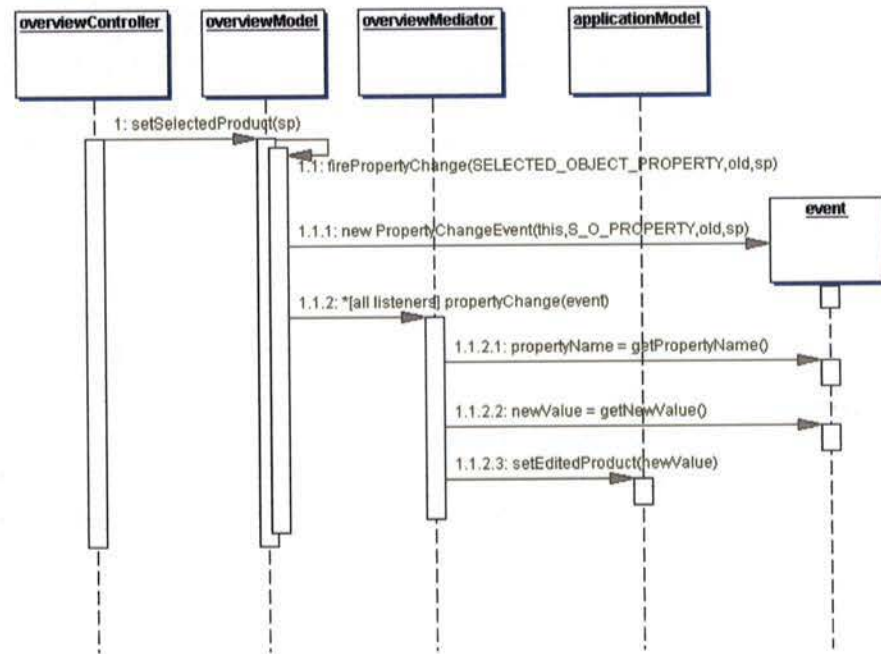
Figure 2.10: Select Product Sequence Diagram
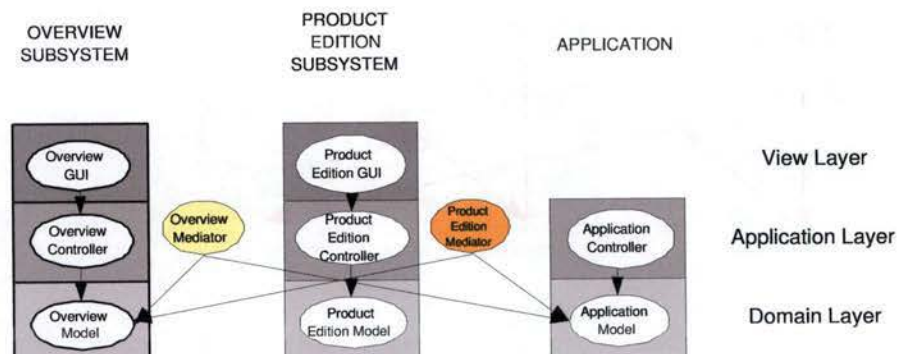
Figure 2.11: Part of Equipment Manager's architecture: last version

Product Edition Model changes, and the opposite is also true. Introducing a Mediator between the two GUI subsystems would not make any sense and would be equivalent to adding redundancy.

At first glance these updates seem to continue indefinitely: when the Overview Model is modified, the Overview Mediator changes the Application Model, the Product Edition is thus updated by the Product Edition Mediator. The Product Edition Model has changed, the Application Model has thus to be modified, and so on. They will stop eventually. When a Mediator receives an event, it checks if the old value of the property is the same as the new one. If it is the case, nothing has actually been modified. The Mediator thus will not propagate anything.

The introduction of Mediators brings a lot of advantages, they allow and simplify the communication between subsystems, they abstract how objects interact. But, on the other hand, they also present some drawbacks, no solution can be perfect... Mediators centralize control; complexity in Mediators is preferred to complexity of interaction. Such a complexity can make a Mediator itself a real monolith that is very hard to maintain and understand.

Furthermore, this architecture with separate subsystems interacting together through Mediators implies a lot of communication. As shown in
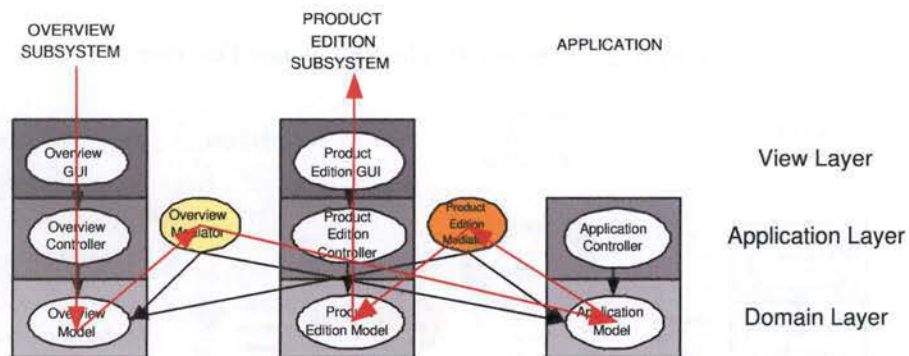
Figure 2.12: Communication increased with Mediators

Figure 2.12, changing selected product implies a lot of events and listeners. One single change in the Overview GUI has repercussions on the Overview Model, through the Overview Controller. The Overview Mediator, listening to every modification of the Overview Model, updates the Application Model in order to keep it accurate. The Application Model has thus just been modified. Two Mediators are listening to this Model: the Overview
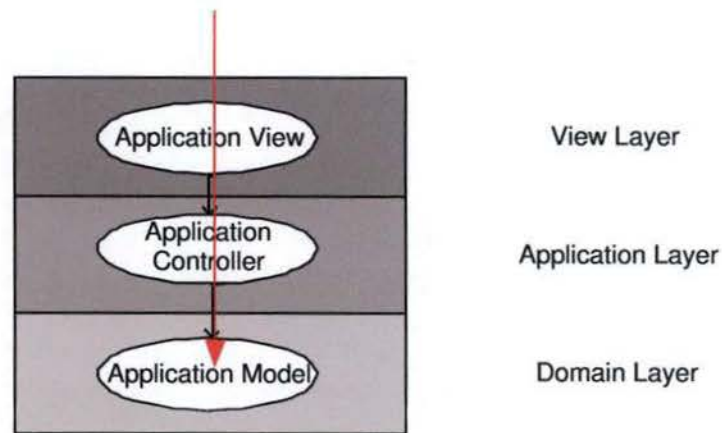
Figure 2.13: Communication in the first draft of architecture

Mediator and the Product Edition Mediator. Nothing happens to the first of them but the second keeps the Product Edition Model up to date by changing some of its properties. The Product Edition Controller just has to bring the changes to the GUI level and the properties of the new selected product are shown to the user.

In the case of the first draft of the Equipment Manager's architecture, the same operation would be much easier; a simple look at Figure 2.13 is sufficient to prove it. However, this architecture presents so many disadvantages that there is absolutely no reason to prefer this one. The purpose of these few remarks was to draw the attention to the fact that Design Patterns are not perfect solutions, but efficient means to improve the design of an architecture.

2.3.3   The Façade pattern

The initial intent of the Façade pattern is to provide a unified interface for a set of interfaces in a subsystem. [GHJV95] The Façade defines a higher-level interface that makes the subsystem easier to use.   As shown in the Figure 2.14, many classes from the outside of the subsystems (let us call them client classes) might need many classes inside the subsystem. The Façade added to the subsystem (cf. Figure 2.15) acts like a front door for the subsystem; the only way to call a subsystem class is through the Façade.
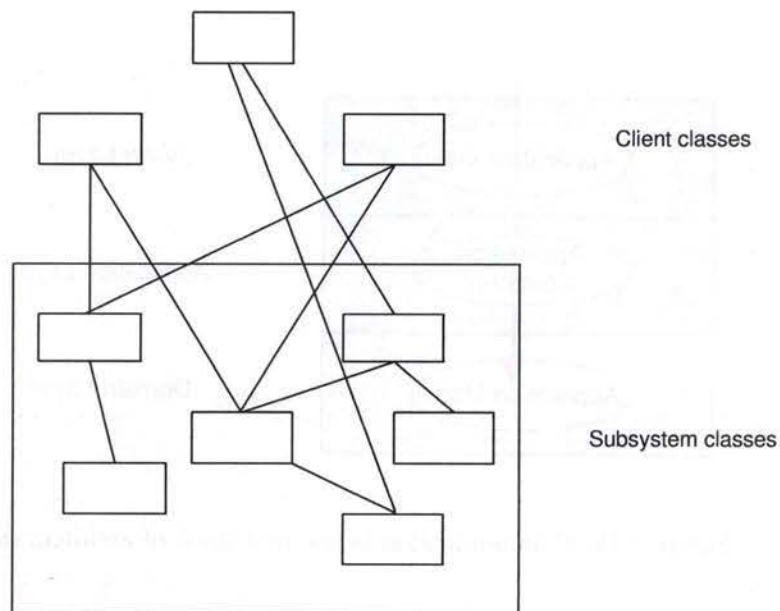
Figure 2.14: Intent of the Façade pattern
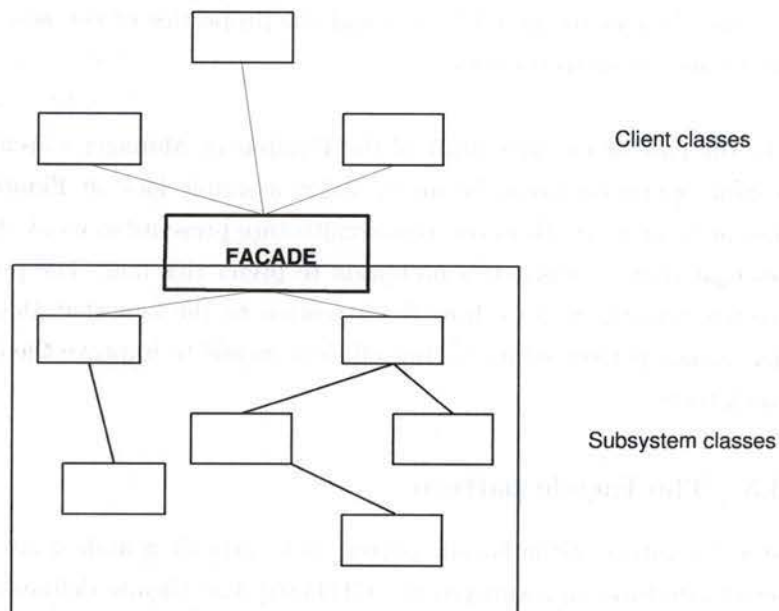
Figure 2.15: Intent of the Façade pattern (2)

As stated before, using subsystems in software architecture helps to reduce complexity. Likewise it allows reusability of subsystems. The condition to reuse subsystems of an application in others is that these subsystems are independent of each other. In fact, if the subsystem A absolutely needs the subsystem B in order to be able to work, it cannot be reused without the subsystem B. This situation cannot be called reusability!

A common design goal is thus to promote weak coupling between subsystems, which means minimizing communication and dependencies between subsystems. Weak coupling between objects or subsystems can eliminate complex or circular dependencies. The Façade pattern has been introduced to achieve this goal. The Façade delegates all the requests coming from the client to the appropriate object in the subsystem. The subsystem classes have to handle the work assigned by the Façade class. They do not even have any knowledge of the existence of the Façade and thus do have any reference to it.

Here follows a sample of pseudo-code to show how subsystems access others through the Façade pattern. In a subsystem A, the AMediator has to handle the communication between the AModel and the BModel from the subsystem B. The AMediator thus needs to know the BModel. It can access it through the BFacade of the subsystem B. When the AMediator is created in the AFacade class, it is given the BModel.

```
1   public class AFacade {
2
3   ...
4
5     public AFacade(BFacade bFacade) {
6
7     ...
8
9     aMediator = new AMediator(A.getAModel(),
10                              bFacade.getB().getBModel());
11
12    ...
```

```
13
14   }
15   ...
16
17 }
```

Both Façades are created in the main application.

SUBSYSTEM A   SUBSYSTEM B
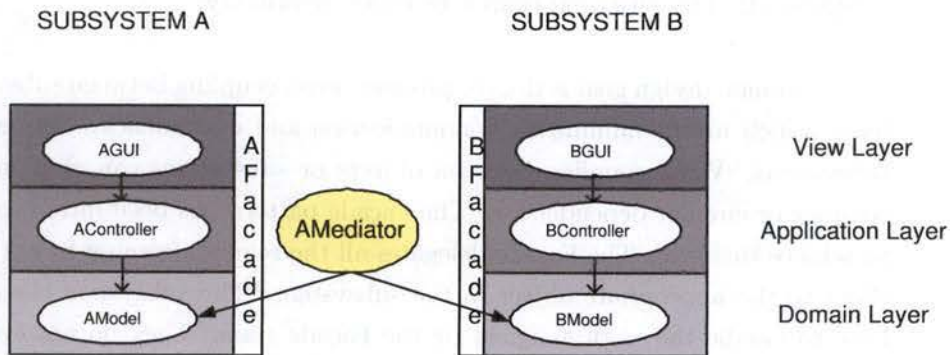
Figure 2.16: Façade and subsystems

2.4   Design Pattern-oriented Subsystems

The term "Design Pattern-oriented Subsystem" will be used throughout this
study. The covered concept is illustrated by Figure 2.17. It gathers to-
gether the three Design Patterns exposed in this section: the Model-View-
Controller, the Mediator and the Façade. It is divided into three layers: the
GUI is part of the presentation layer, the Controller part of the application
layer and the Model part of the domain layer. Layered architecture comes
from another pattern, known as the Layers architectural pattern. [BMR+96]
This way of structuring a subsystem is highly reusable. Almost every sub-
system of the Equipment Manager is built on this structure, with a few
exceptions only. For example, not every subsystem needs a user interface.
As well, a subsystem might not need a Mediator, because it is alone or be-
cause it is the application Model (in this case, every subsystem wanting to
communicate with it needs a Mediator but not the application subsystem
itself). This structure is powerful: it is possible to build a whole application
by simply combining such subsystem (the Equipment Manager demonstrates
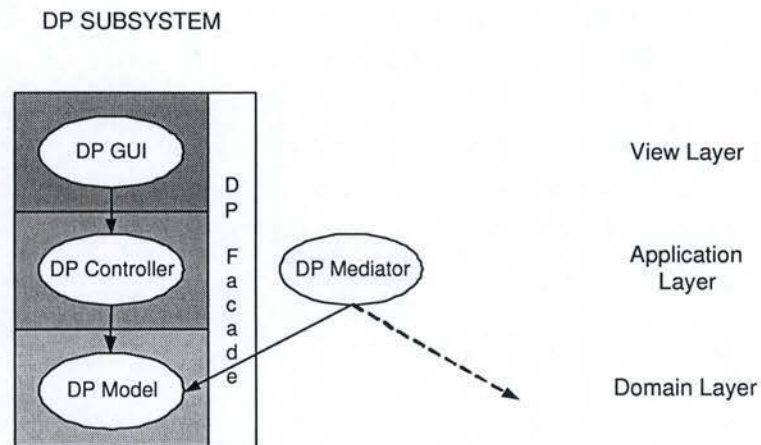
DP SUBSYSTEM

Figure 2.17: Design Patterns Oriented Subsystem

it). "Design Pattern-oriented Subsystems" have all the benefits of subsystems, layered architecture and Design Patterns that compose it.

2.5 Summary

This chapter introduced the key concepts that are software subsystems and Design Patterns. Using the Equipment Manager as an example, it described three Design Patterns: the Observer, Mediator and Façade, showing different phases of their application and the resulting successive improvements. It explained how their use can improve the conception of an architecture by promoting low coupling between objects, allowing reusability and robustness and ensuring consistency.

A new concept, "Design Pattern-oriented Subsystems", has been defined. It combines subsystems and Design Patterns by aggregating an Observer pattern, a Mediator, and a Façade into one entity. "Design Pattern-oriented Subsystems" are to be used as a subsystem foundation for easing the creation of new subsystems in software applications.

This concept will be confronted with different types of subsystems in order to check on its pertinence in specific types of subsystems. Chapter 3 is about business subsystems, which constitute the real heart of an applica-

tion. Chapter 4 looks into GUI subsystems. Chapter 5 is about preferences subsystems. To conclude, Chapter 6 leans on persistence subsystems. This study will be focused on these "classical" and unavoidable subsystems. There are obviously other types of subsystems (subsystem in charge of communication with other applications or a network, security subsystem, etc.) but those will not be covered by this document.

Chapter 3

Business subsystems

This chapter will confront the concept of "Design Pattern-oriented subsystem" defined in Section 2.4 with an Application subsystem, Business subsystem. It will also present a new Design Pattern used in this subsystem: the Decorator.

3.1 Purpose of the subsystem

An Application subsystem contains everything that is typically application-specific. The Application Model is also called the "Truth" since it reflects the application state at every moment. Because this kind of subsystem is so application-specific, it is more difficult to reuse from one application to another.

3.2 Business in the Equipment Manager

Section 2.1.2 gave an overview of the purpose of the Product Data subsystem, also called the Business subsystem or the Application subsystem, since it constitutes the real heart of the Equipment Manager.

The Product Data subsystem directly deals with products coming from the database, called business objects. The Product Data subsystem is the only one to know the database and it accesses it through an interface. These business objects are actually serializers that are generated with Castor. Cas-

tor is responsible for serializing XML[1] documents, but this will be tackled in Chapter 5.

The serializers are generated by Castor in the Persistence subsystem and reused in this subsystem, as the business objects. This is an example of reusability across subsystems.

Accessing the database only through an interface makes the Equipment Manager highly technology-independent. Indeed, the technology used in the Persistence subsystem may totally change, if the database interface is still respected, nothing has to be updated in the other subsystems. This point will be developed in Chapter 6.

It has to be underlined that the Product Data Subsystem is the subsystem which the two GUI subsystems (Product Edition and Overview) interact with. This subsystem is what was called application in Section 2.3.2, since it is the Application subsystem. It is thus very specific to the Equipment Manager.

3.3   A Design Pattern-oriented Subsystem?

A Business subsystem being so application-dependent, it may take several form, depending on the application domain. Hence, it is difficult to confront such a varying subsystem with a "Design Pattern-oriented Subsystem" in general. The following section thus will focuses on the Business subsystem of the Equipment Manager.

The Product Data Subsystem differs in several matters from a typical "Design Pattern-oriented Subsystem" (cf. Section 2.4). Firstly, it has no View at all, since the Overview and Product Edition, that are interacting with it, play that role. In addition there is no Façade for that subsystem. As exposed in Section 2.3.3, a Façade acts like a front door to a subsystem: it defines the only way to call a class of the subsystem. However, this subsystem being a little bit "special", since it is the Business subsystem, it has been decided not to give it a Façade. It is composed of a little number

---

[1]See Section 6.2.5 for motivations about XML

of objects and its Model has to be given to every Mediator of other subsystems. Indeed, the two GUI subsystems, for example, need to interact with the Model of the subsystem, named the Business Model, in order to assume the Equipment Manager's communication and consistency (see Section 2.3.2). The GUI subsystems are thus given the Model of the Product Data subsystem through their own Façade. Communications between this subsystem and the GUI subsystems will not be explained in this chapter as it has already been studied in Chapter 2.

With no View at all, no Façade, and, as it will be exposed in Section 3.4, no real Controller, it does not really fit into the category of a "Design Pattern-oriented subsystem".

Moreover, since it is the Application subsystem, this subsystem is highly Equipment Manager specific and is thus hardly reusable.

3.4   The Decorator Pattern

What deserves some attention is that this subsystem contains an application of another pattern: the Decorator.

The Equipment Manager's robustness diagram (see Figure 2.1) indicates that the Product Data subsystem is made of two components: the CacheController and the ProductDataModel. In this architecture, it appears that the ProductEditionMediator and the OverviewMediator interact with the ProductDataModel. Actually, it is not exactly true. The CacheController seems to be useless, since no objects communicate with it. This is due to the fact that the robustness diagram cannot capture certain features :this kind of schema could not clearly represent what the architecture of that subsystem really is.

As a matter of fact, the CacheController is not really a Controller as explained in Section 2.3.1. It is the Decorator of the ProductDataModel. The Decorator pattern will now be introduced.

3.4.1   Presentation

Intent

The purpose of the Decorator is to dynamically attach some additional responsibilities to an object. Responsibilities of an object are the behaviour other objects expect from that object, it is another term for functionalities. There are two ways to add responsibilities to an object: using inheritance and subclasses, or applying a Decorator. However, inheritance does not allow to add functionalities dynamically: it is done statically. Moreover, extending classes is not always practical. The Decorator provides thus a flexible alternative to subclassing for extending functionalities. [GHJV95]

Decorators are the perfect way to add responsibilities to objects, dynamically, transparently, that is, without affecting other objects.

Participants and collaborations
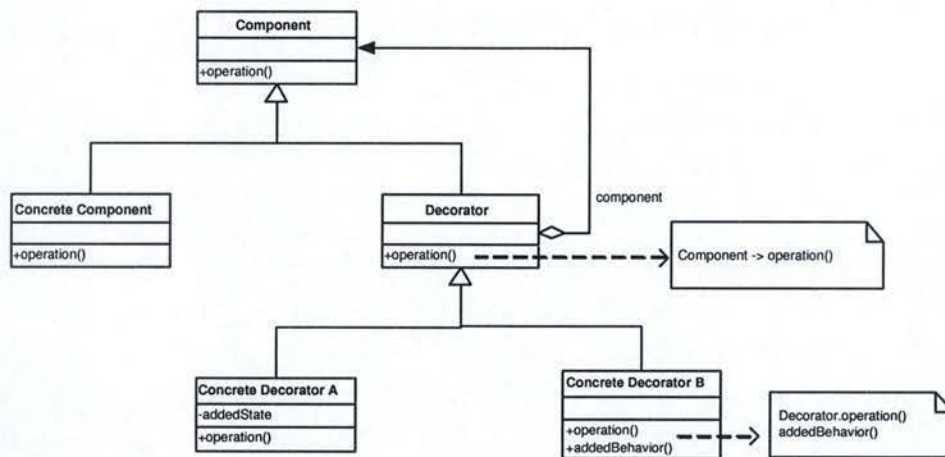
Figure 3.1: Decorator pattern's class diagram

Figure 3.1 presents the participants of the Decorator and their structure. Component defines an interface for objects to which responsibilities can be added dynamically. Concrete Component defines an object to which additional functionalities can be attached. Decorator defines an interface conform to the Component's interface and holds a reference to a Compo-

nent object. As for the Concrete Decorator, it adds responsibilities to the Component. [GHJV95]

As the Decorator is conform to the Component's interface, "clients" of the Component do not even know its presence. These "clients" send requests to the Component. These requests arrive to the Decorator, which forwards them to the Component. It may perform some additional operations before or after forwarding requests.

Example

The following example will clarify the purpose of the Decorator and how it can be an alternative for subclassing. This example is inspired from [GHJV95]. Suppose we have an object that displays a text in a window. It is called TextView. By default, TextView has no scroll bars and no borders, because this is not always needed. A scroll bar is a vertical or horizontal bar that allows the user to navigate into the text, when the text is longer that the available space on the screen. Suppose now we want to add scroll bars and black borders to the text. We want thus to add responsibilities to the TextView. It is feasible, either by using Decorators, or by subclassing.

To add border and scroll bars with Decorators is really easy to perform. Two Decorators are to be defined: a ScrollDecorator and a BorderDecorator. They are respectively responsible for adding scroll bars and borders. As a reminder, the Textview and the two Decorators have to respect the same interface. The ScrollDecorator may the first to apply. "Clients" of the TextView will be given now a reference to the ScrollDecorator. Of course, on their side, they will not see the difference because of the shared interface. The ScrollDecorator holds a reference to the TextView. The ScrollDecorator adds a responsibility to the TextView: from a client point of view, the TextView displays a text in a window and the user can navigate through the text using scroll bars. Request are now arriving to the Decorator and are forwarded to the Component: clients that want to see a text call the TextView, but they are actually using the ScrollDecorator, which forwards the request to the TextView (the text appears on the screen) and perform additional operations (scroll bars also appear on the screen). The same operation can be done with the BorderDecorator: the BorderDecorator holds a reference

to a Component which is the TextView decorated by the ScrollDecorator.

On the other hand, it is possible to add scroll bars and borders to the TextView using subclasses. TextView will have several subclasses: a BorderedTextView, a ScrolledTextView, a BorderedScrolledTextView, etc. BorderedTextView will be used when the user wants to see borders surrounding its text, BorderedScrolledTextView is the class to be used when scroll bars and borders are needed, etc.

Consequences

The main advantage of the Decorator Pattern is that it is much more flexible than static inheritance for extending objects responsibilities. Inheritance creates indeed a new class for each functionality, which, increases seriously the number of classes and the complexity of a system.

Moreover, the application developer does not need to foresee all possible features and the creation of subclasses to support them. With Decorators, features can be added incrementally, avoiding the application to have to pay for features it does not use.

On the other hand, a design that uses plenty of Decorators often results in a system composed of a lot of little objects that all look alike, since the Component and the Decorator share the same interface. But from an object point of view, a decorated component is not identical as the component itself. It should thus avoided to rely on object identity when using Decorators. Such systems, although easy to customize by those who understand them, can be really hard to learn or debug. [GHJV95]

3.4.2   Application

Figure 3.2 shows the main classes of the Product Data subsystem playing a role in the application of the Decorator pattern. ProductDataModel defines an interface that both the Decorator and the Component will have to conform to. The Decorator will be played by the ProductDataCacheHandler, as the role of the Component will be assumed by the BasicProductDataModel.
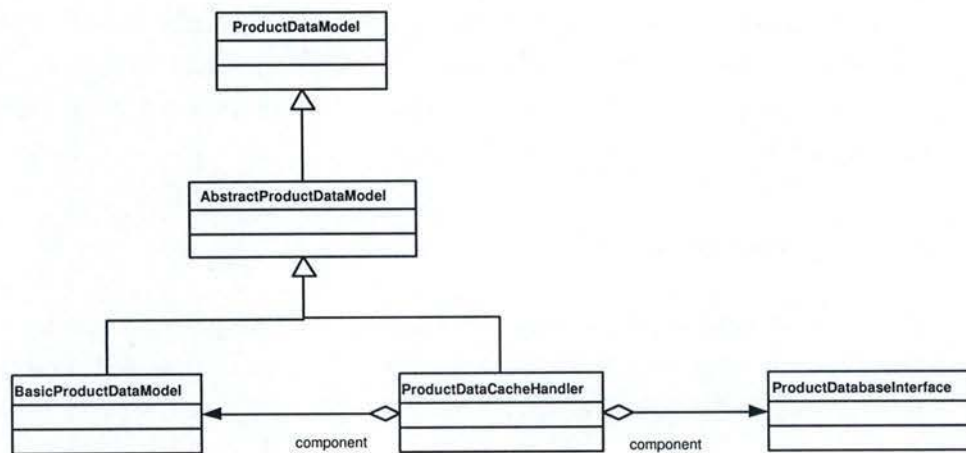
Figure 3.2: Product Data's class diagram

Together, the Decorator and the Component assume the cache of the database. When a product needs to be edited, it is sought in the product database only if it is not present in the cache. The problem that the Business Model (in this case the BasicProductDataModel) cannot play alone the role of the cache; a Model should not be "intelligent" and should not have other responsibilities than holding information and warn when this information changes - otherwise it could not be called Model anymore. That's why some more responsibilities must be attached to the Business Model: hence a Decorator (the ProductDataCacheHandler in this application) is needed.

As they both conform to the same interface, other subsystems are not aware of the Decorator's existence. The two mediators dealing with the Model of that subsystem only know it through the ProductDataModel interface. This way, the Decorator is still transparent.

The Decorator is the only one to know the database through its interface: the ProductDatabaseInterface. Every request coming to the Decorator is simply forwarded to the Component, except for requests that ask for a product. In this case, the Decorator tries to send it to the Component. If the wanted product is not in the Model (the database cache), it forwards a request to the database itself and puts the received object in the cache.

New responsibilities have therefore been attached to the BasicProduct-DataModel, transparently and without changing any other objects. The Decorator pattern has also been used in the Persistence subsystem, tackled in Chapter 6.

3.5   Summary

This chapter presented business subsystems. The business, also called the "Truth", is the heart of an application as it holds the essential data and interacts with all other modules.

Since business subsystems are very application-dependent, no generality can be expressed about them. Because of that, this chapters is unfortunately very Equipment Manager-oriented. The business subsystem of the Equipment Manager was the perfect place to apply the Decorator pattern. This pattern is a good alternative to subclassing when functionalities are to be added to an object. In the case of the Equipment Manager, it empowered the construction of a cache system.

The chapter also demonstrates that "Design Pattern-oriented Subsystems" cannot be systematically used in such a subsystem because business subsystems are so application-specific.

Chapter 4

GUI subsystems

The purpose of this chapter is to expose the problems related to the conception of a graphical user interface and how it is possible to handle them. Expectations about a better solution will be written down and after that, different technologies meeting these expectations or part of will be exposed. Then, the choice of the Bean Markup Language for describing the graphical interface of the Equipment Manager will be discussed.

4.1 Purpose of GUI subsystems

GUI subsystems are in charge of the graphical user interfaces. GUI are very important, because it is the part of the application the user interacts with. Hence, this subsystem is very important: the GUI has to reflect the real state of the application and conversely, user interactions must be brought to the application.

4.2 Building up a graphical interface

The GUI is made of everything that is showed to the user on the screen. It can be a couple of windows with textfields, buttons, menus, etc. They are usually described in separate code files. For the programming language Java, for example, these are Java classes. Some simple and common principles used to elaborate an interface follow. In the coming section the term "GUI component" will be used for every part of the interface - button, textfield or even a whole window. In the Java code, each GUI component is associated with a class that describes it. Each GUI component is declared in the object

77

it will appear in. For example if a window contains three buttons, all these three buttons have to be declared in the window's body.

There must be some intelligence behind the GUI component. A user clicking on a button and having no feedback in return is rarely a happy user. It is imperative that the application knows which GUI component has been used by the user and which action it is to be associated to it. That is what is called - in an abusive way - "intelligence". To meet this requirement a system of events and listeners (similar to the mechanism exposed in Section 2.3.1) is used. In the case of a simple click on a button, an event is created. It has to be possible to figure out easily which component is related to this event. Moreover the event has to be caught otherwise nothing will happen. There is thus an object listening to this component that will react accordingly. This listener can be the same object the component is declared in or another one. In other words, a window (for example) can listen to itself or have a separate listener.

On one side the intelligence lies in the same object containing the GUI component. To gather together the GUI components and the listeners (that handle the events generated by these components) is definitely not a good idea! This would violate the layered architecture principle [1]: there is no more separation between the Presentation Layer and the Application Layer. The GUI component takes place in the Presentation Layer whereas the definition of the behaviour is clearly a part of the Application Layer since it defines how the application should behave after a user action.

On the other side, the layered architecture principle is respected: the intelligence is in a separate object than the definition of the GUI components, which is already much better. The listeners have to make a test to identify the source of the event - in other words which GUI component generated the event - and then give the right answer. However, there are some problems subsisting, that will be exposed in the following paragraphs.

During all the application development process, the GUI may have to go through a lot of changes, even in a short lapse of time. The interface of the

---

[1]See Appendix B for details

Equipment Manager went at least through four of these changes. If changing the GUI is proved to be especially painful, the programmer having to do it can become bald in a very short time. The main reason of this difficulty is all the intelligence behind the interface. If there was nothing behind the components, it would just comes to the creation of new components or the reorganization of the old ones. The real difficulty is not creating new GUI components (which is really painless) or moving components (which is a little bit less painless) but lies in moving their intelligence at the same time! It can even become quite quickly a real nightmare if the listeners of the GUI component are in the same object, meaning that the presentation and the logic behind are mingled. In this case, changes in the graphical interface, even minor, rhyme with a lot of code modifications. The code defining the GUI component and the code defining its behaviour have actually to change or at least move.

In order to cure this evil it might be a interesting to define all the GUI components totally separately from their behaviour. The purpose would be to have nothing to modify when moving a GUI component.

4.3   Expectations

The most expected quality for a mean of GUI description and implementation (as well the declaration of the components as the intelligence hidden behind) is thus the capability to accommodate changes. This quality was also expected from a software architecture.

Resistance to changes is no synonym for highly reduced set of possibilities or esoteric practices. It means that the robustness must not be met at the expense of the simplicity and the effectiveness. It must still be possible to create interfaces without limitations imposed by the need of robustness. And the way to create these interfaces has still to be understandable and usable.

4.4   Existing technologies

This section will highlight some alternatives to create graphical user interfaces. After a small discussion about the "drag and drop GUI builders" (or UI Builders) and the "classical way" of building interfaces, some technologies using XML[2] will be presented. The given list of technologies has absolutely not the pretension to be exhaustive, it only gives some ideas of possibilities to develop graphical user interfaces using the XML technology.

4.4.1   UI Builders

The "drag and drop" tools are a common way to build graphical user interfaces. Users of such tools don't write any code: it is automatically generated by the tool. They just have to compose their graphical elements, for example, they can drag a button and drop it at the place they want it to appear. These tools are very powerful, allowing to make really great user interfaces.

Their principal flaw is that the produced code is almost impossible to change: once designed, it is really difficult to bring some changes in the interface. Basically, these tools make themselves decisions as how to implement the layout of the GUI components, which may not reflect the intentions of the designer, and hence may not behave accordingly when unexpected changes arise (such as window resizing, etc.). Therefore, programmers are better of implementing design intentions on their own rather that letting a program guess them based on a snapshot of the wished result.

Moreover, some existing GUI components might not be handled by the tool. In this case, the user is confronted to choice limitations.

One can conclude that they don't appear to be meet the requirements stated in Section 4.3.

4.4.2   Description in the programming language

Another common way to develop GUI is to directly write the code in a programming language (Java, for example). The importance of separating the declaration of the GUI components and their "intelligence" has already

---

[2]See Section 6.2.5 for motivations about XML

been discussed (see Section 4.2). If this principle is respected, writing Java code without any help from tools or other technologies might present some good results.

4.4.3 Using XML to define GUI

There are, in addition, some really interesting techniques that use the XML format to describe graphical user interfaces. Basically, such techniques can be divided into three major classes, determined by the utilization of XML they make.

Technologies from the **first class** are supported by tools that use XML only "internally". It means that the description of the GUI is stored internally under the XML format (a repository containing a collection of XML files) by the tool. The user does not write its GUI in XML; he/she actually never sees the XML representation. Such tools usually propose only a reduced set of available GUI components. The user interacts with the tool, chooses and assembles the components that will constitute its graphical user interface. XML files are used to store the GUI description, to configure the predefined GUI components, etc. Browser-based Application toolkit and JEasy are examples of such tools. They are both briefly explained in Appendixes E and F.

This type of tool may be very useful for who wants to quickly build a simple graphical user interface. The main advantage is that their use does not require any knowledge of GUI techniques.

However, they are definitely not powerful enough and too restrictive to fulfil the expectations defined in 4.3. Too restrictive because they only offer a set of predefined GUI components. And not powerful enough for many reasons. Indeed, once the GUI is built, it seems very difficult to bring some changes. Once the predefined GUI components are chosen and their arrangement is made, to modify something reverts to change almost everything. Such tools basically suffer from the same drawbacks as the UI Builder. UI Builders might even be less restrictive because offering a wider panel of predefined components. In addition, the user might think that, with tools

such as the BAT[3] and JEasy, he/she will use XML in order to define its GUI, which is absolutely not the case.

The **second class** technologies makes more use of XML. The GUI is still not directly written in XML, but XML is no longer hidden to the user. The XML serialization of Swing components is part of this class.

The **third class** of technologies really enables the description of GUI under the XML format. XML is no longer used only as an internal representation format but as a description mean. The user directly writes his/her GUI using XML. This class contains the Bean Markup Language.

The following sections will focus on the XML serialization of Swing components and on the Bean Markup Language, that is an XML-based language of GUI description.

A) XML serialization of Swing components

Principles   Swing is SUN's library for building user graphical interfaces in Java. As for the serialization, it is a process that supports the encoding of objects, and the objects reachable from them, into a stream of bytes; and it supports the complementary reconstruction of the object graph from the stream.

Swing graphical user interfaces can be serialized as XML documents. The purpose of this XML serialization is the interoperability. *"At the heart of the issue is the question of persistence and how a design can be saved in a format that is not tied to the tool that created it."* [MW99] In order to ensure the GUI to be serialized to an *archive*, a new class has been defined: *XMLOutputStream*. Even listeners can be serialized this way.

Advantages and limits   This technique serializes existing Swing components in XML. This implies obviously that the GUI is developed before its serialization, either with the help of a UI Builder or not. The drawbacks and benefits of such ways to build GUI are thus still valid.

---

[3]Browser-based Application Toolkit

On the other hand, the fact that the GUI is archived in a XML document makes the GUI independent of the tool that generated it. XML is not used only "internally", it is the output format of a process whose main purpose is interoperability.

B) Bean Markup Language

The Bean Markup Language has been chosen to build the user interface of the Product Edition and the Overview subsystems.

Principles Bean Markup Language [4] is an XML-based language use to describe the structure of interconnected JavaBeans [5]. BML is absolutely not just an XMLized Java syntax and is directly executable as it will be shown later.

The main goal of creating BML was to dispose of an XML language allowing to describe declaratively - meaning without procedural code - a whole structure of interconnected beans capable of functioning together as a component, or even as a complete application. *"The development and evolution of BML grew out of a simple challenge: create a mechanism, in Java, using the newly evolving set of XML standards, that can take a description of a hierarchically structured set of data and automatically synthesize a user interface to collect and display the data. Yet, in retrospect, this given direction was pretty close, but not quite correct. A faithful implementation of such a mechanism would only yield an interface for a single instance of data. The true intention had been to be able to generate an interface for the entire class conforming to all the potentially allowable data hierarchies. The correct challenge should thus have been to automatically synthesize interfaces from the DTD [6] or schema describing the hierarchy".* [EWJ99]

Contrary to Java - which loses some of the structure's information in the syntax of the language - BML is a first-class mechanism for capturing the structure of a complete application: it actually gives a complete description of how a set of beans are to be created, configured and interconnected.

---

[4]For short BML
[5]See Appendix C for more details on Java Beans
[6]Data Type Definition

BML defines an application structurally rather than procedurally and can be used to automatically generate interactive interfaces for arbitrary data structures.

As BML is an XML language, it defines a set of tags. Table 4.1) gives an overview of the main BML tags and their signification. [WD99]

Table 4.1: List of most important BML tags

Tags	Description
<bean>	Used for creating or looking up a bean
<args>	Specify constructor arguments
<property>	For the bean property configuration
<field>	For the bean field configuration
<event-binding>	Bind an event from one bean to another
<string>	Create a new string bean or look one up
<call-method>	Call a bean method
<cast>	Explicit type conversion
<add>	Creating bean hierarchy
<script>	Defines a (BML or other) script to be used somewhere

Processing model It has already been underlined that BML was a directly executable language, in opposition to a modelling language. There are actually two different ways to execute the bean markup language. On one hand, the BML Player and on the other hand, the BML Compiler. The BML Player evaluates the BML script at startup-time of an application, since the BML Compiler is a static tool which generates Java code at startup-time, producing this way a bean configuration equivalent to that described in the script. [Joh99a]

Figure 4.1 shows the processing model with the BML Player. The BML Player reads the BML document using an XML parser, which converts the XML to a DOM [7] tree. The BML player then goes through the DOM tree, creating and interconnecting JavaBeans as specified by the tree. The GUI thus appears as described in the BML script. The drawback of this method is that building the DOM tree and then building the resulting structure of

---

[7] Document Object Model
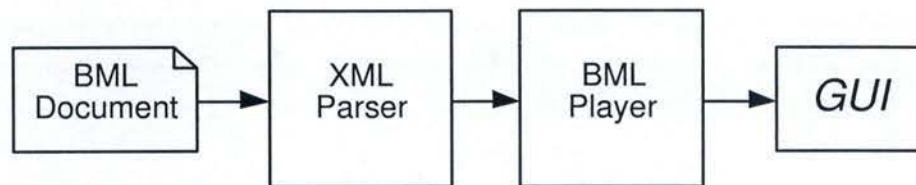
Figure 4.1: BML processing model: the BML Player

beans might result in some time overhead, especially if the operation in-
volves a large number of components.
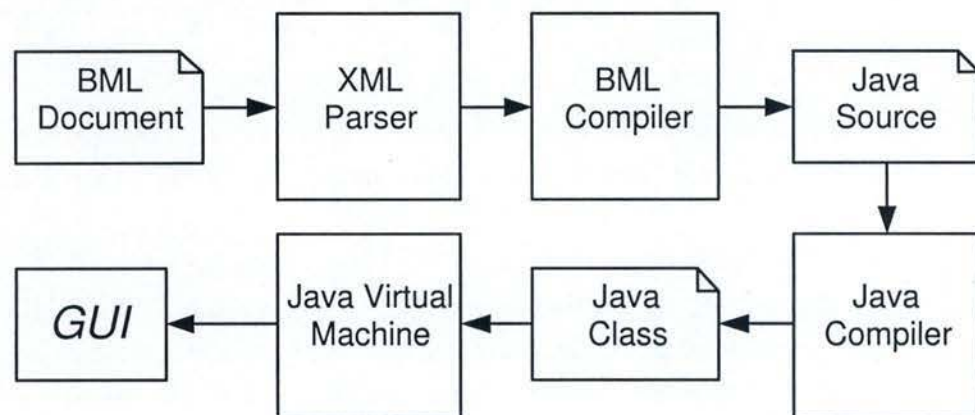
Figure 4.2: BML processing model: the BML Compiler

Figure 4.2 shows the processing with the BML Compiler this time. The
BML Compiler also uses an XML parser to read the XML file, converting it
into a DOM tree. But instead of interpreting this tree as the BML Player
would do, the Compiler generates Java source code which, when compiled
with a Java compiler, results in a class file that will execute as a standalone
program.

4.5   Using BML in the Equipment Manager

In the Equipment Manager, the event-binding capabilities of BML were not
used. As discussed in Section 4.4.3, it is possible to bind event generated
by beans to other beans by using the <event-binding> tag. Instead, an

application of the Observer pattern (detailed in Section 2.3.1) was applied, resulting in a Model-View-Controller architecture. This will be developed further.

All the GUI of the Equipment Manager are described in BML. All the components of the application's main window are described in the same XML file. BML is thus used mostly for its capacity of describing the structure of a set of hierarchically interconnected beans in a declarative way.

For robustness sake, the "intelligence" of the GUI components is separated from their declaration. Declarations (and thus the components structure) may be found in the BML file, while the components behaviour is defined in the Controller.

A simple example of BML utilization is provided in Appendix D. The complete BML code, as well as the Model and Controller codes are given.

The View is constituted by the Java classes resulting of the BML file compilation, using the BML Compiler (see Section 4.4.3). The BML file only contains components declaration. Each bean - which represents a GUI component such as a button or a whole window - receives a unique name in order to identify it. For example, the following line of the BML script defines a bean whose unique name is "product.Top" and which is a Label.

```
<bean class="../bml/macros/RLabelText.bml" id="product.Top">
```

As for the Controller, it contains the behaviour definition of all the components. Since their declaration is made elsewhere, the Controller has to look up for the beans. This is done using a BML parser that traverses the BML file. This is possible thanks to the unique names. The Controller has thus to look up for every bean, no matter where it has been declared, since it does not care of the structure but only of the "intelligence". This constitutes the only difference with a "classical" Model-View-Controller.

Now that the Controller has the components declared in its own body it can handle it. As soon as the user interacts with the GUI, the state of the Controller is changed and reflects this modification on the Model which is a subject for the Controller.

4.6   Advantages and limits of BML

First, it is important to keep in mind that when the advantages and draw-backs of BML will be explained, it is about the use of BML without its event-binding capabilities and combined with the application of the Model-View-Controller to replace these capabilities. Anyway, some of the remarks made in this section are still valid for BML in general.

The principal benefit of BML is surely its huge capacity to accommo-date changes. Most of the time, when a GUI component has to be moved, only the BML is adapted, as long as the component does not change its behaviour. This is feasible thanks to the clear separation made between the declaration of the GUI components (view layer) and their "intelligence" (application layer). Every single GUI component is declared in the BML file, whereas all its behaviour is described in the Controller that is in charge of it. While a component keeps the same unique name and especially the same behaviour, it can move from one place to another without involving any change in the Controller (the only code to modify is of course the BML files). It is obvious that each modification of one "component's intelligence" cannot go without updates of the Controller, but that is bound to happen, whatever technology is used.

The way BML is used in the Equipment Manager implies the application of the Model-View-Controller (exposed in Section 2.3.1). At the same time, it brings all the benefits that go with it, but also the limits and flaws. The consistency in the subsystem is ensured. First, the View will always be the perfect image of the Model and reflect all its changes. Next, the Model will be updated as soon as the View is modified by a user action. On the other hand, the remark passed about the possible unexpected updates is still valid.

In addition, the BML Player enables the developer to have a preview of the GUI he/she is building, without having to launch the application that contains this GUI, which can be very useful when prototyping.

One thing that might dissuade a graphical interface developer is the learning process that it imposes. The user has actually to be used to the XML syntax and has to know at least what a Java Bean is. In addition,

there is no graphical tool supporting the Bean Markup Language. It means that the GUI developer must write the whole BML file without any support, which might seem forbidding because of the XML syntax, especially for beginner user. As an example, the BML file describing the Equipment Manager GUI contains more than three thousand lines... However, this difficulty should not be to hard to pass over because the learning process is really short and the benefits substantial.

Finally, an important comment has to be highlighted: with the Bean Markup Language, there is absolutely no limitation in the GUI component to use for building the graphical interface. This means that every Swing component can be used, or even others - a "home-made" component for example, developed for specific needs.

4.7   GUI subsystems as Design Pattern-oriented Subsystem

Both of the GUI subsystems are typical "Design Pattern-oriented subsystems", as defined in Section 2.4. They have even more been taken as examples to introduce this concept in Chapter 2.

Such subsystems already contain a Model-View-Controller. Then, using BML in order to build the interfaces of these subsystems will be made easier because the Model-View-Controller already exists and does not have to be added. In a typically "Design Pattern-oriented subsystem", the GUI can be described very easily using the Bean Markup Language.

4.8   Summary

This chapter covered the construction of graphical user interfaces. It explained important principles of interfaces construction and listed potential pitfalls. The key principle concerns the clear separation between GUI components declaration and the "intelligence".

In the same line, some expectations about GUI elaboration tools have been established. Among them, it must be commode to operate changes on

the graphical user interface, building up GUI has to be easy and accessible
to everyone, there should not be any restrictions regarding the available GUI
components, etc.

Afterwards, different possibilities to build interfaces have been exposed.
UI Builders, description in the programming language and technologies us-
ing XML have been discussed. Different classes of techniques using XML
were defined, according to the level of use of XML. The XML serialization
of Swing components and the Bean Markup Language have been covered
in depth. Except BML, none of these solutions entirely fulfilled the re-
quirements. Hence, BML has been chosen for describing the GUI of the
Equipment Manager.

At last, the confrontation of "Design Pattern-oriented Subsystems" with
GUI subsystems reveals that "Design Pattern-oriented Subsystems" truly
can improve the architecture of GUI subsystems. This concept is perfectly
adapted to Presentation subsystems.

Chapter 5

Preferences subsystems

This chapter covers preferences subsystems of a software application. It analyzes what a preferences module aims at and what are the different means to achieve these goals. The chapter also illustrates this by a case study on the preferences subsystem of Acme's Equipment Manager. Finally, the chapter studies how good does a "Design Pattern-oriented Subsystem" fit in a preferences subsystem.

5.1 Purpose of the subsystem

Many software applications need to store user-defined settings; settings that are related to the application itself, not to an edited document. They rather concern the behaviour or appearance of the application than the data it is handling. Because users want to be offered to save its settings and wants the application to retrieve them automatically at startup, preferences need to be stored in a permanent way.

The responsibility of managing application settings can be encapsulated in one module. Encapsulation avoids coupling and thus allows this know-how to be fully reusable.

5.2 Storage types

Whatever the file format, user preferences are most often stored in a text file. Advanced systems, though, prefer the use a database.

Text files may rely on an application-defined file format (txt files, ini files) or be based on a standard file format (XML and others) in order to improve portability. Unix systems rather use "conf files" or "dot files"[1], but the principle stays the same.

To illustrate this concept, the **example of XMMS** will be used. XMMS is a cross platform multimedia player that mostly plays audio files such as MP3 files. The application automatically saves user settings in a simple text file (called ".xmms"). The followed convention is to write one setting per line in the file. One setting is defined as follows:

$$field - name = value$$

Among these preferences, XMMS stores window size and position, playing options, and display options. Here is an extract of the ".xmms" file.

```
[xmms]
allow_multiple_instances=FALSE
use_realtime=TRUE
save_playlist_position=TRUE
get_info_on_load=TRUE
get_info_on_demand=TRUE
player_x=749
player_y=0
player_shaded=FALSE
player_visible=TRUE
shuffle=FALSE
repeat=TRUE
autoscroll_songname=FALSE
playlist_x=749
playlist_y=116
playlist_width=275
playlist_height=580
playlist_shaded=FALSE
playlist_visible=TRUE
playlist_transparent=FALSE
```

---

[1]Config files often start with a dot on Unix systems: ".application-name".

```
playlist_position=33
always_on_top=FALSE
sticky=TRUE
```

5.3 Preferences in the Equipment Manager

The Equipment Manager needs to save several **application-level settings**. According to section 1.2.6, these are:

- The list of languages supported by the application,

- Default values for each product property,

- The mark-up type and value.

Albeit the Equipment Manager GUI does not offer any application preferences panel yet, a preferences module already exists to manage the storage of the user settings.

For standardization and potential sharing purposes, Acme chose to store preferences under the **XML format**[2]. The XML files are serialized and unserialized with the help of the "Castor Source Generator".

5.3.1 XML Data Binding

Castor Source Generator is used to perform XML Data Binding. Arnaud Blandin [Bla01] explains XML Data Binding as follows. Many current applications which manipulates XML documents rely on XML Schemas which define the structure, the content and even the meaning of these XML documents. In order to deal with the XML "constraints" defined in the schema, applications need some tools to create and manipulate XML documents that are instances of the given XML Schema.

Such tools might be written using the DOM[3] API[4] or the SAX[5] API, however these approaches are more focused on the structure of an XML document than the data itself, which is a loss of time. Moreover all data

---

[2]See Section 6.2.5 for a more complete list of motivations for the XML format.

[3]Document Object Model, an XML parser

[4]Application Programming Interface

[5]Simple API for XML, an event-driven alternative to the memory-hungry DOM

in these APIs are treated as strings and will likely need to be cast to an
appropriate data type.

It is much easier if these applications can map directly an XML document
to its in-memory object representation that contains all the information
provided by the XML Schema. This is what does XML Data Binding.
[Bla01]

5.3.2   Castor XML Source Code Generator

This presentation of Castor XML Source Code Generator is borrowed from
Exolab.org's user guide for the Source Generator. [Bla01] To represent the
data model of an XML document in memory, developers need to hard-code
the description of the XML document. They need to describe the structure
and the data of the document provided by the XML Schema.

Mapping a String or a Boolean is easy because it is possible to find an
exact mapping in any Object Oriented language. But when it is time to
describe a more complex structure with some inner XML Schema types, it
can become very tedious and complex.

The aim of Castor Source Generator is to provide the necessary code to
describe XML instances of a specific XML Schema with the proper fields
and access methods.

To sum up, one can draw a parallel between the relations XML Schema-
XML and Class-Object: an XML document is an *instance* of an XML
Schema and an Object is an *instance* of a Class. Thus to represent an
XML document as an Object in memory, the Class that describes this ob-
ject is needed.

The Source Code Generator is merely generating the code for this class.
It generates Java source code from an XML Schema. The generated source
includes an object model of the schema as well as the necessary Class De-
scriptors used by the marshalling framework[6] to obtain information about

---

[6]Castor marshalling framework is responsible for doing the conversion between Java
and XML. This is the serialization/unserialization process.

the generated classes.

The object model together with its descriptors will be, from now on, referred to as the "serializer objects". Serializer objects are directly used by the controller of the subsystem in order to store or retrieve information from them. [Bla01]

5.4   Preferences as a Design Pattern-oriented Subsystem

The architecture of a preferences subsystem can be improved by application of the "Design Pattern-oriented Subsystem". The concept defined in Chapter 2 answers well the subsystem needs.

In the Equipment Manager, the structure of the preferences subsystem is just like any other "Design Pattern-oriented Subsystem". A Model-View-Controller pattern encapsulates the knowledge, the display, and the control of the subsystem. One must keep in mind that the View is defined by means of the Bean Markup Language (cf. Section 4.4.3). A Façade pattern also provides a unified interface to the subsystem, making it easier to use from the outside. And a mediator handles communication between this subsystem and the other subsystems of the application.

"Design Pattern-oriented Subsystems" fit perfectly in a preferences subsystem. They can be applied as such.

5.5   Summary

This chapter states that a preferences module is intended to store application-level settings of a user. Encapsulating this responsibility in a subsystem allows reusability across applications.

This chapter highlighted the fact that user settings can be stored in a number of formats. The example of the XMMS software application was provided to illustrate the concept by a typical configuration file.

Storage of preferences in the Equipment Manager is disclosed in this chapter. The properties requiring to be stored, the underlying concept (XML Data Binding) and the central tool of this process (Castor Source Generator) are all covered in depth.

At last, a confrontation of the idea of "Design Pattern-oriented Subsystems" with preferences subsystems in general reveals that "Design Pattern-oriented Subsystems" fit perfectly in this type of module.

Chapter 6

Persistence subsystems

This chapter studies the expectations a developer could have from a persistence subsystem. It also reviews the most popular persistence paradigms before presenting the implemented solution in the Equipment Manager. Finally, the chapter will attempt to prove how useful Design Patterns can be in this type of module.

6.1   Purpose of the subsystem

A persistence subsystem of a software application is **responsible for storing data** in a permanent way. It can be any type of data, stored under any existing format, using a database or not.

Independently of the logical and physical storage model, such a module should be able to store and retrieve data, able to **query and update** the information. Advanced systems could perform **transactions** (group of commands which are to be treated as a single atomic event). **Multi-user and cross-application access** could be required too.

A persistence module should be used as a **service**. Clients will request to store or retrieve data to an independent service, working as a black box. Ideally, requests to this service should be **technology independent**. No client should know which technology is used inside the black box. Therefore, no change of technology should affect a client.

6.2   Persistence paradigms

Persistence can be handled by many different types of systems. It can be managed by a Database Management System (DBMS) or any other persistence system (such as file systems).

A DBMS is a software system that is used both to create databases and manage the information stored within them. The architecture of the DBMS will frequently determine or limit the possible uses of the databases it creates.

In R. Allen Wyke's words, *the most significant difference between them is the model used to store, manage, and query databases. [...] The model used affects the way you will think about data and can be surprisingly difficult to change later.* [WRL02]

This Section will review the most popular Database Management Systems (DBMSs) that have emerged over the years. Other systems, non-database systems, will also be studied.

6.2.1   File Systems

At the very beginning of computer science (1950s), data management was done through "File Systems". The approach was to handle sequential records, each of them containing sequential fields. One can take a look at Figure 6.1 to visualize the type of construction. Such a system relies on indexes for random access.

Disadvantages were numerous: uncontrolled data redundancy, data inconsistency, poor data sharing, difficulty to keep up with changes, low productivity, high maintenance cost. [Hon] Still, file systems are easy and light. They answer well the needs of small systems with few data.

6.2.2   Hierarchical databases

A new concept came into place at attempting to solve part of these problems. Greatly used in the mainframe era (1960s-1970s), Hierarchical DBMSs (HDBMS) *"links records, also called nodes, together like a family tree such*

John Doe
27, Main Street
(234)987-2314

Lisa Smith
67, Border Ave
(234)987-9074

Peter Chaves
34, Appaloosa Circle
(234)987-2839

...

Mike Beauregard
189, Country Lane
(234)987-2839
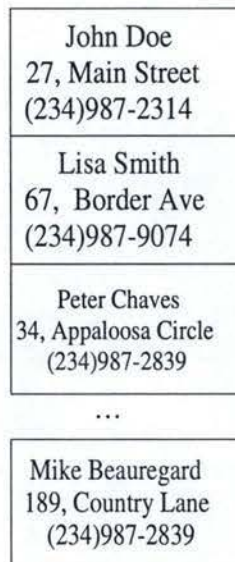
Figure 6.1: A File System structure

that each record type has only one owner." [WRL02]

R. Allen Wyke et al. illustrate this DBMS with an easily understandable example. Figure 6.2 shows a sample hierarchical database containing customers and the orders they have placed.

The database example shown in Figure 6.2 has five nodes of type Customer and five of type Order (because each record has only one owner). These nodes are linked together by pointers that the user must explicitly specify. For example, Order(0706) is linked to Customer(055). All the nodes linked together form a strictly defined tree structure. [WRL02]

Hierarchical DBMS clearly are far from perfect. Some negatives points remain: [Hon]

- Complex record structures

- Difficulty to change (record structures and links)

- High maintenance cost
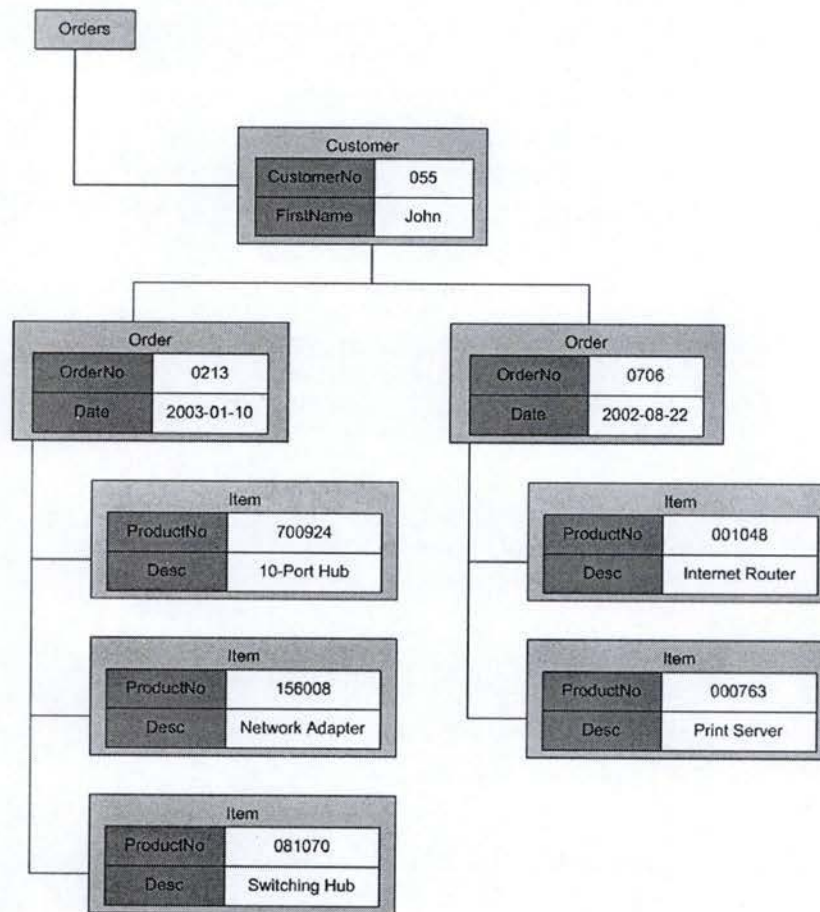
Figure 6.2: An HDBMS example

6.2.3 Relational databases

The relational data model, developed by Todd Codd in 1970 [Cod70], allows multiple tables to be related to one another within a database. Data are modelled as a set of tables where each table consist of a fixed collection of columns, or fields. An indefinite number of rows, or records, can occur within each table.

Relationships between the tables are built by linking key columns from one table to another. The database uses two types of key columns. The first one, called a primary key, is used to uniquely identify rows in a table. The second type, called a foreign key, corresponds with the primary key of

another table to form a parent-child relationship. [WRL02]

Figure 6.3 illustrates the above concepts. Note that CustomerNo is the primary key column of the Customer table, while OrderNo is the primary key column of the Order table. The Order table also has a foreign key column, CustomerNo, which links to the CustomerNo column of the Customer table. Hence, in this case the Customer table is said to be parent and Order the child.

Customer Table

CustomerNo	FirstName
055	John
071	Mary
088	David

Order Table

CustomerNo	OrderNo	Date
071	0105	2002-09-17
055	0213	2003-01-10
071	0221	2003-01-29
071	0358	2003-02-07
055	0706	2003-04-22

Item Table

OrderNo	ProductNo	Desc
0706	000763	Print Server
0706	001048	Internet Router
0221	042209	Keyboard
0213	081070	Switching Hub
0213	156006	Network Adapter
0358	180504	Monitor
0213	700924	10-Port Hub

Figure 6.3: An RDBMS example

The relational model provides a much more flexible framework for data access and manipulation than do the previously studied models. To access the information, users can build queries using the Structured Query Lan-

guage (SQL). With SQL, queries are such that the user specifies *what* data are wanted, and the DBMS figures out *where* and *how* to access the data. These are called associative queries.

For example, to find all Items ordered by Customer John (Figure 6.3), use the following SQL query:

```
SELECT ProductNo, Desc
  FROM Customer, Order, Item
  WHERE Customer.CustomerNo = Order.CustomerNo
  AND Order.ItemNo = Item.ItemNo
  AND Customer.FirstName = 'John'
```

Dr. Shuguang Hong lists the disadvantages of relational DBMSs as follow: [Hon]

- Primitive data values

- Lower level representation

6.2.4   Object-Oriented databases

The Object-Oriented database model emerged in the mid-1980s due to the dissatisfaction of some database users with the limitations of relational DBMSs. The Object-Oriented model defines each piece of data and its associated processes as an individual object. According to the basic tenets of this model, all information about an object is stored in one place instead of being stored across multiple tables, as is done in the relational model.

An Object-Oriented DBMS (OODBMS) has the advantages to group data and processes, it understands complex objects, it is easy to maintain and change, and it improves productivity. OODBMS also integrates more easily with applications that have been written with an Object-Oriented programming language such as C++or Java.

Despite the advantages of the Object-Oriented approach, no standard model for the construction of an OODBMS yet exists, except maybe ODMG. For this reason, at least in part, relational DBMSs still dominate the database market.

6.2.5 XML databases

This section first lists advantages and motivations for the use of the XML format. Then, it explores how XML data modeling is influenced by the type of XML documents developers are dealing with: data-centric documents or document-centric documents. Subsequently, this section will explain how to make the difference between Native XML databases and XML-Enabled databases. The section also will introduce the reader to some interesting standards used to interact with an XML database, such as XPath, XQuery, and XUpdate. At last, the section leans on the XML:DB initiative, which develops specifications for XML databases and data manipulation technologies.

A) Motivations for XML

Over the past five years, XML has become a hugely popular format for marking up all kinds of data, from web content to data used by applications. It is finding its way across all parts of development: storage, display, and transport. Lets have a look at the reasons why XML is so useful for storage.

One obvious advantage to XML is that it provides a way to represent structured data without any additional information. Because **the structure is "inherent" in the XML document** rather than needing to be driven by an additional document that describes how the structure appears as you do with, for example, a flat file, it becomes very easy to send structured information between systems.

Another advantage to the use of XML is the **ability to leverage tools**, either already available, or starting to appear, that use XML to drive more sophisticated behaviour. For example, XSLT[1] may be used to style XML documents, producing HTML documents, WML[2] decks, or any other type of text document. XML servers such as Biztalk allow XML to be encapsulated in routing information, which then may be used to drive documents

---

[1]Extensible Stylesheet Language Transformations. See the W3C recommendation at http://www.w3.org/TR/xslt

[2]Wireless Markup Language

to their appropriate consumers in the specific workflow.

Data serialized in an XML format **provides flexibility with regard to transmission and presentation**. With the recent boom in wireless computing, one challenge that many developers are facing is how to easily reuse their data to drive both traditional presentation layers (such as HTML browsers) and new technologies (such as WML-aware cell phones). XML provides a great way to decouple the structure of the data from the exact syntactical presentation of that data. Additionally, since XML contains both data and structure, it avoids some of the typical data transmission issues that arise when sending normalized data from one system to another (such as denormalization, record type discovery, and so on).

No one can deny the explosion in demand for access over the Internet to the data stored in enterprise databases, nor the explosion in demand for the ability to use the databases to support electronic business operations. These operations include transactions between systems within an enterprise ("enterprise integration"), between businesses in a supply chain ("B2B e-commerce"), and directly to customers ("B2C e-commerce"). XML can provide a huge advantage when **numerous users need different views of the same data**.

B) Types of XML documents: data-centric versus document-centric

It is necessary for the reader to be able to distinguish the two categories of XML documents from each other: data-centric documents from document-centric documents. This categorization is important because it will often define what is possible and what isn't when using XML with a DBMS. Therefore, it is also an important factor in selecting a database. Ronald Bourret[3], an XML database expert, describes these two concepts in his report "XML and Databases" [Bou03a]. The following two definitions are much inspired from [Bou03a].

---

[3]Ronald Bourret is a freelance XML researcher specializing in databases and schemas. He/She has written a number of papers about XML and XML databases. His papers are available at http://www.rpbourret.com/xml

Data-centric documents are *documents that use XML as data trans-port.* [Bou03a] They are designed for machine consumption and application-to-application data exchange. The fact that XML is used at all is usually superfluous. That is, it is not important to the application or the database that the data are, for some length of time, stored in an XML document.

Examples of data-centric documents are sales orders, invoices, flight schedules, scientific data, stock quotes, or application configuration files.

Data-centric documents are characterized by fairly regular structure, fine-grained data[4], and little or no mixed content. The order in which sibling elements occurs is generally not significant, except when validating the document. [Bou03a]

Here is an example of a document that is designed to hold data.

```
<?xml version="1.0"?>
<contacts>

  <contact contactnumber="981240">
    <fullname>Patrick Naughton</fullname>
    <companyname>Acme Corporation</companyname>
    <email>pna@acme.com</email>
    <phone type="business">1-508-766-1601</phone>
    <address>
      <street>1605 Broadway Ave</street>
      <city>Boston</city>
      <state>MA</state>
      <zipcode>02139</zipcode>
      <countrycode>US</countrycode>
    </address>
  </contact>

  <contact contactnumber="981241">
    <fullname>Herbert Schildt</fullname>
    <companyname>Osborne Corporation</companyname>
    <email>hsc@osborne.com</email>
    <phone type="business">1-212-875-1334</phone>
```

---

[4]The smallest independent unit of data is at the level of an element or an attribute.

```
    <address>
      <street>5 West 63rd Street</street>
      <city>New-York</city>
      <state>NY</state>
      <zipcode>01250</zipcode>
      <countrycode>US</countrycode>
    </address>
  </contact>

</contacts>
```

The stored data are contact information for a personal phone book. Notice that every information item, such as full name or the zip code, is represented by an element, and there is no mixed content. The order in which contacts are listed is not meaningful. Similarly, information about one contact can also be permuted between one another without any loss of sense (as long as they stay associated to the top-level entity, the contact).

Document-centric documents are *documents designed for human consumption* [Bou03a], i.e. books, email, advertisements, and HTML/XHTML[5] documents. They are characterized by less regular or irregular structure, larger grained data[6], and lots of mixed content. The order in which sibling elements occurs is almost always significant. Document-centric documents are usually written by hand in XML or some other format which is then converted to XML. [Bou03a]

For example, marking up a paragraph in an article or a book might look like the following:

```
<paragraph>
  <quote speaker="Eustace">"I don't believe I've seen that orange
  pie plate before"</quote>, Eustace said. He/She examined it closely,
  noting that <plot>there was a purple stain about halfway
  around one edge.</plot><quote speaker="Eustace">"Peculiar,"
```

---

[5]Extensible Hypertext Markup Language

[6]The smallest independent unit of data might be at the level of an element with mixed content or the entire document itself.

```
</quote> he declared.
</paragraph>
```

There are two important points to note in this example. Firstly, if the markup was removed, the text of the paragraph itself would still have the same meaning outside the XML document. Secondly, the order of the information is of critical importance to understand its meaning. Reordering elements of the above marked up text radically changes its sense.

Categorizing documents as data-centric or document-centric documents helps deciding what kind of database is best to use. As a general rule (not absolute though), data are stored in a traditional database (relational, object-oriented, or hierarchical) that can handle XML, an *XML-enabled* database. Documents are stored in a *native XML* database, a database designed especially for storing XML, or a *content management system*[7].

C) Types of XML databases: Native XML databases versus XML-Enabled databases

The XML:DB initiative[8] differentiates three different types of XML database: Native XML Database, XML-Enabled Database, and Hybrid XML Database.

A Native XML Database (NXD) is a database that:

1. *"defines a (logical) model for an XML document – as opposed to the data in that document – and stores and retrieves documents according to that model. At a minimum, the model must include elements, attributes, and document order. Examples of such models are the XPath data model[9], and the models implied by the DOM[10] and the events in SAX[11].*

2. *has an XML document as its fundamental unit of (logical) storage, just like a relational database has a row in a table as its fundamental unit*

---

[7] A content management system is an application designed to manage documents and built on top of a native XML database.

[8] See page 113 for more information about the XML:DB initiative

[9] See page 110 for more information about XPath

[10] Document Object Model, an XML parser

[11] Simple API for XML, an event-driven alternative to the memory-hungry DOM

of (logical) storage.

3. *is not required to have any particular underlying physical storage model.
   For example, it can be built on a relational, hierarchical, or object-
   oriented database, or use a proprietary storage format such as indexed,
   compressed files."* [Ini]

An XML-Enabled Database (XEDB) is a database that *"has an
added XML mapping layer provided either by the vendor of the database
or a third party. This mapping layer manages the storage and retrieval of
XML data. Data that is mapped into the database is mapped into applica-
tion specific formats and the original XML meta-data and structure may be
lost. Data manipulation may occur via either XML specific technologies (i.e.
XPath, XSLT*[12]*, DOM or SAX) or other database technologies(e.g. SQL).
The fundamental unit of storage in an XEDB is implementation dependent."*
[Ini]

A Hybrid XML Database (HXD) is a database that *can be treated
as either a Native XML Database or as an XML Enabled Database depend-
ing on the requirements of the application."* [Ini]

To summarize, Native XML databases are new custom-designed databases
built from the ground-up to manage XML and which allow XML documents
to be stored as XML internally. XML-enabled databases are conventional
relational or object-oriented databases that have been fitted with some kind
of front-end XML adaptor to manage the storage of data from XML docu-
ments. Hybrid XML databases can be treated as both.

A close-to-exhaustive list of **XML Database products** is available on
Ronald Bourret's Website [Bou03b]. Before diving into this list, one must
understand the difference between text-based and model-based DB types.
A text-based native XML database (TB) is one that stores XML as text
while a model-based native XML database (MB) builds an internal objects
model from the XML document and stores this model. Here is a summary
of Ronald Bourret's list.

---

[12]Extensible Stylesheet Language Transformations. See the W3C recommendation at
http://www.w3.org/TR/xslt

Table 6.1: Native XML Databases

Product	Developer	License	DB Type
4Suite	FourThought	Open Source	Object-Oriented
DBDOM	K. Ari Krupnikov	Open Source	Relational
eXist	W. Meier	Open Source	Relational
GoXML DB	XML Global	Commercial	Proprietary (TB)
Ipedo XML DB	Ipedo	Commercial	Proprietary
MindSuite XDB	Wired Minds	Commercial	Object-Oriented
Natix	Data ex machina	Commercial	File System
Tamino	Software AG	Commercial	Proprietary/Relational
XDBM	Matthew Parry	Open Source	Proprietary (MB)
X-Hive/DB	X-Hive Corporation	Commercial	OO/Relational
Xindice	Apache Soft. Foundation	Open Source	Proprietary (MB)

Table 6.2: XML-Enabled Databases

Product	Developer	License	DB Type
Access 2002	Microsoft	Commercial	Relational
DB2	IBM	Commercial	Relational
FileMaker	FileMaker	Commercial	FileMaker
FoxPro	Microsoft	Commercial	Relational
Informix	IBM	Commercial	Relational
Objectivity/DB	Objectivity	Commercial	Object-Oriented
Oracle 8i, 9i	Oracle	Commercial	Relational
SQL Server 2000	Microsoft	Commercial	Relational
Sybase ASE 12.5	Sybase	Commercial	Relational

Table 6.3: Hybrid XML Database

Product	Developer	License	DB Type
Ozone	ozone-db.org	Open Source	Object-Oriented

An outstanding research summary paper, titled "XML Database Trends and Influences" has been written by the Intellor Group [IG01]. The reading of this study is strongly recommended for those willing to learn how Native XML Databases and XML-Enabled databases are gaining importance on

today's database market.

D) XML Databases Query Languages

Native XML Databases' efficiency tightly depends on several factors. Efficiency obviously depends on the chosen model, but also on how clients will access data, how clients will store, retrieve and update information. That is the database query language.

A multitude of standards have emerged around the XML concept. These standards are defined by the World Wide Web Consortium (W3C) and are followed by a considerable majority of the developers. Amongst these standards, there are two XML Query Languages: XPath and its successor XQuery.

XPath and XQuery quickly revealed themselves as insufficient. Both of them only enable retrieval of data in the XML database. Editing or updating the value of an element or attribute in an XML document is impossible. The only way around this is to work at the document level: delete the whole document and replace it by the new one.

An XML update language recently appeared on the market (September 2000) at the initiative of the XML:DB project. In March 2003, the W3C has shown its interest in supporting XUpdate and offered to develop a new standard based on the actual specifications.

The attentive reader probably wonders how come there is not only one clearly defined language for all data accesses as there is in the relational model (SQL). To work with an XML database, a set of two languages is needed; one to query and one to update. This is due to the youth of the XML standard and to the multiplicity of developers, each of them adding one brick to the XML wall.

The three of the above mentioned standards (XPath, XQuery, and XUpdate) will be briefly exposed in the following paragraphs.

XPath   XPath, the XML Path Language, has been defined by the World Wide Web Consortium (W3C)[13]. It aims at locating elements, attributes, and other XML document nodes in a concise, interoperable way. XPath uses a compact, string-based syntax, rather that a structural XML-element based syntax[14]. This allows XPath expressions to be used both in XML attributes and in URIs.

Without focusing on the details of the syntax, here are some basic examples of XPath queries. The XPath *//customer/order* selects every element named "order" within top-level elements named "customer" in the selected XML document. Similarly, *//customer/order[@orderID='981240']* returns the customer order whose attribute "orderID" is 981240, and *//customer[2]* selects the second "customer" element of the treated XML document.

As stated here before, the XPath language provides ways to select nodes in an XML document based on simple criteria such as structure, position, or content, but does definitely not permit any modification or update of these nodes.

XQuery   The XML Query language, also defined by the W3C[15], is a powerful and convenient language designed for processing XML data. As for XPath, the initial design of XQuery is focused only on information retrieval and does not provide features for updating existing XML documents.

XQuery is a functional language consisting of several types of expressions that can be composed with full generality. XQuery expressions include path expressions (XQuery is defined as a superset of XPath), element constructors, function calls, arithmetic and logical expressions, conditional expressions, quantified expressions, expressions on sequences, and expressions on types.

One can imagine an auction database from which one wants to extract a list of popular items. The query should generate an XML element, called "popular-item", containing the item number, a description, and a bid count

---

[13]See the W3C recommendation at http://www.w3.org/TR/xpath
[14]As XUpdate does
[15]See the W3C recommendation at http://www.w3.org/TR/xquery

for each item that has more than 10 bids. The example here below illustrates
how such a query would be expressed with XQuery.

The *for* clause and *let* clause produce a binding pair for each item in
items.xml. In each binding pair, the variable $i is bound to the item and
$b is bound to a sequence containing all the bids for that item. The *where*
clause retains only those binding tuples in which $b contains more than ten
bids. The *return* clause then generates an output element for each of these
bindings, containing the item number, description, and bid count.

```
for \$i in document(''items.xml'')/*/item
let \$b := document(''bids.xml'')
   /*/bid[itemno = \$i/itemno]
where count (\$b) > 10
return
    <popular-item>
      {
       \$i/itemno,
       \$i/description,
       <bid-count> {count (\$b)} </bid-count>
      }
    </popular-item>
```

More information about the XML Query language can be found in [Cha].

XUpdate   The XUpdate project[16] of the XML:DB initiative gave itself the
mission to provide open and flexible query and update facilities to modify
data in XML documents. The standard helps updating fragments of docu-
ments and avoids performing modifications at the document level.

XUpdate uses the expression language defined by XPath to query a doc-
ument. An update is rather expressed as a well-formed XML document.

Here is an example of an update operation. The following expression

```
<xupdate:update select="/addresses/address[2]/town">
    New York
```

---

[16]The project home page is http://www.xmldb.org/xupdate

```
</xupdate:update>
```

on this input document

```
<addresses>
  <address>
    <town>Los Angeles</town>
  </address>
  <address>
    <town>San Francisco</town>
  </address>
</addresses>
```

will change the content of the context node to

```
<addresses>
  <address>
    <town>Los Angeles</town>
  </address>
  <address>
    <town>New York</town>
  </address>
</addresses>
```

XUpdate can also insert, append, remove or rename elements/attributes/processing instructions/comments in an XML document.

E) The XML:DB Initiative

XML:DB[17] is an industry initiative formed by SMB GmbH, the dbXML Group L.L.C and the OpenHealth Care Group. XML:DB provides a community for collaborative development of specifications for XML databases and data manipulation technologies. They stimulate the use of standards in the XML database industry.

The XML:DB Initiative's long term goals can be summarized as:

- *"Development of technology specifications for managing the data in XML databases*

---

[17]http://www.xmldb.org

- *Contribution of reference implementations of those specifications under an Open Source License*

- *Formation of a community where XML database vendors and users can ask questions and exchange information to learn more about XML database technology and applications*

- *Evangelism of XML database products and technologies to raise the visibility of XML databases in the marketplace"* [Ini]

One of the XML:DB Initiative projects is the creation of an **XML database API**[18]. The project is intended to develop a unique programming interface for XML databases. This API is wanted to be vendor neutral in order to support the use with the largest array of databases possible.

The XML:DB API is designed to enable a common access mechanism to XML databases. It enables the construction of applications to *store, retrieve, modify and query data* that is stored in an XML database. These facilities are intended to ease the construction of applications built around any XML database that claims conformance with the XML:DB API.

Two Native XML databases from Ronald Bourret's list of products[19], Apache Xindice[20] and eXist, implement the XML:DB API. Ozone, a Hybrid XML database, does too.

6.3   Persistence in the Equipment Manager

6.3.1   Requirements

Business analysts and developers of the Equipment Manager formalized several fundamental requirements of the persistence module of the application.

The Equipment Manager obviously needed to store, update and retrieve data in the product database in a convenient way. The database is needed

---

[18] Application Programming Interface

[19] Ronald Bourret's list of products is available on page 109

[20] Pronounced the Italian way: zeen-dee-chay

as a "shared resource" between three applications and the number of applications might increase in the long run. Hence, a solution favouring transmission of data between applications was also needed.

On top of that, the database is required to be portable since it will hold the pool of products needed by sales representatives working on client sites.

On the financial level, executive officers strongly recommended the use of open-source tools in order to reduce license costs.

6.3.2 Technology selection

Two important decisions had to be taken to choose the most appropriate persistence system. The first one is the database model to use. The second one is the tool that is best adjusted to the specific requirements of the application.

The choice of the database model lead very naturally to XML Databases as XML suits well the needs of a "shared resource". As for the tool, the team decided to use a Native XML database rather than an XML-Enabled. The reason for this is the efficiency provided by a system that allows data/documents to be stored as XML internally, without additional format conversion layers.

After an evaluation in depth of the market, it appeared that most of the available Native XML databases were still very immature. Even those which are compliant to the XML:DB API, such as Apache Xindice or eXist, presented severe incompletions. For example, "eXist" from W. Meier does not offer any update mechanism. Neither XUpdate nor any other update mechanism is implemented yet. "Xindice" from the Apache Software Foundation provides more features and is, in general, a more complete tool than eXist. "Xindice" did not meet the expectations either. Among other elements, its system requirements were too restraining for Acme's software team (requires old version of the Java Development Kit: jdk 1.3). Unfortunately, no Native XML database found in the industry could answer the requirements.

Based upon these observations, the team decided to build its own custom-built Native XML database.

6.3.3   The implemented solution

The Equipment Manager database is a Native XML database based on a
file system with a specific naming convention[21]. The database is a simple
collection of XML files. Every file holds data for one product. These files
are data-centric documents. The collection is indexed in a separate XML
file. The index holds references to each product stored in the database.
It also holds all the information that is relevant at the database level. The
collection, with its index, is compressed in one single file using the zip format.

For reusability purposes, product images are stored separately from the
product data. Indeed, the database structure is constituted of two parts
or folders: product information and product images. The product data file
(from the product information folder) contains a reference to its image in
the database. This way, pictures can be shared and reused easily between
products. Figure 6.4 illustrates the internal structure of a typical database.

Files paths are all constructed the same way, according to well defined
naming conventions. A data file path is formed like */products/vendor/model/sku-
number.xml* and an image file path like */images/vendor/model/image.ext.*
Files within each part are sorted by vendors, then by models. The intention
is to have a unique path for every product. Since two vendors can give their
products the same SKU number or model name, the only possible unique
identifier has to be composed of the three following properties: the vendor
name, the model name, and the SKU number.

The XML files are serialized and unserialized with the help of the "Cas-
tor Source Generator"[22]. To be more precise, Castor generates serializer ob-
jects (Java source files) from the database definition (XML Schemas). The
generated serializer objects handle automatically the marshalling (serializa-
tion/unserialization) process. No query language, such as XPath, XQuery or
XUpdate is therefore needed since database accesses are executed through
get/set methods from the serializer objects.

---

[21]The reader remembers from its reading of Section 6.2.5 (page 107) that a Native XML
database is not required to have any particular underlying physical storage model.

[22]This is covered in detail in Chapter 5
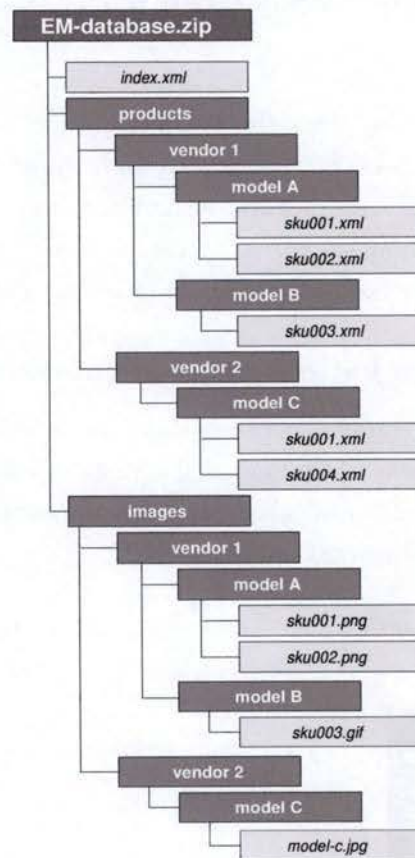
Figure 6.4: The structure of a typical Equipment Manager database

6.3.4 Putting in perspective

One must keep in mind that the exhibited choice does not pretend to be the one solution. Acme favored a database model that satisfies its needs for a light and portable "shared resource". This solution is not to be imperially selected for every software projet. As a matter of fact, larger-scale projects might need more efficient solutions regarding maintenance problems (index fragility), database definition versioning, transaction capabilities, multi-user access, cross-application access, and so on.

6.4 Persistence as a Design Pattern-oriented Subsystem

In a persistence subsystem, Design Patterns can play a substantial role. This section will check if "Design Pattern-oriented Subsystems", as introduced in Chapter 2, can improve the architecture of such a module. It also will demonstrate that several recurrent problems, specific to this kind of subsystems, can be fixed with the help of Design Patterns.

6.4.1 A Design Pattern-oriented Subsystem?

As illustrated in Figure 6.5, a "Design Pattern-oriented Subsystem" gathers three Design Patterns in one entity: the Model-View-Controller, the Mediator, and the Façade. Direct advantages of this composition are low coupling, high reusability, robustness and consistency.

DP SUBSYSTEM
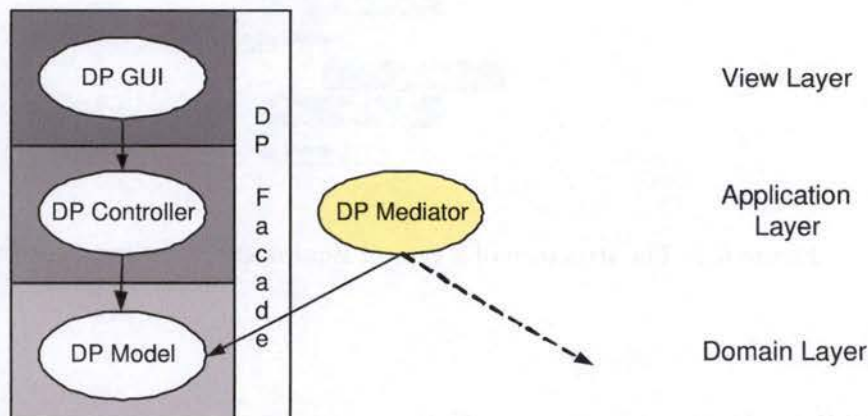
Figure 6.5: A Design Pattern-oriented Subsystem

Unlike other types of subsystems, as detailed in Chapters 3 through 5, it is delicate to create a persistence subsystem as a "Design Pattern-oriented Subsystem". First of all, a persistence module does not need any view layer because, very often, the application has a separated GUI subsystem. It does not require a Model (domain layer) either, since, for efficiency purposes, most databases are accessed directly by the controller.

With no View nor Model, there's no need to apply the Model-View-Controller pattern. And with one object only in the subsystem (i.e. the Controller), the presence of a Façade does not present much interest anymore. Similarly, the responsibility of the Mediator pattern is to communicate between Models. If there is no Model in the subsystem, there shouldn't be a Mediator either.

Towards the end of the chapter, one will progressively realize that the above paragraph is not entirely true. Although it is not recommended to apply as such the "Design Pattern-oriented Subsystem" to a persistence module, one will discover that exerting good architecture principles leads to a slightly similar solution.

A study of the contribution of specific Design Patterns to the architecture of a persistence subsystem will now follow.

6.4.2   Technology independence thanks to the Strategy pattern

Purpose   As mentioned amongst the requirements of a persistence module, it needs to be technology independent. As a matter of fact, the chosen technology can change at different levels. It goes from the logical storage model (hierarchical, relational, XML database, and so forth) to the DBMS tool (Oracle, DB2, Xindice, etc.).

Definition   This is where the Strategy pattern comes in. This pattern obviously is no solution in terms of storage model or DBMS tools. By definition, a Design Pattern is a high-level and generic solution to recurrent problems. The "Gang of Four" describes it as follow: *"The Strategy pattern defines a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it."* [GHJV95]

Using this pattern is very convenient when several related classes provide the same services, but differ in their behaviour. In other words, it is very handy when different variants of an algorithm coexist. Strategy is also useful

when an algorithm uses data that clients shouldn't know about. It avoids exposing complex, algorithm-specific data structures. The structure of the Strategy pattern can be sketched as shown in Figure 6.6.
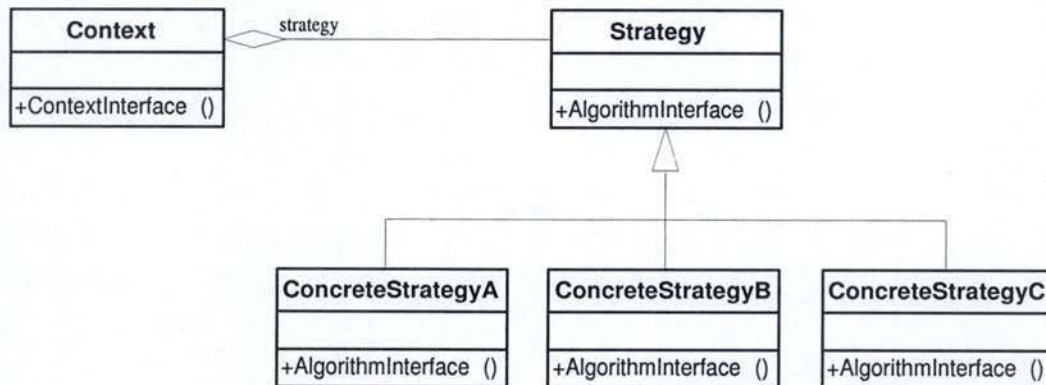
Figure 6.6: The Strategy Pattern

Figure 6.6 illustrates that a Strategy defines an interface common to all supported algorithms. The Context uses this interface to call the algorithm defined by a ConcreteStrategy. A ConcreteStrategy implements the algorithm using the Strategy interface.

The Context's role is to forward requests from its clients to its Strategy. The Context interacts with the Strategy to implement the chosen algorithm. A Context may pass all data required by the algorithm to the Strategy when the algorithm is called. Clients usually create and pass a ConcreteStrategy object to the context, then clients interact with the context exclusively. [GHJV95]

Application   In the case of a persistence subsystem, the function of the Strategy is played by an interface (DatabaseInterface) through which every database access goes. Concrete implementations of the interface may, for example, vary depending on the database model. They could, as well, reflect different space/time trade-offs. Figure 6.7 considers three ConcreteStrategies: a relational database controller, an object-oriented database controller, and an XML database controller.
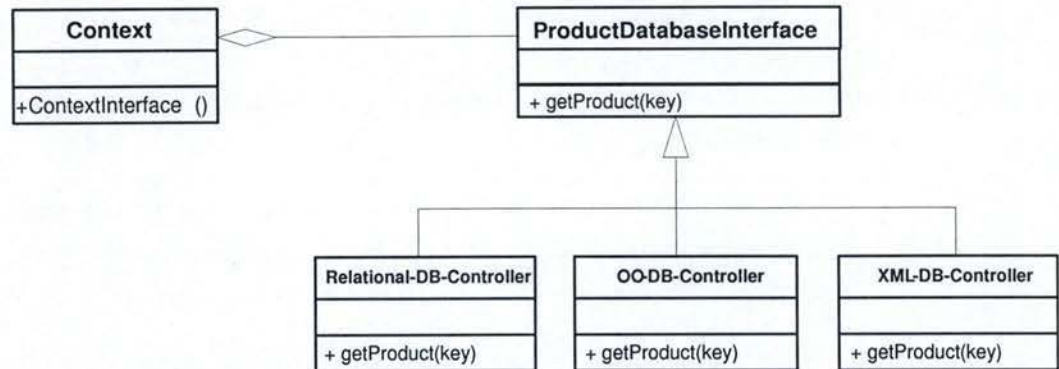
Figure 6.7: A technology independent persistence subsystem

Clients should use the subsystem as a service and should not know anything about the concrete implementation. They definitely should not be able to change database model from one request to another. Once selected, every client should keep up with a ConcreteStrategy, or totally migrate from one to another. One must note that the choice of the ConcreteStrategy can be improved with the Abstract Factory pattern. This is detailed in section 6.4.3.

The database interface should provide operations as generic as possible. A carefully specified interface does not reveal any details of its concrete implementations. For example, operations in the case of product database should look like:

```
public Object getProduct(ProductKey key);
public ProductKey setProduct(ProductKey key, Object product);
public void addProduct(ProductKey key);
public void removeProduct(ProductKey key);
```

Such methods signatures are generic enough because they do not make any assumptions on the implementation that lies beneath it. These signatures are high-level enough to be valid for all concrete strategies. Except for the generic product key interface, return types and parameter types are Objects. Method names also avoid using too specific names. Here are a few examples of technology dependent signatures:

```
public XMLResource getProductXMLFile(FilePath path);
```

```
public void setProductRow(PrimaryKey key, Row product);
```

Benefits   Using the Strategy pattern in a persistence subsystem has the following benefits[23]:

1. *Families of related algorithms.* The database controllers (Relational, Object-Oriented, XML, etc.) form a family of algorithms or behaviours for contexts to reuse.

2. *An alternative to subclassing.* Using inheritance to manage the database controllers infers subclassing a Context class directly. This hard-wires the behaviour into the Context and mixes the controllers implementation with Context's. Using the Strategy pattern instead makes it easier to understand, maintain, and extend. Plus, encapsulating controllers in separate Strategy classes makes it possible to vary controllers independently of their context, easing the switch of controller.

3. *No conditional statements.* As a matter of fact, it would be hard to avoid using nested conditional statements to select the right behaviour if the different algorithms were gathered in one same class. [GHJV95]

Drawbacks   A potential disadvantage of this pattern is that *clients must be aware of different Strategies*: database clients must know the difference between controllers, and must be able to choose one. [GHJV95]

6.4.3   Reduce coupling with the Abstract Factory pattern

Purpose   The use of the Strategy pattern leaves to clients of the persistence module the responsibility to choose which ConcreteStrategy should be used. In order to avoid coupling between clients and the ConcreteStrategy, enforcing the access to the Strategy through a Factory might be pertinent.

Definition   Erich Gamma et al define the Abstract Factory pattern as follow. The pattern intent is to *provide an interface for creating families of related or dependent objects without specifying their concrete classes.* [GHJV95]

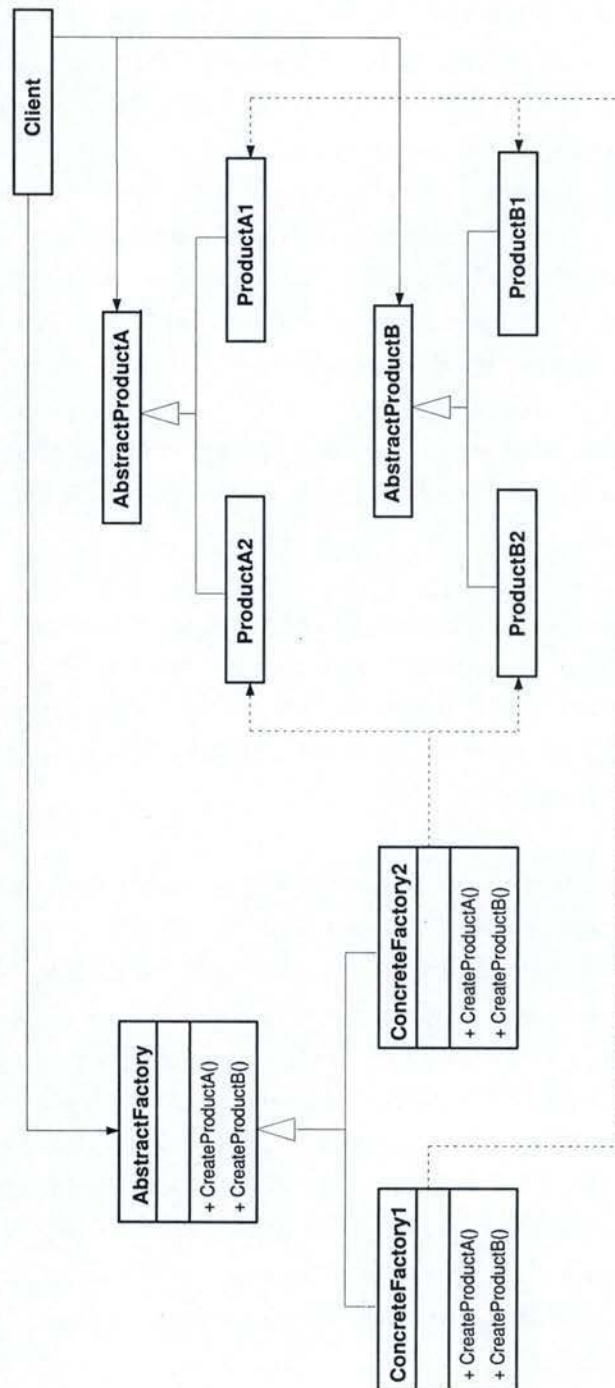[23]The following benefits have been adapted from benefits of the Strategy pattern as listed in [GHJV95].

Figure 6.8: The Abstract Factory Pattern

Figure 6.8 illustrates the UML class diagram of the Abstract Factory pattern. Clients only use[24] interfaces declared by AbstractFactory and AbstractProduct classes. The AbstractFactory declares an interface for operations that **create abstract products** objects. The ConcreteFactory classes implement[25] the operations to create concrete products objects. The AbstractProduct classes, instead, declare an interface for a **type of product** object. Finally, the ConcreteProduct classes define a product object to be created by the corresponding concrete factory. A ConcreteProduct implements the AbstractProduct interface.

The main reason to apply this pattern is the need for a system to be independent of how its products are created, composed, and represented. [GHJV95]

Application   The Abstract Factory pattern is designed to handle several product types, and several products for each type[26]. For the Equipment Manager, there is only one product type (ProductDatabaseInterface), which does contain several products (concrete database controllers). The applied pattern is illustrated in Figure 6.9.[27]

The attentive reader noticed that Figure 6.9 represents a combination of the Strategy pattern together with the Abstract Factory pattern. The set of four classes constituted by ProductDatabaseInterface and the three concrete database controllers plays the role of a set of participants in both the Strategy pattern and the Abstract Factory pattern. Put another way, the application of the Strategy pattern is constituted of ProductDatabaseInterface (the Abstract Strategy) and of Relational-DB-Controller, OO-DB-Controller and XML-DB-Controller (the Concrete Strategies). The Context object from the Strategy pattern is absent, since its role of intermediate between clients and Strategies is played by the Abstract Factory pattern. As for participants of the Abstract Factory pattern, ProductDatabaseInterface

---

[24]Drawn as plain lines

[25]Drawn as dotted lines (creation process)

[26]On Figure 6.8, these two dimensions are represented by letters for product types (A-B) and by numbers for products of each type (1-2).

[27]Note the arrow symbolizing the association of the client with ProductDatabaseInterface. Unlike one could interpret from the graphic, the relationship between these two classes can obviously exist only through the Factory. There is no difference here between the application and the original pattern, as defined by the Gang of Four.
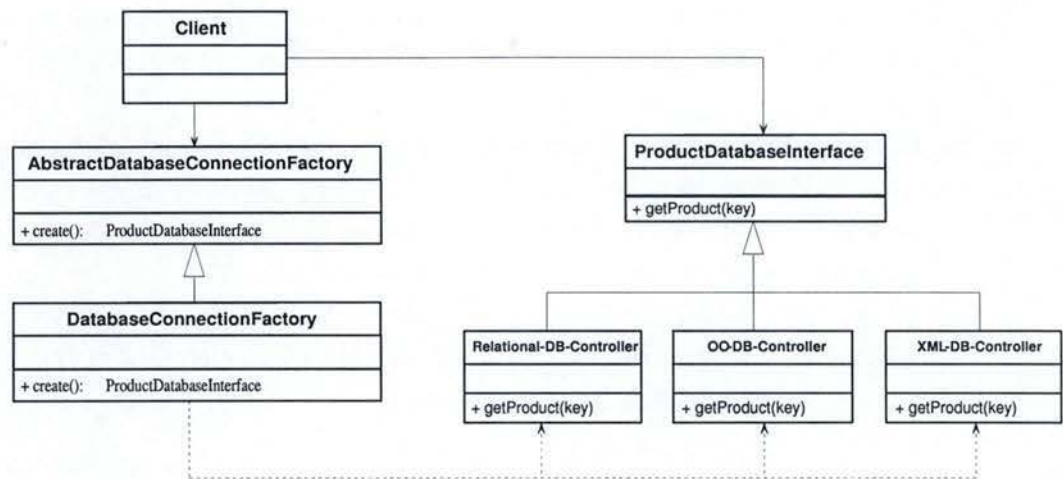
Figure 6.9: An application of the Abstract Factory Pattern

and the database controllers play respectively the role of the Abstract and Concrete products.

As example, the "create()" method of the DatabaseConnectionFactory is as simple as follows:

```
public static ProductDatabaseInterface create() {
  return new ZipXmlDbController(new XmlDbController());
}
```

The return type of the **Factory**'s create() method is a ProductDatabaseInterface (**Abstract Strategy**). What the method truly returns, is the **Concrete Strategy**.

The simultaneous use of two DatabaseControllers/ConcreteStrategies (ZipXmlDbController and XmlDbController) is an application of the Decorator pattern. The reader is referred to the following section about the Adapter pattern (page 130) for more explanations on this subject.

Benefits   Using the Abstract Factory pattern in a persistence subsystem *isolates database controllers*. Encapsulating the responsibility and the process of creating database controllers inside a factory isolates clients from

their implementation. It enforces clients to manipulate instances through their abstract interfaces. Therefore, it reduces coupling. [GHJV95]

Drawbacks *Supporting new database controllers is difficult.* Because the AbstractFactory interface fixes the set of products that can be created, supporting new database controllers involves changing the AbstractDatabaseConnectionFactory class and all of its subclasses. [GHJV95]

6.4.4 Decomposing database controller into logical sub-controllers with the Decorator pattern

Purpose Section 6.4.2 over the Strategy pattern reveals that a persistence subsystem might need to handle different database controllers in order to be technology independent. This may be pushed further by decomposing controllers in logical units.

Definition The Decorator pattern has been introduced and defined in Section 3.4. As a reminder, this pattern *"attaches additional responsibilities to an object dynamically. It provides a flexible alternative to subclassing for extending functionality"*. [GHJV95]
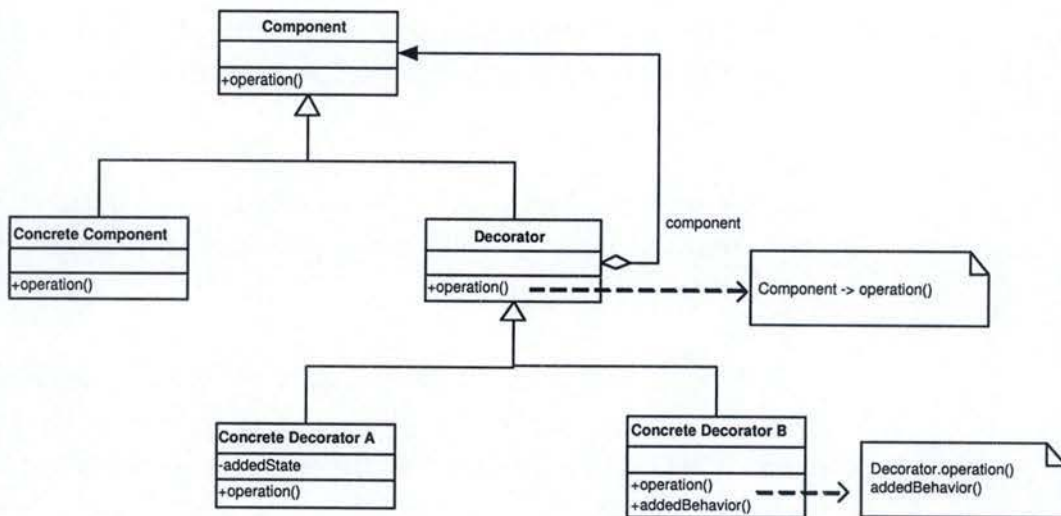
Figure 6.10: The Decorator Pattern

Figure 6.10 outlines a UML class diagram of the Decorator pattern. The

Component object defines the interface for objects that can have respon-
sibilities added to them dynamically. The ConcreteComponent defines an
object to which additional responsibilities can be attached. The Decorator
maintains a reference to a Component object. It also defines an interface
that conforms to Component's interface, so that its presence is transparent
to the component's clients. At last, the ConcreteDecorator adds responsi-
bilities to the component.

In other words, the Decorator merely forwards client's requests to the
component. It may also perform additional actions before or after forward-
ing.

The Decorator pattern is actually more powerful than that. It is pos-
sible to use nested Decorators. Each of them defines one layer of a global
component. The pattern acts as a wrapper[28] and is totally transparent to
the client. [GHJV95]

Application   The reader remembers from its reading of Section 6.3.3 that
the database of the Equipment Manager is an indexed collection of XML
files, compressed in one zip file. The development team chose this solution
for now, but this decision can be changed at any time.

To be fully flexible, it helps decomposing the database controller in sev-
eral parts. For example, the database controller of the Equipment Manager
is divided in two layers: one controller handles the database at the XML
files level (XmlDbController), and another one handles the upper level, the
zip level (ZipXmlDbController).

This way, it is possible (and even very easy) to decide to replace the
zip compression format by Gzip, or by any new revolutionary compression
algorithm. Similarly, the only cost of replacing the XML format would be
to replace the XmlDbController.

Figure 6.11 illustrates how the Decorator pattern is applied to the Equip-
ment Manager.

---

[28]The Decorator pattern is also known as the Wrapper.
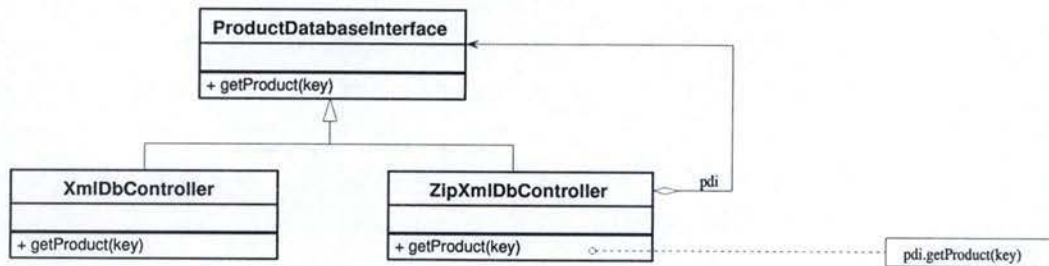
Figure 6.11: The Decorator Pattern applied to the Equipment Manager

Once again, all this is totally transparent to the client, since all controllers comply to the same interface (ProductDatabaseInterface). Clients access controllers through this interface. They use an instance of ZipXmlDbController, which forwards all requests to the XmlDbController (before or after completing operations that are specific to its layer).

For example, here follows the code of the getProduct(key) method from the ZipXmlDbController. The controller simply forwards the request without performing any additional operations.

```
public Object getProduct(ProductKey key) {
  return xmlDb.getProduct(key);
}
```

Unlike the previous example for which it does not make sense to perform any operations at the zip file level, the openDatabaseConnection(databasePath) method does perform zip file-specific operations before and after forwarding the request to the XmlDbController.

```
public void openDatabaseConnection(File databasePath) {
  if (isDatabaseOpen) {
    throw new RuntimeException("Database is already opened.");
  }
  zipDatabasePath = databasePath;
```

```
    databaseWorkingDirectory = getWorkingDirectory();
    deleteDatabaseWorkingDirectoryContents();
    try {
      ZipFileUtilities.extractZipFileToDirectory(zipDatabasePath,
                                                databaseWorkingDirectory);
    }
    catch(IllegalArgumentException iae) {
      iae.printStackTrace();
    }
    xmlDb.openDatabaseConnection(databaseWorkingDirectory);
    isDatabaseOpen = true;
}
```

Benefits   Using the Decorator pattern in a persistence subsystem has the following benefits:

1. *Reusability.* Decomposing database controllers into smaller components gives more reusable logical units. Switching logical units (for example, changing the compression algorithm) becomes easy.

2. *Maintainability.* Decomposing a problem in subproblems always makes it easier to understand, and hence to maintain.

3. *More flexibility than static inheritance.* [GHJV95] The Decorator pattern can dynamically attach responsibilities to a database controller, with much more flexibility than static (multiple) inheritance. Inheritance requires creating a new class for each additional responsibility. This gives rise to many classes and increases the complexity of a system. Furthermore, providing different Decorator classes for a specific Component class (database interface) enables the developer to mix and match responsibilities.

Drawbacks   *A Decorator and its component are not identical.* A Decorator acts as a transparent enclosure, but from an object identity point of view, a decorated component is not identical to the component itself. Therefore, it is not a good idea to rely on object identity when using Decorators. [GHJV95]

6.4.5   Handling incompatible interfaces with the Adapter pattern

Purpose   Sometimes, clients need to access an existing class through a generic interface the class can't comply to. The existing class has to conform to another interface or its implementation cannot be modified. It makes this class and the client incompatible.

Put another way, one may need to use an existing class, but the interface of this class does not match the needed one. This is precisely what the Adapter pattern has been thought for.

Definition   The Adapter pattern *"converts the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces"*. [GHJV95]
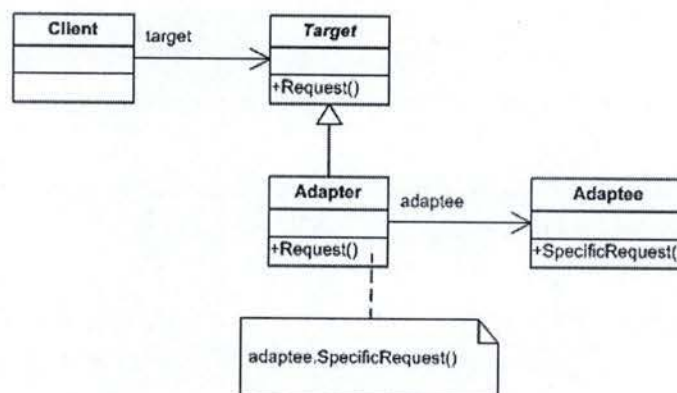
Figure 6.12: The Adapter Pattern

The object adapter[29] is illustrated in Figure 6.12. Client collaborates with objects conforming to the Target interface. The Target defines a domain-specific interface that the Client uses. The Adaptee defines an existing interface that needs adapting. The Adapter adapts the interface of Adaptee to the Target interface.

---

[29]Not to confound with the class adapter. (See [GHJV95] for more details)

Practically, clients call operations on an Adapter instance. The Adapter simply calls Adaptee operations that carry out the request. [GHJV95]

Application   The persistence module of the Equipment Manager uses "Castor Source Generator" to generate serializer objects (Java source files) from the database definition (XML Schemas). The purpose of these generated serializer objects is to handle automatically the database serialization/unserialization process. As these are automatically generated objects, it is not possible to modify them in order to comply to an interface.

Let's take the example of the database index. Based upon the XML Schema Definition (XSD) of a database index, Castor generates at compilation time the corresponding Java class. This class is the IndexSerializer. For robustness and consistency purposes, the persistence system should access the database index through an interface (Index). Ideally, the concrete implementation of Index would be the Castor generated serializer. Obviously, the serializer object can't implement the Index interface. An intermediate actor is required, the Adapter.
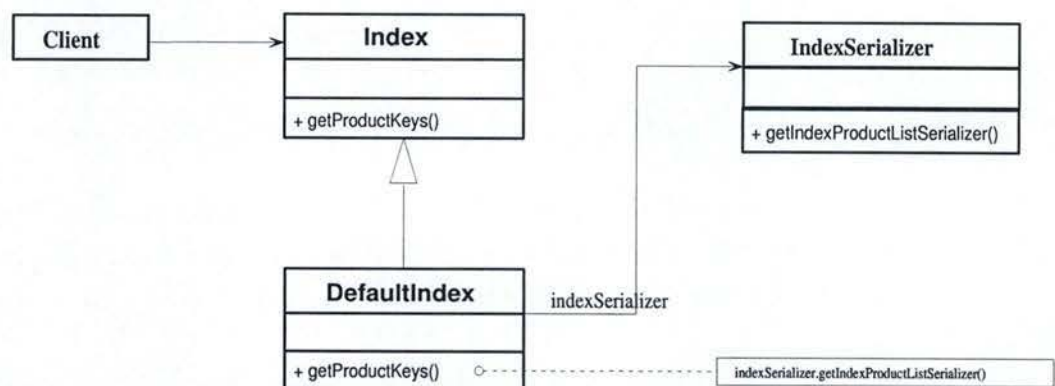
Figure 6.13: The Adapter pattern applied to the database index

On Figure 6.13, the database controller (Client) may access the table of contents of the database with the getProductKeys() operation. The controller holds an instance of DefaultIndex, the implementation of the interface Index. The controller addresses requests to DefaultIndex, which in turn calls the IndexSerializer operations that carry out the request.

Similarly, the Product Key suffers from the same problem. The database index holds a list of product keys, a kind of table of contents. When unserialized, each of these product keys constitutes an IndexProductSerializer instance. Since these are Castor generated objects, they cannot comply to the ProductKey interface through which database controllers access every key. Figure 6.14 shows how the Adapter pattern helps "changing" the interface of IndexProductSerializer. Here, IndexSerializerProductKey adapts ProductKey requests to IndexProductSerializer.
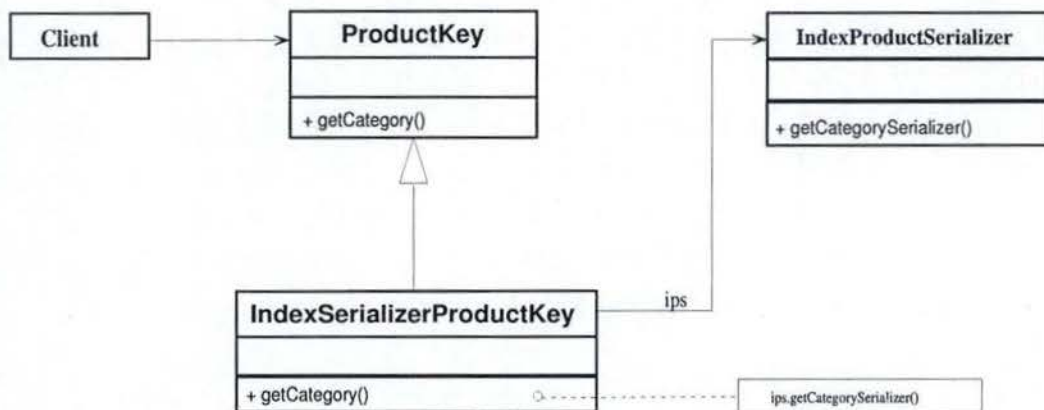
Figure 6.14: The Adapter pattern applied to product keys

Albeit the list of examples is stopped here, the reader understood that the Adapter pattern can be applied to a long list of objects of the Equipment Manager: to every database object managed by Castor Source Generator.

Benefits   The Adapter pattern's most certain benefit is that it *allows objects with incompatible interfaces to communicate.*

Drawbacks   In the case of the Equipment Manager, the potential number of generated classes that would require an adapter is quite important. Sometimes, the *number of potential adaptees* is so big that the development team may decide that applying the Adapter pattern and other architecture principles implies too much fastidious work. They could, in a more simpler way, skip the use of an interface and make database controllers directly work

on generated objects. This obviously would father much coupling.

6.4.6   A Design Pattern-oriented Subsystem? (2)

The idea behind Section 6.4.1 is that it is not efficient to enforce the "Design Pattern-oriented Subsystem" structure as such to a persistence module. Nevertheless, striving for exertion of good architecture principles, such as the application of specific Design Patterns to a persistence module showed that the final solution is quite close to the concept of "Design Pattern-oriented Subsystems".

Indeed, high reusability was compelled thanks to both the Strategy and Decorator patterns, low coupling thanks to the Abstract Factory and Adapter patterns.

On top of that, the assertion stating that no Model[30] is pertinent in a persistence subsystem must be relativized. We could envision intermediaries between database and controllers (the set of Castor generated serializer objects as for the Equipment Manager, a cache system, etc.) as the domain layer of the module, and somehow as the Model. This makes sense for two reasons. The intermediate objects hold the knowledge of the module, as it is the role of the domain layer. The other reason being that they directly interact with the Controller, just like a Model would.

Additionally, the Abstract Factory pattern (and its combination with the Strategy pattern) is the only way clients may access and use the subsystem. It provides a unified interface to the subsystem, making it easier to use, just like a Façade would.

Unlike the Model and the Façade, implementing a Mediator with a persistence module is, as a general rule, more of a challenge. Put another way, encapsulating the interaction between domain layers of several subsystems is hard to implement with a persistence module. This is because databases (when accessed directly by a controller) or intermediate objects (i.e. generated serializer) are no "flexible and adaptable" objects. As a matter of fact, they cannot be modified in order to apply the Mediator pattern on them:

---

[30]In the sense of the Model-View-Controller pattern

no operations for the interaction with the Mediator (events generation and so on) can be implemented.

It seemed at the beginning of this chapter that a "Design Pattern-oriented Subsystem" would not fit best in a persistence subsystem. Albeit this, it is now proved that enforcing high reusability, low coupling and other key architecture principles favours, with few exceptions, a solution that is quite close to the concept introduced in Chapter 2.

6.5   Summary

This chapter first exposes the purpose of persistence subsystems and positions the different persistence paradigms: from file systems to XML databases, through hierarchical, relational, and Object-Oriented databases. As a key technology in the development of the Equipment Manager and as the new way-to-go for the future, the "XML databases" topic has been covered in depth, introducing different types of XML documents, types of XML databases, and specific query languages.

Subsequently, the chapter looks into how persistence was managed in the case of the Equipment Manager. After revealing requirements, the chapter discloses the decision process that lead Acme's technology study to its current choice: a Native XML database based on a file system with a specific naming convention. The process of XML Data Binding is managed by Castor Source Generator.

Finally, this chapter demonstrates the benefits Design Patterns can have on a persistence subsystem. The chapter started by wondering if the concept of "Design Pattern-oriented Subsystem", could fit in or improve the architecture of a persistence module. At first sight, it seemed that it was more appropriate for other kinds of modules than for persistence. After focusing on specific Design Patterns (Strategy - Abstract Factory - Decorator - Adapter), deductions were made that, even though a "Design Pattern-oriented Subsystem" was not the most appropriate solution for a persistence subsystem, the global solution provided by each of these four patterns separately is slightly similar and hence, answers part of the needs.

Chapter 7

Design Patterns Automation

This chapter's intention is to provide a critical analysis about Design Patterns. It will focus on three distinctive topics.

First of all, the chapter leans on the **use of Design Patterns in CASE tools**[1]. This joins into one the potential expectations that a user may have from the software, a study of what exists on the CASE tools marketplace, and a review of the existing (comparison existing-expectations).

Secondly, the chapter puts in perspective the concept of "ready-to-use" **Design Pattern-oriented Subsystems** introduced in Chapter 2.

Finally, a thought on the **pertinence of Design Patterns automation** with the help of CASE tools will conclude the chapter.

7.1 Preliminaries

The basic notions on which rely the following sections must be introduced first. Design Pattern-capable CASE tools may offer several features. Among them, two need to be differentiated.

The term **Design Patterns generation** refers to the creation from ground-up of a new pattern. The user selects the pattern that is to be produced. The CASE tool then assemble from scratch the objects constituting

---

[1] A CASE tool is a computer-based product aimed at supporting one or more software engineering activities within a software development process.

the pattern and creates associations between them.

The above concept is not to be confounded with **Design Patterns application**. This means selecting existing objects and transforming them into a structure that complies to the pattern requirements. It also builds the necessary associations between elements.

Now that the difference between these two concepts is assimilated, the requirements one could have from a Design Pattern-capable CASE tool will be examined.

7.2 Requirements for CASE tools support

This section presents the main requirements that might be requested from a tool dealing with Design Patterns automation. Applying patterns is not as easy as understanding what they are made for, as much as the situation in which patterns have to be applied changes every time. A tool pretending to manage patterns automation must hence meet some requirements and perform some basic functionalities.

7.2.1 Theoretical context

Patterns representation

Before examining the desired functionalities of a CASE tool, some precisions have to be made. [BR] distinguishes two distinct levels of needs regarding patterns representation.

The first one is about the **expression of patterns in languages**, especially the expression of the solution's structure given by the pattern itself.

The second level concerns the **representation of patterns as manipulable entities**. Different approaches (more or less complete) exist for this second level. Every single information given in the pattern's description (as well the intent as the solution, etc.) might be part of this representation of patterns as manipulable entities. Yet, very often, only some kind of information is present (generally about the structural design recommended by the

pattern). What knowledge is part of this pattern representation can depend on different design points of view and/or some efficiency restrictions.

Patterns instantiation

The solution brought by a pattern can be qualified of "abstract", as explained in Section 2.2. A subtle distinction can be made between application and instantiation. The term "application" covers the process, whatever it is, leading to the code for the pattern, in a concrete conception context, while "instantiation" corresponds to the obtainment process of an object from a class. Application has thus a more general meaning than instantiation.

1) Levels of instantiation   As for the expression itself, several levels of instantiation can be found. They are three of them. [BR]

The first one is the **meta-representation of the information constituting a pattern**. This is the level of a pattern's meta-model.

The second level is the **abstract representation of a pattern**. This is the level of the general model of a pattern, instantiating a meta-model.

Finally, the **concrete pattern's representation** is found. It concerns a concrete pattern model instantiating a meta-model and specializing a general model. [BR]

2) Types of tools   Knowing this, the patterns application handled in CASE tools can be of two types: either the tool offers an explicit pattern meta-model and in this case, the instantiation of this model consists of representing the wanted abstract and concrete patterns; either the tool contains an abstract patterns library that might be derived in order to obtain the program's skeleton to be completed, leading to the concrete patterns. [BR]

3) Strategies of instantiation   To finish with instantiation, according to [Mei96] and [BR], three ways of instantiating a concrete pattern can be enumerated.

a) Top - down approach

In this approach, the user typically chooses a pattern to apply, and receives in return the pattern's structure that he/she has to adapt to his concrete conception context.

b) Bottom - up approach

This approach goes in the opposite direction than the top - down approach. The known components of the user's conception have to be linked to the pattern components. In this case, a pattern has been recognized a priori in an components assembly.

c) Mixed approach

The difference of this approach, compared to the bottom - up one, lies in the fact that the component structure given to the program only partially reflects a pattern description. The system completes the structure on its own with some of the pattern missing components.

The first approach meets what has been defined as Design Pattern generation and the two others cover the Design Pattern application (cf. Section 7.1).

The study below will be focused on tools from the second type, (i.e. tools presenting a library of abstract patterns - see patterns instantiation - in opposition with tools giving an explicit pattern's meta-model). Tools from the second type have been chosen because they are accessible to the very beginner user. It may be difficult to handle a pattern meta-representation without having any knowledge about patterns.

7.2.2   Help to conception

The first functionality expected from a tool is to provide some help in the conception. Obviously it must be possible to generate patterns "from scratch". This means that the user can decide to create a pattern without giving the program anything but the name of the pattern he/she wants to create. **Patterns generation** has been defined in Section 7.1.

On the other hand, existing classes can be transformed to play a role in a pattern. This is what is called **Design Patterns application** (cf. Section 7.1). It must obviously be possible to chose which class or object will play which role.

Some care must be given to this point. Applying patterns to exiting classes is intended to improve their structure (bringing the advantages of the Design Patterns) but cannot change functionalities! Enhancing design and structure does not mean the same thing as modifying the behaviour. This exigency is called "**behaviour preservation**" [Pop01]. A tool that would not perform such operations could not even be considered as a Design Patterns automation tool!

7.2.3   Generation of code and documentation

If the tool brings some help to the conception, it must also give some output to the user. And as exposed in [Pop01], this output is supposed to be from a **high level of abstraction**: *"the methodology should aim at the design level rather than the implementation level, since most of the problems of an object-oriented system which can be solved by reorganization concern the design of that system."*

So, what results can a user expect from the tool when he/she wishes to generate a Design Pattern? At least **a graphical and a textual representation** that, for the sake of argument in this case, will be the corresponding class diagram[2] and code. This point will be studied further in Section 7.2.9. The given diagram has of course to be readable, correct, complete and as simple as possible, without being too much simplified or over-simple. The class diagram reflects the classes that are part of the pattern, the relations they have and the operations they provide. In a way, it is the negative of all the actors of a Design Pattern. So it has to be clear and reusable by the user, otherwise he/she will not even be able to understand the transformation operated by the pattern. In addition, the code itself is also supposed to be well-written. If the generated code is incorrect, the whole operation becomes useless. The user might thus demand a readable, complete, cor-

---

[2]See Appendix A for more details on UML

rect, commented and simple code. This code will obviously be twisted by the user: it cannot be totally finished. The tool provides the code related to the Design Pattern, it cannot know about the real situation it is applied in.

Moreover, the tool has to be **language independent**: the results it gives cannot contain some tricks specific to a programming language. The tool cannot be tied up to a unique and particular language. Besides, in order to be as portable as possible, the tool should provide several target programming languages (as Java, C, C++, C# or others).

7.2.4 User-friendliness and ease of use

It is well-known that, the more a program is user-friendly, easy to use and fast, the more the user will tend to use it. And the more he/she uses it, the more he/she will get used to it and he/she will want to master the tool. This point is valid for all tools, in general. Although it concerns a sensitive topic (Design Patterns), the most common requirements have to be met too. Using the tool must then be intuitive and easy.

7.2.5 Wide but structured patterns library

Supporting only a few Design Patterns would not make any sense. If the user does not find a large range of patterns to use, he/she will not use the tool at all. That is why it should not be restricted to some patterns, ignoring others. There are a lot of Design Patterns; those known as the GoF Design Patterns [GHJV95] far from being the only ones. It goes further, Design Patterns are not limited to object-oriented programming, as mentioned in Section 2.2. The more patterns the program is able to handle, the more powerful it will be. There must be a whole library of patterns.

All Design Patterns do not have the same purpose, otherwise there would not be so many of them. Only in the small set of the GoF patterns, some subdivisions can be made. There are two criteria to classify these Patterns. [GHJV95] The first one is the **purpose of the pattern**. Patterns can belong to creational, structural or behavioural patterns. *"Creational patterns*

*concern the process of object creation. Structural patterns deal with the com-
position of classes or objects. Behavioural patterns characterize the ways in
which classes or objects interact and distribute responsibility."* [GHJV95]

The second criterion - the **scope of the pattern** - specifies whether the
pattern is to be applied to classes or objects. The distinction is therefore
made between a class pattern and an object pattern. Class patterns handle
relationships between classes and their subclasses. These relationships are
established through inheritance, so they are totally static and fixed. On
the other hand, object patterns deal with object relationships, which are
more dynamic. It could be inferred that the Mediator, for example, is a
behavioural object pattern (as the Observer pattern), whereas the Façade is
part of the structural object patterns. Knowing all the possible distinctions
between patterns, it might be necessary to present them to the user follow-
ing their category, hence giving him a first idea of the real aim of the pattern.

Some other diagrams can help to understand the pattern: a sequence
diagram (for example) might be useful.

7.2.6 Support to decision process

It is however not sufficient to present all the available Design Patterns by
categories if the user is not given any further information. Each pattern
available must in all circumstances be introduced by a **presentation** (in-
cluding its purpose, participants, consequences, etc.) which can constitute
the first real guidance in the user's decision process.

Nevertheless, despite the patterns classification and presentation, choos-
ing the "right" pattern is still not a sinecure. There are so many of them
that sometimes some patterns seem to do the same task. For this reason,
it is suggested to work out a kind of **wizard** which could lead the way for
the beginner user. The experienced user must of course have the possibility
to skip it. This wizard could ask the user which kind of solutions he/she is
looking for and suggests him some patterns close to what his requirements.
It would operate on a step-by-step basis through questions. But this wizard
obviously can't replace the user's thinking...

7.2.7   Patterns composition

Design Patterns might often need to be combined between one another. A single look to a "Design Pattern-oriented subsystem" proves it. In this kind of subsystem, the Model is part of the Model-View-Controller (application of the Observer pattern) and plays the role of the subject that the observer is listening to. At the same time, the Model plays the role of one of the Mediator's colleagues. It is so implied as well in the Observer pattern as in the Mediator pattern. This is certainly not the only case. If, in reality, Design Patterns can be combined, it be a loss if the tool was not offering possibilities for pattern composition. Design Patterns cannot be grouped in any way. The previously envisaged wizard could ask the user whether he/she is sure about what he/she is asking in case of suspicious operation. For example, an object having the role in a Façade, a Model, a Mediator and a Controller would not make any sense, or at least at first sight. The wizard might prevent the beginner user to perform incoherent actions.

If it is feasible to combine several Design Patterns, it becomes possible to **generate a whole "Design Pattern-oriented subsystem"**, which is an arrangement of a Façade, a Mediator, and an Observer. Generating the whole structure of an entire "Design Pattern-oriented subsystem" by a simple click would be great. And there is actually no reason to restrict it to the only "Design Pattern-oriented subsystems". There must be capabilities to extend the tool. User-defined plug-ins should be easily added; this way the user will be allowed to generate every possible (combination of) patterns or subsystem. This point will be placed into perspective further, in Section 7.6. In any case, the patterns library should be highly extendible.

7.2.8   Consistency checking

As said in Section 2.2, the solution given in a Design Pattern is an abstract design, meaning that it has to be adapted to the real context. If the solution is very often easy to understand, its application might be proving more difficult. A novice user might therefore have trouble to apply a pattern the correct way.

A functionality of consistency verification might hence be very useful: its purpose will be to validate models respecting structures given by patterns

solutions. The user would give the tool his own pattern application, expecting that the tool validates it, meaning that the proposed design respects the pattern's structure and general ideas. In order to illustrate what coherence's verification should be, a simple example follows. Figure 7.1 represents the input of a user who wants to validate his pattern's application. So, could this class diagram (see Figure 7.1) be an application of the Decorator pattern? (See Figure 7.2 for the solution suggested by the Decorator pattern.) It is obvious that this cannot be an application of the Decorator, since the presumed Component and Decorator do not even respect the same interface! The user's input should thus not be validated and the user should be explained why.
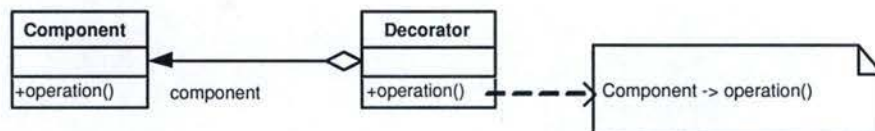
Figure 7.1: Example of consistency verification

This is only possible as long as the tool itself recognizes the pattern application. As exposed in [BG02], one of the biggest flaws of Design Patterns is the **poor traceability**. *"In a large scale application, several Design Patterns can be mixed and can even overlap each other. Different implementations of the same pattern can also coexist as they are each adapted to a particular context. Therefore, in the final design, it is really difficult to see which patterns are involved. Two different programmers could even arrive to two different sets of patterns when trying to identify them. Except perhaps for the documentation or comments scattered throughout the code, the patterns are lost during implementation."* [BG02]

Let us take a small example using the Decorator pattern. Figure 7.2 recalls the participants of the Decorator pattern and their relations, and Figure 7.3 presents an application of the Decorator (as exposed in Section 3.4). Even if it is about only one pattern, it does not leap to the eyes that this diagram represents the Decorator pattern in its application, even with the Decorator class diagram before one's eyes! And it gets even worst when applying several patterns.
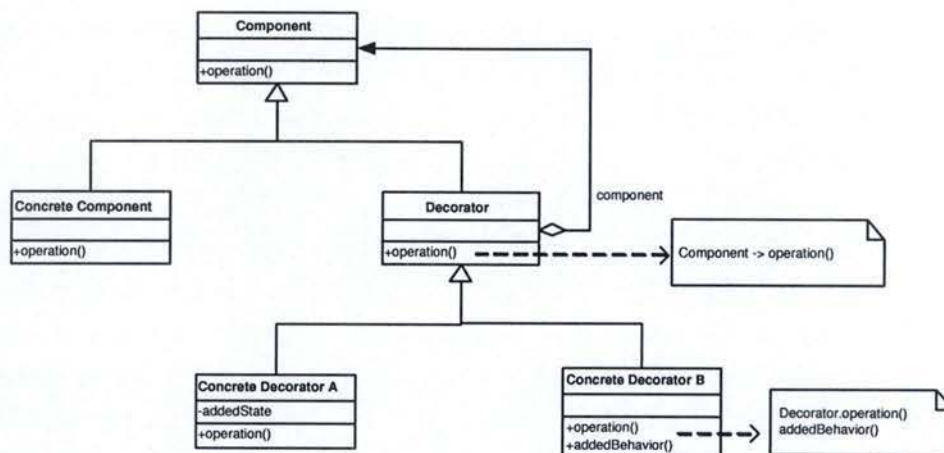
Figure 7.2: The Decorator pattern: class diagram (recall)
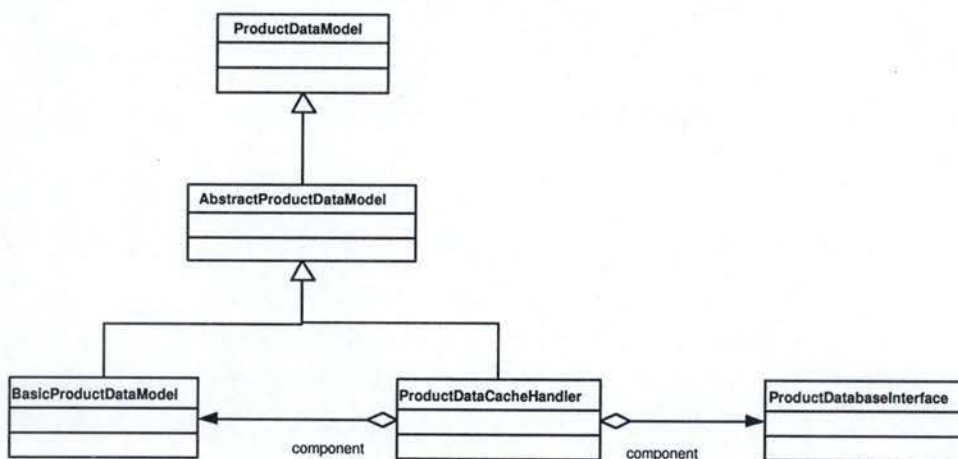
Figure 7.3: Example of pattern low traceability

Thus, in order to realize this operation the tool should dispose of a way to clearly identify each pattern application. This point will be covered more in depth in Section 7.2.9.

7.2.9   Traceable graphical and textual representations

In Section 7.2.3 the need to have some graphical and textual results of the application of the pattern was introduced. The first proposal was to output

a class diagram and the corresponding code. The class diagram gives a good overview of the participant classes, their relations and operations. As for the code, it is necessary given that the user will have to complete it. This documentation would be sufficient if Design Patterns were not suffering from poor traceability.

That is why this kind of documentation is necessary, but not sufficient. In addition to the code and the class diagram, the tool should provide another pattern representation. This third representation should clearly identify which pattern has been used, and which classes or objects are playing what roles in the pattern, thus reducing the lack of traceability of Design Patterns. It should be evident to see, as well for the user, as for the toll itself which pattern(s) has/have been applied. Each class or object playing a role in a pattern should, for example, be linked in some way to a small box representing the pattern application.

7.2.10   Portability

The tool providing Design Patterns support might not offer the developer a full programming environment. This means that the whole application development process is not necessarily done with the same tool. If, for example, the architecture conception is supported by another tool than the Design Patterns tool, it might be difficult to use both of these tools. Schemas from the first one need to be opened and modified by the second one and to re-opened by the first one again. As far as the format used by the tools are compatible, it seems possible, but it is very rarely the case.

Either the tool can be used through the whole development process (but this forces the developer to use only one defined tool) either formats are standard and compatible with other tools (but this supposes the existence of standard formats and agreements). It might be a challenge to meet this requirement.

7.3   Study of the existing

The purpose of this section is to study Design Pattern-related features of
CASE tools. Table 7.1, borrowed from [Des], lists the Design Pattern-
capable CASE tools available on the marketplace.

Table 7.1: Design Pattern-capable CASE tools

Product	Company	Platform
ModelMaker	ModelMaker Tools	Delphi
Describe	Embarcadero	Windows
Rational XDE Professional	Rational	Windows
Together ControlCenter	Borland	Java VM
ObjectiF	MicroTOOL	Windows
Objecteering Enterprise Edition	Softeam	Windows, Unix

To avoid redundancy and go straight to the point, the focus will be
placed on one tool only. Decision was taken to inspect the most complete
tool on the marketplace. According to Paul Pop [Pop01], one of the more
mature tools that offer support for Design Patterns is "**Together Control-
Center**", a software by Borland[3]. Together is a CASE tool which supports
several programming languagues such as Java, C++, C#, CORBA IDL, Vi-
sual Basic, and Visual Basic .NET. It also provides support for common
software design tasks.

Figure 7.4 shows a screenshot of Together ControlCenter 6.1. The mod-
eling tool's GUI is divided in three main panes. The Explorer Pane (left)
illustrates the "Model"[4] of the project. It lists the objects of the current
project and the operations each object provides. The Designer Pane (upper
right) holds modelization diagrams. These may be UML diagrams or any
other types of modelization diagrams, such as XML Structure Diagrams,
or even Entity-Relationship Diagrams. This section concentrates on UML
class diagrams, since Design Pattern-related features in Together only are

---

[3]A trial version of Together ControlCenter is available for download at http://www.
borland.com/products/downloads/download_together.html
[4]As called by Together ControlCenter; not in the sense of the Model-View-Controller
pattern
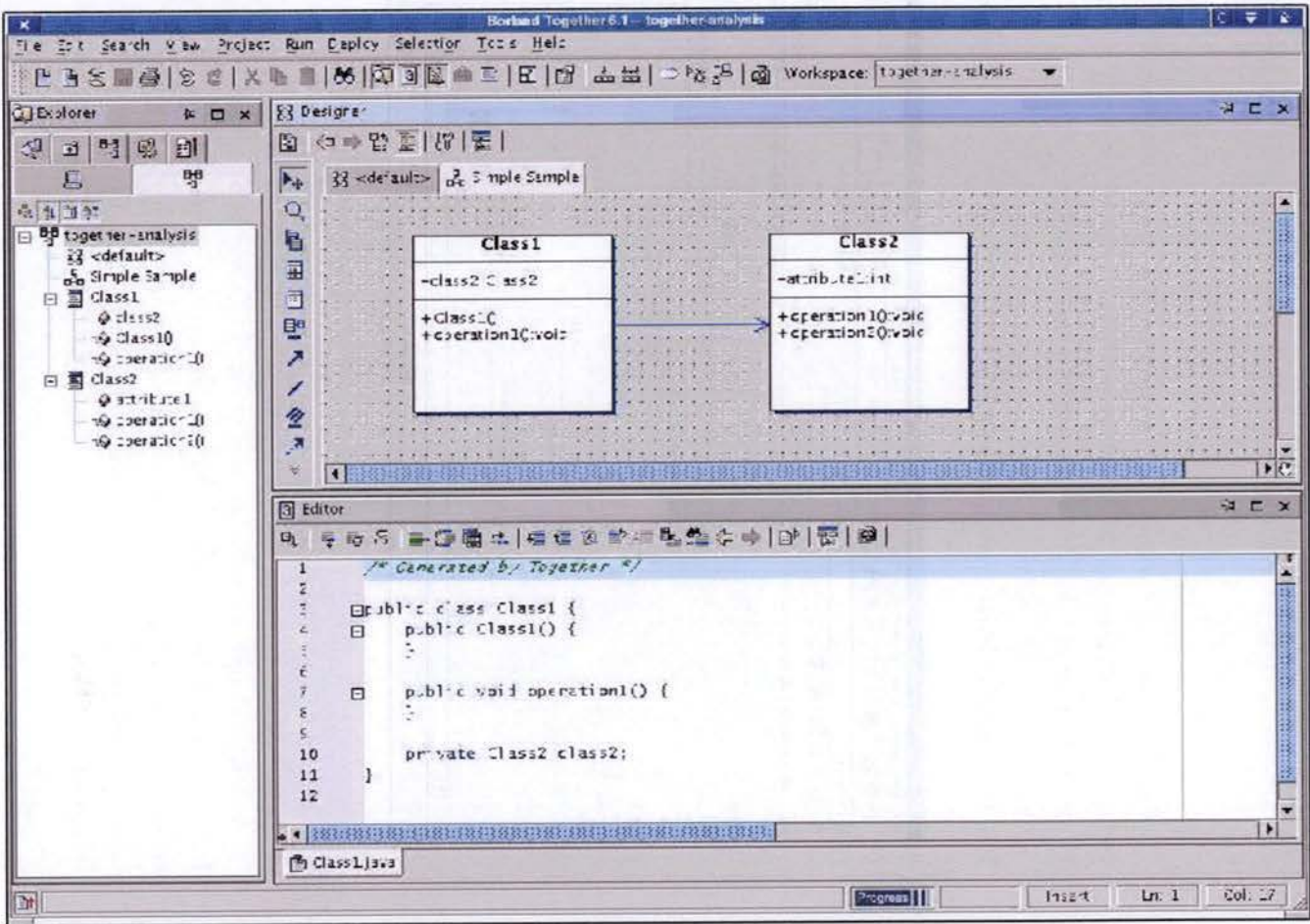
Figure 7.4: Together ControlCenter 6.1 by Borland

available in class diagram mode. Eventually, the Editor Pane (lower right)
displays source code always in sync with model diagrams from the Designer
Pane.

Together ControlCenter (TCC) offers several features that proved to be
useful when designing an architecture with the help of Design Patterns.
These operations are described here below.

7.3.1 Design Patterns generation
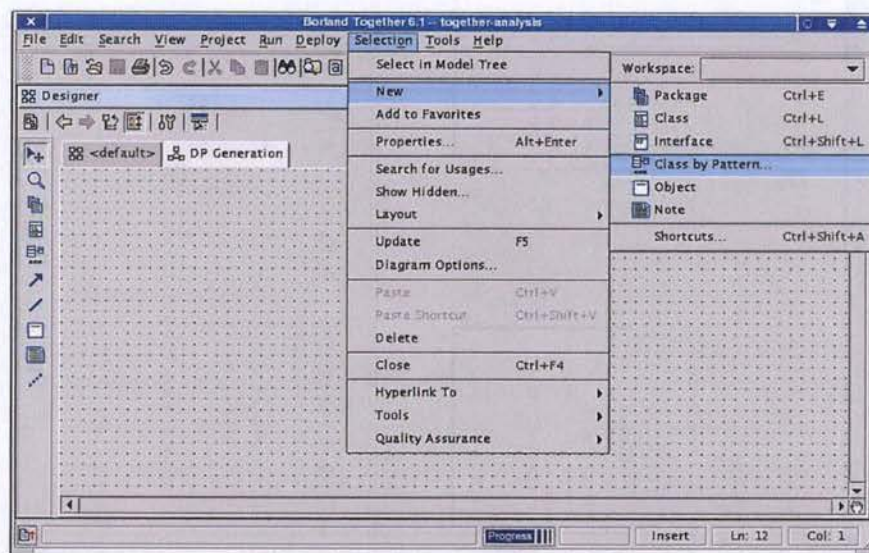
Figure 7.5: Creation of new classes by selecting a Design Pattern

When modeling in class diagram mode, TCC allows to generate Design
Patterns from scratch. Design Patterns generation corresponds to the first
strategy of patterns instantiation, the top-down approach[5]. The action "New
Class by Pattern" (Figure 7.5) opens a new dialog box, from which the user
chooses the Design Pattern to be generated.

Figure 7.6 shows the "Design Pattern selection" dialog box. It proposes
an imposing list of patterns, grouped by type. A "Gang of Four" folder
(GoF) lists 11 Design Patterns defined by Erich Gamma et al.: the Abstract

---

[5]Strategies of instantiation have been introduced in Section 7.2.1 (page 137)
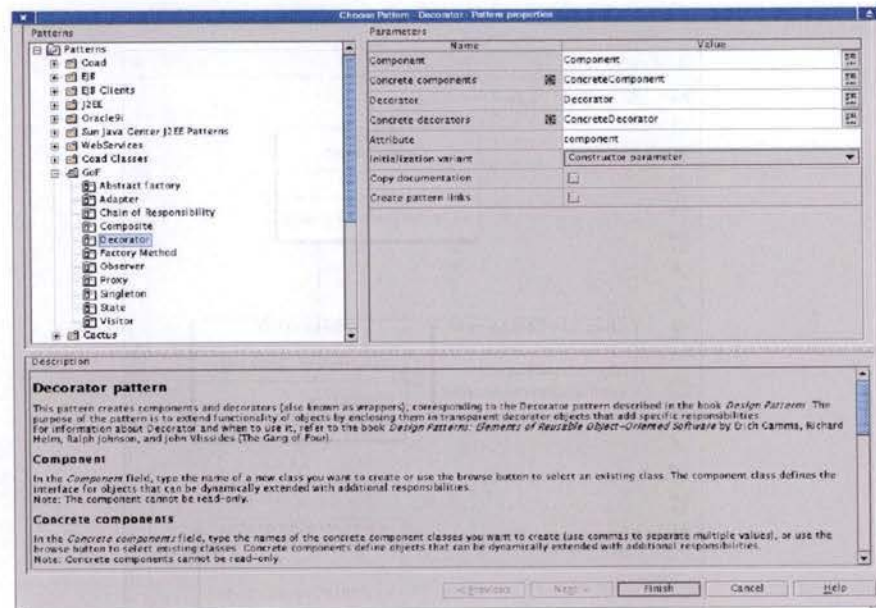
Figure 7.6: Design Pattern selection and configuration

Factory, the Adapter, the Chain of Responsibility, the Composite, the Decorator, the Factory method, the Observer, the Proxy, the Singleton, the State, and the Visitor. The selection of a pattern displays automatically a description of the pattern together with a list of parameters to be configured. The description gives a brief introduction to the pattern and to its participant objects. Understanding the role of each participant helps configuring the parameter list. As a matter of fact, parameters include every participant. Each one must be given a name[6]. A set of other pattern-specific options[7] also needs to be configured. At last, other properties like "Copy documentation" or "Create pattern links" can also be set. The "Copy documentation" option copies comments from methods in interfaces participating in the pattern to methods that the pattern created in classes implementing such interfaces. The reader is directed to section 7.3.4 over Together's traceability features for a complete overview of the concept of Pattern Links. Once all parameters are set to the desired value, the designer presses "Finish" which leads to the automatic generation of the pattern.

---

[6]Or assigned to an existing class in the case of Design Patterns application (Section 7.3.2)

[7]Attribute, Initialization variant, etc.
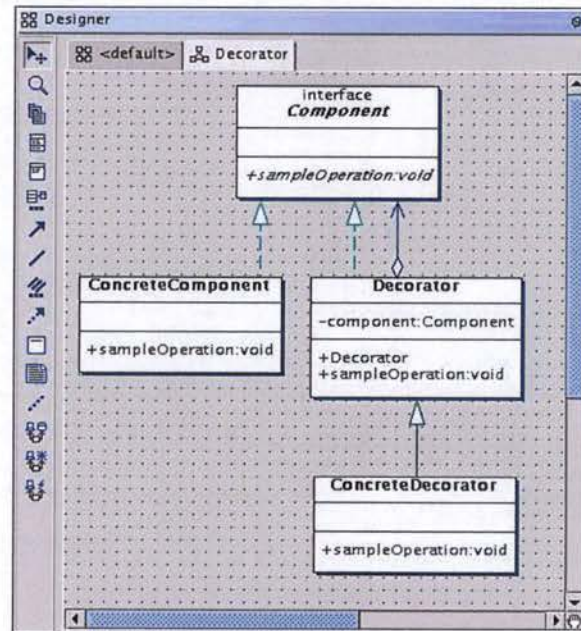
Figure 7.7: Generation of the Decorator pattern

The generated Decorator pattern, depicted in Figure 7.7, is composed of one interface and three classes. TCC creates the appropriate attributes and methods in every object. It also creates the necessary associations between classes.

The corresponding source code is embryonic but neat. Javadoc is included by default[8]. Here follows the generated code. The Component interface appears first, then come the ConcreteComponent class, the Decorator, and finally the ConcreteDecorator.

```
//-----------------------------------------------------------+
// Component
//-----------------------------------------------------------+
1   /* Generated by Together */
2
```

[8] Assuming that project language is Java

```
3  public interface Component {
4    void sampleOperation();
5  }
```

```
//----------------------------------------------------------+
// ConcreteComponent
//----------------------------------------------------------+
1  /* Generated by Together */
2
3  public class ConcreteComponent implements Component {
4    public void sampleOperation(){
5      // Write your code here
6    }
7
8    private int attribute1;
9  }
```

```
//----------------------------------------------------------+
// Decorator
//----------------------------------------------------------+
1  /* Generated by Together */
2
3  public class Decorator implements Component {
4    public Decorator(Component component) {
5      this.component = component;
6    }
7
8    public void sampleOperation(){
9      component.sampleOperation();
10   }
11
12   /**
13    * @link aggregation
14    */
```

```
15    private Component component;
16 }



//----------------------------------------------------------------+
// ConcreteDecorator
//----------------------------------------------------------------+
1   /* Generated by Together */
2
3   public class ConcreteDecorator extends Decorator {
4      public void sampleOperation(){
5         super.sampleOperation();
6      }
7   }
```

7.3.2   Design Patterns application

Together also offers to select existing classes from a UML class diagram and refactor them with a pattern. For that, the user must right-click on one object that needs to be part of the pattern and select the "Choose Pattern (Ctrl+R)" option (Figure 7.8).
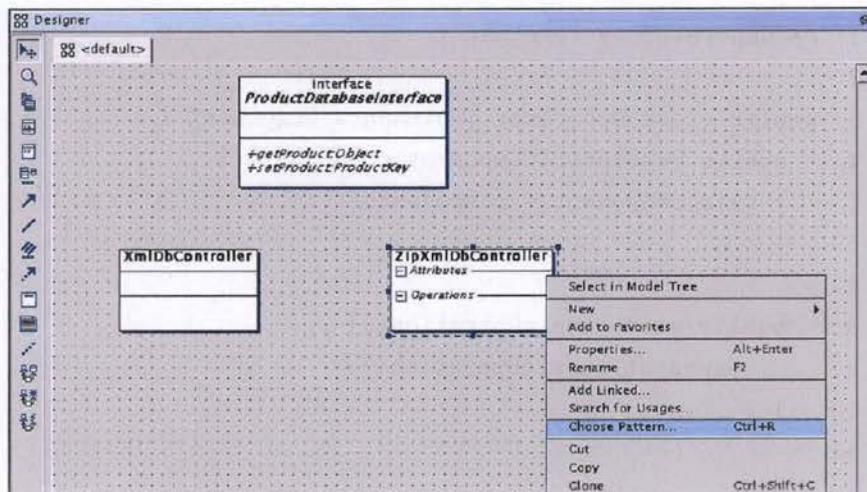
Figure 7.8: Application of a pattern to a set of existing classes

The user is prompted with the "Design Pattern selection/configuration" dialog box (Figure 7.9). The dialog box asks him to choose which role should be playing the selected object. This is done by means of the "Use selected class as" field. To each other participant, may be attributed a new class, or an existing class.
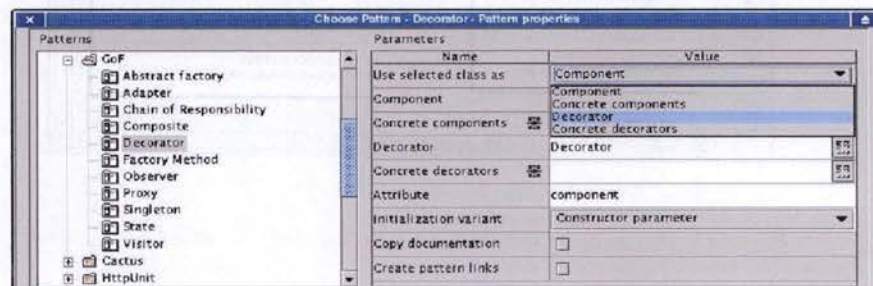
Figure 7.9: Design Pattern selection and configuration

To apply the Decorator pattern to the three classes represented in Figure 7.8, ProductDatabaseInterface must play the role of the Component interface and XmlDbController the role of a Concrete Component. ZipXmlDb-Controller is the Decorator. Once parameters are set, the refactoring results in the class diagram shown in Figure 7.10. Both XmlDbController and ZipXmlDbController implement the same interface (ProductDatabaseInterface), and the Decorator (ZipXmlDbController) maintains a reference to ProductDatabaseInterface.

It is worth showing the generated code for the Decorator (ZipXmlD-bController). The class holds a reference (aggregation) to the Component object. It also implements ProductDatabaseInterface's methods by forwarding calls to the Component object.

```
1  public class ZipXmlDbController implements ProductDatabaseInterface {
2    public Object getProduct(ProductKey key){
3      return component.getProduct(key);
4    }
5
```
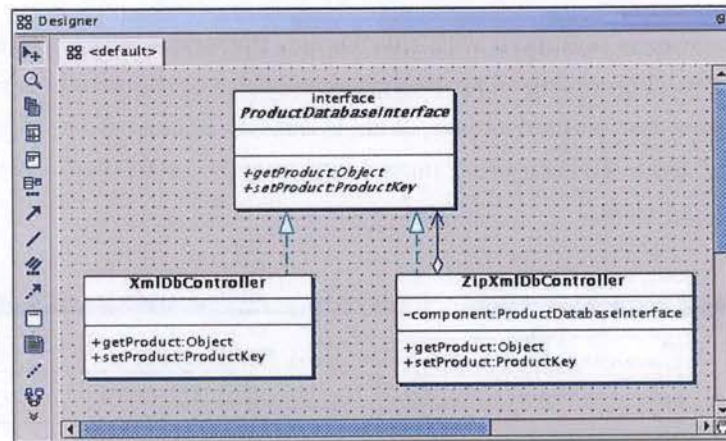
Figure 7.10: The Decorator pattern applied to existing classes

```
6     public ProductKey setProduct(ProductKey key, Object product){
7       return component.setProduct(key, product);
8     }
9
10    /**
11     * @link aggregation
12     */
13    private ProductDatabaseInterface component;
14 }
```

The here above example illustrates a **bottom-up strategy** of instantiation in the sense that every participant of the pattern already exists and that the process of Design Pattern application here consists in establishing links between participants. A **mixed strategy** example would be slightly similar. As a matter of fact, applying the Decorator pattern on only one or two of the three classes depicted in Figure 7.8 would make use of the mixed approach of pattern instantiation instead of the bottom-up approach.

7.3.3   Combination of Design Patterns

Combining two or more Design Patterns is another feature permitted by Together ControlCenter. This is merely due to the fact that TCC allows Design Patterns application. All *Design Patterns combination* means, is

Design Pattern application exerted on an existing pattern.
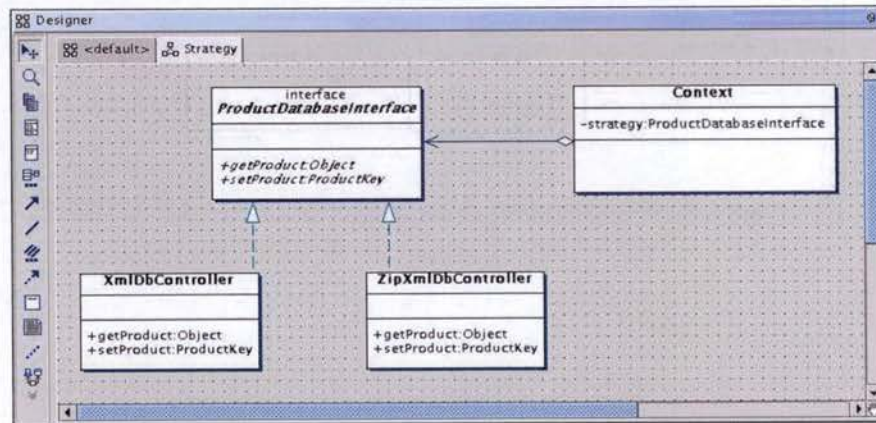
Figure 7.11: An application of the Strategy pattern

Figure 7.12 illustrates the combination of the Strategy pattern (Figure 7.11) with an Abstract Factory. The role of the AbstractProduct (from the Abstract Factory pattern) is played by ProductDatabaseInterface (the Strategy interface). Similarly, Concrete Products are the Concrete Strategies (XmlDbController and ZipXmlDbController). One must note that the Context object does not have its place anymore in such a structure. Its role to be a relay between clients and the ProductDatabaseInterface has been replaced by the Factory itself.

The combination action can merely be performed by right-clicking on any object of the first created pattern, then by selecting the "Choose Pattern (Ctrl+R)" option, and configuring the new pattern by assigning to its participants the appropriate roles.

Design Patterns combination may result in complex objects structures, and may be tough to understand. Additional links[9] have been drawn in Figure 7.12 to better illustrate collaborations between elements.

---

[9]These links are Pattern Links. Pattern Links are covered in section 7.3.4.
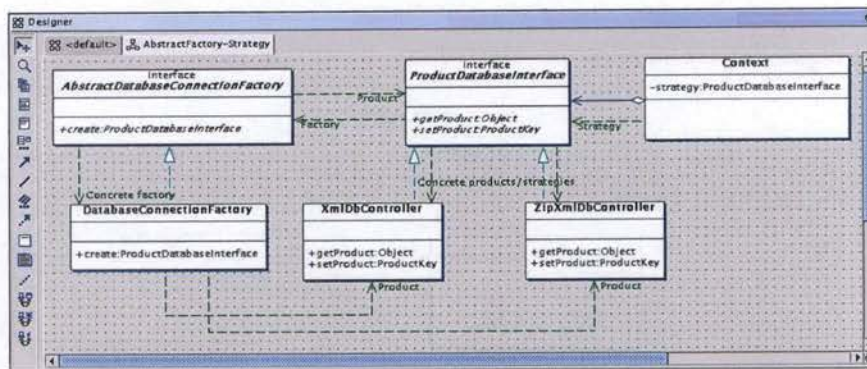
Figure 7.12: A combination of the Abstract Factory pattern with the Strategy pattern

7.3.4 Traceability features

To answer a quite common issue of Design Patterns, traceability, Together offers a feature called "Pattern Links". Pattern Links attempt to identify patterns in a set of classes, and identify participants in a pattern.

If the option is set when generating/applying a pattern (cf. Figure 7.6), TCC generates additional links that can be used by this pattern later to determine classes and interfaces participating in the pattern. This means that if the user checks this option and uses the pattern to create a set of classes and interfaces, the pattern invoked for some participant later (using the "Choose Pattern" command on the right-click menu) will automatically find all other participants (if possible) and fill in participant fields with their names.

Furthermore, if a user applies the pattern with this option checked and later invokes the pattern using the "Choose Pattern" command on the right-click menu for some participant, the additional field called "Use selected class as" contains possible roles only for the selected element.

This option is very useful when the user plans to change something in the classes/interfaces participating in the pattern. For example, if this option is on and after creating the classes and interfaces, the user adds several methods to a certain interface-participant (and this change must be reflected

somehow in other participants), all he/she needs to do is select this changed interface, invoke the "Choose Pattern" dialog for this element and select the original pattern. After that, the pattern determines other participants and the user only needs to click "Finish". The pattern will modify all other classes and interfaces according to changes.

At this point, the reader probably wonders how does Together represent Pattern Links. Pattern Links are defined in the source code of each participant of the pattern. More precisely, their definition reside in documentation comments of every object. A Pattern Link's definition is composed of two parts: the type of link, and the identification of the recipient object. The type of link is described by a set of five special tags. The "@link" tag indicates a link between the present object and the referred object (cf. identification of the recipient object). The "@shapeType" tag takes the value "PatternLink". An "@pattern" tag identifies to which pattern does the object participates. Eventually, "@clientRole" and "@supplierRole" give the role played, in this relationship, by the present object and by the recipient object, respectively. The identification of the recipient object is done through a commented out reference to the desired object. An example will be given here below to help visualizing how Pattern Links are defined textually.

Pattern Links also can be represented graphically between elements of a UML class diagram. They are illustrated by green dotted arrows between participants. Arrow labels indicate the roles played by participants.

In the case of the class diagram illustrated by Figure 7.12, the AbstractDatabaseConnectionFactory interface owns two links to other participants of the pattern: one with its abstract product, the other with its concrete factory. The source code of the AbstractDatabaseConnectionFactory interface holds the definition of these two links. They are detailed here below.

```
/**
 * @link
 * @shapeType PatternLink
 * @pattern AbstractFactory
 * @supplierRole Product
```

```
*/
/*# private ProductDatabaseInterface _productDatabaseInterface; */

/**
 * @link
 * @shapeType PatternLink
 * @pattern AbstractFactory
 * @supplierRole Concrete factory
 */
/*# private DatabaseConnectionFactory _databaseConnectionFactory; */
```

7.3.5   Extension capabilities

Together ControlCenter delivers a unique capability to externally extend its
native functionalities to other patterns. Two approaches coexist: pattern
templates and the pattern API. Their presentation gathers information from
[Bor03, Pop01].

Pattern templates   This approach expresses the pattern as a simple tem-
plate (i.e. an ASCII file). Unfortunately, no instance-specific customization
(and thus Design Patterns application) is possible with this technique, and
there is no completeness or consistency checking applied. These disadvan-
tages are addressed by the second approach, the pattern API.

The Pattern API   The pattern API approach expresses the pattern in
Java using a special API provided by Borland. Before using the API to
write the pattern, two things need to be determined. First of all, the level of
language dependence must be decided. The pattern can be written for only
a single language, or it can be a generic pattern which can be used with any
language. Secondly, it is the type of the pattern that needs to be decided.
Based on this decision, different classes in the pattern API will be used to
derive the new pattern. The pattern type can either be a "class", a "link",
or a "member".

After the pattern has been written using the provided API, the .class

files resulting from the compilation of the Java[10] code have to be placed in a special directory structure[11] that holds all the patterns. TCC interrogates the patterns, which respond appropriately if they adhere to the standard pattern API.

This interaction is done through the SciPattern interface, which has the following properties and members:

- *SciPatternProperty.PATTERN_CATEGORY* *property*
  Indicates the kind of object this pattern is applicable to.

- *prepare()*
  Checks if it is possible at all to apply this pattern to the target objects and makes some start-up preparations for the pattern.

- *canApply()*
  Checks whether the pattern can be applied to the target objects with the current values of pattern's properties.

- *apply()*
  Makes the pattern perform desired actions.

- *PropertyMap* *properties set*
  Defines the behaviour of a pattern.

7.4   Putting in perspective of the existing

One must now check if the tool examined in section 7.3 conforms to the expectations established in section 7.2. This placement in perspective will naturally keep the focus on the one tool Section 7.3 leaned on, Together ControlCenter 6.1.

7.4.1   Situation in theoretical context

As mentioned in Section 7.2.1, this chapter focuses on **tools presenting a library of abstract patterns** rather than on tools handling explicit patterns meta-model. Directly resulting of this, the chosen tool, Together

---

[10] As Together is a full Java application, all extensions have to be written in the same language. This does not affect the language independence of the pattern itself.

[11] In *%TOGETHER_HOME%/modules/com/togethersoft/modules/patterns*

ControlCenter, belongs to the category of tools which **expresses patterns in a language** and not through a representation of manipulable entities. As a matter of fact, it can only store patterns in two forms: through the programming language and the modelization language. TCC directly generates the objects and relationships between them. It also constructs the corresponding UML class diagrams. No meta-model or manipulable entities are created by Together.

As for the **strategies of instantiation**, TCC both offers Design Patterns generation and automation. Automatically, this implies that it handles all three types of concrete pattern instantiation: top-down, bottom-up, and mixed.

7.4.2   Help to conception

Section 7.2.2 asserts that any tool supporting Design Patterns automation should at least offer two basic functionalities: **Design Patterns generation and transformation**. Together ControlCenter clearly fulfills these requirements. Nothing more than examples given in sections 7.3.1 (Design Patterns generation) and 7.3.2 (Design Patterns application) is needed to prove that Together offers these features, that they work properly, and that the "behaviour preservation" requirement (in the case of transformation) is satisfied.

7.4.3   Generation of code and documentation

Patterns are defined[12] inside Together by an abstract representation which models objects and relationships without being tightened to a programming language. When time has come to generate the pattern, the abstract representation is adapted or transformed into **source code** in the project language. Once adapted to the target language[13], the result is high quality code. Albeit the code is rather embryonic, it yet respects conventions such as naming conventions[14] (case, indentation, and so on).

---

[12]Through the Pattern API in most cases

[13]The target language can be any language, provided that TCC supports it.

[14]Sun Microsystems' conventions if the project language is Java

Comments are optional. They consist of Pattern Links, as introduced in Section 7.3.4. Pattern Links play a double role. They both resolve traceability issues, and comment the source code. They actually are quite complete comments, since they describe everything that's needed to know: identification of the pattern, participants, and relationships between objects.

A graphical representation is generated corresponding to the source code. Together systematically represents patterns in UML class diagrams. Addicted users of the Entity-Relationship model might deplore that TCC does not handle this representation. Others might condemn that TCC does not generate sequence diagrams for the patterns for which this may be useful (mostly behavioral patterns).

This chapter decided to keep the focus on tools presenting a library of abstract patterns rather than on tools handling explicit patterns meta-model. Nevertheless, other CASE tools users might deplore that Together ControlCenter does not offer such a **meta-representation** of patterns.

7.4.4 User-friendliness and ease of use

As a general rule, Together ControlCenter is a fairly user-friendly and easy-to-use application. Some may blame TCC for offering **so many options** that the user quickly gets lost in all these menus; but that's the price to pay to be the most complete tool on the market.

Besides, to compensate for this, the software offers to the user a set of four "**user roles**" he/she can choose from. Together roles are predefined setups of the user interface that helps the user work from a specific point of view. For example, an architect designing a new system probably doesn't care about source code, and doesn't need or want to see the Editor or anything related to implementation. After a role option is selected, Together automatically sets up to provide ready access to only the relevant elements of the UI, and to show only the information in the model that best supports the chosen role. UI elements and/or model information that are not generally relevant to the role are hidden. The four roles are:

1. Business Modeler

   The Designer pane is central, with minimal menus for simplicity's sake.

2. Designer

   Both the Designer and Editor panes are central. Design and/or implementation are available up to the point of compilation, but no further.

3. Developer

   Both the Designer and Editor panes are central. Compile, Debug, Assemble, Deploy, and Run features are available in the UI.

4. Programmer

   The Editor pane is central, but the user can view the Designer pane upon demand. Compile, Debug, Assemble, Deploy, and Run are all available.

These role options definitely help to the general user-friendliness of the interface and to the ease of use of the software.

As for **pattern features**, they are quite easy to understand. When the (very) novice user has understood that Design Patterns are only handled in Class Diagram mode (which almost is the only pitfall he/she could encounter), the number of required steps to generate or apply a pattern is rather intuitive and cannot cause much concern.

Nevertheless, one substantial reproach can be directed to the tool regarding its speed. Together is proved to be, depending on the operating system, a quite **slow application**. The software has been written in Java (which is known not to be the language producing the most efficient applications) and relies on many resource-consuming modules.

7.4.5   Wide but structured patterns library

Together ControlCenter definitely offers an **impressive catalogue of patterns**. Figure 7.13 gives an estimate of the list TCC offers. This list holds by default a set of 127 patterns, sorted in 17 categories. Patterns defined by the Gang of Four only constitute one category of the catalogue. On top of that, the high extensibility of the tool allows users to add their own pattern or add any pattern found on the "Borland Developer Network" (BDN) website[15], making the pool of patterns unlimited.

---

[15]http://bdn.borland.com/together
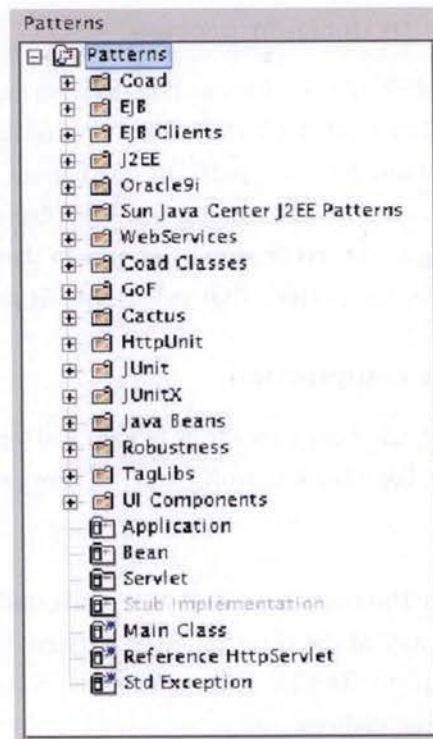
Figure 7.13: Together's pattern library

One regret often heard, though, is that the **GoF pattern catalogue** is not complete. It only lists 11 patterns[16] out of 23. A user desiring to generate or apply any other Design Pattern from the Gang of Four has to go look for it on the BDN, pray that he/she will find it there and, if found, go through the learning process of adding a pattern to Together's modules.

Regarding the **structure** of the pattern library, patterns may be sorted by categories according to their type (J2SE, Enterprise Java Beans, Oracle, Coad, JUnit, GoF, etc.), the Gang of Four category does not respect any requirement defined in Section 7.2.5. GoF patterns are neither (sub)classified by purpose (creational, structural, behavioral), nor by scope (class pattern versus object pattern).

---

[16]The Abstract Factory, the Adapter, the Chain of Responsibility, the Composite, the Decorator, the Factory method, the Observer, the Proxy, the Singleton, the State, and the Visitor

7.4.6   Support to decision process

The expectations of Section 7.2.6 regarding decision support only are covered
half-way by Together ControlCenter. The automation tool certainly does
provide a **description** for each pattern, participant, and parameters of a
pattern. But TCC, unfortunately, did not take decision support one level
higher by providing a **wizard** or step-by-step help that could lead the novice
user to the choice of the pattern that suits best his needs.

7.4.7   Patterns composition

Section 7.3.3 about the combination of Design Patterns demonstrated that
it is possible, with Together ControlCenter, to exert **single patterns** com-
position.

As for "**Design Pattern-oriented Subsystems**", TCC does not offer,
in its huge library, any of the three required patterns[17]. Unhappily, in order
to generate a complete "Design Pattern-oriented Subsystem", the user has
only two unsatisfying choices.

One alternative is to try to find each pattern on the Borland Developer
Network and combine them one by one. This has several disadvantages:

1. Not every pattern is available on the BDN,

2. Available patterns sometimes are user-twisted patterns rather than
   official patterns,

3. The learning process that the user has to go through to use the ac-
   quired patterns.

The other way is to implement a complete "Design Pattern-oriented Sub-
system" with the help of Together's API (cf. Section 7.3.5). Implementing,
from scratch, the generation of the complete subsystem risks to be time-
consuming.

On top of this, the issue persists for all **other types of subsystems**
composed of Design Patterns.

---

[17]Model-View-Controller, Mediator, and Façade

7.4.8 Consistency checking

Consistency checking is the one feature that is **totally inexistant** in TCC. Albeit traceability handling would allow such a functionality, Together offers no patterns validation at all.

7.4.9 Traceable graphical and textual representations

Together resolves traceability issues by offering its **Pattern Links** feature. Albeit Pattern Links are fairly intuitive and easily understandable by the user, they are not portable from one tool to another. Pattern Links are TCC-specific. Importing source files from a Together project into another CASE tool would be unsuccessful or at least incomplete. Pattern Links will stay in the code, as comments, but the tool will most certainly not be able to do anything with it.

7.4.10 Portability

Although Together's generated source code is portable to tools that handle the selected programming language, documentation is not portable at all. Whether documentation means Pattern Links or UML class diagrams, the format to represent them is TCC-specific. It exists no standard on the market for defining Patterns Links nor for a graphical representation of a UML class diagram.

7.4.11 Summary

Together ControlCenter is quite a efficient tool and meets some requirements very well. Among them, the help to conception, the generation of code and documentation, ease of use, and the adopted representations (both graphical and textual). Some expectations are, unfortunately, only half-way met. These are user-friendliness (cf. the speed of the application), the library of pattern (that certainly is wide, but not complete for GoF patterns, nor structured enough for GoF patterns), and pattern composition (works fine for single patterns but cannot handle subsystems). At last, three requirements are not handled at all: consistency checking, decision support (wizard inexistant), and portability.

Table 7.2: Requirements compliance

Requirements	Level of compliance
Help to conception	High
Generation of code and documentation	High
User-friendliness	Medium
Ease of use	High
Wide but structured patterns library	Medium
Decision support	Low
Patterns composition	Medium
Consistency checking	Low
Traceable representation	High
Portability	Low

As a general rule, albeit it is not perfect, TCC handles patterns quite professionaly. One must not overlook that TCC is a very good Design Pattern-capable CASE tool compared to other tools on the market. Many other tools are far away from meeting expectations like Together does.

7.5   Pertinence of Design Patterns automation

The desired results of Design Patterns automation have been stated in Section 7.2. After the automation process, the user should have sufficient graphical and textual representations of his pattern generation. These representations could be the code, a class diagram and another representation, more pattern-oriented (cf. Section 7.2.9).

For the generation (exposed in Section 7.1), the outcome is like an empty shell. As a matter of fact, the tool only provides the abstract design suggested by the pattern's solution part. It cannot be yet adapted to the real situation of the application. This shell has thus to be completed by the user. Section 7.3.1 illustrated this.

Regarding pattern application (see Section 7.1), the result might be slightly different. The user gives existing classes to the tool. On one hand, these classes are totally "complete", meaning that, after the pattern application, it has not to be completed. It implies that all the support for

the pattern has already been written (listeners and events for Model-View-Controller or Mediator, etc.). This presupposes a knowledge in depth of patterns. On the other hand, there will be some work on the resulting classes, in order to support entirely the applied pattern.

Most of the time, it appears thus that the result of the pattern generation is not complete and needs to be adjusted or even adapted. Knowing that, the question of the pertinence of Design Patterns automation is raised.

Two cases must be envisaged, depending of the user's qualification: either the user is experienced with the practice of patterns, is already convicted of their utility and applies them easily, or the user is not that used to patterns or is even an absolute beginner. This distinction might lead to different estimations concerning the pertinence of patterns automation.

First, one must ask if the patterns automation gives some time gain? There might be some gain, but it will not be huge, especially when applying one single pattern. It becomes more interesting for the automation of combined pattern or entire subsystems (even coming from user-defined plug-ins), when an important number of classes are involved. The tool will quickly give a structure to fill in.

A good point would be to know if automation enables a simpler approach of Design Patterns and facilitates their access and understanding. This is true above all for beginner users, since an experienced user does not really need an easier access to patterns. Thus, does automation really facilitate patterns understanding and use? A tool respecting all the requirements defined in Section 7.2 will definitely make the decision process of the beginner user easier. With the presentation of all the available patterns, a wizard to conduct the choice of the user and the coherence verification functionality, the access to patterns is much easier. But it might be difficult to find such a tool, meeting all the requirements (see Section 7.4 for the critic of Together). A tool that would not provide a wizard and a verification functionality does not present much interest anymore. It will not make a lot of difference with a great pattern book. Such a book presents the pattern, its intent, participants and collaborations, solution, advantages and drawbacks, etc. It even

often gives example of the pattern use. And a book is a more classical way to learn and might sometimes be more practical. . .

To conclude, patterns automation has to be taken for what it is. No tool, no matter how powerful, will replace the user's thinking regarding application architecture in general and Design Patterns. The automation results, as complete as they might be, have still to be adapted to the application situation by the user. And sometimes (for example when generating a single simple pattern), the result might be thin and look like an empty shell.

On the other hand, a performing tool could provide some better support to patterns understanding and facility of use. It might perform some useful operation - such as validation - and result in small time gain, especially when generating subsystems or several patterns. It must thus be treated as an useful and potentially powerful support for Design Patterns application.

7.6   Pertinence of Design Pattern-oriented Subsystems

Section 7.2 defined some requirements for a Design Patterns automation tool. One of these was the possibility to handle "Design Pattern-oriented subsystems" and to add some user-defined plug-ins enabling to manage any other subsystem (see Section 7.2.7). Would that mean that everything cannot be done with only "Design Pattern-oriented subsystems" as defined in Section 2.4? Such a subsystem is just an astute arrangement of three Design Patterns: the Mediator, the Observer and the Façade in a layered architecture. It was perfectly adapted to the Equipment Manager situation and needs. It was for the rest defined for subsystems needing a View and a Model (Model-View-Controller) and having to communicate with other subsystems (Façade and Mediator). However, this is not because the whole Equipment Manager is mostly constituted with such subsystems that everything must be built this way. . .

"Design Pattern-oriented Subsystems" are definitely not a universal panacea and, as much as for designing applications architecture than for programming, no solution can be applied blindly and everywhere. In any cases, the

application design should not be adapted to the Design Patterns but the opposite. Chapter 6 accurately illustrates this: the "Design Pattern-oriented Subsystem" was not applied as such.

"Design Pattern-oriented Subsystems" are thus just a configuration among others. That is why, in Section 7.2.7, it was not only expected that the tool allows the user to generate such a subsystem, but enables to generate any other subsystem: it must be possible to extend the tool's skills with user-defined plug-ins (defining, for example, other kinds of subsystems).

It is now established that "Design Pattern-oriented subsystems" are a configuration among others. That is, following the same approach as the one that led to the definition of such a concept, it is possible to invent several others subsystems made of different patterns composition. These new configurations would be adapted to other circumstances than the "Design Pattern-oriented Subsystem" (security, information transport on networks, etc.), for the reason that the patterns they are made of would be chosen according to these circumstances. Depending on the situation, it will thus be possible to chose the most adapted subsystem. In case of proliferation of subsystems based on specific patterns composition, this could lead to a catalogue of patterns subsystems, as the GoF proposes a catalogue of Design Patterns, listing common and recurrent problems in object-oriented programming and their solutions. The difference lies in the fact that the solutions would not be a single pattern anymore, but well a subsystem based on a specific patterns composition.

7.7 Summary

This chapter looks into Design Patterns automation. After defining basic concepts as Design Patterns generation and application, it lists requirements one can expect from a CASE tool supporting patterns automation. Such expectations are about features (help to conception, generation of code and documentation, support to decision process, consistency checking, etc.) but also about the quality of the tool's output (quality of the documentation and the code) or ease of use, extension capabilities, etc.

Afterwards, the chapter gives a detailed analysis of a representative tool: Together ControlCenter 6.1. The analysis covers in depth the processes of Design Patterns generation and application, the ability to combine several Design Patterns, traceability features, and the powerful extensions capabilities of Together.

A putting in perspective of the existing then follows. Together is confronted with each stated requirement. It results that Together Control-Center is quite a efficient tool and meets most requirements very well, but also suffers from absence of rather important features. For example, that TCC provides excellent help to conception and generation of code and documentation. It is also an easy tool to use based on standard and common representations. Anyhow, TCC does not support any consistency checking and provides only narrow decision support (i.e. no wizard helps the user selecting the right pattern to be applied). Besides, outputs of Design Patterns automation by Together is not portable enough.

After this analysis, this chapter tackles the pertinence of Design Patterns automation. Most of the time, the automation result is to be twisted again by the user. And when generating a simple pattern, the output often is very thin. Besides the output quality and level of completion, it might also be interesting to wonder if patterns generation actually results in some gain of time and allows an easier approach to Design Patterns. In the light of all these thoughts, one can conclude that patterns automation will not lead to miracle and has to be taken for what it really is. It will never replace the user's thinking. However, a tool meeting all stated requirements could provide good support in patterns understanding and application. With this condition, patterns automation can be envisionned as a useful and potentially powerful aid to Design Patterns application.

Finally, the main concept of this document, "Design Pattern-oriented Subsystems", is put in perspective. Such a subsystem, subtle arrangement of three Design Patterns (Façade, Observer and Mediator), is not a universal panacea and cannot be blindly applied as such. Moreover, as it is one configuration among others, other subsystems can be build on different patterns composition and be adapted to different situations. This could even lead to

the elaboration of a catalogue of subsystems.

Conclusion

This thesis analyzes the pertinence of Design Patterns in software application modules. It defines the concept of "Design Pattern-oriented Subsystem", a reusable application subsystem combining three Design Patterns in a layered architecture. This concept of subsystem foundation is intended to ease the construction of new modules. This work tests the relevance of "Design Pattern-oriented Subsystems" by confronting the new approach with a range of specific application subsystems. This document also analyzes automation of Design Patterns and "Design Pattern-oriented Subsystems" by CASE tools.

Chapter 1 presents the Equipment Manager and its context. It was our privilege to develop this application for Acme Corporation during our internship in the United States of America. The database editor is introduced to be used as illustration throughout this document. Illustrating theory by the development of this software application contributes in providing a "real-world" view of the use and application of Design Patterns.

Chapter 2 is about software architecture. It emphasizes the benefits of both horizontally layered and vertically cut architecture. Horizontal layering secludes presentation from application logic, domain, and persistence. Vertical division, instead, cuts an application in several subsystems or modules. Dividing a system into subsystem tends to make the software architecture more robust against changes and to make it highly reusable. This key chapter also exhibits GoF Design Patterns and introduces some of them. At last, the chapter combines benefits of Design Patterns and subsystems by creating a new concept, "Design Pattern-oriented Subsystems". This subsystem foundation is built by aggregation of the Observer pattern, the Mediator pattern, and the Façade. Direct advantages of this composition are low cou-

173

pling, high reusability, robustness and consistency.

Chapters 3 through 6 confront "Design Pattern-oriented Subsystems" with typical and unavoidable subsystems of a software application; namely to *business subsystems* (Chapter 3), *GUI subsystems* (Chapter 4)), *preferences subsystems* (Chapter 5), and *persistence subsystems* (Chapter 6). Each chapter follows roughly the same approach. They first explain the purpose of the studied subsystem and cover the existing technologies to make use of it. These chapters also illustrate the type of subsystem by its concrete implementation in the case of the Equipment Manager. On top of that, these chapters investigate about Design Patterns that are particularly useful for each specific type of subsystem. Moreover, these chapters check the pertinence of applying a "Design Pattern-oriented Subsystem" in such a module.

Chapter 3 focuses on Business subsystems. The business, also called the "Truth", is the heart of an application. It holds the essential data and interacts with all other modules; it reflects the application state at every moment. This chapter unluckily suffers from low recoil from the Equipment Manager application because no generality can be stated about this type of subsystem. Business modules typically are not reusable from one application to another. Furthermore, this chapter exhibits the help provided by the Decorator pattern in the Equipment Manager. The Decorator enables dynamic addition of functionalities to the subsystem. More precisely, it empowers the construction of a cache system in the Equipment Manager. The chapter also demonstrates that "Design Pattern-oriented Subsystems" cannot be systematically used for such a module because Business subsystems are too application-specific.

Chapter 4 looks into Presentation subsystems. Such subsystems, also called GUI subsystems, are those the user interacts with. Firstly, this chapter highlights the inherent difficulties in the construction of graphical user interfaces. Three main qualities are expected from graphical interfaces builders. It must be possible to change the interface as quickly and easily as possible since a GUI often undergoes a lot of changes during the development process of an application. The second quality required from the GUI is that it must be described in an easy way. Last but not least, the way of

elaboration of the interface is meant to be powerful and should not suffer from too much restrictions on the available GUI components.

This chapter introduced several ways of constructing a GUI, from the UI Builders to technologies using XML. Among them, the Bean Markup Language has been chosen to construct the GUI of both Presentation subsystems of the Equipment Manager. BML uses XML files to describe GUI components. Their "intelligence" is ensured by an application of the Observer pattern: the Model-View-Controller. In Presentation subsystems, the "Design Pattern-oriented Subsystem" can be applied as such. As a reminder, the two GUI subsystems of the Equipment Manager were used as examples to introduce this concept.

Chapter 5 concentrates on Preferences subsystems. Preferences modules are meant to store user settings in a permanent way. The responsibility of managing application settings can be encapsulated in one module. Encapsulation avoids coupling and thus allows this know-how to be fully reusable across applications.

After briefly positioning popular storage formats, the chapter dives into the preferences subsystem of the Equipment Manager. It reveals that the Equipment Manager stores preferences under the XML format and handles the XML Data Binding[18] process with the help of Castor Source Generator.

The confrontation of "Design Pattern-oriented Subsystems" with the structure of a typical Preferences subsystem discloses that the concept introduced in this thesis fits perfectly the needs of a preferences subsystem. The set of three GoF patterns can be applied as such, in order to be used as a foundation of any preferences subsystem.

Chapter 6 scrutinizes Persistence subsystems. This sort of module is responsible for storing any type of data in a permanent way. The chapter chronologically reviews the most common persistence paradigms: file systems, hierarchical databases, relational databases, object-oriented databases, and XML databases. The XML databases topic is covered in depth, intro-

---

[18]Mapping an XML document to its in-memory object representation

ducing different types of XML documents, types of XML databases, and specific query languages. This paradigm requires to be wholly understood before illustrating persistence by the Equipment Manager.

Acme's requirements for the implementation of the persistence module in the Equipment Manager are briefly exposed in this chapter. The chapter also discloses the decision process for both the database model and for the tool that is best adjusted to the specific requirements of the application. Acme's development team directed its choice on a Native XML database based on a file system with a specific naming convention. The process of XML Data Binding is once again managed by Castor Source Generator.

Additionally, the chapter examines how specific Design Patterns can improve the architecture of a persistence subsystem. The Strategy pattern favours technology independence. The Abstract Factory pattern reduces the coupling with other subsystems. The Decorator pattern eases the decomposition of database controllers into logical sub-controllers. At last, the Adapter pattern manages communication between incompatible interfaces.

The chapter eventually compares "Design Pattern-oriented Subsystems" with persistence subsystems. It seemed difficult at first sight to create a persistence module from a "Design Pattern-oriented Subsystem". Nevertheless, striving for exertion of good architecture principles, such as the application of the four Design Patterns mentioned here above, leads to a solution quite similar to what "Design Pattern-oriented Subsystems" recommend: high reusability thanks to the Strategy and Decorator patterns, low coupling thanks to the Abstract Factory and Adapter patterns.

Chapter 7 achieves a double goal. It both looks into CASE tools capable of Design Patterns automation and relativizes the two notions of Design Patterns automation and "Design Pattern-oriented Subsystems".

The chapter establishes a taxonomy of requirements one could expect from a Design Pattern-capable CASE tool. Among these expectations, one can find some help to conception, generation of code and documentation, user-friendliness and ease of use, a wide but structured patterns library,

decision support, patterns composition, consistency checking, traceable representations, and portability.

A study of the existing then focuses on one of the more mature Design Pattern-capable CASE tool on the marketplace, "Together ControlCenter 6.1" by Borland. The analysis covers in depth the processes of Design Patterns generation and application, the ability to combine several Design Patterns, traceability features, and the powerful extensions capabilities of Together.

In the same line, the chapter reviews requirements one by one in comparison with the study of the existing. It results that Together ControlCenter is quite a efficient tool and meets most requirements very well, but also suffers from absence of rather important features. For instance, TCC provides excellent help to conception and generation of code and documentation. It is also an easy tool to use based on standard and common representations. Anyhow, it appears that TCC does not support any consistency checking and provides only narrow decision support (i.e. no wizard helps the user selecting the right pattern to be applied). Besides, outputs of Design Patterns automation by Together is not portable enough.

Moreover, Chapter 7 puts Design Patterns automation back in its place. Results of patterns automation still have to be checked or modified by the user. On the other hand, it sometimes provides some time-gain, especially when generating combined patterns or whole subsystems. Eventually, a tool meeting all or almost all requirements defined in section 7.2 would definitely enable an easier access to Design Patterns and present a great interest. In the meantime, the interest of patterns automation is more limited and can be seen as little time-gain only.

The chapter also recalls that "Design Pattern-oriented Subsystems" is just an astute arrangement of three Design Patterns but does not pretend to be the one solution for the creation of any type of subsystem of any software application. It is a configuration among others. That is, following the same approach as the one that led to the definition of this concept, it is possible to define others patterns composition, adapted to different situations

than the "Design Pattern-oriented Subsystems". The resulting subsystems could even be put in a catalogue of patterns subsystems, exactly as the GoF proposes a catalogue of Design Patterns, except that the suggested solutions would be entire subsystems.

To recapitulate in a few words, this thesis suggests a new approach to software development using Design Patterns: "Design Pattern-oriented Subsystems". These are a subtle aggregation of three Design Patterns in one entity. The new concept is to be used as a subsystem foundation for easing the creation of new subsystems in software applications. To check on its pertinence, this document confronts "Design Pattern-oriented Subsystems" with a range of typical and unavoidable subsystems. This paper subsequently inspects the requirements one could have from a Design Pattern-capable CASE tool and verifies that tools existing on the market meet these expectations. At last, this work puts in perspective the notions introduced such as "Design Pattern-oriented Subsystems" and Design Patterns automation.

Glossary

This glossary gathers definitions coming from the following sources: [BG02, BMR+96, GHJV95, oEE90, Joh97a, Joh99a].

Abstract class A class whose primary purpose is to define an interface. An abstract class defers some or all of its implementation to subclasses. An abstract class cannot be instantiated.

Abstract coupling Given a class A that maintains a reference to an abstract class B, class A is said to be *abstractly coupled* to B. It is called abstract coupling because A refers to a *type* of object, not a concrete object.

Abstract Factory Creational Design Pattern. Provide an interface for creating families of related objects without specifying their concrete class. See page 122.

Adapter Structural Design Pattern. Convert the interface of a class into another interface clients expect. Adapter let classes work together that couldn't otherwise because of incompatible interfaces. See page 130.

Amplifier (Audio) An electronic component that takes a weak audio signal and increases it to generate a signal that is powerful enough to drive speakers. (General) An electronic component that accepts a low-level signal and recreates the signal with more power.

API Application Programming Interface: the set of services that an operating system or a programming language makes available to programs that run under it.

Application A program or collection of programs that fulfills a customer's requirements.

Architecture See Software architecture.

Backward compatible An application is backward compatible if it can read and handle previous/obsolete versions of documents it has produced.

Bean Markup Language The Bean Markup Language is an XML-based language used to describe the structure of interconnected Java Beans. The main goal of the Bean Markup Language is to describe declaratively a whole structure of interconnected beans capable of functioning together as a component, or even as a complete application. See page 83.

BML See Bean Markup Language.

CASE Computer-Aided Software Engineering. CASE is the use of computer-based support in the software development process.

CASE tool A CASE tool is a computer-based product aimed at supporting one or more software engineering activities within a software development process.

Class A class defines an object's interface and implementation. It specifies the object's internal representation and defines the operations the object can perform.

Class diagram A UML diagram that depicts classes, their internal structure and operations, and the static relationships between them. See page 187. item [Client] Denotes a component or a subsystem that exploits functionality offered by other components.

Component See Software component.

Concrete class A class having no abstract operations. It can be instantiated.

Coupling The degree to which software components depend on each other.

DBMS Database Management System

Decorator Structural Design Pattern. Attach additional responsibilities to an object dynamically. Decorator provides a flexible alternative to subclassing for extending functionality. See page 71.

Design The activity performed by a software developer that results in the software architecture of a system. Very often the term design is also used as a name for the result of this activity. The software design activity is commonly divided into the high-level design and the low-level design. The high-level design results in the structural subdivision of the system. It specifies the fundamental structure of the application. The low-level design results in more detailed planning like definition of interface, data structures, etc.

Design Pattern A Design Pattern systematically names, motivates, explains, and evaluates an important and recurring design in object-oriented systems. It describes the problem, the solution, the conditions needed to apply the solution, and its consequences. It also gives implementation hints and examples. The solution consists in an abstract design: it is a configuration of classes and objects that solve the problem. The suggested solution is to be adapted to the application context. See page 46.

DOM Document Object Model provides a standard set of objects for representing and manipulating HTML and XML documents.

Domain Denotes concepts, knowledge and other items that are related to a subject. Often used as 'application domain' to denote the problem area an application addresses.

Drag and drop User activity supported by modern UI Builders. Drag an drop allows a user to perform an operation on a graphical object by selecting it and dragging it to another place on the screen.

DTD Document Type Definition. Describes the structure and the types of an XML document.

Encapsulation The result of hiding a representation and implementation in an object. The representation is not visible and cannot be accessed directly from outside the object. Operations are the only way to access and modify an object's representation.

Equalizer Electronic device (as in sound-reproducing system) used to adjust response to different audio frequencies.

Façade Structural Design Pattern. Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use. See page 63.

Framework A set of cooperating classes that makes up a reusable design for a specific class of software. A framework provides architectural guidance by partitioning the design into abstract classes and defining their responsibilities and collaborations. A developer customizes the framework to a particular application by subclassing and composing instances of framework classes.

Gang of Four This expression refers to Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides who have written the seminal book "Design Patterns: Elements of Reusable Object-Oriented Software" [GHJV95].

GoF See Gang of Four.

GUI Graphical User Interface. The part of the program that the user sees and interacts with, as opposed to the part of the program that performs its internal processing.

Inheritance A relationship that defines one entity in terms of another. Class inheritance defines a new class in terms of one or more parent classes. The new class inherits its interface and implementation from its parents. The new class is called a subclass or a derived class. Class inheritance combines interface inheritance and implementation inheritance. Interface inheritance defines a new interface in terms of one or more existing interfaces. Implementation inheritance defines a new implementation in terms of one or more existing implementations.

Interface The set of all signatures defined by an object's operations. The interface describes the set of requests to which an object can respond.

Java Bean JavaBeans turns classes into software components by providing several new features. See page 199.

Layer r Layering is one of the most common techniques that software designers use to break apart a complicated software system. When thinking of a system in terms of layers, the principal subsystems in the software arranged can be imagined in some form of layer cake, where

each layer rests upon a lower layer. In this scheme the higher layer uses various services defined by the lower layer, but the lower layer is unaware of the higher layer. Furthermore, each layer usually hides its lower layers from the layers above, so layer 4 uses the services of layer 3 which uses the services of layer 2, but layer 4 is unaware of layer 2. See page 195 for layered architecture.

Loudspeaker Device that changes electrical signals into sounds loud enough to be heard at a distance.

Loudspeaker band data The set of band data (sensitivity, efficiency, power, etc.) determines the contribution of a loudspeaker at a given location and orientation in space to a given listener location. One can accumulate the contributions of all loudspeakers to get an idea of the quality of sound for a listener.

Loudspeaker directivity In a loudspeaker system, the directivity is an indication of how directional the loudspeaker is, or to look at it another way, how effective the speaker is at taking the sound it produces and sending it in one particular direction instead of all directions.

Loudspeaker taps Some loudspeakers have a built in transformer device with a switchable power setting. For example, a loudspeaker may have 1, 2, 4 and 8 Watt taps. This means that the loudspeaker will be roughly 8 times more powerful when set to the 8 Watt tap than the 1 Watt tap. Taps are used when loudspeakers in an audio system need to play sound at different power level.

Mediator Behavioural Design Pattern. Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently. See page 53.

Message Messages are used for the communication between objects or processes. In an object-oriented system, the term message is used to describe the selection and activation of an operation or method of an object. This kind of message is synchronous, which means that the sender waits until the receiver finishes the activated operation.

Method Denotes an operation performed by an object. A method is specified within a class.

Module A syntactical or conceptual entity of a software system. Often used as a synonym for component or subsystem. Sometimes, modules also denote compilation units or files. Other writers use the term as an equivalent to package when referring to a code body with its own name space. This term is used as stated in the first sentence.

Object An identifiable entity in an object-oriented system. Objects respond to messages by performing a method (operation). An object may contain data values and references to other objects, which together define the state of the object. An object therefore has state, behaviour, and identity.

Observer Behavioural Design Pattern. Define a one-to-many dependency between objects so that when one object changes state , all its dependents are notified and updated automatically. See page 48.

Poor traceability One of the main Design Patterns drawbacks. Poor traceability points out that the track of design patterns is lost during implementation. See page 142.

Relationship A connection between components. A relationship may be static or dynamic. Static relationships show directly in source code. They deal with the placement of components within an architecture. Dynamic relationships deal with the interaction between components. They may not be easily visible from source code or diagrams.

Request An object performs an operation when it receives a corresponding request from another object. Is is a common synonym for message.

Responsibility The functionality of an object or a component in a specific context. A responsibility is typically specified by a set of operations.

Reusability The degree to which a software module or other work product can be used in more than one computing program or software system.

Robustness diagram Robustness diagram is part of an extension of UML. It defines the first cut into components of an system. See page 191.

SAX Simple API for XML (SAX) is a standard interface for event-based XML parsing.

Sequence diagram An UML diagram that shows a dynamic view of a system. It enables to represent the collaborations between objects in a temporal point of view. Sequences diagram are useful when illustrating a scenario. See page 189.

Serialization Object serialization supports the encoding of objects, and the objects reachable from them, into a stream of bytes; and it supports the complementary reconstruction of the object graph from the stream.

Signature An operation signature defines its name, parameters, and return value.

Software component Software components are "black boxes" that encapsulate functionality and provide services based on a specification. They are highly reusable and interchangeable. As classes, software components hide implementation, conform to interfaces and encapsulate data. See page 199.

Strategy Behavioural Design Pattern. Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. See page 119.

Subclass A class that inherits from another class. A subclass is also called a derived class.

Subsystem Semantically useful grouping of collaborating components performing a given task. A subsystem is considered as a separate entity within a software architecture. It performs its designated task by interacting with other subsystems and components.

Swing SUN's library for building user graphical interfaces in Java.

Transaction (database) Group of commands which are to be treated as a single atomic event.

UI User Interface. See GUI.

UML The Unified Modeling Language is a standard modeling language for software. It has been thought of for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. Basically, UML enables developers to visualize their work products in standardized blueprints or diagrams. See page 187.

XML The Extensible Markup Language (XML) is language designed to describe data.

XML Data Binding Representing an XML document directly in-memory.

XML Schema An XML Schema is a specific XML language that describes the structure and the types of an XML document.

Appendix A

The UML notation

This short introduction to UML[1] is directly inspired of [JBR99] and [Hab]. This introduction does not pretend to cover in depth the UML subject.

The Unified Modeling Language is a standard modeling language for software. It has been thought of for visualizing, specifying, constructing, and documenting the artefacts of a software-intensive system. Basically, UML enables developers to visualize their work products in standardized blueprints or diagrams.

UML proposes a heterogeneous set of models. The most used models in this document will be introduced: class diagrams, sequence diagrams, robustness diagrams and use cases.

A.1 Class diagrams

A class diagram is a collection of elements from static modeling (classes, etc.), showing the structure of a model. It does not handle dynamical and temporal aspects.

Here are briefly exposed the main concepts relating to class diagrams.

- object: basic concept of the analyzed problem

- class: set of objects sharing some characteristics

---

[1]Unified Modeling Language

- association: correspondence between two objects; associations have two roles

- multiplicity: constraint on a role of an association, determining how many objects are participating

- under-typing: correspondence between two objects representing a relation of generalization/specialization

- attribute: characterizes an object by taking a specific value in a values class (domain)

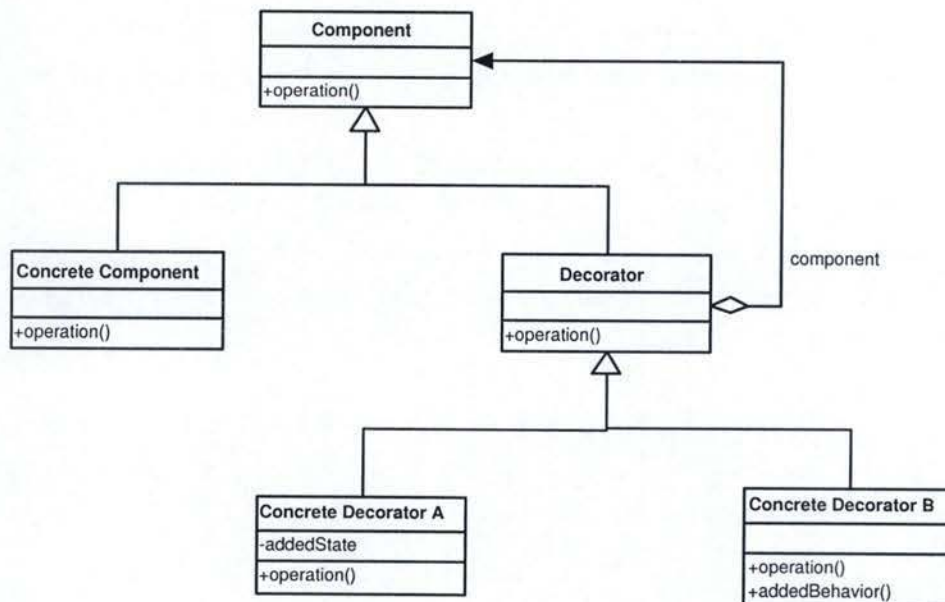- operations: operations associated to classes

Figure A.1: Example of UML class diagram

Figure A.1 and Figure A.2 show examples of simplified - where interfaces are not distinguished from classes, etc. - class diagrams. Classes are represented by a rectangle, with their name in the top of the rectangle (see the class *Concrete Decorator A* for example). If existing, Class attributes are put in the rectangle below (see the attribute *AddedState* of the *Concrete Decorator A*). Eventually, class operations are placed in a rectangle below (see *operation()* in the same class).
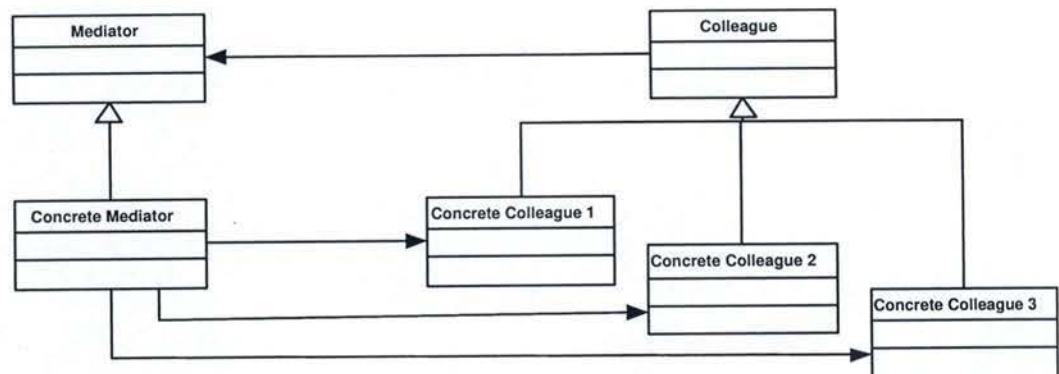
Figure A.2: Simplified UML class diagram

Simple lines between classes represent associations: classes instances are connected. It means that one class has an instance variable that refers to the other class. For example, the *Concrete Mediator* holds references to its colleagues. The arrowed line express that both *Decorator* and *Concrete Component* are *Component*. The "diamond" line indicates that a class contains a collection of instances of another class (see *Decorator* and *Component*).

A.2 Sequence diagrams

Sequence diagrams describe a system: a set of objects interacting through messages. They enable to represent collaborations between objects in a temporal point of view: what is to be shown is the messages chronology. Each object has its own "line of life". The order of the messages is determined by their position on the vertical axe; time runs out from the top to the bottom of this axe. Sequence diagrams are very useful when illustrating a scenario.

Figure A.3 presents an example of a UML sequence diagram. Vertical dashed lines indicate the existence of an object over time. Vertical rectangles show the activity periods of an object. Arrows between vertical lines represent methods calls, or messages. The message might contain the name of the method and the parameters passed in. Messages can be synchronous or asynchronous. A synchronous message blocks the message expeditor until the addressee treats it, in opposition to the asynchronous message. Object creation, not to be confounded with object activation, is symbolized by an
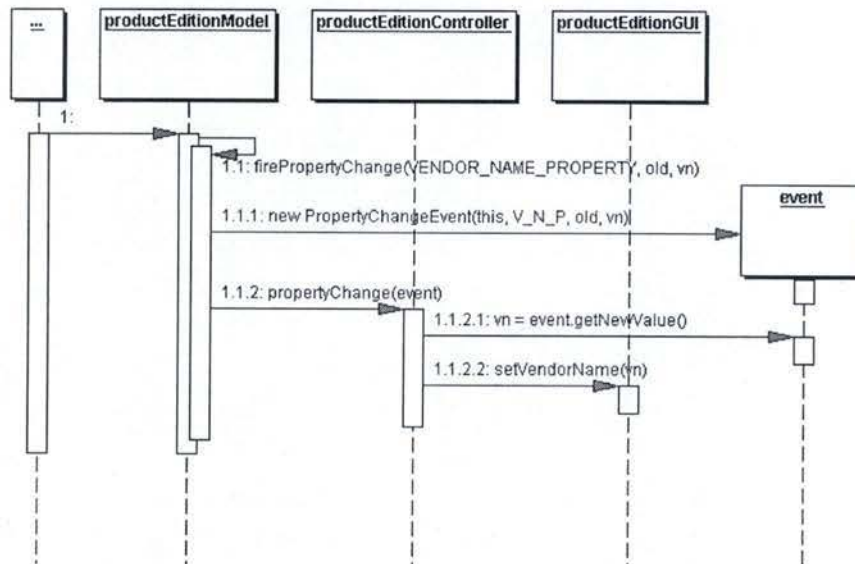
Figure A.3: Example of UML sequence diagram

arrow reaching the rectangle containing the object name. In the example, the *productEditionModel* creates an *event*. When an object calls a method on itself, it is drawn with an arrow loop of which begin and end are the same object.

A.3   Use cases

Use cases offer an external view of the system, in a user's point of view. As the sequence diagram - and contrary to the class diagram, it gives a dynamic sight of the system: it describes a set of scenarios - which are sequences of actions. Finally, it enables the developer to have an "objective-oriented" view, each use case being associated to a user objective. Use cases are therefore a set of "stories" describing how a user interacts with the system in order to achieve its goal.

A.4   Robustness diagrams

Robustness diagrams do not really belong to UML, but it is an extension proposed in 1991 by Ivar Jacobson.

The idea is to refine the use cases to obtain a first cut in components. Four types of components can be defined and robustness diagrams express a first sketch of the interactions between these components.

The four components are:

- actors

- interfaces

- controls

- repositories

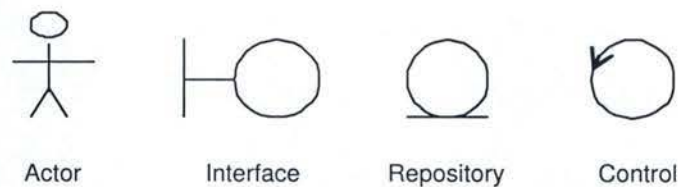Figure A.4 shows the graphical representations of the four robustness components.

Actor   Interface   Repository   Control

Figure A.4: Robustness Components

A.4.1   Actors

The actors are components that correspond to the definition of user in the use cases.

A.4.2   Interfaces

Interfaces are components allowing interactions between a user and the system, as, for example, a ticket-window, an alert message, etc.

A.4.3   Controls

These components, also called controllers, contain some "intelligence" in order to insure the objective of the use case; it is also possible to introduce a structure of these controls with a composition/decomposition relation.

A.4.4   Repositories

Repositories are components responsible for the information stock, as a database, an archives local, etc.

A.4.5   Interactions

Robustness diagrams express interactions between components through links between them. The possible links follow:

- an actor dialogs with an interface

- an interface sends information to a controller

- a controller communicates with another controller

- a controller uses a repository

- a control initiates or solicits an interface

- a control is composed of several controls

A.4.6   Example

The robustness diagram of the Equipment Manager appears in Figure A.5. One will have noticed that all the conventions of the robustness diagram were not respected "as such" (Models, represented as repositories, seem to communicate with databases, also expressed as repositories, etc.). But this diagram still gives an excellent overview of the Equipment Manager architecture (components and interactions between them).
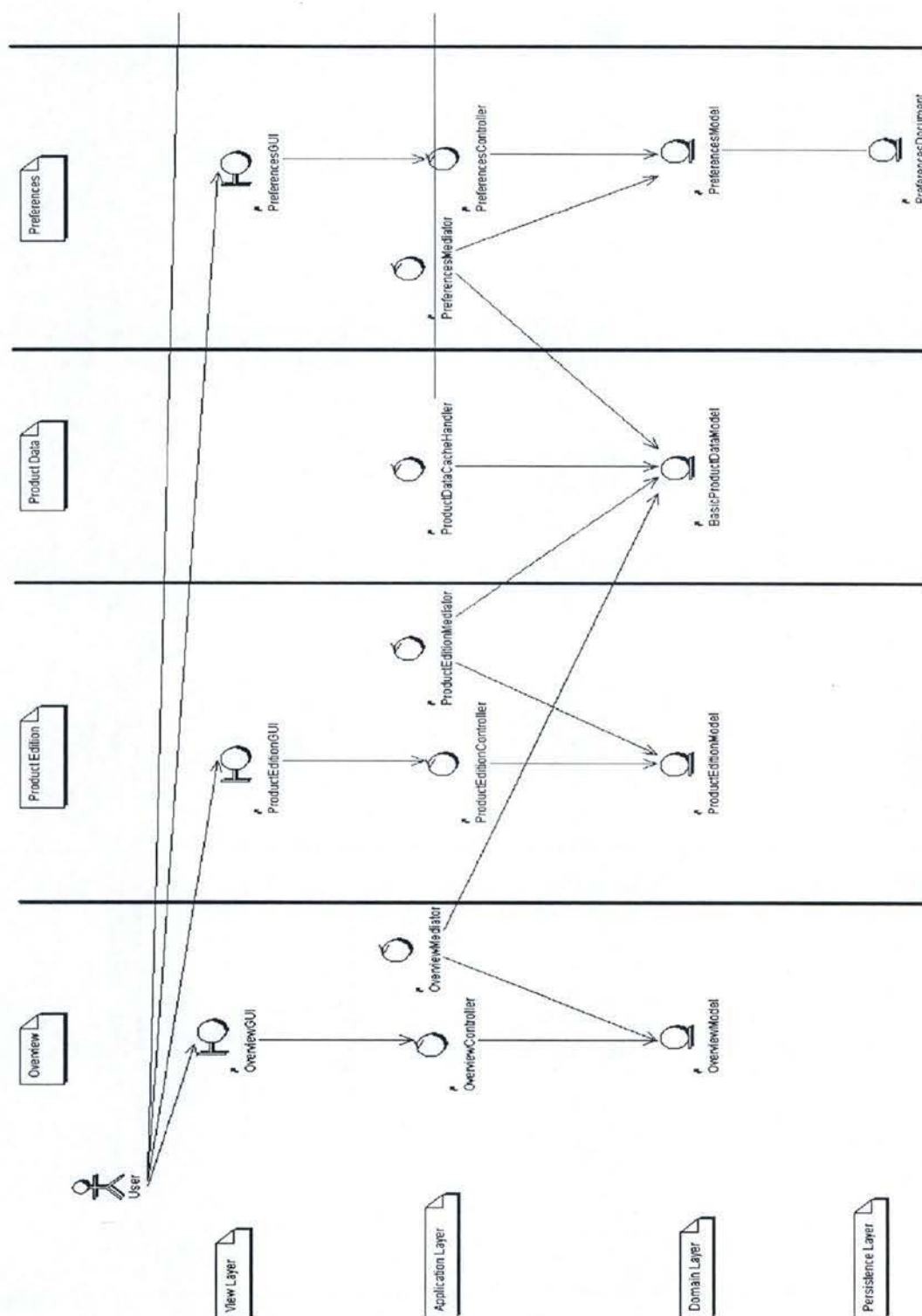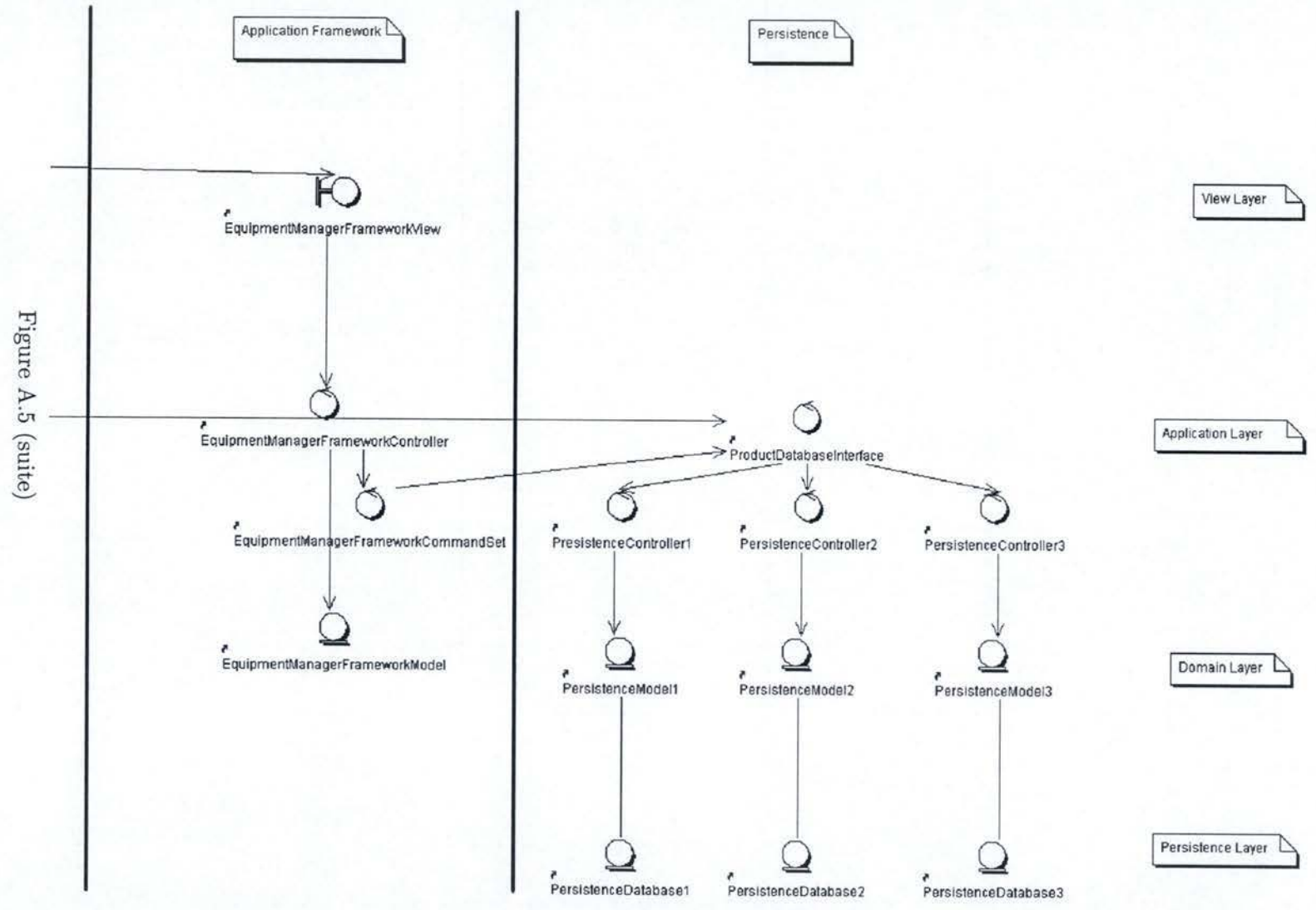
Figure A.5: Robustness Example

The UML notation

Figure A.5 (suite)

Appendix B

Principles of Layered architecture

B.1 Layers architectural pattern

These principles are extracted from the Layers architectural pattern, exposed in [BMR+96]. Networking protocols are probably the best-known example of layered architectures. Each layer deals with a specific aspect of communication and uses the services of the next lower level. A system built following this architecture is divided into an appropriate number of layers, placed one above the other (see Figure B.1).
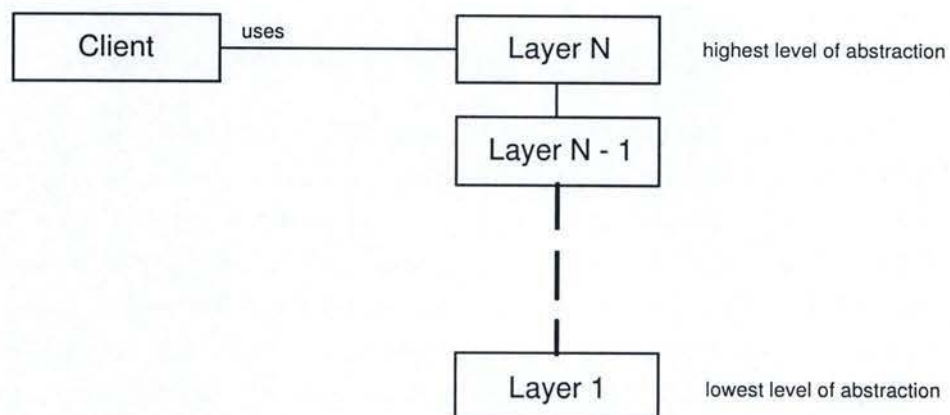
Figure B.1: Layered architecture

The first level corresponds to the lowest level of abstraction, while the last one - the highest - corresponds to the highest abstraction level. Within a layer, all the components work at a same level of abstraction. Most of the services provided by a layer J are actually composed of services that a layer J-1 provides. *"In other words, the services of each layer implement a strategy for combining the services of the layer below in a meaningful way."* [BMR+96]

Apart from networking protocols, other known uses have been made of the layered architecture, especially for Information Systems (IS), or Enterprise Architecture. [Fow02]

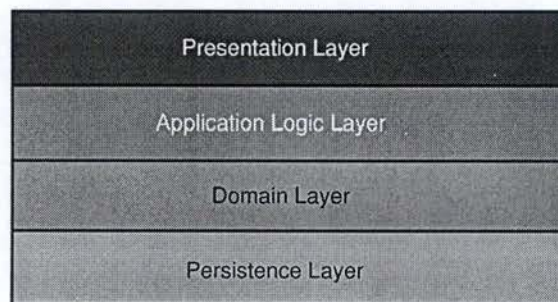## B.2   Layered architecture for Information Systems

Figure B.2: Four-tier architecture

Information Systems from the business software domain often use layered architecture; in this case, layers are also called tiers. The two-tier architecture is an old widespread division for interactive information systems. [BG02] The bottom layer is a database, holding company-specific data, while the top layer consists of many applications working concurrently to fulfil different tasks. This is a very common architecture in Client-Server systems. However, the tight coupling between user interface and data representation leads to several problems, such as a major lack of evolving capacity and reusability. Furthermore, storage mechanisms are often unable to offer a true representation of modelled concepts. This is why a third layer has

been introduced between the database and the interface; it is called the domain layer. Its purpose is to model the conceptual structure of the domain. Moreover, the top layer, still mixing user interface and application, is split in two; the result is a four-tier architecture, as shown in Figure B.2. The whole Equipment Manager is based on a four-tier architecture.

Appendix C

Java Beans

Java Beans are the software components architecture of the Java language. Before going further with beans, it is necessary to give some explanations about software components.

C.1 Software components

"Software components are to software what integrated circuits (ICs) are to electronics: "black boxes" that encapsulate functionality and provide services based on a specification." [Joh97a] They are of course designed to be highly reusable and even interchangeable: they provide specific functionality that can be reused in different places.

As classes in object-oriented languages, software components hide implementation, conform to interfaces and encapsulate data. So, where is the difference between classes and software components? Actually, almost all software components are classes. The only distinction is that components conform to a **software component specification**. [Joh97a] The **JavaBeans specification** is the document specifying what a Java class must do in order to be considered as a Java Bean.

C.2 Java Beans

The only requirement needed to make a class into a Bean is that the class implements the *java.io.Serializable* interface. Serializable classes know how

to package themselves into streams of bytes to be transmitted through networks or saved to disk, awaiting later reincarnation. [Joh97a]

Java Beans thus turn classes into software components by providing several new features. Apart from the serialization, beans have properties, which are attributes of the object. They can be customized through these properties, using accessors (*setProperty()* and *getProperty()*). In general, customization means configuring the internal state of a bean so that it appears and behaves properly in the situation in which it is being used. [Joh97b] The new event handling scheme of Java can also ease communication between beans: a class registers interest in the activities of another class by way of a listener interface.

C.3   XML Java Beans

It is possible to "mix" Java Beans and XML in order to make Java Beans mobile and interoperable, by representing them as XML documents. [Joh99c] XML is used as a serialization format for beans. It is also possible to create XML files specifying values for Java Beans' properties (customization).

Appendix D

Simple BML example

This very simplified example is given in order to show how to apply the Bean Markup Language in a Model-View-Controller architecture. It illustrates the construction and handling of a single window containing nothing but a textfield.

The assumption is made that a bean has already been declared for the panel itself with the unique tag "mainPanel". What is left to do in the BML file is to add the textfield in the panel. One simple layout has been chosen for the panel: the border layout. Components can be placed on the North, South, East, West or Center of the layout's space.

```
<bean source="mainPanel">
  <property name="layout">
    <bean class="java.awt.BorderLayout">
      <args>
        <cast class="int">
          <string value="0" />
        </cast>
        <cast class="int">
          <string value="0" />
        </cast>
      </args>
    </bean>
  </property>
  <add>
```

```
   <bean class="javax.swing.JTextField" id="textfield" />
   <string value="Center" />
 </add>
</bean>
```

The textfield is added to the center of the window. This component is referred by a unique tag such as "textfield". The Controller of the View will have to look up for them using only this name. Using the BML compiler, from this simplified sample of BML file a Java class is generated. This is what will play the role of the View. A Controller and a Model need now to be defined, in order to apply the Model-View-Controller pattern. The View represents the state of the Model, it is a sort of a picture of it. The definition of the Model of this example is thus quite simple. It is above all composed by one field, keeping trace of the textfield from the View. When the user hits the "enter" key after writing something in the text area, this is "recorded" in the Model (under the shape of a String).

```
1   public class ProductEditionModel {
2
3   //------------------------------------------------------------+
4   // Constructors
5   //------------------------------------------------------------+
6
7   public ProductEditionModel() {
8     listenersList = new ArrayList();
9   }
10
11  //------------------------------------------------------------+
12  // Public methods
13  //------------------------------------------------------------+
14
15  public String getText() {
16    return text;
17  }
18
19  public void setText(String text) {
20    Object oldValue = this.text;
21    this.text = text;
```

```
22      firePropertyChanged(TEXT_PROPERTY, oldValue, text);
23  }
24
25  //---------------------------------------------------------------+
26  // Listeners
27  //---------------------------------------------------------------+
28
29  /**
30   * Registers a PropertyChangeListener with this class.
31   **/
32    public void addPropertyChangeListener(PropertyChangeListener pcl) {
33       listenersList.add(pcl);
34    }
35
36  /**
37   * Removes a PropertyChangeListener with this class.
38   **/
39  public void removePropertyChangeListener(PropertyChangeListener pcl) {
40     listenersList.remove(pcl);
41  }
42
43  /**
44   * Notifies all registered PropertyChangeListeners when a bound
45   * property's value changes.
46   **/
47  protected void firePropertyChanged(String fieldName, Object oldValue,
48                                        Object newValue) {
49    if ((oldValue == null && newValue == null) ||
50       (oldValue != null && oldValue.equals(newValue))) {
51     return;
52    }
53    PropertyChangeEvent event =
54      new PropertyChangeEvent(this, fieldName, oldValue, newValue);
55    Iterator listenersListIterator = listenersList.iterator();
56    while (listenersListIterator.hasNext()) {
57      ((PropertyChangeListener)listenersListIterator.next()).
```

```
58          propertyChange(event);
59     }
60  }
61
62  //-----------------------------------------------------------------+
63  // Attributes and properties
64  //-----------------------------------------------------------------+
65
66  public static final String TEXT_PROPERTY = "text";
67  private String text;
68  private List listenersList;
69 }
```

Of course, the Model has to keep a trace of all its listeners, as explained in section 2.3.1. (see line 7 to 10 and 25 to 61). It has also some methods to access the field "text" ("getText()" and "setText()"). On its side, the Controller has to look for all the beans it wants to control. Here follow some samples of code showing how the Controller looks up for the bean (lines 45 to 52) and manages them (lines 56 to 68).

```
1   public class Controller {
2
3       //-----------------------------------------------------------------+
4       // Constructors
5       //-----------------------------------------------------------------+
6
7       Controller (Model model){
8         if (model == null) {
9             throw new IllegalArgumentException("Model cannot be null");
10        }
11        lookupObjects();
12        setModel(model);
13        hookupObjects();
14     }
15
16      //-----------------------------------------------------------------+
17      // Public methods
18      //-----------------------------------------------------------------+
```

```
19
20   public void setModel(Model model) {
21     if (theModel != null) {
22       Model.
23         removePropertyChangeListener(getModelListener());
24     }
25     theModel = model;
26     if (model != null) {
27       model.addPropertyChangeListener(getModelListener());
28     }
29     initializeView(model);
30   }
31
32   //-----------------------------------------------------------------+
33   // Methods
34   //-----------------------------------------------------------------+
35
36   protected void initializeView(Model model) {
37     if (model == null) {
38       return;
39     }
40     text.setText(model.getText());
41   }
42
43   //-----------------------------------------------------------------+
44
45   /**
46    * Lookup for widgets registered in the BML registry.
47    */
48   private void lookupObjects() {
49     BmlParser theBmlParser = BmlParser.theInstance();
50     text = (JTextField)theBmlParser.
51       lookupObject("textfield");
52   }
53
54   //-----------------------------------------------------------------+
```

```
55
56   /**
57    * Attach widgets with their respective listeners who update the model
58    */
59   private void hookupObjects() {
60     EditBeanCommand command =
61       new EditBeanCommand("productEdition.editBeanCommand",null,true);
62     ArgumentFactory factory = new
63       BeanPropertyArgumentFactory(this);
64     TextFieldTrigger textTrigger =
65       new TextFieldTrigger(text,Model.TEXT_PROPERTY);
66     textTrigger.setCommand(command);
67     textTrigger.setArgumentFactory(factory);
68   }
69
70   //------------------------------------------------------------------+
71
72   private PropertyChangeListener getModelListener() {
73     if (theModelListener == null) {
74       theModelListener = new ModelEventHandler();
75     }
76     return theModelListener;
77   }
78
79   //------------------------------------------------------------------+
80   // Private inner classes
81   //------------------------------------------------------------------+
82
83   /**
84    * This class listens to propertychange events triggered by a
85    * Model and update the view accordingly.
86    */
87   private class ModelEventHandler
88     implements PropertyChangeListener {
89     public void propertyChange(PropertyChangeEvent e) {
90       String propertyName = e.getPropertyName();
```

```
91        if (propertyName.equals(Model.TEXT_PROPERTY)){
92          String newValue = theModel.getText();
93          String oldValue = (String)e.getOldValue();
94          if (oldValue!=null && !oldValue.equals(newValue)){
95            text.setText(newValue);
96          }
97          else if (oldValue == null) {
98            text.setText(newValue);
99          }
100     }
101   }
102 }
103
104 //-----------------------------------------------------------------+
105
106 private JTextField text;
107 private Model theModel;
108 private PropertyChangeListener theModelListener;
109}
```

The Controller uses a Command to update the Model when the View has been modified by the user (line 56 to 68). The Command is another Design Pattern. Briefly, a trigger knows which property to change in the Model and the value it has to put in. Each time the textfield in the View undergoes a change of state, the trigger "wakes up" and updates the Model. It is as if a listener was listening to the View and updating the Model accordingly. The Command pattern will not be exposed in this study.

The application of the Model-View-Controller with BML replaced the event-binding abilities of the Bean Markup Language.

Appendix E

Browser-based Application Toolkit

The BAT[1] is a technology that allows to build user interfaces using XML documents.

E.1 Principles

BAT is is a Web presentation framework with extendible building blocks for creating a professional, consistent user interface [Lab02]. It propounds a set of reusable elements to create a GUI. The parameters of these elements are set with XML data. BAT is therefore a customizable user interface framework. It is composed of two major parts: the run time portion and a set of 9 UI[2] elements. Four of them are containers elements, whereas the others are called basic elements. The user cannot define his own components but can use the predefined ones in the way he/she wants (defining complex orderings). [Lab02]

wizard (container) Figure E.1 shows an example of a wizard. A wizard is useful when the user is supposed to enter information into panels, in a specific order. The user must follow the order and cannot jump from one panel to another the way he/she wants. He can just move to the previous or following panel or cancel the current panel.

---

[1] Browser-based Application Toolkit
[2] User Interface

Campaign Initiative General Definition

▸ General
  Where
  Conditions

Initiative name (required)

Description

Start date (required)
  Year   Month   Day
  1970   01   01   [16] Time 00:00
End date (required)
  ☑ Run this initiative indefinitely

Next Cancel

Figure E.1: BAT container: the wizard

notebook (container)

General
  ▸ Registration
  ▼ **Demographics**
     Gender
     Age
     Income
     Marital status
     Children
     Household
  ▸ Address
  ▸ Culture
  ▸ Purchase history
  ▸ Miscellaneous

Marital Status

Define the customer marital status criteria for this customer profile.

◉ Ignore marital status
◯ Target the following marital statuses
  ☐ Not Provided
  ☐ Single
  ☐ Married
  ☐ Common-law
  ☐ Separated
  ☐ Divorced
  ☐ Widowed
  ☐ Other

OK Cancel

Figure E.2: BAT container: the notebook

Figure E.2 gives an idea of what a notebook looks like. With a notebook, the user is allowed to jump from one panel to another. A notebook might be useful when displaying or collecting large sets of information that are not necessarily sequential or closely related.

dialog (container) The dialog is quite a simple container. It is used for

displaying summary or confirmation information and to claim more single input information.

tools UI center (container) The Tools UI Center provides a structured framework for tools to be presented to the user. The Tools UI Center consists of a banner frame, which contains a progress indicator and page history (list of window depth the user is under), a menu frame, and a content frame (see Figure E.3).
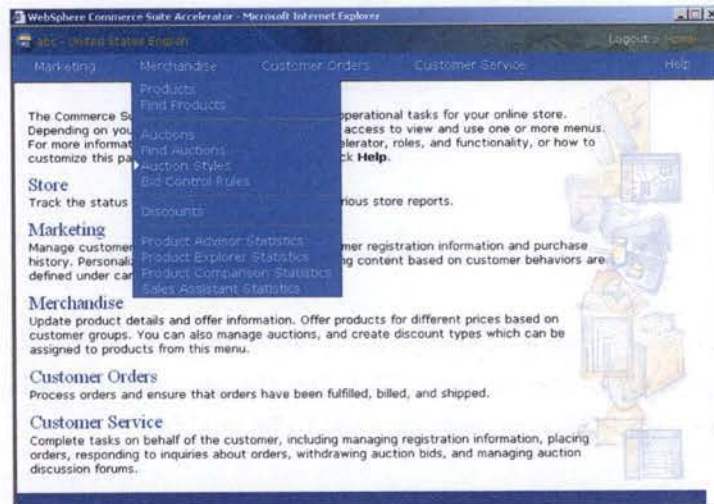
Figure E.3: BAT container: the tools UI center

dynamic list (basic element) The dynamic list is a sortable list element, with navigation controls (Previous and Next buttons) to flip through data. There are also buttons to the right that define actions available for the selected items.

calendar (basic element) The calendar control is a visual control for choosing a specific day, month, and year.

slosh bucket (basic element) The term "slosh bucket" covers exactly the same concept as that developed in [BG02] under the name of "selectable list".

dynamic tree (basic element) A dynamic tree is very similar to a classical tree where nodes can be collapsed or extended.

Figure E.4: BAT basic element: the dynamic list

BAT uses registries during startup and run time in order to configure itself. These registries contains XML files that are used to configure the UI elements. These files are not supposed to be modified.

E.2   Advantages and limits

Without going any further in the functioning of the Browser-based Application Toolkit, it appears that, although it might be very useful and easy to use for a user wanting to create quickly a simple user interface, BAT is definitely not flexible enough to meet the expectations exposed in section 4.3. As a reminder, these expectations were resistance to change, ease of use and no limitations in the choice of GUI components.

First of all, the produced GUI does not seem to accommodate very well late changes. Once the chain of panels is established, redefining it or changing the structure of a panel equals to changing and redefining almost everything.

Moreover BAT allows the user to use only predefined UI elements. The range of existing GUI elements is nevertheless very wide (text areas, trees, dialogs, labels, etc.) The choice is restricted to only 8 of these elements,

which can appear to be really insufficient.

The user might also think that, using BAT, he/she will be describing his graphical user interface in a XML format, which is absolutely not the case. If BAT uses XML effectively, it is only to set some parameters for the UI elements itself. The user actually never sees these XML files. BAT uses XML only internally and the user is not supposed to configure the framework by adding some XML files or modifying them.

However BAT seems a great tool for who wants to rapidly build quickly a simple interface because it is not mandatory to be used to any graphical user interface's language techniques to produce good results. The only drawback is that it is not powerful and flexible enough.
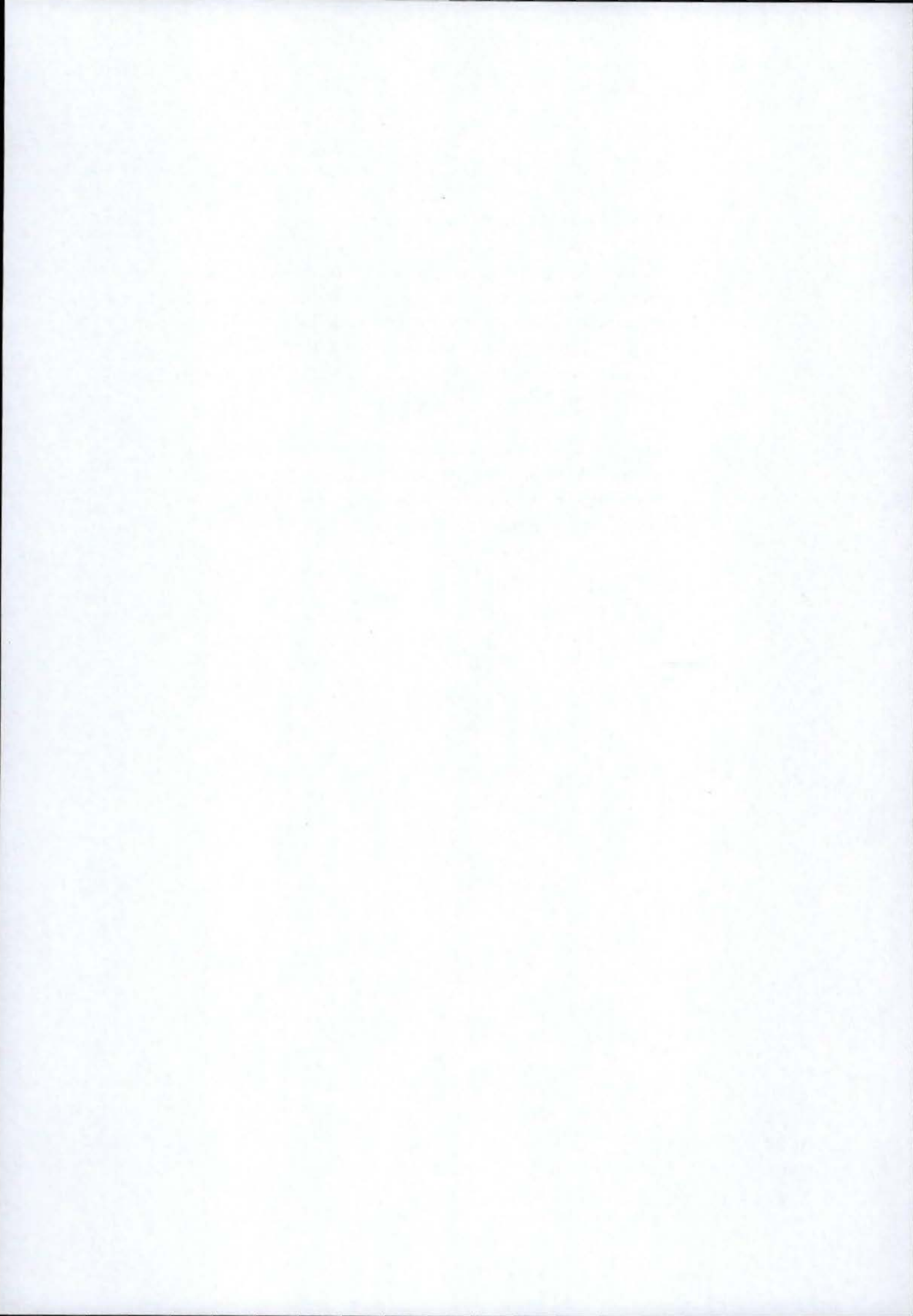
Appendix F

JEasy

JEasy uses Swing components and XML files: all GUI components are stored in a XML file (located in a special directory). [JEa03]

F.1 Principles

F.1.1 Java 2 Swing Components

Swing is SUN's library for building user graphical interfaces in Java.

F.1.2 JEObjects

To almost each Swing component, called JObjects, corresponds one JE component. For example, the JEMenu corresponds to the Swing JMenu, the JEButton is for the JButton, etc. JEObjects read some properties out of the XML file and create JObjects.

F.1.3 XML

All the information concerning the hierarchy (which object contains which object) is part of the XML file. This way, the JEObjects are able to add themselves together to the complete GUI.

F.1.4 Messages

Messages are associated to components which hold data entries. Two methods handle these messages. The *getMessage()* method is responsible for giving back all entries - in other words, all information of a panel - in an

XML format while the *putMessage()* method does exactly the opposite: to fill a panel with information coming from an XML file. This is illustrated in Figure F.1.

Form: PA_Address

Message: MS_Address <->PA_Address

<MS_Address>
<Name>Miller</Name>
<Prename>Jack</Prename>
<City>New York</City>
<Country>USA</Country>
</MS_Address>
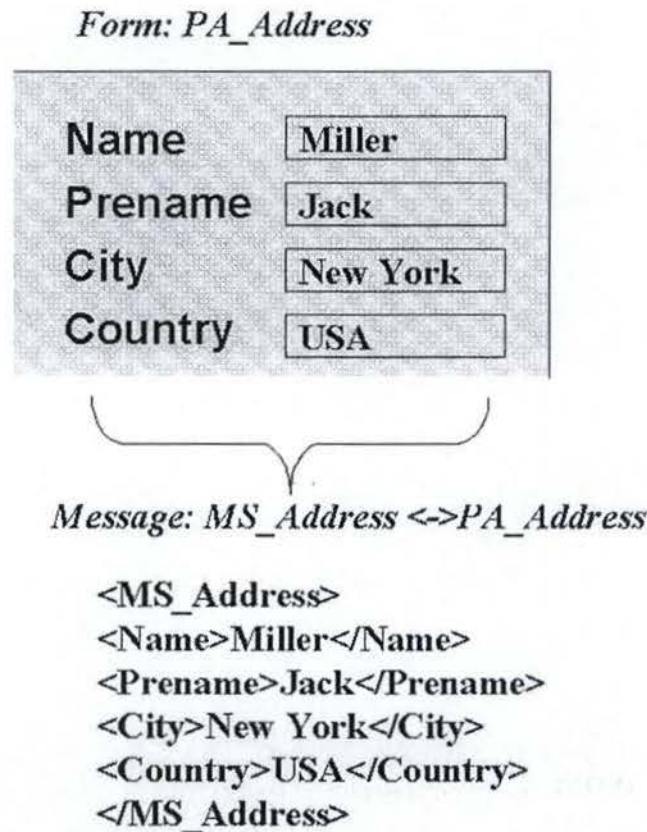
Figure F.1: Messages in JEasy

F.1.5   Repository

The repository contains an XML file with all the objects, properties and relations. JEObjects read these entries at the start up of the program and create all the wished Swing components. Figure F.2 shows the repository's interface of JEasy, from where it is possible to choose the JEObjects.
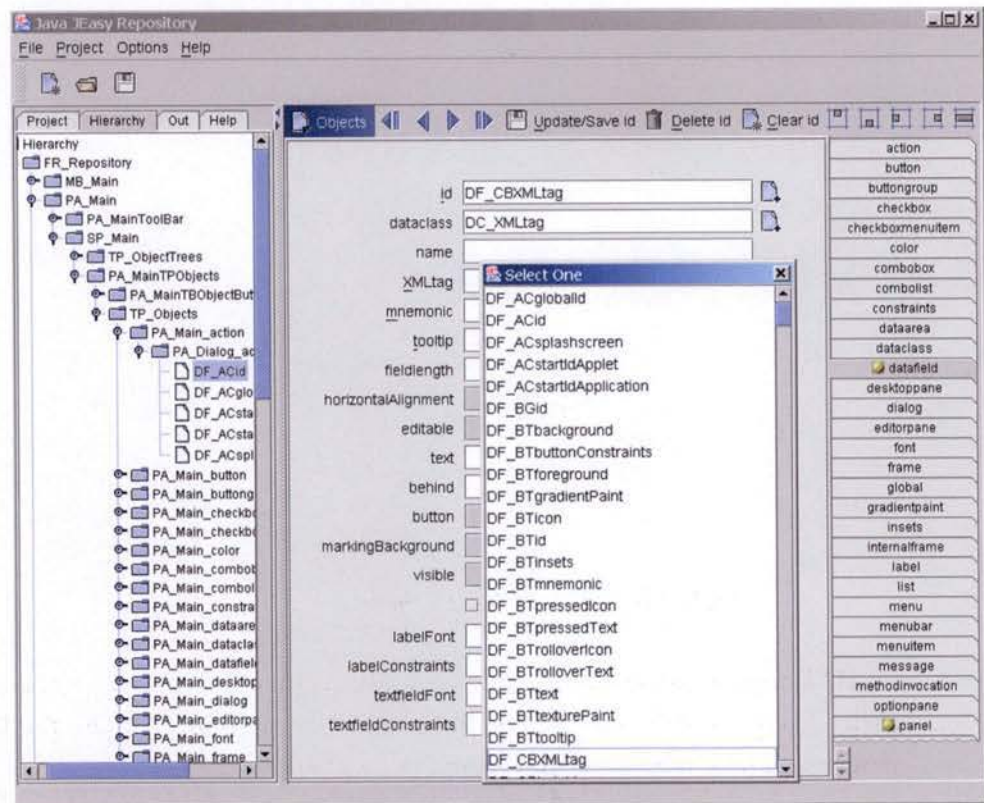
Figure F.2: Interface of JEasy

F.2   Advantages and limits

The conclusion about JEasy will be, more or less, quite the same as the one made for BAT. The user does not write his interface in an XML format: JEasy is in charge of creating the XML file it needs internally and the user is, again, not supposed to work on it.

The range of GUI elements proposed by JEasy seems however wider than the 8 elements of BAT. Indeed, there is a JEObject for almost every Swing component.

Finally, JEasy suffers from the same lack of flexibility than BAT. Once the user has chosen the JEObjects he/she wants to use, there is no more u-turn allowed.

Bibliography

[AIS+77]   Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max
           Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern
           Language.* Oxford University Press, New York, 1977.

[BG02]   Jean Baltus and Nicolas Gilson. *Pertinence of Design Pat-
           terns in Object-Oriented Software Development.* Master's thesis,
           FUNDP, 2002.

[Bla01]   Arnaud Blandin. *Castor XML Source Code Generator - User
           Document.* Exolab.org, July 2001.

[BMR+96]   Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Som-
           merlad, and Michael Stal. *Pattern-oriented Software Architec-
           ture, A System of Patterns.* Wiley, 1996.

[Bor03]   Borland. *User Guide for Together ControlCenter and To-
           gether Solo.* May 2003. `http://info.borland.com/techpubs/`
           `together`.

[Bou03a]   Ronald Bourret. *XML and Databases.* January 2003. `http:`
           `//www.rpbourret.com/xml/XMLAndDatabases.htm`.

[Bou03b]   Ronald Bourret. *XML Database Products.* March 2003. `http:`
           `//www.rpbourret.com/xml/XMLDatabaseProds.htm`.

[BR]   Isabelle Borne and Nicolas Revault. Comparaison d'outils de
           mise en oeuvre de design patterns. In *Revue L'Objet,* volume
           5, nr. 2, pages 243-266, 1999. `http://www-poleia.lip6.fr/`
           `~revault/papers/lObjet-V5-2-99-DP-IBNR.ps.zip%`.

[Bur01]   Didier Burton. *Software Architecture.* 2001.

[Cha] Don Chamberlin. Xquery: An xml query language. In *IBM Systems Journal*, volume 41, nr. 4, pages 597-615, 2002.

[Cod70] Dr. Edgar F. Codd. *A relational model of data for large shared data banks.* Technical report, Communications of the ACM, 1970.

[Des] Objects By Design. *UML Products by Platform.* http://www. objectsbydesign.com/tools/umltools_byPlatform.html.

[EWJ99] David A. Epstein, Sanjiva Weerawarana, and Daniel Jue. *Bean Markup Language from IBM alphaWorks : Enabling active content.* December 1999. http://www-106.ibm.com/ developerworks/xml/library/x-bml.

[Fow02] Martin Fowler. *Enterprise Application Architecture.* Addison-Wesley, draft edition, 2002.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[Gra99] Mark Grand. *Graphic Java, Mastering the JFC.* Addison-Wesley, 1999.

[Gra02] Mark Grand. *Patterns in Java, Volume 1, A Catalog of Reusable Design Patterns Illustrated with UML.* Wiley, 2002.

[Hab] Naji Habra. *Cours de génie logiciel.* Institut d'Informatique, FUNDP, Namur.

[Hol99] Allen Holub. *Problems with Swing's new XMLOutputStream class.* August 1999. http://www.javaworld.com/javaworld/ jw-08-1999/jw-08-toolbox.html.

[Hon] Dr. Shuguang Hong. *Introduction to Database Management Systems.*

[IG01] Intellor-Group. *XML Database Trends and Influences.* 2001. Research Summary.

[Ini] The XML:DB Initiative. http://www.xmldb.org.

[JBR99]   Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[JEa03]   JEasy. *A framework for JAVA applications using XML*. May 2003. http://www.jeasy.de.

[Joh97a]   Mark Johnson. *A walking tour of JavaBeans, What JavaBeans is, how it works, and why you want to use it*. August 1997. http://www.javaworld.com/javaworld/jw-08-1997/jw-08-beans_p.html.

[Joh97b]   Mark Johnson. *"Double Shot, Half Decaf, Skinny Latte" : Customize your Java, How to tailor JavaBeans to fit your application*. September 1997. http://www.javaworld.com/javaworld/jw-09-1997/jw-09-beans_p.html.

[Joh98]   Mark Johnson. *Do it the "Nescafé" way : with freeze-dried JavaBeans, How to use object serialization for bean persistence*. January 1998. http://www.javaworld.com/javaworld/jw-01-1998/jw-01-beans_p.html.

[Joh99a]   Mark Johnson. *Bean Markup Language, Part 1, Learn the ABCs of IBM's powerful JavaBeans connection*. August 1999. http://www-106.ibm.com/developerworks/java/library/j-bean-markup1/.

[Joh99b]   Mark Johnson. *Bean Markup Language, Part 2, Create event-driven applications with BML*. October 1999. http://www.javaworld.com/javaworld/jw-10-1999/jw-10-beans_p.html.

[Joh99c]   Mark Johnson. *XML JavaBeans, Part1, Make JavaBeans mobile and interoperable with XML*. February 1999. http://www.javaworld.com/javaworld/jw-02-1999/jw-02-beans_p.html.

[Lab02]   IBM Toronto Laboratory. *Browser-based Application Toolkit Programmer's Guide*. May 2002. http://www.alphaworks.ibm.com/aw.nsf/download/bat.

[Mei96]   M. Meijers.   *Tool support for object-oriented design patterns.*
            Master's thesis, Utrecht University, 1996.

[Mil99]   Philip Milne. *Using XMLEncoder.* August 1999. `http://java.`
            `sun.com/products/jfc/tsc/articles/persistence4/`.

[MW99]   Philip Milne and Kathy Walrath. *Long-Term Persistence for
            JavaBeans.* August 1999. `http://java.sun.com/products/`
            `jfc/tsc/articles/persistence/`.

[oEE90]   Institute of Electrical and Electronics Engineers. *IEEE Stan-
            dard Computer Dictionary: A Compilation of IEEE Standard
            Computer Glossaries.* New York, 1990.

[Pop01]   Paul Pop. *A survey of three approaches to the automation of De-
            sign Patterns.* Computer and Information Science Department -
            Linkopings Universitet, October 2001.

[UML]   Uml en français. `http://uml.free.fr`.

[Vli98]   John Vlissides.   *Pattern Hatching, Design Patterns Applied.*
            Addison-Wesley, 1998.

[Waya]   Rick Wayne. The real mccoy. In *Software Development Maga-
            zine*, pages 26-28, January 2003. Article about XML Databases.

[Wayb]   Rick Wayne. What'll it be? In *Software Development Magazine*,
            pages 26-28, February 2003. Article about XML Databases.

[WD99]   Sanjiva Weerawarana and Matthew J. Duftler. *Bean Markup
            Language (version 2.3), User's Guide.* September 1999. `http:`
            `//www.alphaWorks.ibm.com/formula/bml`.

[Wil00]   Kevin Williams. *Professional XML Databases.* Wrox Press, 2000.

[WRL02]   R. Allen Wyke, Sultan Rehman, and Brad Leupen. *XML Pro-
            gramming.* Microsoft Press, 2002.