On Usability Pattern Documentation: An XML-based Approach

Faridul Islam

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

October, 2003

Your file Votre référence
ISBN: 0-612-91048-2
Our file Notre référence
ISBN: 0-612-91048-2

ABSTRACT

On Usability Pattern Documentation: An XML-based Approach

Faridul Islam

There are several ways to document patterns. Different formats have been proposed by different authors for capturing the best practices and proven solutions; i.e. patterns. The Alexandrian format of patterns was presented in a fairly informal, narrative style. On the other hand, the Gang-of-Four format of patterns was presented in much finer grained fashion by decomposing each pattern into many sections. The structure of pattern documentation depends on several factors. Each author has his own preferences. Different subject matters may influence the structure. For example, more technical information may require patterns with more structure. Different users may require different structures as well. Novice users may prefer a more prosy style while the experienced users may prefer a more structured one. Thus, there was a need to compromise for having a consistent structure so that patterns are clear, understandable, and reusable.

This thesis explores some of the most popular formats of patterns, proposed by different individuals or communities. After a thorough study of the strength and weaknesses of the existing ones, a new comprehensive format of pattern documentation, comprising seven elements, has been developed. Besides enhancing productivity, the proposed format reduces the communication gap among the three key professional groups; i.e. Patterns Writers, Usability Experts and Software Developers. To support the selection of a pattern for a given context, a classification scheme, which organizes patterns according to granularity, functionality, and structural principles, is also proposed. The study concludes with the development of syntax and semantics of a markup language, named Usability Pattern Markup Language (UPML), based on eXtensible Markup Language (XML).

ACKNOWLEDGEMENTS

I would like to expresses my deep sense of gratitude to my thesis supervisor, Dr. Seffah Ahmed, Assistant Professor, Department of Computer Science, Concordia University, who has supported my work with his patience and kindness. He has given me valuable suggestions, remarks, and clear guidelines that contributed to the success of this thesis.

I am highly glad of being a member of the Human-Centered Software Engineering (HCSE) group at Concordia University and greatly enjoy the new experience that I have gathered during my master' studies – a new era of pattern supportive design, task oriented modeling, and overall integrating the User Centered Design (UCD) concepts in the process of software engineering and re-engineering. I am also thankful to Dr. Seffah Ahmed for giving me the opportunity to serve the HCSE Laboratory as Lab Manager during the last summer.

I am also grateful to Ms. Pai, Hsueh-Ieng for her illuminating discussion and support in writing XML schema for the Usability Pattern Markup Language (UPML). I would like to thank my friends, Kedar Chandra Das and Mahmud Hossain, for their time to help me with proof reading of this thesis.

Finally, I would like to thank my family for their love and support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

Over the past few decades, there have been significant interests in using patterns and pattern languages by different disciplines; e.g. Architecture and Civil Engineering, Chemistry, Biology, Bio-Informatics, Genetic Engineering etc. In fact, the notion of patterns and pattern languages were originated in the late 1970's [Alexander+77] in the civil engineering and urban planning domains as an approach to design buildings, roadways, and towns. After the enormous success of Alexander's works on patterns in civil engineering domain, some usability experts and pattern authors have been trying to promote the use of patterns into software engineering domain (including software architecture, design, and development) since the last decade. In recent years, there have been a series of seminars, symposiums on the usability of patterns in object-oriented software architecture and development. However, relying on the popularity of patterns, many individuals and communities across the globe have been contributing to this field in a scattered manner. As a result, pattern collection became heterogeneous and users need to spend a significant amount of time just to pick the right pattern for their use from this heterogeneous collection. Since software designers and developers are used to work under a tight schedule, they need patterns in a clear and understandable format or ready-to-go state along with a proper organization.

The thesis is an effort to reduce the communication gap among different professional groups, who are interested in patterns and pattern languages, by introducing a methodical approach, in general, and a generalized format, in particular for pattern documentation in a consistent manner. It also provides a scheme for pattern classification. Last but not the least, this thesis will provide syntax and semantics of the Usability Pattern Markup Language (UPML) 1.0 specification, based on the eXtensive Markup Language (XML) notation, in order to write the future patterns.

1.1 A Brief History of Patterns Movement

The movement of patterns originates in the work of a building architect named *Christopher Alexander* during the late 1970s. He started the patterns movement through his works, especially by writing two books, *"A Pattern Language"* [Alexander+77] and *"A Timeless Way of Building"* [Alexander79] which, in addition to giving examples, described his rationale for pattern documentation.

The pattern movement was very quiet until 1987 when patterns appeared again at an OOPSLA conference in Orlando organized by Kent Beck and Ward Cunningham. Since then, many papers and presentations have appeared, authored by a large number of people; i.e. Grady Booch, Richard Helm, and Erich Gamma, Kent Beck, etc. In 1993, the formation of "Hildside Group" by Beck, Cunningham, Coplien, Booch, Johnson and others was the first step forward in evolving a pattern community. From 1987 to 1995, many periodicals, featured articles were published directly or indirectly relating to patterns. In 1995, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (affectionately known as Gang-of-Four) published *"Design Patterns: Elements of Reusable Object-Oriented Software"* [Gamma+95], which has been creating a great interest in using patterns within the object-oriented software developers community.

In the CHI'1997 Workshop, the Human Computer Interaction (HCI) community formed a forum for vigorous discussions on patterns and pattern languages for user interface (UI) designers or even for the software developers. Since then many groups devoted themselves to develop HCI patterns and pattern languages for design [Tidwell97, Coram+98, Welie+00, Pemberton+99, Granlund+01, Brochers00]. The most recent and remarkable addition to this movement is done by Alan S. and James R. T. "Design Patterns Explained – a new Perspective on Object-Oriented Design", in 2002 published by Addison-Wesley [Alan+02].

1.2 Definition of Pattern and Basic Terminology

A **pattern** expresses a relationship among a certain context, a problem and a solution. A **context** is the environment, situation, or interrelated conditions within the scope of which something exists. A **problem** is an issue that needs to be investigated and resolved, and is typically constrained by the context in which it occurs. A **solution** is a response to the problem in a context that helps resolving the issue(s). It also shows the rationale behind the solution of a problem.

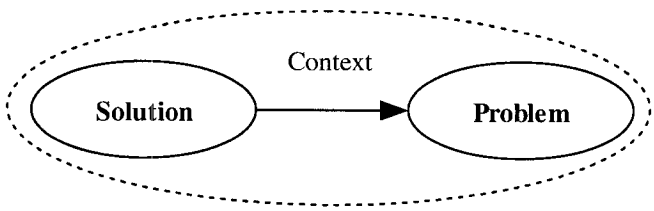


Figure 1.1: The Pattern domain

Therefore, if the context is changed, solution could be different for the same problem that means different Pattern.

Christopher Alexander, the pioneer in this field, has defined **pattern** as: "*A recurring solution to a common problem in a given context and system of forces*" [Alexander+77]. Therefore, the **recurring use** or **reusability** is an important factor to validate the pattern itself. In defining **pattern** as: "*Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution*" [Alexander+77] he also identified, which are the fundamental properties of a pattern.

A **pattern language** formalizes the definition of a pattern by providing a vocabulary that helps the users in understanding patterns. It also provides a notation for writing patterns in a uniform fashion [Appleton00]. Some of the defining characteristics of a pattern language inspired by software engineering principles are: abstraction, anticipation of

change, generality, semantic-relationship (among patterns) that may lead to inheritance (of the solution of one pattern by the other, in whole or in part), and consistency (that one pattern should not provide a solution if it conflicts with the others).

James O. Coplien has defined **pattern language** as a structured collection of patterns that build on each other to transform needs and constraints into an architecture. It is not a programming language in any ordinary sense of the term, but is a prose document whose purpose is to guide and inform the designer [Coplien+95].

In this thesis, **usability patterns** have been characterized as usability-related software design patterns including design patterns, user interface design patterns, HCI patterns, etc. Well, the difference between design patterns and usability patterns can be defined as: *Usability patterns are concerned mostly with an **external** point of view on a software system, whereas design patterns in general are concerned with an **internal**, more structural, point of view.* Moreover, the study will discuss only the patterns that are related to the software engineering domain (including software architecture, design, and development). Therefore, software designers and developers are expected as its potential users. In the remaining parts of this thesis text, the term "users" will simply refer to the above mentioned groups or professionals.

Besides the formal definitions, patterns and pattern languages are means to describe the best practices and experience of the experts in a way such that the users (including novice users) can reuse this experience easily [Linda98]. As we all know, in any Science and Engineering discipline the fundamental issue is to communicate and share experience with others in order to achieve the great mutual benefits, and for that reason a common or understandable vocabulary is necessary. The goal of patterns within the software engineering community is to create a body of literature to help software designers and developers to resolve recurring problems encountered throughout the life cycle of the software development [Alexander79]. Patterns help to create a shared language for communicating insight and experience about these problems and their solutions. Formally codifying these solutions and their relationships help the patterns writers successfully capture the body of knowledge, which defines an understanding of good

architectures. Forming a common pattern language for conveying the structures and mechanisms of the architectures allows patterns writers to intelligibly reason about them. The primary focus of this thesis is more on creating a common notation of pattern documentation in an understandable manner rather than on investigating of its inherent technology. Therefore, a **pattern** may be defined as: "*A literary for capturing the wisdom and experience of expert designers, and communicating it to novice*".

Are Patterns Framework?

Patterns capture the static and dynamic structures and collaborations of successful solutions to problems that arise when developing application in a particular domain. A Framework is an integrated set of components that collaborated to provide a reusable architecture for a family or related applications,

Patterns and frameworks are highly synergistic (i.e. neither is subordinate). Patterns can be characterized as more abstract description of frameworks, which are implemented in a particular language (i.e. Pattern descriptions are often independent of programming language or implementation details). In general, sophisticated frameworks embody dozens of patterns and patterns are often used to document frameworks.

Are Patterns Components?

Patterns mainly used to make the design decisions during the design phase, whereas Components are used during the implementation phase. Since patterns are the abstract description of design solution, it avoids all the implementation details. But, components are basically a ready to use module in a particular programming language.

Are Patterns just Rules?

Rules are not commonly supported by a rationale, nor put in context. A rule may be part of the solution in a pattern description, but a rule solution is neither sufficient nor necessary. Patterns are not designed to be executed or analyzed by computers, as one might imagine being true for rules.

1.3 Significance of Patterns

There are several characteristics that make patterns useful. Some of those are as below:

Formality

Patterns are formal. There have been several efforts [Alur+01, Gamma+94, Bryan01, Appleton00, IEEE87, IEEE98, IBM99a] that provide strategies and suggest guidelines for pattern documentation. However, one of the limitations of these guidelines is that they make broad use of natural language for description. The guidelines are less formal and vague. Although Frequently Asked Questions (FAQs) help in answering user questions, they are usually focused on a single topic, which is often specific to a particular technology. Patterns are more formal in their approach, and exist at a higher level of abstraction than the strategies or guidelines. Patterns offer various advantages over guidelines [Griffiths+01] and are anticipated to play an essential role in information technology [IBM99a]. Nevertheless, patterns do not attempt to necessarily replace the FAQs, strategies, or guidelines. Rather, they should be considered as a key complement to the overall initiative.

Nomenclature

Patterns (and therefore, the concepts they represent) are assigned names. This expedites further discussion, analysis, and reference of previously localized concepts.

Practicality

Patterns provide practical "ready-to-go" solutions. A pattern describes "good" practical solutions to a common problem within a certain context by describing the invariant aspects of all those solutions. Given a problem, patterns include a compact, focused, complete, and straightforward way of describing a solution. Since they provide the consequences of applying that solution, the users can decide and act upon in a timely manner if the solution is applicable to their situation.

Experience

Patterns form an "expert" system in practice. Patterns, when well-defined and organized, are more than a mere static disjoint "collections" of recipes. Patterns are tried-and-tested ways to deal with problems that recur. It is expected that those who have experience in a particular field of knowledge will have certain localized solutions to these problems. As a result, they recognize a problem to be solved and know which solution needs to be applied in a particular situation. A pattern describes this localized experts' knowledge and states the problem, context, and solution, so that others with less experience can be benefited. In this sense, patterns themselves can be considered as a "smart FAQ" or an "expert system" that encapsulates the knowledge and experience of the author. This enables them to be used as a *knowledge base.*

Re-Usability

A pattern presents a higher-level view of the same problem inflicting often multiple industries and provides a solution for it. It may also be connected to other patterns in existence (in the same or other catalogs) for whole or in part of its solution (inheritance). Patterns thus encourage re-use.

Abstract, Modular Framework

Complex problems are often composed of several steps that need to be dealt with independently and then combined to arrive at a solution. Patterns represent these steps at a high-level via "intelligent" distribution and allocation of responsibilities. They provide a framework that works in unison to fulfill a given task.

Community

Patterns help a broader community. Patterns communicate solutions to a community of architects, designers, and engineers, who make use of those at different levels and for different purposes. The goal of the pattern community is to build a re-usable body of knowledge to support design and development in general.

1.4 Objectives and Scope of the Thesis

The main objective of this thesis is to investigate some of the most popular formats of pattern documentation with the goal to establish a generalized format based on XML notation. In order to achieve the goal, the following questions need to be addressed:

1. What are the patterns, pattern languages and usability patterns?
2. What are the popular formats currently being used for pattern documentation?
3. What are the weaknesses and strengths of the above formats?
4. What are the significances of XML for pattern documentation?
5. What are tools or languages for writing future patterns?

This study will also provide the directions as well as scope for further studies.

1.5 Thesis Organization

After the introductory chapter (Chapter 1), this thesis text is organized as follows:

Chapter 2 provides a literature review of some of the most popular formats for pattern documentation, proposed by different individuals and/or communities. It also explores XML as a viable solution for documenting the future patterns, and motivates us to develop the Usability Pattern Markup Language (UPML) 1.0 using XML notation.

Chapter 3 discusses about our two major contributions; (i) A Classification Scheme of Patterns and (ii) A 3D-Reference Model of Pattern Properties along with the definition of the terminology. This chapter also discusses about the weaknesses of the formats under study and finally recommends a generalized format of pattern documentation.

Chapter 4 defines the syntax and semantics of the UPML 1.0 specification. First it lists the UPML modules and their elements and then lists the basic properties of elements and attributes, including conditions on their data types and enumeration. This chapter also provides the definitions of individual elements along with the details of corresponding attributes, sub-elements, and examples.

Chapter 5 summarizes the major contributions of the thesis work as well as some of avenues for future studies.

Chapter 2

An Analysis of Some Popular Formats for Pattern Documentation

This chapter mainly provides a literature review of some of the most popular formats of pattern documentation (section 2.1 through section 2.7). Section 2.8 shows the significance of XML for pattern documentation as well as the motivation for the Usability Pattern Markup Language (UPML) specification. Section 2.9 summarizes the discussions of this chapter and bridges it with the ongoing chapter(s).

There are different formats (with a wide range of properties, elements and attributes) for pattern documentation, which are suggested by different individuals or communities. Some of the most popular formats are as follows:

2.1 Christopher Alexander

Christopher Alexander, the father of the patterns and pattern languages movement, has created his pattern format as below:

Each pattern has the same format. First, there is a picture, which shows an archetypal example of that pattern. After the picture, an introductory paragraph...sets the context ... explaining how it helps to complete certain larger patterns. Then there are three diamonds.... After the diamonds ... a headline, in bold type ... gives the essence of the problem in one or two sentences. After the headline comes the body of the problem. This is the longest section. It describes the empirical background of the pattern, the evidence for its validity, the range of different ways the pattern can be manifested in a building, and so on. Then, again in bold type, like the headline, is the solution, the heart

of the pattern, ... in the stated context. This solution is always stated in the form of an instruction so that one knows exactly what one needs to do to build the pattern.

Afterwards, a diagram of the solution followed by another three diamonds and a paragraph are introduced to complete the pattern.

Each solution is stated in such a way that it gives the essential field of relationships needed to solve the problem, but in a very general and abstract way so that the users can solve the problem in their own way ... Alexander has tried to write each solution in a way which imposes nothing on the users. It contains only those essentials, which cannot be avoided if the users really want to solve the problem. In this context, Alexander has tried, in each solution, to capture the invariant properties common to all (solutions of) the problem.

An Example of Alexandrian Format [Alexander+77]

251 DIFFERENT CHAIRS

. . . when you are ready to furnish rooms, choose the variety of furniture as carefully as you have made the building, so that each piece of furniture, loose or built in, has the same unique and organic individuality as the rooms and alcoves have—each different, according to the place it occupies—SEQUENCE OF SITTING SPACES (142), SITTING CIRCLE (185), BUILT-IN SEATS (202).

✣ ✣ ✣

People are different sizes; they sit in different ways. And yet there is a tendency in modern times to make all chairs alike.

Of course, this tendency to make all chairs alike is fueled by the demands of prefabrication and the supposed economies of scale. Designers have for years been creating "perfect chairs"—chairs that can be manufactured cheaply in mass. These chairs are made to be comfortable for the average person. And the institutions that buy chairs have been persuaded that buying these chairs in bulk meets all their needs.

But what it means is that some people are chronically uncomfortable; and the variety of moods among people sitting gets entirely stifled.

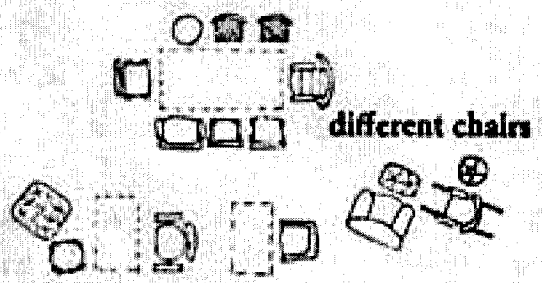
Obviously, the "average chair" is good for some, but not for everyone. Short and tall people are likely to be uncomfortable. And although situations are roughly uniform—in a restaurant everyone is eating, in an office everyone is working at a table—even so, there are important distinctions: people sitting for different lengths of time; people sitting back and musing; people sitting aggressively forward in a hot discussion; people sitting formally, waiting for a few minutes. If the chairs are all the same, these differences are repressed, and some people are uncomfortable.

What is less obvious, and yet perhaps most important of all, is this: we project our moods and personalities into the chairs we sit in. In one mood a big fat chair is just right; in another

mood, a rocking chair; for another, a stiff upright; and yet again, a stool or sofa. And, of course, it isn't only that we like to switch according to our mood; one of them is our favorite chair, the one that makes us most secure and comfortable; and that again is different for each person. A setting that is full of chairs, all slightly different, immediately creates an amosphere which supports rich experience; a setting which contains chairs that are all alike puts a subtle straight jacket on experience.

Therefore:

Never furnish any place with chairs that are identically the same. Choose a variety of different chairs, some big, some small, some softer than others, some rockers, some very old, some new, with arms, without arms, some wicker, some wood, some cloth.



✣ ✣ ✣

Where chairs are placed alone and where chairs are gathered, reinforce the character of the places which the chairs create with POOLS OF LIGHT (252), each local to the group of chairs it marks. . . .

2.2 Gang-of-Four

The format defined by the Gang-of-Four (Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides) is described in the first chapter of their book [Gamma+95], which comprise four essential elements:

1. *The **pattern name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two. Naming a pattern immediately increases our design vocabulary. It helps us design at a higher level of abstraction. Having a vocabulary for patterns helps us to communicate among our colleagues, in our documentation, and even to ourselves. It makes easier to think about designs and to communicate with them and their trade-offs to others. Finding good names has been one of the hardest parts of developing our catalog.*

2. *The **problem** describes when to apply the pattern. It explains the problem and its context. It might describe specific design problems such as how to represent algorithms as objects. It might describe class or object structures that are symptomatic of an inflexible design. Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern.*

3. *The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution doesn't describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations. Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements solves it.*

4. *The **consequences** are the results and trade-offs of applying the pattern. Although consequences are often unvoiced when we describe design decisions, they are*

critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern.

The consequences for software often concern space and time tradeoffs. They may address language and implementation issues as well. Since reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability. Listing these consequences explicitly helps us to understand and evaluate them.

Each pattern in the Gang-of-Four's collection is described following a consistent format:

Pattern Name and Classification:

What is the pattern called? Is the pattern creational, structural, or behavioral?

Intent:

What problem does this pattern solve?

Also Known As:

What are other names for this pattern?

Motivation:

What is an example scenario for applying this pattern?

Applicability:

When does this pattern apply?

Structure:

What are the class hierarchy diagrams for the objects in this pattern?

Participants:

What are the objects that participate in this pattern?

Collaborations:

How do these objects interoperate?

Consequences:

What are the tradeoffs of using this pattern?

Implementation:

Which techniques or issues arise in applying this pattern?

Sample Code:

What is an example of the pattern in source code?

Known Uses:

What are some examples of real systems using this pattern?

Related Patterns:

What other patterns from this pattern collection are related to this pattern?

The Gang-of-Four format has a profound effect on the object-oriented community, but it provides little help for pattern writers. The list of elements created by this thoughtful group of four has contained the essentials for good pattern writing and this thesis work will follow their notations closely for pattern documentation.

Gang-of-Four authors have organized their design patterns based on two criteria: purpose and scope, which is further elaborated in chapter 3. This kind of organization has certainly motivated patterns writers to organize their patterns. However, it faces a serious problem of overlapping patterns among different categories as there are no concrete definitions of the boundary criteria. Therefore, we need to make sure that the set of criteria for pattern classification are clear and concrete. It should not be only applicable to a particular collection, but also need to be applicable for both the existing and upcoming patterns. Otherwise, classification of patterns may create more disagreement rather than agreements.

An Example of Gang-of-Four Format [Gamma+95]

Abstract Factory (Please see **Appendix A** for the details).

2.3 Common Ground (Jenifer Tidwell)

Jenifer Tidwell has formed a platform named "Common Ground" in order to serve the Human-Computer Interface (HCI) community by providing a collective collection of HCI Design Patterns. She felt that the HCI community will be greatly benefited by using

patterns as 40% codes of software applications are related to the Interface Design or simply Graphical User Interface (GUI).

Mostly her work in pattern language, specially pattern format, is intended to form an Alexandrian pattern language, as found in Christopher Alexander's book *"A Pattern Language"* and not a catalog such as is found in the book *"Design Patterns"* of Gang-of-Four. Like other such pattern languages, it does not break new theoretical ground or present innovative new techniques -- it's more likely that you have seen examples of every pattern here. Instead, it captures ordinary design wisdom in a practical and understandable manner.

An Example of Tidwell Format [Tidwell99]

Go Back to a Safe Place

Examples:

- The "Home" button on a Web browser
- Turning back to the beginning of a chapter in a physical book or magazine
- The "Revert" feature on some computer applications

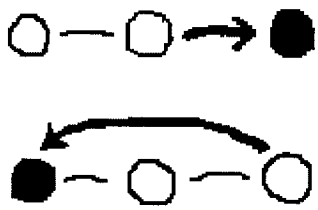
Context: The artifact allows a user to move through spaces (as in Navigable Spaces), or steps (as in Step-by-Step Instructions), or a linear Narrative, or discrete states; the artifact also has one or more checkpoints in that set of spaces.

Problem: How can the artifact make navigation easy, convenient, and psychologically safe for the user?

Forces:

- A user that explores a complex artifact, or tries many state-changing operations, may literally get lost.
- A user may forget where they were, if they stop using the artifact while they're in the middle of something and don't get back to it for a while.
- If the user gets into a space or a state that they don't want to be in, they will want to get out of it in a safe and predictable way.
- The user is more likely to explore an artifact if they are assured that they can easily get out of an undesired state or space; that assurance engenders a feeling of security.
- Backtracking out of a long navigation path can be very tedious.

Solution: **Provide a way to go back to a checkpoint of the user's choice.** That checkpoint may be a home page, a saved file or state, the logical beginning of a section of narrative or a set of steps. Ideally, it could be whatever state or space a user chooses to declare as a checkpoint.



Resulting Context: Go Back One Step is a natural adjunct to this pattern, and is often found along with it. For non-Narrative use, Interaction History is useful too, almost to the point of making Go Back to a Safe Place unnecessary: it may actually help a "lost" user figure out where they are, for instance, or remind an interrupted user of where they are and what they've done.

2.4 Amsterdam Collection (Martijn Van Welie)

The Welie format is closely related to the format of Christopher Alexander. In addition to Alexandrian format, his format describes when and why a pattern will be used in a simple text/narration in order to enhance understandability and learnability of a pattern. Moreover, his format highlights some of the known uses so that the pattern users can easily find out the appropriate place to use by comparing with the existing examples, which intends some of the Gang-of-Four's properties.

An Example of Welie's Format [Welie03]

Breadcrumbs

Home > Products > Flash

Macromedia Flash MX
Product Overview

From www.macromedia.com

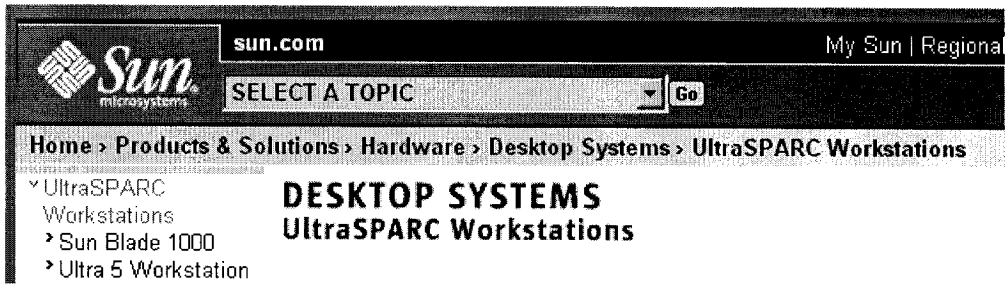
Problem The users need to know where they are in a hierarchical structure

Use when Sites with a large hierarchical information structure, typically more than three levels deep. Such sites are medium to large sized and include shops, catalogs, portals, corporate sites etc. The site has got a main navigation scheme that allows users to traverse the hierarchy. Users may want to jump several steps back instead of following the hierarchy. Users may be unfamiliar with the hierarchical structure of the information. Users may need to know where they can go. Users need to know how they arrived at their current location.

Solution Show the path from the top level to the current page.

Home > NextLevel > NextLevel > **Current page**

The path shows the location of the current page in the total information structure. Each level of the hierarchy is labeled and functions as a link to that level. The current page is marked in order to give the users feedback about where they are

	now. Don't use the current page name in the breadcrumb as the only way to show section title, add a title anyway. The path shows that a top-down path is traversed by using appropriate separators such as > or \ that suggest a downward motion. If the path becomes too long to fit in the designated place, some of the steps can be replaced by an ellipsis e.g. "...". The path is placed in a separate "bar" that preferably spans the entire width of the content area. It is placed close to the content area, preferably above the content area but below the page header.
Why	The bread crumbs show the users where they are and how the information is structured. Because users see the way the hierarchy is structured they can learn it more easily. By making each label a link, the users can quickly browse up the hierarchy. They take up minimal space on the page and leave most of the space for the real content.
More Examples	 This example is taken from Sun's web site and shows the use of bread crumbs in product pages. The path from the top level is visible and the users can go to any of the other higher level product categories.
Known Uses	www.sun.com ; www.useit.com; www.yahoo.com
Related Patterns	This pattern can be combined with most navigation patterns.

2.5 Portland Pattern Repository

The Portland format, so named because its initial three users were all from Portland, Oregon. It is a narrative format, in contrast to the Gang-of-Four format, which captures a

pattern in a series of sections. A pattern of this format contains statements like the following: This set of forces creates this problem, so here's the solution. The pattern takes its name from the solution. Each pattern is a part of a set of related patterns. Higher level patterns resolved forces, which open up new problems with derived forces [Cunningham95].

Actually, Portland format is a fairly direct emulation of Alexandrian format with some simplification in typesetting. Portland Pattern Repository is maintained by Ward Cunningham, who was one of the pioneers of incorporating patterns into software engineering domain.

An Example of Portland Format [Portland95, Cunningham95]

The Portland format basically describes or analyses three major areas in order to solve any problem starting from the initial phase of design. The format uses the following three properties to articulate the best practices:

1. User Decision
2. Task
3. Task Window

User Interface Author: Kent Beck

User Decision:

- Have you collected a set of Stories?
- How do you organize a Story so it can be mapped into a simple and coherent user interface?

Stories give you a lot of raw material from which to mold the user interface. The problem with stories is that they are organized chronologically, while interfaces are organized spatially. We have to break the story into parts, some of which will be spatially closer than others. We need to translate the story from the time domain into the interface domain.

A good user interface has a sense of plot and flow. You naturally follow some path through the parts of the interface, first using one window or kind of window more, then shifting your focus to another. Interfaces that force you to jump back and forth between different styles of interaction (text editing to form fill out to tabular presentation to direct manipulation) violate this sense of easy flow.

Avoiding "interface shock" requires that we put chronologically related parts of the story close together spatially in the interface. At the same time, we should use the best possible presentation and manipulation technique for every part of the story. To start with, we need to identify the "atoms" of the interface so we can begin to find "clumps". *Therefore*:

Make a list of every decision the user has to make during the stories. For each decision, write down the information the user needs to make the decision wisely.

Task:

- Have you listed of User Decisions?
- How do you group decisions into coherent tasks?

One of the marks of a good interface is that everyone feels like they can work the way they want to, that the system does not impose any particular method of solving a problem. Indeed, it is just this feeling that is the primary goal of my whole involvement with patterns- to give users a sense of mastery of their environment.

This feeling of mastery is often at odds with the users' managers' need to feel in control of the users' work. Anyway, we need to design an interface that satisfies both these desires. The user must feel like the system doesn't impose on their prerogatives. Management (who likely decides whether you get paid for this work) must feel that no user can stray too far from productive paths.

Users with more autonomy will not be under the intense scrutiny described above. Even so, we owe these users a debt, a debt of structure. An interface with no structure, no point of view, gives the user no leverage. Such an interface quickly devolves into an exercise in

interface design for the user, through preferences, macros, or even an embedded programming language.

Here is the dilemma in a nutshell. We must structure the interface, because without structure there is no leverage. We mustn't structure the interface, because structure takes away the user's experience of being in control of the machine instead of the other way round.

The artistic "liberating structure" debate arises once again! Short circuiting hours (days weeks months years) of debate, structure liberates, as long as it eliminates unnecessary decisions on one axis in order to open up decisions on another.

You may not be comfortable making these kinds of decisions. How can you possibly know how every user who will ever encounter the system will use it? How can you be sure that you're right, that you aren't restricting important avenues of exploration for the user?

Two answers:

1. The patterns give you a pretty decent chance of being right for most users most of the time, provided you are listening to them.
2. Of course you'll make mistakes. That's no reason not to ship the system. There will always be "power users" who will stretch whatever facilities you provide.

Therefore,

a. Write each User Decision and its required information on an index card.
b. Put cards with the same information touching each other.
c. Put clumps of cards listing similar information close to each other.
d. Each clump of cards is a task.
e. Name it.

Support each Task with a Task Window. Use a Wizard to overlay temporality when the spatial layout becomes confusing.

Task Window:

- Have you found a Task?
- How much of the system do you reflect in a single window?

Interface design tends to fall into one of two camps- the "everything in one window" camp and the "everything in separate windows" camp. Neither of these extremes serves the user of current computer systems.

The "everything in separate windows" is exemplified by early design on the Macintosh. With the extreme limits it had on screen real estate, it was appropriate not to pack too much into any one window. However, it resulted in some software that was hard to use (Finder, ResEdit) because the user spends far too much time managing windows. The need for a separate "Window" pull down menu is a symptom of this problem.

"Everything in the same window" goes too far the other way. Putting all the functionality of a large system (many workstation CAD systems demonstrate this problem) into one window results in a vast sea of choices. The system isn't helping the user structure their time, isn't presenting any flow through the system.

Finding a happy medium is critical to the operation of the interface (and to the viability of the rest of these patterns). How do you tell when you have enough information in a window?

Ask the user to divide their job into identifiable tasks. Create a window for each task. Name the window after the task.

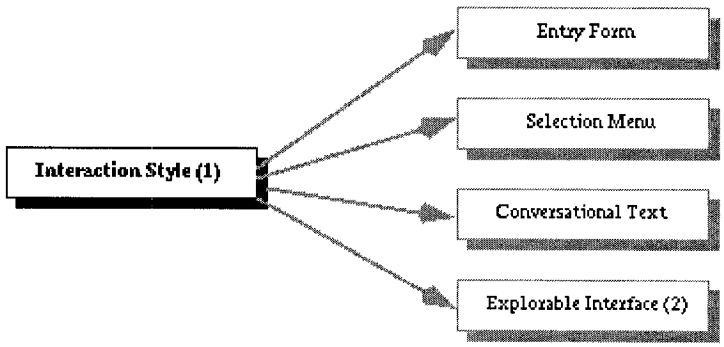
2.6 Todd Coram and Jim Lee

Todd Coram and Jim Lee's works on patterns are documented by using a format which is well known as "Coram Format". This format is basically a modified version of the format described by Christopher Alexander [Alexander+77]. Unlike Alexandrian format (where solution is placed at the ends of the body of a pattern), Coram format places the solution

after identifying the forces. They preface each pattern with an exemplary picture of the pattern at work and a mini-map diagram to show us where the pattern lies within the language.

An Example of Coram Format [Coram+98]

Interaction Style



Interaction Style

...there are many ways the user may interact with your system. The various interaction styles that you can choose will provide the framework for the user's experience.

It is essential that the interaction style match the needs of the audience.

You need to provide an interface for the user to interact with the system. But how much computer experience does your user have? How much training will your user receive? For instance, text editors such as vi or emacs are useful tools for most programmers, but inappropriate for occasional users.

As a designer, realize when users become frustrated with a program, it is because they know what they want to do, but cannot because the program has a ``secret language'' that the user does not understand.

The computer interface can be an unfamiliar and unnatural thing, but a well designed interface will minimize the gap between a user's goals and the knowledge required to use the interface. Edwin Hutchins, James Hollan and Donald Norman defines these as the Gulf of Execution, the effort required determining how to get a program to do what the user wants, and the Gulf of Evaluation, the effort required to interpret the feedback provided by the program.

Selecting an inappropriate interaction style could increase the Gulf of Execution. Consider a factory foreman with poor typing skills. Providing a command language interface would force this user to do a lot of typing, which in this case would exceed the user's capability, and would almost certainly result in a frustrated foreman.

Therefore:

Study the user and his environment. Work with the user to determine what interaction style is best. Keep things simple and consistent. There are several primary interaction styles: Menu Selection, Form Fill In, Command Language, and Direct Manipulation. Use menu selections to structure the decision making path. Use form fill Ins for data entry tasks. Use command languages when expressibility is an overriding factor. Use direct manipulation interfaces for highly interactive tasks.

A menu selection interface guides the user through well defined tasks. Softkey's MPC Wizard uses this type of interface to guide the user through a series of personal computer diagnostic tests.

Tasks requiring a great deal of user entered information, such as Windows95 Dial-In Network Setup window, are candidates for form fill in.

Programmers and scientists often require the flexibility and power of conversing with the computer by using command languages such as C++, Java and Unix shells.

2.7 James O. Coplien and Douglas C. Schmidt

James O. Coplien and Douglas C. Schmidt's works on patterns and pattern languages are presented using a narrative style which is known as Coplien's format.

Coplien's report in the PLoP'94 [Coplien94a] includes the details of his pattern format. He feels that the essence of the Alexandrian format should be present regardless of the style; i.e. there should always be a clear definition of the problem, the forces, and the solution.

Coplien encourages the use of an abstract that helps a reader to determine whether the pattern is relevant for the problem at hand or not. The solution should be layered, with the most general interpretation at the highest level so that more details are uncovered as the reader progresses through sections.

Coplien also focuses on good names. Good names are important as they reflect the problem, the solution, the resulting context or intent. Therefore, extra care is required to choose a right name. Providing alternative names or nicknames relevant to different domains may be helpful for the users.

Coplien and Schmidt offer the following suggestions:

1. Patterns should be general, not abstract.
2. If considerable collaboration is involved, include a diagram of dynamic behaviors; e.g. an interaction diagram.

Therefore, the Coplien format reflects the basic elements found in the Alexanderian format. It delineates pattern sections with the following headings:

- **The Pattern Name**: The Coplien format commonly uses nouns for pattern names. However, short verb phrases are also used.

- **The Problem**: The problem is often stated as a question or design challenge.

- **The Context**: A description of the context in which the problem might arise, and to which the solution applies.

- **The Forces**: The forces describe pattern design trade-offs; what pulls the problem in different directions, toward different solutions?

- **The Solution**: The solution explains how to solve the problem. A sketch may accompany the solution.

- **A Rationale**: Why does this pattern work? What is the history behind the pattern? We extract this so it does not "clutter" the solution. As a section, it draws attention to the importance of principles behind a pattern; it is a source of learning, rather than action.

- **Resulting Context:** This section describes which forces of a pattern are resolved and which forces remain unresolved, and it points to more patterns that might be the next ones to consider.

An Example of Coplien Format [Coplien+95]

Self-Selecting Team

Problem: There are no perfect criteria for screening team members.

Context: You are building a software development organization to meet competitive cost and schedule benchmarks. You are staffing up to meet a schedule in a given market.

Forces:

- Empowerment depends on competency and the distribution of knowledge and power. The worst team dynamics can be found in appointed teams. The best team dynamics can be found in self-selecting teams.
- Broad interests (music and poetry) seem to align with successful team players.

Solution: Build self-selecting teams, doing limited screening on the basis sack record and broad interests.

Resulting Context: An empowered, enthusiastic team willing to take extraordinary measures to meet project goals.

2.8 The Significance of XML for Pattern Documentation and the Motivation for UPML

Among the existing markup languages and tools for pattern documentation, this study finds XML to be the best suitable one. The reasons are as follows:

1. XML provides a standardized way patterns are written, and hence facilitates the share of knowledge as well as the reuse of patterns.
2. XML is more expressive and provides a higher level of granularity than that is possible by pure natural language.
3. XML provides a way to more formally specify patterns.
4. XML allows a pattern service provider (a pattern server) to vary pattern presentations to suit different situations; e.g. different client-side requirements.
5. XML makes it feasible to verify (validate against a schema) the pattern format.
6. XML enables cross-referencing of patterns using hyper linking mechanisms.
7. XML enables automated programs to extract summaries or descriptions of patterns for the purpose of indexing and use in pattern catalogs.
8. XML provides structure, which facilitates contributors in submitting patterns via the Web or otherwise. Making pattern catalogs available on the Web provides various advantages: evolution (for maintenance, extension), global instant access, inclusion of "dynamic" objects as part of solution, and so on.
9. XML makes it easier for users to navigate (via links) or search (via forms) the online pattern collection.
10. XML enables pattern catalogs to be used as a knowledge base.
11. XML provides cross platform compatibility.

Inspired by the above advantages, the study has proposed the generalized format for pattern documentation (please see chapter 3) and developed a markup language named Usability Pattern Markup Language (UPML) using XML notation (please see chapter 4). Please note that a pattern is expressed in UPML can always be down-transformed

into a traditional text-based notation using eXtensible Stylesheet Language (XSL). XSL has two parts: XSLT (XSL Transformations) and FO (Formatted Objects) [Benoit02] as shown in the following figure:

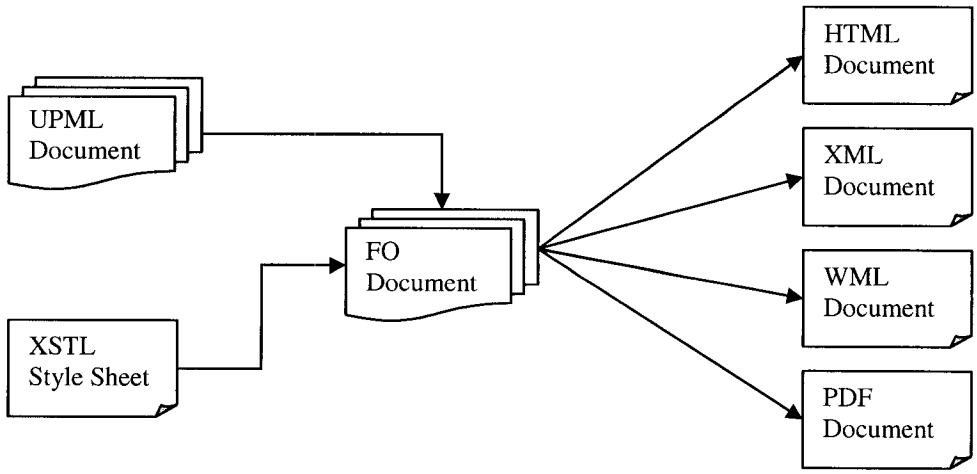


Figure 2.1: Using XSL to Transform UPML documents

2.9 Summary

The comprehensive studies/analysis on some of the most popular formats for pattern documentation provides us the basic platform to carry out our studies; i.e. finding the strengths and weaknesses of the existing formats of pattern documentation with a goal of sketching a generalized format by addressing most of the limitations, which is discussed in chapter 3. The findings of the significance of XML for pattern documentation also motivate us to develop the UPML 1.0 specification, which is elaborated in chapter 4.

Chapter 3

A Generalized Format of Pattern Documentation

This chapter mainly describes two of our major contributions; (i) A Classification Scheme of Patterns (section 3.2) and (ii) A 3D-Reference Model of Pattern Properties (section 3.3). It also shows the development process that leads to establish a generalized format for pattern documentation. Firstly, this chapter explores the weakness of the formats under the study (section 3.1). Secondly, it proposes a Classification Scheme for patterns as well as a 3D-Reference Model of pattern properties with a goal to address some of the weaknesses described in section 3.1. Finally, it outlines the generalized format of pattern documentation into a table (Table 3.2).

3.1 Weaknesses of the Formats

There are several weaknesses found in the study of different formats of pattern documentation. Even now, we do not have any rule in order to validate a pattern or when a best practice will be treated as a pattern. In addition, there is no common vocabulary or notation for patterns in capturing and disseminating the experts' knowledge. This is a major draw back for the usability experts and the HCI community to promote patterns. However, in the recent years, a lot of research on this issue has taken place throughout the world to reach a consensus.

Based on this study on some of the most popular formats for pattern documentation (discussed in chapter 2), we found the following major weaknesses:

1. Lack of naming convention
2. Lack of a classification scheme
3. Lack of coupling and cohesion
4. Inconsistency of elements

5. Lack of implementation strategies
6. Lack of factors, criteria to validate patterns
7. Lack of consistent list of related patterns

Lack of Naming Convention:

There are a numbers of patterns invented by different authors with different names. However, all these patterns address the same problems in the same context. Therefore, it is very hard to apply those patterns by the pattern users or even by the pattern writers. Moreover, it wastes the valuable time and efforts of both patterns writers and users. As the solution of a particular problem has already been addressed, the patterns authors need not re-invent the wheel which is already invented. If someone contradicts with the existing solution s/he might improve the solution by providing her/his valuable recommendation. Thus, the problem can be easily solved if there is a common platform for pattern writers, researchers, and users. Besides, names are frequently changed by the writers (e.g. Jenifer Tidwell and Martijn van Welie) which lead to a serious problem to maintain the pattern dictionary. Therefore, names should be handled in a way so that once a pattern name is assigned and published; names can not be changed any more.

Lack of a Classification Scheme:

The study finds a classification scheme for design patterns provided by the Gang-of-Four [Gamma+95] for their design patterns. They categorized their design patterns based on two criteria. The first criterion, called purpose, reflects what the pattern does. Patterns can have creational, structural, or behavioral purpose. The second criterion, called scope, specifies whether the pattern applies primarily to class or to objects (for the definitions please see section 3.2.4). The following table shows the Gang-of-Four's classification scheme:

Table 3.1: Catalog of Gang-of-Four's Design Patterns

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

According to this classification scheme, distinction between structural and behavioral patterns is too vague. For example, "Mediator" can be used as a structural or as a behavioral pattern, depending on the specific design situation. Moreover "Composite", a structural pattern, is often used with "Iterator" and "Visitor" which are behavioral class pattern, which makes the classification ambiguous. From the scope criterion, we see that "Adapter" is treated as both class and object which can easily leads to confusions. There is another classification scheme presented by Martijn van Welie [Welie03], which is based on the nature of the applications. The following figure shows his classification:

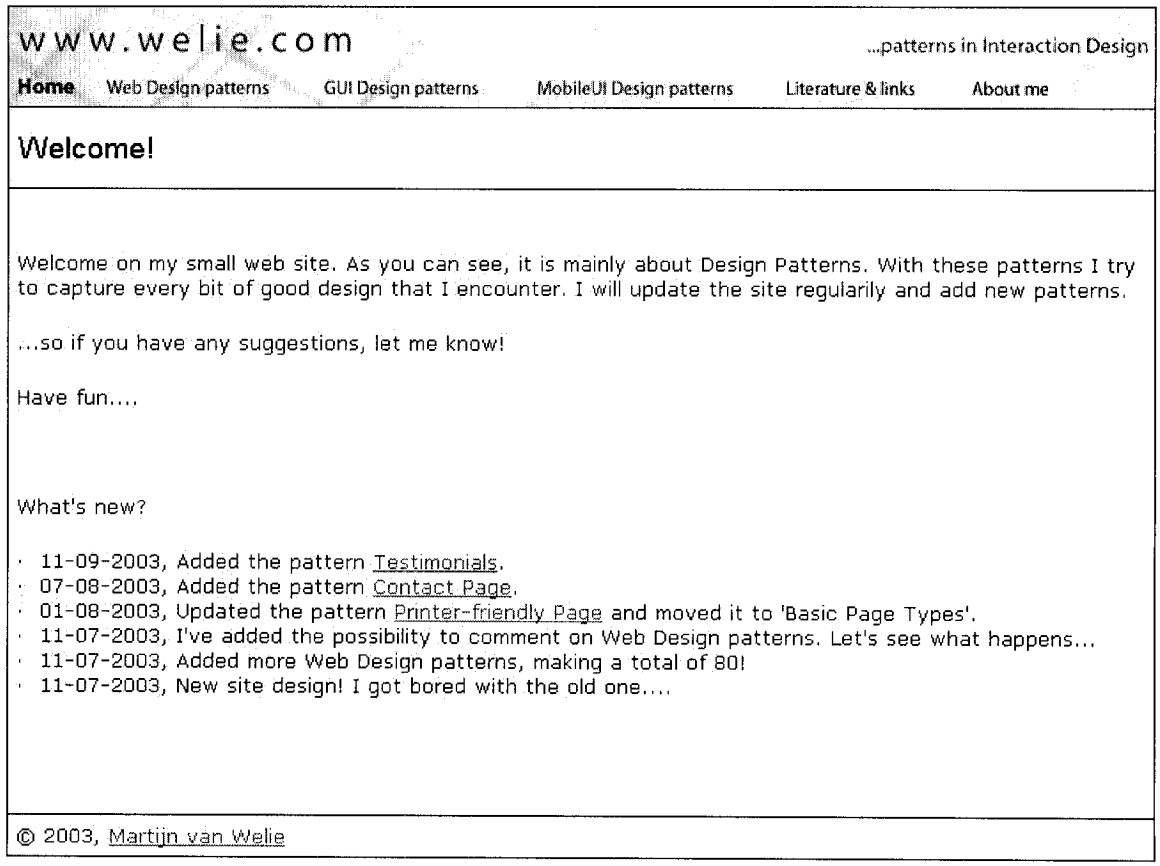


Figure 3.1: Welie's Classification of Patterns

In the above figure, we notice that he classifies the patterns into "**Web Design patterns**", "**GUI Design patterns**", and "**MobileUI Design patterns**" categories. This kind of classification is also ambiguous as we know same pattern may be used by different types of applications. For example, "Search" pattern can be found in all of the above applications.

Lack of Coupling and Cohesion:

Object coupling describes the interdependence (the number of message, the frequency of messages, and the number of arguments) between two objects. There are two types of coupling: "tight" and "loose". Loose coupling is desirable for good software engineering (better encapsulation, fewer objects needlessly affected when making changes) [Coad+95]. On the other hand, cohesion defines the binding of the elements within one method and within one object class. Therefore, high cohesion is desirable for good

software engineering. Most of the formats for pattern documentation are leaning with the Alexandrian format in one way or other. That means most of the formats use the prosy style using natural language to capture their best practices or experience. However, the narrative texts are always tightly coupled and low cohesive. For example, if the Alexandrian format needs to do add one more section, the entire Alexander's works on patterns (about 253 patterns) should be modified manually one by one. This is a very tedious job and inefficient at this scientific age. Therefore, patterns should be documented ensuring the loose coupling and high cohesive characteristics.

Inconsistency of Elements:

This study finds that different authors and communities use different pattern formats including different elements for pattern documentation. They are broadly classified into the following three different sets:

- Name
- Problem
- Context
- Solution

- Name
- Intent
- Context
- Forces
- Solution
- Examples
- Resulting Context
- Related Patterns
- Known Uses

- Name
- Intent
- Also Knows As
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- Sample Code
- Known Uses
- Related Patterns

Even though there are several formats, elements and attributes, there are some similarities in the elements. However, the new users of patterns are getting confused in the first place by having different formats for pattern documentation in the market place. Certainly, the lack of a common set of elements to describe patterns creates complexity and ambiguity among the pattern users or even pattern researchers. To capture and disseminate the knowledge of patterns, patterns need to be clear and understandable, so that it stands on its own, as the author will not be available to answer the users' questions. The users must get a clear understanding of the solution and the trade-offs it makes. The users must understand when it is appropriate to apply the pattern and when it is not. These can be done if pattern writers embrace the fact of having a unified approach for documenting patterns.

Lack of Implementation Strategies:

In most of the formats, except the Gang-of-Four's format, there are no clear implementation strategies i.e. how the pattern can apply to solve the problem under a particular context. For example, if object oriented software developers want to apply one of Alexander's patterns they may face a major problem to interpret this natural description into the implementation process. Therefore, it would be better to have the implementation structure and strategies along with examples, figures, and even a sample code (pseudo code) so that the software developers can easily apply the pattern into their respective domains.

Lack of Factors, Criteria to Validate Patterns:

This is a serious problem related to almost all the formats. In fact, a list of factors, criteria with impacts (positive or negative) is essential to validate a pattern. In addition, this list helps in answering the question: why the users will use it? Some formats have this factor in an indirect way; e.g. within the consequence section. However, this thesis recommends having an explicit list of usability issues in terms of factors and criteria along with the possible impacts (either positive or negative) for a given context. Moreover, it will

facilitate the patterns authenticity. For example, if a pattern claims that it improves the response time but overloads the memory, then users concerned with memory consumption can avoid of using it rather than checking the authenticity after applying it.

Lack of Consistent List of Related Patterns:

A pattern is a generic solution to a problem in a given context. To solve any real world or complex problem, a series of patterns may need to be applied. Therefore, it is essential to have a consistent list of closely related or coherent patterns along with the patterns descriptions. Although some formats do have this list, far behind they are in achieving the goal as they are seriously narrow; i.e. organize their own collections only. For example, "Experience", an article published by Coram T. and Lee J. [Coram+98] shows how badly patterns are related. It may take hours to find out which patterns are related to which one or even to find out the root pattern among those incongruous pattern mappings.

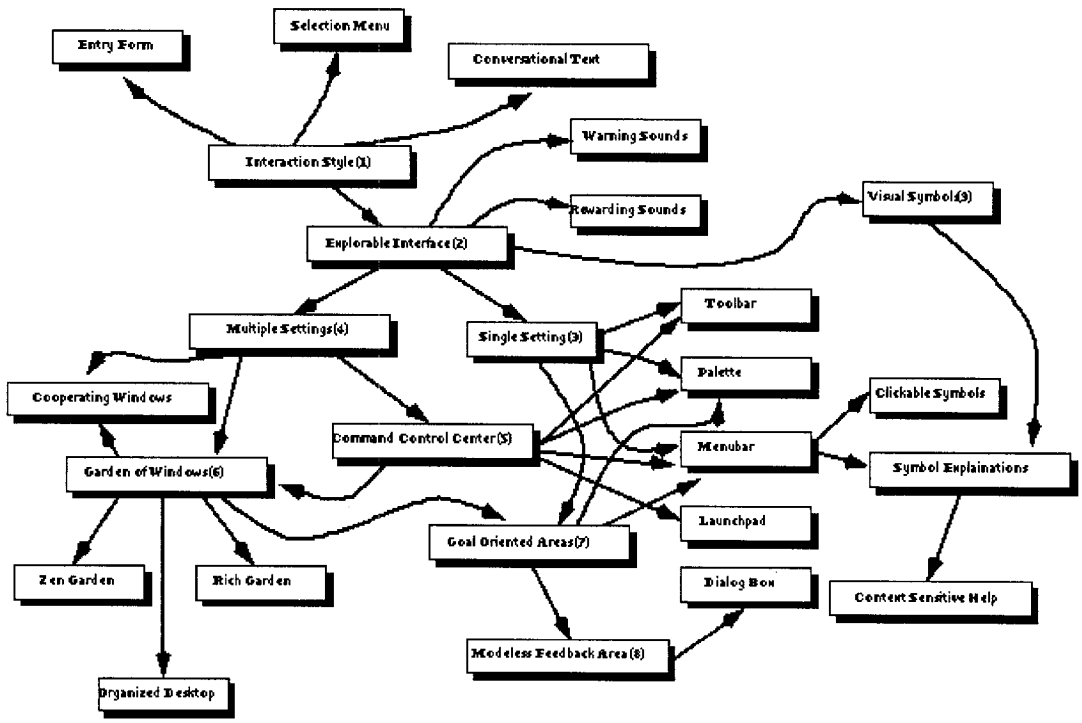


Figure 3.2: Mapping of Patterns

Therefore the best strategy is to have a proper naming convention along with a decisive classification scheme of patterns and an explicit list of the related patterns.

3.2 A Classification Scheme for Patterns

This section defines the proposed classification scheme for patterns. It also describes a list of requirements for the scheme (section 3.2.1), demonstrates how it works with an overview diagram (section 3.2.2), explains the terminology used in this scheme (section 3.2.3), and compares with other classifications, especially with the Gang-of-Four's one (section 3.2.4).

3.2.1 Requirements for the Classification Scheme

As the number of patterns is growing day by day, it is becoming more and more difficult to organize them in an understandable and useable manner. If the users need to read, analyze and understand every pattern in detail to find out the one they need, the pattern collection as a whole is useless, even if its constituent patterns are useful. To handle the entirety of all patterns conveniently within a pattern collection it is therefore helpful to classify them into groups of related patterns. A classification scheme for patterns that supports the development of software systems using patterns should have the following properties:

- It should be simple and easy to learn.
- It should consist of only a few classification criteria that organize patterns efficiently and effectively.
- Each classification criterion should reflect natural properties of patterns, for example the kinds of problems the patterns address, rather than artificial criteria such as whether patterns belong to a pattern language or not.

- It should provide a 'roadmap' that leads users to a set of potentially-applicable patterns, rather than a rigid 'drawer-like' scheme that tries to support finding the one 'correct' pattern.
- The scheme should be open to the integration of new patterns without the need for refactoring the existing classification.

3.2.2 The Proposed Classification Scheme for Patterns

Following the above requirements, this thesis is proposing a classification schema that is simple, flexible to hold up future patterns, and meaningful. This scheme is build upon three classification criteria: "Granularity", "Functionality", and "Structural Principles". Each category represents a crisp set of criteria or design issues that will play a significant and primary role in software development. The following figure gives the overview of the scheme:

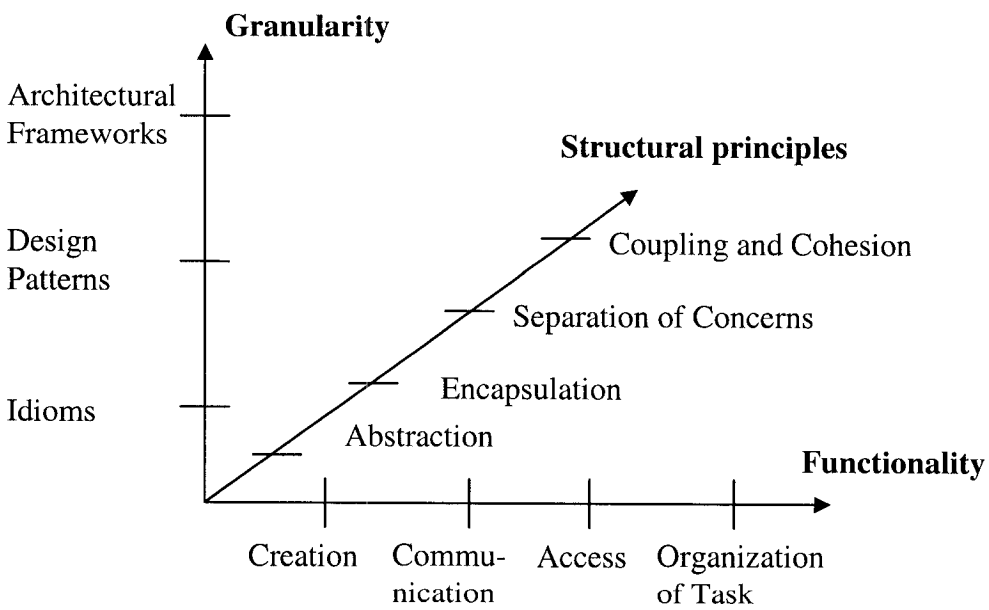


Figure 3.3: A Classification Scheme for Patterns

Every pattern within a pattern collection may be classified according to the above categories. Firstly, patterns are categorized based on granularity (level of abstraction). Then a list of functionalities and structural principles are associated with each pattern. For example, most of the Gang-of-Four's design patterns will fall into second level based on the granularity. Moreover, the proposed scheme provides a list of functionalities and structural principles, which the pattern will address. By doing so, the scheme gives multi-ways of indexing and eliminates the ambiguity that comes from the Gang-of-Four's creational, behavioral, structural based classification and provides the flexibility to categorize patterns (including the future ones) without a major complexity. Therefore, the resulting classification scheme forms a guide that users can use in choosing patterns for specific applications.

To use a classification scheme in a concrete design situation, we may take the following steps:

1. Determine the required granularity for the pattern. This is usually easy to decide, since it should be clear if the pattern serves as a basic structure for a whole application, a scheme for structuring a subsystem or component, or a concrete implementation of a specific design requirement.

2. Select the required functionality. This should be clear as well, since usually only one category of functionality is considered for a given design situation. However, it might be necessary to combine several functional aspects within a single structure. In this case, either a pattern that serves all the desired functionalities is selected or several patterns are selected that together provide the required behavior.

3. Determine the desired structural principles. This is the most difficult step. It is certainly a design decision, because often more than one structural solution is possible for a specific functionality requirement. The proposed principles force

designers to think about what structural property is the most useful and important for the situation at hand.

The classification scheme provides a guide for users, helping them to search for an appropriate pattern for a given context.

3.2.3 Explanation of the Terminology

The three categories can be identified as follows:

Granularity

Developing a software system requires one to deal with various levels of abstraction, beginning with the basic structure of an application and ending with issues regarding the concrete realization of particular design structures. Thus granularity is an important category for classifying patterns. The three levels of granularity are as below:

i. Architectural Frameworks: Every software architecture is built according to an overall structuring principle. These principles are described by architectural frameworks as follows:

> *An architectural framework expresses a fundamental paradigm for structuring software systems. It provides a set of predefined subsystems, as well as rules and guidelines for organizing the relationships between them* [Coplien+95].

Architectural frameworks determine an application's basic structure and have an influence on its subsystem architecture. Thus architectural frameworks allow one to handle the high level structural complexity of software systems. The selection of a particular architectural framework for a software system is a fundamental design decision.

To construct a concrete software architecture based on an architectural framework, the application's functionality must be mapped into its structure. In addition, it must be enhanced and specified in detail. With the help of smaller patterns, its predefined subsystems have to be refined, and the relationships between them have to be fully specified.

Example: The Model-View-Controller framework, Java Strut Studio, J2EE, etc.

ii. Design Patterns: Software architecture usually consists of several smaller architectural units. These are described by design patterns as follows.

> *A design pattern describes a basic scheme for structuring subsystems and components of a software architecture, as well as the relationships between them. It identifies, names, and abstracts a common design principle by describing its different parts and their collaboration and responsibilities* [Gamma+93].

Design patterns can be seen as micro architectures [Gamma+93]. They are software architectures. As they used to structure software systems they are micro architectures. A design pattern is less than a complete software architecture or architectural framework.

A design pattern may also be interlocked with other design patterns and composed out of several smaller patterns.

Example: Master-Slave pattern, Producer-Consumer pattern etc.

iii. Idioms: Idioms deal with the concrete realization and implementation of particular design issues.

> *An idiom describes how to implement particular components (parts) of a pattern, the components' functionality, or their relationships to other components within a given design. They often are specific for a particular programming language* [Coplien+95].

Idioms represent the lowest level of a pattern. They are closely related to a particular programming language. Often the same idiom looks different in different languages, and sometimes an idiom that is useful in one programming language does not make sense in others. For example, a C++ idiom may describe how to implement reference counting to correctly handle multiple referenced objects. In Smalltalk such an idiom is not needed because of the garbage collection mechanism integrated in the language.

Since idioms address aspects of both the design and the implementation of a particular structure, they close the gap between the design and implementation phase of software development.

Example: The Counted Body Idiom [Coplien94b].

Functionality

The second criterion for classifying patterns is functionality. Each pattern serves as a template for implementing a particular functionality. However, the various classes of functionality are of a general nature rather than specific for a certain application domain. The following categories of functionality can be distinguished:

i. Creation of Objects (Creation): Patterns may specify how to create particular instances of complex recursive or aggregate object structures.

ii. Guiding Communication Between Objects (Communication): Patterns may describe how to organize communication between a set of collaborating objects that may also be independently developed or remote.

iii. Access to Objects (Access): Patterns may describe how to access the services and state of shared or remote objects in a safe way, without violating their encapsulation of state and behavior.

iv. Organizing the Computation of Complex Tasks: Patterns may specify how to distribute responsibilities among cooperating objects in order to solve a more complex function or task.

Structural Principles

To realize their functionality, patterns rely on certain architectural principles. These principles form the third and final criterion.

i. Abstraction: A pattern provides an abstract or generalized view of a particular (often complex) entity or task in a software system.

ii. Encapsulation: A pattern encapsulates details of a particular object, component, or service to remove dependencies on it from its clients or to protect these details from access.

iii. Separation of Concerns: A pattern factors out specific responsibilities into separate objects or components to solve a particular task or provide a certain service.

iv. Coupling and Cohesion: A pattern removes or relaxes the structural and communicational relationships and dependencies between otherwise strongly coupled objects.

3.2.4 Comparison

The Gang-of-Four's schema has two dimensions: purpose and scope. The following paragraphs are an excerpt from the Gang-of-Four book.

> *The first criterion, called purpose, reflects what a pattern does. Patterns can have creational, structural, or behavioral purpose. Creational patterns concern the process of object creation. Structural patterns deal with the composition of*

classes or objects. Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility.

The second criterion, called scope, specifies whether the pattern applies primarily to classes or to objects. Class patterns deal with relationships between classes and their subclasses. These relationships are established through inheritance, so they are static-fixed at compile-time. Object patterns deal with object relationships, which can be changed at run-time and are more dynamic.

According to this classification schema, a distinction between structural and behavioral patterns is too vague. For example, "Composite", a structural pattern, is often used with "Iterator" or "Visitor" which are behavioral class pattern, which leads confusion among users. Furthermore, the Gang-of-Four's scope criterion will not provide any help to software developers for selecting a pattern. This is because it does not relate to any specific design situation or activity, and also does not fit with non-object-oriented patterns such as Layers or Pipes and Filters.

On the other hand, the proposed classification scheme categorizes patterns based on level or granularity. At the lowest level are language specific patterns, known as idioms. For example, C++ Orthodox Canonical Class Form idiom and the Handle/Body Class idiom [Coplien94b]. Design patterns, such as those catalogued in Gang-of-Four, fall at the middle level. Design patterns are not language specific and can be implemented in a variety of languages. At the highest level are architectural patterns; e.g. the Model-View-Controller (MVC). This classification is also concerned with the structure of the pattern, which helps in identifying patterns in an existing software system. It also addresses one or more functionalities which makes the classification scheme flexible enough to hold the future patterns.

Other organizational schemes for patterns are presented in several articles [Eisenhauer+94, Zimmer94, and Buschmann+94]. R. Eisenhauer builds on problem categories, such as transactions or bridging the gap between object-oriented applications

and relational databases, in the same way that the proposed scheme does. Zimmer focuses on relationships between patterns; e.g. "pattern A uses pattern B" or "pattern A is similar to pattern B" in his solution [Zimmer94].

Unlike Buschmann's [Buschmann+01] two dimensional classifications of patterns, pattern categories and problem categories; the proposed scheme is three dimensional. The first two dimensions--called 'granularity' and 'functionality' - correspond directly to his pattern and problem categories. The third dimension, 'structural principles', depicts the technical principles of underlying solutions that the patterns propose.

3.3 A 3D-Reference Model of Pattern Properties

This section defines the proposed 3D-Reference Model of Pattern Properties, its elements and attributes. It also describes a list of requirements for the model (section 3.3.1), demonstrates how it works with an overview diagram (section 3.3.2), and explains the terminology used in the model (section 3.3.3).

3.3.1 Requirements for the 3D-Reference Model

According to the Gang-of-Four description, the fundamental properties of pattern are as follows:

Each pattern

- Provides a predefined scheme for implementing a particular structural or functional principle for software systems, by describing its different parts as well as their collaboration and responsibilities
- Captures existing, well-proven design experience
- Identifies names, and specifies abstractions that are above the level of classes and instances

- Provides a common vocabulary and understanding for design principles.
- Helps to handle the complexity of software
- Serves as a reusable building block for software development
- May be either domain-independent or domain-specific (such as exception handling)
- Addresses both functional and nonfunctional aspects of software design

Besides the fundamental properties, the proposed 3D-Reference Model needs to have the following characteristics:

Encapsulation: Each pattern needs to encapsulate a well-defined problem/solution [Parnas79]. Patterns should be independent, specific, and precisely formulated in order to make it clear, understandable, and reusable.

Generativity: Each pattern needs to contain a local, self-standing process prescription describing how to construct realizations as patterns may be usable by all development participants, not only the trained designers but also the novice users.

Equilibrium: Each pattern needs to identify a solution space containing an invariant that minimizes conflict among forces and constraints. When a pattern is used in an application, equilibrium provides a reason for each design step, traceable to situational constraints. The rationale that the solution meets this equilibrium may be a formal, theoretical derivation, an abstraction from empirical data, observations of the pattern in naturally occurring or traditional artifacts, a convincing series of examples, analysis of poor or failed solutions, or any mixture of these. Equilibrium is the structural side of optimality notions familiar in computing, and can be just as hard to find a basis for [Johnson92]. Alexander argues for establishment of objective equilibria based in the "quality without a name" even (or especially) when surrounding aesthetic, personal, and social factors. He also notes the elusiveness of this goal artifacts more often than not fail to achieve this quality despite the best of efforts.

Abstraction: Patterns need to be represented as the abstractions of empirical experience and everyday knowledge. They are general within the stated context, although not necessarily universal. Pattern construction (like domain analysis [Prieto+89]) is an iterative social process collecting, sharing and amplifying distributed experience and knowledge. Sometimes, patterns may be constructed more mechanically by merging others and/or transforming them to apply to a different domain.

Openness: Patterns need to be extended down to arbitrarily fine levels of detail. Patterns are used in development by finding a collection of entries addressing the desired features of the project at hand, where each of these may in turn require other sub-patterns. For example, while only a small set of patterns would typically apply in the design of a certain housing community, each house will itself be unique due to varying micro-patterns. Because the details of pattern instantiations are encapsulated, they may vary within stated constraints. These details often do impact and further constrain those of other related patterns. But again, this variability remains within the borders of higher-level constraints.

Composibility: Patterns need to be hierarchically related. Coarse-grained patterns are layered on top of, relate, and constrain fine-grained ones. These relations include, but are not restricted to, various whole part relations [Civello93]. Most patterns are both upwardly and downwardly composible, minimizing interaction with other patterns, making clear when two related patterns must share a third, and admitting maximal variation in sub-patterns. Pattern entries are arranged conceptually as a *language* that expresses this layering. Because the forms of patterns and their relations to others are only loosely constrained and written entirely in natural language, the pattern language is merely analogous to a formal production system language, but has the same properties, including infinite nondeterministic generativity.

Therefore, patterns must have the characteristics that comply with the object-oriented design paradigm or software engineering principles in order to capture and disseminate the best practices and proven solutions effectively. In addition, the study always highlights the fact of simplicity and learnability [Seffah+00, Ning01]. Hence, the study

revisited the Alexandrian format for the simplicity and the Gang-of-Four formats (the master in object-oriented design patterns) for accommodating the necessary features of the object oriented design paradigm.

3.3.2 The Proposed 3D-Reference Model of Pattern Properties

The main objective of this thesis is to reduce the communication gap among different professionals groups (Pattern Writers, Usability Experts and Software Designer and Developers) by providing a consistent approach for pattern documentation. This is achieved by finding the commonalities among different formats after a through study. A list of the concerning issues about patterns, raised by different professional groups, is also needed to be considered for the better understandability and reusability. Thus, the study uses the Alexandrian format as the fundamental basis to develop the proposed generalized format (which is outlined in Table 3.2). The study also incorporated some of the design issues from the Gang-of-Four format in order to accommodate the object-oriented design paradigm as well as reduce the communication gap, particularly with the software developers. The following figure shows an overview of various concerns related to patterns that came from the three key professional groups (pattern writers, usability experts, and software developers):

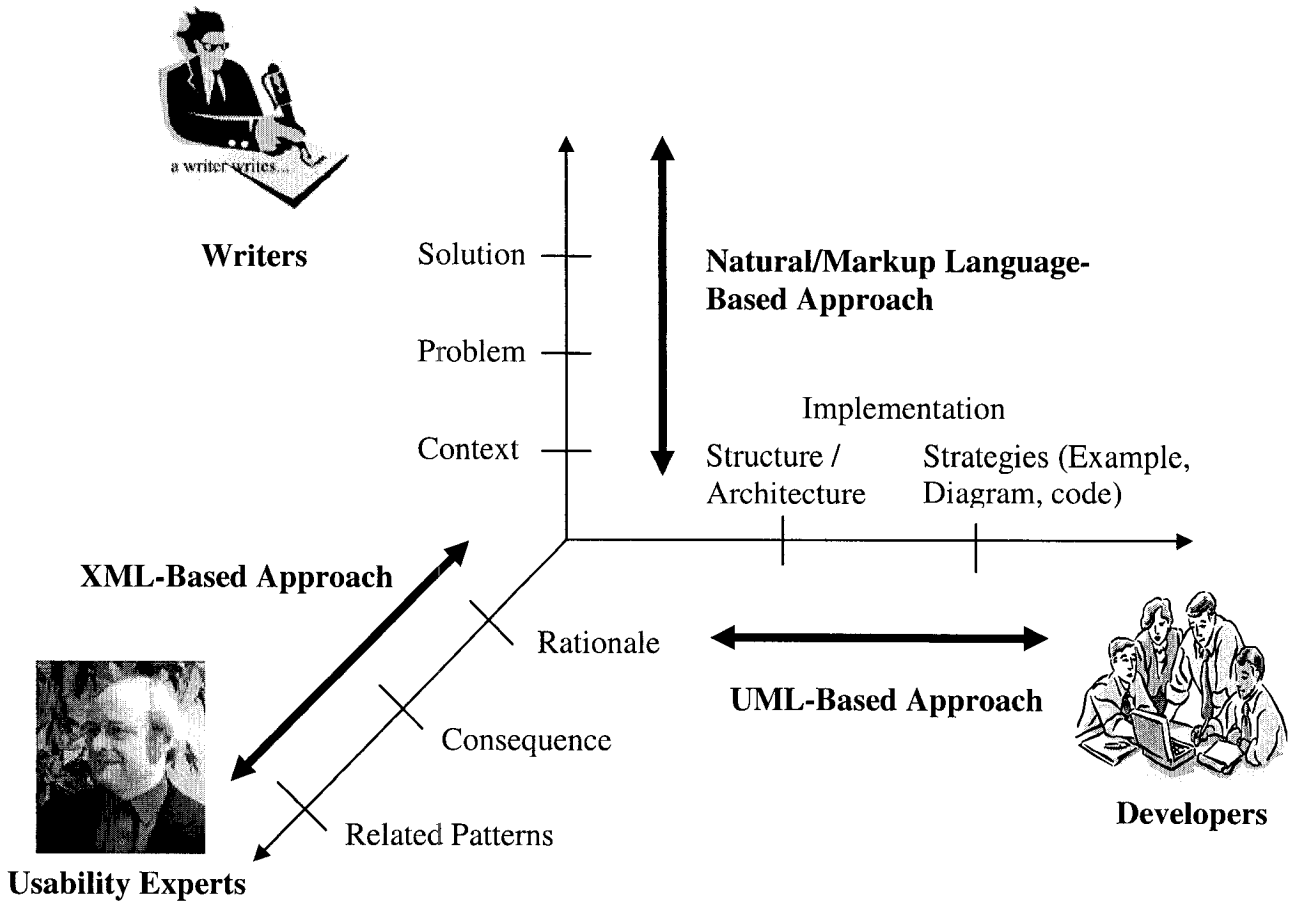


Figure 3.4: A 3D-Reference Model of Pattern Properties

The above model clearly shows the major concerns of each professional group related to patterns. By introducing this Reference model, the study basically provides a common platform where each professional group can be aware of its own concerns as well as the others which may result in utilizing the great benefits of patterns efficiently and effectively. It also reduces the communication gap among those three key professional groups as this platform gives them an opportunity for a good team work. Therefore, each group may consider the others' concerns while contributing towards patterns rather than working isolated and focusing only for their own benefits. Moreover, this is a vital issue to address, which is also one of the key objectives of this thesis. Otherwise, the main goal of patterns, ease to understand and use, will never be achieved in this software engineering domain.

3.3.3 The Proposed Generalized Format

Bearing the above facts shown in the 3D-Reference model (Figure 3.2), this thesis has proposed a generalized format as below. This may be treated as a "Standard Template" in future (we hope) based on Industry feedback and/or empirical studies.

Table 3.2: The Generalized Format of Pattern Documentation

Element	**Sub-Elements**	
Identification	Name	
	Alias	Optional
	Author(s)	
	Date	
	Category	Patterns Classification
	Keyword	For searching
	Related Pattern(s)	Superordinate
		Subordinate
		Sibling/Neighboring
		Competitor
Context of Use	User	Category of users; i.e. novice, expert etc.
	Task	Tasks are structured hierarchically. All sub-tasks should be originated from a root.
	Platform Compatibility and Constraints	Environmental and Technical aspects.
Problem	Give a statement of the problem that this pattern resolves. The problem may be stated as a question.	
Forces	Forces describe the influencing aspects of the problem and solution. This can be represented as a list for clarity.	

Solution	Give a statement of the solution to the problem including the rationale behind the solution. It could also provide the references for further understanding.	
Implementation	Structure	It's a high level abstraction done by visual modeling notation.
	Strategy	Including examples, figures, sample-code etc.
Consequences	Trade-off and results of using the pattern. It can be described by a list of usability factors, criteria, or metrics.	

3.3.4 Explanation of the Terminology

Pattern Name and Alias

Names are used to succinctly convey the purpose of the patterns and to provide a mnemonic aid to remember them. The name allows one to use a single word or short phrase to refer to the knowledge a pattern encompasses. According to Alexander, names can name the thing created by the pattern, the process of creating it, or some attributes of the solution.

While it is difficult to fully encompass a single pattern in its name, the pattern names are intended to provide sufficient insight into the function of the pattern. Well-chosen pattern names form a vocabulary for discussing design problems and solutions. A pattern should have a meaningful name that represents the problem it is addressing. The names should be as granular as possible. A carefully chosen pattern name may be used as identifiers for further processing, and can assist in indexing and querying a pattern database. Therefore, inclusion of white space and shell meta-characters in the name should be avoided.

Context of Use

The context is the set of conditions under which the problem recurs, and for which the solution is desirable. Apart from the problem description, the context also provides

criteria for determining when the pattern is applicable. This section describes the context in which the problem occurs including the characteristics of the user, tasks, as well as of the technical, physical and organizational environment. Each of these is identified by a set of attributes. Not all of these attributes will be relevant in any particular system, and/or additional attributes may need to be used.

Some aspects of Context of Use can be summarized as follows:

User = Novice, Intermediate, Expert, Occasional

Task = Duration, Frequency, Flexibility

Platform Compatibility and Constraints = Environmental and Technical aspects

Problem

While using a user interface, the users can face many problems. These problems are typically task related. The users need to know about the rational inputs that they have to give to have the rational outputs from the application(s). This section describes the user problem which the pattern attempts to solve within the given context and constraints of the problem.

Forces

Forces reveal the intricacies of a problem and define the kinds of trade-off that must be considered in the presence of the tension or dissonance they create. A good pattern description should fully encapsulate all the forces that have an impact upon it. A description of the relevant forces and constraints includes how they interact/conflict with one another and with goals we wish to achieve (perhaps with some indication of their priorities). A concrete scenario which serves as the motivation for the pattern is frequently employed or used.

Examples

This section gives instance(s) of situations where the pattern is used. Examples help usability engineers to understand the scope and domain of applicability of the pattern. This also enforces the fact that the pattern describes a proven solution. The section may

also include "counter-examples" (example(s) of interface(s) where the pattern should have been used but was not) and "non-examples" (example(s) of interface(s) where the pattern should not have been used but was).

The example(s) can be provided in several ways: prose, diagrams, pictures (hand sketched or photographed), and so on, which illustrate the use of the pattern.

Rationale

Given a problem, and a large collection of patterns, one faces the issue of making a viable choice. This section describes, in a solution-independent manner, the reasoning behind and suitability of the pattern as a justified choice towards solving the usability problem. The rationale assists a usability engineer in making an appropriate choice by describing how and why the pattern works, with an insight into the internal structure and key mechanisms of the system.

Solution

This section describes the actual solution provided by the pattern to solve the problem. It describes the solution approach briefly and the solution elements (such as, prose, diagrams, and pictures) in detail. The solution elements identify the pattern's structure, presentation, logic, and behavior.

The solutions that the patterns suggest are based upon different design principles. A principle is a theoretical framework that is expressed in prose and a result of an amalgamation of an expert's ideology and practical experience with users. This section describes the underlying principle and is intended to provide a guideline and direction for future pattern researchers to work. The principle may also be used to structure a collection of patterns.

Implementation

Given the large variety of applications, usability patterns upon implementation should exist in various formats. The implementation section may include:

- **Structure:** Describes the high level abstraction of a pattern using a graphical notation. For example, use of UML Class Diagrams to show the basic structure of the solution. Similarly, UML Sequence Diagrams present the dynamic mechanisms (data flow) of the solution. This is supplemented by a detailed explanation of the participants and collaborations.

- **Strategies:** Describe different ways a pattern can be implemented. Strategies promote better communication, by providing names for lower-level aspects of a particular solution. Engineers discover and invent new ways to implement the pattern, producing new strategies for well-known patterns. To accommodate that, strategies provide an extensibility point for each pattern.

Consequences

This section describes usability-related impact and trade-offs. In general, this section focuses on the results of using a particular pattern, and notes the pros (such as, what usability aspects have been improved) and cons (such as, what usability aspects have worsened) that may result from the application of the pattern. It is likely that a pattern may improve one aspect at the cost of deteriorating others.

This section describes relevant factors and criteria that are used by usability engineers to justify the usability of the design solution. Factors and criteria together provide a quantitative picture of the complexities of the problem(s) and help to define the kinds of trade-offs that must be considered. A pattern resolves one or more of the factors under given criteria.

Some usability factors and criteria can be summarized as follows:

- Factors = {Efficiency, Effectiveness, Satisfaction, Productivity, Safety, Accessibility, Universality}

- Criteria = {Understandability, Operability, Aesthetics, Compliance, Consistency, Flexibility, Minimal Action, Minimal Memory load, Guidance, Accuracy, Completeness, Required Resources, Helpfulness, Controllability}

The following are the measurable (quantitative and qualitative) aspects of usability that have been identified over the time in the usability pattern literature:

- Learnability: How easy the system is to use and learn. How steep is the learning curve.
- Task Completion: What degree the task could be completed to.
- Error Analysis: The number of errors made by a user while using the system. The degree (such as, minor, major, fatal) and reasonability (such as, unacceptable, ignorable) of errors.
- Performance: How quickly a user can accomplish the task (or a sub-task) using the system?
- Satisfaction: What is the level of satisfaction on part of the user?

Related Patterns

This section provides other patterns (either super-ordinate, subordinate, competitor, or neighboring patterns) that are related, with pointers to where they can be found. For each related pattern, there is a brief description of its relationship to the pattern being described.

Chapter 4

UPML Specification

This chapter defines the syntax and semantics of the UPML 1.0 specification. Section 4.1 lists the UPML modules and their elements. Section 4.2 lists the basic properties of elements and attributes, including conditions on their data types and enumeration. Sections 4.3 through 4.7 provide definitions of individual elements along with the details of corresponding sub-elements, attributes, and examples. Here, several definitions have been repeated from chapter 3 for smooth reading. Section 4.8 provides definitions of each of the attributes. Section 4.9 provides objects of UPML identification, namely UPML media type and UPML namespace. Finally, section 4.10 discusses the issue of conformance with respect to UPML.

The current status, as being the first draft, of all UPML related documents, grammars, and schema is of version 1.0.

4.1 UPML Module and Element Definitions

UPML is composed of five sets of semantically-related elements and attributes, which are: Association Module, Meta-information Module, Problem Module, Solution Module, and Structure Module.

UPML has thirty one elements; each one is responsible for a specific functionality of the language. The following table lists the UPML modules along with the alphabetical list of corresponding elements.

Table 4.1: UPML Modules

Module	**Elements**
Association Module	category, link, reference, related-pattern

Meta-Information Module	alias, author, date, identification, keyword, metadata, name, term, title
Problem Module	context, forces, platform-compatibility, problem, task, user
Solution Module	consequences, example, implementation, rationale, sample-code, solution, strategy, structure
Structure Module	body, head, pattern, upml

The following figure (Figure 4.1) will give a structural overview of UPML that displays its **key** elements.

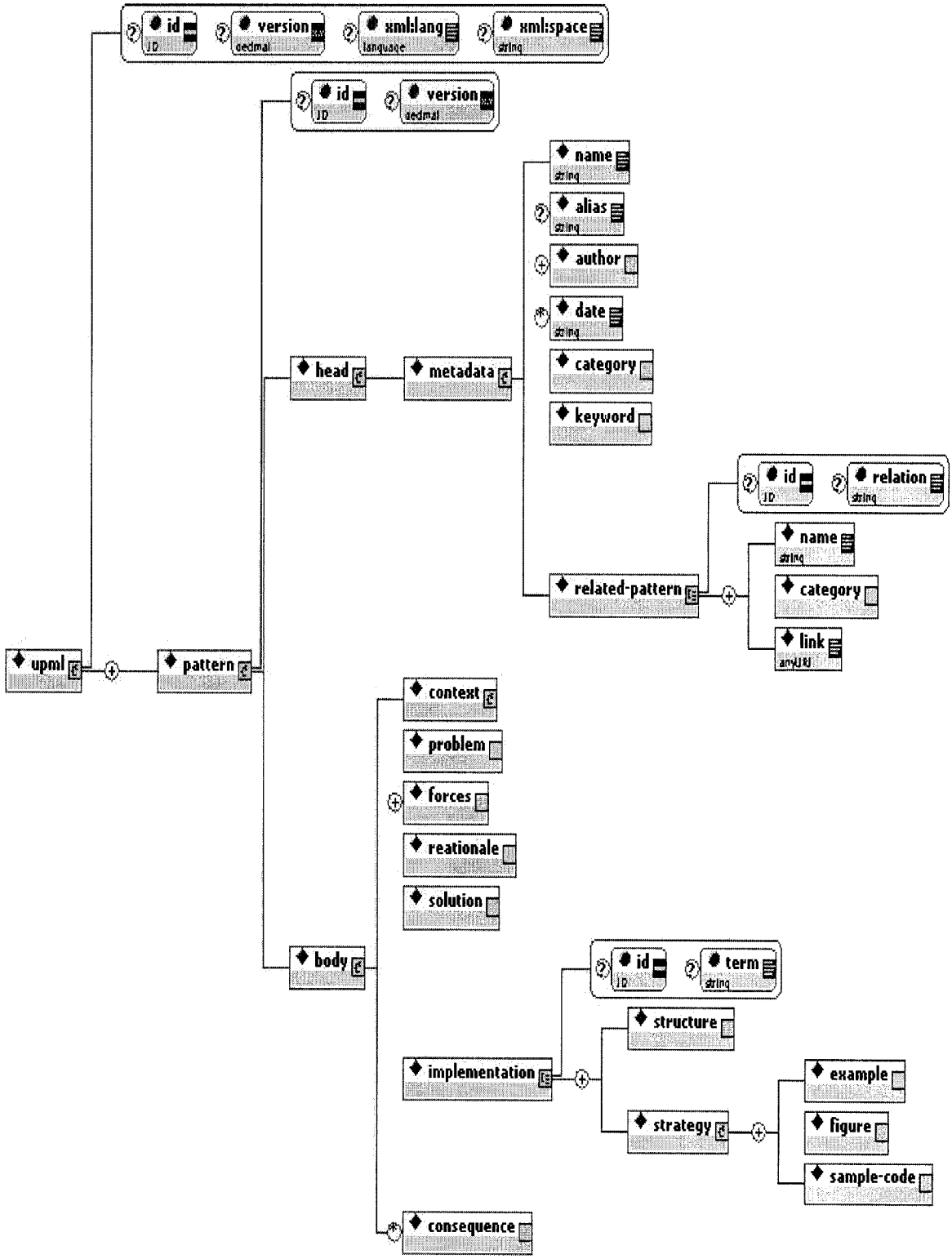


Figure 4.1: Structure of UPML

4.2 Properties of UPML, Elements and Attributes

The two properties of interest are data type and enumeration.

4.2.1 UPML Data Types

UPML data types determine the type of data allowed in elements content and attribute values.

A "**Singleton**" data type indicates that the element content consists of only character data and does not have any sub-elements. An "**Aggregate**" data type indicates that the element content consists of only sub-elements. A "**Mixed**" data type indicates that the element content is a combination of character data and sub-elements. The class of characters, allowed as character data, is determined by the XML 1.0 specification [W3C00a, W3C00b]. For the types of character data in element content and attribute values, a library of data types such as the one provided by XML Schema [W3C01] can be used.

Empty elements do not contain any content. All UPML elements, unless stated otherwise, are non-empty.

4.2.2 Enumeration

Enumeration determines the number of times an element can occur; i.e. its multiplicity.

The occurrence of elements is classified as **Required**, **Conditional**, or **Optional**. An element that is labeled as "**Required**" must be present in every UPML document; an element that is labeled as "**Conditional**" must be present in a UPML document under the given conditions; an element that is labeled as "**Optional**" may not be present in every UPML document. The concept of cardinality, which indicates the number of times an element can occur, if at all, is related to occurrence. If an element is a child of another element, the cardinality indicates the number of times the child element can occur in its parent. A cardinality of "N" indicates that the element occurs N times, where N=1, 2, ... A cardinality of "N..M" indicates that the element can occur N to M times, where N=0, 1,

2, ..., M=1, 2, ..., and M>=N. A cardinality of "N...Unboundeded" indicates that the element can occur N or more times, where N=0, 1, 2, ... An element whose occurrence is **Required** will have a cardinality of (at least) 1; an element whose occurrence is either **Optional** or **Conditional** will have a cardinality of (at least) 0. Note that the root element is always **Required** with cardinality = 1.

The occurrence of attributes is classified as **Required**, **Conditional**, or **Optional**, and is indicated in the parenthesis next to its name. An attribute that is labeled as "**Required**" must always be present in its corresponding element; an attribute that is labeled as "**Conditional**" must be present in its corresponding element under the given conditions; an attribute that is labeled as "**Optional**" may not always be present. A "**None**" indicates that the element does not have any attribute. An attribute can occur only once in an element.

The following sections specify the individual definitions of elements and attributes.

4.3 Association Module

The Association Module signifies the association of a pattern to its class (category), to other pattern(s), or to a reference. The association is made possible by the linking mechanism.

Table 4.2: The `category` Element

Element Name	`category`
Module	Association Module
Description	The patterns are classified into several categories as discussed in section 3.2 of chapter 3. The `category` element names the classification scheme to which the pattern belongs to. It also helps to organize patterns efficiently.
Data Type	Singleton

Attribute(s)	`id` (Optional)
Parent Element(s)	`identification, metadata`
Child Element(s)	None
Example	`<category>Design Pattern</category>`

Table 4.3: The `link` Element

Element Name	`link`
Module	Association Module
Description	The purpose of the `link` element is to provide linking semantics to the pattern element it is associated with. **Note:** When transforming a UPML document to other formats, the `link` element can be mapped to sophisticated hyper linking schemes such as XLink [W3C02], which provides support for both uni-directional and bi-directional linking.
Data Type	Not Application (Empty Element)
Attribute(s)	`id` (Optional), `uri` (Required)
Parent Element(s)	`reference`
Child Element(s)	None
Example	`<link uri="http://www.welie.com/patterns"/>`

Table 4.4: The `reference` Element

Element Name	`reference`
Module	Association Module
Description	The `reference` element includes information on references related to the pattern problem and/or solution that may be helpful towards further understanding. It suggests that a referenced item should be canonical and follows standard guidelines of a bibliography. If a referenced item is accessible via Web, the URL should be provided.

Data Type	Aggregate
Attribute(s)	`id` (Required)
Parent Element(s)	`problem, solution`
Child Element(s)	`title` (Required, Cardinality = 1), `author` (Required, Cardinality = 1..Unbounded), `date` (Required, Cardinality = 1..Unbounded), `link` (Required, Cardinality = 1)

Example	`<reference id = "Appleton00">`
	`  <title>Patterns and Software: Essential`
	`  Concepts and Terminology</title>`
	`  <author>Appleton, B.</author>`
	`  <date>2000</date>`
	`  <link uri="http://www.eteract.com/~bradapp`
	`  /docs/patterns-intro.html"/>`
	`</reference>`

Table 4.5: The `related-pattern` Element

Element Name	`related-pattern`
Module	Association Module
Description	The `related-pattern` element provides information on other pattern(s) that is related to the pattern being described. It may include the type of relationship to the described pattern, class to which it belongs to, and pointer to where it can be found. The relationship can be categorized as **superordinate**, **subordinate**, **sibling/neighboring**, or **competitor**. A **superordinate** pattern is the super set of the described pattern. It can therefore contain the described pattern and possibly other patterns. A **subordinate** pattern is a subset of (i.e. it can be embedded into) the described pattern. It is therefore a part of the described pattern. A **sibling/neighboring** pattern belongs to the

	same pattern category as the described pattern. It provides either replaceable or enhanced function to the described pattern, but not necessarily in the same context. Finally, a **competitor** pattern can provide the identical or similar function as the described pattern. Thus, it can replace the described pattern in the same context.
Data Type	Mixed
Attribute(s)	`id` (Optional), `relation` (Required)
Parent Element(s)	`identification, metadata`
Child Element(s)	`name` (Required, Cardinality = 1), `category` (Required, Cardinality = 1), `link` (Optional, Cardinality = 0..1)

Example	`<related-pattern relation = "sibling">`
	`<name>Bridge</name>`
	`<category>Design Pattern</category>`
	`</related-pattern>`

4.4 Meta-Information Module

Literate Programming [Knuth92] advocates program literacy and emphasizes that program should be written in such a manner, so that compilers as well as human beings can read that program. UPML documents should follow a similar approach, which is the motivation for adopting the Meta-Information Module.

Table 4.6: The `alias` Element

Element Name	`alias`
Module	Meta-Information Module
Description	The `alias` element describes the other well known names of the pattern.
Data Type	Singleton
Attribute(s)	`id` (Optional)

Parent Element(s)	`identification, metadata`
Child Element(s)	None
Example	`<alias>Wrapper</alias>`

Table 4.7: The `author` Element

Element Name	`author`
Module	Meta-Information Module
Description	The `author` element contains the name of the pattern writer(s) and contributor(s). The names can be expressed in first name-last name or last name-first name formats.
Data Type	Singleton
Attribute(s)	`id` (Optional)
Parent Element(s)	`identification, metadata, reference`
Child Element(s)	None
Example	`<author> Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides</author>`

Table 4.8: The `date` Element

Element Name	`date`
Module	Meta-Information Module
Description	The `date` element provides a timestamp (namely, pattern creation, publication, and revision). It may be used to tabulate the date(s) of evolution of the pattern. It can also be used for the date of reference. The date can be expressed in a Gregorian format of recurring day, month, and year.
Data Type	ISO 8601 Format
Attribute(s)	`event` (Optional)
Parent Element(s)	`identification, metadata`
Child Element(s)	None
Example	`<!-- The 1st of March, 2003 -->`

	<date event = "creation">2003-03-01</date>

Table 4.9: The identification Element

Element Name	identification
Module	Meta-Information Module
Description	The identification element describes how a pattern can be uniquely disguised within a patterns' repository. Certainly it is important for increasing vocabulary of the pattern dictionary. It also helps to communicate with others effectively.
Data Type	Aggregate
Attribute(s)	id (Optional)
Parent Element(s)	head
Child Element(s)	name (Required, Cardinality = 1), alias (Optional, Cardinality = 0..1), author (Required, Cardinality = 1..Unbounded), category (Required, Cardinality = 1) related-pattern (Optional, Cardinality = 0..1)
Example	<identification> <name>Adapter </name> <alias>Wrapper</alias> <author> Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides </author> <category>Design Pattern</category> <related-pattern relation="sibling"> <name>Bridge</name> <category>Design Pattern</category> </related-pattern> < identification>

Table 4.10: The `keyword` Element

Element Name	`keyword`
Module	Meta-Information Module
Description	The `keyword` element includes a sequence of term(s) relevant to the pattern.
Data Type	Mixed
Attribute(s)	None
Parent Element(s)	`identification, metadata`
Child Element(s)	`term` (Required, Cardinality = 1..Unbounded)

Example	`<keyword>`
	`<term>Design</term>,<term>Pattern</term>`
	`</keyword>`

Table 4.11: The `metadata` Element

Element Name	`metadata`
Module	Meta-Information Module
Description	The Web was originally built for human consumption. Although everything on the Web is machine-readable, this data is not always "machine understandable." It is very difficult to automate tasks, such as indexing, filtering, and searching on the Web. The solution proposed here is to use metadata ("data about data") information to describe UPML documents published on the Web. Any information that is "about" the pattern is known as pattern metadata. The `metadata` element is a container for pattern metadata, including the pattern name, the classification scheme to which the pattern belongs to, pattern author-related information, pattern history, a brief synopsis, and keywords related to the pattern.
Data Type	Aggregate

Attribute(s)	None
Parent Element(s)	`head`
Child Element(s)	`name` (Required, Cardinality = 1), `category` (Required, Cardinality = 1), `author` (Required, Cardinality = 1..Unbounded), `date` (Optional, Cardinality = 0..Unbounded), `keyword` (Optional, Cardinality = 0..1)

Example	`<head>`
	`  <metadata>`
	`  <!-- Include Elements for Pattern, such as name, date, category, etc. -->`
	`  <name> ... </name>`
	`  </metadata>`
	`</head>`

Table 4.12: The `name` Element

Element Name	`name`
Module	Meta-Information Module
Description	The `name` element encapsulates the name of the pattern. A pattern should have a meaningful name that represents the problem it is addressing. A good name is vital, because it will become part of the design vocabulary. A name is usually a string of alphabetic characters. It can be a single word or a short phrase to refer to the knowledge a pattern encompasses. Patterns often make connections to other available patterns, and may need to be referenced from contexts outside their scope of existence (the catalog). The pattern names alone can not guarantee globally unique identification. The hierarchical nature of our pattern classification can be useful in devising a universal naming scheme for cross-referencing. We can adopt the Java

	package specifying convention to serve as a resource identifier.
Data Type	Singleton
Attribute(s)	`id` (Optional)
Parent Element(s)	`identification, metadata`
Child Element(s)	None
Example	`<name>Adapter</name>`

Table 4.13: The `term` Element

Element Name	`term`
Module	Meta-Information Module
Description	The subjects of pattern, UPML, XML, Software Engineering, and the application domain in which the pattern itself exists, all have their own "vocabulary", for example, the term can be used in the metadata keywords, extracted to form a glossary, and so on. The `term` element encapsulates the terminology related to the pattern.
Data Type	Singleton
Attribute(s)	`term-type` (Optional)
Parent Element(s)	`keyword, implementation`
Child Element(s)	None
Example	`<term>Design Pattern</term>`

Table 4.14: The `title` Element

Element Name	`title`
Module	Meta-Information Module
Description	The `title` element provides the title of the document. This title can be used either as the title of the UPML document or the title of a reference for further understanding.
Data Type	Singleton
Attribute(s)	None
Parent Element(s)	`upml, reference`

Child Element(s)	None
Example	<title>Patterns and Software: Essential Concepts and Terminology</title>

4.5 Problem Module

The Problem Module provides the problem that the pattern attempts to solve within a given context and constraints. It also specifies the forces to justify the technical decisions being made towards design solution of the problem.

Table 4.15: The `context` Element

Element Name	`context`
Module	Problem Module
Description	The `context` element outlines the conditions under which the problem recurs, and for which the solution is desirable. These conditions may be characteristics of the user, task, as well as the technical and physical organizational environment. Apart from the problem description, the context also provides criteria for determining when the pattern is applicable. Context also helps to answer when (forces) the pattern will be used. Moreover, context serves to help prioritize strong and weak forces.
Data Type	Aggregate
Attribute(s)	`id` (Optional)
Parent Element(s)	`body`
Child Element(s)	`user` (Required, Cardinality = 1), `task` (Required, Cardinality = 1..Unbounded), `platform-compatibility` (Optional, Cardinality = 0..Unbounded), `forces` (Optional, Cardinality = 0..Unbounded)
Example	<context>

	<user>Visually-disabled Users</user> <task>Simple Computing</task> <platform-compatibility>PDA </platform-compatibility > <!-- Forces --> </context>

Table 4.16: The `forces` Element

Element Name	`forces`
Module	Problem Module
Description	The `forces` element describes relevant constraint(s) of the problem and how the elements interact/conflict with one another. They are used by the engineers to justify the technical decisions for their design. Forces provide a clear picture of the complexities of the problem(s) and help defining the kinds of trade-offs that must be considered.
Data Type	Mixed
Attribute(s)	`id` (Optional)
Parent Element(s)	`body`
Child Element(s)	`term` (Optional, Cardinality = 0..Unboundeded)
Example	<forces>Limited <term>Memory</term> <term>Screen Size</term> </forces>

Table 4.17: The `platform-compatibility` Element

Element Name	`platform-compatibility`
Module	Problem Module
Description	The `platform-compatibility` element describes constrains of the technical and environmental aspects where users

	want to accomplish the task.
Data Type	Singleton
Attribute(s)	`id` (Optional)
Parent Element(s)	`context`
Child Element(s)	`None`
Example	`<platform-compatibility>PDA </platform-compatibility>`

Table 4.18: The `problem` Element

Element Name	`problem`
Module	Problem Module
Description	The `problem` element describes the problem that the pattern attempts to solve within a given context and constraints of the problem.
Data Type	Mixed
Occurrence	Required
Attribute(s)	`id` (Optional)
Parent Element(s)	`body`
Child Element(s)	`reference` (Optional, Cardinality = 0..Unboundeded), `term` (Optional, Cardinality = 0..Unboundeded)

Example	`<problem>`
	`To create an <term>eBook</term> that is`
	`<term> device independent.</term>`
	`</problem>`

Table 4.19: The `task` Element

Element Name	`task`
Module	Problem Module
Description	The `task` element describes a specific task that users want to accomplish under a certain context.

Data Type	Singleton
Attribute(s)	`id` (Optional)
Parent Element(s)	`context`
Child Element(s)	None
Example	`<task>To do a simple computation</task>`

Table 4.20: The `user` Element

Element Name	`user`
Module	Problem Module
Description	The `user` element articulates the behavioral aspects of the users from different points of views. For example, skill, education, age, disability issues etc.
Data Type	Singleton
Attribute(s)	`id` (Optional)
Parent Element(s)	`context`
Child Element(s)	None
Example	`<user>Visually disabled User</user>`

4.6 Solution Module

The Solution Module provides the design solution including its structure and strategy for the implementation of usability problems specified within a given context.

Table 4.21: The `consequence` Element

Element Name	`consequence`
Module Name	Solution Module
Description	The `consequence` element describes impact and trade-offs from the application of the pattern. It is likely that a pattern may improve one aspect at the cost of deteriorating others. In general, this section focuses on the results of using a particular pattern, and notes the pros (e.g. what aspects have been improved) and cons (e.g. what aspects have worsened) that may result from the application of the pattern.
Data Type	Mixed
Attribute(s)	`impact` (Required)
Parent Element(s)	`body`
Child Element(s)	`term` (Optional, Cardinality = 0..Unboundeded)

Example	`<consequence impact ="positive">`
	`Adapter will increase Usability.`
	`</consequence>`
	`<consequence impact ="negative">`
	`Composite will decrease Learnability.`
	`</consequence>`

Table 4.22: The `example` Element

Element Name	`example`
Module Name	Solution Module
Description	The `example` element gives instance(s) of "real-world" situations where the specified pattern has been used by the author and/or by other users. The section may also include "counter examples" (examples of cases where the pattern should have been used but was not) and "non examples" (examples of cases where the pattern should not have been used but was).

	The examples included in this section help engineers to understand the scope and domain of applicability of the pattern. They also enforce the fact that the pattern describes a proven solution. This is crucial in judging the viability of and quantifying the actual use of the pattern. The example(s) can be provided in several ways: prose, figure, markup etc. that illustrate the use of the pattern.
Data Type	Singleton
Attribute(s)	`id` (Optional)
Parent Element(s)	`strategy`
Child Element(s)	`None`

Example	`<examples>`
	`The <term>eBook</term>s are used by established book distribution enterprises (such as, Amazon.com, and Barnes and Noble) and by noted publishers (such as, Addison-Wesley).`
	`</examples>`

Table 4.23: The `implementation` Element

Element Name	`implementation`
Module Name	Solution Module
Description	The `implementation` element includes the actual implementation of the solution suggested by the pattern.
Data Type	Aggregate
Occurrence	Required
Attribute(s)	`id` (Optional)
Parent Element(s)	`body`

Child Element(s)	`structure` (Required, Cardinality = 1), `strategy` (Required, Cardinality = 1), `example` (Optional, Cardinality = 0..Unbounded), `sample-code` (Optional, Cardinality = 0..Unbounded)

Example	`<implementation>`
	`<!-- Includes structure and strategy -->`
	`</implementation>`

Table 4.24: The `rationale` Element

Element Name	`rationale`
Module Name	Solution Module
Description	Given a problem and a large collection of patterns, one might face a serious problem of choosing the right pattern. The `rationale` element describes the reasoning behind including the suitability of the pattern as a justifiable choice for solving the problem. The rationale assists a usability expert in making appropriate choice by describing how and why the pattern works, with an insight into the internal structure and key mechanisms of the system.
Data Type	Mixed
Attribute(s)	None
Parent Element(s)	`body, solution`
Child Element(s)	`term` (Optional, Cardinality = 0...Unbounded)

Example	`<solution>`
	`<rationale>`
	`The rationale for presenting a solution based on <term> XML </term> is that it has several advantages. By the application of the single-source approach, it makes possible for author to`

	document once and serve them everywhere. </rationale> </solution>

Table 4.25: The sample-code Element

Element Name	sample-code
Module Name	Solution Module
Description	The sample-code element illustrates how we can implement the pattern in any object-oriented programming language by providing some pseudo-codes.
Data Type	Singleton
Attribute(s)	None
Parent Element(s)	strategy
Child Element(s)	None

Example	<sample-code>
	class eBook {
	//constructor
	eBook();
	//destructor
	~eBook();
	//methods
	Chapter* GetChapter();
	void SetChapter(Chapter *ptrChap);
	…
	//attributes
	integer iNumberOfChapters;
	Chapter* chapter;
	…
	};
	</sample-code>

Table 4.26: The `solution` Element

Element Name	`solution`
Module Name	Solution Module
Description	The `solution` element includes a description of the actual solution provided by the pattern to solve the problem. It describes the solution approach briefly and the solution aspects in details. The solution aspects identify the pattern's structure, presentation, logic, and behavior.
Data Type	Mixed
Occurrence	Required
Attribute(s)	None
Parent Element(s)	`body`
Child Element(s)	`reference` (Optional, Cardinality = 0..Unbounded)

Example	`<solution>`
	`<!-- brief description of solution -->`
	`</solution>`

Table 4.27: The `strategy` Element

Element Name	`strategy`
Module Name	Solution Module
Description	Software developers/engineers discover and invent new ways to implement the pattern, producing new strategies for well-known patterns. To accommodate the discovered or invented ways, strategies provide an extensibility point for each pattern. The `strategy` element includes a description of different ways a pattern can be implemented.
Data Type	Mixed
Attribute(s)	Object-type
Parent Element(s)	`implementation`
Child Element(s)	`example`(Optional, Cardinality = 0..Unbounded), `sample-`

	code (Optional, Cardinality = 0..Unbounded), figure (Optional, Cardinality = 0..Unbounded), term (Optional, Cardinality = 0..Unbounded)
Example	<strategy> The tree structure of an <term>eBook </term> can be implemented in several ways. The most reader-friendly way is to follow the format of a regular book. There is a cover page, table of contents, and a set of chapters. The items in the table of contents link to preface, individual chapters, and sections. ... </strategy>

Table 4.28: The structure Element

Element Name	structure
Module Name	Solution Module
Description	The structure element includes a description of a pattern at a high level abstraction. This can be done by using prose or by using a visual modeling notation. For example, use of the UML diagrams can be drawn to show the basic structure and data flow diagram of a solution.
Data Type	Mixed
Attribute(s)	None
Parent Element(s)	implementation
Child Element(s)	term (Optional, Cardinality = 0..Unbounded)
Example	<structure> The solution structure of eBook is a

	tree where all the nodes are accessible from the root. </structure>

4.7 Structure Module

The Structure Module provides the high level abstraction of Pattern Documentation using XML-based notations for Web publishing.

Table 4.29: The body Element

Element Name	body
Module Name	Structure Module
Description	The body element is the container that holds the actual content of a pattern.
Data Type	Aggregate
Occurrence	Required
Attribute(s)	None
Parent Element(s)	pattern
Child Element(s)	context (Required, Cardinality = 1), problem (Required, Cardinality = 1), forces (Required, Cardinality = 1..Unbounded), solution (Required, Cardinality = 1), rationale (Required, Cardinality = 1), implementation (Required, Cardinality = 1), consequence (Required, Cardinality = 1..Unbounded)
Example	<body> <!-- Other Elements Here --> </body>

Table 4.30: The `head` Element

Element Name	`head`
Module Name	Structure Module
Description	The `head` element is a container of information, which is not directly related to the patterns content. However, it provides the identification information of the pattern.
Data Type	Aggregate
Attribute(s)	None
Parent Element(s)	`Pattern`
Child Element(s)	`metadata` (Required, Cardinality = 1)

Example	`<head>`
	`  <metadata>`
	`  <!-- Other Elements Here -->`
	`  </metadata>`
	`</head>`

Table 4.31: The `pattern` Element

Element Name	`pattern`
Module Name	Structure Module
Description	The `pattern` element is a container for pattern information organized into head and body.
Data Type	Mixed
Attribute(s)	`id` (Required), `version` (Required)
Parent Element(s)	`upml`
Child Element(s)	`head` (Required, Cardinality = 1), body (Required, Cardinality = 1)

Example	`<pattern>`
	`  <head>`
	`  <!-- Other Elements Here -->`
	`  </head>`

	<body> <!-- Other Elements Here --> </body> </pattern>

Table 4.32: The upml Element

Element Name	upml
Module Name	Structure Module
Description	The upml element is the root of a UPML document. It can contain one or more pattern elements.
Data Type	Aggregate
Attribute(s)	id (Optional), version (Required), xml:lang (Required), xml:space (Optional),
Parent Element(s)	None
Child Element(s)	pattern (Required, Cardinality = 1..Unbounded)
Example	<upml version = "1.0" xml:lang = "en"> <pattern> <!-- Other Elements Here --> </pattern> </upml>

4.8 UPML Attribute Definitions

UPML defines eight attributes: event, id, impact, object-type, relation, term-type, uri and version. They are described in the following tables. Besides, UPML borrows two attributes from the XML 1.0 [W3C00a], which are: xml:lang and xml:space. xml:lang is used to indicate the natural language being used in the document. xml:space gives directions to the processor for

controlling white spaces. UPML also uses an attribute from Namespace in XML specification [W3C99], `xmlns`, to uniquely identify its elements and attributes.

A "|" indicates an Exclusive-OR.

Table 4.33: The `event` Attribute

Attribute Name	`event`
Description	The `event` attribute provides the context of time stamping. For example, the context can be related to the evolution, creation, publication, modification, etc. of the patterns.
Data Type	creation | publication | revision
Related Element(s)	`date`

Table 4.34: The `id` Attribute

Attribute Name	`id`
Description	The `id` attribute uniquely identifies an element within a document. Its value is an XML identifier.
Data Type	XML ID. For acceptable values, see the XML 1.0 Specification [W3C00a]
Related Element(s)	`implementation, pattern, reference, upml`

Table 4.35: The `impact` Attribute

Attribute Name	`impact`
Description	The `impact` attribute provides the type of consequence that a pattern may have as a result which is applied at a given situation.
Data Type	positive | negative
Related Element(s)	`consequence`

Table 4.36: The `object-type` Attribute

Attribute Name	`object-type`
Description	The `object-type` attribute states the types of object included in the UPML document. The possibilities are: figure, markup, or sample-code. A figure can be used to describe the structure of the pattern solution and the pattern implementation. Markup provides an implementation of the pattern solution in the form of full/partial document based on a markup language (preferably XML). A sample-code (pseudo code of a program, or a script, or a style-sheet) provides an implementation of the pattern solution based on a formal (non-markup) language.
Data Type	figure | markup | sample-code
Related Element(s)	`strategy`

Table 4.37: The `relation` Attribute

Attribute Name	`relation`
Description	The `relation` attribute symbolizes the type of relationship between the related pattern and the pattern under study. The relationship can be categorized as superordinate, subordinate, sibling/neighboring, or competitor.
Data Type	superordinate | subordinate | sibling | competitor
Related Element(s)	`related-pattern`

Table 4.38: The `term-type` Attribute

Attribute Name	`term-type`
Description	The `term-type` attribute symbolizes the nature of the term that may exist in various forms.
Data Type	abbreviation | concept | principle | technology | tool
Related Element(s)	`term`

Table 4.39: The `uri` Attribute

Attribute Name	`uri`
Description	The `uri` attribute provides the URI of the content it is associated with.
Data Type	anyURI. For acceptable values, see IETF RFC 2396 [Berners+98]
Related Element(s)	`link`

Table 4.40: The `version` Attribute

Attribute Name	`version`
Description	The `version` attribute provides a numerical value of the release date of either the UPML document or the pattern.
Data Type	decimal
Related Element(s)	`pattern, upml`

4.9 UPML Identification

UPML documents need to be identified by user agents and processors. This section describes the facilities provided by UPML; e.g. UMPL MIME type, namespace, files extension, etc.

UPML Internet Media (MIME) Type:

In accordance with IETF RFC 3023 [Kohn+01], UPML documents should be reserved

```
application/upml+xml
```

as its MIME type. Until user agent recognizes the '+xml' suffix for XML-based MIME types, UPML documents may be served as the media type '`text/xml`'.

UPML Namespace:

The XML Namespace assigned to UPML is

```
http://localhost:80/upml
```

The prefix "`upml:`" is used by convention to denote the UPML namespace. However, any prefix can be used. This is necessary when authoring and delivering UPML documents, particularly those containing any non-UPML markup fragments.

UPML File Extension:

The `upml` is recommended as the filename extension for UPML documents.

4.10 UPML Conformance

The section provides the desirable criteria for conformance of a UPML document and that of a UPML processor.

4.10.1 UPML Document Conformance

A UPML conforming document must satisfy the following criteria:

1. It must conform to the XML 1.0 specification [W3C00a].
2. It must specify a character encoding in the XML processing instruction.

3. It must declare `upml` as its root element.
4. It must contain at least one `pattern` element.
5. It must validate against the UPML schema (please see "Appendix B" for details).
6. If any namespaces other than UPML are used in the document, it must conform to the namespaces in XML [W3C99a]. Particularly, the namespaces need to be specified at the root element of a UPML document, if it includes any fragments of non-UPML markup. On the other hand, to separate any UPML fragments from non-UPML fragments, it may be specified with the prefix "`upml:`" for all UPML elements.
7. It shall conform to Cascading Style Sheets (CSS), Level 2 specification [W3C98] if there is any use of it.
8. It shall conform to associating eXtensible Style-sheet Languages (XSL) with XML Documents [W3C99b], if it refers to any external style-sheets.

4.10.2 UPML Processor Conformance

A UPML processor must be conforming to an XML processor as defined in Section 5 of the XML 1.0 Specification [W3C00a]. Here, the XML schema has been developed using TIBCO's TURBO XML. Further the schema and UPML documents have been validated using Xerces parser, which is easy to bind with Apache Server.

Xerces (named after the Xerces Blue butterfly) parser provides the world-class XML parsing and generation. Fully-validating Xerces parsers are available for both Java and C++, implementing the W3C XML and Document Object Model, DOM (Level 1 and 2) standards, as well as the de facto Standard API for XML, SAX (version 2) standard. These parsers are highly modular and configurable. Initial support for XML Schema (draft W3C standard) is also provided [Benoit02] with these parsers.

Chapter 5

Conclusion and Future Work

Although patterns have several advantages in the software engineering domain, there has been a big communication gap among the professional groups. Moreover, there was a lack of consistency of elements in pattern documentation as well as organization of patterns. The proposed format with a unified set of elements will increase the acceptability of using pattern in the software engineering domain and hence ease to reuse of architecture and design by capturing and disseminating the experts' knowledge and experience. In addition, the suggested format addresses the consistency and organizational issues of a pattern to make it more clear, understandable and reusable.

The study has brought forth the following contributions / recommendations:

Firstly, this thesis has analyzed some of the most popular formats for pattern documentation, proposed by different individuals and communities, while highlighting their strengths and weaknesses. This study has proposed a comprehensive format, comprising the following seven elements: identification, problem description, context, forces, solution, implementation strategies, and consequences.

Secondly, a 3D-Reference Model of pattern properties, which describes the major concerns of different professional groups related to patterns, has been proposed. This decreases the unwanted gap in communication among the various professional groups including Patterns Writers, Usability Experts and Software Developers.

Thirdly, the study has proposed a classification scheme, which organizes patterns according to granularity, functionality, and structural principles, for choosing the appropriate pattern(s) effectively and efficiently.

Finally, syntax and semantics of the Usability Pattern Markup Language (UPML) specification, based on XML notations, has been developed. This provides the higher level abstraction and a validation schema for pattern documentation. The proposed specification, as being the first version, has been named as 'UPML 1.0'.

The outcome of the study may be revised based on the feedback from various industrial projects in the future. Empirical studies, questioning the pattern writers, usability experts, and software developers may also contribute to the improvement of this study. After collecting feedbacks and conducting empirical studies, the XML schema for validating the UPML documents may be reviewed to develop a newer version.

There are several other projects that have been carried out by the Human-Centered Software Engineering (HCSE) group in the Department of Computer Science at Concordia University. Among these projects, the following two are mentioned (as examples) which will be benefited by the outcome of this thesis work. At the same time, this study may be enriched by the above projects.

1. **UPADE** (Usability Pattern Assisted DEsign), a Java based editor for documenting, editing, developing pattern-oriented design, and combining existing usability patterns. Pattern oriented design can be enriched by using the UPML 1.0. On the other hand, UPADE tool can help UPML documents writers, users by providing them with a UI (User Interface) for authoring and validating usability patterns.

2. **MOUDIL** (Montreal Online Usability patterns DIgital Library), a MySQL based relational database for storing patterns along with HTML based online browsing, navigation, and searching tool, can be improved by using the UPML 1.0. On the other hand, MOUDIL can be of assistance to the UPML documentation by providing the online digital repository of usability patterns.

The study provides a new outlook on the format for pattern documentation following the XML approach and justifies the needs of this study.

References

A

[Alan+02] Alan S. and James R. T. *Design Patterns Explained – A New Perspective on Object-Oriented Design*, Addison-Wesley, 2002.

[Alexander+77] Christopher Alexander et. al, *A pattern language*, New York: Oxford University Press, 1977.

[Alexander79] Christopher Alexander, *The timeless way of Building*, New York: Oxford University Press, 1979.

[Alur+01] Deepak Alur, John Crupi, and Dan Malks, *Pattern Template in sun Java Center J2EE Patterns*, Sun Java Center, March 2001, http://developer.java.sun.com/developer/technicalArticles/J2EE/patterns/PatternTemplate.html

[Appleton00] Appleton, B., *Patterns and Software: Essential Concepts and Terminology*, 2000, http://www.entract.com/~bradapp/docs/patterns-intro.html

B

[Benoit02] Benoit Marchal, *XML By Example (Second Edition)*, QUE, USA, 2002.

[Berners+98] Berners-Lee, T., Fielding R., Masinter, L., *RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax*, IETF, August 1998, http://www.ietf.org/rfc/rfc2396.txt

[Brochers00] Brochers J.O., *A Pattern Approach to Interaction Design,* in *Proceedings of the DIS 2000 International Conference on Designing Interactive Systems,* New York, Aug 16-19, 2000, ACM Press, pp 369-378.

[Bryan01] Bryan, M. (Project Editor), *ISO/IEC JTC 1/SC34 Information Technology–Document Description and Processing Language*, Working Draft, October 22, 2001

[Buschmann+01] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, Pattern-Oriented Software Architecture, John Wiley & Sons Ltd., Feb 2001

[Buschmann+94] F. Buschmann, R. Meunier, *A System of Patterns*, Proceedings of PLoP'94, pp 325-343, PLoP94

C

[Civello93] Civello, F., *Roles for Composite Objects in Object Oriented Analysis and Design, Proceedings, OOPSLA'93,* ACM, 1993

[Coad+95] Peter Coad, David North and Mark Mayfield, *Object Models: Strategies, Patterns, and Applications*, Prentice Hall, NJ, 1995

[Coplien94a] Coplien, J.O., *Progress on patterns: Highlights of PLoP'94,* In *Proceedings of Object Expo Europe,* 1994, ftp://st.cs.uiuc.edu/pub/patterns/papers/ObjectExpoPLoP.ps

[Coplien94b] Coplien, J.O., *Description of the Envelope-Letter Idiom,* OOPSLA Pattern Mailing Reflector, 1994

[Coplien+95] Coplien, J.O., & Schmidt, D.C., *Pattern Language of Program Design*, Addison-Wesley, MA, 1995

[Coram+98] Coram T. and Lee J., *Experiences – A Pattern Language for User Interface Design*, 1998, http://www.maplefish.com/todd/papers/Experiences.html

[Cunningham95] Ward Cunningham, *WikiWikiWeb*, 1995 (posted, but updated most frequently), http://c2.com/cgi/wiki?WelcomeVisitors

E

[Eisenhauer+94] R. Eisenhauer, S. Kumsta, F. Miralles, K. Mobious, U Steinmuller, P. Stobbe, C. Vester, *Hand Book for Software Architecture*, Siemens Nixdorf Information System AG, Internal Report, 1994

G

[Gamma+93] E. Gamma, R. Helm, and J. Vlissides, *Design Patterns: Abstraction and Reuse of Object-Oriented Design,* ECOOP 7 Proceedings. Berlin: Springer-Verlag, 1993, pp 406-431.

[Gamma+95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison Wesley, 1995.

[Granlund+01] Granlund A., Lafreniere D., and Carr A., *A Pattern-Supported Approach to the User Interface Design*, in *9th International Conference on HCI,* New Orleans, Aug 5-10, 2001, http://www.sm.luth.se/~david/papers/HCIInt2001Final.pdf

[Griffiths+01] Griffiths, R., Pemberton, L., *Don't Write Guidelines, Write Patterns*, University of Brighton, Brighton, UK, 2001.

H

[Hillside00] Hillside.net, *Patterns Home Page*: *Your Patterns Library*, 2000, http://www.hillside.net/patterns/

I

[IBM99a] IBM, *Patterns for E-Business*, IBM, 1999.

[IBM99b] IBM, *Xenna*, IBM Alpha Works, March 1999 (posted), June 2000 (updated), http://www.alphaworks.ibm.com/tech/xeena

[IBM99c] IBM, *XML Generator*, IBM Alpha Works, September 1999 (posted), August 2001 (updated), http://www.alphaworks.ibm.com/tech/xmlgenerator

[IEEE87] IEEE, *Recommended Practice for Software Design Descriptions*, IEEE STD 1016-1987, 1987.

[IEEE98] IEEE, *Recommended Practice for Software Requirement specifications*, IEEE STD 830-1998, 1998.

J

[Johnson92] Johnson, R., *Documenting Frameworks Using Patterns, Proceedings, OOPSLA' 92*, ACM, 1992

K

[Knuth92] Knuth, Donald E., *Literate Programming*, CSLI Lecture Notes, November 27, 1992

[Kohn+01] Kohn, D., Murata, M., St. Laurent, S., *RFC 3023: XML Media Types*, IETF, January 2001

L

[Linda98] Linda Rising, *The Patterns Handbook – Techniques, Strategies, and Applications,* Cambridge University Press, UK, 1998.

N

[Ning01] Ning, N., *A Usability Pattern Language and Tool for Web Application*, Master Thesis, Concordia University, 2001

P

[Parnas79] Parnas, D., *Designing Software for Ease of Extension and Contraction, IEEE Transactions on Software Engineering*, March 1979

[Pemberton+99] Pemberton L., and Griffiths R., *The Brighton Usability Pattern, Collection*, 1999, http://www.it.bton.ac.uk/Research/patterns/home.html

[Portland95] Portland, Portland Pattern Repository, 1995, http://c2.com/ppr/

[Prieto+89] Prieto-Diaz, R., & G. Arango, Domain Analysis: Acquisition of Reusable Information for Software Construction, IEEE Computer Society Press, 1989

R

[RFC98] T. Berners-Lee, R. Fielding, L. Masinter (editors). IETF (Internet Engineering Task Force) *RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax*, eds. August 1998

S

[Seffah+00] Javahery, H. and Ahmed, S., *A Model for Usability Pattern-Oriented Designs*, Concordia University, 2000

T

[Tidwell99] Tidwell J., *Common Ground: A Pattern Language for Human-Computer Interface Design,* 1999, http://www.mit.edu/~jtidwell/common_ground.html

W

[W3C] W3C's XML Schema Definition Language (XSDL)

[W3C00a] Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E. (Editors), *Extensible Markup Language (XML) 1.0 (Second Edition)*, W3C Recommendation, October 2000, http://www.w3.org/TR/REC-xml

[W3C00b] Davis, M., Hors, A., Robie, J., Wood, L. et. Al. (Editors), *Document Object Model (DOM) Level2 Core Specification"*, W3C Recommendation, November 2000, http://www.w3.org/TR/DOM-Level-2-Core

[W3C01] Fallside, David C. (Editor), *XML Schema Part 0: Primer, W3C Recommendation*, May 2001, http://www.w3.org/TR/xmlschema-0

[W3C02] Dardailler, D., Palmer, S. B. (Editors), *XML Accessibility Guidelines, W3C Working Draft3*, October 2002, http://www.w3.org/TR/xag

[W3C98] Bos, B., Lie, H-W., Lilley C., Jacobs, I. (Editors), *Cascading Style Sheets: Level 2 (CSS2) Specification*, W3C Recommendation, May 1998

[W3C99a] Bray, T., Hollander, D., Layman, A. (Editors), *Namespace in XML, W3C Recommendation*, January 1999

[W3C99b] Clark, J. (Editor), *XSL Transformations (XSLT) Version 1.0*, W3C Recommendation, November 1999, http://www.w3c.org/TR/xml-stylesheet

[Welie+00] Van Welie M. Van der Veer G.C., and A. Eliens, *Patterns as Tools for User Interface Design, in International Workshop on Tools for Working with Guidelines*, October 7-8, 2000, Biarritz France, pp 313-324.

[Welie03] Martijn van Welie, *Patterns in Interaction Design*, 2003 (posted, but updates most frequently), http://www.welie.com/

Z

[Zimmer94] W. Zimmer, *Relationships Between Design Patterns*, Proceedings of PLoP'94, pp 345-363, PLoP94.

Appendix A

Abstract Factory — Object Creational

Intent

Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

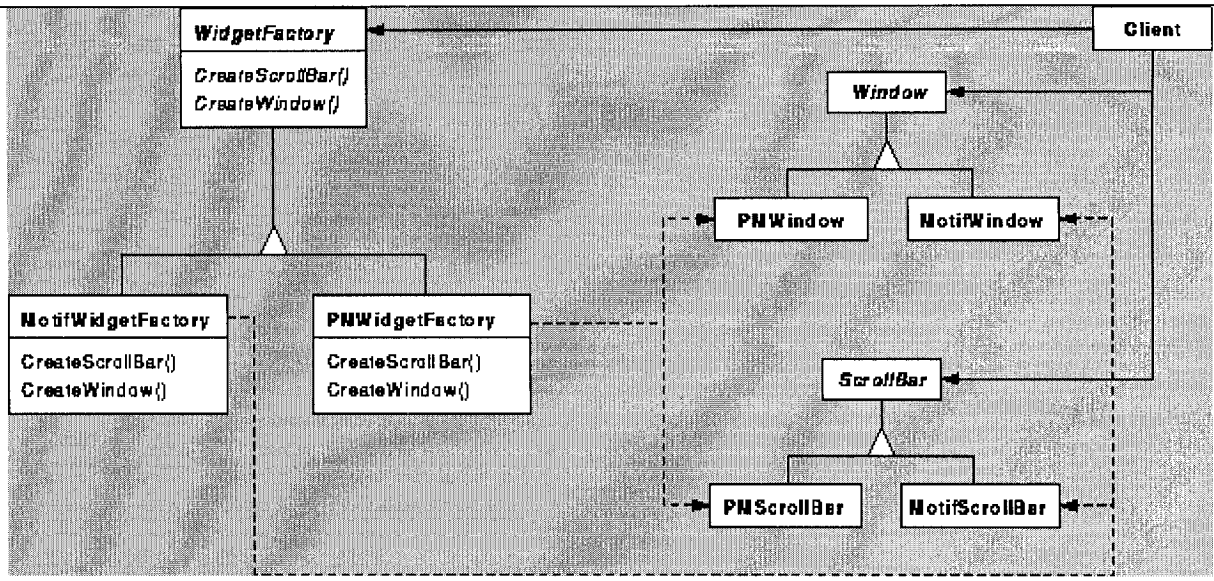
Also Known As

Kit

Motivation

Consider a user interface toolkit that supports multiple look-and-feel standards, such as Motif and Presentation Manager. Different look-and-feels define different appearances and behaviors for user interface "widgets" like scroll bars, windows, and buttons. To be portable across look-and-feel standards, an application should not hard-code its widgets for a particular look and feel. Instantiating look-and-feel-specific classes of widgets throughout the application make it hard to change the look and feel later.

We can solve this problem by defining an abstract WidgetFactory class that declares an interface for creating each basic kind of widget. There's also an abstract class for each kind of widget, and concrete subclasses implement widgets for specific look-and-feel standards. WidgetFactory's interface has an operation that returns a new widget object for each abstract widget class. Clients call these operations to obtain widget instances, but clients aren't aware of the concrete classes they're using. Thus clients stay independent of the prevailing look and feel.



There is a concrete subclass of WidgetFactory for each look-and-feel standard. Each subclass implements the operations to create the appropriate widget for the look and feel. For example, the CreateScrollBar operation on the MotifWidgetFactory instantiates and returns a Motif scroll bar, while the corresponding operation on the PMWidgetFactory returns a scroll bar for Presentation Manager. Clients create widgets solely through the WidgetFactory interface and have no knowledge of the classes that implement widgets for a particular look and feel. In other words, clients only have to commit to an interface defined by an abstract class, not a particular concrete class.

A WidgetFactory also enforces dependencies between the concrete widget classes. A Motif scroll bar should be used with a Motif button and a Motif text editor, and that constraint is enforced automatically as a consequence of using a MotifWidgetFactory.

Applicability

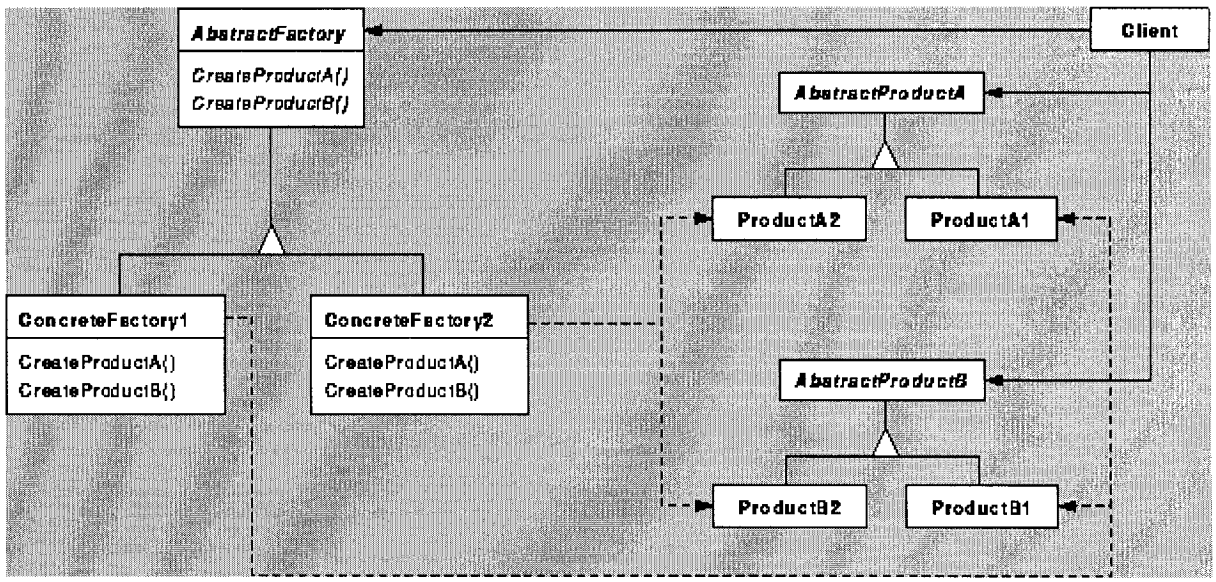
Use the Abstract Factory pattern when

- a system should be independent of how its products are created, composed, and represented.
- a system should be configured with one of multiple families of products.
- a family of related product objects is designed to be used together, and you need to

enforce this constraint.

- you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

Structure



Participants

- **AbstractFactory** (WidgetFactory)
  - declares an interface for operations that create abstract product objects.
- **ConcreteFactory** (MotifWidgetFactory, PMWidgetFactory)
  - implements the operations to create concrete product objects.
- **AbstractProduct** (Window, ScrollBar)
  - declares an interface for a type of product object.
- **ConcreteProduct** (MotifWindow, MotifScrollBar)
  - defines a product object to be created by the corresponding concrete factory.
  - implements the AbstractProduct interface.
- **Client**

- uses only interfaces declared by AbstractFactory and AbstractProduct classes.

Collaborations

- Normally a single instance of a ConcreteFactory class is created at run-time. This concrete factory creates product objects having a particular implementation. To create different product objects, clients should use a different concrete factory.
- AbstractFactory defers creation of product objects to its ConcreteFactory subclass.

Consequences

The Abstract Factory pattern has the following benefits and liabilities:

1. *It isolates concrete classes.* The Abstract Factory pattern helps you control the classes of objects that an application creates. Because a factory encapsulates the responsibility and the process of creating product objects, it isolates clients from implementation classes. Clients manipulate instances through their abstract interfaces. Product class names are isolated in the implementation of the concrete factory; they do not appear in client code.
2. *It makes exchanging product families easy.* The class of a concrete factory appears only once in an application---that is, where it's instantiated. This makes it easy to change the concrete factory an application uses. It can use different product configurations simply by changing the concrete factory. Because an abstract factory creates a complete family of products, the whole product family changes at once. In our user interface example, we can switch from Motif widgets to Presentation Manager Widgets simply by switching the corresponding factory objects and recreating the interface.
3. *It promotes consistency among products.* When product objects in a family are designed to work together, it's important that an application use objects from only one family at a time. AbstractFactory makes this easy to enforce.
4. *Supporting new kinds of products is difficult.* Extending abstract factories to produce new kinds of Products isn't easy. That's because the AbstractFactory interface fixes the set of 0products that can be created. Supporting new kinds of products requires

extending the factory interface, which involves changing the AbstractFactory class and all of its subclasses. We discuss one solution to this problem in the Implementation section.

Implementation

Here are some useful techniques for implementing the Abstract Factory pattern.

1. *Factories as singletons.* An application typically needs only one instance of a ConcreteFactory per product family. So it's usually best implemented as a Singleton.
2. *Creating the products.* AbstractFactory only declares an *interface* for creating products. It's up to ConcreteProduct subclasses to actually create them. The most common way to do this is to define a factory method for each product. A concrete factory will specify its products by overriding the factory method for each. While this implementation is simple, it requires a new concrete factory subclass for each product family, even if the product families differ only slightly.

   If many product families are possible, the concrete factory can be implemented using the Prototype pattern. The concrete factory is initialized with a prototypical instance of each product in the family, and it creates a new product by cloning its prototype. The Prototype-based approach eliminates the need for a new concrete factory class for each new product family.

   Here's a way to implement a Prototype-based factory in Smalltalk. The concrete factory stores the prototypes to be cloned in a dictionary called `partCatalog`. The method `make:` retrieves the prototype and clones it:

```
make: partName
    ^ (partCatalog at: partName) copy
```

   The concrete factory has a method for adding parts to the catalog.

```
addPart: partTemplate named: partName
    partCatalog at: partName put: partTemplate
```

   Prototypes are added to the factory by identifying them with a symbol:

```
aFactory addPart: aPrototype named: #ACMEWidget
```

A variation on the Prototype-based approach is possible in languages that treat classes as first-class objects (Smalltalk and Objective C, for example). You can think of a class in these languages as a degenerate factory that creates only one kind of product. You can store *classes* inside a concrete factory that create the various concrete products in variables, much like prototypes. These classes create new instances on behalf of the concrete factory. You define a new factory by initializing an instance of a concrete factory with *classes* of products rather than by sub-classing. This approach takes advantage of language characteristics, whereas the pure Prototype-based approach is language-independent. Like the Prototype-based factory in Smalltalk just discussed, the class-based version will have a single instance variable `partCatalog`, which is a dictionary whose key is the name of the part. Instead of storing prototypes to be cloned, `partCatalog` stores the classes of the products. The method `make:` now looks like this:

```
make: partName
    ^ (partCatalog at: partName) new
```

3. *Defining extensible factories.* AbstractFactory usually defines a different operation for each kind of product it can produce. The kinds of products are encoded in the operation signatures. Adding a new kind of product requires changing the AbstractFactory interface and all the classes that depend on it.

   A more flexible but less safe design is to add a parameter to operations that create objects. This parameter specifies the kind of object to be created. It could be a class identifier, an integer, a string, or anything else that identifies the kind of product. In fact with this approach, AbstractFactory only needs a single ``Make" operation with a parameter indicating the kind of object to create. This is the technique used in the Prototype- and the class-based abstract factories discussed earlier.

   This variation is easier to use in a dynamically typed language like Smalltalk than in a statically typed language like C++. You can use it in C++ only when all objects have the same abstract base class or when the product objects can be safely coerced to the correct type by the client that requested them. The implementation section of Factory

Method shows how to implement such parameterized operations in C++.

But even when no coercion is needed, an inherent problem remains: All products are returned to the client with the *same* abstract interface as given by the return type. The client will not be able to differentiate or make safe assumptions about the class of a product. If clients need to perform subclass-specific operations, they won't be accessible through the abstract interface. Although the client could perform a downcast (e.g., with `dynamic_cast` in C++), that's not always feasible or safe, because the downcast can fail. This is the classic trade-off for a highly flexible and extensible interface.

Sample Code

We'll apply the Abstract Factory pattern to creating the mazes we discussed at the beginning of this chapter.

Class `MazeFactory` can create components of mazes. It builds rooms, walls, and doors between rooms. It might be used by a program that reads plans for mazes from a file and builds the corresponding maze. Or it might be used by a program that builds mazes randomly. Programs that build mazes take a `MazeFactory` as an argument so that the programmer can specify the classes of rooms, walls, and doors to construct.

```
class MazeFactory {
public:
    MazeFactory();

    virtual Maze* MakeMaze() const
        { return new Maze; }
    virtual Wall* MakeWall() const
        { return new Wall; }
    virtual Room* MakeRoom(int n) const
        { return new Room(n); }
    virtual Door* MakeDoor(Room* r1, Room* r2) const
        { return new Door(r1, r2); }
};
```

Recall that the member function `CreateMaze` builds a small maze consisting of two rooms with a door between them. `CreateMaze` hard-codes the class names, making it difficult to create mazes with different components.

Here's a version of `CreateMaze` that remedies that shortcoming by taking a `MazeFactory` as a parameter:

```
Maze* MazeGame::CreateMaze (MazeFactory& factory) {
    Maze* aMaze = factory.MakeMaze();
    Room* r1 = factory.MakeRoom(1);
    Room* r2 = factory.MakeRoom(2);
    Door* aDoor = factory.MakeDoor(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, factory.MakeWall());
    r1->SetSide(East, aDoor);
    r1->SetSide(South, factory.MakeWall());
    r1->SetSide(West, factory.MakeWall());

    r2->SetSide(North, factory.MakeWall());
    r2->SetSide(East, factory.MakeWall());
    r2->SetSide(South, factory.MakeWall());
    r2->SetSide(West, aDoor);

    return aMaze;
}
```

We can create `EnchantedMazeFactory`, a factory for enchanted mazes, by subclassing `MazeFactory`. `EnchantedMazeFactory` will override different member functions and return different subclasses of `Room`, `Wall`, etc.

```
class EnchantedMazeFactory : public MazeFactory {
public:
```

```
        EnchantedMazeFactory();

        virtual Room* MakeRoom(int n)  const
            { return new EnchantedRoom(n, CastSpell()); }

        virtual Door* MakeDoor(Room* r1, Room* r2)  const
            { return new DoorNeedingSpell(r1, r2); }

    protected:
        Spell* CastSpell() const;
    };
```

Now suppose we want to make a maze game in which a room can have a bomb set in it. If the bomb goes off, it will damage the walls (at least). We can make a subclass of `Room` keep track of whether the room has a bomb in it and whether the bomb has gone off. We'll also need a subclass of `Wall` to keep track of the damage done to the wall. We'll call these classes `RoomWithABomb` and `BombedWall`.

The last class we'll define is `BombedMazeFactory`, a subclass of `MazeFactory` that ensures walls are of class `BombedWall` and rooms are of class `RoomWithABomb`. `BombedMazeFactory` only needs to override two functions:

```
Wall* BombedMazeFactory::MakeWall () const {
    return new BombedWall;
}

Room* BombedMazeFactory::MakeRoom(int n) const {
    return new RoomWithABomb(n);
}
```

To build a simple maze that can contain bombs, we simply call `CreateMaze` with a `BombedMazeFactory`.

```
MazeGame game;
BombedMazeFactory factory;

game.CreateMaze(factory);
```

`CreateMaze` can take an instance of `EnchantedMazeFactory` just as well to build enchanted mazes.

Notice that the `MazeFactory` is just a collection of factory methods. This is the most common way to implement the Abstract Factory pattern. Also note that `MazeFactory` is not an abstract class; thus it acts as both the AbstractFactory *and* the ConcreteFactory. This is another common implementation for simple applications of the Abstract Factory pattern. Because the `MazeFactory` is a concrete class consisting entirely of factory methods, it's easy to make a new `MazeFactory` by making a subclass and overriding the operations that need to change.

`CreateMaze` used the `SetSide` operation on rooms to specify their sides. If it creates rooms with a `BombedMazeFactory`, then the maze will be made up of `RoomWithABomb` objects with `BombedWall` sides. If `RoomWithABomb` had to access a subclass-specific member of `BombedWall`, then it would have to cast a reference to its walls from `Wall*` to `BombedWall*`. This downcasting is safe as long as the argument *is* in fact a `BombedWall`, which is guaranteed to be true if walls are built solely with a `BombedMazeFactory`.

Dynamically typed languages such as Smalltalk don't require downcasting, of course, but they might produce run-time errors if they encounter a `Wall` where they expect a *subclass* of `Wall`. Using Abstract Factory to build walls helps prevent these run-time errors by ensuring that only certain kinds of walls can be created.

Let's consider a Smalltalk version of `MazeFactory`, one with a single `make` operation that takes the kind of object to make as a parameter. Moreover, the concrete factory stores the classes of the products it creates.

First, we'll write an equivalent of `CreateMaze` in Smalltalk:

```
createMaze: aFactory
    | room1 room2 aDoor |
    room1 = (aFactory make: #room) number: 1.
    room2 = (aFactory make: #room) number: 2.
    aDoor = (aFactory make: #door) from: room1 to: room2.
```

```
        room1 atSide: #north put: (aFactory make: #wall).
        room1 atSide: #east put: aDoor.
        room1 atSide: #south put: (aFactory make: #wall).
        room1 atSide: #west put: (aFactory make: #wall).
        room2 atSide: #north put: (aFactory make: #wall).
        room2 atSide: #east put: (aFactory make: #wall).
        room2 atSide: #south put: (aFactory make: #wall).
        room2 atSide: #west put: aDoor.
        ^ Maze new addRoom: r1; addRoom: r2; yourself
```

As we discussed in the Implementation section, `MazeFactory` needs only a single instance variable `partCatalog` to provide a dictionary whose key is the class of the component. Also recall how we implemented the `make:` method:

```
make: partName
    ^ (partCatalog at: partName) new
```

Now we can create a `MazeFactory` and use it to implement `createMaze`. We'll create the factory using a method `createMazeFactory` of class `MazeGame`.

```
createMazeFactory
    ^ (MazeFactory new
        addPart: Wall named: #wall;
        addPart: Room named: #room;
        addPart: Door named: #door;
        yourself)
```

A `BombedMazeFactory` or `EnchantedMazeFactory` is created by associating different classes with the keys. For example, an `EnchantedMazeFactory` could be created like this:

```
createMazeFactory
    ^ (MazeFactory new
        addPart: Wall named: #wall;
        addPart: EnchantedRoom named: #room;
        addPart: DoorNeedingSpell named: #door;
        yourself)
```

Known Uses

InterViews uses the ``Kit'' suffix to denote AbstractFactory classes. It defines WidgetKit and DialogKit abstract factories for generating look-and-feel-specific user interface objects. InterViews also include a LayoutKit that generates different composition objects depending on the layout desired. For example, a layout that is conceptually horizontal may require different composition objects depending on the document's orientation (portrait or landscape).

ET++ uses the Abstract Factory pattern to achieve portability across different window systems (X Windows and SunView, for example). The WindowSystem abstract base class defines the interface for creating objects that represent window system resources (MakeWindow, MakeFont, MakeColor, for example). Concrete subclasses implement the interfaces for a specific window system. At run-time, ET++ creates an instance of a concrete WindowSystem subclass that creates concrete system resource objects.

Related Patterns

AbstractFactory classes are often implemented with factory methods, but they can also be implemented using Prototype. A concrete factory is often a singleton.

Appendix B

UPML Schema

```
<? xml version ="1.0" encoding ="UTF-8"? >
<!-- Generated by Turbo XML 2.3.1 -->
<schema
    xmlns = "http://www.w3.org/2001/XMLSchema"
    targetNameSpace = "http://localhost:80/upml"
    xmlns:xs = "http://www.w3.org/2001/XMLSchema"
    xmlns:upml = "http://localhost:80//upml"
    elementFromDefault = "qualified"
    attributeFromDefault = "qualified">

<!-- ..... -->
    <annotation>
      <documentation xml:lang = "en">

          File : upml.xsd
          Author : Faridul Islam
          Date: March 1, 2003
          Version: 1.0
          Description: XML Schema for Generalized Format of Pattern Documentation
                       This Schema is identified by the local XML Namespace:
                            http://localhost:80/upml
          Copyright © 2003 Faridul Islam
          Permission is granted to copy, distribute, and/or modify this document under
          the terms of the GNU Free Documentation License.

      </documentation>
    </annotation>

<!-- ..... -->
<!-- Import xml:lang and xml:space Attributes..... -->
<!-- ..... -->
    <import namespace = "http://www.w3c.org/XML/1998/namespace"
            schemaLocation = "http://www.w3c.org/2001/xml.xsd">
    </import>

<!-- ..... -->
<!-- ASSOCIATION MODULE ..... -->
<!-- ..... -->

<!-- The category Element .....-->
    <element name="category">
```

```
        <complexType>
          <choice>
              <element ref="upml:term" minOccurs="1" maxOccurs="unbounded"/>
          </choice>
          <attribute name="id" type="ID" use="optional"/>
        </complexType>
    </element>

<!-- The link Element ..... -->
    <element name="link">
        <complexType>
          <simpleContent>
              <extension base = "string">
                  <attribute name="id" type="ID" use="optional"/>
                  <attribute name="uri" type="anyURI" use="required"/>
              </extension>
          </simpleContent>
        </complexType>
    </element>

<!-- The reference Element ..... -->
    <element name="reference">
        <complexType>
          <choice>
              <element ref = "upml:title"/>
              <element ref = "upml:author" minOccurs="1" maxOccurs="unbounded"/>
              <element ref = "upml:date" minOccurs="1" maxOccurs="unbounded"/>
              <element ref = "upml:link"/>
          </choice>
              <attribute name="id" type="ID" use="required"/>
        </complexType>
    </element>

<!-- The related-pattern Element ..... -->
    <element name="related-pattern">
        <complexType mixed="true">
          <choice maxOccurs = "unbounded">
              <element ref="upml:name"/>
              <element ref="upml:category"/>
              <element ref="upml:link"/>
          </choice>
          <attribute name="id" type="ID" use="optional"/>
          <attribute name="relation" use="required">
            <simpleType>
              <restriction base="string">
                  <enumeration value ="superordinate"/>
```

```
                    <enumeration value ="subordinate"/>
                    <enumeration value ="sibling"/>
                    <enumeration value ="competitor"/>
                </restriction>
              </simpleType>
            </attribute>
          </complexType>
      </element>

<!-- ..... -->
<!-- META-INFOMATION MODULE ..... -->
<!-- ..... -->

<!-- The alias Element .....-->
      <element name="alias">
          <complexType>
            <simpleContent>
                <extension base ="string">
                    <attribute name="id" type="ID" use="optional"/>
                </extension>
            </simpleContent>
          </complexType>
      </element>

<!-- The author Element .....-->
      <element name="author">
          <complexType>
              <attribute name="id" type="ID" use="optional"/>
          </complexType>
      </element>

<!-- The date Element .....-->
      <element name="date">
          <complexType>
            <simpleContent>
                <extension base ="string">
                    <attribute name="id" type="ID" use="optional"/>
                </extension>
            </simpleContent>
            <attribute name="event" use="optional">
              <simpleType>
                  <restriction base = "string">
                    <enumeration value = "creation"/>
                    <enumeration value = "publication"/>
                    <enumeration value = "revision"/>
              </simpleType>
```

```
            </attribute>
        </complexType>
    </element>

<!-- The identification Element ..... -->
    <element name="identification">
        <complexType>
          <sequence>
              <element ref="upml:name"/>
              <element ref="upml:alias"/>
              <element ref="upml:author"/>
              <element ref="upml:date" minOccurs="0" maxOccurs="unbounded"/>
              <element ref="upml:category"/>
              <element ref="upml:keyword"/>
              <element ref="upml:related-patterns"/>
          </sequence>
              <attribute name="id" type="ID" use="optional"/>
        </complexType>
    </element>

<!-- The keyword Element..... -->
    <element name="keyword">
        <complexType mixed = "true">
          <choice>
              <element ref="upml:term" minOccurs="1" maxOccurs="unbounded"/>
          </choice>
        </complexType>
    </element>

<!-- The metadata Element ..... -->
    <element name="metadata">
        <complexType>
          <sequence>
              <element ref="upml:name"/>
              <element ref="upml:alias"/>
              <element ref="upml:author"/>
              <element ref="upml:date" minOccurs="0" maxOccurs="unbounded"/>
              <element ref="upml:category"/>
              <element ref="upml:keyword"/>
          </sequence>
        </complexType>
    </element>

<!-- The name Element ..... -->
    <element name="name">
        <complexType>
```

```
            <simpleContent>
                <extension base ="string">
                    <attribute name="id" type="ID" use="optional"/>
                </extension>
            </simpleContent>
        </complexType>
    </element>

<!-- The term Element.....-->
    <element name="term">
        <complexType>
          <attribute name="term-type" use="required">
          <simpleType>
              <restriction base="string">
                  <enumeration value="abbreviation"/>
                  <enumeration value="concept"/>
                  <enumeration value="principle"/>
                  <enumeration value="technology"/>
                  <enumeration value="tool"/>
              </restriction>
          </simpleType>
          </attribute>
        </complexType>
    </element>

<!-- The title Element.....-->
    <element name="title" type = "string">
    </element>

<!-- ..... -->
<!-- PROBLEM MODULE ..... -->
<!-- ..... -->

<!-- The context Element .....-->
    <element name="context">
        <complexType>
          <sequence>
              <element ref="upml:user"/>
              <element ref="upml:task"/>
              <element ref="upml:platform-compatibility"/>
          </sequence>
          <choice>
              <element ref="upml:term" minOccurs="0" maxOccurs="unbounded"/>
          </choice>
              <attribute name="id" type="ID" use="optional"/>
        </complexType>
```

```
    </element>

<!-- The forces Element.....-->
    <element name="forces">
        <complexType mixed="true">
          <choice>
              <element ref="upml:term" minOccurs="0" maxOccurs="unbounded"/>
          </choice>
              <attribute name="id" type="ID" use="optional"/>
        </complexType>
    </element>

<!-- The platform-compatibility Element.....-->
    <element name="platform-compatibility">
        <complexType>
              <attribute name="id" type="ID" use="optional"/>
        </complexType>
    </element>

<!-- The problem Element .....-->
    <element name="problem">
        <complexType mixed="true">
          <choice>
              <element ref="upml:term" minOccurs="0" maxOccurs="unbounded"/>
              <element ref="upml:reference" minOccurs="0"
                        maxOccurs="unbounded"/>
          </choice>
              <attribute name="id" type="ID" use="optional"/>
        </complexType>
    </element>

<!-- The task Element .....-->
    <element name="task">
        <complexType>
              <attribute name="id" type="ID" use="optional"/>
        </complexType>
    </element>

<!-- The user Element .....-->
    <element name="user">
        <complexType>
              <attribute name="id" type="ID" use="optional"/>
        </complexType>
    </element>
```

```
<!-- ..... -->
<!-- SOLUTION MODULE ..... -->
<!-- ..... -->

<!-- The consequences Element.....-->
    <element name="consequences">
        <complexType mixed="true">
          <choice>
              <element ref="upml:term" minOccurs="0" maxOccurs=Unbounded"/>
              <element ref="upml:reference" minOccurs="0"
                        maxOccurs="unbounded"/>
          </choice>
          <attribute name="id" type="ID" use="optional"/>
          <attribute name="impact" use="required">
          <simpleType>
              <restriction base="boolean">
                  <enumeration value="positive"/>
                  <enumeration value="negative"/>
              </restriction>
           </simpleType>
          </attribute>
        </complexType>
    </element>

<!-- The example Element.....-->
    <element name="example">
        <complexType>
          <choice>
              <element ref="upml:term" minOccurs="0" maxOccurs="unbounded"/>
          </choice>
              <attribute name="id" type="ID" use="optional"/>
        </complexType>
    </element>

<!-- The implementation Element .....-->
    <element name=" implementation">
        <complexType>
          <sequence>
              <element ref="upml:structure"/>
              <element ref="upml:strategy"/>
          </sequence>
          <choice>
              <element ref="upml:term" minOccurs="0" maxOccurs="unbounded"/>
          </choice>
              <attribute name="id" type="ID" use="optional"/>
        </complexType>
```

```
    </element>

<!-- The rationale Element.....-->
    <element name="rationale">
        <complexType mixed="true">
          <choice>
              <element ref="upml:term" minOccurs="0" maxOccurs="unbounded"/>
          </choice>
              <attribute name="id" type="ID" use="optional"/>
        </complexType>
    </element>

<!-- The sample-code Element.....-->
    <element name="sample-code">
        <complexType>
          <choice>
              <element ref="upml:term" minOccurs="0" maxOccurs="unbounded"/>
          </choice>
              <attribute name="id" type="ID" use="optional"/>
        </complexType>
    </element>

<!-- The solution Element .....-->
    <element name="solution">
        <complexType mixed="true">
          <choice>
              <element ref="upml:term" minOccurs="0" maxOccurs="unbounded"/>
          </choice>
              <attribute name="id" type="ID" use="optional"/>
        </complexType>
    </element>

<!-- The strategy Element.....-->
    <element name="strategy">
        <complexType mixed="true">
          <sequence>
              <element ref="upml:example"/>
              <element ref="upml:sample-code"/>
          </sequence>
          <choice>
              <element ref="upml:term" minOccurs="0" maxOccurs="unbounded"/>
          </choice>
              <attribute name="id" type="ID" use="optional"/>
        </complexType>
    </element>
```

```
<!-- The structure Element.....-->
    <element name="structure">
        <complexType mixed="true">
          <choice>
              <element ref="upml:term" minOccurs="0" maxOccurs="unbounded"/>
          </choice>
              <attribute name="id" type="ID" use="optional"/>
        </complexType>
    </element>

<!-- ..... -->
<!-- STRUCTURE MODULE ..... -->
<!-- ..... -->

<!-- The body Element .....-->
    <element name="body">
        <complexType mixed = "true">
          <sequence>
              <element ref="upml:context"/>
              <element ref="upml:problem"/>
              <element ref="upml:forces" minOccurs="1" maxOccurs = "unbounded"/>
              <element ref="upml:solution"/>
              <element ref="upml:rationale"/>
              <element ref="upml:consequence" minOccurs="0"
                        maxOccurs = "unbounded"/>

          </sequence>
        </complexType>
    </element>

<!-- The head Element .....-->
    <element name="head">
        <complexType mixed = "true">
          <sequence>
              <element ref="upml:metadata"/>
          </sequence>
    </element>

<!-- The pattern Element .....-->
    <element name="pattern">
        <complexType>
          <sequence>
              <element ref="upml:head"/>
              <element ref="upml:body"/>
          </sequence>
              <attribute name="id" type="ID" use="optional"/>
```

```
            <attribute name="version" type="decimal"/>
        </complexType>
    </element>

<!-- The upml Element ..........-->
    <element name="upml">
        <complexType>
          <sequence>
              <element ref="upml:pattern" minOccurs="0" maxOccurs="unbounded"/>
          </sequence>
              <attribute name="id" type="ID" use="optional"/>
              <attribute name="version" type="decimal" fixed="1.0"/>
              <attribute name="xml:lang" use="required"/>
              <attribute name="xml:space" default="preserve"/>
        </complexType>
    </element>
</schema>
<!-- ..... -->
```