NOTE TO USERS

Studies on Pattern Dissemination and Reuse to Support Interaction Design

Ashraf Gaffar

A Thesis

In

The Department

Of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Doctor of Philosophy

Concordia University

Montreal, Quebec, Canada

September 2005

Your file Votre référence
ISBN: 0-494-09967-4
Our file Notre référence
ISBN: 0-494-09967-4

Abstract

STUDIES ON PATTERN DISSEMINATION AND REUSE
TO SUPPORT INTERACTION DESIGN

Ashraf Gaffar

The success of interactive software systems can be attributed to many technical and human factors working in harmony. Designing a new interactive system is a complex undertaking that must carefully consider this "harmony". Because this harmony is hard to predict before a system is actually put to work, extensive design experience and collaboration are crucial. For additional support, interaction design patterns have been proposed as a means to discover, encapsulate, and disseminate user interface design knowledge and best practices, hence improving the chances of success of new interactive systems.

Despite the obvious and acclaimed potential to support the design process, and the rich variety of pattern collections we have today, the reuse of HCI patterns has not achieved the acceptance and widespread applicability foreseen by pattern advocates. It has been recently identified in the research community that patterns are greatly underused by mainstream interface designers.

Within the scope of this thesis, we conducted an empirical study and a survey to gain better understanding of the problem of pattern underutilization. Accordingly, we point out and demonstrate the lack of suitable representation as a major cause. This thesis explores two different avenues in solving the problem:

On designer's side, we demonstrate the potential of patterns in enhancing user interface design in two investigations. (i) We explore the important but often neglected interaction between interfaces and the underlying system. We provide several examples and show how patterns can support this interaction for better interfaces design. (ii) We look at current approaches of user interface design processes and the commonly used models. Then we show potential improvements attainable through informed application of patterns.

In the second avenue, we conclude that a new pattern representation can help improve HCI pattern dissemination and reuse. We provide a model for the current pattern lifecycle and propose an additional layer to it, and a new pattern representation model. A dissemination method is provided to collect and organize all relevant activities and models within a comprehensive and structured approach. This addition is supplemented with the needed infrastructure in terms of supporting software as well as human activities.

Acknowledgements

My first acknowledgement should go to my supervisor Dr. Seffah for his support. I would also like to acknowledge the experience of Dr. T. Radhakrishnan which was evident in his assistance and feedback in many parts of my thesis.

Dr. Greg Butler was a wealth of knowledge. Besides the high scientific and moral standards that I learned from him -as a dedicated and excellent supervisor of my Masters degree- I appreciate his continued help and advice during my PhD. He raised the bar of academic values for me to a really high level. The bar was raised even higher every time I interacted with Dr. Peter Grogono for the last six years. His dedication to science and to helping students is unprecedented; not just towards his students, but to any one who wanted to learn.

I was able to improve the quality of my research because I met and worked with other people who really wanted to help, cooperate, and give. Dr. Peter Forbrig at Rostock University in Germany was an excellent example. I would like to thank him for his great and versatile support. I also leaned from Dr. Van der Poll (john) from the University of South Africa. It was a pleasure working with him. The feedback I received from Dr. Jean Vanderdonckt clarified several issues in my thesis. I admire his knowledge and energy.

I would like to thank Dr. Jean-Mark Robert and Dr. Sofiene Tahar for their useful and constructive feedback which was instrumental to my work.

My research at Concordia was only made possible through several scholarships and grants. I would like to thank the Government of Quebec for offering me the FQRNT scholarship. I also appreciate the Federal Government of Canada for financially supporting some of my work through several NSERC grants. The scholarship of Concordia University Research Chair as well as the Concordia

Fellowship for Academic Excellence were two major supports to my PhD. I am also thankful to Daimler Chrysler. Their Web Engineering project was a great experience for me.

To my parents, your unconditional love and support have inspired me through the toughest times of my life.

To my wife, thank you for taking over when I was too busy to be there for our family, and I know I was busy all the time.

To my lovely daughters, Rouan, Judy and Rana, your smiles and hugs rejuvenated my soul and recharged my batteries everyday. I appreciate your understanding that daddy's "endless" homework was important for all of us. I love you.

To my Father

I still can't find the words...

God bless you

Table of Contents

List of Figures

List of Tables

List of Abbreviations

GPM: Generic Pattern Model, see sec. 5.7.1

GPT: Generic Pattern Type, see sec. 5.7.2

GUI: Graphical User Interface

HCI: Human Computer Interaction

IPE: Integrated Pattern Environment, a unifying concept that offers interoperable models, a notation to implement these models using XML or an object oriented language, and an underlying multi-tier information system to store the essential data and to offer interaction with it.

Minimal triangle: The *problem*, *context* and *solution* predominantly used in patterns.

MVC: Model View Controller. An architectural model that separates application's data, processing and display into a model and several views and controllers.

PAC: Presentation, Abstraction, Control. An object oriented model composed of a hierarchy of cooperating PAC agents used in implementing user interfaces.

Smart Data: Smart data is data that is capable of describing its contents independent of any application. Smart data is often associated with and described in XML to offer a language independent, application independent representation of data. It can be accessed and parsed by different XML applications to retrieve its contents. The non-smart data often refers to object dumps as binary files. They can only be used by their own applications and their

data contents can only be processed through the application that created them or through an appropriate application-specific transformation.

Smart Patterns: Smart patterns are pattern objects that are capable of communicating with other object and tools in a runtime environment. They offer different Application Programmer Interfaces (API's) to allow interaction with them.

UI: User Interface

UIML: User Interface Markup Language, an XML-compliant markup language. It focuses on describing the user interfaces in an abstract way, independent of any platform or programming languages.

USIXML: **US**er **I**nterface e**X**tensible **M**arkup **L**anguage. It is an XML-compliant markup language. It focuses on describing the user interface in multiple contexts of use like command line interface, graphical user interface.

XML: **E**xtensible **M**arkup **L**anguage. It is a simple, flexible ASCII-based Markup language. It is used to exchange wide variety of data between applications and is extensible by allowing the creation of arbitrary XML-compliant sublanguages for different purposes.

XUL: **X**ML **U**ser Interface **L**anguage. XUL is an XML-based User interface definition Language. It allows building feature-rich cross platform applications' interfaces that can run connected or disconnected from the Internet.

Chapter 1

Introduction

Having a good language of patterns at your disposal
is like having an extended team of experts
sitting at your side during development

Grady Booch

Thesis Scope

An Interaction design pattern[1] is generally defined as a solution to a usability problem which occurs in different contexts of use. The concept of patterns was inspired by the work of Christopher Alexander In towns and building architecture [Alx79]. Interaction design patterns are a means to transfer experience from Human Computer Interaction (HCI) experts to software developers [WVE00]. Several languages have been used to represent patterns including natural languages, programming languages, design languages (like UML), markup languages (like HTML and XML), and simple sketches (we can see them as visualization languages). These notations have some benefits and drawbacks. This thesis looks into the way we represent interaction design patterns. Our goal is to **evaluate** and **facilitate** their discovery and delivery.

---

[1] Within this thesis, we will interchangeably use the term interaction design pattern with HCI pattern as well as UI design pattern, design pattern, pattern, user experience pattern, usability pattern to refer to the large variety of patterns used to make any type interactive system usable. This includes Web, GUIs, mobile and other highly interactive applications.

HCI pattern writers, who are usability experts with background in psychology, focus on usability and human aspects of the user interface design. They generally prefer to use narrative formats to convey solutions to common user problems with supporting theories and concepts of interaction design and human factors. On the other hand, user interface designers are typically software developers who need concise and pragmatic guidance through their design and coding activities. They often find it hard to translate text pattern knowledge into concrete design [LNHL00], [GSP05]. Moreover, with the plethora of patterns available today, mainstream developers get inundated with huge number of pattern literature and links in many books and on the Internet. They have to manually read and sift through piles of texts looking for some useful patterns to apply [Gaf05b] and [Gaf05c]. Once found, narrative descriptions make it prohibitively difficult to automate the interoperability between patterns and design tools, an indispensable help to developers today [CHV00].

In this regard, besides focusing on **what** should be presented in terms of information contents within patterns, a fundamental challenge is **how** it should be packaged and offered to developers in an appropriate way to help understand and apply them correctly and efficiently. As will be discussed in this thesis, more flexible notations and views are required to represent different types of pattern information. The pattern lifecycle needs to be investigated, defined and modeled to allow for better understanding of pattern dynamics as an emerging design technique. In what follows we will explore the multi-faceted problem of pattern representations.

The Essence of Using Patterns

Alexander, Ishikawa, and Silverstein [AIS77] promoted the idea that one could achieve excellence in architecture and building construction by learning and using a carefully-defined set of design rules in form of patterns. Although the quality of a well-designed building is hard to put into words, the patterns

themselves that make up that building are remarkably simple and easy to understand [Tid97]. The software engineering community has popularized patterns throughout the entire software design spectrum from requirements to testing, deployment and reengineering.

In the field of Human Computer Interaction, patterns have been proposed as a step towards building more usable user interfaces. The idea is to capture information about frequently encountered- and hard to solve problems and explain how they can be solved in an optimal way. By recording the indications and remedies of typical problems, patterns can improve communication among developers and support design reuse. When several patterns are collected together, they form a "language" that provides a process for the orderly resolution of software development problems. Pattern authors have introduced pattern languages using different approaches and terminology. They are not formal languages, but rather a collection of interrelated patterns, though they do provide a vocabulary for talking about a particular problem.

A View on Pattern Benefits

Out of a large literature about pattern benefits, we will briefly explore one as cited by Grady Booch in [AMC03].

> "*...at its most mature state, a pattern is full of things that work, absent of things that don't work, and revealing of the wisdom and rationale of its designers*".

Being carefully stated to summarize several aspects of patterns, this statement is acceptable on its own merits. The statement focuses on the benefits that stem from the basic nature of pattern concept. We briefly provide some insight into this statement.

"**...most mature state**": from that we can understand that patterns are mature artifacts; or mature solutions. The maturity process makes the difference between a solution and a pattern, and again makes for the goodness of patterns as "tried, tested, and true". In the current shift towards mobile and pervasive computing, universal usability, and context-aware interfaces [WP05], [FFS+04], [VCBT04], pattern oriented design processes as well as the pattern lifecycle should be investigated, understood and promoted within a pattern community to look for and validate pattern discovery and reuse. Mature patterns can provide robust and more predictable solutions in new paradigms than new, untested solutions.

"...full of things that work" indicates that a pattern is more than a simple solution. In today's sophisticated and highly interactive software, it is not effective to just build the system as a collection of simple entities and assemble them according to one's own knowledge. We once learned machine language instruction sets as all what was need to solve a problem and we wrote programs that worked well. Since then we continuously upgraded our language approach towards higher level languages and more abstractions. That definitely increased our ability to address and solve more complex problems with relatively less effort. As the complexity increased, we tended to reuse existing composite structures instead of building new ones every time, and we are constantly looking for better aggregation and abstraction techniques, both in languages as well as in running software. Patterns enhance this trend -in a context sensitive paradigm- as a collection of usable solutions that happen to appear repeatedly together in successful applications. Being known -or proven- to work better than other combinations, we need to discover and collect them, understand why they work well as a group, and in which context, then put them in a suitable format that insure their effective reuse.

"Absent of things that don't work" reveals another advantage of patterns, namely the reduction of failure by warning us of concealed traps. It is a human

nature that we prefer to focus on things that we have seen before and ignore things that are unfamiliar to us. In other words, the ability to analytically observe things is biased towards their familiarity to the observer, and not towards their actual importance [Coc01]. Consequently, we could underestimate things that are important just because we don't understand them or because we never saw them before or, even worse, because they are accompanied by other things that are more familiar to us. That might explain why experts do better jobs in recognizing more important things faster.

In the huge stream of knowledge that is passing by software designers, they tend to catch only those things that they know, and anchor them to their memory, and to their design. This can have negative effect as designers may see the system from a programmer's point of view or point of understanding and not from the user's. That said, an interface designer might be tempted to think that a certain combination of objects will definitely work in the envisioned interface, and indeed they will work, only not for the end user's satisfaction, but rather for the designer's.

Realizing this fact has promoted the rise of usability patterns, user centered design, and quality-in-use concepts. While it is hard to explain why certain things that had great design were less successful than anticipated, they help guide interaction designers into seeing things from the user's perspective and hence avoid things that "don't work"

"...revealing the wisdom and rationale of its designers" indicates an important facet of patterns. Many of us have had the opportunity to work with mentors that left great impression on us about their professional "wisdom". We then learned how to follow the same approach or a similar one in other situations. While often implicit and hard to quantify, we use our intelligence and reasoning to acquire, adapt and reuse this wisdom to new situations, or contexts. Besides giving a solution to a problem, patterns often provide other information that

reveals the wisdom of the designer or of the solution. The other end of the spectrum would be the famous expression of "reinventing the wheel".

Why the Benefits Were not Reached?

As a promising approach in interactive software design, the benefits of HCI patterns have been largely discussed and the applicability of patterns has been promoted and cost-justified [AMBC98], [GL99], [MMP02]. As a result, many patterns were provided to interface designers. However, very little work has been done to evaluate the acceptance and the success of these patterns as originally predicted. Recently, researchers started to observe that the expected benefits have become conditional or unattainable [CHV00], [LNHL00]. We believe that additional work is needed in order to represent patterns in a more suitable way to achieve these benefits. More analysis of the problem is provided later in the chapter.

The Need for Modeling Pattern Lifecycle and Representation

The best patterns are not so much invented
as they are discovered and then harvested
from existing, successful systems.

Grady Booch

From the above statement, we can look at patterns in general as artifacts that have three main milestones, organized from a user perspective (Figure 1).

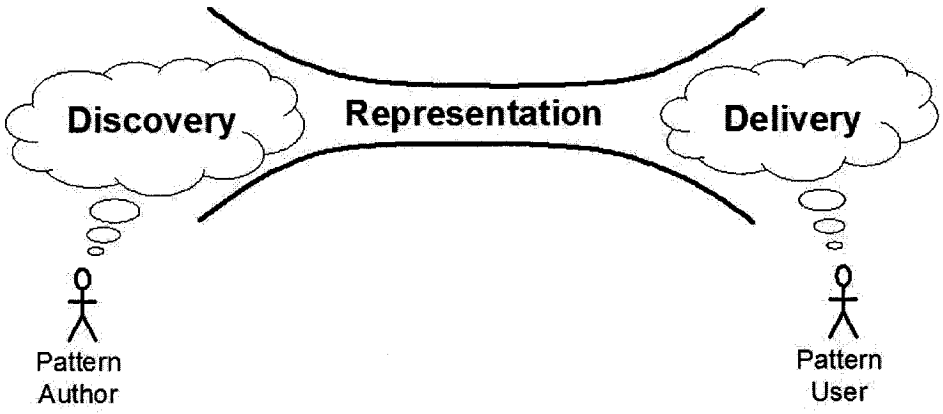


Figure 1: Major milestones and users of patterns

Pattern Delivery

On the pattern user side, we can say that patterns are harvested and represented with the main goal of being delivered to other users who implement them as solutions. A delivery paradigm is essential in the pattern approach because it indicates that patterns arrived effectively to potential users; a knowledge dissemination view. This means that patterns should be represented in a way that software developers can learn, master and apply easily and effectively in their context. This implementation highlights the main role of patterns, promoting effective reuse. If patterns were harvested and written down just for the sake of archiving them then we have missed on the great benefits of patterns.

Pattern Discovery

On the pattern writer side, the discovery of a pattern is only the beginning. Harvesting is a carefully selected metaphor that indicates the hard work associated with patterns. By observing existing artifacts and problems that have been solved successfully, we can detect a repeated structure or behavior that is worth recording. By asserting its importance, we can write down the essential components and –if possible- analyze them. An expert can provide insight as to why this combination is good or why it works well and in what context. Finally

guidance of how to reuse this solution can be added to assist in modifying and reapplying the solution. In Chapter 3, we will provide some work in this area.

Pattern Representation

Representation of patterns can be seen as a vehicle –a medium or an infrastructure –to bridge the gap between the two main activities; delivery and discovery. This representation is essentially about how to format the solution in a way that allows it to mature from its solution-format into a pattern. In essence, a pattern is a solution alongside other information that supports it. The reason is that in order for a solution to be used by others, they have to be convinced that this is a good solution. Part of this come by annotating pattern solution with expert analysis and comments, listing of some cases where the solution has been applied and the "success indicators", and possibly some code examples. Bearing in mind that no two systems are exactly the same, and that every new software is a new adventure, patterns are typically annotated with important guidance on how to apply them in different contexts and situations. Some details are left out to allow the end user to rematerialize an abstract pattern back into a concrete solution that is adapted to the new design. Having decided on what to write, the sibling question would be how to best represent this information: through UML diagrams, simple diagrams, images, text, source code, or a combination of all of them.

The success of pattern approach depends on all those three milestones. As we discuss the potential benefits of applying patterns in design reuse, we can not claim that patterns are "silver bullets". Due to the inherently creative nature of design activities, the direct reusability of designs represents only a small portion of the total effort [YA99]. It requires a considerable amount of experience and work to modify existing designs for reuse. Many design ideas can only be reused when abstracted and encapsulated in suitable formats. Despite the creative nature of their work, software designers still need to follow some structured process to help control their design activities and keep them within the available

resources. Partial automation of this process, combined with sound experience and good common sense can significantly facilitate the analysis and design phase of software development [Art00]. Within this process, tools can help glue patterns together at higher design levels the same way we do with code idioms and programming language structures [CHV00]. For example, the Smalltalk Refactoring Browser, a tool for working with patterns at the source code level, assists developers using patterns in three ways [FMW97]:

- Generate program elements (e.g. classes, hierarchies) for new instances of a pattern, taken from an extensible collection of "template" patterns.
- Integrate pattern occurrences with the rest of the program by binding program elements to a role in a pattern (e.g. indicating that an existing class plays a particular role in a pattern instance)
- Check whether occurrences of patterns still meet the invariants governing the patterns and repairing the program in case of problems

Research Statement

As highlighted in the previous sections, patterns are ideally meant to help in developing concrete design artifacts. Developers can reuse them in building more usable interactive systems. However, the lack of tool support coupled with unclear description of the process of collecting, documenting, disseminating and implementing them in new designs compromises these benefits [BFVY96]. The text-based formats currently used for documenting patterns cause them to have problems similar to those of design guidelines – they become ambiguous, abstract, and therefore difficult to fully understand and master [WVE00].

We can see that the lack of communication and coordination between pattern authors and users is one of the main obstacles to effective adoption of pattern-oriented designs. Different interviews, debates and studies [CHV00], [Gaf04], [GWC+00] and [LNHL00] revealed that designers found patterns highly beneficial

when presented to them. Yet many designers experienced difficulties when attempting to find, understand and apply patterns correctly in their design context. They also found it more challenging when they were asked to implement same patterns for different platforms. Once they were trained, translating a pattern into a program block was a straightforward task.

The lack of communication is apparent in other issues associated with patterns. Logistical concerns are often tied to the storage, publishing and delivery of patterns to their users. Not all experts with genuine ideas can –or care to- afford publishing their patterns in books or keep them in an updated Website.

Our work proposes a new approach to document and represent patterns. We do not suggest a fixed template to be enforced on pattern authors, but rather an abstract "generic model" that covers a wide range of formats and promotes interoperability between them. The generic model allows for instantiating concrete pattern objects, or "complex types" in the object oriented paradigm terminology. This idea allows pattern authors to use their creativity and their own formats in writing patterns and then adds an additional modeling layer to transform them into software components with the same semantics, and without losing any knowledge contained in them. Once transformed, patterns become software objects in any runtime environment and can be manipulated by developers using tools within a software development environment. The concept equally encourages new patterns to be written originally as objects instead of text artifacts.

Research Methodology

This objective has been realized in 4 main undertakings:

1- Identify, via a comprehensive literature review, the state-of-the-art of research on HCI patterns, languages and catalogs as well as the

prevailing myths and misconceptions in terms of pattern discovery, representation and delivery. In Chapter 2, we will discus the results of this literature review from an analytical perspective while paving the road for an informed approach to pattern representation.

2- To complement this analytical review, we conducted an empirical study in a usability lab setting (Concordia Usability and Empirical Studies Lab) to assess the pattern approach from users' perspective. A large scale online questionnaire-based survey was also conducted to investigate how pattern are currently perceived and used in the industry.

3- Explore, via a major experiment, how patterns should be represented to enhance their role and increase their reusability within architecture driven development. In particular, we study the intricate relationship between user interfaces and the underlying system. Despite the prevailing trend to completely decouple them using popular architectures like "Model View Controller" or the "observer pattern", we shed new light on "invisible" associations that need to remain coupled for better interface designs. An interface that offers "Undo" functionality must be connected to a system that can safely support many "frantic" Undo/Redo combinations without crashing or corrupting user's data. If not, it might be more prudent to not offer the Undo in the interface or to constrain its use. Patterns are proposed to identify and improve this association. We identify and enhance the role of these patterns within a usability context to abate some negative consequences and improve the overall software architecture. These aspects are detailed in Chapter 3.

4- With a focus on the interface, we present another major experiment to investigate the role of modeling and the emerging trends in model-based approaches in interface design. In this regard, we identify some existing shortcomings in current model-based interface design techniques and look

into reconciling patterns and models to improve them. These aspects are detailed in Chapter 4.

5- Based on the 1, 2, 3 and 4 above, we have been able to propose a comprehensive model for the existing pattern lifecycle and define the main activities associated with it. We propose a new methodology, called *the Seven C's* to govern the essential steps and roles for effective pattern reuse. We also extend the representation of patterns to be accessed and manipulated within different programming paradigms including object oriented languages, XML and relational data models.

Thesis Organization

This thesis is organized in 6 chapters.

In this introductory **chapter**, we motivate our research by highlighting the current problem in HCI pattern representation.

In **Chapter 2** we investigate the concept of HCI patterns in some details and discuss common facts and fallacies about them.

In **Chapter 3** we show how patterns can help improve the usability of interactive systems and demonstrate the significant but invisible association between user interfaces and the underlying system.

Chapter 4 shows the strong connection between user interface design and models, and the beneficiary interaction between them. In it we investigate the prevalent modeling approaches and how we can support them with patterns.

Chapter 5 culminates the work by providing an integrated pattern environment to support pattern dissemination and reuse. The system offers patterns in multiple

formats that promote interoperability with other software users and tools in different programming environments.

Chapter 6 provides the conclusion and an insight into related future work.

Chapter 2

A Literature-Based Analysis of HCI Patterns Dissemination and Reuse

Abstract: In this chapter, we look at how patterns are currently used in the HCI community, who is proposing them, what types of HCI patterns are being proposed and how they are used. Our investigations and analysis have been proposed in several HCI-related workshops and conferences and the feedback from these activities has been guiding us through our work. Since the Internet is used as one of the main media of disseminating HCI patterns, we also include an analysis of several studies of the Internet, and evaluate its role and its effectiveness in the dissemination process.

2.1 Introduction

You ought to be able to show that you can do it with the regular tools before you have a license to bring in your own improvements

Ernest Hemingway

Designing a usable and easy-to-maintain user interface is a challenging task. Indeed, user interfaces are engineered in a wide variety of ways, based on hundreds of design heuristics and thousands of design "tips". Only few of them are drawn from empirical evidence. In general, there is no feedback on the quality or usability of the interface design until relatively late in the design cycle when usability testing is performed. Not only does it come late in the design

process, but also it is expensive and time-consuming. Unlike code fragments or a complete source code of an application, Interface testing is hard to perform in a formal way. It has to deal with many new users and collect their feelings and opinions about interacting with the application. Due to the expense and difficulty of setting up tests with humans and the frequent lack of interest or exactness by some participants, it is not always possible to truly evaluate or improve the usability of a design. Results can be coarse-grained and mainly qualitative, often lacking sufficient insights into specific cause or solution of a problem.

Besides testing, there have been many partially successful approaches to collect, represent and deliver best design practices. The most popular ones are [Cas97]:

- Study of exemplars
- Practice under the instruction of a mentor
- Design principles to capture the mentor's implicit knowledge
- Design rationale for organizing application of principles to cases
- Design guidelines and style guides making principles specific
- UI toolkits embodying some guidelines

The common difficulties associated with testing, and the attempts to overcome them have helped in the rise of interaction design patterns as a cost-effective vehicle for capturing and disseminating best practices in interface design. They have the advantage of giving designers immediate feedback on their design in progress and the ability to select and combine different successful design alternatives. Consequently, many HCI experts devoted themselves to developing HCI pattern languages, encapsulating their knowledge in a reusable format for designers. Among the heterogeneous collections of pattern, "Common Ground" [Tid97], "Amsterdam [Wei98], "Experience" [CL98] and "The Design of Sites" [DLH03] play a major role and wield significant influence. It has often been reported that all HCI patterns and pattern languages are useful design tools [Erc00], [GL99], [GSSF04].

In this chapter, we first consider the facts about patterns, starting with the history and then moving on to the definition of patterns and anti-patterns. There are different motivations for using patterns and correspondingly different opinions about them. We discuss them as well together with the recurring questions and debates about how many times a solution to a design problem should be used before it can be called a pattern. also take a deeper look into the existing patterns, reflecting on their internal structure, inter-collection structures and intra-collection structures. HCI design pattern languages are then discussed at a conceptual level. We consider the semantic model of patterns as seen by a pattern author versus the mental model as perceived by a pattern user. We explore how this can be exploited to promote better communication between the two parties, hence improving the reuse of patterns. We finally highlight some pitfalls in using patterns based on assumptions and fallacies in discovering, composing, implementing, and validating patterns.

2.2 The History of Patterns

Since their first inception, patterns have been accompanied by certain expectations that rose from logical thinking of researchers. Many of these expectations have proven to be true. In this section, we will evaluate them and discuss how well-known patterns originated as well as how they are currently perceived and used.

The first major milestone is often attributed to the architect Christopher Alexander, in the 1970s. His two books, *A Pattern Language* [AIS77] and "*A Timeless Way of Building*" [Alx79] contained collections of pattern examples, all following a similar structure. Alexander had an earlier book "Notes on the Synthesis of Form" [Alx70] which paved the road for his other books and provided some analytical insight into pattern-related philosophy. The pattern concept was not well known in software community until 1987 when patterns

appeared at OOPSLA, the object orientation conference in Orlando. There Kent Beck and Ward Cunningham introduced pattern languages for object-oriented program construction in a seminal paper [BC87]. Many papers and presentations have then appeared, authored by renowned software design practitioners such as Grady Booch, Richard Helm, Erich Gamma, and Kent Beck. In 1993, the formation of "Hillside Group" [Hil93] by Beck, Cunningham, Coplien, Booch, Johnson and others was a step forward to forming a design patterns community in the field of software engineering. In 1995, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (the Gang-of-Four; GoF) published "*Design Patterns: Elements of Reusable Object-Oriented Software* [GHJV95] which gave patterns a new object-oriented dimension. This book is widely accepted and used in the software developers' community.

Following the great success of design patterns by GoF, patterns were further proposed to HCI practitioners at a "CHI 1997" Workshop by Erickson and Thomas [ET97]. The HCI community immediately adopted the concept of patterns for UI design. Several individuals and groups contributed swiftly to the development of HCI patterns for interaction design like [Tid97], [Cas97], [ET98] [CL98], [PG98], [GL99], [Bor00], [Wel00] and many others. Every year, new pattern collections are added through publications in conferences, workshops and books.

2.3 Abstract Pattern View

The concept of design *patterns* can have different meanings in different domains. This can lead to confusion when these domains intersect. An example is in the two domains of artificial intelligence (pattern recognition) and interaction design (HCI design patterns). While the concept of patterns and the activities associated with them look different in both domains, they have –in fact- strong similarities that can be used in a constructive way.

A mathematics forum at Drexel University [Dre02] defines patterns as *"A unit, repetition, and a system of organization".* This is an abstract definition that broadly applies to different domains and uses in both art and science. In artificial intelligence, for example, the units of patterns exist in the environment, and through their repetition (temporal, spatial, or otherwise), they are observed and recognized. The system of organization is the environment in which these patterns occur. Similarly, in interaction design, the unit could be a proposed solution to a problem. The repetition is the fact that the proposed solution is good enough that it has been successfully used in existing applications and is therefore recommended for reuse in other designs[2]. The system of organization is the design artifact in which the pattern is applied.

Similarly, in textile industries, patterns have a parallel analogy. The unit is the graphical shape printed on the fabric. The repetition is the fact that these shapes are printed repeatedly on the fabric, which also serves as the system for organizing these patterns together.

Our field analysis revealed an important observation . A concrete design goal has to be explicitly specified to the textile pattern designer. Namely, the designer has to devise patterns according to one of three (apparently contradicting) criteria:

- Pattern shape and size should work out to hide the repetition completely
- Patterns should be placed together to emphasize the repetition
- Patterns should visually interact to give the illusion of other virtual patterns

For example, to emphasize the third goal, we show the pattern in Figure 2.

---

2 This statement raises two important issues: The first one is about the famous debate of how many times a solution should have been used before calling it a pattern. We will come back to this point later. The second issue is using the word pattern "reuse", and not pattern "use", due to the fact that –by definition- patterns have been used as solutions and only as they are reused, they become patterns.

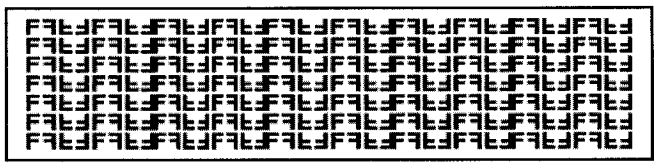


Figure 2: The visual effects and illusion of patterns

The unit here is the letter "F", and it is organized to give the illusion of several virtual patterns like "columns of 7 letters", "double columns of mirrored letters", "inverted double columns", "squares of 4 double columns touching each other", and even the illusion of some " corrugated 3D wall effect".

At a practical level, repetition is emphasized as one of pattern fundamentals. [AIS97] defined a pattern as: "*A recurring solution*". Therefore the recurrent use or reuse-ability is an important factor to validate patterns.

Alexander defines a pattern as a three-part rule which expresses a relationship between a certain context, a problem and a solution [Als79]. The *context* is the environment, situation, or interrelated conditions within the scope of which something exists. The *problem* is an issue that needs to be investigated and resolved, and is typically constrained by the context in which it occurs. The *solution* is a response to the problem that resolves the issue in a specific context. In general, patterns also provide the rationale behind the proposed solution, as well as the consequences that might be encountered if the solution was applied. Patterns are essentially context dependent. If the context is changed, the solution could be different for the same problem, which may lead to a different pattern being applied. This is one of the fundamental pillars of patterns. Without a context, patterns loose their applicability and the relevance of much of their knowledge and analysis. It is the context that makes rise to the particularity of patterns as packaged "ready to go solutions with insight". A context-free pattern would be too abstract to apply or even understand.

Nonetheless, a certain degree of abstraction –or invariance- is still needed to prepare patterns for reuse in different designs. In software engineering, the GoF [GHJV95] stated that a *problem* has a set of goals and constraints collectively referred to as *forces*, which occur in a specified context. The *context*, in this case, is the recurring set of situations within which the pattern applies. The pattern is *a generalised solution that can be applied to resolve the forces.*

Some defining characteristics and basic terminologies that are relevant to patterns are commonly understood and used among pattern authors. They include: **identification** of the problem in context with imposed constraints, **existence** of the solution, **recurrence** of the problem, **invariance** abstraction of aspects of the solution, **practicality** of the solution, which needs to strike a balance between **optimality** and **objectivity**, and **communicability** of the problem and the **process** of conveying the solution to the user. The relationship between some of these characteristics is illustrated in Figure 3.

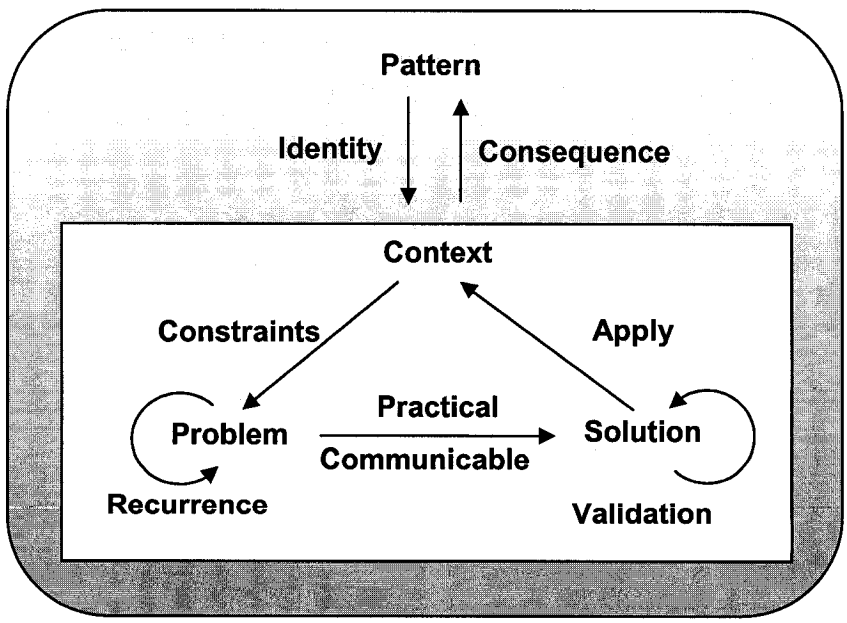


Figure 3: Pattern's anatomy and cycles

2.4 Anti-patterns

Over the last few years, the idea of anti-patterns has started to gain momentum in software pattern research [MJ99] [WV03]. Anti-patterns document what usually goes wrong in software development, and how one can avoid these mishaps [CBE+04]. The rationale in publishing anti-patterns is to identify recurring design flaws for the purpose of preventing other people from making the same mistakes [BMMM98].

An anti-pattern has two possible forms: it either "*provides knowledge on how to go from a problem to a bad solution*" or shows "*how to go from a bad solution to a good solution*" [CBE+04]. The former will be referred to as a *warning* anti-pattern, and can be seen as a "warning before falling into a trap". The latter is commonly called an *amelioration* anti-pattern, and is seen as "recovery from- or surviving a trap". As in patterns, an anti-pattern has more information and analysis than just a problem and a solution statement. If described properly, an anti-pattern tells the designer why the recurring bad solution looks attractive, why it turns out to be bad and what positive patterns are applicable in its stead. Anti-patterns therefore concentrate on presenting negative solutions [BMMM98]. Warning anti-patterns are not very useful to the designer, behaving as mere examples of what could go wrong. They might be browsed quickly before or during design, but they rarely take major role in design, or motivate designers to examine them carefully. On the other hand, amelioration patterns are constructive and useful to designer since they show how a bad solution can be refactored. They often take a major role, especially in reengineering projects where problems are already present, showing in test results or even crippling a working system.

The fact that warning patterns are less useful compared to amelioration patterns may look counter-intuitive. By default, it makes more sense to avoid a trap than

to first fall into it and then try to find a safe exit or a recovery strategy. However, several factors might contribute to the opposite:

- **Lack of motivation**: It is a human nature that we are not always motivated to heed warnings until they happen to us, or at least until we see them happening close to us.
- **Overconfidence**: designers tend to like or believe in their own point of view of an issue more than that of an anti-pattern author.
- **Underestimation**: As discussed earlier, in the huge stream of knowledge that is passing by us, we tend to only catch those things that we know. Designers could underestimate potentially serious problems just because they don't understand them. Therefore, an eminent trap might be simply ignored.
- **Perceived overhead**: Design process is often restricted by limited resources. Tight schedules and limited budgets might force designers to go ahead with their design with minimal preparations. They are often tempted to believe they have little time to spend looking at anti-patterns and carefully studying them.

Empirical studies are needed to determine these factors and their effect.

2.4.1 Anti-patterns in the Design Process

Due to the fact that applying patterns and anti-patterns have added complexity to any projects, some refactoring proponents call for having a no-pattern design first. After having an application that is up and running, patterns can be added at a subsequent refinement stage. This approach definitely has its merits in simplifying the design process. By not focusing on patterns early on, this idea lends itself naturally to the concept of iterative design and early working versions, as promoted in XP and agile programming methodologies. This might be a controversial issue with positive and negative sides depending on the nature of

each project. For example, amelioration anti-patterns are the foundation of many reengineering projects. The use of anti-patterns and their role in the design process is definitely an interesting issue that warrants more research

2.4.2 Patterns are not Anti-anti-patterns

In the theory of pattern languages, now –ironically- developed more extensively in software than in building architecture, the concept of 'anti-pattern' plays a central role. Unfortunately, the solution space is not one-dimensional; a pattern may have a number of associated anti-patterns, and a pattern used in a different context may become an anti-pattern. Furthermore, the negative of an anti-pattern is not necessarily a pattern – it could merely be another anti-pattern. At this stage research into software design anti-patterns abounds [CBE+04], but none of the renowned HCI pattern collections investigated make any reference to anti-patterns yet.

2.5 The Perception of HCI Design Languages

A tomato does not communicate with a tomato,
we believe.
We could be wrong.

Gustav Eckstein

There is abundant literature that discusses the issue of languages from different points of view, and there are several ways of grouping and categorizing them. Broadly speaking, we can see a language as "a means of communicating knowledge between two physical or logical parties".

[RE96] state that design languages have been used to design all kinds of objects and services and that these languages are "*often used unconsciously, arising out of the natural activity of creation and interaction with the created things*". They emphasize that when consciously understood, developed, and applied, design languages can build on and improve this natural activity, and "*can result in dramatically better interactions, environments, and things of all kinds*". A successful design language is commonly used by many designers (we can refer to UML as a good example). However, as pattern authors have developed different "pattern languages", the mere facts of multitude of these languages and their incompatibilities seem to make them ineffective. Users have to define their own "activities" of interacting with patterns within each pattern language, and have to figure out how to select and apply them, which make pattern languages hard to learn. Ideally, when the activities of pattern reuse are explicitly defined and understood, design languages can build on and improve this natural activity, and can result in dramatically better interactions.

A pattern language is more than just a collection of patterns. A catalogue or collection of items can be viewed as a dictionary, which offers a way to access items individually, looking for their internal contents or characteristics. Apart from providing an organized access method to individual items, dictionaries seldom provide information on how items can be interrelated or used together.

In contrast, a significant part of pattern knowledge lies in how they can be combined together in a dynamic network of pattern associations and consequences of their interactions [GSP05], [GMS05]. A successful pattern language contains information regarding when and how patterns can be combined in order to form larger-granularity pattern blocks. Ideally, this activity can be iteratively applied to produce concrete design artifacts. This concept is often dubbed as pattern-oriented design (POD) approach. The POD in itself is not a comprehensive design process that relies fully on patterns and their interactions. It would be unrealistic to generate a design synthesized solely from

patterns. On the other hand, POD is not a simplistic activity of using few patterns to manually modify the layout of a web page or an interface. A realistic POD would fit somewhere between these two extremes. The POD approach is usually integrated into another design process –a host process- where POD activities are inserted within specific phases of the host process [GSSF04], [GSP05].

In the context of accumulating experience, Nonaka [Non98] distinguishes between *tacit* and *explicit knowledge* and suggests four modes of knowledge transfer:

1. **Internalization**: Creating tacit knowledge by acting upon explicit knowledge
2. **Combination**: Creating new explicit knowledge by linking existing explicit knowledge
3. **Socialization**: Sharing tacit knowledge with others through implicit interaction
4. **Externalization**: Sharing tacit knowledge with others through explicit verbalization

In HCI, the concept of a pattern language is often considered in an analogous approach to three of Nonaka's four modes of transfer. Creating a new pattern language strongly resembles the first two modes, internalization and combination. The former corresponds to observing current solutions, collecting and analyzing them, and the latter corresponds to building a new pattern language by combining patterns into a collection.

When it comes to pattern reuse, Tidwell [Tid97] sees pattern languages as a *"way of communicating design knowledge and idioms between design team members"*, which is a form of externalization. Gaffar [GSSF04] explains that in an HCI pattern language, communicating patterns is done using a natural language: something like "Are we going to use **menu bar** or **sitemap** within the **homepage**?" where the three boldface words are actual HCI patterns. Duyne [DLH03] explains "*In fact, though you may not know it, you may already be using*

some form of pattern languages to articulate and communicate your design". The presence of externalization is evident since patterns are mainly used for sharing some knowledge by explicitly writing it in a pattern format. We agree that without pattern names, communication between designers would be more difficult and implicit; resembling a socialization knowledge transfer. But we also believe that there is more to a pattern language than just that.

2.5.1 Executable Design Languages

As mentioned earlier, there are plenty of pattern languages, and the number is growing. Apparently, there is an evident need to transfer knowledge, and this need is likely to remain in the foreseeable future. Nonetheless, looking into the quality of pattern languages as a tool for knowledge transfer is as important as looking into their quantity.

Traetteberg [Tra02] mentions that the role of representing knowledge is both to "capture knowledge", and to "support the design process[3]". Many of the works on pattern languages focus on the first point alone. We can see the challenge in crossing the existing gap from the first point to the second. The large number of patterns "captured" is not scalable when it comes to "supporting" the design process. Currently, the common methods of implementing Traetteberg's two points are through human activities. People write patterns in a natural language for other people to read and understand, and then manually produce new design artifact. We can see the scalability problem of this approach by considering the thousands of patterns offered today.

One categorization of languages in the context of communicating parties can be as shown in Figure 4 [GSSF04].

---

[3] We will come back to these two items later in the thesis, and will refer to them as "Traetteberg's two points" for simplicity

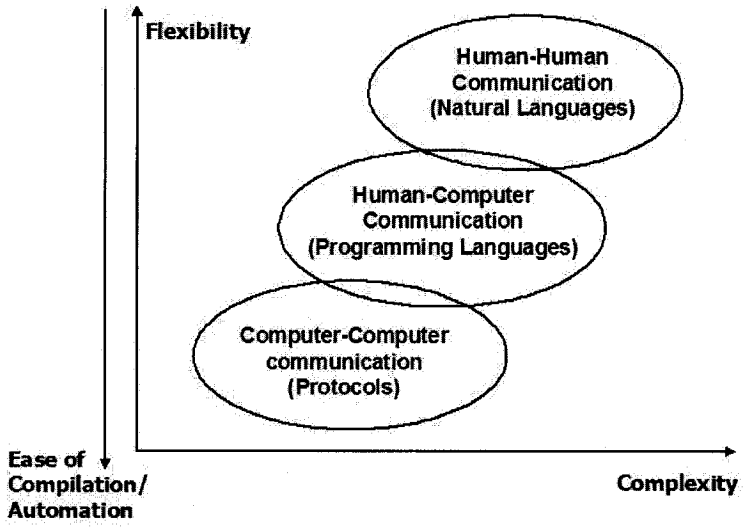


Figure 4: Languages and the communicating parties

In the paradigm of today's pattern languages in HCI domain, the two communicating entities are humans. The computer is not meant to play any role in this language concept. Adopting this point of view, we define the issue of this argument as an answer to the question: "Is it possible to redefine the HCI pattern language to be an Human-Computer Communication (HCC) Language, and not just an Human-Human Communication (HHC) Language?" It is worth mentioning that the HHC, HCC and CCC language spaces are general and are not confined to HCI patterns[4].

From the above figure, it seems that moving pattern languages from HHC to HCC space improves the level of automation (and hence the scalability) by allowing computers to help in preprocessing patterns for users, but it could also reduce flexibility by moving pattern representation from free text formats to a

[4] As we first published the subject in [GSSF04], we used "interaction" instead of "communication" in all three ellipses, so HHC, HCC and CCC were originally referred to as HHI, HCI, and CCI. However, due to the confusion with HCI acronym in HCI patterns, we replace the interaction with communication.

more restricted language model. This issue is explored in more details later in the thesis (in section 5.4).

2.5.2 Mental Models vs. Semantic Models of Pattern Languages

The importance of mental models has been long identified in literature. They are small-scale psychological representations of real, hypothetical, or imaginary situations [Cra43]. They are constructed to anticipate events, to reason and to underlie explanation. Cognitive scientists have since argued that the mind constructs mental models as a result of perception, imagination and knowledge, and the comprehension of discourse [JGL98]. It is therefore reasonable to assume that as designers read patterns, they construct mental models to represent them in their problem context. A successful pattern allows users to construct the correct mental model the pattern author intended in their envisioned design artefact. Referring back to the two points of Traetteberg, we can see the mental model applicability as implementing the second point, namely supporting the design process, albeit only manually so far.

An interesting observation is the relationship between mental models and semantic models. We discuss the semantic model applicability as implementing the first point. Data is written with specific syntax and semantics. The syntax is determined by the language alphabets and grammar used to write data. Data semantics describe the underlying meaning of data regardless of the way it is written. While a specific syntax is required to represent data, it is the underlying semantics that carry the actual knowledge within it. In [GP04], Bosak asserts that true data interoperability requires not just interoperable syntax, but interoperable semantics. Being able to explicitly represent the underlying semantics of data allows the computer to process it the same way we do as we read a text document. For example, an XML document is represented in a format that

explicitly exposes its semantics in such a way that can be understood and processed by software.

2.6 Major Uses of Patterns and Pattern Languages

In this section, we define and categorize the prevailing concepts underlying the usage of patterns as seen by different people. We also analyze the goals and expectations behind them. We emphasize that these concepts and approaches are not orthogonal; they can complement- or intersect with one another.

2.6.1 Knowledge Encapsulation

Patterns represent design knowledge in a form that can be understood, learned, and applied to situations that are partially similar to original cases. The rationale is that abstracting and encapsulating experience and best practices in a comprehensible format would allow novice users to produce quality design artifacts without relying on direct help from experts. [Cas97] demonstrates the potential of using patterns over templates, standards and guidelines in creating usable interactive systems. HCI and other software patterns are primarily concerned with the similar objective of supporting knowledge transfer [Bor00].

2.6.2 Lingua Franca

The interface design of an interactive system can be a challenging task, especially for software developers who are not familiar with usability engineering techniques and human interaction theories [MR92]. Experienced designers focus on the creation of an artifact that integrates various behavioral theories and technologies with no real expectation of evaluating one variable at a time [ZEBP04]. They use their implicit knowledge to produce quality design. Usability

experts often take a more scientific approach, looking at specific elements and their demonstrable optimization with some scientific analysis. Additionally, software developers are interested in finding the applicable design and implementing it correctly, right out of the box. This is a fertile soil for patterns as they provide the mechanism to integrate and satisfy different goals by different stakeholders.

In this direction, patterns have been seen as a *lingua franca* or common design language for improving communication by providing a common vocabulary for design. The focus is on sharing basic, broad meaning with less emphasis on technical design issues. The use of a lingua franca is intended to help cross the cultural and professional barrier between these stakeholders [Eri02].

2.6.3 Building Blocks for Knowledge Construction

As building blocks, patterns have a larger scale of granularity than objects and classes to compose systems [CHV00]. Given a broad collection of patterns on different scales of granularity (e.g. architectural patterns, design patterns, language-idioms) it should be possible to combine and glue them together into a design that can be mapped to different programming languages. Based on the abstraction concept of patterns, this approach enables designers to talk about software in different design stages using patterns. At some point it is sufficient to know that we are using a specific pattern (its name and general characteristics). Details about the constituents of this pattern or how it is actually implemented are only the subject of later stages.

2.6.4 Patterns and the Design Process

It has been reported that HCI patterns and pattern languages can act as driving force in the model-based development approach as well as the user-centered design lifecycle in general [LG99], [Bor00], [Eri00]

Within the scope of the development of interactive applications, POD, as discussed earlier is one attempt to seemingly integrate patterns into the design process. On the basis of the UPADE Web Language [GMS05], the POD approach aims to demonstrate when a pattern is applicable during the design process, how it can be used, as well as how and why it can or cannot be combined with other related patterns. Developers can exploit pattern relationships and the underlying best practices to come up with concrete and effective design solutions.

Similar to POD, the "Pattern Supported Approach" (PSA) [GL99], [GL01] addressed patterns not only during the design phase, but during the entire software development process. PSA aims to support early system definition and conceptual design through the use of HCI patterns. In particular, patterns have been used to describe business domains, processes, and tasks to aid early system definition and conceptual design. The main idea of PSA is that HCI patterns can be documented according to the development lifecycle. In other words, during system definition and task analysis, depending on the context of use, it can be decided which HCI patterns are appropriate for the design phase. In contrast to POD, the concept of linking patterns together to result in a design is not tackled. Study of this topic is elaborated in Chapter 4.

2.6.5 Patterns as Building Blocks

[BFVY96] developed a tool to generate C++ code from the patterns published by Gamma et al. [GHJV95]. Generating code from pattern descriptions gives rise to the idea of generative design patterns. In the field of software engineering, generative design patterns (GDP's) are used to promote rapid prototyping and provide a mechanism for code reuse at a high level of abstraction [SBG99]. GDP's are both useful and productive in leveraging design re-use into code re-use [MSS+04].

In [MMP02], Molina et al. proposed the Just-UI framework, which provides a set of conceptual patterns that can be used as building blocks to create interface specifications during analysis. In particular, conceptual patterns are abstract specifications of elemental interface requirements such as: *how to search*, *how to order*, *what to see*, and *what to do.* Molina also recognized that the mostly informal descriptions of patterns today are not suitable for tool use. Within the Just-UI framework, a fixed set of patterns has been formalized in order to be processable by the "OliverNova" tool [CT04]. Eventually, based on the analysis model, the JUST-UI framework uses code generation in order to derive an implementation of the UI.

2.7 Formats for Writing and Structuring Patterns

There are different ways to document patterns, including text, figures, diagrams and code fragments. Different formats have been proposed for organizing this information. Alexander's original patterns were presented in a fairly informal, narrative style. The text had an implicit structure as will be shown later. The Gamma format presented in the GoF book "Design Patterns" was much finer grained with more details, decomposing each pattern into many sections. The structure used in patterns depends on several factors. Authors have their own preferences. Different subject matters may influence the structure. For example more technical information can call for patterns with more examples. Diverse audiences may call for different formats as well. Novice readers may prefer a more prosy style while more experienced readers may prefer a more formal approach. What matter is that we should provide a consistent structure so patterns may be easily understandable and comparable.

2.7.1 Presentation vs. Structure-Oriented View

In the *presentation-oriented* view, a pattern is seen as a "Problem, context, solution" triangle. This view is more user-interface related, and has more focus on the presence of a solution and less focus on how to implement it. In this view, the focus is roughly equally distributed between the three parts of the triangle. Granularity is very high. Solutions are generally sketchy, providing only high-level recommendations with much less focus on final details or implementation. Examples are present in the form of simple graphs, or a snapshot of an existing user interface with few, if any, details about implementations. This is typically common in many HCI patterns.

This form of solutions can be strongly linked to an interesting observation by Christopher Alexander in his book *"Notes on the Synthesis of Form"* [Alx70]. He indicates the relationship between repetition and patterns with the example of a traffic junction as depicted in Figure 5. Wider lanes reflect more usage (more repetition) than thinner ones, and arrows (the pointed tips of lines) show the direction of flow. If we imagined observing an existing "natural" junction in a trail in some forest or grassland after being used over a long time, we can correlate the width of each lane to the traffic flow in it.

From this existing evidence of repetition, let us assume that we want to replace this natural junction with a new paved road. Unless there is a problem that necessitates change, we should design the new junction to look like the existing one; that is how it has been successfully used over time. We can design the width of the new traffic lanes as a solution to the lane width problem without relying on complex formal methods to derive and validate the solution. The problem, context, and solution elements are present here, and they have matured over time. We have an excellent pattern. Repetition is present here as discussed in the beginning of the chapter. The same junction can also be used

as a solution to the intersecting trails problem in other places with similar context of use[5].

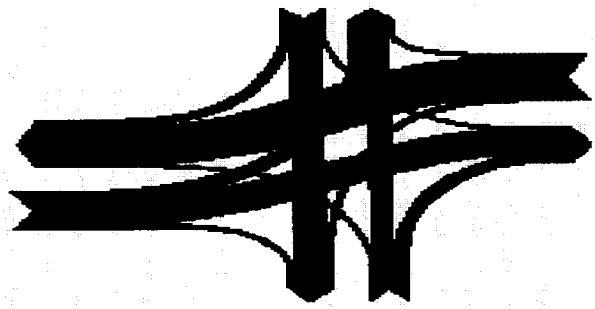


Figure 5: The traffic junction form and pattern

Similarly, in presentation-oriented view of user interface patterns, interfaces are observed over and over again, and repeated patterns are recorded to reflect the successful interaction performed by users. Validation exists by nature of this repetition. The behavior of users as to explain "why this is a successful interaction" might follow, but it is not used as the main method of finding the solution. Like the traffic junction example, designers did not invent or validate the solution; they just observed an existing junction.

In the *structure-oriented* view, prevalent is software engineering, a pattern is seen as a collection of interacting classes and objects. It works as a micro-solution that can participate in the final design and implementation. Most of the effort is directed towards a solution, with brief analysis of the problem and context. More details are provided as to how a solution can be actually implemented. The details of few classes or objects working together to form a

---

[5] This example is for demonstration purpose only. Civil engineers use formal methods to find and validate a solution. In HCI, we still have few –if any- formal methods to find and validate a UI design, so the observation of repetition is still widely applicable in HCI domain.

solution is typical of this view. Often, the details of some classes and objects are also provided in the form of platform-specific source code implementation examples (object-level granularity). This view shows some examples down to the tiniest details as made by experts. But it is also harder to create such a pattern, as it requires a higher level of expertise than the presentation-oriented view.

While complete design solutions are considered coarse-grained and classes and objects are considered fine-grained, patterns are seen as medium-grained components in both views. They can be anywhere between these two ends of granularity spectrum. They offer design suggestions that are "tested and true", thus alleviating the burden of formally predicting the success or failure of new solutions. While both views rely on the concept of "tested and true", there are subtle differences.

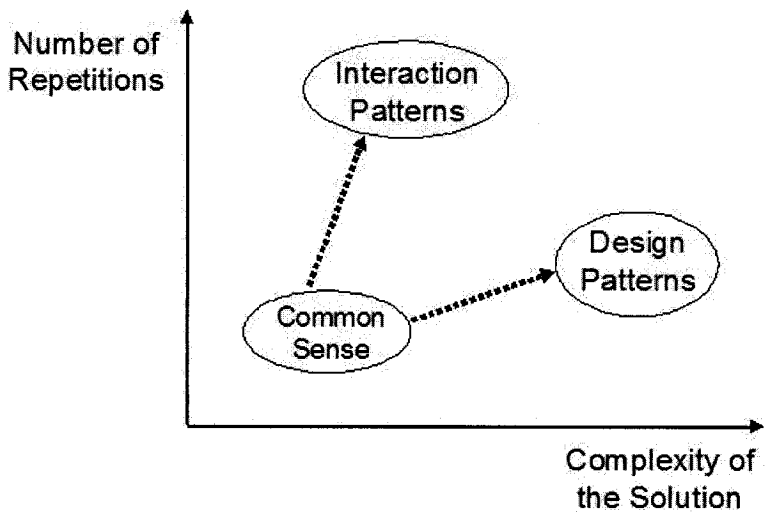


Figure 6: Patterns' complexity vs. number of uses

Figure 6 compares patterns based on the number of uses before a pattern is named, and the complexity, or the "ingenuity" of the solution that underlie the pattern. As stated in [GHJV95], design patterns originated as common sense solutions known in the programming community. They were then improved and

refined by their authors over several development projects until they reached their recent formats. They often have elegant and sophisticated ideas that are not easy to come across by a novice designer. Interaction design patterns, on the other hand, typically originate from authors observing many interfaces (Web and software application) and recording the common behavior of users and the common interaction idioms. Being used regularly, an interaction pattern would likely indicate a "solution" to an issue, but not necessarily a complex one. Moreover, while design patterns were used –and refined- few times by their authors before being called patterns, interaction patterns are typically used many more times –possibly several order of magnitudes- before being identified as patterns. Figure 6 also explains the difference between design patterns, interaction patterns and a "common sense" solution, which we can see as a low-complexity solution that is used only a few times.

2.7.2 How many times: A Famous Question

Figure 6 can provide discussion visualization to the debate about the number of uses of a solution before calling it a pattern. Some arguments call for a minimum number of actual applications before a solution can be called a pattern (the rule-of-three, [App00], however it is not specified if these applications should be done internally by the pattern author, as in the Gamma collection or externally as used by different interface examples, as in Welie's collection. While the discussions about numbers continues in different usability issues [BBC+03], some authors [Fin02] argue that even without any repetition, a solution can be called a pattern when it is "invariant" allowing it to be applied in different contexts.

Originally, Nielsen provided a mathematical model to argue for the sufficiency of five users in usability testing [Nie93]. After great controversy, he recently called for caution and pointed at the danger of quantitative studies [BBC+03] and [Nie04]. He warned against using unnecessary numbers, calling it "*number Fetishism*" in qualitative research. Therefore we believe that determining the

number of repetitions before validating a pattern should be left to individual cases.

2.7.3 Internal Pattern Structure

Pattern contents have been presented to users in different formats. Christopher Alexander captured the spirit of patterns in an organized approach in his patterns. While written in text format, the text was structured around an implicit algorithm, known as the Alexandrian Algorithm (HCIPattern.org).

```
If   you find yourself in CONTEXT
     For EXAMPLE,
     With PROBLEM,
     Entailing FORCES

Then For some REASONS,
     Apply DESIGN FORM and/or RULE
     To construct SOLUTION
     Leading to NEW CONTEXT and OTHER PATTERNS
```

Following this algorithm, he built several patterns to help architecture designers make use of his renowned insight and experience. The main components of each pattern are clear. The text format and the implicit structure make it interesting to read Alexander's patterns, and make them more interesting as you read them again. We believe that he intended to provide a high quality text that is meant to be read by highly intellectual people, over and over again. That said, we also believe that the text is less suitable when it comes to reading a growing number of patterns written by new authors. Moreover, it is obviously hard to have tool support for this highly intellectual text format.

The next milestone in design pattern development is attributed to the well-known book of "Design Patterns" [GHJV95]. As shown in Figure 7, they used simple graphical illustrations, UML diagrams and code segments. The book became an unprecedented success that drew the attention to patterns as an effective way of communicating experience to novice users.

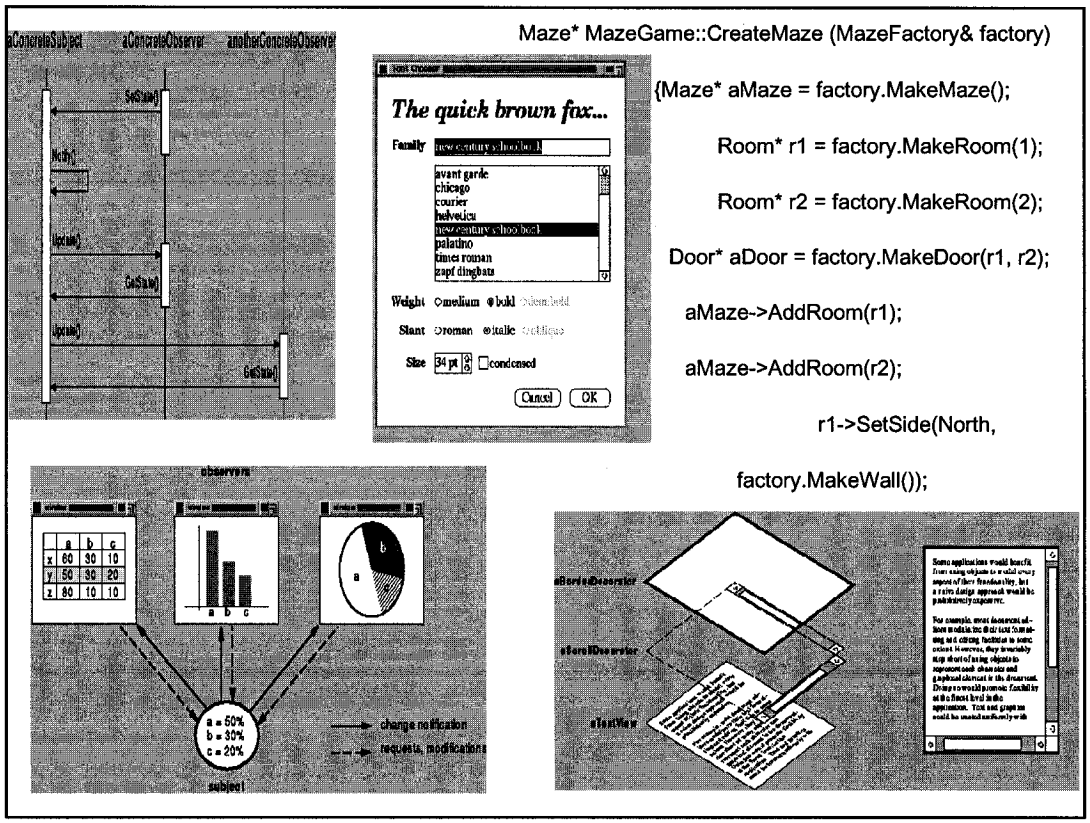


Figure 7: Presentation style of design patterns of GoF

Several contributions followed this work, and they varied greatly in their format. In the domain of Human Computer Interaction, [Bor01] published his collection of interaction patterns following the exact same template proposed by Christopher Alexander. [CL98] built another collection of HCI patterns, named "Experience" using same style. In their collection, they used a free narrative text to present patterns. No template was explicitly used. However, their text was implicitly structured the same way Alexander did in his format.

Common Ground [Tid97] and Amsterdam collection [Wei00] were other early attempts to publishing more HCI patterns. Unlike Borchers', Welie and Tidwell used new templates. The major difference is that they used explicit format, and provided names for each part of a pattern. They used different names, and order. Table 1 depicts Van Welie's template and compare it to Tidwell's. It is clear that the basic ingredients are the same, though some labels and the order of the labels started to vary from Alexander's format and from each other.

Table 1: The names and order of pattern items in two collections

Attribute Vs. Author	**Van Welie**	**Tidwell**
Name	Yes	Yes
Problem	Yes	Yes
Forces	No	Yes
Context	Yes (called When Use)	Yes
Solution	Yes	Yes
Why	Yes	No
Examples	No	Yes
More examples	Yes	No
Bad examples	No	Yes
Known uses	Yes	No
Related pattern	Yes	No
Resulting context	No	Yes
Graphics and images	Yes	No

Other collections quickly followed the trend and were published on the Internet and in books.

2.7.4 Inter-collection Structure

A way of organizing several patterns within a collection is to connect them together in a network of patterns, with some hints about how they could be combined together. This was applied in many pattern collections, either by graphically illustrating the network, or by connecting some pattern together with added text (e.g. pointers in a "Related Pattern" section of each pattern). It was

agreed upon that this network be called a "pattern language" which typically covers all patterns in one collection. Most pattern authors connected their pattern in a similar way.

2.7.5 Intra-collection Structure

As seen above, most authors imposed some structure within each pattern in their collection (an algorithm or a simple template) as well as a structure between patterns within the same collection. But as the number and formats of patterns increased, the next logical step was to try to arrange the collections together.

Jan Borchers listed several links to other collections to guide users to them. Other major players like Welie and Tidwell agreed to do the same thing. However, this approach led to the creation of a complex ad hoc "pattern maze" and large closed reference loops where users keep following some links in closed circles [Gaf04].

There were other, more formal, attempts to put some structure between different collections. Meszaros and Doble published a "Pattern Language for Pattern Writing" [MD97]. They used a fixed template for their work, which was closely similar to Alexander's, but explicit. The main goal of their collection was proposing how to formalize the activity of writing pattern for all pattern authors, a kind of meta-pattern language. For example they suggested the following patterns:

- Pattern: *Pattern* (an abstract concept of when to write a pattern)
- Pattern: *Pattern Language* (an abstract concept of when to write a pattern language)
- Pattern Structure Patterns (several guidelines for structuring pattern contents)
- Pattern Naming Patterns (unified naming conventions for patterns)
- Patterns for making Patterns Understandable (How to write data contents)

These meta-patterns were meant to suggest a standard concept for pattern authors to writing patterns and pattern languages. The idea of providing abstract concepts about patterns was not followed by further activities to apply the abstraction in a common framework.

2.8 Fallacies Related to Misrepresentation

Patterns came with a promise that has been accepted among researchers; the reuse of design knowledge. Logically speaking, there is nothing wrong with the concept of using patterns as a means to abstract and encapsulate knowledge for reuse. However, as we observe and analyze the actual practices associated with this concept and implemented by several people, we can identify some of the common, but inaccurate assumptions that underlie patterns and that are greatly dependant on the way patterns are represented to their users.

To investigate that, we designed an experiment to observe pattern reuse in two main parts. Part 1 was a lab experiment involving seven groups of volunteer interface designers with a moderate experience in interface design (two or more years). In this part of the experiment, we asked the participants to accomplish simple tasks of specific patterns lookup, comprehension and reuse. In part 2 of the experiment we observed and recorded the status and progress of pattern generation and reuse activities over a three-year period ([Gaf04] and [Gaf045c] as well as in appendix E). These informal investigations were useful in quantifying the difficulties faced by novice pattern users trying to find and apply patterns.

In part 1 of the experiment, we used some usability tests and inspection methods to identify the status of pattern reuse and the dissemination process. For example, we used heuristic evaluation as an easy, straightforward technique that can be applied by experts to evaluate pattern reuse activities. Interface designers

were introduced to the advantages of pattern reuse, and then they were asked to postulate some heuristics about how to find and use patterns in their own design environment as well as within a common interface we chose. Designers were asked to reuse patterns in the lab for both the arbitrary as well as the selected interfaces to observe the validity of their heuristics. They were then asked to evaluate if they were able to find suitable patterns and if they applied them in designing both interfaces; the one we selected and the one they chose. The selected interface helped us to evaluate HCI patterns reuse in the same domain by different designers hence focusing on the interface as an invariant. The arbitrary interfaces helped to evaluate the reuse in different domains hence focusing on the expertise as an invariant (each designers chose the interfaces in the domain they were most familiar with). Both cases lead to similar findings as explained next.

2.8.1 Fallacy 1: Finding the Right Pattern is Straightforward

In the experiment, we have observed a steady and even abrupt increase in the number of pattern-related books on the site of Amazon.com. The number of matches increased as follows:

Table 2: Pattern-related books at Amazon.com

Date	Number of matches
June 2002	8633
January 2003	11852
January 2003(2 weeks later)	11857
March 2004	114, 101
September 2004	172, 140

Besides books, we also observed other categories that involved patterns. The matches included books about patterns in virtually every imaginable domain (some examples are textile and carpet patterns, emotional and psychological

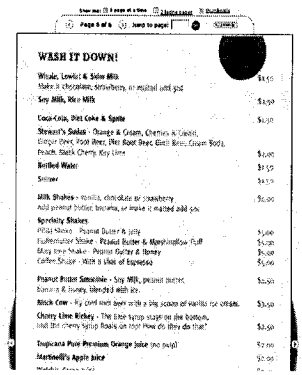
patterns, children games as well as programming and design patterns; all mixed together). We also observed that there were no concrete criteria offered to narrow our selections of patterns. We used different keyword search to reduce the number of hits without much success. In the March 2004 observation for example, the smallest number returned from search refinements were still over 2500 books. Some refinements returned all the 114, 101 books. While it was possible to rank the results according to different criteria (like the sales ranking, or by author), some very good books about patterns were not observed in the first five hundred results. In appendix E we provide more details about the problems associated with ranking the results and other issues. An important conclusion of the study is that "*unless it is know in advance about a good book of patterns or a pattern collection, it is unlikely to be located by a novice user*". Another important conclusion is that "*There are no tools that can help in effectively locating patterns on the Internet*". Moreover, "*when using the standard search engines, there is no common cataloging or categorization concept in place to help in the lookup for patterns*". We conducted a similar study using Google, and the results were similar. The problem is emerging and has been recently recognized in other areas too. As reported in IEEE Spectrum journal in November 2004, [Luc04] explains that attempts to look up specific contents on the Web using Google returned 1.8 million hits in 0.2 seconds, and he emphasizes that he "was impressed with the speed, but not the results"

Another observation of the study was the "wrong hits". We present three examples in Figure 8:

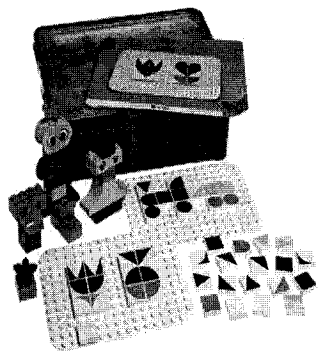
1. Under the keyword "Learning patterns", we got an infant-to-toddler-rocker chair with learning
2. Under the keyword "mathematical patterns", we got "math, shape, and patterns set" which was a game by Lego Dacta for children 4 years an up.
3. Under the keyword "menu patterns" we got several patterns from actual restaurant menus.

By looking at the nature of these unwanted hits, we estimate that a considerable number of useful HCI patterns could simply go unnoticed. Many patterns on the Internet could be missed by users due to the lack of a suitable keyword. In the study we show that the very high popularity of the word "pattern" has a major effect on the ineffectiveness of text lookup.

(a) Learning patterns



(b) Menu patterns



(c) Math patterns

Figure 8: Search results from Amazon.com

2.8.2 Fallacy 2: Comparing Patterns is Straightforward.

If we observe patterns, we find that different authors or communities use different formats, elements and attributes, to describe their patterns. They are broadly classified here into the three different sets shown in Table 3. This listing is by no means conclusive and new sets are still emerging).

Even though we have different pattern properties, elements and attributes, introduced by pattern authors, we have many similarities in the elements and the order. The paradox is that novices, for whom the patterns are intended, are confused by having different formats for documenting patterns in the market

place. The lack of common elements to describe patterns creates complexity and ambiguity among pattern users and even educators and researchers. If we want to promote the use of patterns we have to make them simple, easy to understand and flexible to use.

Table 3: Different pattern attributes

BASIC	**Extended**	**Elaborate**
Name	Name	Name
Problem	Intent	Intent
Context	Context	Applicability
	Forces	Motivation
		Also Known as
Solution	Solution	Structure
		Participants
		Collaborations
	Resulting Context	Consequences
		Implementation
	Examples	Sample Code
	Known Uses	Known Uses
	Related Patterns	Related Patterns

This can be done only if pattern writers compromise to embrace the fact of having a generalized format to describe the patterns. It is not the case anymore that a pattern user will have to pick up and use only one collection. There are several good collections out there. We can attribute the difficulty in understanding and comparing patterns to the inconsistency of pattern formats. In 2003, Gaffar et al. published a position paper in CHI workshop, calling for the unification of formats [GSJS03]. After collaborating with many HCI pattern authors including Vanderdonckt, Tidwell, Welie, and Borchers as well as several researchers from IBM, we collectively agreed on a common format for patterns, the Pattern Language Markup Language PLML [FF03] for the first time. Later in the thesis, we will provide more details on how we expanded this format into a generic pattern model.

2.8.3 Fallacy 3: Implementing Patterns is Straightforward

In many pattern representations, unlike the format of Gamma (GoF), there are hardly any hints about implementation structure and strategies. If an object oriented software developer wants to apply text patterns he/she might face a major challenge in interpreting them into the implementation process. Therefore, it would be helpful to have the implementation structure and strategies along with examples, figures, and –ideally- sample code or pseudo code so that the software developers can easily apply the pattern into their respective domain. The use of patterns is consequently hampered by lack of implementation strategies.

2.8.4 Fallacy 4: Patterns Explain the Consequences of Using Them

Design patterns are defined as solutions; therefore users assume that a pattern is a guarantee of solving a problem. The possible negative implications or misapplications are not always considered. There are often consequences or side effects that can arise as a pattern is applied. Some authors vaguely provide a list of positive and negative implications. The consequences should be carefully investigated and then made clear by providing a scientific analysis bases –for example- on sound usability concepts and theories as well as empirical validation. This is an important difference between a solution and a pattern, and we believe it makes for good patterns. This can facilitate the patterns authenticity, trustworthy and acceptability. For example, if a pattern is claimed to improve the response time but also to overload the memory, this knowledge will provide two-fold benefits to the users. First the user may decide to use it but will be aware of checking the response time to validate the improvement. Second, if

users need to reduce memory loads, they might avoid using this pattern altogether, which will save time and efforts and obviously the development cost.

2.8.5 Fallacy 5: Finding a Functional Combination of Patterns is Straightforward

To solve any real world or complex problem we may need to apply a series of patterns. Therefore, it is essential to have a consistent and complete list of closely related or coherent patterns along with the pattern description. Even though in some formats of patterns we do have this list, it is typically limited in the sense that they are listed based within their own local collection, using narrow domain of applications. Moreover, the lack of naming convention for the pattern elements, which was discussed earlier, and having several aliases for some patterns, makes the list of related patterns more complicated. For example, "Experience" described by Coram T. and Lee J. [CL98], Figure 9, shows how pattern relationships should be. While the names of the patterns are not clear in the figure, our focus is on the complexity of the network connecting them. Being sophisticated, it could take a long time to find out which patterns are related to which or just to find out the root pattern. More importantly, it is not always easy to figure out how to link this network to a comprehensive design process. With the current number of patterns offered, users could simply switch to a less sophisticated collection, without trying to understand the contents or the quality of this work.

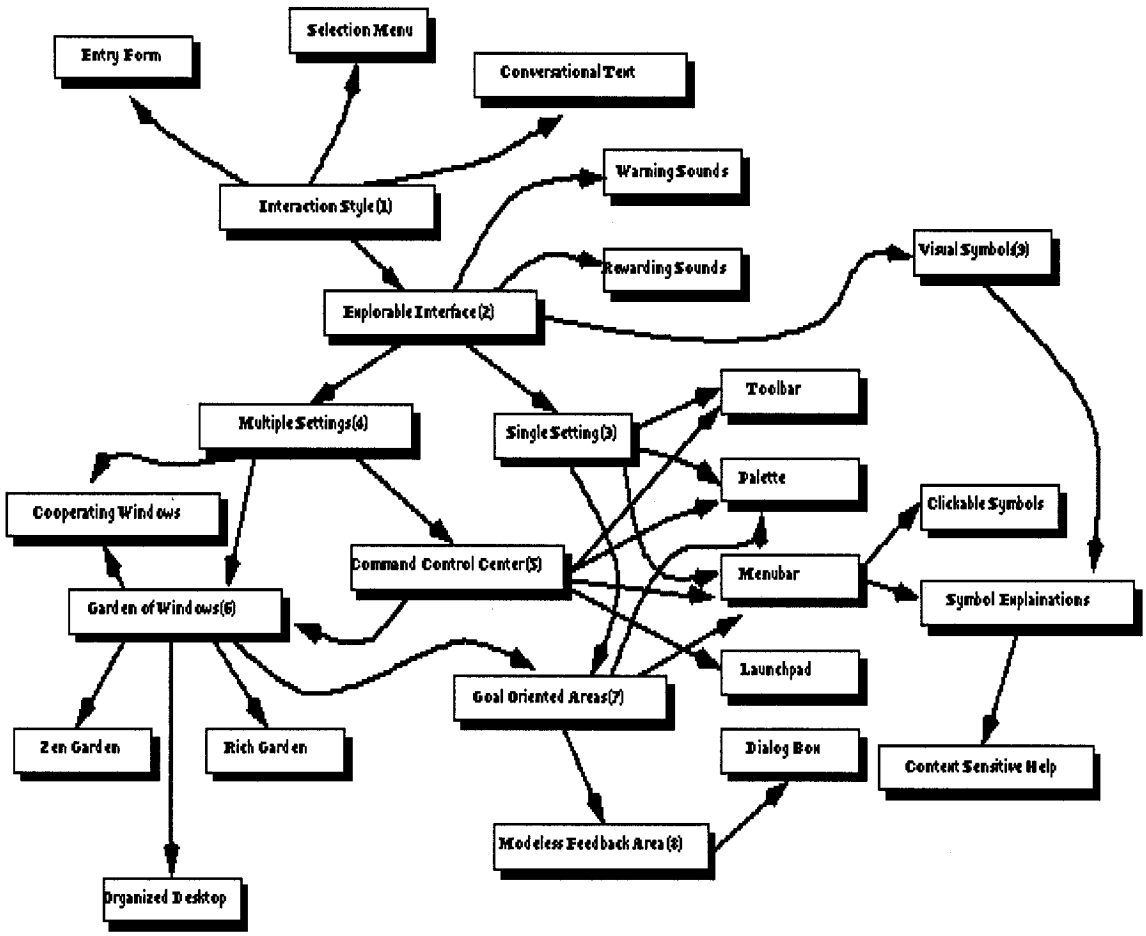


Figure 9: Maps of pattern relationships in Coram and Lee

2.9 Conclusion

In this analysis of the origin, definition and development of patterns, user interface design patterns are considered within the bigger picture of patterns. HCI patterns have shown to be useful in supporting UI designers as an alternative or an addition to working directly with HCI and usability experts or using design guidelines [AMC03], [Fin02]. Several pattern authors have supported this idea by publishing patterns to encapsulate best design knowledge and practices. Little has been done, though, to ensure the effective documentation, representation and delivery of these patterns to patterns users. The perception of pattern

languages is often limited to collection of statically connected patterns, in terms of simple arrows. The majority of pattern users have difficulties locating useful patterns, instantiating and applying them in their context. This chapter highlights the problem of ineffective pattern reuse by identifying some of the fallacies involved in finding and using patterns. We established that the effective reuse of design knowledge is more than just publishing patterns using different formats. Patterns should be appropriately documented to increase their usefulness, usability and accessibility. Pattern users should be guided through the process of reusing patterns.

The lack of common and standardized notation and a central repository for patterns make it hard to achieve this goal. A further challenge is the lack of tool support, which makes it difficult to capture, disseminate and apply patterns effectively and efficiently. Tools need to be developed with three major objectives in mind: First, as a service and support to UI designers and software engineers involved in UI development. Second, as a research forum for understanding how patterns are really discovered, validated, used and perceived. Third, to use as a prototypical implementation that support every step of a complete patterns lifecycle.

Chapter 3

Using Patterns to supplement Software Architecture with Usability

ABSTRACT: Traditional interactive system architectures such as MVC and PAC decompose the system into subsystems that are relatively independent, thereby allowing the design work to be partitioned between the user interfaces and underlying functionalities. Such architectures also extend the independence assumption to usability, approaching the design of the user interface as a sub-system that can be designed and tested independently from the underlying functionality. This Cartesian dichotomy can compromise usability as functionalities offered in the user interface can depend on the internal architecture of the system. We give several scenarios to demonstrate this effect. Based on our day to day experience with usability practitioners we model the relationships between internal software attributes and externally visible usability factors. We propose a pattern-based approach for dealing with these relationships to enhance usability aspects of the system. We conclude by discussing how these patterns can lead to model-driven approach for improving interactive system architectures, and how these patterns can support the integration of usability in the software design process.

3.1 Introduction

Software architecture is defined as the fundamental design organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design, development and evolution (ANSI/IEEE 1471-2000, Recommended Practice for Architectural Description of

Software-Intensive Systems). In addition, it encapsulates the fundamental entities and properties of the application that generally insure the quality of application.

In the field of interactive systems engineering, architectures of the 1980s and 1990s such as MVC and PAC are based on the principle of separating the functionality from the user interface. The functionality is what the software actually does and what information it processes, thereby offering behavior that can be exploited to achieve some goals. The user interface defines how this behavior is presented to end-users and how users interact with it. The underlying assumption is that usability, the ultimate quality factor, is primarily a property of the user interface. Therefore separating the user interface from the application's logic makes it easy to modify, adapt or customize the interface after user testing. Unfortunately, this assumption does not ensure the usability of the system as a whole.

We now realize that system features can have an impact on the usability of the system, even if they are logically independent from the user interface and not necessarily visible to the user. Bass observed that even if the presentation of a system is well designed, the usability of a system can be greatly compromised if the underlying architecture and designs do not have the proper provisions for user concerns [BJK01]. We propose that software architecture should define not only the technical interactions needed to develop and implement a product, but also interactions with users.

At the core of this vision is that invisible components can affect usability. By invisible components, we mean any software entity or architectural attribute that does not have visible cues on the presentation layer. They can be operations, data, or structural attributes of the software. Examples of such phenomena are commonplace in database modeling. Queries that were not anticipated by the modeler, or that turn out to be more frequent than expected, can take forever to

complete because the logical data model (or even the physical data model) is inappropriate. Client-server and distributed computer architectures are also particularly prone to usability problems stemming from their "invisible" components.

Designers of distributed applications with Web interfaces are often faced with these concerns: They must carefully weigh what part of the application logic will reside on the client side and what part will be on the server side in order to achieve an appropriate level of usability. User feedback information, such as application status and error messages, must be carefully designed and exchanged between the client and server part of the application, anticipating response time of each component, error conditions and exception handling, and the variability of the computing environment. Sometimes, the Web user interface becomes crippled by the constraints imposed by these invisible components because the appropriate style of interactions is too difficult to implement.

Like other authors [BJK01] and [FB02], we argue that both software developers implementing the systems features and usability engineers in charge of designing the user interfaces should be aware of the importance of the intimate relationship between these features and the user interfaces. This relationship can inform architecture design for usability. With the help of patterns, this relationship can help integrate usability concerns in software engineering.

In section 3.3 we will identify scenarios where invisible components of an interactive application will impact on usability; we will also propose solutions to each scenario. The solutions are presented in the form of patterns. Beyond proposing a list of patterns to solve specific problems, we discuss our long-term goal to define a framework for studying and integrating usability concerns in interactive software architecture via patterns.

3.2 Background and Related Work

The concept of separating the view from the real object is relatively old. It reflects the fundamental need of reducing system complexity while improving its quality.

In database domain, a clear separation between table and view help achieve many goals [Gaf01]:

- Increase security by masking classified information in tables from the views presented to unauthorized users.
- Reduce redundancy by storing atomic, normalized data in efficiently designed tables while displaying them in different views according to users' needs
- Increase system flexibility by allowing the addition of new views as needed without the need to change the actual tables which is a costly and risky task
- Increase data integrity by limiting the changes to normalized tables and simply updating the views as needed

This separation model is so deeply rooted in the concept of database that it is supported not only at design level but also at the language level. Most query languages offer direct manipulation to creating and manipulating tables and views separately.

The same need for separation has been identified in software interfaces. A large number of architectures for interactive systems has been proposed, e.g., Seeheim model, Model-View-Controller (MVC), Arch/Slinky, Presentation Abstraction Control (PAC), PAC-Amadeus and Model-View-Presenter (MVP) [BCK98]. Most of these architectures distinguish three main components: (1) abstraction or model, (2) control or dialog and (3) presentation. The model contains the functionality of the software. The view provides graphical user

interface (GUI) components for a model. It gets the values that it displays by querying or receiving notification from the model of which it is a view. A model can have several views. When a user manipulates a view of a model, the view informs the controller of the desired change. Figure 10 summarizes the role of each one of these three components for an MVC-based application.

The motivation behind these architecture models is to improve, among others, the adaptability, portability, complexity handling and separation of concerns of interactive software. However, even if the principle of separating interactive software in components has its design merits, it can be the source of serious adaptability and usability problems in software that provides fast, frequent and intensive semantic feedback. The communication between the view and the model makes the software system highly coupled and complex. Simplified versions like the observer model have emerged to reduce this complexity, but they don't acknowledge the presence of dependencies between the model and the views.

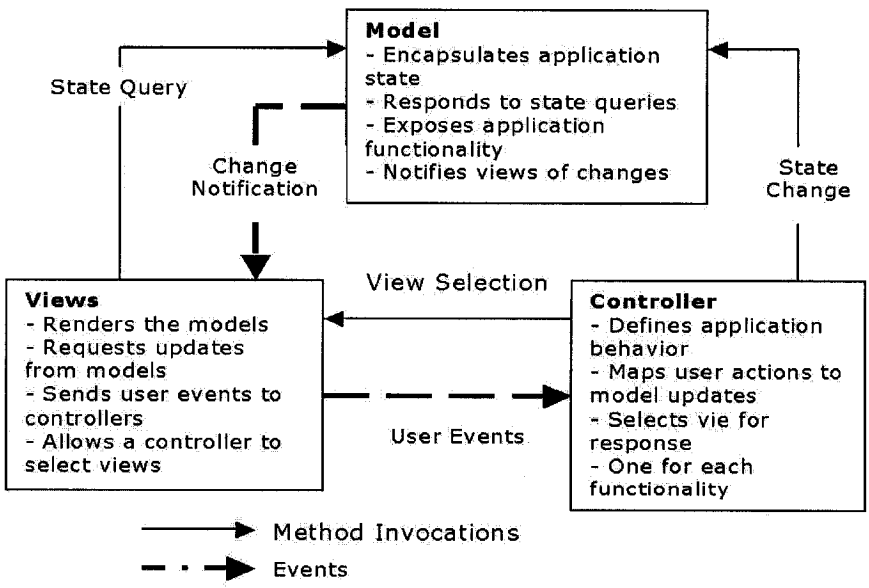


Figure 10: The roles of MVC architecture components (Sun Microsystems)

In general, these architectures lack provisions for integrating usability in the design of the model or abstraction components. For example, Len Bass et al. [BJK01] identified specific connections between aspects of usability (such as the ability to "undo") and the model response (processed by an event-handler routine).

3.3 Identifying and Categorizing Typical Scenarios

To study this intimate relationship between the model and the interface, we proposed the following methodological framework to:

1. Identify and categorize typical design scenarios that illustrate how invisible components and their intrinsic quality properties might affect the usability
2. Model each scenario in terms of a cause/effect relationship between (a) the attributes that quantify the quality of an invisible software entity and (b) well-known usability factors such as efficiency, and satisfaction.
3. Suggest new design patterns or improve existing ones that can solve the problem described in similar scenarios
4. Illustrate, as part of the pattern documentation, how these patterns can be applied within existing architectural models such as MVC.

The first step in our approach for achieving usability via software architecture and patterns is to identify typical situations that illustrate how invisible components of the model might affect usability. Each typical situation is documented using a scenario. Scenarios are widely used in HCI and software engineering [Car00]. Scenarios can improve communication between user interface specialists and software engineers who design invisible components -- this communication is essential in our approach to patterns. In this context, we define a scenario as a

narrative story written in natural language that describes a usability problem (effect) and that relates the source of this problem to an invisible software entity (cause). The scenario establishes the relationship between internal software attributes that are used to measure the quality of the invisible software entity and the external usability factors that we use for assessing the ease of use of the software systems.

The following are some typical scenarios we extracted from our empirical studies and from a literature review. Other researchers also proposed other scenarios [BJK01]. The goal of our research is not to build an exhaustive list of scenarios, but rather to propose a methodological framework for identifying such scenarios and to define patterns that can be used by developers to solve such problems. The scenarios are therefore intended as illustrative examples.

Scenario 1: Time-Consuming Functionalities

It is common for some underlying functionalities of an interactive system to be time consuming. The lack of several quality attributes can increase the time for executing these functionalities. A typical situation is the case where a professional movie designer expects the high bandwidth of high-speed Internet access when downloading large video files, but the technology available for Internet connection has lower speed, making downloading overly slow.

The user needs feedback information to know whether or not an operation is still being performed and how much longer he will need to wait, but sometimes this information is not provided. Feedback tends to be overlooked in particular when the designers of the user interface and those developing the features are not in the same team and that there is a lack of communication between them.

Scenario 2: Updating the Interface When the Model Changes its State

Usability guidelines recommend helping users understand a set of related data by allowing them to visualize the data from different points of view. A typical method is to provide graphical and textual representations of the same underlying data model.

Whenever the data model changes, the underlying model should update the graphical and the textual representations. Two main techniques to deal with this issue are *polling* –where interfaces are designed to poll the system periodically for changes- and *broadcasting* –where the system notifies the interfaces of new changes. Depending on the nature and rate of changes, the cost of notification, and the overall context of use, designers choose an optimal updating technique, and select the rate of polling or broadcasting if applicable. In certain cases, the system might not be designed to automatically update all views when one view changes. This can result in inconsistent views that can in turn increase the user's memory load, frustration, and errors.

Scenario 3: Performing Multiple Functionalities Using a Single Control

It is easier and more straightforward to use a dedicated control for each functionality and in particular for critical functions, even at the expense of more buttons and menus. This is the current practice in many standard functionalities like *File Save*, *Save As*, and *Print* options. However, for complex domain-specific functionality, this is not always the case. When a single control performs multiple operations, it requires a complex menu structure and choice of modes, which increases the likelihood of mode errors and other usability problems.

Unfortunately, there is a design tradeoff between simplicity in appearance and simplicity in use. Aggregating several related functionalities under one control, or in one procedure makes it easier for users to find and use them in one "click",

and offer a lower number of total controls, increasing the learnability of the system. This is a dangerous design trap as it clearly limits the flexibility of interacting with the system and the effectiveness of accomplishing unforeseen complex tasks. Alas, consumers (and organizations) make purchase decisions based on appearance first, so this is a fundamental conflict [Nor02].

Scenario 4: Invisible Entities Keep the User Informed

We know that providing the user with an unclear, ambiguous or inconsistent representation of the system's modes and states can compromise the user's ability to diagnose and correct failures, errors and hazards, or even simply interact with the system. This can happen when a system functionality allows the user to visualize information that competes or conflicts with currently or previously displayed information in other views. A well known example is when a user opens a Microsoft explorer window to navigate the file system and the available drives on the computer, then adds a USB memory stick (external storage device) to the system. Depending on the version of the software, the user may not be able to see the new addition in the current explorer window at all, and they may have to open a new explorer window. In other cases, they might not see it in the main window, but can see it under "My Computer" within the main window, which is an inconsistency in displaying the system state. In older systems that are patched up to support this new technology, the user can remove the USB storage device but it remains displayed in the explorer window, even if it is no more functional. All these cases vary by the version of the operating system, and are especially seen in older versions where "true plug and play" feature was not available. This feature is indeed challenging to implement and requires modifications to the file explorer software to dynamically detect the system state, and consistently refresh user's views. It is even more challenging to modify older versions by new patches to accommodate this feature. Without going into technical details, we can see the intricate relationship between the

interface and the underlying system and the confusion the inexperienced users might go through in these situations.

To avoid such situations, it is important for the functionality developers to accurately communicate the system's modes and states to the user interface designer. Ignoring this informative feedback can lead to users making wrong assumptions that may lead to inefficient or incorrect interaction. User interface designers should inform the developers about all the tangible consequences related to the states and modes of the systems.

Scenario 5: Providing Error Diagnostics When Features Crash

When a feature failure occurs due for example to exception handling, the interface sometimes provides unhelpful error diagnostics to the user.

The user should be notified of the state that the system is currently in and the level of urgency with which the user must act. The system feature should help the user to recognize potential hazards and return the system from a potentially hazardous state to a safe state. Messages should be provided in a constructive and correct manner that helps restore the system to a safe state.

Scenario 6: Technical constraints on dynamic interface behavior

Particularly in Web-based transactional systems, technical and logistic constraints can severely limit dynamic behavior of the interface within a highly interactive page. It can therefore be difficult or impossible to design elements that automatically update as a result of an action elsewhere on the same page. For example, in a series of dependent drop-down lists "Country", "Province" and "City", it may be challenging to automatically update "Province" as a function of the "Country" selection without referring back to the server after each selection to download the next dependant list. The complexity increases when combined with

business rules and restrictions. For example in an e-banking system, a user who transfers money from her checking account to pay her credit card of the same bank can see the new balance on the checking account immediately but keep seeing the old credit card balance without update. While it might look like an interface problem and the user might become upset when seeing the money deducted from the checking account but not added to the credit card, the fact might be that the bank policy explicitly prevents displaying credit card account updated until they are manually verified; within 36 hours. A perfectly correct interface would still display this inconsistency for the next 36 hours. A usable interface would be aware of some business rules and hence of this potential inconvenience. The user interface (client side) would simply notify the user of the reason, saving her a lot of frustration, especially when many users are weary of technology glitches or have less trust on Web based transactions than on teller-based interactions.

These technical constraints against dynamism are often imposed in Web-based client-server contexts due to the dictum that the business rules must be separate from the user interface. Dynamic interface behavior of an interactive system can require the user interface to have a degree of intelligence that incorporates certain business rules, which conflicts with the "separate layers" dictum. In few cases, the alternative is for the client to call the server more frequently to refresh the page dynamically, but architects tend to avoid this approach because of the presumed extra demand on bandwidth. In other cases, the only alternative is for the client to call or visit the bank to inquire about the allegedly missing money. We can see that a usable interface can be much more useful.

There is no easy solution to this problem. The most important principle in this situation is to analyze user needs relating to dynamism before making technology decisions that could have an impact on dynamism. Transactional systems often require considerable dynamism, whereas purely informational systems can often get by without dynamism in the user interface. If it is

unacceptable for business rules to immediately incorporate electronic transactional changes (as one example) which will have an impact on the interface behavior, then the interface should be aware of this business rule. it would help to incorporate some business rules into the client side. In other cases, business rules are confidential and can not be incorporated in client interfaces, reducing the usability of interface. When fully dynamic behavior is not possible, it would help to increase the network bandwidth so as to better support pseudo-dynamic behavior, involving more frequent page refreshes through calls to the server.

The preceding scenarios are used as an illustrative sample. In total, we have identified more than 24 scenarios. Len Bass also described a list of 26 scenarios, some of which were a source of inspiration for our work. Providing an exhaustive list of scenarios is certainly useful from the industry perspective. However our goal for this research is to better understanding and validate how software features affect usability in general, and as such our focus is to model the scenarios in term of a cause/effect relationship. This relationship connects the quality attributes of invisible components with recognized usability factors. Section 3.5 details this perspective.

3.4 Patterns as a Tool to Document Scenarios

There are different ways to document common solutions to the recurring problems described in the preceding scenarios, including patterns as detailed in Chapter 2. As a start, we have used text format to formalize patterns. Since the relationship between usability and internal software properties defines the problem, it has been added into the pattern descriptions that follow. This measurement relationship is what makes a pattern a cost-effective solution. In short, if a pattern does not improve at least one of the factors described in the

measurement relationship, then it is not a good pattern for the problem described in the scenario. This aspect is detailed in the next section.

The relationship between specific patterns within the same collection has been intensively investigated. We explore this aspect in Chapter 4. Meanwhile, the relationship between pattern concepts and activities in different domains has received a fair share of attention. In this regard, our main focus is on the correlation between software design patterns and interaction design patterns. Zimmerman et al. [ZEBP04] discuss it in some details. In Chapter 2, we presented and discussed them from an analytical point of view. In this section we compare them in the context of architecture-sensitive patterns:

- **Software design patterns.** The aim of these design patterns is to propose software designs and architectures for building portable, modifiable and extensible interactive systems. A classical pattern of this category is the Observer that acts as a broker between the user interface (views) and the model [GHJV95]. When the observers receive notification that the model has changed, they can update themselves. This pattern provides a basic solution to the problem described in scenario 3.2.
- **Interaction design patterns**, defined at the level of the graphical user interface. These are proven user experience patterns and solutions to common usability problems. A number of pattern languages have been developed over the last few years as shown in Chapter 2.

Software design patterns, widely used by software engineers, are a top-down design approach that organizes the internal structure of the software systems. Interaction design patterns, promoted by human computer interaction practitioners, are used as a bottom-up design approach for structuring the user interface. We believe that these two categories of patterns can be used together to provide an integrated design framework to problems described in our scenarios. To illustrate how these diverse patterns can work together to provide

comprehensive solutions, in the following sections we describe our five scenarios using interaction and design patterns.

Although a number of de facto standards have emerged to document patterns, we use a simple description with the following format:

- "Name" is a unique identifier.
- "Context" refers to a recurring set of circumstances in which the pattern applies.
- "Force": The notion of force generalizes the kind of criteria that we use to justify designs and implementations. For example, in a straightforward, simple study of functionality, the main force to be resolved is efficiency (resources complexity) or effectiveness (task complexity). However, patterns deal with the larger, harder-to-measure and conflicting sets of goals and constraints encountered in the design of every component of the interactive system.
- "Problem" refers to a set of constraints and limitations to be overcome by the pattern solution.
- "Solution" refers to a canonical design form or design rule that someone can apply to resolve these problems.
- "Resulting context" is the resulting environment, situation, or interrelated conditions. Again, in a simple system this can be easily predictable, while in complex interactive system it can be hard to find out in a deterministic way.
- "Effects of invisible components on usability" which defines the relationship between the software quality attributes and usability factors.

3.5 Software Design Patterns

The first pattern that we have considered is the Abstract Factory pattern which provides an interface for creating families of related or dependent objects without specifying their concrete implementations (e.g. The Toolkit class). In other words, this pattern provides the basic infrastructure for decoupling the views and the

models. Given a set of related abstract classes, the Abstract Factory pattern provides a way to create instances of those abstract classes from a matched set of concrete subclasses. The Abstract Factory pattern is useful for allowing a program to work with a variety of complex external entities such as different windowing systems with similar functionality. A second pattern is the Command pattern which complements the abstract factory by reducing the view/controller coupling.

To complement these basic patterns we also introduce the Working Data Visualization pattern.

Name: Working Data Visualization

(Scenarios addressed: 2. Updating the Interface When the Model Changes its State)

Problems

If the user cannot see working data in different view modes so as to get a better understanding of it, and if switching between views does not change the related manipulation command, then usability will be compromised.

Context

Sometime users want to visualize a large set of data using different points of view, so as to better understand what they are doing and what they need to edit to improve their documents.

Forces

- Users like to gain additional insight about working data while solving problems.
- Users like to see what they are doing from different viewpoints depending on the task and solution state.
- Different users prefer different viewpoints (modes).
- Each viewpoint (mode) should have related commands to manipulate data.

Solution

Data that is being viewed should be separate from the data view description, so

that the same data can be viewed in different ways according to the different view descriptions. The user gets the data and commands according to the user-selected view description.

Effects of invisible components on usability

Effect 1

- Quality attributes of invisible components: Integrity
- Usability factors affected: Visual consistency

Other relevant patterns we used include Event Handler, Complete Update, and Multiple Update [San01]. We use them to notify and update views (scenario 1) using traditional design patterns such as Observer and Abstract Factory. We incorporated these patterns into the Sub-form pattern that groups the different views in the same container, called the Form (Table 4). The Event Handler, Complete Update, and Multiple Update patterns can be applied in two phases. The first phase changes the states of the user interface models in response to end user events generated by the visual components, and the second phase updates the visual components to reflect the changes in the user interface model. Since the update phase immediately follows the handling phase, the user interface always reflects the latest changes.

Table 4: Example of design patterns

Pattern	***Problem***	***Solution***
Event Handler	How should an invisible component handle an event notification message from its observable visual components?	Create and register a handler method for each event from observable visual components.
Complete Update	How to implement behavior in the user interface to update the (observer) visual component from the model	Assume all (observer) visual components are out-of-date and update everything.
Multiple Update	How to implement changes in the model of sub-form to reflect parent of sub-form, child of sub-form, siblings of sub-form	Each sub-form should notify its parent when it changes the model. The parent should react to changes in the sub-form via the Event Handler and update its children components

		via Complete Update.
Sub-form	How to design parts of user interfaces to operate on the model in a consistent manner	Groups the components that operate on the same model aspect into sub-forms.

The next example of software design patterns we propose is the Reduce Risk of Errors pattern.

Name: Reduce Risk of Errors

(Scenarios addressed: 2. Updating the Interface When the Model Changes its State; 4. Invisible Entities Keep the User Informed)

Problem

How can we reduce the likelihood of accidents arising from hazardous states?

Forces

- Hazardous states exist for all safety-critical systems; it is often too complex and costly to find every hazardous state by modeling all system states and user tasks;
- Risk can be effectively reduced by reducing the consequence of error rather than its likelihood;
- When a hazardous state follows a non-hazardous state, it may be possible to return to a non-hazardous state by applying some kind of recovery operation.

Solution

Enable users to recover from hazardous actions they have performed. Recovering a task is similar to undoing it, but promises to return the system to a state that is essentially identical to the one prior to the incorrect action. This pattern may be useful for providing a Recover operation giving a fast, reliable mechanism to return to the initial state. Recovering a task undoes as much of the task as is necessary (and possible) to return the system to a safe state.

Resulting Context

After applying this pattern, it should be possible for users to recover from some of their hazardous actions. Other patterns can be used to facilitate recovery by

breaking tasks into sub-steps, each of which may be more easily recovered than the original task. The user should be informed of what the previous state is that the system will revert to.

Effects of invisible components on usability

Effect 1:

- Quality attributes of invisible components: Integrity
- Usability factors affected: Visual consistency

Effect 2:

- Quality attributes of invisible components: Suitability
- Usability factors affected: Operability

The last example of software design patterns is the Address Dynamic Presentation pattern.

Name: Dynamic presentation in user interface

(Scenarios addressed: 6. Technical constraints on dynamic interface behavior)

Problem

How can we avoid technical constraints on dynamic behavior of the user interface?

Forces

- Users benefit from immediate feedback on their actions.
- Dynamically updating fields can reduce the time required to accomplish a task.

Solutions

- Analyze user needs relating to dynamism before making technology decisions that could have an impact on dynamism. Transactional systems often require considerable dynamism.
- If it is unacceptable for business rules to be incorporated into the client, then it might be possible to make a business case for increasing the network bandwidth

so as to better support pseudo-dynamic behavior, involving more frequent page refreshes through calls to the server.

Resulting Context

After applying this pattern, users will have more immediate feedback on the consequences of their actions, increasing the understandability of the user interface and reducing errors; in addition, time and effort to accomplish a task will be reduced in certain cases.

Effects of invisible components on usability

Effect 1:

- Quality attributes of invisible components: Functionality
- Usability factors affected: Understandability

Effect 2:

- Quality attributes of invisible components: Suitability
- Usability factors affected: Operability

3.6 Interaction Design (HCI) Patterns

Many groups have devoted themselves to the development of pattern languages as detailed in Chapter 2. We also adapted and used some of these patterns. The first basic HCI pattern that we used is the Progress Indicator pattern [Tid97]. It provides a solution for the time-consuming features scenario (scenario 1).

Name: Progress Indicator

(Scenarios addressed: 1. Time-Consuming Functionalities)

Problem

A time-consuming functionality is in progress, the results of which are of interest to the user. How can the artifact show its current state to the user, so that the user can best understand what is happening and act on that knowledge?

Forces

- The user wants to know how long they have to wait for the process to end.
- The user wants to know that progress is actually being made, and that the process hasn't just "hung."
- The user wants to know how quickly progress is being made, especially if the speed varies.
- Sometimes it's impossible for the artifact to know how long the process is going to take.

Solution

Show the user a status display of some kind, indicating how far along the process is in real time. If the expected end time is known, or some other relevant quantity (such as the size of a file being downloaded), then always show what proportion of the process has been finished so far, so the user can estimate how much time is left. If no quantities are known -- just that the process may take a while -- then simply show some indicator that the process is ongoing.

Resulting Context

A user may expect to find a way to stop the process somewhere close to the progress indicator. It's almost as though, in the user's mind, the progress indicator acts as a proxy for the process itself. If so, put a "stop" command near the Progress Indicator if possible.

Effects of invisible components on usability

Effect 1:

- Quality attributes of invisible components: Performance
- Usability factors affected: User satisfaction

The second pattern is the Keep the User Focused pattern, which brings an integrated solution to the problems described in scenarios 2, 3 and 4.

Name: Keep the User Focused

(Scenarios addressed: 2. Updating the Interface When the Model Changes its State; 3. Performing Multiple Functionalities Using A Single Control; 4. Invisible Entities Keep the User Informed)

Context

An application where several visual objects are manipulated, typically in drawing packages or browsing tools

Problem

How can the user quickly learn information about a specific object they see and possibly modify the object?

Forces

- Many objects/views can be visible but the user usually works on one object/view at a time.
- The user wants both an overview of the set of objects and details on attributes and available functions related to the object he or she is working on.
- The user may also want to apply a function to several objects/views.

Solution

Introduce a focus in the application. The focus always belongs to an object present in the interface. The object of focus on which the user is working determines the context of the available functionality. The focus must be visually shown to the user, for example, by changing its color or by drawing a rectangle around it. The user can change the focus by selecting another object. When an object has the focus, it becomes the target for all the functionality that is relevant for the object. Additionally, windows containing relevant functionality are activated when the focus changes. This reduces the number of actions needed to select the function and execute it for a specified object. The solution improves the performance and ease of recall.

Resulting Context

The "Keep the User Focused" pattern complements the software design patterns in the following situations:

- Helping users anticipate the effects of their actions, so that errors are avoided before calling the underlying features
- Helping users notice when they have made an error (provide feedback about actions and the state of the system);
- Providing time to recover from errors;

- Providing feedback once the recovery has taken place.

Effects of invisible components on usability

Effect 1:

- Quality attributes of invisible components: Integrity
- Usability factors affected: Visual consistency

Effect 2:

- Quality attributes of invisible components: Functionality
- Usability factors affected: Understandability

Effect 3:

- Usability factors affected: Operability
- Quality attributes of invisible components: Suitability

There is not a one-to-one mapping between software design patterns and HCI patterns. The problems described in a specific scenario can require any number of HCI and software design patterns, and each pattern may be affected by a number of problems described in different scenarios. In our approach, we argue that using a few patterns can be very valuable, even without an entire pattern language.

3.7 Cause-Effect Relationships between Software Elements and Usability

So far in this chapter we focused on specific ways in which internal software properties can have an impact on usability criteria. In this section, we attempt to provide a more general, theoretical framework for the relationships between usability and invisible software attributes. In particular, among the huge or potentially infinite number of ways that invisible components can affect usability, our main goal is to understand whether there are specific places where we are

more likely to find these relationships or effects. Another goal is to verify whether there is any structure underlying these relationships, which would allow us to define a taxonomy of how usability issues arise from invisible components.

3.7.1 Traditional Model of Relationship between Invisible Software Elements and Usability

Usability is often thought of as a modular tree-shaped hierarchy of usability concepts, starting at the level of GUI objects, and abstracting progressively up towards low-level usability criteria or measures and then into high-level usability factors. Figure 11 illustrates this definition of usability and its relationship to parallel "towers" of other software attributes.

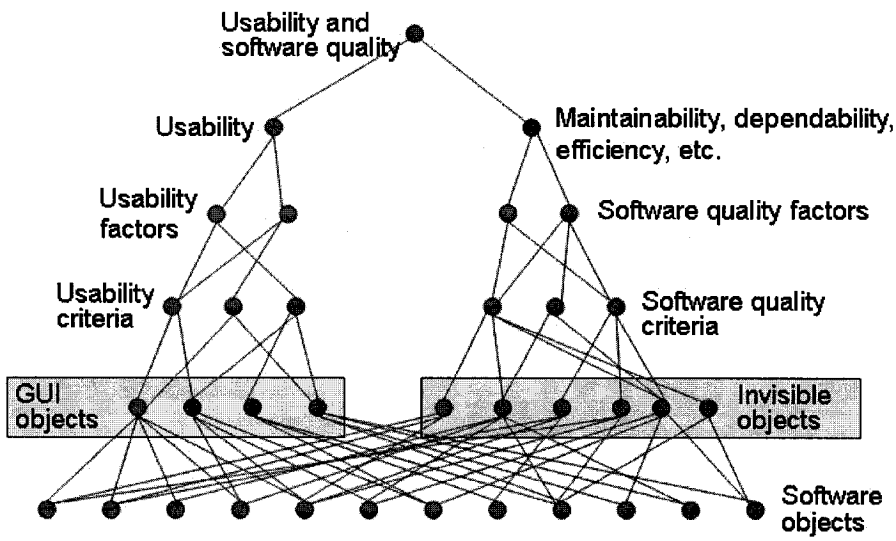


Figure 11: Traditional "twin towers" model of usability and other software quality factors

Table 5 provides more detailed information on the software quality factors and criteria referred to schematically in the right-hand branch of Figure 11. (In

principle, each quality factor would form a separate branch.) In our work, we have adopted the software quality model proposed by ISO 9126. Table 5 is an overview of the consolidated framework we have been using [SAKS03]. The table shows the criteria for measuring usability as well as five other software quality factors including functionality, reliability, efficiency, maintainability and portability. This measurement framework automatically inherits all the metrics and data that are normally used for quantifying a given factor. The framework helps us to determine the required metrics for (1) quantifying the quality factors of an invisible software entity, (2) quantifying the usability attributes and (3) defining the relationships between them.

Table 5: A partial vision of the ISO 9126 measurement framework

Software quality factor	**Measurement criteria**
Functionality	Suitability Accuracy Interoperability Security
Reliability	Maturity Fault tolerance Recoverability
Usability	Understandability Learnability Operability Attractiveness
Efficiency	Time behavior Resource Utilization
Maintainability	Analyzability Changeability Stability Testability
Portability	Adaptability Instability Co-existence Replaceability

3.7.2 Taxonomy of Usability Issues Arising from Invisible Components

Relationships between software attributes of invisibles components and usability factors have two properties:

1 They are lateral relationships between the modules of usability and architecture.
2 They are hierarchical relationships between two or more levels of description, since usability properties are a higher-level abstraction based on architectural elements.

Thus to understand the relationship, we need an approach that takes into account both modularity and hierarchy.
In software engineering, software modules have two features that need to be considered during design, namely coherence and coupling. Coherence refers to how relevant the components of a subsystem are to each other, and it needs to be maximized. On the other hand, coupling refers to how dependant a subsystem is on other subsystems, and it needs to be minimized.

In a similar approach in "The Architecture of Complexity" [Sim62], Herbert Simon discusses "nearly decomposable systems". In hierarchic systems, interactions can be divided into two general categories: those among subsystems, and those within subsystems. In a simplified approach, we can describe a system as being "decomposable" into its subsystems by basically assuming that we are fully aware of all interactions between subsystems and that every thing has been "taken care of". At this stage we can go on with the studying and development of each subsystem separately, relying on our limited model of interaction. However as a more refined approximation, it is more accurate to speak of a complex system as being "nearly decomposable", meaning that there are complex

interactions between the subsystems, and that after separation, these interactions remain active and non-negligible.

Nearly decomposable systems have two properties:

1- Modularity: In the short-run, the behavior of each subsystem is approximately independent of the other subsystems;
2- Hierarchy (or aggregation): In the long-run, the behavior of any one subsystem depends only in aggregate way on the other subsystems.

These properties indicate that in reality, the traditional model of usability is oversimplified. Although the usability subsystem is fundamentally different from the architecture, Simon's principle of nearly decomposable systems predicts that it is possible for usability properties to be affected to some degree by architectural properties. Figure 12 illustrates an interpretation of this alternative model of usability.

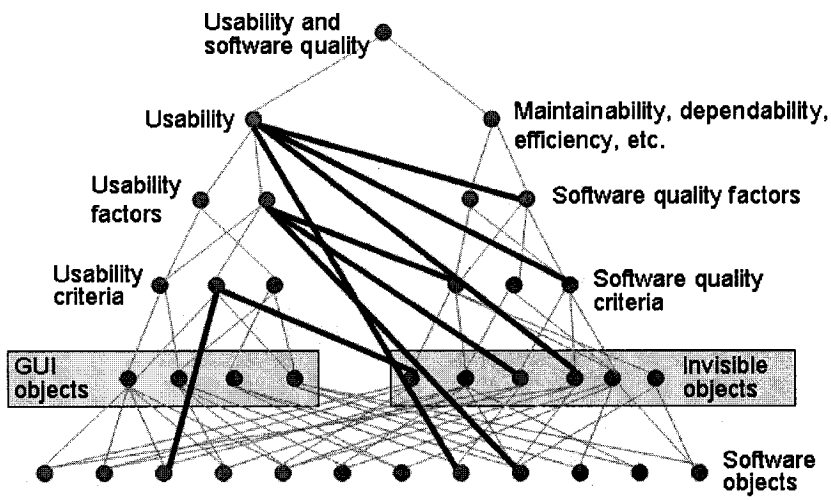


Figure 12: Revised model of usability,
including possible types of cross-relationships with architecture (bold links)

In this figure, a node (usability property) at any level of usability can potentially be influenced by nodes at any lower level of architecture, or conceivably even by combinations of several different levels of architecture. Figure 12 is a first approximation.

Simon's second principle of near-decomposability states that subsystems depend in only an aggregate way on other subsystems. This principle implies that if architecture has an effect on usability, it will tend to be in an aggregate way and therefore at a higher level of architecture, rather than through the effect of an individual low-level architectural component. We interpret this principle to mean that the effects of architecture on usability will tend to propagate from levels of architecture that are closer to the level of usability, rather than farther away.

Therefore to refine the model, we will assume that the most likely relationships occur between usability properties and the immediately closest lower architectural level, and that more distant architectural levels have an exponentially decreasing probability of having an effect on usability. The revised model, based on this assumption, is illustrated in Figure 13. This model reflects a more clearly recursive definition of usability.

Based on Simon's principles of nearly decomposable systems, we can conclude that these types of relationships between architecture and usability are the exception to the rule, but frequent enough that they should not be neglected.

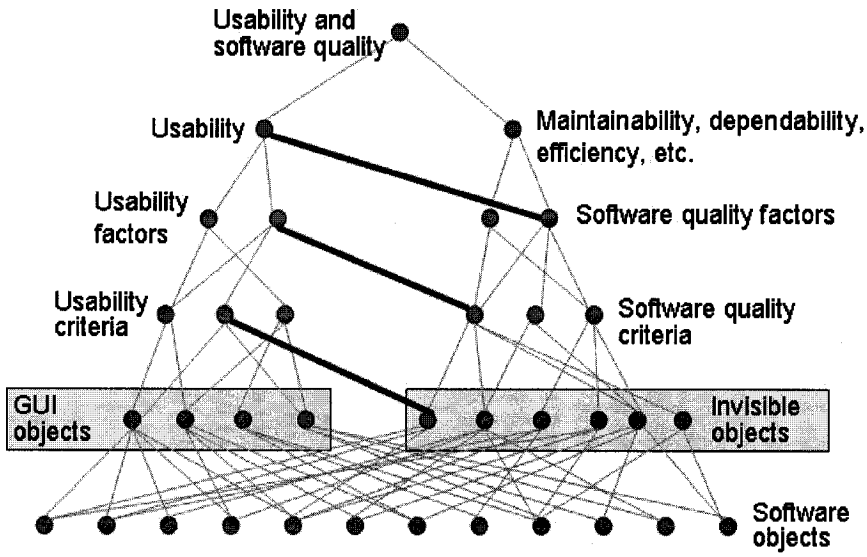


Figure 13: Most probable types of cross-relationships between usability and architecture (bold links)

3.8 Application

This model provides a framework within which to visualize and explore these exceptional ways that architecture can affect usability, so as to work toward a more complete model of usability. The model is useful because it helps us know where to look for, investigate, and experiment on relationships between architecture and usability. Further progress will require detailing the hierarchies on both sides of the tree, and considering each possible relationship between nodes at proximate levels. Another goal will be to provide other heuristic principles to further narrow down the likely interrelationships between these two branches.

Table 6 provides examples of the specific types of relationships that occur in the scenarios described in section 3.2. The second column refers to the invisible object's properties and software qualities identified in the right-hand branch of

figures 11 through 13, and the third column represents the usability properties identified in the left-hand branch of those figures.

For example, scenario 1 can be modeled as a relationship that connects the performance of the software feature with certain usability attributes such as user satisfaction. It can lead to the following requirement related to scenario 1: "To ensure an 80% level of satisfaction, the maximum acceptable response time of all the underlying related feature should not exceed 10 seconds; if not the user should be informed and a continuous feedback needs to be provided".

Table 6: Relationships between invisible software entities and usability factors

Scenario	**Quality attributes of invisible components**	**Usability factors affected**
1.	Performance	User satisfaction
2.	Integrity	Visual Consistency
3.	Functionality	Understandability
4.	Suitability	Operability
5.	Recoverability	Attractiveness

3.9 Conclusion

In this chapter, we first identified specific scenarios of how invisible software components can have an effect on the usability of the interactive system. Then, we provided a list of patterns that solved the problems described in the scenarios. This research effort can benefit software architecture designers and developers, who can use our approach in two different ways. First, the scenarios can serve as a checklist to determine whether important usability features (external attributes) have been considered in the design of the features and the related UI components. Secondly, the patterns can help the designer incorporate some of the usability concerns in the design.

More than defining a list of scenarios and patterns that describe the effects of invisible software attributes on software usability, the long-term objective is to build and validate a comprehensive framework for identifying scenarios. The goal of the framework is to define these patterns as a relationship between software quality factors and usability factors. In this chapter we have suggested different HCI and software design patterns as solutions to the problems described in these scenarios and in similar ones. Every pattern has a set of problems to be solved and a set of goals to be achieved.

As designers gain a better understanding of the relationship between interaction design patterns and software architecture patterns, this knowledge will affect the evolution of standards in architecture design and GUI software libraries. Some developers are making proper use of standard GUI libraries and respecting interface design guidelines in a way that considerably increases the usability of interactive applications. However, more can be done in this direction, and the approach we have outlined in this chapter is an attempt to build a better and more systematic understanding of how usability and software architecture can be integrated.

Chapter 4

Patterns in Model-Based User Interface Engineering

Abstract: In the previous chapter we showed the intricate relationship between architecture and usability and how patterns can support this "invisible" yet inseparable interplay between them. In this chapter we summarize another experiment to evaluate pattern reuse and provide some proposals on how patterns can supplement model based approaches in a comprehensive way. The main idea behind model-based approaches for User Interfaces (UI's) is to identify useful abstractions that highlight the core aspects associated with the design of an interactive system. However, certain limitations prevent mainstream developers from adopting model-based approaches for UI engineering. One such limitation is the lack of reusability of design practices. To foster reuse in various contexts of use, we introduce patterns as abstract UI building blocks which can be used to construct different models and then instantiated into concrete UI artefacts.

4.1 Introduction

The model-based approach was introduced to support the specification and design of interactive systems at a semantic, conceptual and abstract levels, as an alternative to dealing with low-level implementation issues earlier on in the development lifecycle [Pat00]. This way, designers can concentrate on important conceptual properties instead of being distracted by the technical and

implementation details. As a result, the increasing complexity of the user interface is more easily managed. Moreover, the UI architecture is simplified, hence allowing for better system comprehension and traceability for future maintenance in different context of use.

Unfortunately, model-based methods are rarely used in practice [Tra04]. One major reason for this limitation is that creating several models, instantiating and linking them together and then using them to generate more elaborate lower level models is a tedious and very time-consuming work, especially when most of the associated activities have to be done manually; tools provide only marginal support. This presents an overhead that is unacceptable in many industrial setups with limited resources, tough competition and short time-to-market. This crippling overhead can be partially attributed to the fact that model-based methods (for example TADEUS [TAD98], MOBI-D [MOB99] and TERESA [TER04]) lack the flexibility of reusing knowledge in building and transforming models. At best, only few approaches offer a form of copy-and-paste reuse. Moreover, many of these reuses involve the "reuser" merely taking a copy of a model component and manually changing it according to the new requirements. No form of consistency with the original solution is maintained [MLS98]. Copy-and-paste analysis and the concept of fragmented design models are clearly inadequate when attempting to integrate reuse in a systematic and retraceable way in the model-based UI development life cycle.

This practical observation motivates the need for a more disciplined form of reuse. We will demonstrate that the reusability problems associated with current model-based approaches can be overcome through patterns. They are usually presented as a vehicle to capture the best practices and facilitate their dissemination. Because patterns are context-sensitive, the solution encapsulated in a pattern can be customized and instantiated to the current context of use before being reused [AIS77]. Nevertheless, in order to be an effective

knowledge-capturing tool in model-based approaches, the following issues require investigation:

- A classification of patterns according to models must be established. Such a classification would distinguish between patterns that are building blocks for models and patterns that drive the transformation of models, as well as create a concrete UI.
- Providing tool support to assist developers when selecting the proper patterns and help in instantiating them once selected, as well as when combining them to create a model.

These two aspects are the essence of this chapter. After a brief overview of existing model-based approaches, we will introduce how we have been combining model-based approaches and several patterns to build a framework for the development of user interfaces. A clear definition of the various models used and an outline of the UI derivation process are also given. Furthermore, we will suggest how to enhance this framework using patterns as a reuse vehicle. We will demonstrate how HCI patterns can be used as building blocks when constructing and transforming the various models, and list which kind of HCI patterns lend themselves to this use. We also demonstrate the potential of supporting pattern combinations within the design process. A brief case study is presented in order to validate and illustrate the applicability of our approach.

4.2 Background Work

In this section we provide an analytical review of the work on modeling and show how it is proposed as a support to the design of user interfaces. We will also highlight some identified shortcomings.

4.2.1 Model-Based User Interface Engineering

Model-based UI development has been investigated for more than a decade. In most approaches, model-based UI design is defined as the process of creating and refining models [Sil00], [VLF03]. Many models exist in order to describe the user interface at different levels including task, user, presentation, dialog, and platform models. Until now, no consensus has been reached as to which models are the best for describing UI's [Sil03] and which model can be instantiated and transformed at each step to create a concrete user interface. Moreover, it is to be noted that instead of automation (JANUS, [Bal96], AME [Mar96], most model-based approaches (MOBI-D [MOB99], TERESA [TER04] provide little support in helping developers to interactively define the mappings between various models in the design. Consequently, the mapping between models is done outside the semantics of these models, and mostly manually. This is a tedious and error prone practice that most designers prefer to avoid. We often end up with limited local modeling or no modeling at all.

Constructing and transforming models is a very time consuming task even for small size software. For larger systems, it becomes prohibitively expensive. It requires different skills and knowledge about the various models. Unfortunately, current approaches for model-based tools offer very limited -if any- support for reusing model fragments or templates. Rine [RN00] defined reuse as the use of existing software artefacts or knowledge to create new software components. Within the scope of model-based UI development, reuse is particularly critical. For example, developing the task model for a medium to large-size system that is highly interactive and providing context-dependent UI's is quite difficult, as shown in Paquette [PS04]. Based on a case study, Paquette also observed that task modeling becomes tedious when a task model is specified with all its necessary details. However, these details are often crucial if task models are to be used as a starting point for the evolution of the final UI and for preliminary evaluations.

Furthermore, in order to incorporate a great level of details, the various models may grow too large to be easily understood or fully comprehended. The MOBI-D tool suite does not offer any functionality that allows importing model or design fragments. Similarly, TERESA only supports cutting and pasting static task fragments. Important tasks in knowledge reuse such as resolving name or relationship conflicts are not supported either. In essence, starting a new project means reinventing the wheel by building and linking all models from scratch. Constraints have to be manually applied and reworked between models, leading sometimes to inconsistencies that are only discovered in later design stages, implementation, or testing. Almost no design reuse from previous works, in terms of model fragments, is possible. Multiple views at various levels of abstraction and granularity should be provided.

4.2.2 Patterns in Model Construction and Transformation

As will be detailed here, UI design patterns have the potential to provide a solution to the reuse problem while acting as driving artefacts in the development and transformation of models.

Similar to the rest of the software engineering community, the Human Computer Interaction (HCI) design community has been a forum for vigorous discussion on pattern languages for user interface design and usability engineering as shown in Chapter 1. The goals of HCI patterns are to share successful HCI design solutions among HCI professionals and to provide a common ground for anyone involved in the design, development, evaluation or use of interactive systems [Bor01]. Several HCI practitioners and interactive application designers have become interested in formulating HCI patterns and pattern languages. A number of concrete pattern languages for interaction design have been suggested, as detailed in Chapter 2.

However, in order to facilitate their reuse and applicability, patterns should be presented within a comprehensive framework that supports a structured design process, not just according to the structure of each individual aspect of the application (e.g., page layout, navigation, etc.), which is currently the case for most HCI patterns. This demonstrates the virtue of using model-based design approach in comparison to manual design practices. UPADE (Gaffar and Seffah [GS05]) is one attempt to seemingly integrate patterns in the design process. Based on the UPADE tool, the approach aims to demonstrate when a pattern is applicable during the design process, how it can be used, as well as how and why it can or cannot be combined with other related patterns. Developers can exploit pattern relationships and the underlying best practices to devise concrete and effective design solutions. Details of UPADE are provided at the end of the chapter

Similarly, the "Pattern Supported Approach" (PSA), Granlund and Lafrenier [GL99] address patterns not only during the design phase, but throughout the entire software development process. In particular, patterns have been used to describe business domains, processes and tasks to aid in early system definition and conceptual design. In PSA, HCI patterns can be documented according to the development lifecycle. In other words, during system definition and task analysis, it can be determined which HCI patterns are appropriate for the design phase, depending on the context of use. However, the concept of linking patterns together to put up a design is not tackled.

In addition, Molina et al. [MMP02] found that existing pattern collections focus on design problems, and not on analysis problems. As a result, he proposes the Just-UI framework, which provides a set of conceptual patterns that can be used as building blocks to create UI specifications during analysis. In particular, conceptual patterns are abstract specifications of elemental UI requirements, such as: *how to search*, *how to order*, *what to see* and *what to do.* Molina also

recognized that the relatively-informal descriptions of patterns used today are not suitable for tool use, a major problem identified in Chapter 1. Within the Just-UI framework, a fixed set of patterns has been formalized so they can be processed by the "OliverNova" tool [CT04]. Eventually, the JUST-UI framework will use code generation to derive a UI implementation based on the analysis model.

Breedvelt [BPS97] discusses the idea of using task patterns to foster design knowledge reuse while task modeling. Task patterns encapsulate task templates for common design issues. This means that whenever designers realize that the issue with which they are contending is similar to an existing issue that has already been detected and resolved, they can immediately reuse the previously-developed solution (as captured by the task pattern). More specifically, task patterns are used as templates (or task building blocks) for designing an application's task model. According to Breedvelt, another advantage of using task patterns is that they facilitate reading and interpreting the task specification. Patterns can be employed as placeholders for common, repetitive task fragments. Instead of thinking in terms of tasks, one can think in terms of patterns at a more abstract level. Such an approach renders the task specification more compact and legible.

However, Breedvelt considers task patterns as individual static encapsulations of a (task-related) design issue in a particular context of use. Concepts for a more advanced form of reuse, including customization and combination, are not presented. Our approach [GMS05] and [GSP05] has highlighted an important aspect of the pattern concept: *pattern combination*. By combining different patterns, developers can utilize pattern relationships and combine them in order to produce an effective design solution. We will consider this principle in section 4.7 and suggest a tool for combining patterns. As a result, patterns become a more effective vehicle for reuse.

4.3 PD-MBUI: A Pattern-Driven Model-Based User Interface

In this section we provide a comprehensive view of different models used in the activities related to UI design. We organize them in a logical sequence to collectively lead to a complete UI design process.

4.3.1 Basic Concepts and Terminology

Pattern Driven Model-Based UI, PD-MBUI aims to provide a pattern-driven and model-based framework that consists of:

- A set of models including task, domain, user, environment, dialog, presentation and layout model. These models are defined in the next section.
- A method for constructing these models while creating a concrete UI
- A set of HCI patterns that can be used in this construction. Examples of patterns are discussed in 4.4.
- A tool called the *Pattern Wizard* that helps user interface developers in using patterns when constructing and transforming the various models to a concrete user interface.
- Another tool, UPADE demonstrates the potential of automating the combination of patterns

Within our approach, we will be using some notions as defined here:

- HCI patterns as a solution to a user problem that occurs in various contexts. An example of pattern is *Multi–Value Input Form* [Pat00]. It provides a solution to the typical user problem of entering a number of related values. These values can be of different data types, such as "date," "string" or "real".

- User interface component such as button, windows, dialog boxes are generally defined as object-oriented classes in UI toolkit such as "Java Swing".

- An artifact is an object that is essential in order for a task to be completed. The state of this artifact is usually changed during the course of task performance. In contrast to an artifact, a tool is merely an object that supports performing a task. Such a tool can be substituted without changing the task's intention [FDRS03].

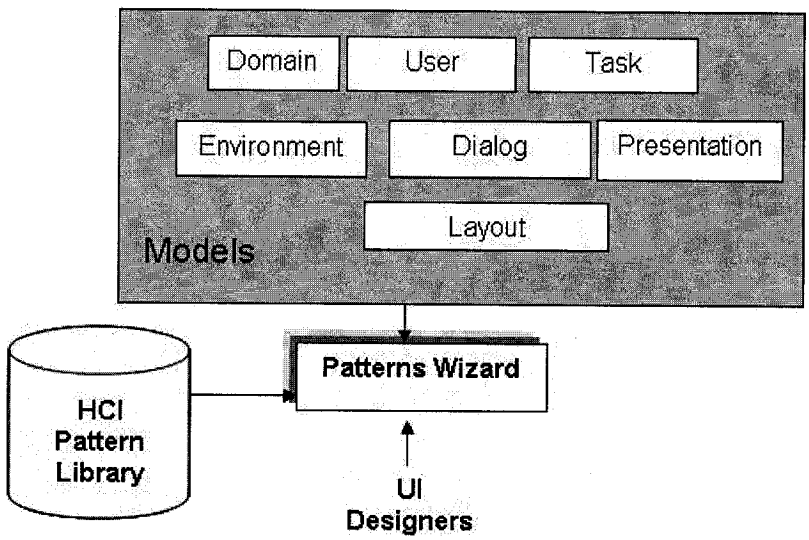


Figure 14: PD-MUI framework

4.3.2 PD-MBUI Models

Figure 14 depicts the models we considered within our framework. We have selected these models based on the fact that they have been largely cited in the scientific literature [Sch96], [Pue97], [Tra02]. However, we spent some effort to define them in such a way that they do not overleap and that the relationships

between them are also clearly stated. This is fundamental in order to know which types of patterns are needed and when they may apply.

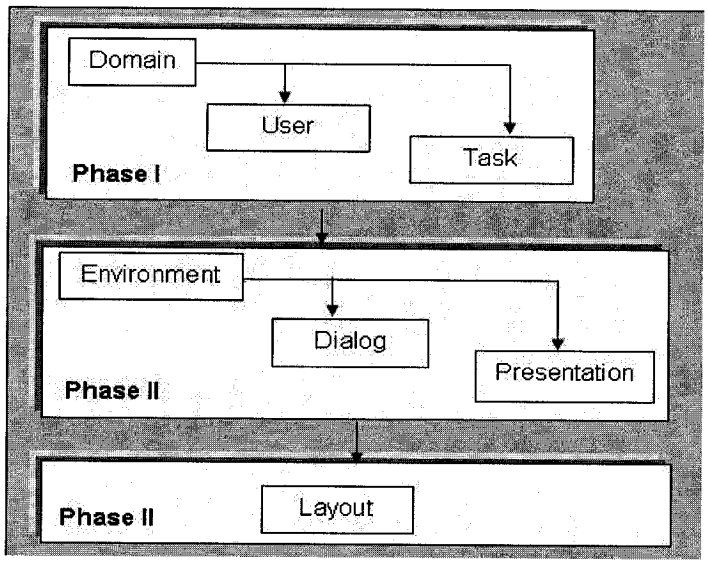


Figure 15: Models and their relationships in the PD-MBUI framework

The process of constructing these models distinguishes three major phases as depicted in Figure 15:

The starting point of phase I is the domain model. This model encapsulates the important entities of an application domain together with their attributes, methods and relationships [Sch96]. Within the scope of UI development, it defines the objects and functionalities accessed by the user via the interface. Such model is generally developed using the information collected during the business and functional requirements stage. Two other models are then derived from this model: user and task models.

The user model captures the essence of the user's static and dynamic characteristics. Modeling the user's background knowledge is useful when personalizing the format of the information (e.g. using an appropriate language that is understood by the user). The task model specifies what the user does or

wants to do, and why. It describes, at an abstract meaning, the tasks that users perform while using the application, as well as how the tasks are interrelated. In simple terms, it captures the user tasks and system behavior with respect to the task set. Beside natural language, notations such as GOMS and CTT are generally used to document task models. The task model is constructed in mutual relationship to the user model, representing the functional roles played by users when accomplishing tasks, as well as their individual perception of the tasks. The user model is also related to the domain since the user may require different views of the same data while performing a task. Moreover, a relationship must be formed between the domain model and the task model, because objects of the domain model may be needed in the form of artifacts and tools for task accomplishment.

The second stage starts with the development of the environment model. This model specifies the physical and organizational context of interaction [Tra02]. For example, in the case of a mobile user application, an environmental model would include variables such as the user's current location, the constraints and characteristics presented by this location, the current time and any trigger conditions specified or implied by virtue of the location type. This model also describes the various computer systems that may run a UI [Pra96]. The platform model describes the physical characteristics of the target platforms, such as the target devices' input and output capabilities. Based on this model and the models developed in the first stage, the dialog and the presentation model can be developed.

The dialog model specifies when the end user can invoke functions and interaction media, when the end user can select or specify inputs, and when the computer can query the end user and present information [Pur97]. In particular, this model specifies the user commands, interaction techniques, interface responses and command sequences permitted by the interface during user sessions. The presentation model describes the visual appearance of the user

interface [Sch96]. The presentation model exists at two levels of abstraction: the abstract and the concrete presentation model. The former provides an abstract view of a generic interface, which represents a corresponding task and dialog model.

In the last stage, the "Layout Model" is realized as a concrete instance of an interface. This model consists of a series of UI components that define the visual layout of UI and the detailed dialogs for a specific platform and context of use. There may be many concrete instances of layout models that can be derived from a presentation and dialog model.

4.4 Proposed HCI Patterns and their Description

As previously suggested in Figure 15, patterns are used to construct various models. The following are the major patterns we considered:

- Task and feature patterns are used to describe hierarchically structured task fragments. These fragments can then be used as task building blocks for gradually building the envisioned task model.
- Patterns for the dialog model are employed to help with grouping the tasks and to suggest sequences between dialog views.
- Presentation patterns are applied to map complex tasks to a predefined set of interaction elements that were identified in the presentation model.
- Layout patterns are utilized to establish certain styles or "floor plans" which are subsequently captured by the layout model.

The following table summarizes some of the patterns we considered.

Table 7: Pattern summary

Pattern Name	Type	Problem
Browse	Task	The user needs to inspect an information set and navigate a linear ordered list of objects such as images or search results.
Dialog	Task	The user must be informed about something that is requiring attention. The user must make a decision that will have an impact on further execution of the application, or the user must confirm the execution of an irreversible action.
Find	Task	The user needs to find any kind of information provided by the application.
Login	Task	The user needs to be authenticated in order to access protected data and/or to perform authorized operations.
Multi–Value Input Form [Pat00]	Task	The user needs to enter a number of related values. These values can be of different data types, such as "date," "string" or "real."
Print Object	Task	The user needs to view the details related to a particular information object
Search	Task	The user needs to extract a subset of data from a pool of information.
Wizard [Wel00]	Dialog	The user wants to achieve a single goal, but several consecutive decisions and actions must be carried out before the goal can be achieved.
Recursive Activation [Pat00]	Dialog	The user wants to activate and manipulate several instances of a dialog view.
Unambiguous Format	Presentation	The user needs to enter data, but may be unfamiliar with the structure of the information and/or its syntax.
Form	Presentation	The user must provide structured textual information to the application. The data to be provided is logically related.
House Style [Tid97]	Layout	Applications usually consist of several pages/windows. The user should have the impression that it all shares a consistent presentation and appears to belong together.

4.4.1 Pattern Instantiation and Application

In the previous section, we stated that patterns can be used as building blocks for different models throughout the UI development approach. For the

construction of models, the following process, which is part of the PD-MBUI, was proposed to instantiate and apply patterns:

1. **Identification**: A subset *M'* of the target model *M* is identified thus: $M' \subseteq M$. This relationship should reduce the domain size and help focus attention on a smaller, more pertinent subset for the next step.

2. **Selection**: An appropriate pattern *P* is selected to be applied to *M'*. By focusing on a subset of the domain, the designer can scan *M'* more effectively to identify potential areas that could be improved through patterns. This step is highly dependent on the experience and creativity of the designer.

3. **Adaptation**: A pattern is an abstraction that must be instantiated. Therefore, this step has the pattern *P* adjusted according to the context of use, resulting in the pattern instance *S*. In a top-down process, all variable parts are bound to specific values, which yield a concrete instance of the pattern.

4. **Integration**: The pattern instance *S* is integrated into *M'* by connecting it to the other elements in the domain. This may require replacing, updating or otherwise modifying the other objects to produce a seamless piece of design.

Variables are used as placeholders for the context of use. During the process of pattern adaptation, these placeholders are replaced by concrete values representing the particular consequences of the current context of use.

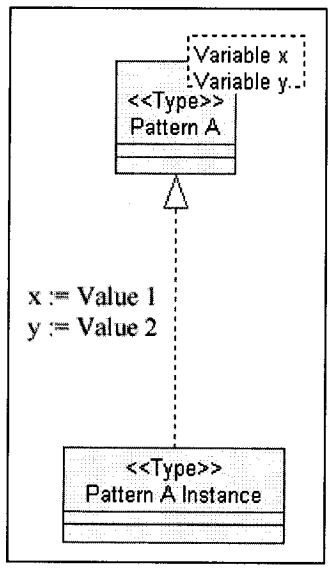


Figure 16: Interface of a pattern

Figure 16 shows the interface of pattern *A*. The UML notation for parametric classes is used to convey that the pattern assumes two parameters (variables *x* and *y)*. In order to instantiate the pattern, both variables must be assigned concrete values. In practical terms, the interface informs the user of the pattern that the values for variables *x* and *y* must be provided in order for the pattern to be used. In the figure, pattern *A* has been instantiated, resulting in Pattern *A Instance*. In addition, UML stereotypes are used to signal the particular type (role) of the pattern.

Patterns are often implemented using other patterns, *i.e.*, a pattern can be composed of several sub-patterns. This pattern-sub-pattern relationship, based on the concept of class aggregation, is presented in Figure 17. Pattern *A* consists of the sub-patterns *B* and *C*. If we place patterns in this kind of relationship, special attention must be given to the patterns' variables. A variable defined at the super-pattern level can affect the variables used by the sub-patterns. In Figure 17, variable *x* of pattern *A* affects the variables *yy* and *zz* of sub-patterns *B* and *C*. During the process of pattern adaptation, variables *yy* and *zz* will be bound to the value of *x*. As such, we observe how modifying a high-level pattern can affect all sub-patterns.

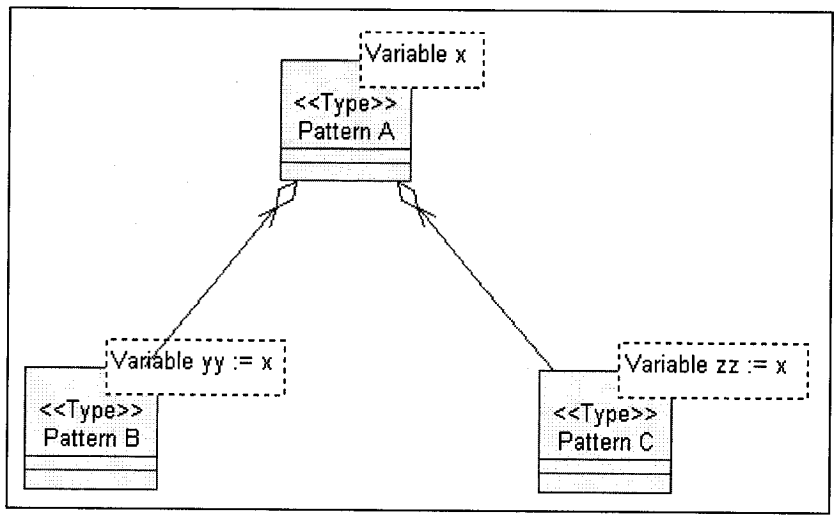


Figure 17: Pattern aggregation

4.5 Constructing Models Using Patterns

In previous sections, we showed how patterns can be applied to models and how they can be aggregated. This section provides an in-depth discussion of how different categories of patterns can be used together when constructing the task, dialog, presentation and layout model.

4.5.1 Patterns in Task Modeling

Patterns for the task model describe generic reusable task fragments that can serve to establish the task model. In particular, instances of task patterns (*i.e.*, already customized patterns) can be used as building blocks for the task model. Examples of such patterns for the task model include: *Find* something, *Buy* something, *Search for* something, *Login* to the system or *Fill out* an input form.

A typical example of a task pattern is *Search* [GSSF04]. The pattern is suitable for interactive applications that manage considerable amounts of user-accessible data. The user wants to have fast access to a subset of this data.

As a common and recurring solution, the pattern suggests giving users the possibility to enter search queries. On the basis of these queries, a subset of the searchable data (*i.e.*, the result set) is calculated and displayed to the user. The *Multi-Value Input Pattern* [Pat00] may be used for the query input. After submission, the results of the search are presented to the user and then they can either be browsed (*Browse Pattern* [Sin04] or used as input for refining the search.

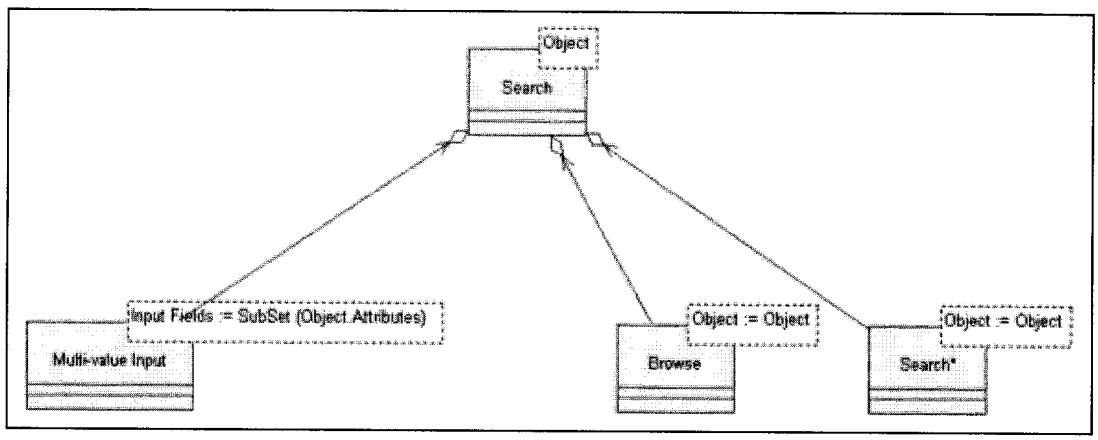


Figure 18: Interface and composition of the *Search Pattern*

As mentioned before, a pattern can be composed of several sub-patterns. Appendix A elaborates more on this type of relationship. Figure 18 illustrates how the *Search* pattern is composed of the sub-patterns *Multi-Value Input* and *Browse*, as well as of recursive references to itself (*Search**). It also demonstrates how the variables of each pattern are interrelated. The value of the "Object" variable of the *Search Pattern* will be used to assign the "Object" variable of the *Browse* and Sub-*Search Patterns*. In addition, a subset of the "Search" object attributes is used to determine the various "Input Fields" of the

Multi-Value Input Pattern, which is in turn responsible for capturing the search query. During the adaptation process, variables of each pattern must be resolved in a top-down fashion and replaced by concrete values.

The suggested task structure of the *Search Pattern* is illustrated in Figure 19. In order to apply and integrate the task structure, the pattern and all its sub-patterns must be instantiated and customized to the current context of use.

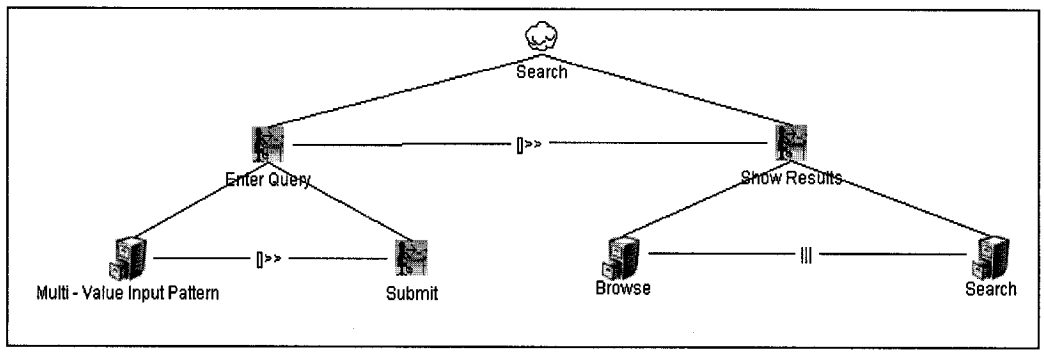


Figure 19: Structure of the *Search Pattern*

The top-down process of pattern adaptation can be greatly assisted by tools such as wizards. A wizard moves through the task pattern tree and prompts the user for information whenever it encounters an unresolved variable. Later in this chapter, we introduce the Task Pattern Wizard, a tool that assists the user in selecting, adapting and integrating task and feature patterns.

4.5.2 Patterns in Dialog Modeling

Our framework's dialog model is defined by a so-called dialog graph. Formally speaking, the dialog graph consists of a set of vertices (dialog views) and edges (dialog transitions). Creating the dialog graph is a two–step process: first, related tasks are grouped together into dialog views. Second, transitions from one dialog to another, as well as trigger events are defined.

In order to foster establishing the dialog model, we believe that patterns can help with both grouping tasks to dialog views and establishing the transition between the various dialog views.

A typical dialog pattern is the *Recursive Activation Pattern* [BPS97]. This pattern is used when the user wishes to activate and manipulate several instances of a dialog view. In practical terms, it suggests a dialog structure where, starting from a source dialog, a specific creator task can be used to instantiate a copy of the target dialog view. The pattern is applicable in many modern interfaces where several dialog views of the same type and functionality are concurrently accessible. A typical example of an application scenario is an e-mail program that supports composing several e-mails concurrently during a given session.

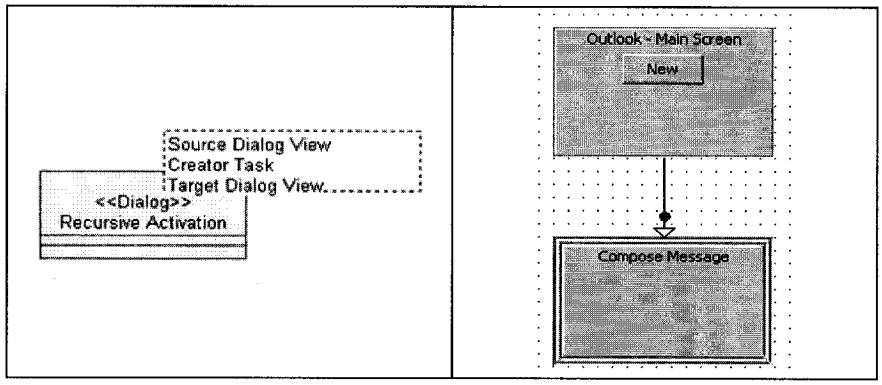


Figure 20: Presentation and one instantiation of *Recursive Activation Pattern*

In the left pane of Figure 20, we observe that in order to adapt (instantiate) the pattern, the source dialog view and the corresponding creator task, as well as the target dialog view must all be set. A specific instance of the pattern is shown in the right part of Figure 20, simulating the navigational structure of Microsoft Outlook when composing a new message. For this particular example, the visual notation of a tool called the Dialog Graph Editor [FDRS03] was used.

4.5.3 Patterns in Presentation Modeling

The abstract portrayal of user interface is determined in the presentation model through a defined set of abstract UI elements. Examples of such elements are Buttons, Lists or more complex aggregated elements such as trees or forms. Note that all interaction elements should be described in an abstract manner without reference to any particular interface components or environment. Likewise, style attributes such as size, font and color remain unset, pending definition by the layout model. Abstraction is a key to the success of presentation model as it frees the designers from unneeded details and allows for more efficient reuse on different platforms.

Patterns for the presentation model can be applied when describing the abstract UI elements. However, they can be more effective when applied for defining and mapping complex tasks (such as advanced search) to a predefined set of interaction elements. In this many-to-many interaction, patterns can provide insight into proven solutions ready to reuse.

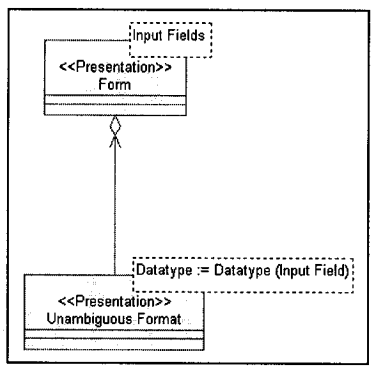


Figure 21: Interface of the *Form Pattern*

One illustrative example of a presentation pattern is the *Form Pattern*. It is applicable when the user must provide the application with structural, logically-related information. In Figure 21, the interface of the *Form Pattern* is presented,

indicating that the various Input Fields to be displayed are expected as parameters. It is also shown that the *Unambiguous Format Pattern* can be employed in order to implement the *Form Pattern*.

In particular, the *Unambiguous Format Pattern* is used to prevent the user from entering syntactically-incorrect data. In conjunction with the *Form Pattern*, it determines which interaction elements will be displayed by the input form. XUL code will be produced for the most suitable interaction element, depending on the data type of the desired input as shown in Figure 22. We elaborate on XUL and "Velocity" tool later in section 4.6, Tool Support.

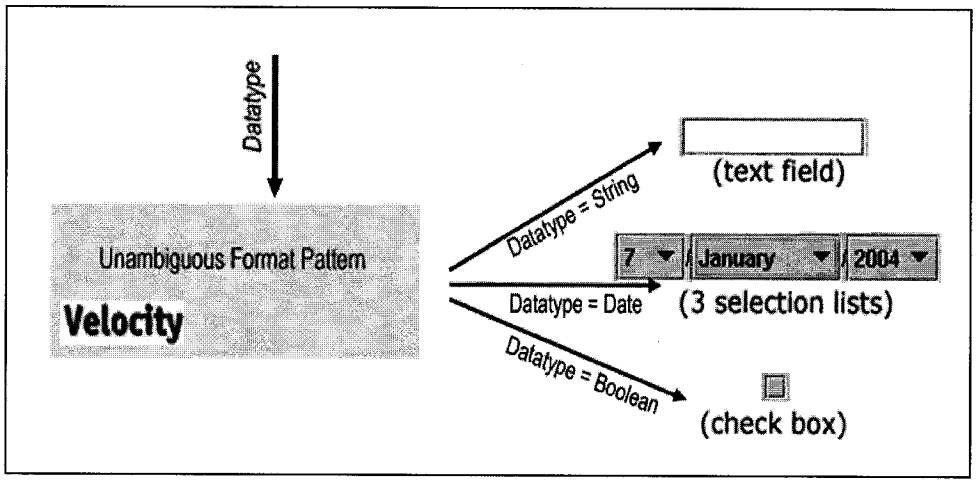


Figure 22: The rendered interaction elements with three different instances

4.5.4 Patterns in Layout Management Modeling

In Layout Modeling, the abstract UI elements of the presentation models are physically positioned following an overall layout or floor plan, which yields the layout model. Furthermore, the visual appearance of each interaction element is specified by setting fonts, colors and dimensions.

There are two different ways in which patterns can be employed when defining the layout model to reuse successful designs: (1) by providing a floor plan for the UI and (2) by setting the style attributes of the various widgets of the UI. The proposed solutions and the criteria of selecting between different designs depend –among other factors- on the context of use, nature of the application and satisfaction of users. Aesthetic and human behavior aspects can complicate the design and make the final results unpredictable. Therefore, patterns come in handy as shortcuts to analyzing some of these considerations by offering solutions that have been used before with good results.

The layout planning consists of determining the composition of the UI by providing a floor plan. Examples of such patterns are the *Portal Pattern* [Wel00], *Card Stack* [Tid97], *Liquid Layout* [Tid97] and *Grid Layout* [Wel00]. Figure 23 presents the floor plan suggested by the *Portal Pattern*, which is applicable for web-based UIs.

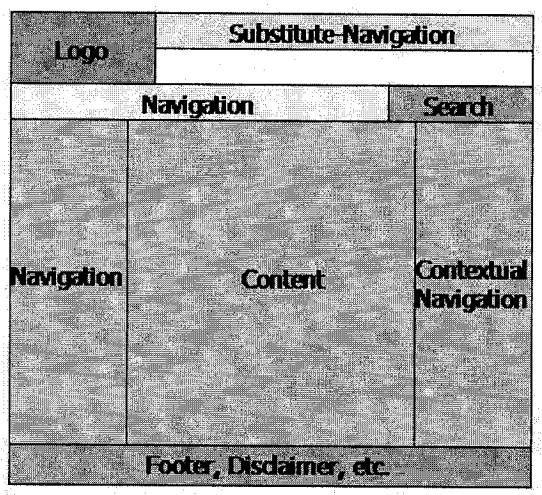


Figure 23: Floor plan suggested by the *Portal Pattern* [Wel00]

In the style planning, *Layout Patterns* are beneficial when the style attributes of the various widgets of the UI must be configured. For instance, the *House Style Pattern* suggests maintaining an overall look-and-feel for each page or dialog in

order to mediate the impression that all pages share a consistent presentation and appear to belong together.

Figure 24 collectively portrays the UI facets as they are modeled and reflects the relevance between these models and the final result in form of one ore more user interfaces. Task wizard helps improve the modeling process by receiving XUL code and annotating it with patterns then returning another –improved- XUL code and automatically rendering the interfaces.

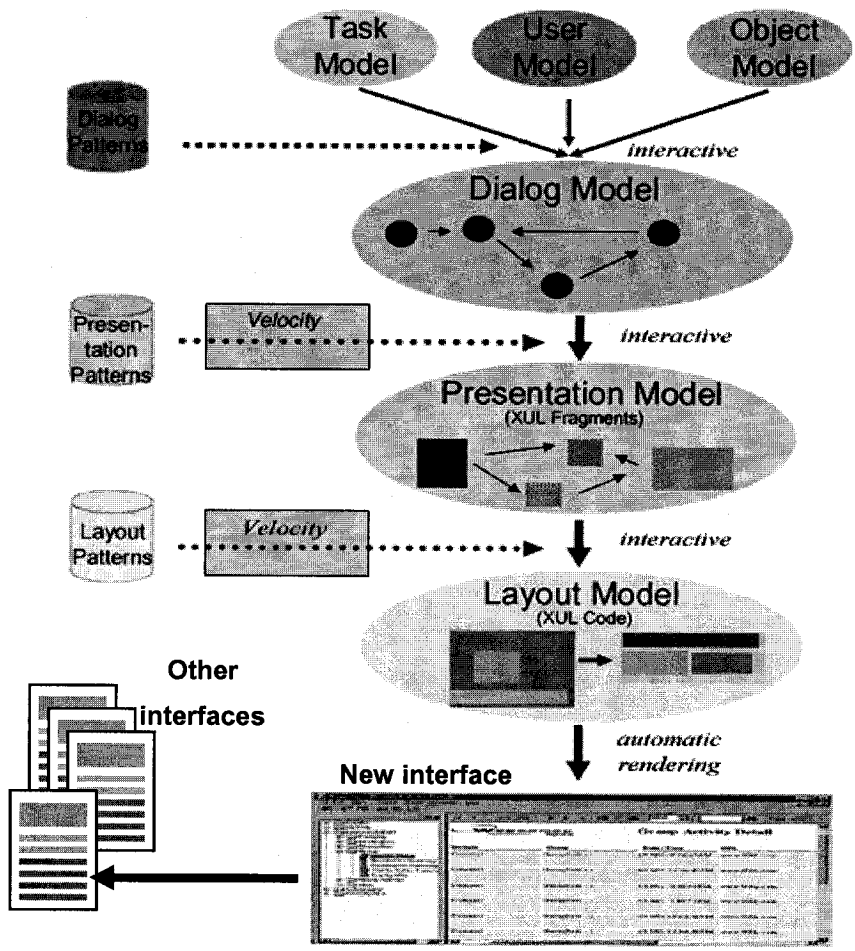


Figure 24: A comprehensive view of a model based process

4.5.5 Patterns in Interface Architectural Modeling

Arranging the interfaces of a large system to collaborate together in a seamless architecture and to allow for easy navigation and interaction with the whole system can be a complex endeavor. In Figure 24 we showed the multiplicity of models needed to move from task and user models into a final layout of an interface with layout models. However, the resultant user interfaces related to a cluster of tasks don't live in isolation. They work together as pieces of a larger puzzle representing the complete interface model of the interactive application. While this complete model is not easy to put down in one single figure or document, architectural modeling of the overall interface arrangement helps complete the design and understanding of this picture. Figure 25 gives a view of a composite interface architecture which is made of different clusters of architectural patterns. According to the navigational and interaction needs of each cluster of the system interfaces, several arrangements can be used in interface architecture like *Grid Pattern* and *Sequence Pattern*. While users do not necessarily see the whole picture of interface architecture, they can feel the effect of good interface design by being able to easily navigate and interact with the system. Nonetheless, additional patterns like '*Site Map Pattern*" are intended to provide the user with a visual summary of the overall interface architecture to help them in navigation. In systems with extremely large number of interfaces (for example large Websites like IBM or Yahoo and most commercial software applications) the reliance on navigating the overall architecture is often not enough to help users access all parts of the system. Therefore additional navigational help is added. An example is adding a local "*Search Pattern*" to help user go directly to relevant interfaces of the Website, or adding "*Help Pattern*" to help users find desired functionality and their relevant interfaces as well as guidance in using them.

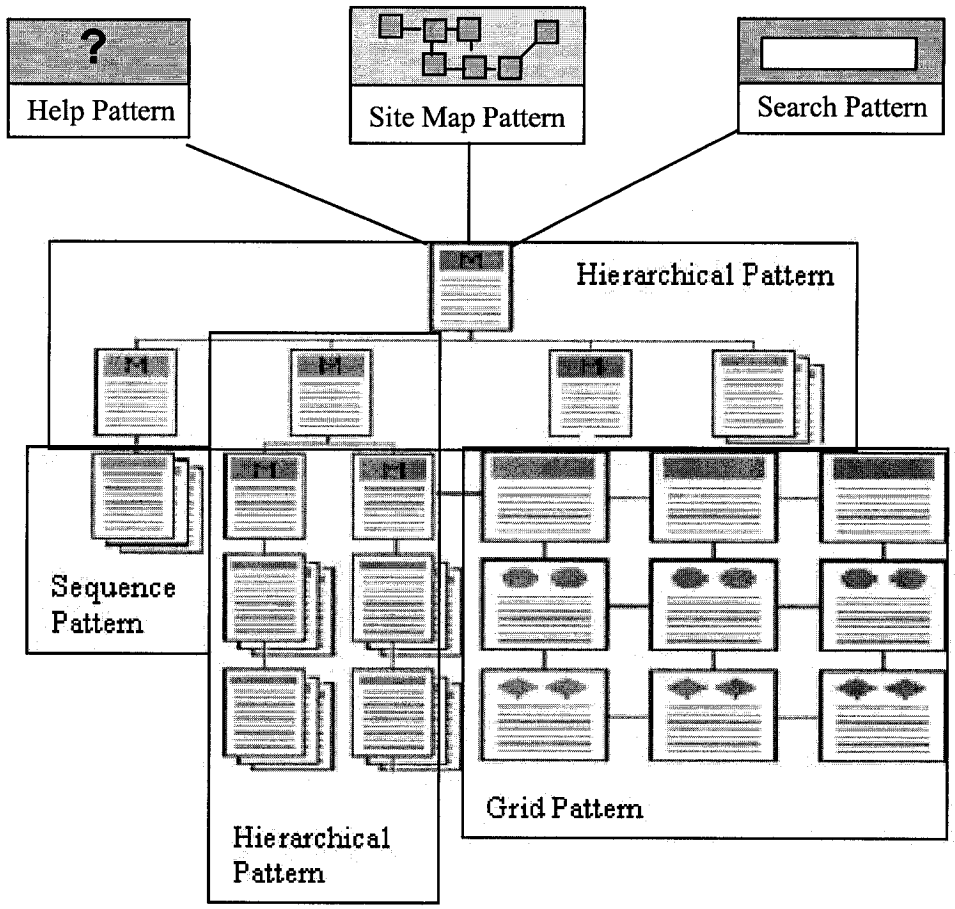


Figure 25: A Composite pattern for a complete application.

4.6 Tool Support

Efficiently choosing and applying patterns requires tool support. By integrating the concept of patterns in development tools, patterns can be a driving force throughout the entire UI development process. In response, we present a prototype of a task pattern wizard that is designed to support all phases of pattern application for the task model, ranging from pattern selection to adaptation to integration. After parsing the pattern, the tool guides the user step by step through the pattern adaptation and integration process. In what follows, we provide a brief description of how the Task Pattern Wizard is used. Particular

attention is given to outlining how the tool supports each phase of pattern application.

The generic user interface description language, XUL, has been selected as the medium by which the presentation model and layout model are described. XUL provides a clear separation between the high level user interface definition (in terms of abstract widgets that comprise the UI) and its visual appearance (the final rendering in terms of layout and *look-and-feel*[6]). XUL is particularly well suited to our approach because we also distinguish between the abstract definition of the UI (presentation model) and the actual visual appearance (layout model). Since the presentation model is written in terms of XUL code fragments, patterns for the presentation model are also used to generate XUL code as an output. Each presentation pattern has been formulated as a XUL Velocity template. Velocity is a Java-based template engine. The focus of the Velocity Template Language is to describe the generation of any form of mark up code such as XUL.

Technically speaking, the various XUL fragments of the presentation model are merged in the layout model, resulting in aggregated XUL code. The loosely-connected XUL pieces are nested and associated together. The way these fragments are merged depends on the overall layout of the application. In addition, any style attributes that are not set are bound with concrete values. Because Velocity templates can be used to generate XUL fragments (for the presentation model), they can also serve when aggregating XUL code. Consequently, layout patterns are also formalized as Velocity XUL templates. There are four main steps in applying patterns here:

---

[6] The layout refers to the arrangement of the user interface real estate. The "look" refers to the rendering of each component; its shape and color, while the "feel" refers to how it behaves when the user interact with it. The "look-and-feel" is often used by java to emphasize how same interfaces look and behave differently on different platforms

Identification:

In the identification phase, we identify a node of an existing task structure to which a pattern will be applied. After opening the XUL file, the Task Pattern Wizard uses a tree element to display the task structure. Next, the user chooses a node representing a particular task. The user is then prompted to decide how the new task structure, brought in by the pattern, should be integrated into the existing tree. Some characteristics of the new task structure can be integrated as being optional and/or iterative. Optional tasks can be executed, but are not mandatory. Iterative tasks can be repeated more than once. These two characteristics are orthogonal to each other and any combination of them is logically correct, but the user should decide on their correctness according to the nature of each task and the context of use. In addition, the new task's temporal relationship with the other tasks can be defined. Figure 26 is a screenshot showing the identification of a target node from the task model.

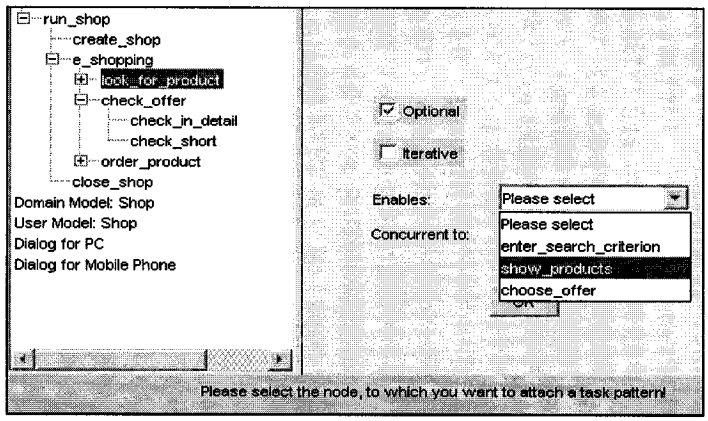


Figure 26: The *Task Pattern Wizard* selection screen

Selection:

In the selection phase, an applicable pattern is chosen. In order to perform the selection operation, the Task –Pattern Wizard presents the currently-opened pattern according to the Alexandrian form [AIS77]. The pattern is displayed in narrative form to suit human interaction at this stage. The user is presented with:

what issue will be solved, when the pattern can be applied (context), how it works (solution) and why it works (rationale) to help them in their design decisions.

Adaptation:

Once the appropriate pattern has been chosen, it must be adapted to the current context of use. Each pattern generally contains variables that act as placeholders for context conditions. Different kinds of variables exist, such as "substitution variables" and "process variables."

Substitution variables are simply used as placeholders for certain values (such as the task name). During the pattern adaptation process, the Task Pattern Wizard will prompt the user to enter values for these variables. Then, each occurrence of the substitution variable will be replaced (substituted) with this value in a top-down process.

Process variables are used to describe the structure of the task fragment that will be created by the pattern. For example, entering values in a form is very repetitive. The same basic task (*i.e.*, entering a value) appears over and over again. Each peer task can be distinguished simply by its name and input type. Thus, instead of describing each of these tasks individually, process variables that indicate the number of respective tasks can be used.

After all variables have been defined, a pattern instance that can be integrated into the task model is created.

Integration:

In the integration phase, the pattern instance is incorporated into the current task model. In short, a new branch is added to the task tree or an existing branch is replaced. The new modified task structure can then be saved in XUL format. Within our tool set, XUL serves as a universal exchange format. This approach

enables tools such as the XIML-Task-Simulator and the Dialog Graph Editor [FDRS03] to further process the new task model.

4.7 Pattern Relationships

Many tentative solutions for pattern reuse have been proposed. Most of them projected some relationships between patterns and then exploited them in specific design frameworks. In their book "Design Patterns", Gamma et al. [GHJV95] underline the importance of relationships between patterns as an essential part in understanding and using them. Zimmer [Zim95] implements this idea by defining different types of relationships between the patterns of the Gamma catalog. We provide more details of pattern relationships in appendices A and B. Van Duyne et al. compiled their patterns [DLH03] addressing many aspects of web page design and emphasizing relationships between all 90 patterns. The patterns are arranged in 12 groups that exist at different levels of hierarchy. The highest level is called "Site Genres," which provides a convenient starting point into the language and enables the user to choose the type of site to be created. Starting from a particular site genre pattern, various lower patterns are subsequently referenced. In this way, the authors have succeeded not only in providing a starting point into their pattern language, but also in demonstrating how patterns of different levels interact with each other. The organization of van Welie's Interaction Design Pattern language [Wel00] is comparable to that of van Duyne [DLH03]. Van Welie also views web design as a top-down activity. The user enters the language by selecting a "Posture Pattern". According to Welie, posture patterns describe the site's overall type or genre. The user can then follow the references that lead to lower-level patterns, which have been placed at the so-called "Experience," "Task" and "Action" levels.

In order to facilitate their reuse and applicability within general design processes, patterns need to be presented within a comprehensive framework that supports patterns from different collections, independent of the structure and classification

of each pattern language, which is currently the case for most HCI patterns. This demonstrates the virtue of using model-based design approach in comparison to manual design practices. Patterns from different collections can be used together in one design and can be integrated together if they have a common high level model that guide designers in their design process. This can greatly help automate the interoperability between them.

Despite its importance, pattern relationship information has been generally underestimated, and is often presented as a small addition to patterns; mostly as a simple pointer from one pattern to another. In [GSP05] we emphasize the importance of relationship information between patterns, and show that relationships can have more constituents than just directed arrows. Otherwise, important information can go unnoticed and are only clear to the authors. We state that "*this part of information is generally implicit, and only clear to experts, contributing to the difficulty of reusing patterns*" In our framework for pattern representation model, we clearly separate the pattern relationship part from intrinsic pattern data, and present the former as *extrinsic* information that should reside outside the internals of a pattern. Nevertheless, they are not physically separated in another file; they are still represented as part of the overall pattern data set, only in a logically different category. This allows for treating them as separate entity with their own model that connects to and interacts with the intrinsic pattern model. Following this principle, more information can be built within the extrinsic part of patterns, the relationship information.

In [GSSF04] we demonstrate how we built the model of *extrinsic information*. We represent extrinsic data as a binary relationship R_i from pattern P_x to pattern P_y in the form

$$(R_i, P_x, P_y)$$

Likewise, we represent ternary relationship as

$$(R_i, P_x, P_y P_z)$$

In the framework, we have not felt the need for using higher rank relationships. Nonetheless, higher ranking relationships can be represented following the same concept.

We give some examples of extrinsic binary relationships:

- (Subordinate, X, Y) if X can be embedded in Y. Y is also called superordinate of X
- (Equivalent X, Y) if X and Y can replace each other
- (Competitor X, Y) if X and Y cannot be used together
- (Neighbor X, Y) if X and Y belong to the same pattern category (family) or to the same design step as the described pattern

In [GSSF04] we emphasize that ternary relationships are generally more complex, and contain more subtle information that could be easily overseen by an inexperienced user. The concept of extrinsic data covers them equally consistently. To demonstrate, we provide the following relationships:

- (Combine, X, Y, Z) if X, Y, and Z can be combined together in a design artifact.
- (ExclusiveCombine, X, Y, Z) if X can be combined with either Y or Z, but not with both of them at the same time. The notation is read "*X exclusivecombine y and z*"

It is worth mentioning that unlike "Competitor, Y, Z", the "ExclusiveCombine, X, Y, Z" denotes that Y and Z are only competitors in the presence of X. Appendix A gives more insight into the pattern relationships. Appendix B shows how this approach can be extended to facilitate ontology of patterns based on our relationship model.

In [GMS05] we show how we used UPADE to expand this approach and put it into work to demonstrate its feasibility. UPADE is a pilot project to test the possibility of using pattern tools as design building blocks to highlight the

intricacy of pattern relationships and their mutual interaction. The feedback from this project -collected from actual users and lab testing- has been used in building more comprehensive pattern dissemination and assimilation environment for developers. In the next section we provide some details on UPADE.

4.7.1 UPADE Architecture and Basic Features

UPADE provides a unified interface to support the development of user interface designs and improve software production. It is a prototype written in Java that aims to support HCI pattern writers and user interface developers. By leveraging the portability and flexibility of Java, UPADE enables developers to easily and effectively describe, search, exchange and extend their own patterns as well as those created by others. At the same time, UPADE offers an environment to combine patterns, support their integration at high design level and automate their composition.

UPADE Architecture

As a tool for automating the development of UI designs, UPADE embodies several functionalities. It helps both pattern writers and developers to use existing relationship between patterns to define new patterns or create a design by combining existing patterns. Moreover, in order to facilitate pattern combinations, the tool supports different hierarchical, traceable design levels. In our case study associated with UPADE, three levels are possible: Pattern Level, Design Level, and Code Level. At the pattern level, developer can see description of patterns, search a specific pattern, create a new pattern and save it into the database. At the design level, developers can combine patterns, support the integration of patterns at different design stages, replace one pattern occurrence by another and validate the selected pattern compositions. Finally at code level, developers can see the structure of the design in terms of classes, methods, associations

and inheritance relationships in a particular programming language. Additionally, UPADE provides a mechanism to check and control how patterns are created or modified. By using the database information, UPADE automatically examines the patterns and offers a related feedback to the designer.

Feature Description

Each pattern describes a collection of elements that provide a solution to a problem in a given context. The main user interface includes the following components:

-"Browse" provides a description of existing patterns, some illustrated diagrams and several practical examples. In this mode, UPADE produces and delivers patterns information. The information is presented using the incorporated format showing related design processes, pattern category, name, description and examples. Categories are presented as a browse tree for navigation as shown in Figure 27. By default, UPADE allows browsing patterns with their associated process name. However, software developers can switch to browse by category.

-"Search" improves efficiency of using UPADE by accelerating the searching of patterns. The search window presented to the developer offers two kinds of search. Users can have a simple search for patterns by keywords. They can also select from several advanced search criteria in the 'Criteria Combo Box' and apply it.

-"Edit" helps developers create their own patterns or modify existing ones. Since patterns are reusable components, a well-developed pattern should be saved for reuse in other designs [FMW97]. UPADE allows developers to create new patterns and associate new implementation rules -or constraints- with them. The use of constraints allows developer to decide how certain patterns can be combined with each other in the design mode.

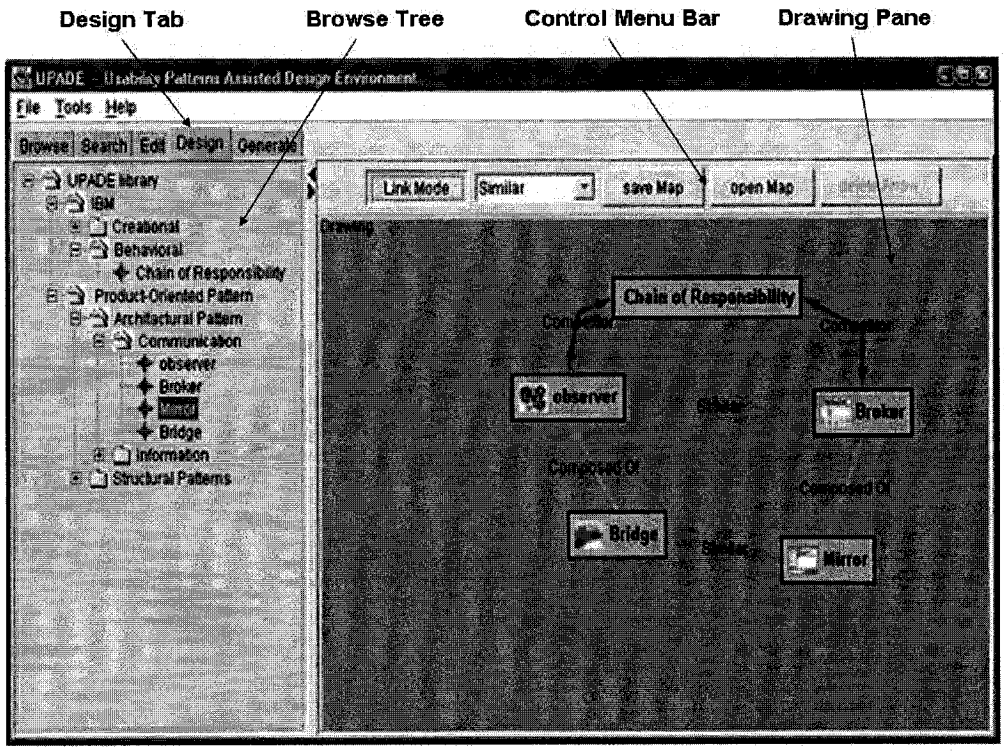


Figure 27: Navigating and selecting patterns

-"Design" develops structured steps to combine patterns, support integrating them at several phases of design, and verify their composition against the combinations and relationship constraints suggested by pattern authors. As shown before, besides describing a solution, a pattern also describes several possibilities of how it can relate to other patterns and how it can be composed of other patterns. The creativity of design is preserved by allowing designers the freedom to mix different patterns together. The process of freely mixing patterns together can be hard to verify in a complex network of patterns. Relationship constraints guide users into avoiding "invalid" or "less preferable" combinations and warn about unforeseen consequences. By offering verification of pattern relationships, UPADE helps designers in selecting more appropriate combinations during their design. The main interface of "Design" workspace is separated into three areas:

- **"Browse Tree Pane"** is the section where developers can browse the entire pattern collections in the **UPADE** database. They can expand any collection to see the available patterns in it.
- **"Control Menu Bar"** is the menu for developers to control the drawing process and create their custom design.
- **"Drawing Pane"** is the area where developers can compose patterns by simply using drag and droop from the browse tree pane.

Practicing Pattern-Oriented Design

In "Design" mode, UPADE can support combination and organization of existing patterns from more general to more specific details. For example, The Software developer can embed "*Page Managers*" patterns into "*Information Architectural*" patterns, and both "*Navigation Support*" patterns and "*Information Containers*" patterns into "*Information Architectural*" patterns. Moreover, the designer has the freedom to organize "*Navigation Support*" patterns and "*Information Containers*" patterns inside the layout; they can move, combine or delete them altogether. These activities aim to explore how to organize and combine existing patterns to customize and format the new ones. "Design" editor provides a mechanism to check the validity of combined patterns using the set of constraints associated with them. It examines the compatibility of certain patterns and gives the related instruction to the designer.

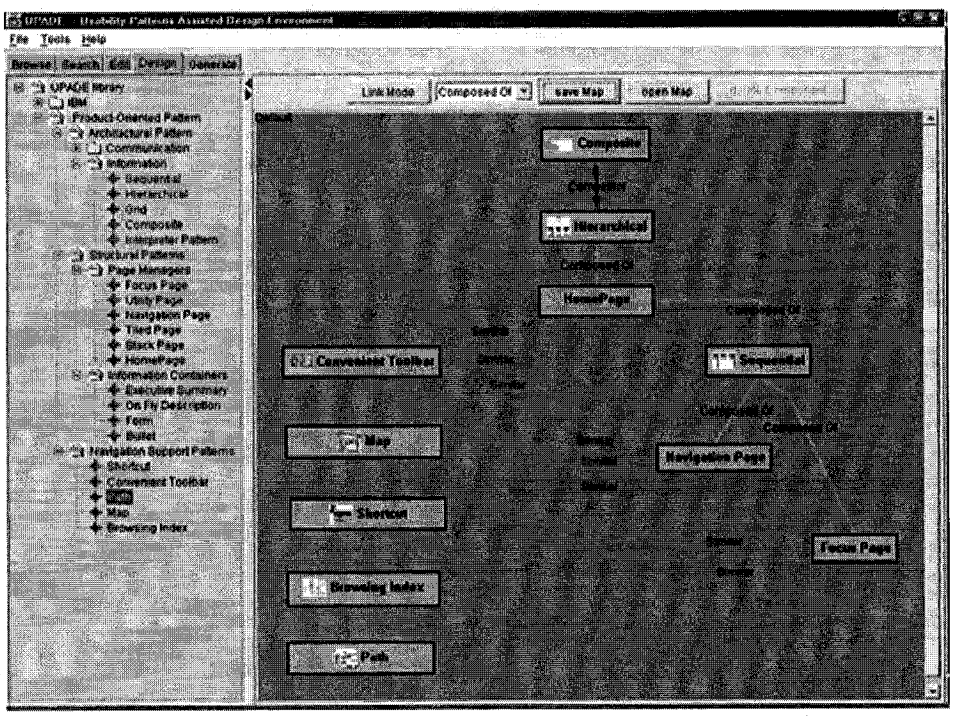


Figure 28: Combining patterns in a new design

Once the "Design" Tab is selected (Figure 28), UPADE can help start a new pattern-oriented design as follows:

- Pattern developers need to browse the tree in the "**Browse Tree Pane**" to view available patterns.
- Then they can select a pattern and drag and drop it into the **"Drawing Pane"** area.
- They repeat these two steps until all desired patterns are collected into the drawing area.
- Next, by selecting "**Link Mode**" button, developers are guided to connect each pattern to the others by choosing from different relationship types that are available in the combo box of the control menu. While developers can choose the way they want to connect patterns to generate their own design and the type of relationships, UPADE will check the validity of all connections selected by users and allow only valid ones. Users can override this

mechanism, but they are provided with the consequences of their selections to help them take informed decisions. At the end, the developer can save the new pattern composition map into XML format for use by other XML-compatible tools.

UPADE is designed to be customized and extended, with the realization in mind that some designers have achieved a local set of patterns and conventions for style and structure, and only need a tool to assist them in creating new design more quickly that honors those conventions. In our survey (appendix D) we have collected feedback from several participants to support this claim. One of the key features of UPADE is that developers have visual control with drag and drop mechanism.

4.8 Conclusion

In this chapter, we demonstrated how HCI patterns could be delivered and applied within model-based UI development approaches (MBUI). Within our proposed framework for Pattern Driven MBUI (PD-MBUI), patterns were introduced to overcome the lack of model reusability and construction, which represents one of the major limitations of existing model-based UI development frameworks. In particular, we illustrated how different kinds of patterns can be used as building blocks for the establishment of task, dialog, presentation and layout models. In order to foster applicability in different contexts of use, we defined the general process of pattern reuse, in which patterns are abstractions that must be instantiated. In addition, we described an approach for combining patterns together within a design process, UPADE.

The applicability of the proposed pattern-driven model-based development approach as well as combining them has been demonstrated through different tools and case studies to validate the idea. We demonstrate the potential of patterns to support model reuse in the construction of specific models and their

transformations. While this approach can be used to complement other modeling approaches and support reuse, it would be less effective in manual design processes where modeling approaches are not used.

Traditionally, patterns are encapsulations of a solution to a common problem. In our research, we extended the pattern concept by providing an interface for patterns in order to combine them. In this vein, we proposed the general process of pattern application, in which patterns can be customized for a given context of use. We then incorporated the pattern concept into the domain of model-based UI development. In order to foster reuse and avoid reinventing the wheel, we demonstrated how task, dialog, presentation and layout patterns can be used as reusable building blocks when creating the corresponding models, which are the core constituents of our development approach.

In order to demonstrate the applicability of our approach, we developed a UI prototype for a hotel management application. An illustration is given in appendix C. In this elaborate case study, patterns are identified and applied for each of the models that were used during development. The main purpose of the example is to show that model-based UI development consists of a series of model transformations, in which mappings from the abstract to the concrete models must be specified and –more importantly- automatically supported by tools.

In the last chapter, we will expand the modeling concept into an integrated pattern environment, IPE, which provides a framework to integrate this and other tools into a generalized pattern driven development environment. This environment supports different platform and programming languages. The transformation between models is automated by the use of XML as a common medium to communicate the modelling semantics between different models. This helps tailor the application and corresponding models to different platform and user roles.

Chapter 5

Towards an Integrated Pattern Environment

Abstract: In the literature analysis of Chapter 2 we evaluated the current state of pattern proposals within HCI community and how they are represented among pattern authors. In chapters three and four, we experimented on the applicability of HCI patterns in the design domain and provided two case studies of pattern reuse to enhance usability within the architecture and to enhance interface design. We also conducted a survey to evaluate the state of pattern reuse among mainstream programmers. Based on these investigations and studies, we propose a new approach for pattern engineering through an integrated pattern environment. The environment is realized using XML and supporting models to allow patterns to be used and manipulated within a programming environment while keeping its original information contents and text format.

5.1 Introduction

Even when we have an effective representation, turning static descriptions of patterns into useful, automated components that provide valuable resources and knowledge for developers requires a combination of methodology and tool support both for developing these new pattern artifacts and for using them.

Several pattern authors have valuable information and they spend efforts to write them down for users. As we have seen so far, the mainstream programmers

often have little time to spend looking for patterns in books and on the Web. The gap is well known and evident in the literature as we discussed in Chapter 1, and as seen in the survey (appendix D). The large number of patterns and the narrative formats add a cognitive load on pattern users and prevent them from finding and using patterns to their full extent (we refer to it as the scalability problems). Newer patterns also have little chance of making their way to the mainstream (we refer to it as the visibility problem) and they often end up lost on the Web. So far, we have identified the potential of patterns in different aspects of HCI domain. We showed the synergy between patterns and modeling and how they can work together as an enabling technology towards automation of pattern reuse.

In this chapter we go one step further by proposing a new approach to enhance pattern dissemination and reuse. This comes in the following steps:

-Define an information model for patterns that covers its main constituents

-Define the current pattern dissemination model and propose an extension to it to enhance the pattern lifecycle

-Define a process to implement the pattern lifecycle

-Define models to support the process and help transform text patterns into program components

-Provide an infrastructure to support pattern manipulation at runtime besides their narrative format.

The process covers human activities as well as theoretical and technical aspects that collectively work to move patterns from a narrative format towards a more programmable format. This format allows patterns to be machine readable; hence they can be processed by other software artifacts. In the meanwhile, the semantics of patterns are preserved which allows the system to accurately reproduce pattern in their original text format for human readers.

The infrastructure to support the process is an Integrated Pattern Environment, IPE, which implements this idea. We look at the semantics of patterns and how we can identify and formalize them in a generic model. This model is the core of the proposed pattern lifecycle. It allows interacting with other software at the code level resulting in machine readable patterns. The model needs to be well defined to preserve the correct semantics of patterns and in the meanwhile to allow for writing new software to interact with concrete interfaces provided by the model. Following this approach, we see patterns as program objects or software components as opposed to text documents. Instead of saving patterns as a large monolithic text document or set of tagged text documents, we can save them as objects that can be accesses directly by other software at runtime. In [GSP05] we show that the semantics of the pattern model allows for an efficient lookup and transformation of pattern components according to their internal information (intrinsic), their relationships to other patterns (extrinsic), or according to their applicability within specific stages of design phases (assimilation).

5.2 A Schema for Patterns

The concept of a schema is used in many domains to present a high level, common view of different artifacts, or low level details of them. In this section, we present a high level schema of patterns based on our investigations in the thesis so far.

5.2.1 Why a Schema?

To conveniently handle the entirety of available patterns within a pattern system it is helpful to define and separate different parts of each pattern according to the interest of the reader. A pattern classification schema that supports the development of software systems using patterns should have the following properties:

- **Simplicity**: It should be simple and easy to understand, learn and use. A complex schema would be hard to validate and will deter many users from using it.
- **Objectivity**: Each classification criterion should reflect functional properties of patterns, for example the kinds of problems the patterns address, the related design phase and the context of applicability rather than non-functional criteria such as the pattern author or whether patterns belong to a pattern language or not.
- **Guidance**: It should provide a 'roadmap' that leads users to a set of potentially applicable patterns, rather than a rigid 'drawer-like' schema that tries to support finding the one 'correct' pattern.
- **Generality**: Like programming languages, the schema should be independent of specific domain, platform or technology.
- **Extensibility**: The schema should be open to the integration of new patterns without the need for refactoring the existing classification.

5.2.2 The Proposed Schema

Based on our literature review of Chapter 2 and the feedback we obtained from the empirical usability study (appendix E) as well as the survey (appendix D), we suggest a schema as shown in Figure 29. This schema can be seen as relevant to the end-user experiences, the HCI/usability expert and the software and UI engineers.

- A pattern is a solution provided by an HCI/usability expert to a user problem, which can occur in different contexts of use.
- The forces, the consequences of the problems as well as the rationale for the solution have to be detailed qualitatively
- Usability/HCI engineers have to ground the patterns in the HCI theory and principles. UI Developers should also provide an implementation or strategies for implementing the patterns. Consequences should be linked to usability

measures that provide a more objective way to assess the pattern's applicability.

This abstract schema shows the major issues related to patterns from the perspective of different users. In it, each professional group is aware of its own concerns as well as the others'. Therefore, each group may be able to address the needs of the other groups while contributing towards patterns and pattern languages. In this view, we are incorporating some concepts from different pattern collections in order to reduce the communication gap between pattern writers and software developers and to provide the object-oriented outlook of patterns.

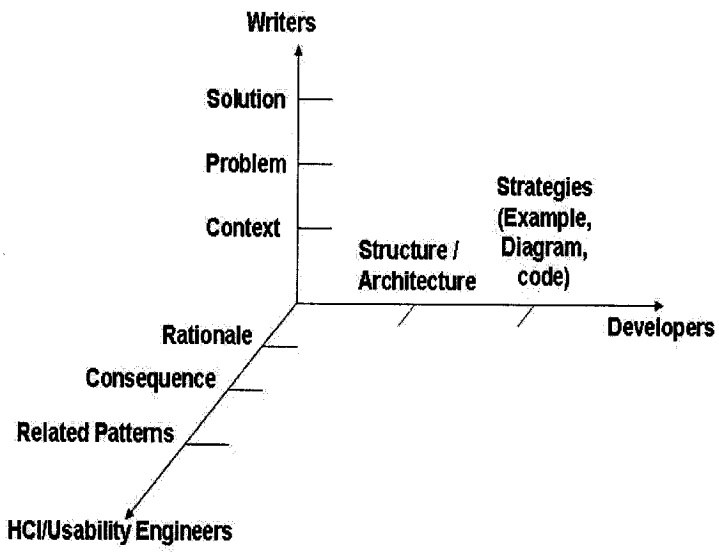


Figure 29: A 3D presentation of pattern properties

This 3D abstraction highlights the major ingredients for documenting a pattern. Table 8 provides a summary of the key components that can be used in different pattern templates. This template could serve as a basis for future expansions to serve different needs.

As discussed earlier, In [GSJS03] we called for an approach to generalizing the format of patterns to facilitate the creation of a pattern database. As the idea was discussed, we implemented the first attempt of this task together with major HCI pattern authors (John Vanderdonckt, Martijn van Welie, Jennifer Tidwell and Jan Borchers) and a research team from IBM. The resulting format, the Pattern Language Markup Language (PLML) was created in the workshop to mark the first step towards a common pattern format [FF03]. Since then, we went much further into improving the presentation from a common but static presentation (The PLML is a static format) into a dynamic runtime module. We then upgraded PLML into the Generic Pattern Model (GPM) [Gaf05b] that offers the flexibility of changing its components as needed. GPM is then used in a comprehensive software development as will be shown later in this chapter.

Table 8: The proposed format of pattern documentation

Element	**Sub-Elements**	**Requirements**
Identification	Name	
	Alias	
	Author(s)	
	Date	
	Category	Patterns Classification
	Keyword	For Search
	Related Pattern(s)	Super-ordinate
		Subordinate
		Sibling/Neighboring
		Competitors
Context of Use	User	Category of users, personas or profile, etc.
	Task	Tasks are structured hierarchically. All sub-tasks should be originated from a root.
	Platform Capabilities and Constraints	Information should be organized in device-independent way.
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. They are often 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 could be described by a list of Usability Factors/Criteria/Metrics.	

5.3 A Database for Patterns

A common activity during pattern delivery is to put patterns on-line through the Web or other Intranet facilities, a significant medium of communication. However, as shown in the empirical study (appendix E) and in Chapter 2, the outcome is much less than anticipated. We need more than just posting patterns online. Too often, good patterns are hidden in Web pages that become severely underused in the daily activities of interface designers. The Web page approach alone fails to provide the means to access appropriate patterns as needed and does little on the way of reusing them as an integral part of the development processes [Gaf04] and [Gaf05c]. Even if pattern authors are actively discovering and writing new patterns, it is difficult for users to keep pace with changes in the HCI community at large just by searching the Internet. Designers and developers often work under tight constraints and limited resources. Eventually, if software developers have to manually read, analyze and understand every pattern in details to select the ones they need, the pattern system becomes unmanageable, even when it included useful patterns.

Together with tool support shown in Chapter 4, a database of patterns can become a valuable resource that software developers and project managers depend on for efficient information retrieval and reuse. Figure 30 gives a schematic view of the concept of pattern database.

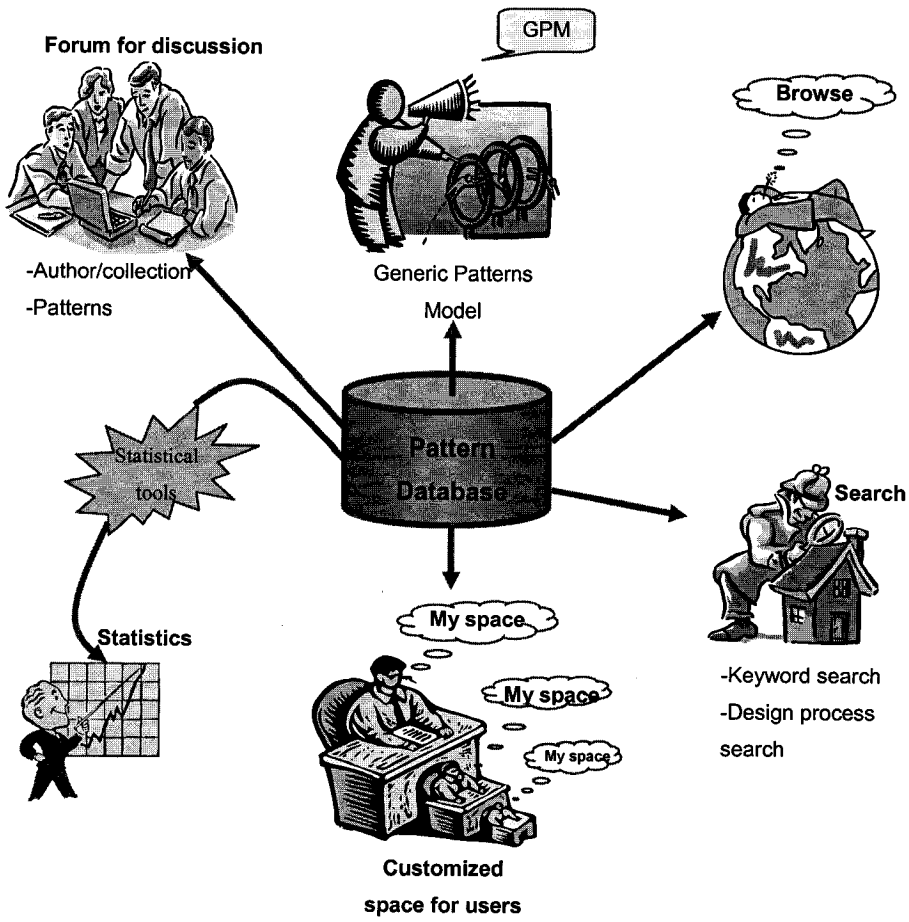


Figure 30: The auspice of database

As mentioned before, the goal would be to facilitate information retrieval and reuse. As we promote the concept of database, we emphasize some advantages attainable by using it [Gaf01]:

- **Scalability**: A separate database can store and manage larger data volume than files attached to a specific application.
- **Facilitate efficient updates**: Unlike flat text files, a database categorize its contents into tables or objects which help improve the lookup and editing of contents

-**Connectivity**: Several applications can connect to the same database, reducing the need for extra copies of data files and multiple updates (see integrity).

-**Promote interaction with other software**: A database can provide application-independent information that can be called and used by several software tools and applications

-**Reliability**: A database usually has advanced capability to enforce reliability against mishaps like programmer errors and power outages

-**Integrity**: If properly designed, database can ensure integrity by having a single copy of each data object so changes can be centralized to one place. This can greatly reduce redundancy and inconsistency

5.4 A Suitable Data Type for Patterns

For every complex question there is a simple and wrong solution.

Albert Einstein

Building a database to support pattern interoperability and automation is a challenge. Databases support many kinds of data types and access method [BCC+02], [Gaf01]. However, the underlying challenge is that for patterns, we don't have a specific "data type" per say or a well defined access method. Therefore, the first obstacle lies in the multitude of pattern formats and how to model them as a data type. The text representation used by pattern authors makes it unsuitable for database without modification. The modification can be a simple "listing" of the title and subtitles of each pattern before saving the associated text chunks linked under each of them. However, this will have negative effect on pattern lookup and automation. Without proper semantics, the text lookup will remain manual, or by keywords at best. The empirical study

[Gaf04] and [Gaf05c] showed the inefficiency of this approach. Without comprehensive modeling, tool support will not be feasible. Writing efficient tools to process text documents is hard. In [GSP05], we investigated the "scale of restriction" and showed that with less restrictions imposed on a schema or a data model, it is harder to write tools that can automatically access and process documents written using this model.

While text clusters can help store patterns in a database, we believe there is more space for improvement. Saving patterns as "smart components" with well defined semantics facilitates the interaction between these components and other software that uses patterns, the "calling software[7]". While these calling software artifacts are not aware of a particular pattern they are calling, or the particular constituent of a pattern, they are fully aware of the semantics of these constituents and they know how to look for what they need inside the database. In the end, if the calling applications can make the link between the behavior needed to solve a problem and the behavior offered by a specific pattern, they can make the selection and call that pattern. Modeling is the enabling approach we apply to solve this challenge. Unlike concrete artifacts, models provide us with a "*look ahead*" capability. By look ahead we mean that while specific patterns are not known, a pattern model helps the calling software manipulate the existing pattern model in their interaction. If the models provide clear semantics that represent patterns well, software can interact with this model, and query the database to look for suitable patterns that offer the desired semantics. In appendix A we provide some examples of these semantics and demonstrate how they can be used to build arbitrary algorithms to automatically process patterns using external software tools.

When talking about a database, we don't necessarily restrict our approach to a centralized, Internet-based database that is globally accessed. Our survey

---

[7] Here we are using the famous Hollywood Principle *"don't call us. We will call you"*

(appendix D) showed that local database could also be appreciated in closed communities once they provide semantic modeling for patterns and facilitate interaction with other software and tools. In [Gaf05a] and [GSP05] we show how the concept of programmable patterns has been abstracted above this strategic goal and has been adopted within the context of both a Web-based (global) database and a local one.

The view of generic interoperability motivated us to build a generic data type for patterns. As we investigated several pattern formats on the Web, we extracted and analyzed the similarities and differences between them. Our analysis revealed several observations that guided us in constructing the schema:

- All patterns started with a pattern name and a pattern author.
- Patterns predominantly have three major components: the *problem*, the *context* and the *solution*. We call them the "minimal Triangle". The majority of pattern definitions involve this triangle in some variation, making it the common denominator between most pattern definitions. Appendix A provides more details on it.
- Different authors added different items to the minimal triangle.
- The solution part also had some commonalties and differences inside them.
- All patterns had examples to show. They differed in where to put them in the pattern. Some started off with an example as the very first part, while others had their examples scattered throughout the pattern definition.
- Much of the data related to patterns lies not only within each pattern, but also in the kind of relationships between different patterns. That is why most pattern representations had some entries regarding the relationship between each pattern and other patterns in the collection.

The above observations led us to build a generic schema to instantiate our conceptual approach. The schema allows us to include patterns from different collections in a programmable approach. This generic schema covers different

pattern formats in a structured predefined model. Concrete interfaces are derived from this schema, hence allowing communication with other tools. We briefly discuss some of these ideas below.

We divided the pattern representation into four main parts: *head*, *body*, *relations,* and *assimilations*. They reflect the three main categories that we made distinct in [GSP05]. The *head* is an additional part that holds essential metadata about a pattern. It contains the unique keys to a pattern (like a unique name and pattern ID) as well as the pattern authors. The *body* contains the intrinsic information about the pattern, while the *relation* part collects the information pertinent to the relationship between this pattern and other members of the collection (the extrinsic information). The assimilation part is used to hold variables that need to be instantiated in a concrete design as shown in Chapter 4.

We elaborate on one part –the body- as an illustration. The body part has two main subcategories, namely, the theory and the practice.

The *theory* part contains all information about a pattern as seen by the pattern author(s), resulting from the author's analysis and understanding of the pattern. This part encourages pattern authors to express their insight and analysis of a pattern. It contains the *minimal triangle* and other relevant information. The *minimal triangle* refers to the triad *problem*, *context*, and *solution*, which are the common denominator between most patterns and essentially the minimum information for a useful pattern. While other theory items can be missing in any pattern, we found that this triangle is critical for realistic pattern documentation and reuse. Any missing one of the three parts can turn a pattern into a *Trivial Pattern* (appendix A).

The *practice* part collects all information about a pattern as being used in practice by the author and/or other users. This part encourages users by giving them real-life examples or implementation strategies that they may want to follow, and

possibly other bad examples or pitfalls that they may want to avoid (anti-patterns as discussed in Chapter 2). Figure 31 shows a view of this type.

This view is a high level abstraction that reflects the requirements of the pattern model from the problem domain point of view, regardless of how it can be implemented or on which platform. Due to its generic nature, we called it the Generic Pattern Model, GPM [Gaf05]. Details of GPM role within the system are given later in this chapter.

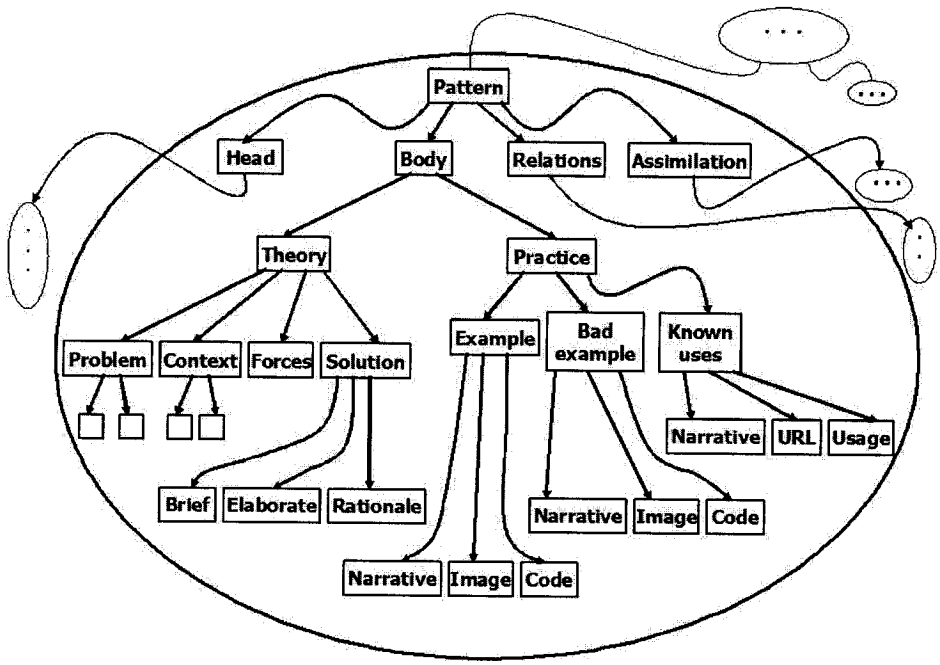


Figure 31: The hierarchy of the GPM

5.5 Modeling a Complete Pattern Lifecycle

In the previous sections, we proposed a schema and discussed its usefulness from a delivery perspective. Here we emphasize this idea and argue that a key factor to increasing the usability and usefulness of patterns is the adoption of a model for a complete pattern lifecycle. This would underwrite the activities that should be taken when identifying a pattern, documenting, delivering, applying

and maintaining it. Creating mechanisms to manage the overall pattern lifecycle has received little attention in the HCI patterns community so far. Our investigations and work as presented in the previous chapters provide a basis for better understanding of the process of discovery, representation delivery and applications of HCI patterns. However, the currently prevalent pattern approaches can be divided into two major activities only: pattern discovery, and pattern reuse.

- **Pattern Discovery** refers to the activity of writing patterns by domain experts.
- **Pattern Reuse** refers to the activity of applying patterns in a useful design as recommended by pattern authors.

We could not find significant work on guiding the user through the activity of finding the suitable patterns for reuse. This can have negative impact on promoting pattern reuse (as shown in our empirical study, [Gaf04] and [Gaf05c] and appendix E). In this regard we introduce an intermediate logical layer between pattern discovery and reuse, namely the dissemination process.

- **Pattern Dissemination** refers to guiding pattern users through the activities of locating all useful patterns, selecting some of them using different criteria and then applying them in different phases of design and implementation process.

Based on the empirical study and the proposal of the dissemination process, we can now redefine the lack of effective pattern reuse as a symptom and not a problem. The problem in this chapter is better identified as

1- The lack of a dissemination process
2- The lack of a common and programmable pattern representation to help in combining and reusing patterns and pattern tools in practical design environments.

Having attributed the problem to the current narrative format for documenting patterns, we propose an approach to represent patterns as software components by identifying and rewriting their semantics as a model for designers as well as design tools. This model transforms text patterns into programmable objects with well defined interfaces that reflect the knowledge underlying them. This makes them accessible in any object oriented programming language as well as in XML for tool interoperability. We also provide a framework that supports a comprehensive dissemination process.

5.5.1 A Model for Pattern Lifecycle

In document-based description, we directly transform available data into text documents that are instances of our information objects. In model-based description, an additional layer is inserted between the data and the documents, namely the modeling layer. We first design documents as abstract models, and specify the desired structure, syntax and semantics using these models. We describe the desired behavior of each model and the interaction (or interoperability) between different models as well as the integration of these models into different design processes and artifacts. During the modeling phase, we can use sample or simulated data as well as algorithms and software tools to experiment and validate the models. This modeling approach has many advantages. At different design phases, it allows designers to manipulate abstract models with just the right amount of information and integrate them into their design artifacts without cluttering them with too many details. At the code level, it enhances interaction and integration with the application source code once data models have well known interfaces that accurately represent their semantics. Moreover, scalability is enhanced by allowing tools to interact with these models in an automated way, and in much greater numbers than the human processing capacity with the narrative format. Looking up information using different queries and search criteria will be greatly improved.

Figure 32 gives a simplified view of our approach, seen as an addition to the current approach of pattern reuse. The lower box represents the current concept and the upper box represents our system. The lower box has two layers. The bottom layer is a virtual link between a domain expert (the source of experience) and a pattern user (the destination of experience). This experience can be transferred physically with direct, mentored interaction between both roles, or through the next upper layer of the lower box, using patterns. In this model we focus on patterns as the main method of dissemination process. A more comprehensive model for other knowledge dissemination techniques (e.g. [Cas97]) can include other means of knowledge transfer as discussed in section 2.1. Here we focus on modeling pattern lifecycle.

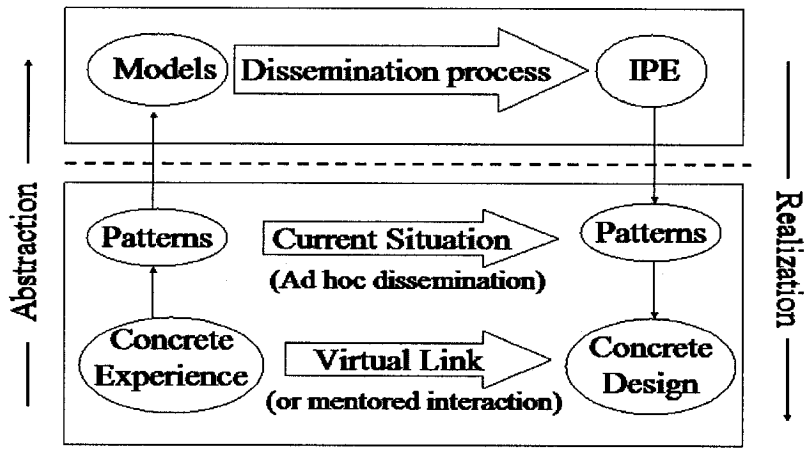


Figure 32: A model for pattern lifecycle

In the added third layer we apply modeling techniques to specify scalable dissemination and assimilation processes. As narrative patterns are rewritten according to the suggested models, they conform to our software system specifications and thus offer the interoperable behavior. An Integrated Pattern Environment, IPE, is the system that helps users get access to a large number of patterns as software components, and offers the functionality needed to efficiently manipulate them. To explain the idea, we abstract the concept of

Figure 32 into the lifecycle of patterns as shown in Figure 33. The key points of this lifecycle are the dissemination process, the assimilation process, and the Integrated Pattern Environment, IPE connecting these two processes together.

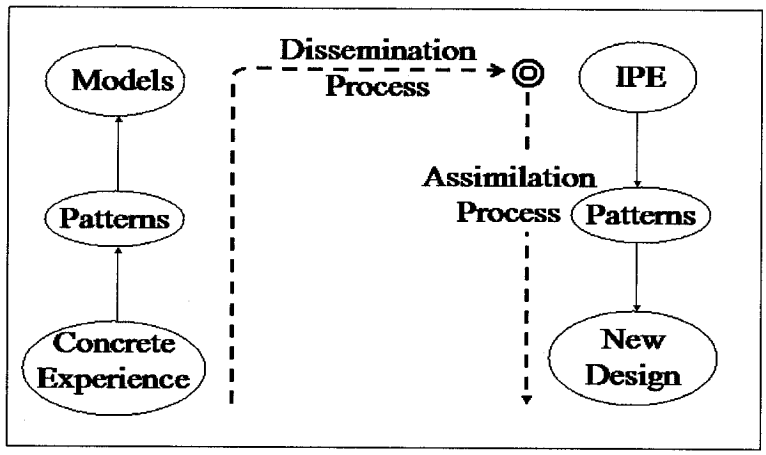


Figure 33: The abstract lifecycle model

5.5.2 Defining the Dissemination Process

Dissemination refers to "*the activities associated with delivering knowledge and experience from pattern authors to pattern users –or designers*". For efficient dissemination, we need to reduce the time spent by users in looking up patterns, and the ability to locate all patterns that can be useful according to some search criteria that a user can apply. So far, these activities have been left to the user. Pattern authors simply publish their patterns, generally in books or on the Internet and the dissemination process stops there. Some pattern authors recognized the problem and added links within their collections to other collections. This, however, adds to the confusion of the users as they see similarities between collections. They get distracted or lost in the available maze of patterns [Gaf05c]. In this context, we have defined the *visibility problem* that new patterns suffer as "*new patterns become diluted in huge pattern offerings and hence get no significant chance of making their way to users*". This has been confirmed by the results of our empirical study [Gaf04] and [Gaf05c].

Consequently, many designers limit their pattern repository to few patterns that they already know, and rarely look for new patterns.

5.5.3 Defining the Assimilation Process

Assimilation refers to the design decisions made by pattern users as to how selected patterns can be applied during different stages of design, and how they can be combined together within the artifact being designed. As we have shown in Chapter 2, pattern authors have often connected several patterns in their collections and presented them in graphs showing patterns as boxes and relationships as arrows. They generally suggested how each pattern could interact with other patterns in an "approved" way. However, it is left up to the pattern user to figure out how to incorporate patterns from different collections and in different design phases as well as which patterns correctly belong to each phase. On the other hand, we showed in Chapter 4 that existing modeling approaches and design process allow specific integration of selected patterns only. Generic tool support is essential in this stage to help users manipulate and integrate patterns within a well defined pattern-oriented dissemination process but also using patterns from other collections. Pattern components work as objects that offer their semantics in XML as well as java/C# classes with well defined interfaces to communicate with the design environment, the IPE.

5.6 High Level View of a Dissemination System

Good people with a good process will outperform
good people with no process
every time.

Grady Booch

Software systems can be more successful when designed with an eye on their context of use. Following a structured process that incorporates the context of use can ensure relevant functionality and usable design. In planning for an IPE as a software artifact, we adopt this concept by designing a process as a technique for organizing and documenting the structure, activities, and flow of data through our system.

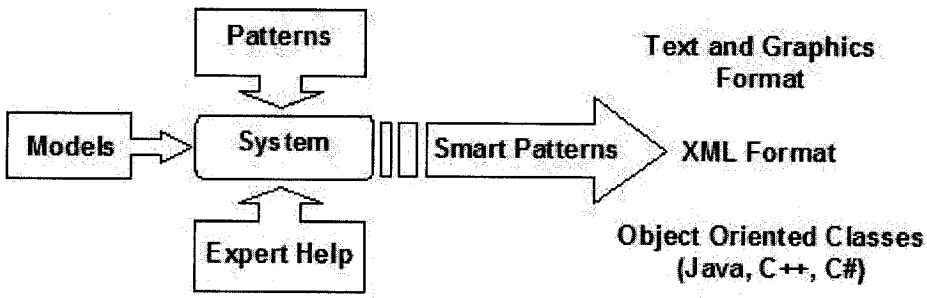


Figure 34: *Smart Patterns* delivery system

Starting at an early phase of proof of concept, we will not clutter the system with too many details. We present an abstract view first. [LM95] emphasize that abstract representation of a system helped designers focus on the contents, while too many details distracted them into discussions about unimportant issues. We start by presenting the main components of our actual system as shown in Figure 34. The system is intended to combine existing textual patterns with models to produce programmable pattern objects, or "smart patterns". Smart patterns are pattern objects that are capable of communicating with other object and tools in a runtime environment. Expert help is needed to manually collect and rewrite existing patterns using the models. This manual help is performed at the initial construction of the dissemination system and for adding more patterns to it.

Pattern authors are strategically replaced by expert help performed on available patterns. The idea is that we can not force pattern authors to write their patterns directly into a dissemination system using any predefined format. They usually

prefer to not be bound to a specific layout. Therefore, the system allows patterns in textual formats to be rewritten using the given models without losing any information. They can be retrieved and restored to their original form or manipulated as software objects. Finally, system designers and system builders are not shown here as their roles are non-functional to the dissemination of pattern knowledge.

5.6.1 The 7C's Process

The central aspect of process oriented approach [Flo87] to automating pattern-oriented design is its dependence on a predefined process. An established design process instigates quality design by allowing designers to follow structured methods in their activities. In our approach, we established the need for both dissemination and assimilation processes. We implement the dissemination process completely decoupled from any specific assimilation process. This allows it to offer patterns that can be integrated simultaneously in several assimilation processes. "Free patterns" that do not belong to any process are hard to integrate in design. Similarly, "proprietary patterns" that are specifically tailored to manually fit one design process using one specific example defeat the main purpose of pattern generality and abstraction. We see that a pattern can be integrated in several assimilation processes by properly encapsulating its knowledge and presenting its behavior through a well defined interface. Any assimilation process can then lookup pattern objects and select the appropriate ones using different search criteria. The selected pattern components can then be integrated in new designs or used to generate code fragments.

We define the 7C's as "*a structured process with the main objective to replace the huge cognitive load of manipulating HCI patterns with a dissemination system of smart patterns*". The 7C's process identifies both logical and physical aspects of the system. A logical process focuses on *what* actions and activities need to

be done. A physical process complements the logical process by specifying the roles associated with the process, and details *who* is going to do what [HGV02], [WBD01]. As part of the pattern reuse problem is associated with missing roles in the dissemination activities (all left to the user), the 7C's process addresses both how these activities need to be done, and who should be doing each of them. In short, the 7C's process moves gradually from current unplanned discovery and use of patterns into building an automated pattern collection. The process comprises seven steps:

Collect: Place Different Research Work on Patterns in One Central Data Repository

Despite the proliferation of research into HCI design patterns since the 1990's, there has been no successful attempt yet to unify these efforts or collections. Numerous works on patterns have been developed in the HCI community, however they are scattered in many different places. A central repository of patterns will allow users to concentrate on knowledge retrieval rather than spending time on search for patterns.

Clear Out: Change from Different Formats into One Style

Ideally, different works on patterns deal with different problems. However, as we went through step 1, we were able to identify that some patterns are dealing with different sides of the same problem (correlated patterns), some patterns are offering different solutions to the same problem (peer patterns/competitors) and some are even presenting the same solution to the same problem (similar patterns), only in different collections with different presentation formats. Since a large number of patterns have different presentation formats, it is difficult to detect these redundancies or useful relations with other patterns until the user has spent some unnecessary time with several of them. Putting patterns in a unified format helps discover these relationships, put related patterns closer together, and possibly remove the redundancies and inconsistencies.

Certify: Define a Domain and Clear Terminology

This activity is necessarily human-driven. While inferencing can find useful relationships amongst patterns, the validation of good patterns will largely be a matter of people assessing them against experiences and through use to decide if they were "really good patterns". We use a process similar to that of Answer Garden [AM90] where new pattern proposals are routed to a distributed set of experts in different usability areas. These experts can then provide feedback to pattern writers, pointing them to similar patterns, and otherwise facilitating the process of creating a useful set of patterns.

Contribute: Receive Input from Pattern Community

New patterns emerge all the time in many areas of the scientific community, including HCI. It is very difficult to keep track of these emerging patterns. Typically, it would take years before an expert can come up with a thorough collection of patterns [AIS77], [GHJV95] or have time to update an existing collection [Tid97], [Wel00]. Having a central repository for patterns help unify pattern knowledge captured by different individuals in the future. Furthermore, putting such a repository to use in actual design situations will help to spot areas of design activities where there is a shortage of patterns so that they are made available to the community. The "central" concept refers to either a community-wide Web-based repository or a local repository within a smaller group of people. In both cases, a repository will help unify the effort of collecting and contributing to patterns.

Connect: Establishing Semantic Relationships between Patterns in a Relationship Model

A significant part of knowledge associated with patterns lies in the relationships between them. After the Clear Out step removes redundancies, the Connect step is meant to build new connections between patterns. Finding and documenting these relationships will allow developers to easily use patterns as an integral part to develop applications instead of relying on their common sense and instinct to

pick up patterns that seem to be suitable. A proven model for the pattern collection helps to define ontology for the pattern research area with all proper relationships such as inference, equivalence and subsumption between them.

Categorize: Define Clear Categories for Patterns that Map them into Assimilation Processes

Within the collection, we need to create pattern classifications or categories to make them more manageable. The first goal of categorization is to reduce the complexity of searching for, or understanding the relationship between patterns. For example, some patterns are just abstractions of other patterns. The second and more important goal is to build categories that can be mapped to different design approaches and methodologies, and then put patterns under their appropriate categories. This is the enabling technique to integrating patterns into different phases of several design approaches. As explained earlier, decoupling the dissemination and the assimilation processes allows same pattern to belong to different categories and be used in different assimilation processes.

Control – Machine Readable Format for Future Tools

Defining pattern models that accurately represent pattern semantics through their interfaces and rewriting patterns according to these models enhances the process of automating UI design using different assimilation processes. The ultimate goal of the 7C's process is to allow user to interact with the machine as a viable partner that can read and understand patterns, and then process them in an intelligent way. Having machine-readable patterns is the last step in the process of pattern dissemination and the first step towards assimilating them.

5.6.2 The Integrated Pattern Environment, IPE

While it can be seen as another "tool", we can more accurately explain the notion of IPE as a "unifying concept" that offers interoperable models, a notation to implement these models using XML or any object oriented language, and an

underlying multi-tier information system to store the essential data and to offer interaction with it.

Motivating the Use of XML

XML is becoming a ubiquitous standard in software connectivity. Not only does it facilitate the communication between programs, but it has many advantages that can enhance system capabilities and interoperability. The rich set of XML technologies that can be applied with XML-compliant systems can have great advantages added to the system

- **Storage:** XML has proven its power as a markup language for documenting structured information in different fields. Patterns can be stored in an XML-compatible database. Using XML compatible technologies like Cascading Style Sheets (CSS), XPointer or XSL, the presentation of pattern documentation can be customized and delivered to the developers (using XUL for example) or delivered as an HTML code for rendering. While pure XML databases (commonly known as native XML databases) suffer from efficiency and security problems, the XML community provides many solutions to overcome these drawbacks. Similarly, major vendors like Oracle provide XML support to non-XML databases.

- **Interoperability:** Several XML-based languages for UI development have been introduced recently including UIML (User Interface Markup Language), XUL (Extensible User Language) UsiXML, and XIML (eXtensible Interface Markup Language). XML allows communicating between them at the code level.

- **Automation:** XML is a good option for building pattern ontology that can support some semantic reasoning such as when applying a pattern and

why other patterns can be used also. In appendix B we provide a case study on pattern Ontology

The Role of XML in Pattern Automation

Every text can be rewritten in XML by simply adding markup tags to explain what the subsequent piece of text is[8]. The mere fact that a document is written in XML allows for some automatic processing to be applied to it using most programming languages, while keeping it readable by humans. Simply stated, the contents of the document can be "automatically manipulated". This is often referred to as *document-oriented XML*.

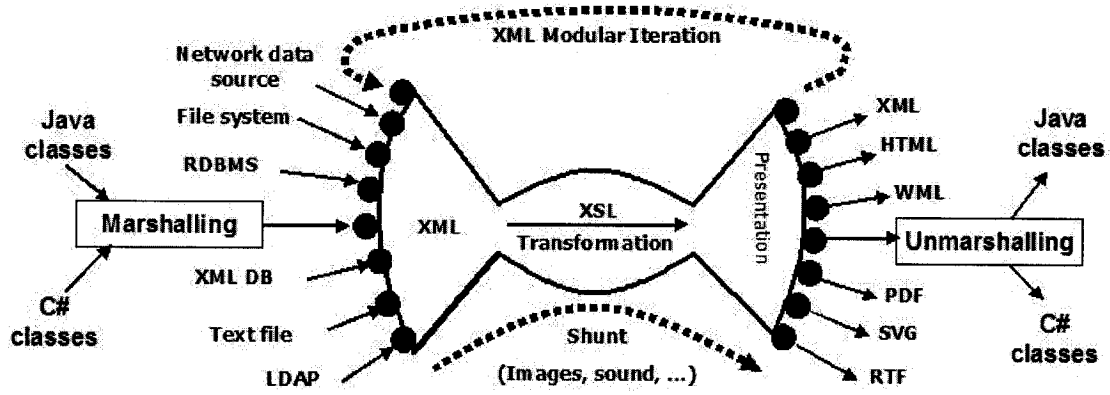


Figure 35: The XML space

However, XML can offer much more than that. Goldfarb and Prescod [GP04] explain that true interoperability requires not just interoperable syntax, but interoperable semantics. If the XML document was additionally structure and built according to a valid schema following predefined models that allow for

[8] To be more exact, the text also needs to be saved as a plain text document in ASCII format. Simple software like notepad can do the job. Accordingly, all formatting and styling of the text will be lost, but this is the essence behind XML: it focuses on the meaning of the text and not on the formatting. Formatting can be annotated later to any XML document.

interoperability, the capability and the power of processing the data becomes much greater as it becomes fully structured. This is referred to as *data-oriented XML* and is intended mainly for automatic processing, not for humans to read. It will be easier to write tools to manipulate the contents of a document. More importantly, as explained earlier, it will open the gate to a large number of available XML technologies to be directly applied to the document for automatic processing.

To achieve universal data manipulation, the input to XML space is generalized into abstract data source and the output to abstract data sink. Accordingly, any data format can be seen as an "information source" that feeds into the XML space by initially transforming it into XML (Figure 35). After the transformation phase, the document can be rendered into the format of any "information sink" at the output, or redirected back into the XML space for further transformation. This allows several XML technologies to be pipelined in a modular way to generate arbitrarily complex processing and rendering schemes.

Smart Data

The enabling key into this XML space is to represent data in a properly expressive format that displays the desired structure and behavior in it, regardless of any specific application. This can be referred to as "smart data" contrary to smart applications. In smart applications, data is created through- and belongs to- a specific application (e.g. Excel spreadsheets, MS Word or an Adobe Acrobat document). Data is represented internally in different proprietary formats and are accessible only by running the "owner" application. Transferring data from one application to another requires knowledge of all details of proprietary formats of both applications. For m applications, this could mean as many as m*(m-1) transformations, a tedious and unpractical approach. Smart data, on the other hand, present all semantics within data itself in the common, non-proprietary format of XML. Any application or a tool can be seen as an

information source that exports its proprietary data once to XML space. Similarly, applications can be seen as information sinks, and XML data can be imported by them from the XML space. To cover all possible transformations, this needs to be done only twice for each application; in and out of XML space. For m applications to fully communicate, the total number of transformations is reduced to 2*m or less (for bidirectional transformations). For a large m, this is a significant saving. Following this concept, we use XML as the central representation of pattern components, and we use tools to automatically transform them back and forth between object oriented classes, text as well as other XML based languages.

5.7 The System Design and Implementation

In between the nice ideas...the vision,
and a working software product,
there is much more than programming

Philippe Kruchten

At this stage we zoom in at the system architecture by refining Figure 34 into Figure 36. Looking at the data pathways depicted in this figure, we see that the input information is now bypassed from the direct path (from pattern collections directly to user) -marked as A- into the system pre-storage phase, the pattern corpus, and then to the rest of the system -marked as B.

The **Manual Processing** block represents the manual activity by users to search for patterns and read through the text, analyze its contents, and figure out which patterns to choose and how to apply them in design. We compare the two processes, referred to as A and B. If we considered them as two different dissemination processes of patterns, we can refer to the Integrated Capability Maturity Model, CMMI of the Software Engineering Institute to briefly evaluate them.

- **Process A**: it does not follow a particular approach of dissemination except for relying on users' manual processing (looking up patterns, understanding them, and applying them in an ad hoc fashion). We evaluated this process to be at *CMMI Level 1 (Initial).*
- **Process B**: As suggested by the dissemination system, there is a process in place to help users interact with patterns in a structured way. Moreover, this process relies heavily on feedback, and is constantly changing, as seen in the 7C's process within the system. When fully applied, we estimate it at *CMMI Level 4 (Managed).*

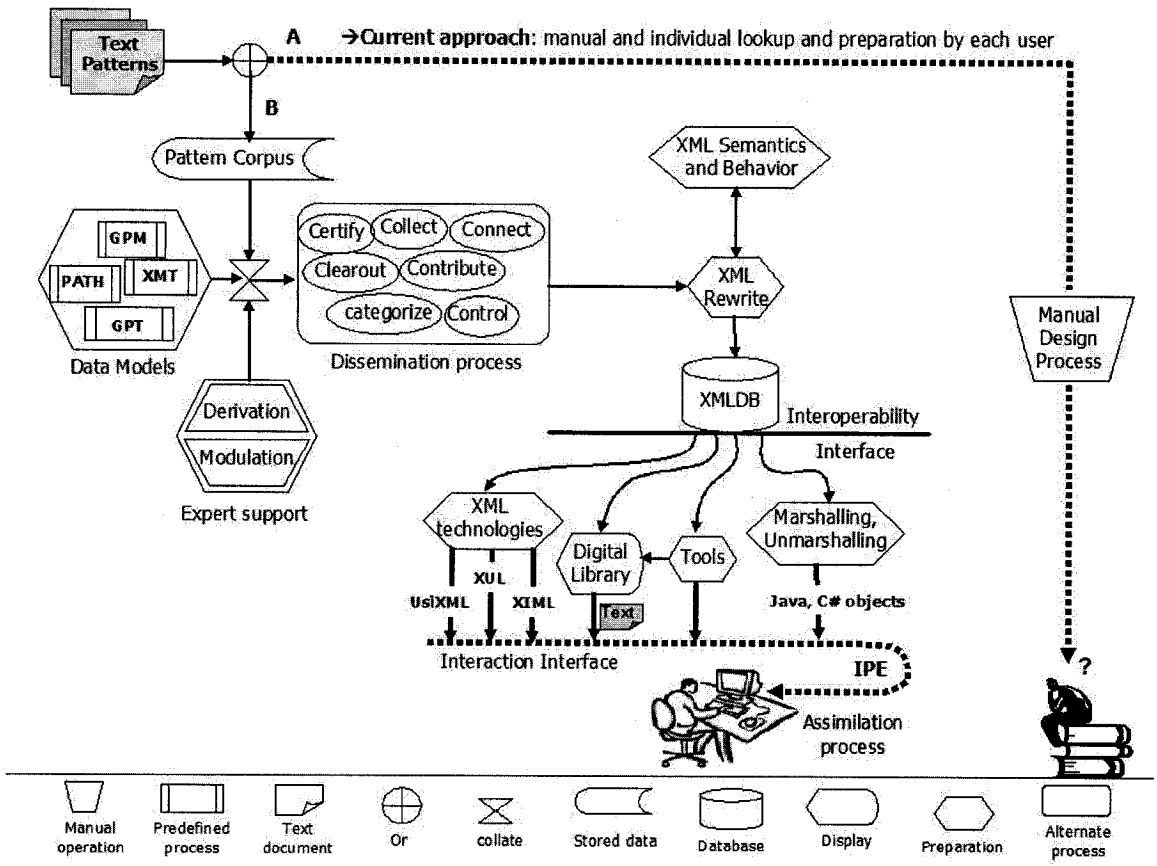


Figure 36: A comprehensive system to disseminate *HCI Patterns*

In Figure 36 we show the main modules and the activities associated with them as implemented in our system. We have three main subsystems: Input, Processing and Output.

Input: The three input modules to the system are the *pattern corpus*, the *data models*, and the *structured expert support*. The pattern corpus is a pre-processed collection of useful patterns obtained from the available pattern collections. The data models implement the aforementioned additional modeling layer as explained in the pattern lifecycle model. These data models allow the rewrites of patterns to be interoperable and machine-readable. The Structured Expert Support, SES, is a methodical human activity that has two major activities: *Derivation*, and *Modulation*. The former refers to reducing redundancies between existing patterns, which is abundant in current collections. The latter refer to rebuilding the derived patterns to conform to the defined models. We provide more details of them later in this section.

Processing: The three input modules are collated together by applying the system process (the 7C's process) which defines the systematic activities associated with the three input modules. It defines a step-by-step implementation of processing the pattern corpus and applying structured expert support (SES) on it according to the defined system models.

Output: The system process transforms the input into pattern objects called "XML rewrites" to conform to the given models. XML rewrites are validated against the semantics and behavior specified by these models. They are then stored in an XMLDB, and are presented for interaction as a back-tier of the three-tier system as shown on the right hand side aggregate within Figure 36. The middle tier of the system represents the interoperability offered by the system in the form of modular components that interact with the database and implement any desired algorithms. These components can range from tools to read the XML patterns and apply new functionality to a digital library that offers the contents of

the DB for browsing, lookup, or updating [GS05]. This layer also links the XMLDB into existing tools that read XML documents and transforms them to XUL, UIML or other specifications. Similarly, the layer allows for automatically transforming patterns into objects in different object-oriented languages and back (marshalling / unmarshalling patterns). In our implementation, we automatically generate fully functional java and C# objects from the XML patterns. These objects are then used in java and .NET platforms as regular software objects that encapsulate pattern information and behavior. The front tier of the system is the presentation environment, which is the interface aggregating the available tools and using the pattern objects directly. We briefly explain some of the key modules of the system.

5.7.1 The Generic Pattern Model, GPM

The GPM defines a generic model to describe the structure and the functionality of patterns at conceptual, high level semantics. It covers the necessary aspects required for patterns. The standard behavior specified by the GPM allows for common understanding and works as the middle ground between pattern writers, tool writers and pattern users.

5.7.2 The Generic Pattern Type, GPT

The abstract view of the generic pattern model has to be implemented as detailed and concrete data types needed for software implementation. The GPT defines arbitrary, concrete implementations of the conceptual GPM depicted earlier in Figure 31. According to different design requirements, programming paradigms and platforms, each concrete instance of GPM is a GPT. GPT is a well defined type as defined in strongly-typed object oriented programming language, similar to integer, double and complex types. A concrete GPT is well defined in the sense that all its constituent components and their primitive types are well defined. While GPM is a conceptual model that helps abstract and standardize pattern behavior, the GPT is a low level complex type that

encapsulates pattern information and provides concrete interface which allows access to all data and functionality of patterns. Each text pattern is rewritten as a new object by declaring a new instance of the GPT. These classes are easily transferred back and forth between object oriented paradigm, normalized relational tables and XML using existing tools.

According to the selected programming paradigm, we implemented a concrete GPT from the GPM in a variety of ways:
-In XML paradigm, we implemented a GPT as an XML schema and patterns were written as XML documents that are validated against the schema
-In entity-relation (ER) paradigm, we implemented a GPT as an ER model that was converted into relational database with tables holding pattern information
-In object-oriented paradigm, we implemented a GPT as class "pattern" that aggregates different subclasses inside it. In their renowned book "design Patterns" [GHJV95], Gamma et al. emphasize that whenever possible, programmers should lean towards aggregation over inheritance. Therefore we opted for implementing the GPT as an aggregate class in this paradigm. Nonetheless, building a GPT from the abstract GPM using inheritance is possible, albeit not efficient.

In all cases, building a GPT as a composite structure gives a great flexibility for changes. When small modifications are needed, they can be limited to one part of the GPT (for example one subclass) without the need to rebuild the whole system. Great implementation flexibility is attained by building the GPT as a complex aggregate type.

Automatic Transformation between Paradigms

Using available tools, we were also able to increase the flexibility of the system by automatically transforming the same GPT between the three programming paradigms mentioned above and between several platforms and languages. The

manual work of building a specific GPT was made only once as text patterns are loaded into the system using a pattern loader tool. The tool accepts text patterns using a graphical user interface. Patterns are transformed either into an XML document (according to a validating GPT schema), as a complex java class, or as a complex C# class. Several existing tools allowed us to automatically generate the equivalent GPT's in the other two paradigms with the same semantics. This allows for interacting with the same pattern in different programming environments without losing or distorting any information. Regardless of the free manipulation options, patterns are saved as one entity in one persistent layer in the backend of the system.

5.7.3 The EXtensible Minimal Triangle, XMT

XMT allows for automating the manipulation of patterns using an extensible repository of predefined keywords added to each pattern within the GPT. Tools can be written to automatically process pattern contents according to the semantics of XMT. For example, we can define an algorithm to compare pattern similarity and equivalence by comparing the relationships between the keywords of the *problem*, the *context*, and the *solution* of different patterns. A simplified example to demonstrate the algorithm is that patterns that offer *different solutions* to *same problem* in *same context* are considered equivalent patterns in the algorithm of this tool. Additional details are given in appendix A.

5.7.4 The Progressive Abstraction Type Hierarchy, PATH

PATH categorizes patterns in a hierarchy that maps directly to different steps of software design process (please see the Categorize step of the 7C's process). This allows for effective selection and assimilation of specific patterns in each stage of the design. The hierarchy is logically represented as a virtual tree, but

physically it is implemented by adding the category information of each pattern into the GPT. A tool can then extract this information from the patterns and render the hierarchy as a tree to the user. The more important function of this model is for design tools to extract and link patterns into corresponding design phases. Other hierarchies are being modeled to reflect categorizations proposed by different pattern authors. As explained, patterns are then examined to see how they can belong to each of these models. Additional details and demonstration are given in appendix A.

5.7.5 Structured Expert Support, SES

Part of the problem identified with patterns is the large cognitive load to understand and process patterns and apply them in an effective way. As we have noticed in our empirical study, users encounter a large number of patterns in many collections with several redundancies and ambiguities. In this work, we are reorganizing patterns in two different ways:

Derivation

As pattern experts look at patterns in different collections, they instantly notice several redundancies. Other redundancies are not immediately clear despite their presence. Besides greatly confusing the reader, these redundancies contribute to the bloated number of patterns offered to the user. After collecting and analyzing tens of redundancies, we were able to define different categories of them –from total to subtle redundancy- and develop structured procedures to reduce them. The resulting, less redundant patterns can be seen as derivatives of the original ones. This is a highly technical activity, and -if not well parameterized- it can be subjective and even divisive. We see that controlling the redundancies will alleviate some of the cognitive load of processing patterns, an activity often left to the user as we highlighted. We also see that it is simply too much for one novice user to do all that, and for each user to have to do it from

scratch; something very similar to "reinventing the wheel". From our analysis, we assert that four factors are adding up to making this part of expert support a fruitful and cost-effective effort:

-We define the types of redundancies and how to remove them

-We delegate this activity to experts

-We combine the efforts of more than one expert due to the large number of patterns, and the strong coupling between their contents

-We record the outcome of this cleanup process and present it to users

Modulation

After building models to improve pattern reuse, we utilize the help of pattern experts to rewrite parts of the patterns to conform to these models without losing the knowledge within patterns. We call this activity "modulating patterns".

Looking to the fact of the multitude of pattern authors and collections, we can see that it could be a difficult –if not impossible- task to convince pattern authors to follow any structured process until it is "tried-and-true". Therefore, we made a tactical decision by rely on the *Structured Expert Support* to bypass this obstacle and make it uncritical to the system inception and success.

5.8 The Multi-tier Software System

Figure 37 focuses on the architecture of the software as an essential part of the overall pattern dissemination system. It is implemented as a three-tier system composed of storage (back tier), processing logic (middle tier) and an interaction layer (the front tier) [GS05]. The three layers working together, the supporting models and the process, as well as the expert support work collectively to offer the integrated pattern environment, IPE to designers [Gaf05].

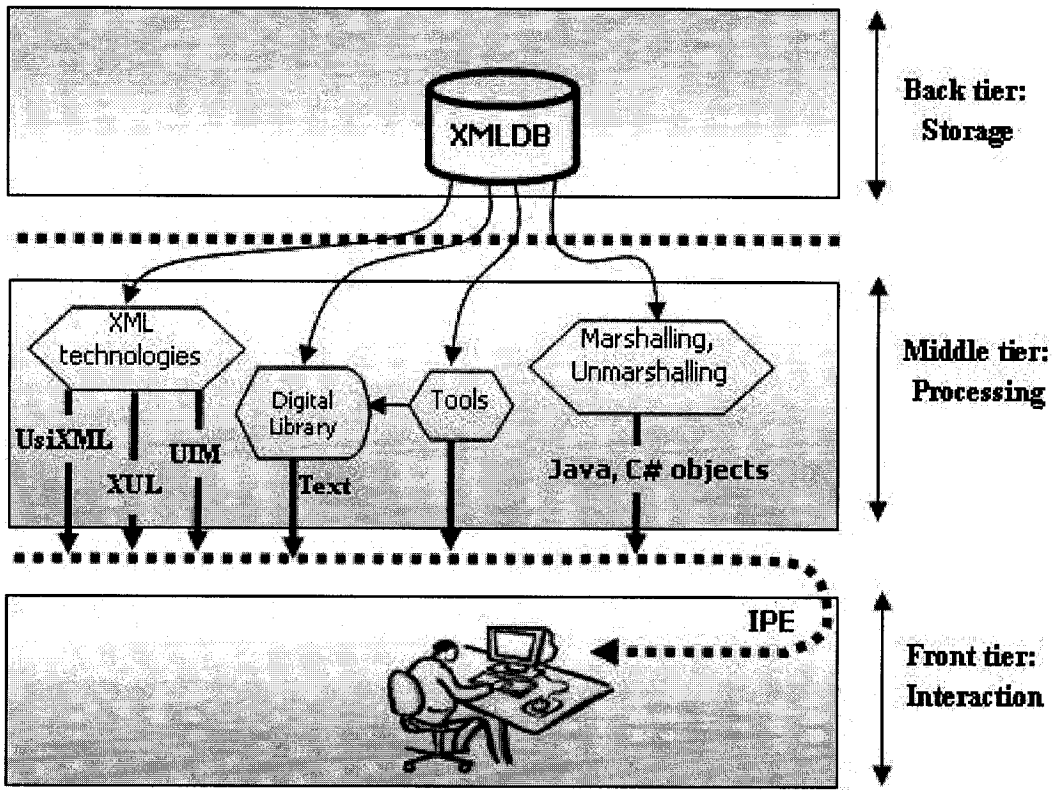


Figure 37: The multi-tier software system

5.9 Conclusion

Despite the wide acceptance of patterns within HCI research community, the current approach of pattern reuse is a simple process of publishing numerous patterns using different media and leaving it up to HCI designers to do their best in figuring out how to trace and apply them in new interfaces. From our investigations and experiments we conclude that this unnecessarily represents a cognitive load on designers who want to look for patterns and reuse them in their design.

We suggested an addition to this current approach to help define and standardize the process of pattern dissemination and assimilation which can lead

to an effective reuse of the knowledge contents within patterns. We proposed to represent patterns as software objects that encapsulate pattern semantics and allow interaction with them through their interfaces. We have developed several prototypes under two broad categories, namely as an online digital library with universal contributions, and as a personal digital library on personal computers for small scale pattern collections. In both categories we prototyped several options at the database level (persistent layer), the processing level (tools), and the interface. The models and the associated notation are provided and adopted to facilitate interaction with users and to pave the road for tool support to enhance the system.

Chapter 6

Conclusion

This thesis investigated the way HCI patterns are currently represented with the perspective to enhance their reuse in the design environment. The analysis of patterns in the research community and pattern authors' domain (Chapter 2) as well as the experiments of pattern applicability in design environment (Chapters 3 and 4) lead to the proposal and the new approach of a comprehensive pattern environment as shown in Chapter 5. We summarize our contributions to the field of patterns within the scope of interactive software design, and our conclusions. We also discuss some limitations to our approach that may open opportunities for future work.

1 Pattern Reuse

To first explore and highlight their practical role in software, we studied and showed new potential for patterns. They can provide better support for the entire software development process of interactive systems, including the interface and the underlying system. We analyzed the prevailing trend towards separating interfaces from the rest of the software, and the widespread use of MVC and other models to do just that. Then we proposed a number of scenarios to highlight invisible and inseparable interactions between interfaces and the rest of the system, and we looked at some negative consequences of separating them. We showed how patterns can be used to shed new lights on this area and help designers avoid this impact on the overall quality of using the system.

From this we conclude that in interactive systems –as commonly promoted- complete separation of interface design from the rest of the system clearly has

negative effects on the quality of using the system. However, this separation is inevitable to help mitigate the complexity of design. Patterns can help in improving communication between designers of interfaces and those of the underlying system. This warrants further research to collect more evidences, analyze them and study their effect.

To further highlight the role of patterns and the dynamics of their reuse in current interface design practices, we investigated model-based design approaches. These are promising solutions to the difficulties associated with today's highly interactive, mobile and adaptive interfaces. We analyzed the models commonly used and provided some potential for improvements by reconciling existing model-based approaches and patterns.

From these domain studies, we conclude that the potential of applying patterns in interface design is evident and strong. However, pattern reuse in the interface design processes has always been local to a small set of patterns or few pattern collections, and carried out only by few expert researchers and industrial gurus. This defeats an important goal of patterns, namely disseminating experiences to a wide range of designers to readily reuse them. Our survey indicated that interface designers are familiar with only few HCI patterns and apply them manually. With the growing number of HCI patterns, this approach is not scalable to handle the large number of patterns available, and is not spreading to the majority of programmers as originally intended.

Our investigations also lead to the finding that current ineffectiveness in pattern reuse can be attributed to the way they are represented to interface designers. Pattern authors write down immense amount of valuable knowledge in text formats; only usable by human readers. Humans do have superior ability to read text and understand its contents, meaning and perform sophisticated reasoning about it; only in limited amounts. When we reformulate patterns in a machine readable format, computers can come in handy to greatly increase the scalability

of pattern reuse. While many activities will still depend on human decisions, we can apply several rule-based actions on patterns automatically; hence process many more patterns in much shorter time. We demonstrated the idea using several case studies and models.

2 Pattern Dissemination

We investigated the deficiencies associated with the current dissemination methods of HCI patterns. We propose a comprehensive process to support it and collect essential activities within one approach. Besides the proposal, we further developed it into a comprehensive system that offers a concrete development environment to developers.

A database for pattern has shown to be beneficial and essential in our dissemination approach. We showed the added advantages of using our machine readable format when retrieved from a central or local database. Software tools have direct access to the database and different algorithms can be applied. Using model driven architecture (www.omg.org/mda), we designed the system architecture and the supporting models and activities from the problem domain perspective (the user domain). Then we transferred this system into the solution domain and implemented it as a working software to provide the necessary infrastructure.

The results are attractive and encouraging, as seen in workshop discussions and publications. The hierarchical pattern model GPM so developed covers a broad range of patterns in a consistent way. The solution system is implemented as a concrete pattern type (the Generic Pattern Type, GPT) with all the necessary implementation details; again in multiple steps to separate the solution from different programming paradigms and platform details. The modularity of the system multi-tier architecture and the separation between the process and the

models allows for flexibility in implementing it in different ways ranging from a local PC system to a Web-based large scale application.

In the end, text patterns are available as program objects for storage, processing and tool support in three environments: object oriented programming, XML space and relational model. Each of the three environments has great opportunities and flexibility in processing patterns and adding value to their contents, and they are all needed for our comprehensive pattern support, the **I**ntegrated **P**attern **E**nvironment IPE. As planned from the beginning, the IPE in non-intrusive in the sense that it does not lose the original contents or format of patterns. Patterns within the IPE remain available in their original text form besides the other environments.

3 Limitations

Validating the concept of HCI patterns as a means of capturing and disseminating design knowledge has not yet been proven in an objective and tangible manner within the HCI community. To date, only some benefits have been highlighted and studied.

It takes several years and many large experiments and concrete applications to sufficiently demonstrate the usefulness of HCI patterns beyond doubts. These studies are certainly beyond the scope of this thesis. However, we do believe that the proposals made in this thesis pave the road for a more disciplined process. We have shown that creating patterns is not enough to warrant their reuse. We highlighted the cognitive load associated with their reuse and proposed another way to alleviate this obstacle. Like many other proposals and frameworks, it takes years before we can measure with confidence its acceptance level within the community.

Future Work

The new models presented in this thesis are interesting from a performance perspective. There is a need to expand the work in different directions.

1 Enriching the System

On the manual side, it takes enormous effort and time to compare patterns and detect redundancies and inconsistencies. We demonstrated the kind of work needed in this regard by finding tens of patterns that are identical or have many parts in common. We also provided algorithms to define different types of redundancies (appendix A). Many more cases are out there, and some are already known; only to experts. They just don't know what to do about them. We provided the framework needed to do the next logical step of registering this valuable knowledge about pattern similarities, redundancies and other relationships and present them to the user the same way we present patterns to them. This can be done within our framework, which is designed for that. It can save users a lot of confusion and help them navigate through patterns in an informed way.

On the tool side, we also presented examples of algorithms that work with our framework. Parallel to essential manual work we explained in Structured Expert Support, we provided several tools to help in pattern lookup (the dissemination process) as well as pattern reuse within the design process (the assimilation process).

We provided algorithms to help find identical and similar patterns. The effectiveness of these algorithms can be greatly improved by extending the "eXtensible Minimal Triangle, XMT". A rich repository of standardized keywords for the problem, context and solution can improve the quality of the provided algorithms. Similar work and advances in standardization of categories and

services have been made in other domains like Web Services and library systems. Few categories (like book category and reader's level) with only limited number of keywords allow for a successful categorization of millions of books for generic search besides search by author's name or title. The same can be extended to the XMT repository. Once a rich enough repository of standardized keywords is built, the provided algorithms will return more significant results for lookups. Algorithms allow for writing software that do useful job for us, in a much higher capacity. More algorithms can be added to the system.

In the same regard, more categorization work needs to be added to patterns. Several efforts have been recently done on categorizing patterns as shown in the thesis, but like other valuable research; they end up buried somewhere. The framework we are providing supports an environment that collectively includes many categorizations within one pattern model. Existing categories need to be collected and added as pointers within each pattern. As explained, the framework allows for the same pattern to belong to different categorization schema and collections. This can enhance reuse because a design process normally look for patterns based on well known categories among other lookup criteria. We also provided an example of a new categorization criterion based on the fact that some existing patterns are only abstractions of other patterns. A user might get confused thinking that they are just different patterns. The "*Progressive Abstraction Type Hierarch, PATH*" gives some examples of these common abstractions. More patterns need to be recognized according to this model. Each of the stages in the hierarchy is linked to a corresponding design phase which will help reduce the number of patterns looked up in each stage of the design.

2 Long Term Vision

One of our main goals for generic dissemination was to completely decouple pattern representation from its reuse. We defined dissemination and assimilation as two different processes and we separated them in our system. Previous works

have strongly coupled few patterns to one assimilation process, and provided simple examples to demonstrate how they work together. The problem with the strong coupling is that the given examples were fully aware of what they needed, and they had previous information about the few patterns they were looking for, defeating one of the main purposes of patterns, namely the dissemination of "new knowledge". On a large, general corpus of patterns within the HCI or any other community, the strong coupling approach proved less useful; most of the patterns outside the examples were ignored because they were "*not visible*", Even when some patterns were found, they simply "did not belong". The "known" patterns were fit manually, again reducing the chance of new tools to work on patterns at large.

Once this problem was identified, our goal from the beginning was to generalize the concept of patterns and to represent them in a generic way, independent of any specific design process or example. This is similar –in a way- to providing a programming language independent of any specific applications that can be written with it. That said, we provided some examples and tools to work with our generic approach. More design methods need to be able to call the generic model and look for patterns in the database according to their specific needs and to the semantics of the generic model. This is not a simple task, but we have demonstrated that it is not impossible. And, we believe this is a better way to approach patterns at large.

Reconciling Usability and the Architecture Models

In Chapter 3 we focused on specific ways in which internal software properties can have an impact on usability criteria. We provided several examples to validate our claim and show some negative consequences of decoupling interfaces from the underlying systems and how to remedy them. We identified them as scenarios and then represented them as patterns.

We provided a more general and theoretical framework for the relationships between usability and invisible software attributes. We explored whether there are specific places where we are more likely to find these relationships or effects. More places can definitely be identified and added. The goal of the framework is to define these patterns as a relationship between software quality factors and usability factors.

We also explored the relationship and the mutual interaction between the usability model and the architecture model of software systems. We simplified and approximated it to be able to demonstrate our point. The relationship between the two models is indeed important but complex. It warrants additional research to identify more of its effects.

From Code Fragments to Implementation Strategies

There is usually more than one way to implement a specific pattern in different software systems. Furthermore, given the wide variety of user interface styles and development platforms, each pattern implementation can exist in various formats. For example, the Web *Convenient Toolbar* pattern that provides direct access to frequently used pages such as *What's New*, *Search*, *Contact Us*, *Home Page*, and *Site Map*, can be implemented differently for a Web browser and a Personal Digital Assistant (PDA). For a Web browser, it can be implemented as a toolbar using embedded scripts or a Java applet in HTML. For a PDA, this pattern is better implemented as a combo box using the Wireless Markup Language (WML). It becomes more convenient due to the PDA-related limitations like screen area, bandwidth, memory and processor speed.

Empirical studies and day-to-day observations show that HCI designers and software developers have no trouble translating a well-documented usability pattern into a program block [FMW97] and [BFVY96]. It has been reported that designers occasionally attempt to reinvent patterns by producing multiple

implementations. Changing a design could translate to extensive reimplementation because different design choices in the pattern can lead to vastly different programs.

To avoid the problem of reinvention, pattern authors and software developers often include code samples of each pattern using several concrete solutions to aid in the reuse. We can take a new approach by removing implementation details from pattern documentation itself and by adding a strategy for pattern application. This strategy is assigned the following properties:

- A strategy defines one or more roles that may be mapped to concrete components and their elements
- A strategy provides a mechanism for constraining which components and elements may fill each role
- A strategy define one of many possible implementations of a pattern solution
- Patterns are often composed of other patterns. A strategy addresses this "pattern nesting" by being composed of other strategies. This scalability allows the description of a complete pattern-oriented design.

A strategy is not required to be associated with one specific pattern. Again, decoupling them is a key factor. A strategy may instead serve to define other strategies. Each pattern may be associated with multiple strategies, each of which defines one implementation of a pattern solution. A pattern is not directly aware of its strategies because we do not wish to limit the number of strategies available and the association of them to a pattern at just creation time. It is conceivable that people other than the original author may later discover new strategies for implementing a pattern.

Bibliography

[ACM05] ACM technews "Viruses, Security Issues Undermine Internet" Volume 7, Issue 809, June 27, 2005.

[AIS77] Alexander, C., Ishikawa, S., and Silverstein, M., "A Pattern Language: Towns, Buildings, Constructions", Oxford University Press publishing, New York, NY, USA, 1977.

[Alx70] Alexander, Christopher. "Notes on the Synthesis of Form", Harvard University Press publishing, Cambridge, Massachusetts. USA, 1970.

[Alx79] Alexander, Christopher. "The Timeless way of building", Oxford University Press publishing, New York, NY, USA, 1979.

[AM90] Ackerman, Mark S. and Thomas W. Malone. Answer Garden: A tool for growing organizational memory. Proceedings of the ACM Conference on Office Information Systems, April 25-27, 1990, Cambridge, Massachusetts, USA, ACM Press publishing New York, NY, USA, pp. 31-39,1990.

[AMBC98] Astrachan, Owen; Mitchener, Garrett; Berry, Geoffrey and Cox, Landon. Design patterns: an essential component of CS curricula. Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education, Atlanta, Georgia, USA, p.p.153-160, February 26-March 01, 1998.

[AMC03] Alur, Deepak; Malks, Dan and Crupi, John. Core J2EE Patterns: Best Practices and Design Strategies. Sun Microsystems Core Design Series, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[App00] Appleton, B. Patterns and software: essential concepts and Tterminology, 2000, retrieved May 7, 2003 from http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html.

[Art00] Arutla, K. Tool support for pattern oriented analysis and design. Master thesis, Department of Computer Science and Electrical Engineering. University of West Virginia, Morgantown, West Virginia, 2000.

[Bac73] Charles W. Bachman. The programmer as navigator, Communications of the ACM, volume 16, No. 11, pp 635-658, November 1973.

[Bal96] Balzert, H. From OOA to GUIs: The JANUS System. Journal of Object-Oriented Programming, pp. 43-47, February 1996.

[BBC+03] Bevan, N.; Barnum, C.; Cockton, G.; Nielsen, J.; Spool, J.; and Wixon, D. The "magic number 5": is it enough for Web testing? Proceedings of CHI '03, International Conference on Human Factors in Computing Systems, extended abstracts, April 5-10, 2003, Ft. Lauderdale, Florida, USA, volume 5, issue #1, pp. 698 – 699, 2003.

[BCC+96] Beck, K.; Coplien, J. O.; Crocker, R.; Dominick, L.; Meszaros, G.; Paulisch F.; et al. Industrial experience with design patterns. Proceedings of the 18th International Conference on Software Engineering, IEEE Computer Society Press publishing, 1996.

[BCC+02] Butler G., Chen L., Chen X., Gaffar A., Li J, Xu L. *The Know-It-All project: A case study in framework development and eEvolution, Chapter 6 in "Domain oriented systems development: perspectives and practices", Kiyoshi Itoh, Satoshi Kumagai (eds.),* Taylor & Francis; 1st edition, December 6, 2002*, UK, 2002,* ISBN 0-415-30450-4, pp. 101-117.

[BCK98] Bass, Len; Clements, Paul; and Kazman, Rick. Software architecture in practice, Addison-Wesley publishing, Reading, MA, USA, 1998.

[BCK03] Bass, Len; Clements, Paul; and Kazman, Rick. Software architecture in practice, Second edition, Addison-Wesley publishing, Reading, MA, USA, 1998.

[BF00] Berners-Lee, Tim and Fischetti, Mark. Weaving the web: The original design and ultimate destiny of the World Wide Web. Harper Collins Canada publishing, Scarborough, ON, Canada; 2000.

[BFVY96] Budinsky, F.; Finnie, F.J.; Vlissides, J.M. and Yu, P.S. Automatic code generation from design patterns. Journal of *Object Technology*, volume 35, issue No.2, 1996.

[BH99] Brinck, T. and Hand, A.: What do users want in an HCI Website. EACE Quarterly (European Association of Cognitive Ergonomics) volume 3, issue no. 2, August 1999.

[BJK01] Bass, L.; John, B. E.; and Kates, J. Achieving usability through software architecture. SEI, Software Engineering Institute report, Carnegie Mellon University, Pittsburgh, PA, USA, March 2001.

[BKB00] Bouch, Anna; Kuchinsky, Allan; Bhatti, Nina. Quality is in the eye of the beholder: meeting users' requirements for Internet quality of service. Proceedings of the SIGCHI conference on Human factors in computing systems. The Hague, The Netherlands, ACM Press publishing, New York, NY, USA, pp. 297–304, 2000.

[BMMM98] Brown, W. J.; Malveau, R.; MCCormick, H. W.; and Mowbray, T. J. Anti Patterns - Refactoring Software, Architectures and Projects in Crisis: Wiley publishing, Chichester, West Sussex, England, 1998.

[BMR96] Buschmann. F.; Meunier, H.; Rohnert, P. S. and Stal M. Pattern-oriented software architectures: A system of patterns, John Wiley publishing, Chichester, West Sussex, England, 1996.

[Bor00] Borchers, Jan. A pattern approach to interaction design. Proceedings of the DIS 2000; international conference on designing interactive systems, August 17-19, 2000, New York, New York, USA. ACM Press publishing New York, NY, USA, pp. 369–378, 2000.

[Box79] Box, George. E. P. Robustness in scientific model building. In R. L. Launer, and G. N. Wilkinson (Eds.), Robustness in statistics, Academic Press publishing, New York. NY, USA, pp. 201-236, 1977.

[BPS97] Breedvelt, Ilse M.; Paternò, Fabio; and Severiins, C. Reusable structures in task models. Proceedings of DSVIS 97, Design, Specification, Verification of Interactive Systems, Granada, Spain, June 4-6 1997. Springer Verlag publishing, Granada, Spain pp. 251-265, June 1997.

[Cal02] Calder, Neil. SLAC Houses world's largest database. Stanford University Report, stanford University, Stanford CA USA, April17, 2002.

[Car00] Carroll J.M. Scenario-based design of human-computer interactions. MIT Press publishing, Boston, MA, USA, September 2000.

[Cas97] Casaday, G. Notes on a pattern language for interactive usability. Proceedings of CHI '97, the international conference on computer human interface, human factors in computing systems. 18-23 April 1997, Atlanta, Georgia, USA. ACM Electronic Publishing. Retrieved Mai 2, 2003 from http://www.acm.org/sigchi/chi97/proceedings/short-talk/gca.htm.

[CBE+04] Cockburn, A.; Baruz, A.; Engelund, A.; HANES, P. B. et al. Anti Pattern. At http://c2.com/cgi/wiki?AntiPattern, retrieved May 20th 2004.

[CC01] Collins Cobuild English Dictionary, Harper Collins Publishing, Bishopbriggs, Glasgow, UK, 2001, ISBN 0-00-710201-1.

[CDI+04] Challenger, James R. ; Dantzig, Paul ; Iyengar, Arun ; Squillante, Mark S. and Li Zhang. Efficiently serving dynamic data at highly accessed web sites, IEEE/ACM Transactions on Networking (TON), ACM Press publishing, New York, NY, USA, Volume 12 , Issue 2, pp. 233 – 246, April 2004.

[CHV00] Chambers, Craig; Harrison, Bill and Vlissides, John. A debate on language and tool support for design patterns. POPL 2000, 27th ACM conference on Principles of Programming Languages, January 19-

21, 2000, Boston, MA, USA. ACM Press publishing, pp 277-289, 2000

[CL98] Coram, T. and Lee, J. Experiences: A pattern language for user interface design., 1998 Retrieved Mai 2nd, 2003 from http://www.maplefish.com/todd/papers/experiences

[CL99] Clancy, Michael J. and Linn, Marcia C. Patterns and pedagogy. Proceedings of the thirtieth SIGCSE technical symposium on computer science education, New Orleans, Louisiana, United States, ACM Press publishing, New York, NY, USA, pp 37 – 42, 1999.

[CMMI02] Capability Maturity Model Integration, version 1.1 (CMMI SE/SW/IPPD/SS, V1.1), SEI; Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, March 2002.

[Coc01] Cockburn, Alistair. Agile Software Development, 1 edition, Addison-Wesley Professional publishing, Boston, MA, USA, 2001.

[CP96] Colleen M. K. and Pitkow J. E. Surveying the Territory: GVU's Five WWW User Surveys, The World Wide Web Journal, CA, USA, volume 1, no. 3, pp 77-84, 1996,.

[Cra43] Craik, K. *The Nature of Explanation*: Cambridge University Press, Cambridge, Massachusetts. USA, 1943.

[CT04] OliverNova. CARE Technologies. Retrieved January, 2004, from http://www.care-t.com/.

[DFG99] Dewan, Rajiv; Friemer, Marshall; Gundepudi, Pavan. Evolution of internet infrastructure in the twenty-first century: the role of private

interconnection agreements. Proceeding of the 20th international conference on Information Systems, Charlotte, North Carolina, USA, ACM digital Library, Association for Information Systems Publishing, Atlanta, GA, USA, pp. 144 – 154, 1999.

[Dia03] Diamond Bullet, Website Design: Web Survey Results, at http://www.usabilityfirst.com/websites/web-survey-results.txl, Retrieved March 21st, 2003.

[DLH03] van Duyne, D. K.; Landay, J. A. and Hong, J. I. The design of sites. patterns, principles, and processes. Pearson Education, Addison-Wesley publishing, Boston, MA, USA, 2003

[DNB02] Dhyani, Devanshu; Ng, Wee Keong and Bhowmick, Sourav S. A survey of Web metrics, ACM Computing Surveys (CSUR), ACM Press publishing, New York, NY, USA, Volume 34, Issue 4 pp. 469 – 503, December 2002.

[Dre02] The Math Forum @ Drexel." http://www.mathforum.org/drexel/, retrieved June 23rd, 2002.

[DZS97] DeLine, Robert; Zelesnik, Gregory and Shaw, Mary. Lessons on converting batch systems to support interaction: experience report. ICSE 97, proceedings of the 19th international conference on Software Engineering, Boston, Massachusetts, USA, ACM Press publishing, New York, NY, USA, 195 – 204, 1997.

[Erc00] Erickson, T. Lingua franca for design: Sacred places and pattern languages, DIS 2000: Proceedings of Designing Interactive Systems, New York, August 17-19 2000, ACM Press publishing, New York, NY, USA, 2000.

[ET97] Erickson, T.; and Thomas, J. C. Putting it all together: Towards a pattern language for interaction design: a Workshop on Pattern Languages, in CHI97, March 22-27, 1997. ACM SIGCHI, ACM press publishing, New York, NY, USA, volume 30, issue 1, pp. 17-23, January 1998.

[FB02] Folmer, E. and Bosch, J. Architecting for usability; a survey. Journal of systems and software, Reed Elsevier publishing, Orlando, Fl, USA, Volume 70, issue 1, pp. 61- 78, 2002.

[FDRS03] Forbrig, P.; Dittmar, A.; Reichart, D.; and Sinnig, D. User-centred design and abstract prototypes. In proceedings of BIR 2003. Perspectives in business informatics research, September 19-20, 2003, Berlin, Germany. SHAKER publishing, 2003, pp. 132-145, September 2003.

[FF03] Fincher, S. and Finlay, J. CHI 2003 report: Perspectives on HCI patterns: Concepts and tools; introducing PLML. Interfaces, the international journal of human computer interaction, British HCI Group publishing, Winchester, Hampshire, UK, volume 56, pp. 26-28, Autumn 2003.

[FFS+04] Furtado, Elizabeth; Furtado, Vasco; Sousa, Kênia Soares; Vanderdonckt, Jean; Limbourg, Quentin. KnowiXML: a knowledge-based system generating multiple abstract user interfaces in USIXML Proceedings of TAMODIA '04; the 3rd annual conference on task models and diagrams, Prague, Czech Republic ACM Press publishing, New York, NY, USA, pp. 121 – 128, 2004

[Fin02] Fincher, S. Patterns for HCI and cognitive dimensions: Two halves of the same Story ? PPIG 14th Annual Workshop, Brunel University, 18-21 June 2002, In J. Kuljis, L., Baldwin & R. Scoble (Eds) pp 156-172, 2002..

[Flo87] Floyd, C. Outlines of a paradigm change in software engineering. In Bjerknes, G.; Ehan, P. and Kyng, M (Eds.), Computers and Democracy Avebury, England. Gower publishing co. Ltd., Aldershot, England, pp. 193-210, 1987.

[FMW97] Florijin, G.; Meijers, M. and van Winsen, P. Tool support in design patterns., Proceedings of ECOOP '97, 11th European Conference on Object-Oriented Programming, Utrecht University, Jyväskylä, Finland, June 9-13 1997. In: Askit, M., Matsuoka, S. (Eds.) Lecture Notes in Computer Science no. 1241 Springer Verlag publishing, Berlin, Heidelberg, Germany, 1997.

[FR82] Feldman, Michael B. and Rogers, George T. Toward the design and development of style-independent interactive systems. Proceedings of the 1982 conference on Human factors in computing systems. NBS, National Bureau of Standards and ACM Washington DC Chapter, Gaithersburg, Maryland, United States. ACM Press publishing, New York, NY, USA, pp. 111 – 116, 1982.

[Gaf01] Gaffar, Ashraf. Design of a framework for database indexes. Master of Computer Science Thesis, Computer Science Department, Concordia University, September 2001.

[Gaf04] Gaffar. Ashraf. The other side of patterns: A user-centered analysis. Preliminary results of Pattern Usability Study, presented at UPA: Usability Professionals' Association, Bloomingdale, Illinoy, USA in

conjunction with CRIM (Computer Research Institute of Montreal), February 2004, available at http://www.utilisabilitequebec.org/archives.htm.

[Gaf05a] Gaffar, Ashraf. Component-Based Generalized Database Index Model, Encyclopaedia of Database Technologies and Applications, Idea Group Publishing, Hershey, PA, USA, April 2005, ISBN 1-59140-560-2, pp. 87-92.

[Gaf05b] Gaffar, Ashraf. The 7C's: An iterative process for generating pattern components. Proceedings of HCI International, the 11th International Conference on Human-Computer Interaction, Las Vegas, Nevada, USA, July 22-27, 2005, CD-ROM.

[Gaf05c] Gaffar, Ashraf. Des aiguilles dans une botte de foin: Une étude sur la visibilité de L'information sur Internet. Final results of Pattern Usability Study, presented at UPA: Usability Professionals' Association, Bloomingdale, Illinois, USA in conjunction with CRIM (Computer Research Institute of Montreal), March 2005, available at http://www.utilisabilitequebec.org/archives.htm.

[GHJV95] Gamma, E.; Helm, R.; Johnson, R. and Vlissides, J. Design patterns: Elements of reusable object-oriented software, Addison-Wesley-Longman publishing, MA, USA, 1995.

[GJSS03] Gaffar, Ashraf; Javahery, H; Seffah, Ahmed. and Sinnig, Daniel. A pattern framework for eliciting and delivering user centered design knowledge and practices, proceedings of HCI International, the 10th International Conference on Human-Computer Interaction '03, Crete, Greece, June 24-27, 2003. Appeared in Julie Jacko and Constantine Stephanidis, Human Computer Interaction: Theory and Practice,

Lawrence Erlbaum publishing, vol. 1, pp. 108-112, 2003, ISBN 0-805-84930-0

[GL99] Granlund, A. and Lafreniere, D. PSA: A pattern-supported approach to the user interface design process, Position paper for the Usability Professionals Association Conference, June 29-July 2, 1999, Scottsdale, Arizona, USA.

[GMS05] Gaffar, Ashraf; Moha, Naouel and Seffah, Ahmed. User-centered design process management and communication, Proceedings of HCI International, The 11th International Conference on Human-Computer Interaction, Las Vegas, Nevada, USA, July 22-27, 2005, CD-ROM.

[GP04] Goldfarb, C. F. and Prescod, P. The XML Handbook, 5th edition. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[GS05] Gaffar, Ashraf and Seffah, Ahmed. An XML multi-tier pattern dissemination system, Encyclopaedia of Database Technologies and Applications, Idea Group Publishing, Hershey, PA, USA, April 2005, ISBN 1-59140-560-2, pp. 740-744.

[GSJS03] Gaffar, Ashraf; Sinnig, Daniel; Javahery, H. and Seffah, Ahmed. MOUDIL: A comprehensive framework for disseminating and sharing HCI patterns. Short paper in CHI 2003 Workshop: Perspectives on HCI Patterns; Concepts and Tools, Ft. Lauderdale, Florida, USA, April 6-7, 2003.

[GSP05] Gaffar, Ashraf; Seffah, Ahmed, and van der Poll, John. HCI patterns semantics in XML: A pragmatic approach, Proceedings of HSSE '05, International workshop on Human and Social Factors of Software

Engineering, in ICSE 2005, the 27th International Conference on Software Engineering, St. Louis, Missouri, USA, May 15-21, 2005, ACM Press, New York, NY, USA, 2005, pp. 1-7.

[GSS03] Gaffar, Ashraf; Sinnig, Daniel; and Seffah, Ahmed. Towards a universal forum for patterns, CUSEC '03, Canadian Universities Software Engineering Conference, Montreal, Canada, January 16-18, 2003.

[GSSF04] Gaffar, Ashraf; Sinnig, Daniel; Seffah, Ahmed and Forbrig, Peter. Modeling patterns for task models. Proceedings of TAMODIA, 3rd International Workshop on Task Models and DIAgrams for user interface design, Prague, Czech Republic, November 15-16, 2004, ACM Press, New York, NY, USA, pp. 99-104.

[GWC+00] Graham, T. C. Nicholas; Watts, Leon A.; Calvary, Gaëlle; Coutaz, Joëlle; Dubois, Emmanuel and Nigay, Laurence. A dimension space for the design of interactive systems within their physical environments. Proceedings of the conference on designing interactive systems: processes, practices, methods, and techniques, New York, NY, USA. ACM Press publishing, New York, NY, USA, pp. 406 – 416, 2000.

[HCG+96] Hewett, Thomas T.; Baecker, Ronald; Card, Stuart; Carey, Tom; Gasen, Jean et al. ACM SIGCHI Curricula for Human-Computer Interaction, ACM Special Interest Group on Computer-Human Interaction Curriculum Development ACM Press publishing, New York, NY, USA, 1996, ISBN 0-89791-474-0.

[HCI02] Human-Computer Interaction encyclopedia article, history, biography http://encyclopedia.localcolorart.com/encyclopedia/Human-computer_interaction/ Retrieved March 6, 2002.

[HGV02] Hoffer, J. A.; George, J. F. and Valacich, J. S. Modern system analysis and design. 3rd edition, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2002.

[Hil 93] The Hillside Group, formed in 1993." http://hillside.net/, retrieved June 23, 2002.

[HL04] Jason I. Hong and James A. Landay An architecture for privacy-sensitive ubiquitous computing, Proceedings of the 2nd international conference on Mobile systems, applications, and services, Boston, MA, USA, ACM Press publishing, New York, NY, USA, pp. 177–189, 2004.

[HR02] Horrigan, John B. and Rainie, Lee. Getting serious online. Pew Internet and American Life Project, Washington D.C., USA, retrieved March 3rd, 2002, www.pewinternet.org/Pdfs/pip_Getting_Serious_Online3ng.pdf

[JGL98] Johnson-Laird, P. N.; Girotto, V. and Legrenzi, P. Mental models: a gentle guide for outsiders, 1998. Retrieved 17 April, 2004, from www.si.umich.edu/ICOS/gentleintro.html

[JTV00] Jarvenpaa, Sirkka L.; Tractinsky, Noam and Michael Vitale. Consumer trust in an Internet store information Ttechnology and management, Kluwer Academic Publishing, Hingham, MA, USA, Volume 1, Issue 1-2, pp. 45 – 71, 2000.

[KB00] Kosala, Raymond and Blockeel, Hendrik. Web mining research: A survey. ACM SIGKDD Explorations Newsletter, ACM Press publishing, New York, NY, USA, Volume 2, Issue 1, pp. 1 – 15, June 2000.

[Lay04] LayNetworks Glossary, at http://www.laynetworks.com/glossary/u.htm, Retrieved march 3rd, 2004.

[LC98] Lending, Diane and Chervany, Norman L. The use of CASE tools. Proceedings of the 1998 ACM SIGCPR conference on computer personnel research, Boston, Massachusetts, USA, ACM Press publishing, New York, NY, USA, pp. 49 – 58, 1998.

[Lin80] Ling, Robert F. General considerations on the design of an interactive system for data analysis. Communications of the ACM, ACM Press publishing, New York, NY, USA, Volume 23, Issue 3 pp. 147–154, March 1980.

[LM95] Landay, James. A. and Myers, Brad. A. Interactive sketching for the early stages of user interface design. In proceedings of CHI 95, vol.1 pp. 43-50, 1995.

[LNHL00] Lin, J. M.; Newman, W.; Hong, J.I.; and Landay, J. A. DENIM: Finding a tighter fit between tools and practice for web site design. CHI letters: Human factors in computing systems, CHI '00, pp. 510-517, 2000.

[Luc04] Robert W. Lucky. Does Google like me? IEEE Spectrum: Tomorrow's technology today, pp 136, November 2004.

[LV03] Lyman, Peter and Varian, Hal R. How Much Information 2003?, School of Information Management and Systems, University of California at Berkeley, Retrieved on October 2004 from http://www.sims.berkeley.edu/how-much-info-2003.

[Mar96] Märtin, C. Software life cycle automation for interactive applications: The AME design environment. Proceedings of CADUI '96, 2nd International Workshop on Computer-Aided Design of User Interfaces June 5-7, 1996, Namur, Belgium.

[May05] Maynes-Aminzade, Dan. Edible Bits: Seamless interfaces between people, data and food. Proceedings of CHI '05, the international conference for human-computer interaction, April 2-7, Portland, Oregon, USA. ACM Press publishing, New York, NY, USA, 2005.

[MB02] Mellor, S. J. and Balcer, M. J. Executable UML: A foundation for Model-Driven Architecture, 1st edition. Addison-Wesley Professional publishing, Boston, MA, USA, 2002.

[Mcc04] McConnell, Steve. Code Complete, 2nd edition, Microsoft Press publishing, Redmond, WA, USA, 2004.

[MD97] Meszaros, G. and Doble, J. A pattern language for pattern writing, 1997. Retrieved Mai 5th, 2003 from http://hillside.net/patterns/writing/patternwritingpaper.htm

[MJ98] Mahemoff, M. J. and Johnston, L. J. Pattern languages for usability: An investigation of alternative approaches, Proceedings of APCHI, third Asia-Pacific Conference on Human Computer Interaction, July 15-17, Hayama-machi, Kanagawa, Japan,1998.

[MJ99] Mahemoff, M. J. and Johnston, L. J. The planet pattern language for software internationalization. Proceedings of PLoP '99 annual conference on Pattern Languages of Programs, August 15-18, University of Illinois at Urbana-Champaign, Urbana, IL, USA.

[ML01] Mahemoff, M. and Lorraine J. J. Usability pattern language: The language aspect. In Hirose M. (ed.), Human Computer Interaction: Interact '01, Proceedings of IFIP TC.13 International Conference on Human-Computer Interaction, July 9-13, 2001, Tokyo, Japan. Ohmsha publishing, Tokyo, Japan, pp. 350-358, 2001.

[MLS98] Mens, T.; Lucas, C. and Steyaert, P. (1998,). Supporting disciplined reuse and evolution of UML models. UML98: Beyond the notation, June 3-4, Mulhouse, France, 1998.

[MM97] Mendelzon, Alberto O. and Milo, Tova. Formal models of Web queries. SIGMOD: ACM Special Interest Group on Management of Data. Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems Tucson, Arizona, USA, ACM Press publishing, New York, NY, USA, pp134 – 143, 1997.

[MMP02] Molina, P. J., Melia, S., & Pastor, O. (2002). Just-UI: A Model for User Interface Specifications. In Proce. CADUI 2002, Valenciennes, France, Kluwer Academics, May 2002.

[MOB99] MOBI-D. The MOBI-D interface development environment. Retrieved February, 2004, from http://smi-web.stanford.edu/projects/mecano/mobi-d.htm

[MQGS05] Moha, Naouel; Qing, Lin; Gaffar, Ashraf and Seffah, Ahmed. Enquête sur les pratiques de tests d'utilisabilité, IHM 2005, 17e Conférence Francophone sur l'Interaction Homme-Machine, 27 au 30 Septembre 2005, Toulouse, France.

[MR92] Myers, B. A., and Rosson, M. B. Survey on user interface programming. Human factors in computing systems, Proceedings of SIGCHI' 92, Monterey, Ca., USA. ACM Press publishing, New York, NY, USA, pp. 195-202, 1992.

[MSG03] Metzker, Edward; Seffah, Ahmed and Gaffar, Ashraf. Towards a systematic and empirical validation of HCI knowledge captured as patterns. Proceedings of HCI international, the 10th international conference on Human-Computer Interaction 2003, Crete, Greece, June 22-27, 2003, appeared in Julie Jacko and Constantine Stephanidis. Human Computer Interaction: Theory and Practice, Lawrence Erlbaum publishing, vol. 1, pp. 168-172, 2003.

[MSS+ 04] McNaughton, M. C. M.; Szafron, D.; Schaeffer, J.; Redford, J.; Parker, D. ScriptEase: Generative design patterns for computer role-playing games. Proceedings of 19th International Conference on Automated Software Engineering, Linz, Austria, September 20-24, 2004. IEEE Computer Society Press publishing, Washington DC, USA, 2004.

[NEC98] NEC Research Institute. September 1998 search engine coverage update, September 1998. Retrieved February 18th, 2003, from http://www.neci.nj.nec.com/homepages/lawrence/websize98.html.

[New97] Newman, William M. Better or just different? On the benefits of designing interactive systems in terms of critical parameters.

Proceedings of the conference on designing interactive systems: processes, practices, methods, and techniques Amsterdam, The Netherlands, ACM Press publishing, New York, NY, USA, pp. 239 – 245, 1997.

[Nie93] Nielsen, J., and Landauer, T. K. A mathematical model of the finding of usability problems. Proceedings of INTERCHI '93, joint conference of ACM SIG-CHI and INTERACT, April 24-29, 1993, Amsterdam, The Netherlands. ACM Press publishing, New York, NY, USA, pp. 206-213, 1993.

[Nie04] Nielsen, J. Risk of quantitative studies. Alertbox, March 1 2004 issue, retrieved July 20th, 2004 from http://www.useit.com/alertbox/20040301.html.

[NL95] Newman W. and Lamming M. G. Interactive System Design, Addison-Wesley Professional publishing, Boston, MA, USA, 1995.

[Non98] Nonaka, Ikujiro and Takeushi, Hirotaka. A theory of the firm's knowledge-creation dynamics. Chapter 10 in: Chandler jr. Alfred D.; Hagstrom, Peter and Sovell, Orjan. O. (eds) The dynamic firm: The role of technology, strategy, organization and regions. Oxford University Press, Corby, Northants, UK, April 1998.

[Nor02] Norman, D. Beyond the computer industry CACM, Communications of the ACM, ACM Press Publishing, New York, NY, USA, volume 45, issue 7, pp. 120, July 2002.

[Pat94] Paternò, Fabio. A formal approach to the evaluation of interactive systems, ACM SIGCHI Bulletin, ACM Press publishing, New York, NY, USA, volume 26 , issue 2, pp 69 – 73, April 1994.

[Pat00] Paternò, Fabio. Model-based design and evaluation of interactive applications: Springer Verlag publishing, Berlin Heiderlberg, Germany, 2000.

[PB03] Podlipnig, Stefan and Böszörmenyi, Laszlo. A survey of Web cache replacement strategies, ACM Computing Surveys (CSUR), ACM Press publishing, New York, NY, USA Volume 35 , Issue 4, pp. 374 – 398, December 2003.

[PC96] Pitkow J. E. and Colleen M. K: Emerging trends in the WWW user population. Communications of the ACM, volume 39, no. 6, 1996

[PC04] Porter, Ron and Calder, Paul. Patterns in learning to program: an experiment. ACM International Conference Proceeding Series. Proceedings of the sixth conference on Australian computing education - Dunedin, New Zealand, Australian Computer Society publishing Inc., Darlinghurst, Australia, Volume 30, pp. 241 – 246, 2004.

[PG98] Pemberton, L. and Griffiths, R. The timeless way: Making living co-operative buildings with design patterns In co-operative buildings: Integrating information, organization, and architecture. Lecture Notes in Computer Science, Springer Verlag publishing, Darmstadt, Germany, February 1998.

[Por03] Portland pattern repository, survey results, retrieved on March 21st, 2003 at http://c2.com/cgi-bin/survey.

[Pra96] Prasad, S. Models for mobile computing agents. ACM Computer Survey, volume 28, issue 4, 1996.

[PRC03] Pandey, Sandeep; Ramamritham, Krithi and Chakrabarti, Soumen Monitoring the dynamic web to respond to continuous queries. Proceedings of the ACM 12th international conference on World Wide Web, Budapest, Hungary, SESSION: Web crawling and measurement, ACM Press publishing, New York, NY, USA, pp. 659 – 668, 2003.

[PRC04] Pew Research Center, Washington, DC, USA. http://pewresearch.org/

[Prs01] Pressman, Roger. S. Software Engineering, A practitioner's approach. fifth edition, McGraw Hill, New York, NY, USA, 2001.

[PS04] Paquette, D. and Schneider, K. Interaction templates for constructing user interfaces from task models. Proceedings of CADUI '04, international conference on computer aided design of user interface, January 13-16, 2004, Madeira, Portugal.

[Pue97] Puerta, A., A Model-based interface development environment. Journal of IEEE Software, volume 14, p. 41-47, 1997.

[RE96] Rheinfrank, J and Evanson, S. Design languages. In: Bringing design to software, Winograd, Terry (ed.), ACM Press books and Addison Wesley publishing, New York, NY, USA , pp 63-80, 1996.

[Rog99] Roget's 21st Century Thesaurus, Philip Lief Group, Inc., Dell Publishing, New York, NY, USA. ISBN: 0-440-23513-8.

[Rou81] Rouse, William B. Human-computer interaction in the control of dynamic systems, ACM Computing Surveys (CSUR) ACM Press

publishing, New York, NY, USA, Volume 13, Issue 1 pp 71 - 99 , March 1981.

[RN00] Rine, D. C. and Nada, N. Three empirical studies of a software reuse reference model. Journal of software practices and experiences, volume 30, issue 6, May 2000. John Wiley & sons, publishing, New York, NY, USA, pp. 685-722, 2000.

[Run93] Runde, Audun S. The challenge of the Internet explosion: ideas for staying ahead of the users. Proceedings of the 21st annual ACM SIGUCCS conference on user services, San Diego, California, USA. ACM Press publishing, New York, NY, USA, pp. 150 – 154, 1993.

[SAKS03] Seffah, A; Abran A.; Khelifi A. and Suryn W. Usability meanings and interpretations in ISO standards. Software Quality Journal, volume 11, issue 4, December, 2003.

[San01] Sandu D. User interface patterns. 8th conference on pattern languages of programs September 11-15, 2001 Allerton Park Monticello, Illinois, USA.

[SBG99] Schmidt, A.; Beigl, M.; and Gellersen, H.-W. There is more to context than location. Journal od Computers and Graphics, volume 23, pp. 893-902, 1999.

[Sch96] Schlungbaum, E. Model-based user interface software tools - current state of declarative models. Technical report of Graphics, Visualization and Usability Center, Georgia Institute of Technology, Georgia, USA, pp. 96-30, 1996.

[SEW02] Search Engine Watch, Report 1: Getting serious online, March 3rd, 2002, at http://www.searchenginewatch.com, retrieved on February 18th, 2003.

[SEW03] Search Engine Watch, Report 2: Getting serious online, One year later, September 5th, 2002, at http://www.searchenginewatch.com, retrieved on February 18th, 2003.

[SGFS04] Sinnig, Daniel; Gaffar, Ashraf; Forbrig, Peter and Seffah, Ahmed. Patterns, tools and models for interaction design, in Hallvard Trætteberg, Pedro J. Molina, Nuno J. Nunes (eds). "Proceedings of the first international workshop on making model-based user interface design practical: usable and open methods and tools". Joint conference of ACM intelligent user interfaces 2004 (IUI'2004) and computer aided and design of user interfaces 2004 (CADUI 2004). Funchal, Madeira, Portugal. January 13, 2004. CEUR-workshop proceedings, Vol. 103, 2004.

[SGR+04] Sinnig, Daniel; Gaffar, Ashraf; Reichart, Daniel, Forbrig, Peter and Seffah, Ahmed. Patterns in model-based engineering. In proceedings of CADUI 2004; 5th International conference on computer-aided design of user interfaces, jointly with ACM IUI 2004; International conference on intelligent user interfaces, Funchal, Madeira, Portugal, January 13-16, 2004, pp. 197 – 210.

[Sil00] da Silva, Paulo Pinheiro. User Interface Declarative Models and Development Environments: A Survey. In proceedings of 7th international conference DSV-ID; Design, specification, and verification of interactive systems. Limerick, Ireland, June 2000. In Palanque, Phillip, and Paterno, Fabio (eds). LNCS Vol. 1946,

Springer Verlag publishing, Berlin Heiderlberg, Germany, pp. 207-226, 2000.

[Sim62] Simon, H. The architecture of complexity. Proceeding of American philosophy society, volume 106, 1962.

[Smt70] Smith, Lyle B. A survey of interactive graphical systems for mathematics. ACM computing surveys (CSUR), ACM Press publishing, New York, NY, USA, Volume 2, Issue 4 pp 261-301, December 1970.

[Som01] Sommerville, Ian. Software Engineering. Sixth Edition, Pearson Education, Addison Wesley, Harlow, Essex, England, 2001.

[ST96] Smith, C. and Tabor, P. The role of art in design. In: Bringing design to software, Winograd, Terry (ed.), ACM Press books and Addison Wesley publishing, New York, NY, USA, pp 37-57, 1996.

[STL99] Smith, Jeffrey D.; Takahashi, Kenji and Liang, Eugene. Living Web: supporting Internet-based user-centered design. ACM SIGGROUP Bulletin, ACM Press publishing, New York, NY, USA, Volume 20, Issue 1 , pp. 45 – 50, April 1999.

[TAD98] TADEUS. Task-based development of user interface software, 1998. Retrieved February, 2004, from http://www.icg.informatik.uni-rostock.de/~schlung/TADEUS/

[TAW03] Titchkosky, Lance; Arlitt, Martin, and Williamson, Carey. A performance comparison of dynamic Web technologies, ACM SIGMETRICS performance evaluation review, ACM Press

publishing New York, NY, USA, volume 31 , issue 3 , pp. 2 – 11, December 2003.

[TER04] TERESA. Transformation environment for interactive systems representations. Retrieved February, 2004, from http://giove.cnuce.cnr.it/teresa.html

[Tid97] Tidwell, Jennifer. Common ground: A pattern language for human-computer interface design, 1997. Retrieved January 28th, 2003 from http://www.mit.edu/~jtidwell/common_ground.html

[Tra02] Traetteberg, Halvard. Model-based user interface design. Doctoral thesis at Norwegian University of Science and Technology, Faculty of Information Technology, Mathematics and Electrical Engineering, Trondheim, Norway. 2002.

[Tra04] Trætteberg, Halvard. Integrating dialog modeling and application development, Proceedings of the international conference on intelligent user interfaces 2004 (IUI 2004) and computer aided and design of user interfaces 2004 (CADUI 2004). Funchal, Madeira, Portugal. January 13-16, 2004.

[UCB00] UC Berkeley, Robust hyperlinks cost just five words each, at http://www.cs.berkeley.edu/~phelps/Robust/papers/robust-hyperlinks.html. Technical paper, web pages lexical code, January 2000.

[VCBT04] Vanderdonckt, Jean; Chieu, Chow Kwok; Bouillon, Laurent and Trevisan, Daniela. Model-based design, generation, and evaluation of virtual user interfaces Proceedings of the ninth international

conference on 3D Web technology, Monterey, California 2004, ACM Press New York, NY, USA, pp. 51 – 60, 2004.

[VLF03] Vanderdonckt, J.; Limbourg, Q.; and Florins, M. Deriving the navigational structure of a user interface. Proceedings of INTERACT '03, the ninth IFIP TC13 International Conference on Human Computer Interaction, September 1-5, 2005 Zurich, Switzerland, pp. 455-462, 2003.

[WBD01] Whitten, J. L.; Bentley, L. D.; and Dittman, K. C. System analysis and design methods, 5th edition, McGraw Hill Irwin, New York, NY, USA, 2001.

[Win96] Winograd, Terry. Bringing design to software, ACM Press books and Addison Wesley publishing, New York, NY, USA, pp. 63-80, 1996.

[WP05] Want, Roy; Pering, Trevor. System challenges for ubiquitous and pervasive computing. Proceedings of the 27th international conference on Software engineering St. Louis, MO, USA, May 17-19, 2005. ACM Press publishing New York, NY, USA, pp. 9 – 14, 2005.

[WVE00] van Welie, Martijn; van der Veer, G.C. and Eliëns, A. Patterns as tools for user interface design. International workshop on tools for working with guidelines, Biarritz France, 2000.

[WV03] van Welie, Martijn. and van der Veer, C. G.. Pattern languages in interaction design: Structure and organization. INTERACT '03, Proceedings of INTERACT '03, the ninth IFIP TC13 International Conference on Human Computer Interaction, September 1-5, 2005 Zurich, Switzerland.

[Wel00] van Welie, Martijn. Patterns in interaction design, The Amsterdam collection. Retrieved July 8th, 2002, from http://www.welie.com

[YA99] Yacoub, S. and Ammar, H. Tool support for developing pattern-oriented architectures, Proceedings of the 1st symposium on reusable architectures and components for developing distributed information systems, RACDIS'99, August 2-3, 1999, Orlando, Florida, USA, pp 6658-670.

[ZEBP04] Zimmerman, J.; Evenson, S.; Baumann, K. and Purgathofer, P. Workshop on the relationship between design and HCI. Proceedings of the International Conference on Human Factors in Computing Systems, CHI '04 extended abstracts on Human factors in computing systems, April 24-29, Vienna, Austria 2004. ACM Press publishing, New York, NY, USA, pp. 1741 – 1742, 2004.

[Zim95] Zimmer, W. Relationships between patterns In: Coplien, J.O. and Schmidt, D.C. (eds.): Pattern languages of program design, Addison-Wesley publishing, Reading, MA, USA, 1995.

Appendices

Appendix A

Additional Semantics of the System

In this appendix, we show some semantics associated with the Generic Pattern Model and some semantics of equivalent/identical relationship between patterns. They are showing possible extensions of the system in form of tool-support on top and outside the core system

The eXtensible Minimal Triangle, XMT

Our data models have syntax and semantics. The XML handbook [GP04] explains that semantics can be further divided into *semantic labeling* of contents, and *abstract interoperable behavior* as demonstrated in Figure 38.

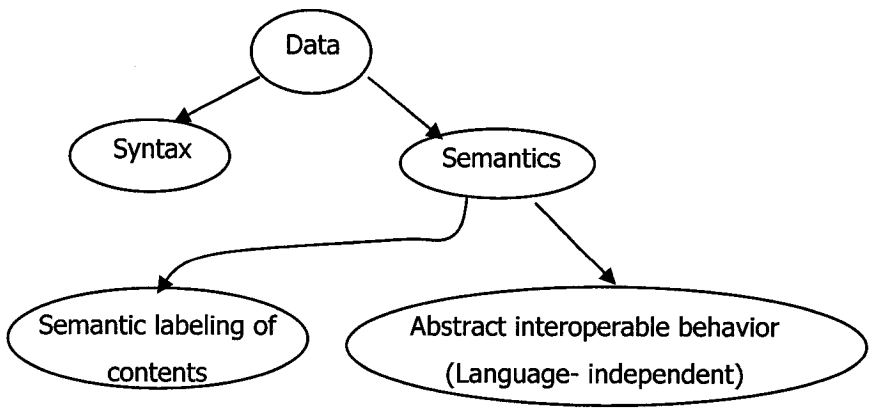


Figure 38: The date constituents

In the GPT model (Chapter5), we defined the syntax of the generic pattern template, and the semantic labels used for the syntax. Our work on XMT model focuses on the abstract interoperable behavior, which involves the behavior underlying the meaning of some of the tags we are using. We start by some preconditions and definitions.

As explained earlier, the common denominator of all pattern definitions is 'a problem to a solution in a context". Based on that, we defined the minimal triangle as in Figure 39.

> **Definition:** *The Minimal Triangle* is a representation of a pattern that has the three elements: a problem, a context, and a solution. No other elements are present in the minimal triangle.
>
> The minimal triangle represents the core meaning of a pattern. Any missing element of the three will result in a trivial pattern.

> **Definition:** *A trivial pattern* is a pattern that has at least one empty element from its minimal triangle.

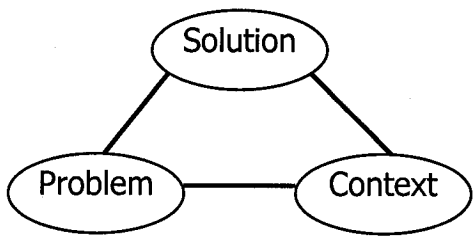


Figure 39: The Minimal Triangle

Identical Patterns

Informally, identical patterns are the same pattern mentioned in different collections. To be able to use some of our models, we introduce the following formal definitions:

Precondition: Each pattern has a unique ID

Precondition: Each non-trivial pattern has an existing, non empty minimal triangle (this can be inferred from the definition of trivial pattern).

Precondition: The following definitions strictly apply to non-trivial patterns. For trivial patterns, the definitions can be ambiguous.

Precondition: *Pattern_name(A)* refers to a single object called "the name of pattern A", of type "string".

Precondition: *Pattern_ID(A)* refers to a single object called "the identification of pattern A", of type "string".

Precondition: *Pattern_aliases(A)* refers to the set of zero or more objects called "the aliases of pattern A", of type "set of string objects".

Precondition: There is a one-to-one correspondence between *Pattern_name(A)* and *Pattern_ID (A).* We emphasize the word "correspondence" to denote a "*one-to-one*" and an "*onto*" function between Pattern_name (A) and Pattern_ID (A). In other words, the inverse relationship between Pattern_name(A) and Pattern_ID(A) is also a one-to-one function.

Notation (from the set theory): (A) € (B) refers to the object A being an element of the set of objects B.

These are important precondition that we will use in our next definitions to disambiguate some patterns that are identical or similar at different degrees. Some patterns are already connected to others, but the majority of them are not. We have identified many of these redundancies.

Definition: *Identical Pattern*

Pattern A is identical to pattern B, written A = B, if [9]

[9] This is a sufficient, but not a necessary condition. We can make it necessary by augmenting the aliases set of each pattern with our findings of identical patterns

Pattern_name (A) = Pattern_name (B) or
Pattern_name (A) € Aliases (B) or
Pattern_name (B) € Aliases (A)

Similar Patterns

As in several other applications, it is sometimes useful to separate between the look (the presentation) and the behavior. The same concept is used in separation of java's look and feel, contents and presentation in CMS (Content Management Systems)and other models. We define the next pattern relationships based on a similar concept: Patterns that look similar and act similar (identical- and similar patterns), and patterns that look different but act similar (equivalent patterns).

Precondition: The next definition is based on the definition of the *Minimal Triangle*.

Definition: *The XMT repository* specifies three finite sets of keywords corresponding to the three MT items:
The first set defines the specific kinds of **problems** covered by the patterns in form of reserved keywords.
The second set defines the specific kinds of **contexts** covered by the patterns, in form of reserved keywords.
The third set defines the kinds of **solutions** covered by the patterns, in form of reserved keywords.

Definition: The *Extensible Minimal Triangle* model refers to the three parts *problem*, *context*, and *Solution* of a given pattern as presented in the GPT and using a subset of reserved keywords elaborated in the *repository* of the XMT model.

The extensibility comes as a key concept of the XMT. If new patterns are to be added, and the keywords of any of its MT parts are not in the space of the XMT, the XMT must be extended by carefully defining the new keywords and adding them to the XMT repository. Using reserved keywords for context, problem, and solution allows for automated text processing, a scalable solution. However, the three items of the MT will have to be divided into a "brief" part, containing the reserved keywords, and an elaborate part containing the explanation and details of each item. This is sometimes applied to the solution item by providing a thumbnail solution "solution-brief" and an elaborate solution.

Definition: *Similarity Criteria* is a set of logical conditions that decide if a pattern A is similar to a pattern B.

Definition: *Similar Patterns (a similarity criterion)*
Pattern A is similar to pattern B, denoted A ≡ B, if [10]
MT (A) = MT (B) (where MT denotes the minimal triangle)

Definition: *Equivalent patterns* (a similarity criterion)
Pattern A is equivalent to pattern B, denoted A ≈ B, if
Problem (A) = Problem (B) and
Context (A) = Context (B)

We can see that an equivalence relationship is a superset of a similar relationship.

Two issues arise:
-Identical patterns are to be determined by pattern author (by including aliases to a pattern either to refer to another known name, or another pattern), but will also

[10] As in 4, this is a sufficient, but not a necessary condition

be complemented by our Structured Expert Support (SES) as shown in Chapter 5.

-Similar and equivalent patterns can be determined within the activity of SES in two different ways:

> **Formally**, by comparing the three MT items and applying the definitions of similarity given above. Once patterns are modulated according to the XMT model, this process can be automated using simple tools.
>
> **Informally**, by manually selecting patterns according to SES similarity and redundancy criteria (This structured activity is elaborated in Chapter 5)

The Progressive Abstract Type Hierarchy, PATH

Our previous models concentrate on semantically building patterns to allow for automating algorithms that can process their knowledge contents. We also need to be able to model the process of assimilating patterns as useful artifacts within the design process. Often, in order to achieve that, we need to remove some details that clutter the essence of patterns and blur the reason of using patterns at early stages of design.

The PATH model organizes patterns from an assimilation point of view (as opposed to the structural GPT and the behavioral XMT). Therefore it is located at the assimilation part of the pattern lifecycle (Chapter 5). We look at patterns from the way they can participate in a design process, so we define patterns to reflect different activities in the design:

-Defining the **intent**
-**Enumerating** the actions that realize design intentions
-Choosing concrete objects to **represent** these actions
-Looking at some **implementation** (or a prototype) details
-Writing the **source code** or building visual programming objects

These sequential activities are some of the design steps that could involve interaction with patterns. Using the PATH model, we can reorganize patterns to reflect each of these steps as shown in Figure 40. Several patterns from different collections already exist at all levels of abstractions depicted in this model, but without mentioning this abstraction relationship between them. Users are often confused when they try to figure out the difference between the "*Go Back to a Safe Place Pattern*" and the "*Bread Crumb Pattern*". They are simply the same type of patterns at different levels of assimilation abstractions.

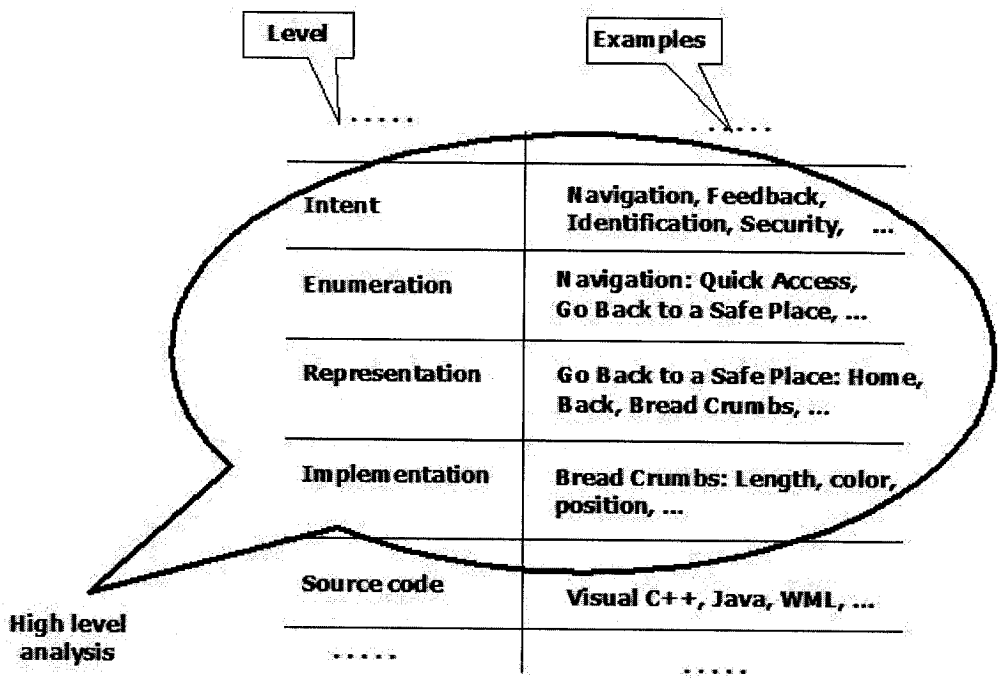


Figure 40: Snapshot of the PATH model

This PATH model is progressive in the sense that it provides additional details at each lower levels of the model for the same pattern, and provides several patterns as possible siblings at the same level with possible parents at higher levels. The same pattern is hence presented at different level of the model by

aggregating several existing patterns in an is-a relationship. The levels of the model map to stages of design process. The model will help in several ways:

-Putting patterns in a defined relationship will help reduce ambiguity
-At each stage of the design process, the selection of related pattern will be reduced by limiting the selection of patterns to the relevant ones only
-Many patterns exist without their upper level abstractions. We need to fill these semantic gaps by adding the missing pattern types to help group patterns together in an assimilation tree without isolated clusters
-The assimilation process will be clarified and automated by descending down the PATH tree and pruning out the unwanted selections

As a start, it is sufficient to focus on four levels of the assimilation process. Lower levels of source code patterns are a natural part of this model, but there is no need to add its associated patterns. Same applies to higher levels of the hierarchy like the goals and objectives of the design that can affect the selections and pruning of the PATH branches.

Structured Expert Support Implementation

Beside models, manual work is essential to both clearing out, and uploading modeled patterns. Here we create new assumption set, and define some activities to help build the new system.
To address the relationships between patterns as explained in the extrinsic data part within the GPT [GSP05], we have defined two types of relationships:

Structural relationships

These are patterns that have some common structure (like a common MT; or part of it).

Assimilation relationships

Those patterns are considered from the design process point of view. Two patterns that are completely different in structure can still have an assimilation relationship, like complementing or competing with each other.

Argument

These two types might look orthogonal to each other, but -generally speaking- we assert that structural relationships should be included as a subset of all legitimate assimilation relationships. During the design process, patterns can be replaced based on several criteria. Similarity of pattern structure is a criterion that warrants the possibility of replacement. This concept will be demonstrated with some examples later in the appendix. We will discuss this further with the issue of redundancies and show the unwanted effect of structurally entwined patterns.

Gamma et al [GHJV95] emphasize in their book "Design Patterns" that defining the contextualized relationship between patterns is a key notion in the understanding of patterns and their usages. Zimmer [Zim95] implements this idea by dividing the relations between the patterns of the Gamma catalog itself in 3 types: "*X is similar to y*", "*X uses Y*", and "*Variants of X uses Y*". Based on Zimmer's work, we showed in [GSS04] as in Chapter 5 some additional relationships between patterns from an assimilation point of view, not a structural one, we restate them here:

-**Subordinate (X, Y)** if and only if X is embeddable in Y. Y is also called superordinate of X.

-**Equivalent (X, Y)** if and only if X and Y can be replaced by each other.

-**Competitor (X, Y)** if X and Y cannot be used at the same time for designing the same artifact.

-Neighboring (X, Y) if X and Y belong to the same pattern category (family) or to the same design step as the described pattern.

A major nuisance to our work on pattern relationships, and certainly that of other users is the "*noisy similarities*" between patterns. We can visualize the relationships between patterns as in Figure 41. Part of the similarity is a healthy relationship of **similar** or **equivalent** patterns that can easily replace each other; represented by the intersection area of the two ellipses. Several dissimilar patterns have relationships like **competitor**. A considerable part of similarities, however, has redundancies in it, and we excluded them from the concept of pattern relationships. They are represented in the part of the "pattern similarities" ellipse outside the relationships ellipse. According to our investigation on many patterns, it is not easy to decide on the nature of the relationships in these cases.

To demonstrate, if we said that pattern A is a competitor to pattern B, and then we have another pattern C which is slightly (or –in general- vaguely) similar to pattern B, can we say that pattern C is also a competitor to pattern A. The answer is that we have to resolve the vague similarity between B and C to remove this ambiguity first. Logically speaking, this will not guarantee a solution to the question. But if we were able to assert –for example- that pattern B and pattern C have an inheritance relationship, we might assume that A and C are likely competitors as well. If we were able to assert that pattern B and pattern C are similar, then we can remove pattern C altogether. In many cases we found that some patterns contain a full detail of other patterns with a slight addition and often with a slight omission. Other similarities have more entwined structure that has no clear answer. We provide more discussion in the next section.

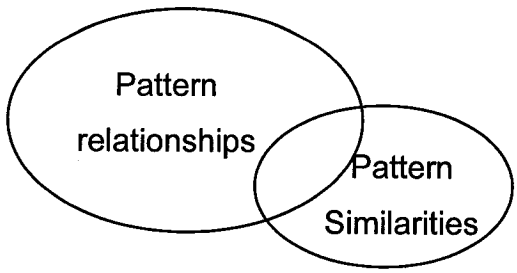


Figure 41: Useful and noisy similarities

Inter-collection Redundancies, ICR

Entities should not be multiplied unnecessarily

William of Occam

Besides our automatic discovery approach through the XMT model, similar patterns have been discovered and recorded manually. These two approaches are going in parallel to complement each other as explained earlier. We have identified many identical patterns and similar patterns with varying degrees of similarity. In some cases we were able to identify patterns in different collections that are exactly the same, but were presented by different authors under different names, despite their identical analysis. This is an "easy work", and these are marked simply for removal. Other identical patterns are more difficult to locate.

The challenge comes with what we define as *entwined patterns*. They represent another case of partially similar patterns that can be confusing and has to be resolved. It is when two (or more) patterns have common features but each one still has a unique set of features. Despite the several types of entwining - depending on the type of redundant features-They can all be logically represented by the case of entwined brackets

{ Part of pattern A (common part } Part of pattern B)

In several domains, like in programming languages, this is generally not allowed. In pattern domain, our observations show that this is one of the most confusing redundancies between patterns. We need to dissolve these patterns either by combining them into one pattern or separating them into two distinct patterns.

Note:

Analogous to the defined relationships of inheritance (is-a) and aggregation (has-a) in object oriented programming, we also have pattern inheritance and aggregation relationships. In patterns domain, these are not a problem of similarity or a source of confusion; the former is addressed in the PATH model earlier. The latter is a clear case of assimilation relationship. Connecting these patterns together is done through the extrinsic data part of the GPT.

Appendix B

A Case Study on Pattern Ontology

As discussed in the thesis, the semantics of the generic pattern model is a key issue towards scalable processing of patterns. The Semantic Web implements the idea of having data on the Web defined and linked in a way that can be processed interactively by both humans and machines, as explained by Berners-Lee [BF00]. Semantic Web technologies, such as RDF, DAML, and OWL are built on top of XML, providing enhanced capabilities to express the meaning of knowledge on the Web. The creation of ontologies is an explicit formal approach to represent the objects, concepts and other entities in a specific domain of interest and the relationships that hold true among them. This formal representation facilitates the creation of agents that can compute while retaining human readability.

Semantic pattern ontologies are specifically designed for computation by agents. For example, the relationship that using one pattern requires the use of another pattern can be represented in pattern ontology. In addition, inference engines are built on top of the ontologies to retrieve and extract grounded facts that are already declared in the ontology. Many such engines are capable of transitivity and other inferencing capabilities. This provides a degree of intelligence in finding relationships among usability patterns and can be used to find applicable and/or useful patterns.

Relationships between patterns are complex and hard to formulate exhaustively. Well known and well stated relationships can be automated in a design tool as shown with UPADE. However, the implications of using patterns in known

combinations and the extent of proposing new combinations have no limit except designers' creativity. For experienced user interface designer who has a strong background in HCI, new combinations can significantly increase the usability of the Website. However, making sure that they work correctly is not an easy task in particular for novice developers.

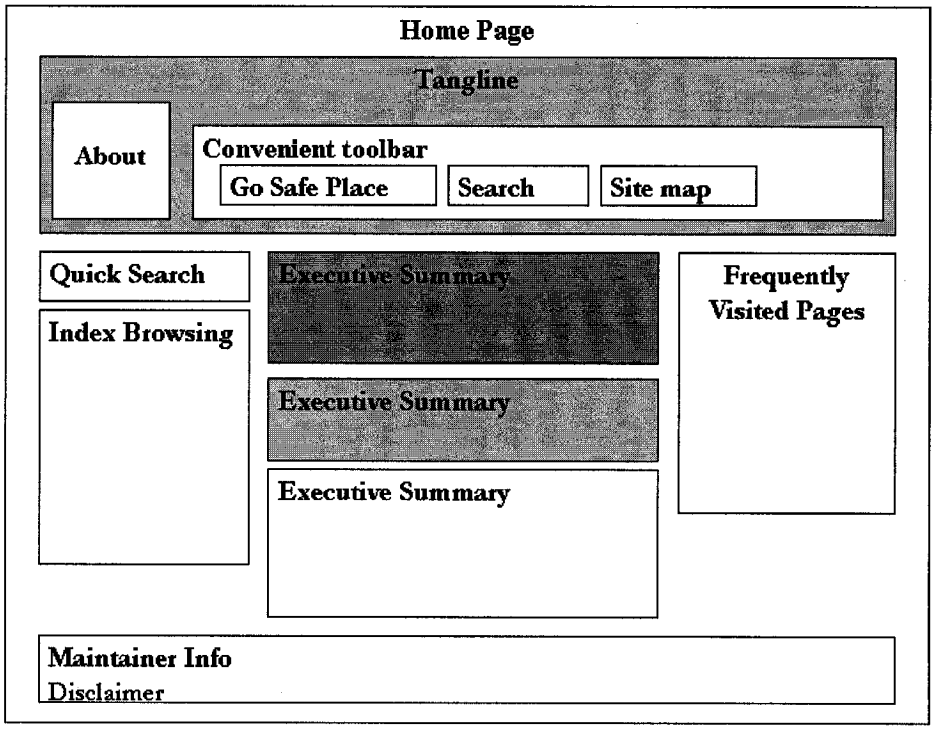


Figure 42: Typical HCI patterns applied in page layout

For example, Figure 42 describes how patterns could be combined in a design of a home page. Some of the patterns are competitors while others are simply similar.

We introduce a simplified practical example in Figure 43 to illustrate the role of ontology for patterns. For the context of a large Web site, we have the three patterns

-Site map

-Index browsing

-Home page

The following two relationships exist between them:

-Site map can be "**combined with**" index browsing.

-Site map can be "**embedded in**" home page.

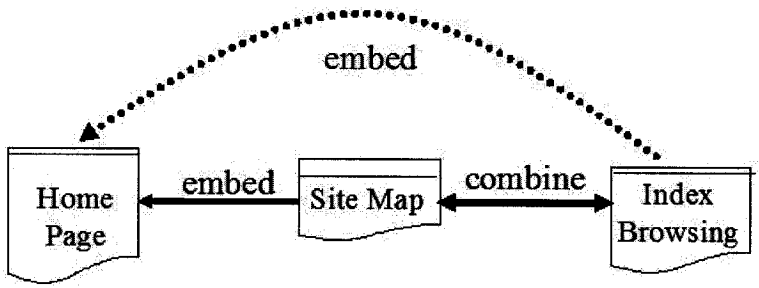


Figure 43: Deducing additional relationships

Using simple human common sense, we can deduce the following new relationship from the two relationships we already have: 'Index browsing can be "embedded in" home page'. For a computer, however, we need to develop a mechanism of structuring data and relationships in a suitable way before it can arrive at the same conclusion as we did. Adding semantics alone is not enough. We can add tags to tell that *home page* is a type of patterns, and that *combined with* is a type of binary relations, but this is not enough to deduce the third relation by a machine. We can have a schema for our document to specify that – for example- a unique ID and at least one author must follow each pattern name in our collection. Schema adds more structure to our patterns but is still not enough in our case.

What we need is to be able to specify the semantics (semantic tagging as well as the meaning) of each relation. Equally important, we need to specify the behavior of each relation regarding its interaction with other relations (in other words, the implications that can rise from mixing two or more relations). Ontology can help in this case. It offers an infrastructure for representing data items (patterns in our case) and relations between them in a precise way that emphasizes the meaning of these relations. Logic and reasoning tools can then take over to formally analyze the ontology, verify the existing relations, discover any contradictions, and look for other relations that can be deduced from the existing ones. The DARPA Agent Markup Language (DAML, at http://www.daml.org) reported 45 ontology-specific tools in 2002, mostly open source, that can help in this approach. In 2005, this number has grown to 87 within a total of 243 XML tools (http://www.daml.org/tools/). We have used several of them in our approach to implement smart patterns. These 243 tools implement XML related technologies as specified in the W3C specifications, the world wide authority responsible for developing and maintaining the XML (http://www.w3.org/XML/). Besides extracting the implicit information, Ontology can help in discovering where some information might be missing and highlights the need for further investigations to fill them.

The following example demonstrates this idea as depicted in Figure 44. For the context of a small Web site, we have the four patterns

-Site map
-Index browsing
-Menu bar
-Home page

The following ternary relationship exist between them:

- In *home page, site map* must be **"replaced by"** *menu bar*

We can deduce that *menu bar* can be "**embedded in**" *home page*, but we cannot tell if *index browsing* could be embedded in *home page* as well. The reason is that we don't have enough information to know if it were possible. We discover the need to check for a possible relation between *menu bar* and *index browsing*. This missing piece of information is needed. If we found –for example- that these two patterns can be combined together with no restriction, then we can answer the question by deducing the relation: *index browsing* can be embedded in *home page*.

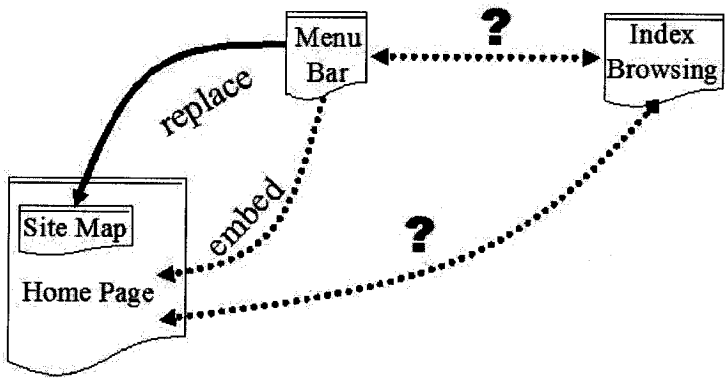


Figure 44: Missing information in pattern relationships

Appendix C

A Case Study on Pattern Assimilation

In this appendix, we provide a case study to support the reconciliation view we proposed in Chapter 4. The management of a hotel is going to be computerized. The hotel's main business is renting out rooms of various types. There are a total of 40 rooms available, priced according to their amenities. The hotel administration needs a tool capable of booking rooms for specific guests. More specifically, the application's main functionality consists of adding a guest to the internal database and booking an available room for a registered guest. Moreover, only certified guests have access to the main functionality of the program. Eventually, the application would be running on WIMP-based systems.

Note that only a simplified version of the hotel management system will be developed. The application and corresponding models will not be tailored to the different platform and user roles. The main purpose of the example is to show that model-based UI design consists of a series of model transformations, in which mappings from the abstract to the concrete models must be specified. Furthermore, it will be shown how patterns are used to establish the various models, as well as to transform one model into another. A summary of all patterns used in this article can be found in Table 7 of Chapter 4.

The Task Model

Figure 45 depicts the coarse-grained task structure of the envisioned hotel management application. Only high-level tasks and their relationships are portrayed. An impression about the overall structure and behavior of the

applications is given. The structure provided is relatively unique for a hotel management application. The concrete "realization" of the high-level tasks has been omitted. The Pattern Task symbol is used as a placeholder representing the suppressed task fragments.

A large part of many interactive applications can be developed from a fixed set of reusable components. If we decompose the application far enough, we will encounter these components. In the case of the task model, the more the high-level tasks are decomposed, the easier the reusable task structures (that have been gained or captured from other projects or applications) can be employed. In our case, these reusable task structures are documented in the form of patterns. This approach ensures an even greater degree of reuse, since each pattern can be adapted to the current context of use.

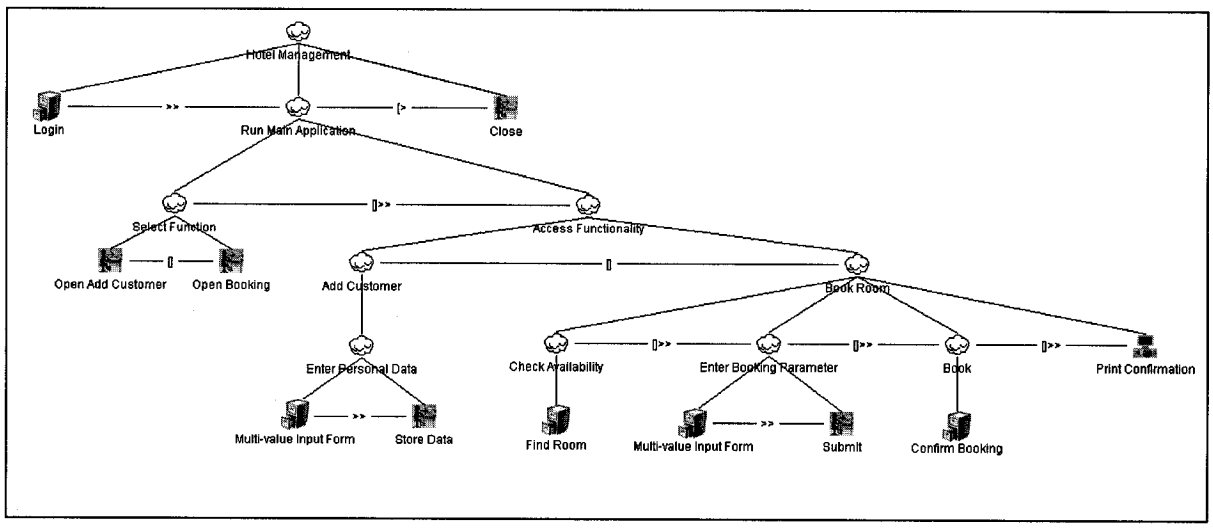


Figure 45: Coarse-grained task model of the hotel management application

The main characteristics of the envisioned hotel management application, modeled by the task structure of Figure 45, can be outlined as follows:

Accessing the application's main functionality requires logging in to the system (the login task enables the management task). The key features are "adding a

guest" by entering the guest's personal information and "booking a hotel room" for a specific guest. Both tasks can be performed in any order. The booking process consists of four consecutively-performed tasks (related through "Enabling with Information Exchange" operators):

1. Locating an available room
2. Assigning the room to a guest
3. Confirming the booking
4. Printing a confirmation

The *Login*, *Multi-Value Input Form*, *Find* and *Dialog* patterns of table 7 can be used in order to complete the task model at the lower levels. In the next section, the application of the *Find Pattern* will be described in greater detail.

Completing the Find Room Task

The *Find Pattern* is essential to completing the "Find Room" task. In contrast to the patterns already used in this example, the *Find Pattern* suggests a number of options rather than providing a task structure. Figure 46 illustrates how finding an object can be performed by searching, browsing or employing an agent, depending on the pattern.

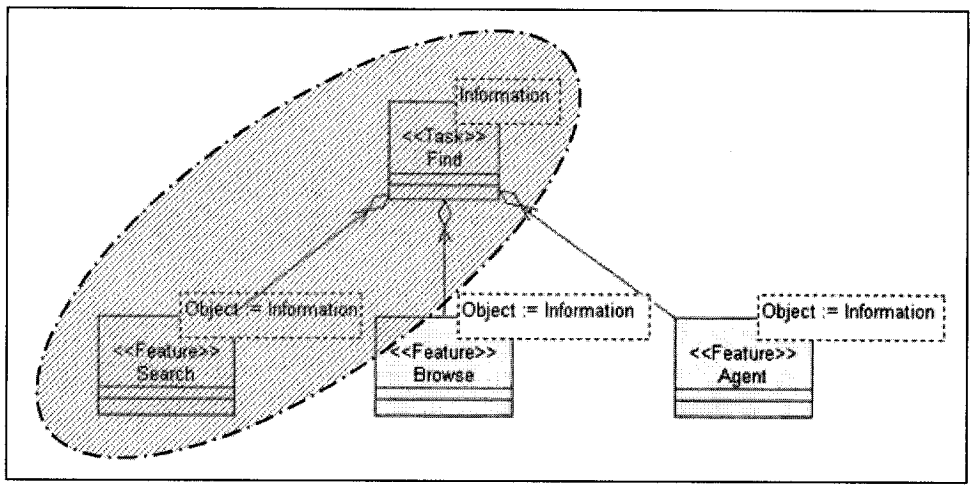


Figure 46: Interface and structure of the *Find Pattern*

Within the scope of the hotel management application, the task of finding an available room should only be performed by searching with query parameters, as shown in the grey ellipse of figure 46. The "Information" variable of the *Find Pattern* (in this case, a placeholder for the "Hotel Room" value) is used to assign the "Object" variable of the *Search Pattern*.

The *Search Pattern* suggests a structure in which the search queries are entered then the search results are displayed. Again, the *Multi-Value Input Form Pattern* is used to model the tasks for entering the search parameters into a form. The following search parameters can be used when searching for an available room: "Arrival Date," "Departure Date," "Non- Smoking," "Double / Single," "Room Type." After submitting the search queries, the search results (*i.e.*, the available hotel rooms) can be manually scanned using the *Browse Pattern* or, based on the search results, a refinement search can be performed by employing the *Search Pattern* recursively. Within the scope of our case study, refinement searches are unnecessary, and the search results should only be browsed.

According to the *Browse Pattern,* the list of objects (the hotel rooms) is printed, after which it can be interactively browsed as an option. Details of the hotel room

can be viewed by selecting it. The *Print Object Pattern* is used to print out object's properties. It suggests using application tasks to print the object values that can be directly or indirectly derived from the object's attributes. In the case of the hotel management application, the following hotel room attributes should be printed: "Room Number," "Smoking / Non-Smoking," "Double / Single," "Room Type," and "Available Until."

After adapting all patterns to suit the hotel management application, the concrete task structure displayed in Figure 47 is derived. Note that the "Make Decision" task has been added manually, without pattern support.

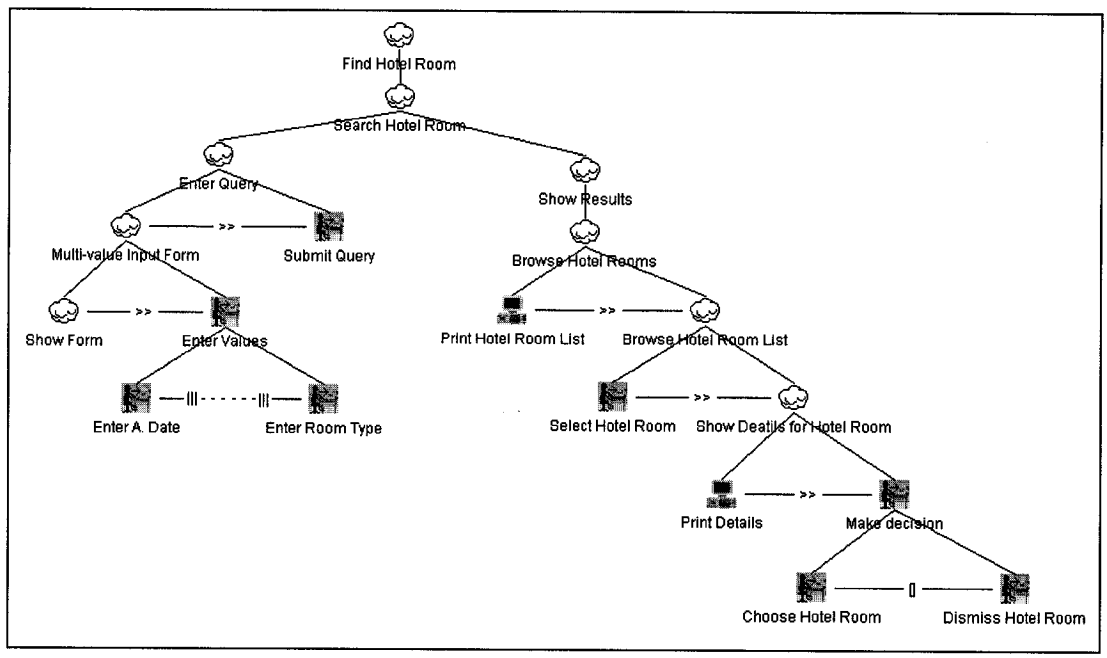


Figure 47: Concrete task structure delivered by the *Find Pattern*

A first draft of the envisioned task model can be derived once all patterns have been adapted and instantiated. At this point, first evaluations can be carried out. For instance, the XIML-Task-Simulator [FDRS03] can be used to simulate and

animate possible scenarios. Results of the evaluation indicate that preliminary modifications and improvements of the task model are possible.

Designing the Dialog Structure

After establishing the envisioned task model in our case study, the dialog models can be interactively derived. In particular, the various tasks are grouped to dialog views, then transitions are defined between the various dialog views. Since the desired target platform of the hotel management application is a WIMP-based system, a dialog view will be subsequently implemented as either a window or a container in a complex window.

When designing the dialog graph for the hotel management application, we designated the login dialog view as modal. After executing Submit, the "Main Menu" dialog will be opened. As such, a sequential transition between both dialog views is defined. From the main menu, either the "Add Guest" or "Search Applicable Room" dialog view can be opened by a sequential transition. After completing the "Add Guest" dialog view, the main menu will be re-opened. For this reason, a sequential transition to the "Main Menu," initiated by the "Store Data" task, must be defined.

The application's booking functionality consists of a series of dialog views that must be completed sequentially. The *Wizard* dialog pattern emerges as the best choice for implementation. It suggests a dialog structure where a set of dialog views is arranged sequentially and the "last" task of each dialog view initiates the transition to the following dialog view. Figure 48 depicts the *Wizard Pattern's* suggested graph structure.

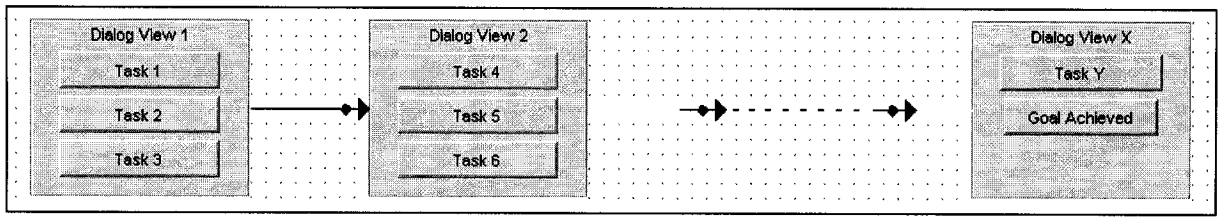


Figure 48: Graph structure suggested by the *Wizard Pattern*

After applying the *Wizard Pattern*, the dialog views "Search Applicable Room," "Browse Results," "Show Details," "Enter Booking Parameters," "Confirm Booking" and "Print Confirmation" are connected by sequential transitions.

However, the sequential structure of the booking process must be slightly modified in order to enable the user to view the details of multiple rooms at the same time. Specifically, this behavior should be modeled using the *Recursive Activation* dialog pattern. This pattern is used when the user wishes to activate and manipulate several instances of a dialog view. In this particular case, the user will be able to activate and access several instances of the "Show Room Details" dialog view. This pattern suggests the following task structure: starting from a source dialog view, a creator task is used to concurrently open several instances of a target dialog view. In our example, the source dialog view is "Browse Rooms" and the "Select Room" task is used to create an instance of the "Show Room Details" dialog view.

A premature exit should be provided to offer the user the possibility to abort the booking transaction. In the hotel management application, this is achieved by the "Confirm Booking" dialog view. At this point, the user can choose whether to proceed with the booking or to abort the transaction. Another sequential transition must therefore be defined: one which is initiated by the "Select Cancel Booking" tasks and leads back to the main menu. The hotel management application's complete dialog graph, as visualized by the Dialog Graph Editor tool, is depicted in Figure 49.

The next step is to evaluate the defined dialog graph. The dialog graph can be animated using the Dialog Graph Editor to generate a preliminary abstract prototype of the user interface. It is possible to dynamically navigate through the dialog views by executing the corresponding tasks. This abstract prototype simulates the final interface's navigational behavior. It supports communication between users and software developers: design decisions are transparently intuitive to the user, and stakeholders are able to experiment with a dynamic system.

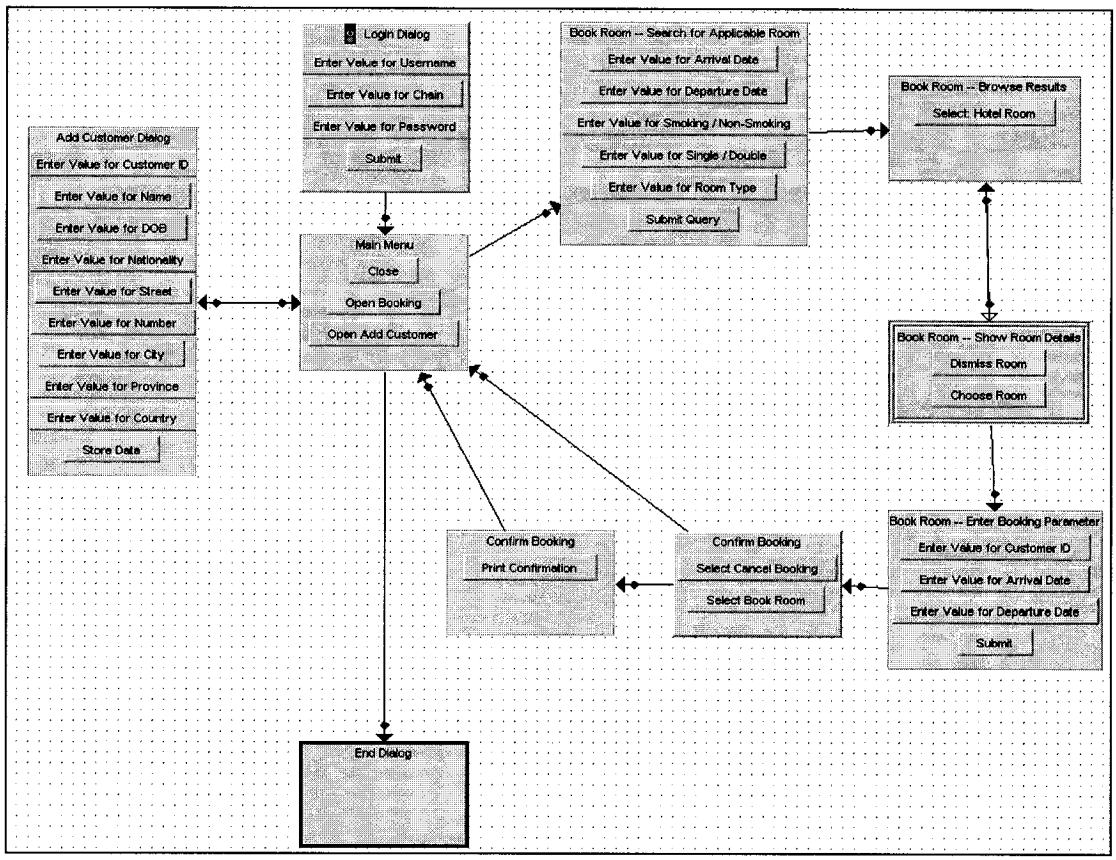


Figure 49: Dialog graph of the hotel management application

Defining the Presentation and Layout Model

In order to define the presentation model for our example, the grouped tasks of each dialog view are associated with a set of interaction elements, among them forms, buttons and lists. Style attributes such as size, font and color remain unset and will be defined by the layout model.

A significant part of user's tasks while using the application revolves around providing structured text information. This information can usually be split into logically related data chunks. At this point, the *Form Presentation Pattern*, which handles this particular task, can be applied. It suggests using a form for each related data chunk, populated with the elements needed to enter the data. Moreover, the pattern refers to the *Unambiguous Format Pattern*, in conjunction with which it can be employed.

The purpose of the *Unambiguous Format Pattern* is to prevent the user from entering syntactically-incorrect data. Drawing on information from the business object model, it is able to determine the most suitable input element. In other words, depending on the domain of the object to be entered, the instance of the pattern provides input interaction elements chosen in such a way that the user cannot enter syntactically-incorrect data.

Figure 50 shows the windows prototype interface rendered from the XUL fragments of the hotel management application's presentation model for the "Login," "Main Menu," "Add Guest" and "Find Room" dialog views. All widgets and UI components are visually arranged according to the default style.

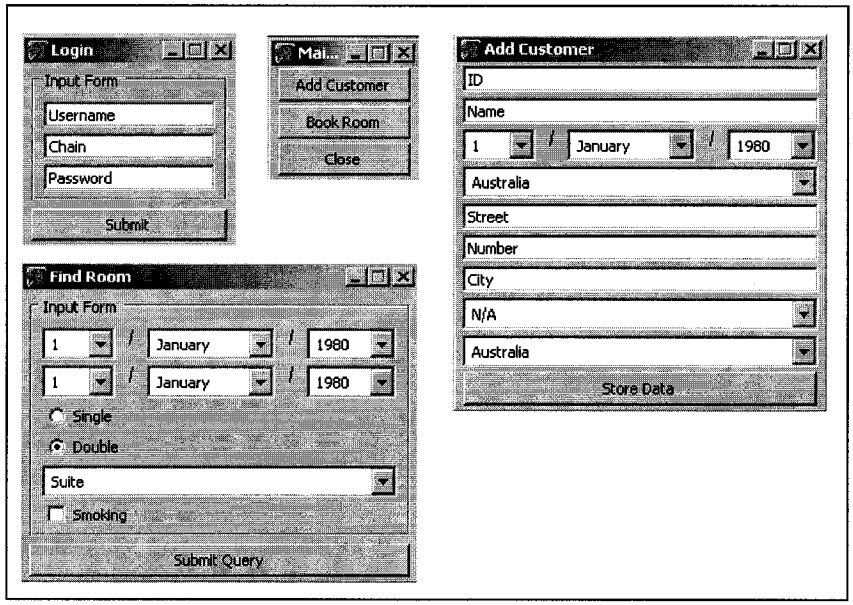


Figure 50: Screenshots of visualized XUL fragments

In the layout model, the style attributes that have not yet been defined are set to conform with the hotel management application's standards. According to the *House Style Pattern* (which is applicable here), colors, fonts and layouts should be chosen to give the user the impression that all windows of the application share a consistent presentation and appear to belong together. Cascading style sheets have been used to control the visual appearance of the interface. In addition, to assist the user when working with the application, meaningful labels have been provided. The *Labeling Layout Pattern* suggests adding labels for each interaction element. Using the grid format, the labels are aligned to the left of the interaction element.

The layout model determines how the loosely connected XUL fragments are aggregated according to an overall floor plan. In the case of this example, this is fairly straightforward since the UI is not nested and consists of a single container. After establishing the layout model, the aggregated XUL code can be rendered together with the corresponding XUL skins as the final user interface. Figure 51 shows the UI rendered on Windows XP.

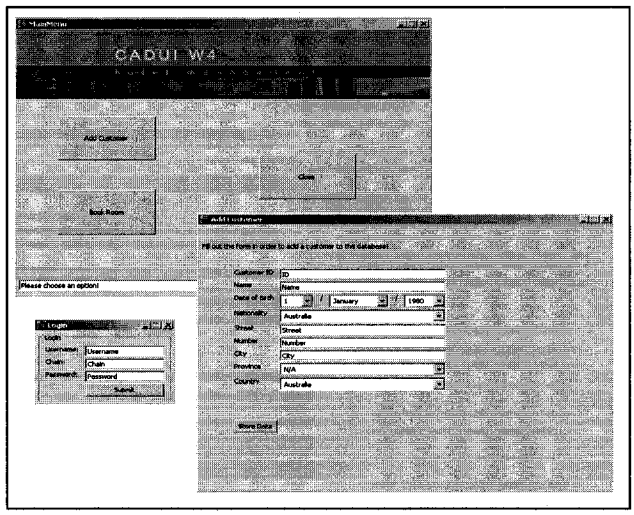


Figure 51: Screenshots from the hotel management application

Appendix D

A Survey on Pattern Reuse in Practice

Abstract. This Appendix reports the preliminary results of a survey on the popularity of patterns among mainstream developers in industrial settings. Our focus is on design patterns for interactive systems as identified early in the design community [Lin80] and [Rou81]. They include the design and implementation of user interfaces for highly interactive software systems. The last few years have seen growing number of new UI-related patterns. Numerous research projects and articles focus on how patterns "should" be generated and used [AIS77] and [Tid97]. We focus on the other end of the journey, namely how patterns are "actually" being used. The primary contribution of this survey is to investigate how patterns are perceived and applied in practical industrial settings by software developers. Our goal is to identify the current state of affairs and measure the effectiveness of existing pattern approaches and tools. Preliminary results of the survey showed that patterns are less popular among designers than commonly anticipated by research community, even after valuable theoretical analysis and rich pattern literature are produced. The results reflect the strong belief that patterns can indeed be helpful, however it shows a major gap between this belief and the fact that only few pattern collections are popular, generally the older ones. New patterns are much less popular regardless of their quality. The survey also shows that the number of developers who are actually applying patterns in their work is much less than the number of developers who are just familiar with them.

Introduction and related work

This article reports the preliminary results of a survey to ask mainstream software professionals about their practices with patterns. We focus on the development of interactive systems, which involve the development of software systems with user interfaces. Traditionally, the interface part received considerably less attention than the underlying software system. Developers spent most of their efforts focused on the underlying system; interfaces were then "thrown in" at later phases of the development. This approach did not work well with the growing complexity of software system. Moreover, the type of typical users expanded from "experts only" with specialized training and sophisticated background to "ordinary users" with much simpler background.

Several failures of good software systems were attributed to the poor design of their interfaces, so researchers started calling for separation between the interface and the applications, and for spending more time and expertise on the interface design. Accordingly, we saw the emerging fields of user interface- and interaction design grow rapidly. Interface-related patterns also emerged in books and on the Internet and significant research and contributions have quickly accumulated. Researchers used feedback from each other to build, validate and improve theories about patterns and how they should be applied.

While pattern authors remained focused on discussing how patterns can be used, we did not find significant studies on how these patterns are actually perceived and applied in practice or how much they contributed to improving the quality and usability of interfaces. Some surveys and studies were done on interface design methods [Dia03], on users [CP96] and [PC96] as well as interface usability and how to improve it [BH99]. Other surveys have focused on existing pattern collections in research community [ML01] and [Por03]. We also found empirical studies on how design patterns are perceived and promoted in the academia as a pedagogic tool [PC04] and [CL99]. However, we found only

one survey made in 1996 on design patterns usage in practice [BCC+96]. The need for more surveys is necessary to assess the large body of research that has been done on patterns since then, and to examine the different directions in which it is evolving.

Our research goal is to identify the current state of affairs of patterns, and to measure how effective they are delivered and used in software industry. The study enables us to reveal the strengths and weaknesses of pattern practices and to measure the success or failure of current role of patterns in supporting the UI design process. This allows us to shed lights on some grey areas in patterns and pattern tools and to have a better understanding of the needs of UI designers from a pragmatic point of view.

The Survey Structure and Population

The survey came in the form of a questionnaire divided in 8 sections totaling 20 questions. The first three sections were devoted to get some information about responders and their work environment as well as sources of their professional knowledge. Section 4 and 5 were designed to evaluate their general perceptions and usage of guidelines and pattern. Section 6 and 7 enabled us to get more detailed information about the practices of using patterns and tools in UI design. The last section collected feedback about existing and future research trends and proposals. The survey was distributed during 6 months and sent as a broadcast to selected professional mailing lists as well as personalized emails. We focused on software professionals in industrial settings who are practically involved in the development of software systems and especially in user interfaces. We targeted large companies as well as medium size companies and consultants. The qualified number of replies was 121.

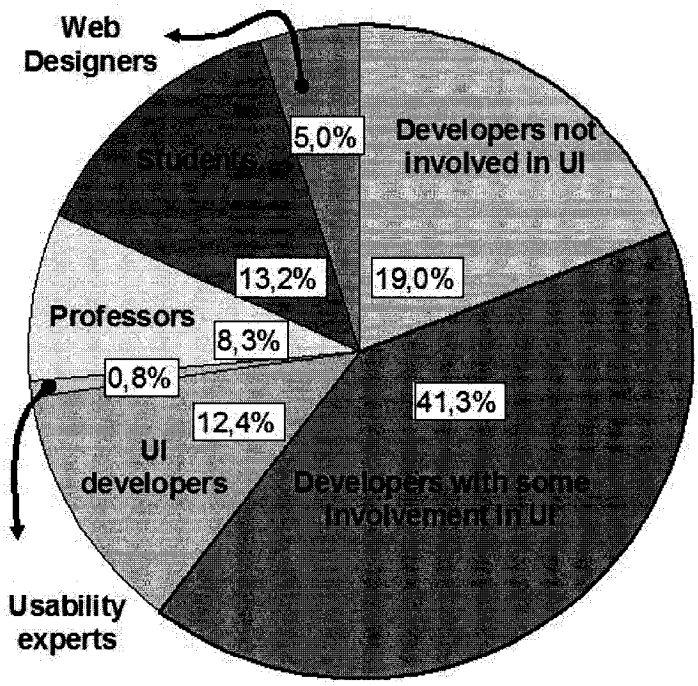


Figure 52: The distribution of respondents

Analysis Method

The survey analysis is organized under three categories. First we ran a frequency analysis on users' profile as well as their work environment and knowledge sources. We also evaluated the popularity of pattern collections and tools. Next we cross-tabulated some questions to report the opinions and practices of patterns among different respondents. All the data collection and statistical analysis were done using SPSS (www.spss.com). Finally, we collected all open-ended questions and analyzed them manually.

Summary of Survey Findings

The majority of replies (41.3%) came from developers with involvement in user interfaces (Figure 52). 12.4% of all respondents were dedicated user interface developers, 19% were system developers with no involvement in user interface

and 5% were web developers. While we only targeted industrial settings, we had 8.3% professors and 13.2% students in the replies, which reflect the fact that some people from the academia are indeed working in the industry as well.

Who Develops the User Interface?

The analysis of work environment and responsibilities delegated to development teams (Figure 53) revealed that while a high percentage of respondents worked in user interface related teams (50%), only a minority (16%) worked exclusively in interface development teams. The rest (34%) worked simultaneously in software and interface development. It is clear that the promoted idea of separate and dedicated teams for interface development and the underlying software development did not make its way completely in the industry. However, we did not correlate this to the size of the companies.

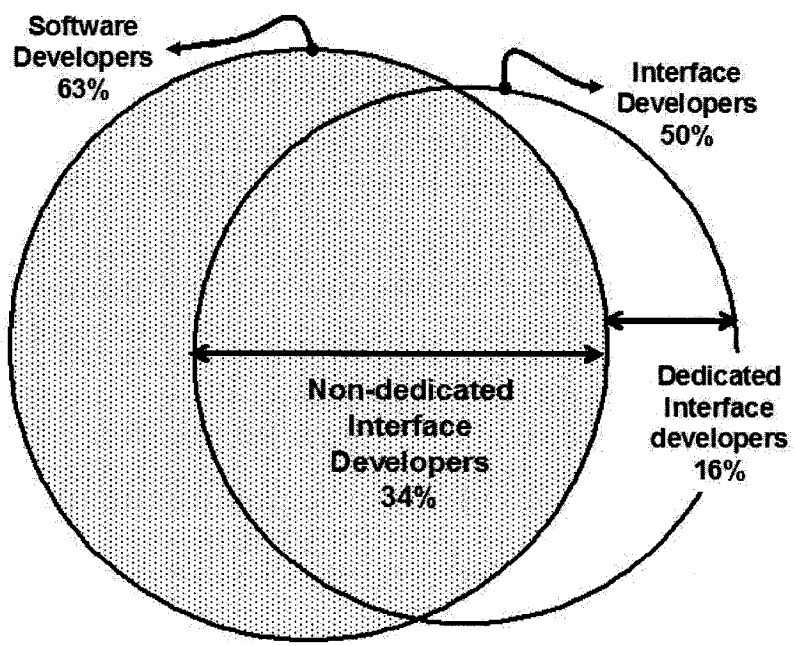


Figure 53: The responsibilities delegated to development teams

When asked about some sources of professional knowledge, the Web was the most commonly used source (86%). Academic training was next (76%), books (63%), reuse of similar work (59%), professional training (46%), working with mentors was the least popular at 27%.

The Current Practices of Guidelines and Patterns

While guidelines are generally considered an important support for designers, only 2.5% take time to read them carefully and apply them. 23 % browse them occasionally, and 66% don't use them at all in their work. As for the popularity of patterns, 77% were familiar with design patterns (GoF) and 40% with some UI patterns. However familiarity did not match the actual use of these patterns: only 15% used UI patterns in industrial projects, showing that most developers know about UI patterns, but don't use them.

The number of popular UI pattern collections was also low. Only 7 collections were known to the respondents, generally the early ones. Despite the popularity of the web as a source of professional knowledge, many new UI pattern collections on the web were not known at all in the survey, regardless of their quality as known in the research community. This certainly requires further study.

The Status of Pattern Tools

We identified similar gap regarding existing tools for patterns. Despite the extensive research and rich publications of tool prototypes, only 10% of respondents used tools. These were mostly general design (CASE) tools like Rational Rose, Eclipse and Sun Java Studio, rather than specialized pattern tools. Only 10 tools were reported in the survey, 6 of them were mentioned only once.

The Mainstream Perception about Patterns

We also asked participants how they thought about patterns. The majority (59%) saw them as an effective concept while 35% were not sure. Only 5% did not like the concept of patterns altogether.

Conversely, when asked about the effectiveness of patterns in practice today, only 29% found them useful in their design while 44% found them not useful. Nonetheless, developers showed they still have faith on pattern; only 7% were on the pessimist side about the future, saying that patterns will never be a real help to developers. 36% were optimistic and believed that patterns will help future developers while 56% were not sure. This shows that patterns are underutilized in daily practices in the industry, but people still believe they are a good concept. The difference between the high acceptability of patterns as an idea (59%) and between finding them useful in design (29%) may indicate that we need to support pattern reuse more actively. It points to the need to improve our techniques to facilitate pattern reuse among industrial developers.

Conclusion

This survey presents the preliminary findings of, to the best of our knowledge, the first survey on pattern practices in the industry in the last decade. A total of 121 software professionals and developers participated in the survey. The survey focused on patterns outside the research community and sought the opinion of interface practitioner. Based on the survey results, current practices of patterns are reported, as well as some observations. The survey pointed at the problem of patterns underutilization in the industry, contrary to what is expected by pattern authors. We conclude that besides discovering new patterns, we need to develop more techniques to improve existing pattern reuse and integration into the daily activities of mainstream programmers. We also need more pattern tool support to facilitate this reuse.

Appendix E

Broadcasting HCI Patterns on the Web:

Its effectiveness as a Dissemination

Method

Internet is large, and is growing fast. Its unabated growth and increasing significance has resulted in a flurry of research activity to improve its capacity for serving information more effectively [DNB02]. It is hard to come up with accurate estimation of its size or behavior. Several studies focus on estimating the current size, and indicate a number of challenges associated with managing its information contents [DFG99], [PB03]. Other studies focus on observing the behavior and dynamics of the Web and how it is being used [TAW03] [CDI+04]. [DNB02] provide a detailed study of Web metrics commonly used to evaluate the Web performance in a quantitative way. Web users also received a fair share of studies. [BKB00] and [Run93] focus on the human factors and provide measurements for the usability of Web pages (quality of use). We are interested in the current methods of how pattern community, mainly pattern authors use the Web as a medium to communicate and deliver patterns to mainstream users. In the study we took measurements in several search experiments to observe the changes in pattern-related searching on the Web, and evaluate how it has changed during the 3-year span of the study.

Introduction

Internet is a major player in today's information revolution. We are seeing an unprecedented growth and ubiquity across many areas of research and industry. It has grown so much that it is becoming hard to perceive or estimate its size, contents or behavior. The term "Internet explosion" emerged over a decade ago [Run93] when the Internet size was actually negligible compared to today's size (Figure 54). The Internet has been recently dubbed "*the Billion User network, with no owner*" by ACM [ACM 05]. Many research communities rely on the Internet for communication and knowledge dissemination, including pattern researchers. Patterns are currently an active area, spanning many engineering domains. Pattern authors often use the Web to broadcast their patterns to potential users. In this study we investigate the effectiveness of this method for delivering patterns. We focus on two main aspects: the size of information on the Internet, and the popularity of Web search using keywords like "pattern". When combined together in one context, these two aspects shed new lights on the dynamics of data retrieval from the Web and the inadequacy of current pattern broadcasting practices. We can see how patterns behave during their journey until they reach their users. We argue that in terms of Internet-based User-Centered Design [STL99], we need a different approach than broadcasting. We present some benchmarks taken during the last 3 years and identify some shortcomings of current approaches

Who is Using the Internet?

The diversity of information on the Internet, and especially on the Web is impressive. We can find answers, full articles or at least comments about virtually any question or topic that crosses our mind; in fraction of a second. Besides searching, Internet has served in connecting the world across all domains and nations. It is quickly spreading across research, government and industrial communities as well as at personal level. "Pew Internet & American Life Project",

undertaken by Pew Research Center "http://pewresearch.org/" estimates in their large scale report [HR02] that in 2002, over 72 million persons in USA alone went online everyday, and that 29% (over 20 million persons) of them used a search engine every day to find information.

The Rate: An Exponential Growth of the Internet?

The direct estimation of the Internet size or dynamics can be a tricky task. We can find numbers that give good indications about it in search engines. They use web crawlers [CAL02], [LV03] to access as many pages on the Web as possible using exhaustive search and return them to search engine to be used in building indexes for keyword search. The increase in index sizes can give use an idea about the rate of Internet growth.

Search Engine Watch (www.SearchEngineWatch.com) has published several charts about index size growth of some known search engines [SEW02] and [SEW03]. Between 1996 and 1999 the number of pages pointed at by search engines grew from 20 to 150 million pages. In 2001 the reported number of pages reached 600 million pages, and then grew to 1600 million pages the following year. Instead of reaching saturation as anticipated, the size jumped to 3500 million pages within another year. The last reported numbers - almost a year later- reached over 8100 million pages. While these numbers are growing at a staggering level, they don't seem to be leveling out soon. Figure 54 visualizes the approximate rate of this growth.

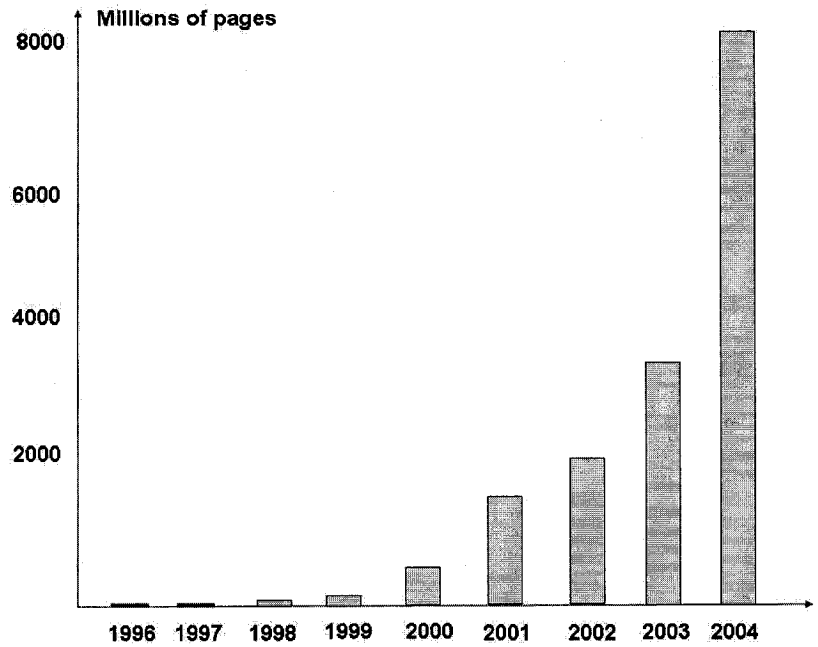


Figure 54: Internet indexes growth

While these numbers give a clear idea about the rough size of the Internet, and its exponential growth rate, we can not rely on them for an accurate estimation of the actual size of information on the web for several reasons:

-The indexes point to selected pages only. Different elimination criteria are used to disqualify pages from inclusion on different index

-The number of pages gives no idea about the amount of information on each page, or its contents

The study [Gaf05c] provides more details on this point and discusses their relevance to the growth rate

The Size: "How Much Information 2003?"

This was the title of a major study at the University of California at Berkeley, UCB in 2004 [LV03]. The project spanned several years and involved many professors and students. They gave accurate estimations of many aspects of the Internet and the amount of information in it for the year 2003. The study estimates that we

produced 5 exabytes of new information in 2002 alone and that –more significantly- the annual rate of growth of new information is 30%. This means that we ware producing 11 Exabytes of new information in 2005 (please note that this is not the entire body of information, but rather the new information produced in 2005 alone). While we can explain that an Exabyte is 10^{18} bytes, it is really tricky to perceive this size of information. To translate it to a more conceivable figure, we used the estimations provided in the same study and simply calculated the following:

-A library floor of academic journals = 10^{11} Bytes
-11 exabytes = $11 * 10^{18}$ bytes
$(11 * 10^{18})/10^{11} = 11 * 10^{7}$ = **{110 * one Million}**

This shows that 11 Exabytes of information –if printed in books and journals- would need one million academic libraries of 110 floors each (higher than the Empire State building) filled with books in each floor. This explains the finding by the same study that only 0.01% of the new information is printed on paper. It would be simply impossible to manage the whole amount if printed.

The UCB study points out an important difference between "surface web" and "deep web". The first refers to static pages stored directly on servers as HTML files with fixed contents. The Latter refers to web pages which are only generated to the user on demand (at run time) while their information contents are normalized and stored as other information files (like XML) or in databases. Contents Management Systems (CMS) are emerging trend to increase our ability to manage information contents of large web sites. In our system (Chapter 5), we also propose to move patterns from text files to database as one of the much needed improvements

While more information is better, we need to be able to handle them effectively. Putting more information on the Internet may flood the Internet space and make it

harder to find quality or relevant information faster. We also run the risk of not finding some existing information at all. Searching information might become more difficult and search index will definitely grow bulkier and harder to maintain and update. Similarly, new information will have less visibility in larger body of information. We should be aware of the consequences associated with this unprecedented and unconceivable growth, and readjust our approaches towards the use of the Internet accordingly. We will focus on some of these consequences on patterns in the next section

Pattern Broadcasting on the Web

The dynamics of Internet were paralleled by a similar -but much smaller- growth in pattern research. Looking at the growth of the Internet, many pattern authors have opted for the Web as the best means to broadcast their patterns to prospective pattern users. Many research projects and experts posted patterns and pattern-related articles on the Web to ultimately reach mainstream designers outside the research community.

As part on an experiment, we asked 7 groups of participants to collect as many pattern collections on the Web as possible within a specific time. Several collections were not found by some groups, and no group was able to build a universal set. Some collections were not found by any of the seven groups. The common complaints were about being lost in huge number of links -some of them going in closed circles of hyperlinks- and the huge number of irrelevant search results returned by search engines. Some web pages contained over 70 links each and some of these links pointed to other hyperlink-loaded pages. After few clicks, some users simply lost track of their search ground. Parts of the findings are presented in Chapter 2. Further study is needed to evaluate these results and find if users just skipped some collections out of frustration, or due to being overwhelmed, or if some collections are really hard to find.

Based on the information collected by the participating groups, we estimated the total size of pattern-related information on the Web to be less than 50 Megabytes. Similarly, we estimated a size of 250 megabytes of information currently available in books. In an average academic library, much more information is available, but there is often an intricate cataloging system that organizes them, and allow for effective search. This is not the case within the pattern community. There is no system available for organizing pattern material at large, besides linked pages or lookup by keywords. In the section "Keyword Indicators" later in this appendix, we will look into the inadequacy of using "pattern" in search by keywords.

We also ran several benchmark tests within the 3 years on search engines and online book stores using common search keywords. The following trail of results is one example of what we obtained from one search engine using the same word:

29,700,000 matches in 0.73 seconds (on 02/2003)
35,300,000 matches in 0.28 seconds (on 02/2004)
43,500,000 matches in 0.19 seconds (on 02/2005)

The number of matching web pages –in tens of millions- shows that it is impractical to look for patterns on the web by keywords. Too generic keywords returned millions of results, and too specific key words made several relevant Web pages disappear. Web Pages use different taxonomy for their research. As discussed in the Thesis, there is no common agreement yet on the taxonomy or classification of patterns and their related domains. Appendix F provides an overview of some of the emerging standards like ISO and ACM.

We also observe that the search speed improved (from 0.73 down to 0.19, and for more results), obviously due to improvements and investments in new hardware technology. But the impracticality of the number of results is ignored.

No human can look up 29 million results, let alone 43 millions. There is no point for us to keep improving our storage and processing capability, Internet infrastructure, network bandwidth and index sizes just to get several more millions of search results. We have to solve this scalability problem as well. The software community is starting to become aware of this emerging challenge. A similar study in an IEEE journal in 2004 referred to the same problem [Luc04]. The author indicates that we assume that search engine ranking directs us to the most relevant and informative Web pages, but that was not true. He explains that in his search attempts, the first few hundred pages were "useless" and that the actual page was possibly "somewhere down past the million mark in the list". We believe that we all share the same misleading perception to rely on the ranking of the search engine. We can always use refined or specialized search, but as we proposed in the term "pattern visibility", in the absence of common taxonomy, more relevant pages will disappear from the results as we use more specialized search words.

Keyword Indicators

A major contributing factor is the fact that the word *pattern* is used in almost all areas of life. The Study at UCB uses a "keyword Indicator" to estimate the functionality or the context of each page by the presence of some keywords in the page. For example if the page contained the word "search", this may indicate that the page belongs to a complex Website, and using "login" or "password" indicate protected pages.

We need to evaluate the adequacy of the word "pattern" as a keyword indicator to be used in simple or complex search for pattern-related pages. The Collins Cobuild Bank of English [CC01] "reflects the whole range of today's language". It provides one of the largest bodies of English language corpuses, which is used to keep track and evaluate the English language dynamics and use. It currently has about 500 million words in its archive, and is constantly collecting and analyzing most of new English texts. While there is a large and rich vocabulary,

English uses fairly small number of words for most purposes. An average dictionary will have few tens of thousand entries. Cobuild selects as few as 14600 words as those that "most people come across everyday". Moreover, it provides a 5 point rating of the commonality of these words; the frequency of how people use each one. A 5-point word is used in almost every sentence we write (e.g. **is**, **has**), and there are 680 words in this group. Unexpectedly, we found that the word pattern is at the next highest level of popularity, receiving 4-point rating. The total number of this group is only 1040 words. This puts pattern at the same popularity level as other words in this group like "**will**", "**object**" and "**normal**" which we use in literally every article. This alone might explain the huge number of pages returned from any web search as we look for patterns, but strongly indicate the inadequacy of the word in a common Web search.

Furthermore, one thesaurus [Rog99] gives 38 other words that we use interchangeably with the word pattern, like **shape**, **type** and **copy**, which means that people simply replace any one of these commonly used words with the word "Pattern" in different contexts. This observation draws attention to the misconception that broadcasting good research on patterns will guarantee that it will reach the mainstream developer. As part of this study, Chapter 2 discussed some of these misconceptions in details.

Looking into the popularity of the word "pattern", we can explain why any keyword search that has it will return colossal number of hits, most of them are irrelevant to pattern community.

To further investigate this aspect, we can measure the "total size of pattern information on the Web, $\mathbf{P_i}$" compared to the "total size of all information on the Web, $\mathbf{W_i}$". From the UCB study [LV03] we can calculate the total body of information on the Web, $\mathbf{W_i}$ to be over 785 Petabytes ($785 * 10^{15}$ bytes). We earlier estimated $\mathbf{P_i}$ to be $50 * 10^6$ bytes. We can calculate the ratio $\mathbf{P_i/W_i}$ to be

around 10^{-10}, or 10 orders of magnitude less[11]. From that we conclude that since the word pattern is very popular as a general purpose word, and by looking at the small size of actual pattern information, there is only an insignificant probability of finding relevant pattern information using random search on the Web. Many new patterns remain invisible to mainstream users and do not show in a normal search activity by interested new users.

The problem of effectively locating web pages has been recently identified in other studies as well. University of California at Berkley [UCB00] provides another study that proposes Web pages to be assigned a lexical code to make it easier to locate them using keyword search. We need to investigate these approaches and popularize them among pattern community to improve the broadcast effectiveness.

Automated Search

The Internet is growing so much and so fast that it is becoming a double-edged sward. As we have demonstrated, it can be useful if we sufficiently understand its dynamics. It can also be misleading if we just keep dumping more knowledge into it than it already has, hoping that whatever target audience we have in mind will find our information "easily". The study shows that this might not be the best approach, especially for patterns. Patterns have to be able to find their users or at least make themselves readily visible on the Internet. We need to find other search methods than random search which is starting to show some weaknesses with the growing size of both the Internet and the number of pattern collections.

---

[11] In this calculation, we compared similar values of Web pattern information and Web information. In another comparison, we compared two other but also comparable values: total pattern information (Web, journals and books) to the total information body we have today. The results were close, at 11 orders of magnitude.

Solutions

Pattern communities and groups already exist as one solution towards guiding users in their search for patterns, but we have shown that they are not well organized, and they randomly point to each other back and forth. They also have high fan out where each pattern page can have tens of links, each is loaded with many other links, some pointing back to previous pages. Users end up in frustration as they keep going in closed loops of links, or they may simply avoid search attempts altogether.

The study proposes to modify the question "**who** can go through 43 million pages looking for anything?" into "**what** can go through 43 million pages looking for anything?" and the obvious answer would be "A Software". Based on the study, we propose to separate actual patterns from the rest of pattern literatures, and to build suitable pattern semantics to allow for interoperability with software, not only for lookup, but also for reuse. We see them as "smart patterns". Broadcasting patterns as text-based Web pages on the Internet does not result in effective search and reuse. We can see the advantages of storing smart patterns in a database that allow for accessibility and interoperability with other software. Depending on the user's needs, the concept of pattern database is applicable both for local database as well as for a community-wide database on the Internet. We have demonstrated the feasibility of both approaches. Technical details are presented in Chapter 5 of the thesis.

Appendix F

Key Terms and Concepts

The difference between
the almost-right word and the right word
is the difference between
the lightning bug and the lightning

Mark Twain

Similar terms have different definitions and understanding in different disciplines. [Som01] explains *"there is immense scope of misunderstanding because of the different terminology used..."* When it comes to interdisciplinary work, the lack of common agreement becomes apparent and sometimes confusing. Special attention is required to reduce this misunderstanding. In this section, we define and discuss some terms and explain the view that we are adopting.

System Engineering: This term is used under very broad definitions, depending on the system being built. [Som01] defines it as "*an interdisciplinary activity involving teams drawn from different backgrounds*". Although many phases and activities of the system engineering process are similar, regardless of the system, we should be cautious about the distinctive nature of software-based system engineering, and carefully identify its unique elements. [Prs01] specifies that "*a high-technology system encompasses a number of elements: software, hardware, people, database, documentation, and procedures*", and he emphasizes that "*system engineering helps to translate customer's needs into a model of a system that makes use of one or more of these elements*". This is a fundamental definition to our work. We are working on five of these six elements, namely software, people, database, documentation, and procedures, and we are definitely using the sixth; namely hardware.

Software-Based System Engineering is concerned with applying system engineering concepts to build a software system within the big picture of a complete system. Somerville [Som01] refers to it as computer-based system engineering

Interaction Design: Smith and Tabor [ST96] define it as "*Designing the way people interact with objects and systems, especially with computer software*". The fact that computer software is "virtual" was a major driver for a lot of research. Winograd [Win96] asserts that "*interaction design cannot dispense with scientific method and engineering knowledge*". He emphasizes the essentiality of merging them, but also the difficulties of doing that.

The domain of Human Computer Interaction HCI: As an emerging domain, there is no concrete agreement yet on the definition and boundaries of HCI or on the description of its interaction with other domains, many of which still have unclear boundaries themselves. We give some commonly used definitions that - while not the official or unanimously adopted ones- give a clear understanding of some terms.

Human-Computer Interaction (HCI): [HCI02] defines HCI as the study of interaction between people (users) and computers. It is an interdisciplinary field, relating computer science, psychology, cognitive science, human factors (ergonomics), design, sociology, library and information science, artificial intelligence, and other fields. Interaction between users and computers occurs at the user interface, which includes both hardware (i.e. input and output devices) and software (e.g. determining which, and how, information is presented to the user on a screen).

As one of the authoring and most influential organizations in software engineering, the Association for Computing Machinery, ACM (www.acm.org)

defines HCI as "*a discipline concerned with the design, evolution and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them*" [HCG+96]

ACM attempts to delimit the scope of concerns of HCI, and to specify its connections with other fields. It emphasizes that Human-Computer Interaction has science, engineering, and design aspects, and is concerned with

-The joint performance of tasks by humans and machines
-The structure of communication between human and machine
-Human capabilities to use machines (including the learnability of interfaces)
-Algorithms and programming of the interface itself
-Engineering concerns that arise in designing and building interfaces
-The process of specification, design, and implementation of interfaces; and
-Design trade-offs

To help establish some internal boundaries within the domain, five interrelated aspects of human-computer interaction are identified:

1- The nature of human-computer interaction, denoted "N"
2- The use and context of computers, denoted "U"
3- Human characteristics, "H"
4- Computer system and interface architecture, denoted "C"
5- The development process, denoted "D"

ACM provides a visualization of the main HCI constituents and the relationships between them as shown in Figure 55.

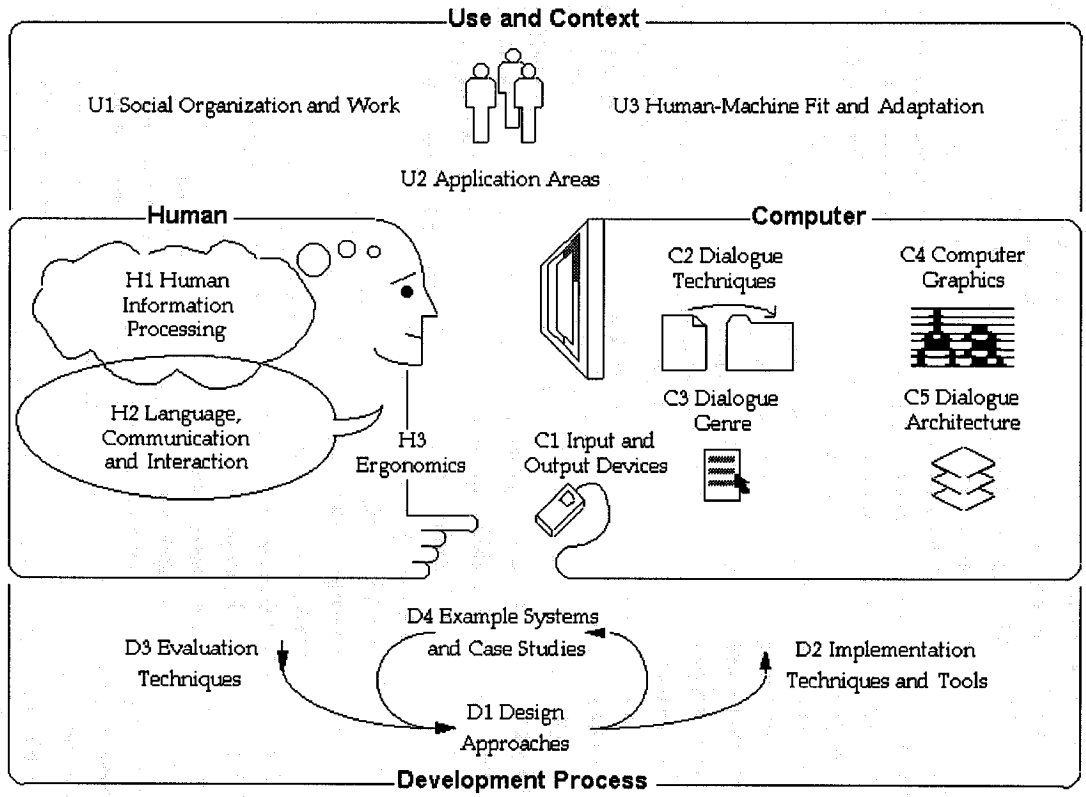


Figure 55: HCI as depicted by ACM

Expanding the Classical View of HCI:

While the classical view of **Interaction** is a **Human** using a graphical user interface running on a desktop **Computer**, this view is changing rapidly in all its three constituents; H, C, I. We need to generalize it into a less restrictive view:

The "human" is definitely changing from the stereotypical highly educated and well trained user into the more relaxed view which include people from many other domains, with no prerequisite on education, training or experience. This also allows for the inclusion of special groups like children and persons with disabilities.

The "computer" is also changing due to the unprecedented technological advances in hardware and the drop in prices. Many consumer electronic gadgets are miniaturized and computerized. A small device like a personal digital assistant (PDA) that fits on the palm of a hand has more computational power, storage capacity and screen resolution than the desktop of just a decade ago. The integration of computers into other sophisticated systems like the cockpit of modern aircrafts and cars shows the extent of the word computer.

The "interaction" is equally expanding from WIMP (**W**indows, **I**cons, **M**enus and **P**ointing device) into stylus, voice, eye tracking and gesture interaction. The concept of Graphical User Interface (GUI) is preceded by Tangible User Interface (TUI). Just a year ago, people could not imagine "Edible User Interfaces" or EUI, an ongoing research at Stanford University. In April 2005, it was presented at an ACM conference, CHI 2005 [May05]. Again, the spread of computers into more complex domains forces the interaction style to transcend its classical boundaries.

HCI, CHI, and IHM:

Different literature uses either the HCI or the CHI acronyms. Generally speaking, they refer to the same thing. HCI is now more widely accepted among researchers while CHI is becoming outdated. In the francophone world, "Interaction Homme-Machine" is a close French translation of HCI. It follows the same trend of "human" preceded by "computer". The computer is often referred to as a computational machine, or simply a machine.

Interface

In its widest sense, the term interface denotes the logical or physical layer that separates two objects. In computer science, we often regard an interface as the point of communication between two objects. For example, in object-oriented programming, an interface defines a formal way of interacting with a program

object and is often associated with abstract data types and encapsulation. Each object can have one or more interfaces associated with it, hence offering multiple points of communications.

User

A user is the ultimate source or destination of a message or the ultimate cause or effect [Lay03]. In computer science, a user is not necessarily a human. A software application can have another software or non-software system as a user. The UML uses other terms like a role to abstract the concept of a user into a functional point of view.

User Interface

In software engineering, we can view the user interface as the collection of components of a computer system that the operator uses to interact with the computer - the screen, keyboard, mouse, touch controls, as well as the virtual controls displayed on the screen. Unlike the broader definition of *"user"* in conjunction with software in general, the term user in "user interface" implicitly narrows the interaction with computer system to a human.

In HCI domain, we refer to the user interface as the specific hardware and operating software by which an operator executes procedures on computer and the means by which the computer conveys information to the person using it; the controls and displays. This view clearly put emphasis on the **components** as well as the **interaction** itself.

End User

In software domain, an end user is a person who interacts with a software application after it has been developed. The person uses the application to accomplish specific tasks, but is not interested in computers per se (Brad Meyers and David Maulsby, http://www.acypher.com/wwid/BackMatter/Glossary.html). They are also referred to as interface users, or simply users.

Pattern User

Patterns are intended to disseminate design knowledge from experts to other designers. A pattern user is a designer who uses a pattern as a guide to solve a design problem similar to that explained in the pattern.

Usability: ISO 9241-11 defines usability as the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use.

In this context, we can define the following three qualifiers used in the definition of Usability:

Effectiveness

The accuracy and completeness with which specified users can achieve specified goals in particular environments. Effectiveness measures the degree of completing the tasks, regardless of the effect on resource expenditure.

Efficiency

The resources expended in relation to the accuracy and completeness of goals achieved. Efficiency often refers to the operating costs, but can also refer to production. A more efficient car takes less fuel to drive, and more efficient software take fewer resources to accomplish the work.

Satisfaction

The comfort and acceptability of the system to its users and other people affected by its use.

It is important to emphasize the difference and the relationship between software effectiveness and efficiency. More effective software will help users achieve more goals in a more accurate way. More efficient software will consume fewer resources to do the work. In many systems, the tradeoff is apparent between

both parameters. It is often the case that we can achieve most of the goals at an acceptable cost, but the cost grows dramatically when we need to achieve all goals with great accuracy. If we added the factor of satisfaction, we can say that it might take more resources to operate more pleasant software. The study of usability works to find a good equilibrium between them.