A Software Customization Framework

by

Jeffrey William Michaud
B.A., University of Guelph, 1999

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In the Department of Computer Science

University of Victoria

ABSTRACT

Highly configurable software systems are being built to address the ever increasing demands of users and businesses. Successful systems that support the needed customization are difficult to design, develop, deploy and manage. To help address some of these difficulties we have developed a software customization framework. The framework includes a taxonomy of customization concepts as well as how customization plays a role for the various stakeholders in a software project, such as the software architect, software designer, and end user. The framework also includes a model of the software customization process produced by combining existing models from program comprehension and HCI. We add to the framework guidelines generated from this model and details of how to include customization through the system's software design/architecture as well as through interaction styles. We then apply the framework to the Map Explorer project to demonstrate the framework's usefulness and effectiveness.

Table of Contents

List of Tables

List of Figures

Acknowledgements

I would like to acknowledge the support given to me by all of those who have aided me in developing my thoughts and ideas that are present here in this thesis. Chief among my supporters is my advisor Peggy. My thanks also go out to my fellow grad students in the CHISEL lab for their feedback and for making all the research activities exciting, fun, and challenging. Special thanks go to Neil and Liz for proof reading and providing guidance in the final stages of writing.

I also would like to express my thanks to my friends and family for pushing me along and encouraging me to not give up. And, to Ivan, mi rey, thank you for your understanding, your patience, and for filling my life with more than school and work.

Chapter 1

Software Customization, the Problem and the Approach

The majority of software applications developed today need to be customizable. The reasons for this are diverse: the accelerating pace at which business needs are changing; new hardware innovations; the requirement to support multiple platforms; and increasingly demanding users. There are, however, many problems and challenges linked to projects that involve considerable use of customization that need to be tackled. It is these challenges that this thesis addresses.

Our own work in software customization began with a struggle. While working on a project called Map Explorer, we discovered that although we understood that the tool needed to support extensive customization, we were at a loss for determining what shape

that support should take, how to implement it, or even the best way to deploy it. We then learned that the research community and the best practices of business did not provide much aid to help us with our difficulties. Next, we endeavoured to solve this problem ourselves – we constructed an understanding of software customization. We produced a software customization framework which exposes and explains the concepts of customization, gives it a proper vocabulary, models its processes, and provides guidelines that we have found to be useful in our Map Explorer project.

In this first chapter, we introduce software customization with an overview of its benefits and challenges. We then provide further details of our approach to address these challenges and describe how we developed our software customization framework.

1.1 Introducing Customizable Software

Beginning with the very first software program, customization of software has been required. Software developers have continuously been requested to empower their tools to be amenable to change. Users, contexts of use, requirements, and operating environments are in a constant state of flux meaning there a high necessity for adaptable and configurable software. The need for such flexibility has also been increasing in recent years. The degree to which businesses rely upon computers and software has grown significantly. With this growth, the size and diversity of the user population has also increased. To address the challenges of engineering software to meets the needs of today's businesses and today's users, software engineering is maturing and moving

towards a more appropriate paradigm. A statement by Paul Dourish from Xerox PARC expresses this:

> "... we begin to see delivered systems as not being closed and static, but rather as infrastructures for further specialization, refinement, and end-user design. They provide a framework within which users can change and adapt the basic system to their own patterns of usage" [1, p. 45]

As Dourish describes, part of this maturing process is refining the technique of developing software while increasing the quality of the product. Producing a customizable software system is the result of a disciplined application of the methodologies and techniques in software engineering (and other disciplines) and is often considered the elegant solution. In the next section, we take a detailed look at the advantages customizable software offers.

1.2 Benefits of Customizable Software

Highly tailorable software systems offer many attractive advantages. These benefits will differ according to the various stakeholders of the project. In this section we identify how software producers (Designers and developers), software maintainers, and users can profit from customizable software.

1.2.1 Software Producer Benefits

Software producers will realize several benefits from developing a tailorable software system. The first benefit appears during the design phase of the application. It is often the case that the requirements gathered for the system contain conflicting needs from different users or user groups. One method of addressing these conflicts is to incorporate the necessary flexibility within the software tool so that it can accommodate each user.

Another advantage of customizable software is enabling the software application to meet the undocumented requirements of the users. As Bonnie Nardi noted in her research on software customization "[It] has been shown time and again, no matter how much designers and programmers try to anticipate and provide for what users will need, the effort always falls short because it is impossible to know in advance what may be needed" [2, pg 3].

A third potential benefit is increasing the longevity of a software system. A tailorable software system offers the possibility that the system's functionality can adapt to the user's changing needs and requirements over time and thus stay current, applicable, and useful longer [1]. An example of such a system is IBM's Eclipse, an integrated development environment for software developers that incorporates special features to design program extensions that can be easily distributed and installed within any user's Eclipse tool [3].

Software producers are also realizing they can make their products appeal to a wider segment of the market by promoting a comprehensive set of configurable features. In fact, many companies are demanding that new software systems be highly configurable. Companies have come to realize the benefits of software systems that can adapt to their current environments and integrate with their existing systems and processes [4]. It is very important to note that the notion of a highly configurable system is different than a product line architecture [5]. In a product line architecture the software producers have control over every variation of an application. The type of project we are focusing on in

this thesis is different. Here we are looking at the situation where clients and third party companies may customize the application for their own purposes and not necessarily coordinate the various versions into one collection of products.

1.2.2 Software Maintainer Benefits

For the maintainers of the software, additional benefits can be noted. Typically maintenance tasks consist of bug fixes and developing the desired enhancements chosen by the users. Statistics show that this work can take up as much as 50-70% of the total software engineering time spent on the project [6]. The first benefit tailorable systems can offer is a potential reduction in the amount of maintenance needed on the software system. Some of the maintenance tasks may be offloaded to local developers. Secondly, when faced with a maintenance task involving system enhancement, highly configurable systems offer the maintainer the option to utilize a customization mechanism to develop the enhancements. Such mechanisms may offer to the programmer easier and faster methods of developing and distributing the enhancement and thus take less time and effort.

1.2.3 User Benefits

Users will also benefit from customizable software in several ways. Firstly, functionality can be changed, added, and deleted as required, improving upon usability and potentially productivity [7, 8]. Many of today's users are becoming accustomed to using computer systems that are specifically tailored to their needs and are demanding that all their software contain personalization features [9].

Personalization (the result of customizing the tool) provides several benefits for the user. For example, research indicates that presenting a user with an application that the user feels has been specifically built and tailored for them is comforting and reinforces their sense of individuality [7]. Research also indicates that such personalization can instil a sense of empowerment in the user. Customizable software "...allow[s] users to invest the world with their meaning, to enrich the environment with the fruits of their vision and to use them for the accomplishment of a purpose they have chosen" [7, p. 398].

Customizable systems can also lead to more knowledgeable users. Users that participate in the design process of an application (whether by specifying their specific needs to a developer or trying to implement customizations themselves) can gain valuable knowledge and insight into that application as well insight into the application's domain. Simply using the software may not achieve the same effect. "Wisdom is a kind of 'meta-knowledge' of processes and relationships gained through experiences. It is the result of contemplation, evaluation, retrospection, and interpretation." [10] Wise users will make better and more efficient use of the system as well as be able to better communicate their thoughts and concerns about the system to others.

Increased communication among users and developers is a common side effect of customizable software and can eventually lead to the creation of a culture around the application and even a user community. These communities can be an undeniable influence over the use and success of the software [2, 7, 11]. First, these communities provide users with a resource to field questions and garner support. Second, Mackay

noted that new standards can be established and perpetuated when users share customization settings with others in the community [12]. Another important benefit associated with such communities is their ability to promote the tool and increase the demand for it. Finally, these user communities become an invaluable resource to the software developers when gathering new requirements or when looking for users to test and try out new ideas and tools.

Clearly, highly customizable software offers many benefits. Unfortunately, successful customizable software systems are not easily produced. In the next section we will discuss the issues and problems that are commonly linked to customizable software.

1.3 The Problem

One of the major issues with customizable software is the lack of consideration for customization at the very beginning of a project, at the requirements stage. It is very difficult to elicit requirements for the features of software; this difficulty multiplies when choices have to be made as to how those features will be configurable and customizable. In fact, this is a noted issue and gap in the requirements engineering field [13]. This difficulty can lead to inaccurate requirements and thus lead software developers and architects to make poor decisions. For instance, customization support may not be included for an important feature or the type of customization support provided is not well suited to the domain or users involved.

Even if customization is properly considered from the beginning of the project, there are still many problems to overcome. One problem that we have experienced in our own

projects is that often software customization crosscuts all aspects of software development – from the architecture design, to the development phase, and through to the user interface design. Unfortunately, our current software processes and tools tend to focus our efforts within these distinct phases and there is no coherent, overarching description of customization that can provide guidance to the stakeholders throughout a project.

Without a guide to the customization process, inexperienced teams with respect to customization are often not aware of the decisions that need to be made to deliver customizable software and thus may make poor choices. For instance, when building customizable software, it is important that the following decisions are carefully considered:

- What features should be customizable?
- In what manner will these features be customized?
- How should the customizable features be implemented?
- Who is qualified to perform the available customizations?
- When will these customizations be made?

Unfortunately, the research literature does not have much guidance on the answers to these questions and how to manage these difficulties.

Furthermore, documenting and describing the needs for customization and the resulting design decisions are often sorely lacking. This can lead to poor decision making with regards to the software's design as well as lead to software that is difficult to maintain.

With our current tools and methods, it is often impossible to relate architecture designs to the decisions made concerning the customization facilities provided for the end-user.

In terms of designing a technical solution for a highly customizable system, there are many options to choose from, such as object-oriented design, design patterns, components, and others. These techniques have proven themselves over time and have even become topics to be included in computer science education; however, there is no integrated understanding of how to appropriately choose between these options. The tradeoffs between different implementation paths are difficult to choose from because priorities are difficult to establish without an assimilated view of the customization process and its stakeholders.

1.4 Our Approach

During our reflection upon the issues related with software customization, we identified that a broad understanding of the software customization process is lacking. Our approach is to develop a software customization framework to address this need. This framework encompasses many of the relevant factors needed to make software customization related decisions in addition to a set of suggestions to get initiatives in software customization started.

To produce such a framework, we look to research for ideas, guidance, and inspiration. As noted, this endeavour not only contains perspectives relevant to technical issues, but also to social issues. Software customization is a process involving different people with different needs and different skills. Therefore, our research takes us through many

diverse topics such as software architecture, software modelling, human computer interaction, program comprehension, end user programming and others.

We begin the construction of the framework in Chapter 2 by developing descriptions and classifications of the different kinds of customization that exist, how they can be achieved, who is involved in the process, and the various stages of the customization process. In Chapter 3, we continue the development of the customization framework by developing a model of the software customization process. This model builds on existing models from Human Computer Interaction (HCI) and program comprehension. We then use this new model to reflect upon software customization and produce some general recommendations for future stakeholders involved with highly configurable software projects.

In Chapter 4 we explore various implementations of system architectures that will lead to a tailorable system. Chapter 4 contains a description of how object-oriented design, components, design patterns, and software frameworks can be useful tools for the software designer as well as for the persons responsible for customizing the system through modifications of the source code. At the end of this chapter there is no conclusion concerning the 'right' way to architect a customizable software system. However, we suggest that designers of new systems should consider these design methodologies and take into consideration our comments and suggestions.

Chapter 5 complements the work of Chapter 4 by focusing on how to customize a software system without direction manipulation of the system's source code. In this chapter we describe the research concerning end user programming and other interaction styles. A comparison of the most common interaction styles is presented. Through this work, we extract the set of ideal characteristics for interaction styles in the context of software customization.

At this point we have a framework for software customization that contains:

- a classification of software customization features, methods, stakeholders, and stages;
- a model that describes the process of software customization;
- a set of generalizations based upon that model that can offer guidance when deciding how to provide customization support in an application;
- a set of recommendations for providing customization via the system's implementation; and
- a set of recommendations for providing customization via the system's interface or other interaction style.

In Chapter 6 we provide further details of the Map Explorer project. First, we describe the frustrations we experienced when we attempted to conceive and design the tool as a highly configurable system. Then, we describe how the Map Explorer project was positively impacted by the development and application of our software customization framework. We describe how it organized our team members, enabled us to make decisions regarding customization, and provided useful guidelines for the implementation of the Map Explorer tool.

In Chapter 7 we discuss the results of the Map Explorer project, review the contributions this thesis offers the research and software engineering community, and provide details of some ongoing and future work.

Chapter 2

Describing Software Customization

We begin our research with an examination of the software customization process. Our goal in this investigation is to delve into the concept of software customization in order to discover and understand the elements that customization involves. In this chapter we structure this exploration by endeavouring to answer the following questions:

- *What* is customized during software customization?
- *How* is software customized?
- *Who* performs the software customizations?
- *When* does software customization occur?

By answering these questions and organizing the results, we will establish a structured terminology of software customization. At the end of this chapter this terminology will

represent our increased understanding of software customization as well as enable concise discussions in the remaining chapters.

2.1 What about software is customized

To improve our understanding of how software is customized, we need a structure for defining and extracting the different types of customization. Inspiration for this structure comes from the work of Wasserman who recognized three different levels upon which two or more applications may be integrated: data, presentation, and control [14]. The similarities between customizing a software system and integrating multiple software systems make using these categories to classify software customization appropriate.

2.1.1 Data Customization

Many desktop applications have a standard data exchange format (a plain text format such as XML, or a binary format such as Microsoft Word's .doc). Data customization enables an application to adapt the data exchange mechanism to exchange information with virtually any environment or application. Within an application's data stream (file or networked) there are two types of data: data and meta-data. Data encodes the concrete information acting as input and output for the application, whereas the meta-data is data which describes this data and may include processing instructions. Both the *format* and *content* of the data and meta-data can be customized.

Customizing the *format* of data and meta-data may be required to make the application read and write compatible with different or additional information sources. *Content* customization is typically applied to meta-data. By changing the meta-data, one can

effectively customize how the data, which is described by the meta-data, is managed and processed.

A common example of meta-data customization appears in web page authoring. The standard format for web pages is HTML. In HTML, the presentation of content (data) is managed by incorporating tags (meta-data) around elements within that content. Changing these tags can change how the web page is presented. However, not all data customizations involve working directly with a file. For example, email applications allow us to specify, in the application interface, some meta-data about the email we are writing. For instance we can specify that an email is of urgent priority as opposed to normal priority email. This metadata is encoded into the email and, when the recipient views the email, they are made aware that the email is urgent.

2.1.2 Presentation Customization

The visual appearance of an application is another major category where customization takes place. We call this type of customization **presentation customization**. With the emergence of a plethora of online service providers such as Yahoo and MSN that capitalize on web server technology to enable a personalized experience, presentation customization is progressing from being a 'nice to have' feature, to a requirement in many systems [9].

Changes in the visual presentation of the application can also occur in two ways. The first is *information architecture. Information architecture* describes how information is organised and accessed through the display.

> "Information architecture determines what content and data is included, where it resides, what it is called, how users find and access it, and to a certain degree, how it appears in a display. It involves primarily the organizing and naming of things and ensures that users find what they want without getting lost or distracted while navigating " [15]

Adjusting the organization of menus and buttons and the layout of windows within an application are examples of customizing the *information architecture.*

When designing a new application, the *information architecture* is generally done before the *graphical design. Graphical design* affects the aesthetics of an application since it involves the look and feel of the application. This can include the theme or metaphor of the design (fun, serious, or corporate standard) and involves the selection of colours, shapes, and images. *Graphical design* and *information architecture* are dependent on each other and therefore are often complementary. For instance, a group of buttons identified to be used for a specific task can have a similar colour or shape.

2.1.3 Control/Behaviour Customization

Tailoring how an application behaves is a fairly broad concept. Simple tasks, such as selecting the features to include in a view, to more complex tasks, such as writing macros to automate tasks, fall under the umbrella of **control/behaviour customization**. For this reason, customizations in this category are split into six sub-classifications or types. These types are *feature selection, options specification, feature addition, feature enhancement, feature constraint,* and *feature coordination.*

Occasionally, the implementation of a software system contains many more features than is needed by a user to perform their work. The process of *feature selection* involves analyzing each of the system's abilities and deciding whether or not it should be incorporated into the interface. For instance, installation wizards will often let the user select a subset of features, possibly leaving functionality uninstalled.

Once a feature is selected for inclusion, there may be standard variations in how that feature may operate. To customize these standard variations, one would make use of *option specifications*. Option screens (forms using check boxes, combo boxes etc.) are a common method to enable users to perform simple (and foreseeable) customizations. For example, in Microsoft Word the options for each of the features are organized in a collection of tabs (see Figure 2.1).

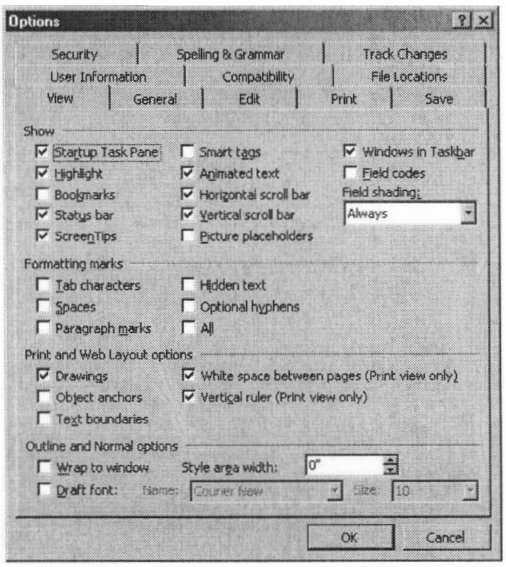

Figure 2.1 Microsoft Word presents each feature's options in separate tabs.

Often it becomes apparent that the required functionality is not available in an application. This scenario requires a customization process called *feature addition*,

which involves the design and building of a completely new feature. Once available, this feature should then be selectable within the application via *feature selection.*

The fourth classification of control customization is *feature enhancement.* In this case, an existing feature almost fulfills the requirements of the user and only needs to be enhanced. For example, it may be desirable to enhance the searching feature of a word processor so that it employs regular expressions.

Contrasting with *feature enhancement* is *feature constraint.* A feature may need simplification to be more usable or not all of its functions may be required for certain users or tasks. For example, an address book feature of an email tool may be simplified by removing the ability to add multiple addresses at the same time.

The last form of behaviour/control customization is *feature coordination.* Each feature of a software system (whether it is selected, added, enhanced, or constrained) is usually intended to work in tandem with one or more features. For example, the preview feature of a word processor is often linked to the ability to print. Customization through *feature coordination* involves changing how two or more features work together to achieve a single task or set of tasks.

2.1.4 Summary of What is Customized

This section categorizes the different types of customizations into three groups: data, presentation and control. Data customizations include modifications made to the format and content of meta-data. Presentation customizations involve changing the logical

organization of the view (information architecture) as well as the aesthetics (graphic design). Control/behaviour customizations include feature selection, option specification, feature addition, feature enhancement, feature constraint, and feature coordination.

Many customizations will involve aspects from data, presentation and control customization. For instance, adding the ability to draw graphics within a word processor is a behaviour/control customization (*feature* addition) that requires customizations in the data category (*format* to encode picture data), and in the presentation category (*information architecture* to determine how to provide the drawing tools and how the user will find them). Customizations that crosscut the difference categories, such as the one above, are likely to be more complex and thus more difficult to provide. Only by ensuring that each type of *what* is supported, will these types of customization be made feasible.

2.2 How is software customized?

The methods that can be used to implement a customization are actually quite varied and can best be placed in two categories. The first category includes all the methods that operate directly on the source code and as such we refer to it as *Source Code Customization*. Customizing a system through its source code is likely the most difficult, but it often offers the greatest amount of flexibility. *Source Code Customization* is discussed in more detail in a later chapter.

The second category includes all the customization methods that do not involve changing the source code and we refer to these types of customization as *Interaction Styles*.

Depicted in Table 2.1 is a list of the most common methods to perform *Interaction Style Customization*. They are loosely ordered according to difficulty, where the first one listed, Option Screens, is generally the easiest to make use of, to the most difficult, Application Generators.

Customization Style	Description
Option Screens	Uses check boxes, combo boxes, etc.
Wizards	Guides user through a set of choices
Configuration Files	Settings are recorded in a separate textual format
Macros	Recorded user actions to automate common tasks
Visual Builders	Interactive diagram
Scripts	Custom or common script language
Application Generators	Interpret domain specific specifications to automate the generation of the software from pre built modules.

Table 2.1 Interaction Style Customizations

Option Screens, Wizards, Configuration Files, and Macros are commonly found in today's applications. Visual Builders and Application Generators are less common. Visual builders typically involve users selecting a set of widgets and arranging them in a staging area. Common examples of Visual Builders are report construction tools and Graphical User Interface construction tools. Application Generators are tools that produce applications for a very specific context. Typically, users can pick from a standard set of options and the Application Generator outputs the program. Since Application Generators are extremely expensive to construct and have limited scope, they are rarely found [16]. Further details on these interaction styles are presented later in the thesis.

2.3 Who will customize and when

To be able to deliver an effective application to a large user base, the application should support customization in the three categories of data, presentation, and control. However,

achieving these customizations will require a large body of skills and knowledge spread across the participants of the project. We believe that in an ideal situation, there are three groups of people involved in successful customizable projects: Designers, Customizers, and Users.

2.3.1 Designers

Designers are the designers and builders of the original tool and are, in a way, the first to customize the application. Designers choose default settings and set the stage for any future customizations. Designers perform their work at the *Original Development* stage of software customization and are responsible for defining three major important aspects of the software:

1. The set of features for the application and how these features can be used to perform tasks.
2. The interaction styles that will be included so that the features described above can be customized without interacting with the source code.
3. The implementation of the system which realizes the feature set and the interaction styles. An important aspect of the implementation is the software's architecture which will determine how the software can be customized through its source code.

2.3.2 Customizers

Customizers are the individuals who take an existing software product and tailor it to suit the needs of a group of users. The term Customizer often refers to individuals given a different label. For instance, Bonnie Nardi describes the people within a group of Users with the inclination and skills necessary to customize their tools (and those of their peers)

as Gurus[2]. There also many companies whose primary business is the customization of existing tools for other businesses (such as SAP [17]). Customizers in this context are often simply referred to as programmers. These programmers specialize in understanding how existing software products work and how they can be adapted.

The role of the Customizer is to understand the needs of the Users and to then adapt the software to meet those needs. The Customizer leverages the interaction styles built into the tool and/or makes changes to the source code to realize the needs of the Users they have identified.

Since Customizers transform a software system from something considered general purpose into something specifically designed for the users, we label this process as *System Refinement*. In addition, it is often the case that the Customizer will be responsible for the installation of the software and may perform some additional customizations at that time. We refer to this phase as *System Installation*.

2.3.3 End Users

End Users are the individuals who are going to use the application to perform tasks. Customizations performed by an *End User* could involve any of the types of customization described above. However, extensive customization is unlikely. The typical *End User* does not have the skills necessary to perform complex customizations while learning more about the application. Typically users do not read documentation nor do they extensively explore the software. Therefore, advanced features, such as those involved with customization, are often never discovered or if known, avoided [2]. Reasons for this trend will be explored further in the coming chapters.

An *End User* may also be involved with the *System Installation* phase. For example, they may be directed by a wizard to perform some needed customizations (typically feature selection) while installing the software.

The distinction between these three groups helps us to appreciate that there are several different roles, skill sets, and needs involved in building, customizing, and using software. However, the actual makeup and arrangement of people involved is often quite different. For example, the Customizers may be the same as the Designers, and sometimes may be the same as the Users. There are many different possible combinations of how the participants in customizable software project will be configured. For simplicity, however, we will refer to the Designers, the Customizers, and the Users as being separate groups throughout the thesis.

2.4 Summary

In this Chapter, we have named and described the elements of software customization which we have summarized in Figure 2.2. With this taxonomy of software customization we have a core element of our software customization framework completed. In the next chapter we continue our construction of our framework by applying and extending existing research to produce a model of the software customization process.

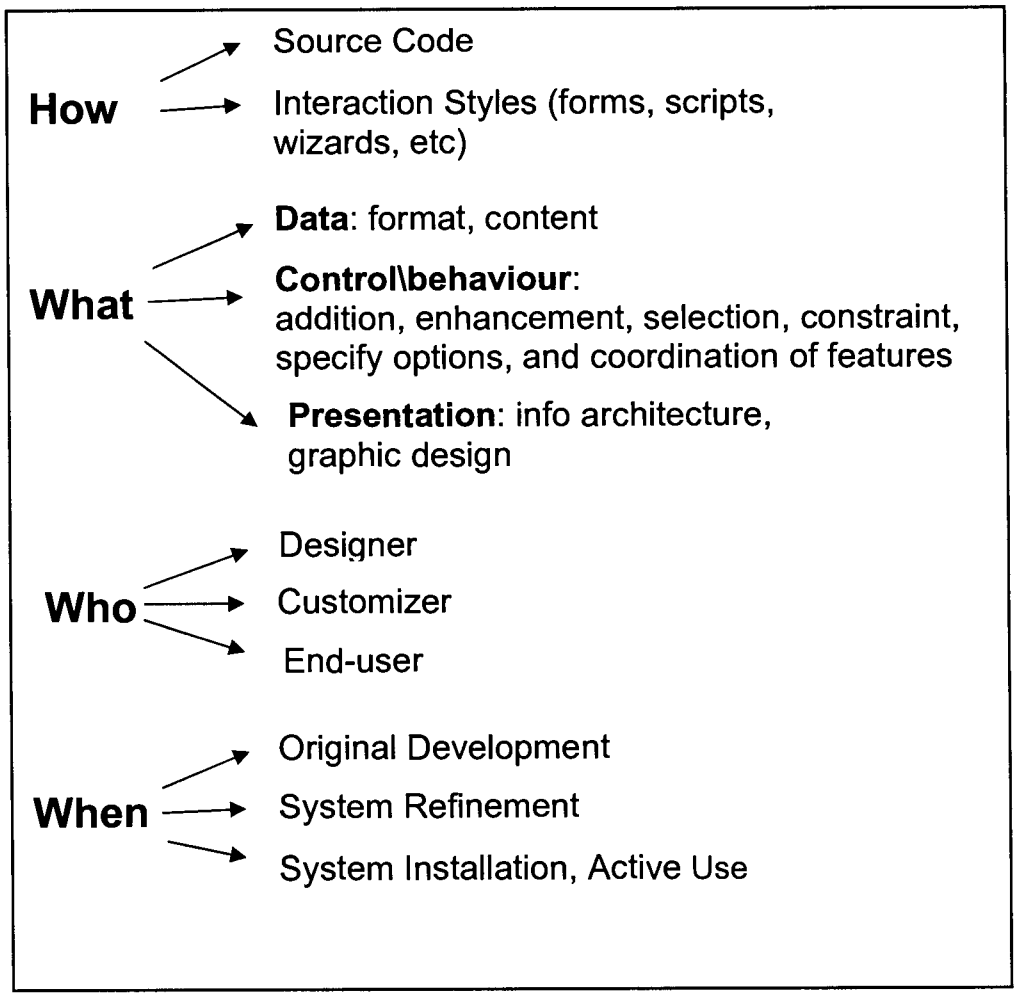

Figure 2.2 Summary of How, What, When, and Who with respect to Customization

Chapter 3

Modelling Software Customization

In the previous chapter we itemized the aspects associated with software customization. Namely, we described the people, the events, the methods, and the objects involved. We now switch our focus to how some of these pieces fit together: we propose a model of the software customization process. We begin the chapter by describing our motivations for producing such a model and then proceed with a step by step description of our model of the software customization process. The chapter concludes with our analysis of the model and a discussion of what we can learn from it.

3.1 Motivation to Model Customization

The concept of successful software customization is not new; customization exists successfully in many of today's software tools. On the other hand, there are many systems and situations where there are failures in this regard. As stated in Chapter 1, the reasons for this are many. We believe that chief amongst these reasons is the lack of an overarching understanding of software customization. Our motivation for producing this model is to provide a structure that encompasses the customization process. We propose that such a model should stretch across the boundaries of software and humans: it must represent how software is constructed and comprehended. We look to the field of research called program comprehension for modelling guidance as this discipline considers aspects of software construction and human understanding.

3.2 Program Comprehension

Researchers involved with program comprehension are interested specifically in understanding how programmers learn software systems during maintenance and evolutionary programming tasks. This is a difficult affair considering the complexity of programming as well as the intricacies of the human mind.

To abstract from some of the complex details, researchers of program comprehension formulate models, of which there are several [18-21]. Von Mayrhauser and Vans have conveniently extracted and presented the common elements of these models in [22]. Specifically, they identified the following three common elements: the programmer's knowledge, the programmer's mental model, and the programmer's expert characteristics. We explore each of these elements further in the rest of this section.

3.2.1 Knowledge

In the context of program comprehension, there exist two types of knowledge: *general knowledge* and *software specific knowledge. General knowledge* encompasses the programmer's knowledge of the world, programming languages, programming principles, algorithms, the operating system, etc. *General knowledge* is completely independent of the projects that the programmers are working on. Contrasting this is *software specific knowledge,* which is gathered by reading the source code and documentation of the software, and includes knowledge about the application's domain.

Decomposing the concept of knowledge further is the chronological separation of knowledge into *existing knowledge* and *new knowledge*. Existing knowledge consists of all general and specific knowledge the programmer has before working on the software project in question. Once work has begun on the software, new knowledge is acquired. For the most part, the new knowledge consists of software specific knowledge such as the software's architecture, the data structures, the control flow, etc. However, during the course of the programmer's work, the programmer may be required to learn a new 'general purpose' algorithm, meaning that some of the programmer's new knowledge may include some general knowledge [22].

The process of understanding the application consists of matching software specific knowledge with existing knowledge until the programmer believes he understands the software. This mapping forms the mental model and may be complete or incomplete. We discuss the notion of a mental model further in the next subsection.

3.2.2 Mental Model

We begin with a general definition of a mental model [23, p. 38] that applies to every day life:

> "Mental models, our conceptual models of the way objects work, events take place, or people behave, result from our tendency to form explanations of things. These models are essential in helping us understand our experiences, predict the outcomes of our actions, and handle unexpected occurrences. We base our models on whatever knowledge we have, real or imaginary, naïve or sophisticated."

This definition supports what was established in the description of knowledge: that new mental models are formed with the help of existing knowledge. Another key point about mental models is that their structure is going to be heavily influenced by context. For instance, in this definition of a mental model from a programming context, we see specific structures that only apply to software [22, p. 45]:

> "The mental model is an internal, working representation of the software under consideration. It contains static entities such as text structures, chunks, plans, hypotheses, beacons, and rules of discourse. A chunk's construction combines several dynamic behaviours, including strategies, actions, episodes, and processes. "

In many of the comprehension models, the mental model is considered the key and central component. The mental model will play a significant role in our software customization model.

3.2.3 Expert Characteristics

The third common element of a program comprehension model is the programmer's expert characteristics. The level of a programmer's expertise in a domain is an important factor in program understanding. Experts in a domain have specialized schemas of knowledge that permit them to view situations from an abstract point of view. This enables them to efficiently decompose and comprehend problems and be flexible in the manner in which they tackle them [22, 24].

3.2.4 Software Customization and Program Comprehension

Program comprehension models itemize information about the mental models, knowledge, and expertise of programmers within the context of software maintenance. The models that result from this work are then used to improve support for programmers performing software maintenance tasks in tools [21, 25-28]. In the next section, we apply these concepts of modeling to develop a software customization model.

3.3 Customization Model

In this section we take an incremental approach to present our software customization model. The first step requires describing Donald Norman's cognitive model which we use as a base for our model. This model provides a simple view of software, software users, and software Customizers. The next steps are to enhance Norman's model as we incorporate additional details. These enhancements integrate concepts from our software customization classification established in Chapter 2, as well as the common elements of program comprehension we just described.

3.3.1 Donald Norman's Model

We begin our modelling task by taking a look at a simple model from Donald Norman, author of the popular HCI reference, "The Design of Everyday Things"[23]. Norman describes the model of users interacting with software by expressing three simple sub-models (see Figure 3.1). These models are: the designer's model, the user's mental model, and the system image.

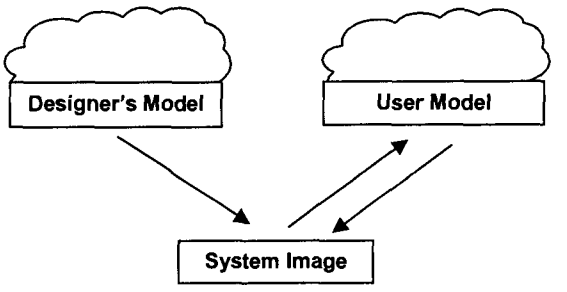

Figure 3.1 Norman's model of how users understand tools such as software [23].

In Norman's model, the Designer's Model represents how the software designer views the system as well as how the software designer *intends* the system to be viewed by the Users. Arrows from the Designer's Model to the System Image are intended to indicate that the Customizer creates the System Image. The absence of arrows between the Designer and the User indicates that the User does not interact with the Designer to learn the model of the system, but rather that the User interacts with the System Image. This indirect transference of the intended model of the system from the Designer to the User requires that the System Image be carefully designed if it is to be successful. Two arrows between the User's Mental Model and the System Image indicate that the user learns from the System Image while making adjustments to it through their interactions.

3.3.2 Adding the Customizer to the model

This model is a good start as it captures the essence of creating a software system by one person that is to be passed on and learned or used by another. However, our understanding of customization requires some enhancements. First, in our idealized description of the participants involved in software customizations, we include another category or role: the Customizer. The Customizer takes the system originally developed by the Designer, and customizes it for the Users.

Another enhancement we have made to Norman's model is a legend identifying the meaning of each of the arrows. We now can see clearly that the Designer builds the System Image which is learned by the Customizer. Then the Customizer modifies (through source code or interaction styles) and presents a Refined System Image to the End User. Finally, the End User also performs some learning and modification tasks. The addition of the Customizer, legend and arrows is represented in Figure 3.2 shown below.

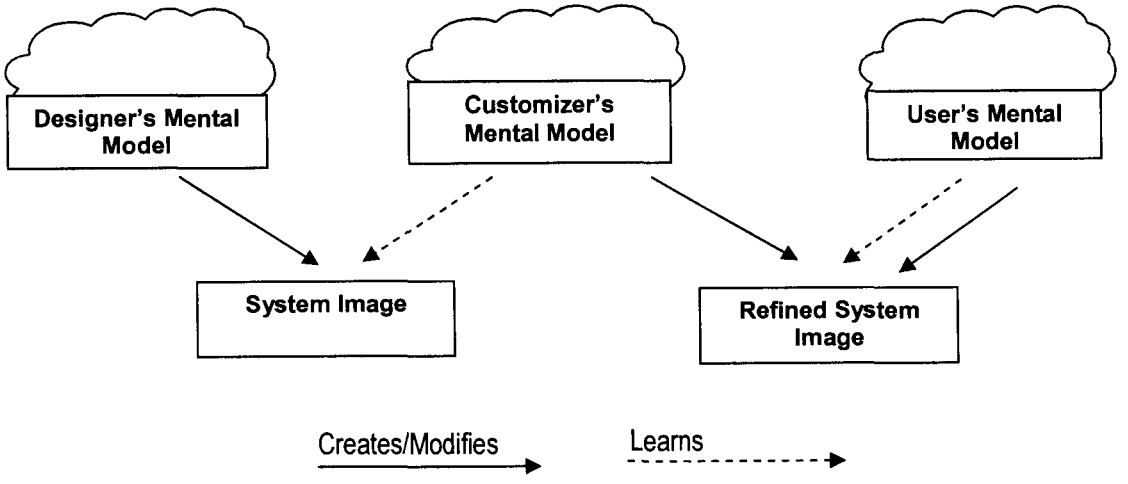

Figure 3.2 Adding the Customizer to the model

The model does not incorporate a third version of the System Image (to represent the individually customized system) as we believe that the modifications by the End User are not extensive enough to warrant it. This becomes clearer as we provide more details of our model.

3.3.3 The Mental Models in More Detail

In our review of program comprehension modelling, we discovered that it is important to determine some context-specific details of the mental model. We need to make known what the sub mental models are for each stakeholder. Since the context of this research is software customization, through either its source code or through a secondary interaction style, we suggest that these two ideas be incorporated as sub mental models. Source code and related artifacts would fall under the categorization of the *Implementation Model*, while the interaction-style type customizations can be separately understood under the label *Customizations Model.*

In addition, we are supporting customization with the purpose of changing how a program works to optimize it for the user and their domain. The program, in many respects, reflects the domain itself. Therefore, we should also incorporate into the mental models, sub models that reflect the stakeholders understanding of the domain. These types of model are referred to as *Domain Models*. The details of the mental models can be seen in Figure 3.3. Note, the User's Mental Model does not incorporate the Implementation Model as the User typically does not know how to program and is not given access to the source code.

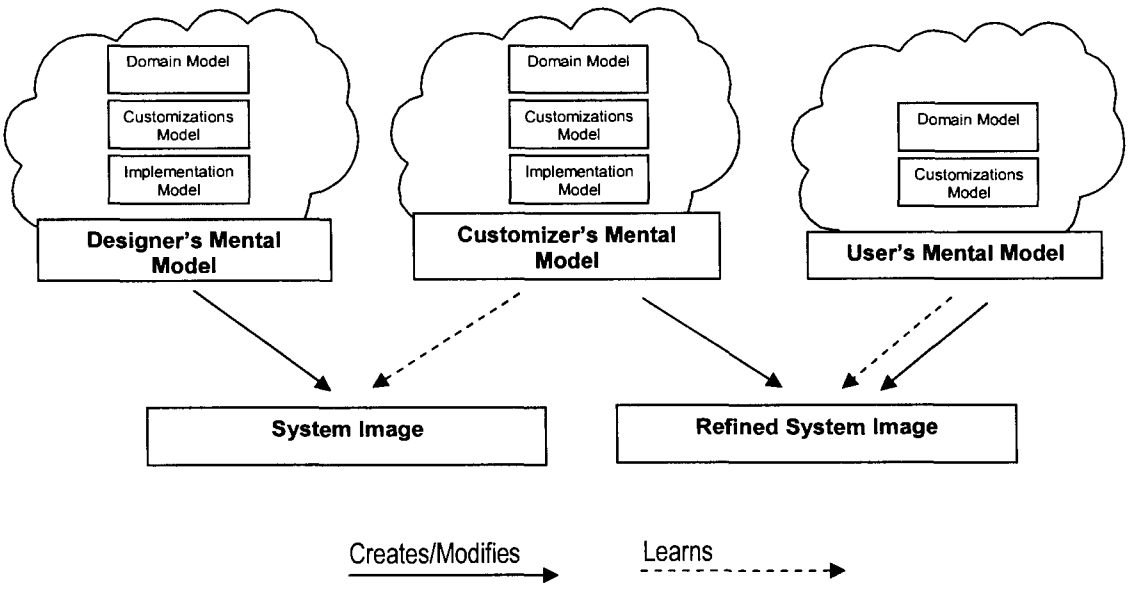

Figure 3.3 Adding the concepts of Domain, Customization, and Implementation to the model

3.3.4 Expanding the System Image

A limitation of the model so far is that it is unclear how the System Image supports building the mental models we identified so far. We further decompose the System Image to include other elements such as the Documentation and the Source Code (see Figure 3.4). The reference to Image now refers only to the application and how it behaves. It is now clearer how the mental models can be acquired from the System.

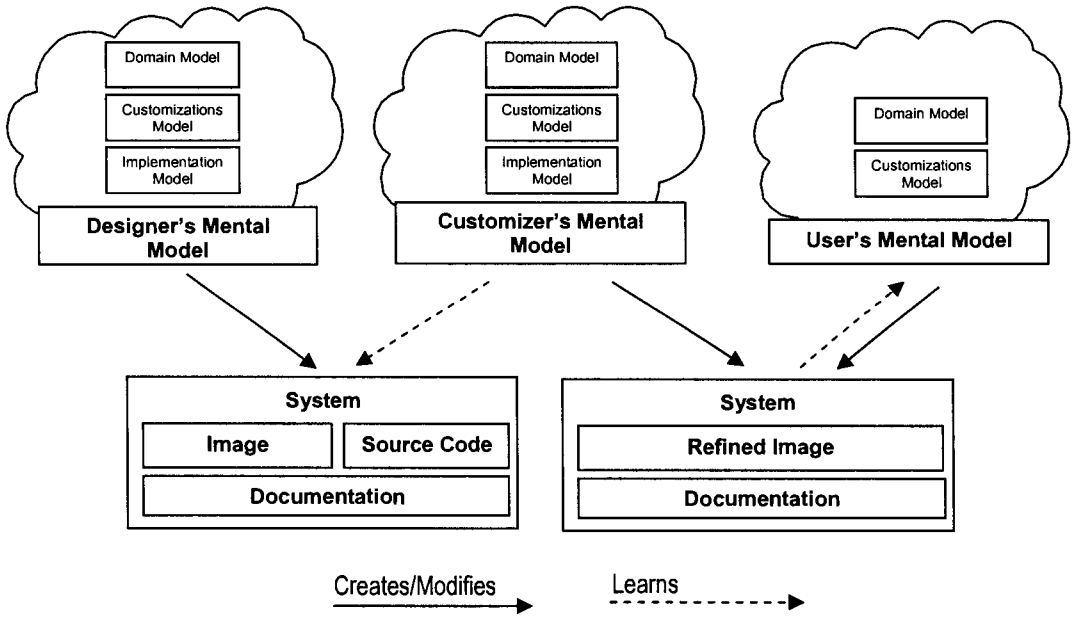

Figure 3.4 Adding the Source Code and Documentation artifacts to accompany the System Image.

3.3.5 Incorporating the concept of Knowledge

Our model now contains the core mental models we associate with software customization. We will now add to the model the next common element of program comprehension models: knowledge. Knowledge, as described in Section 3.2.1, differs from a Mental Model. Knowledge describes all the information known within a topic, while a Mental Model describes the understanding of a particular instance of a situation/problem (or class of situations/problems). Consequently, we distinguish between the mental models and knowledge (see Fig. 3.5).

A certain amount of Domain Knowledge should be had by each of the Designer, the Customizer and the User. Domain Knowledge is used to determine the features of the software, how they should be presented, and how they should be explained in the documentation. The Designer and the Customizer also have Programming Knowledge

which facilitates communication of the Implementation Model through the Source Code. Lastly, the Customizer tends to have a specialized knowledge of the User (gathered through experience or specific requirements gathering procedures) which empowers them to make the choices when they customize the software for the User. These additions are reflected in the updated model in Figure 3.5.

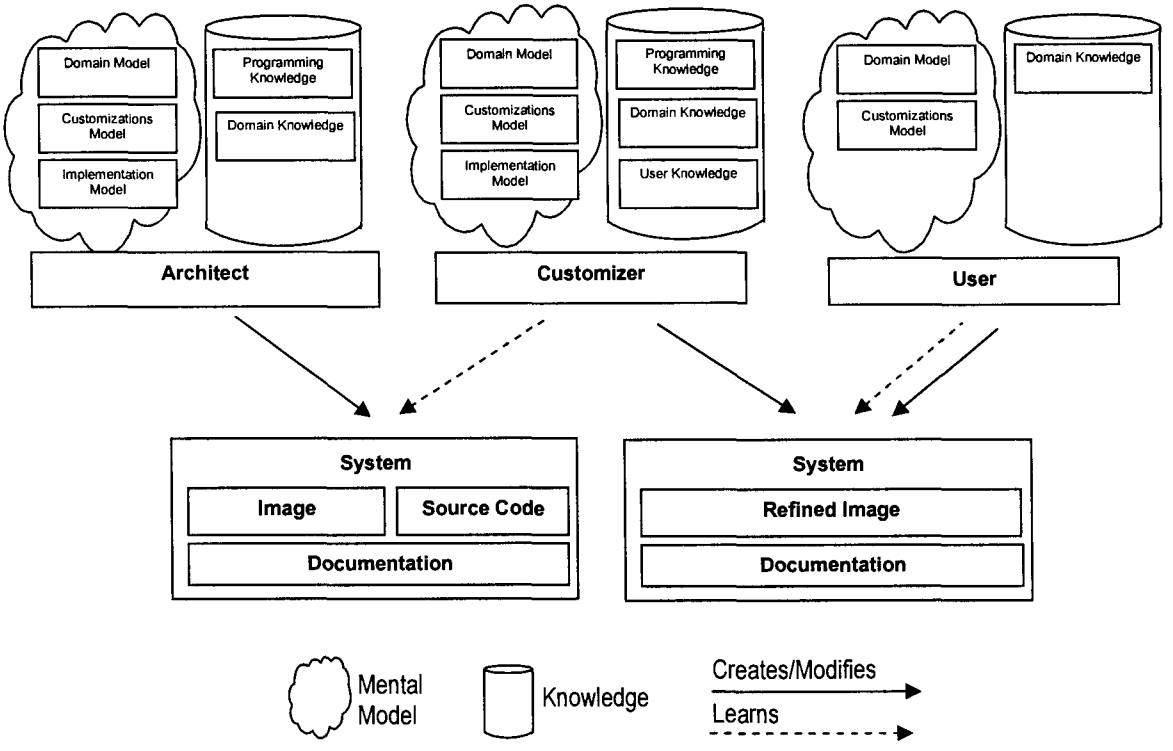

Figure 3.5 Adding the concept of knowledge

3.3.6 Identifying Expertise

The third important item in program comprehension models is the expertise had by individuals. Expertise reflects where one has special skills and advanced abilities in a particular domain as described in Section 3.2.3. Here we provide some general observations. A Designer's expertise is in programming (designing and building software). Not all Designer's become true experts in a domain as many move from project to project. Users, on the other hand, typically have extensive exposure to their

domain, providing them with expert abilities in understanding and tackling problems from the domain. The Customizer is typically also an expert in the application's domain. Many Customizers gain this expertise through time spent as a User (Guru) or through extensive study.

The Customizer should have much expertise about Users and their needs (from being a User or through study). In general, however, the Customizer is not always an expert in all aspects of programming. Their programming skills tend to be only strong in terms of the domain they specialize in. The model presented in Figure 3.6 displays the incorporation of expertise by shading in those aspects of knowledge we have identified here as being at the expert level.

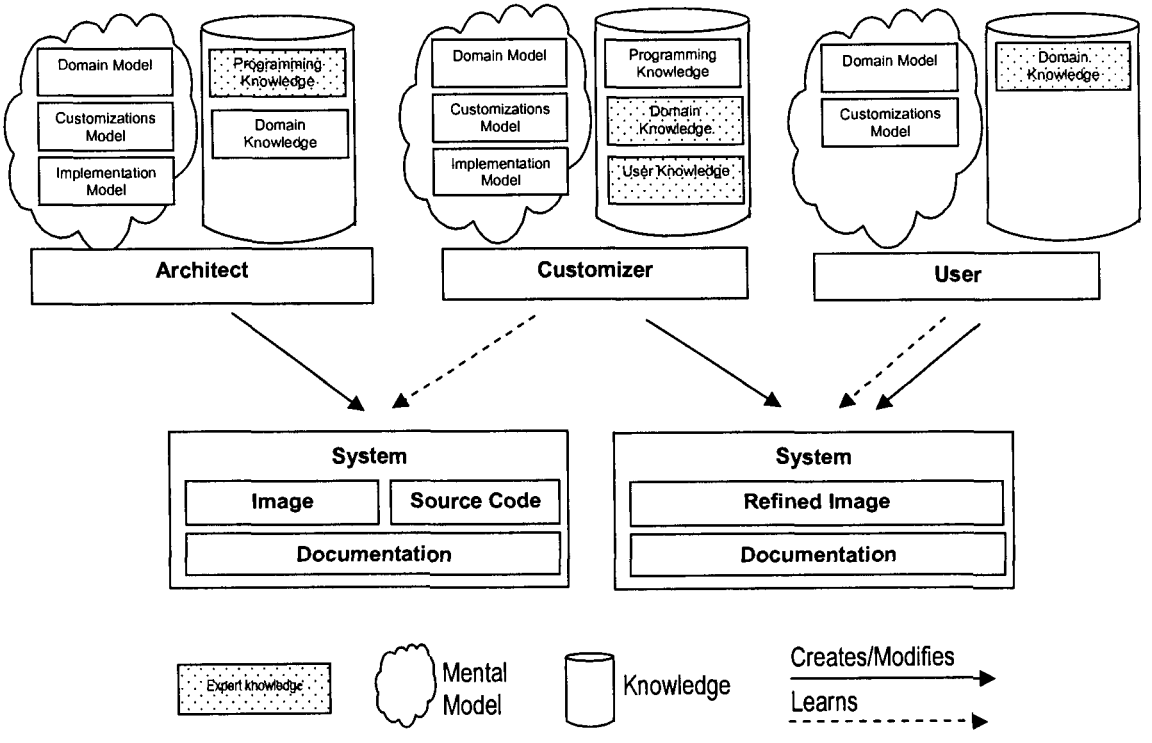

Figure 3.6 Highlighting the knowledge that is at an expert level

3.4 What we can learn from the model

The goal of establishing this model was to clarify the customization process and to produce a set of guidelines for teams developing tailorable software. The guidelines we are able to interpret from this model include determining who can customize what, how customization should be enabled, and where support should be afforded. In the following subsections we provide details of our analyses and views.

3.4.1 Who Can Customize What

In terms of providing customization facilities for the User, we can see from the model that the Designer should not be responsible for these customizations directly. Instead, the Designer should *enable* customizations through the application's architecture and design. These decisions regarding customization support in the implementation are very important and significant; they constrain the Customizer (and the User) in what can be customized in the application and how.

Typically, the Customizer is the most qualified to tailor the application. First of all, the Customizer may need to tailor the application's features to the specific needs of the Users. Secondly, the Customizer may develop interaction styles that will enable the User to tailor the application. It is important to note that the facilities provided to the User should only include the types of customization that can be achieved with the User's skills and knowledge.

In Chapter 2, we described the concept of customization by organizing different types of customization into three categories. In Table 3.1, we present a general summary of what

can be customized with interaction styles appropriate for Users. Here we can see that both types of presentation customization can typically be configured by Users. Customizing the content of data files, performing feature selection, and specifying options for those features are also customizations within the skills of Users. The remaining types of customization should only be done by a Customizer.

Type of Customization	What is Customized	Supportable with an Interaction Style Appropriate for Users
Presentation	Information architecture	Yes
	Graphical Design	Yes
Data	Format	No
	Content	Yes
Control	Feature Selection	Yes
	Option Specification	Yes
	Feature Addition	No
	Feature Enhancement	No
	Feature Constraint	No
	Feature Coordination	No

Table 3.1 Some customization types are appropriate for Users

3.4.2 Where customization support should be provided

Having a general concept of *who* can customize *what* leads naturally to determining *how* these customizations should be supported. From our model, it becomes apparent that the Customizer has the responsibility of customizing the majority of the application for the User. For the Customizer, this can be a difficult task. It requires the Customizer to perform an analysis of how the application meets the User's needs, followed by determining where customization should be required. Next, the Customizer must learn enough of the implementation to enable them to program the chosen customizations. Considering the magnitude of this effort it is important that the Customizer be properly supported in his/her efforts. Therefore, the Designer should make the effort to make the system comprehensible, maintainable, and extensible. Since the Customizer is often not

an expert programmer, customizations to the system should not necessarily require expertise in programming.

In this chapter we indicated that experts have specialized schemas that enable them to produce plans to quickly and effectively solve problems. If the Designer cannot rely on the Customizer to have these specialized schemas, the Designer should try to supplement them. In the coming chapters we propose how such schemas can be supplemented.

In Chapter 4 we take a look at how the source code of a system can be organised using components, design patterns and frameworks. These architectural elements are abstractions that can be used to quickly form the needed schema of the application. In Chapter 5, we will present work on how a system can be customized without source code, and instead use an interaction style that allows a Customizer to leverage their expert Domain Knowledge.

Chapter 4

Customization with Source Code

From the analysis of the customization model in Chapter 3 we saw that the Designer must provide to the Customizer a software solution that is easily customizable. To achieve such a goal, we consider object oriented design, design patterns, components and frameworks. We discuss how these technologies could enable the Designer to create a system that the Customizer can understand how to customize.

4.1 Object Oriented Design

Designing and building a customizable software system is a complex task. Object oriented technology has been demonstrated to be the most preferable (if not the most capable) choice when building complex software [29]. The features of OO languages (such as abstraction, encapsulation, information hiding, inheritance, and polymorphism)

enable a complex problem to be expressed in such a way that the design can be effectively communicated and understood [30]. In essence, object oriented design, when effectively applied, forms a model of the problem enabling a high level of understanding. According to Korsen and McGregor:

> "The design philosophy of the object-oriented paradigm takes a modeling point of view. This allows the Customizer to work with one approach which begins in the problem domain and transitions naturally into the solution domain. By building a model of the problem into the application system, the resulting design is more responsive to changes in knowledge about the problem situation. The modularity of these designs and information-hiding capabilities of most object-oriented languages contribute to the technique's responsiveness to modifications" [30, p. 60].

OO design is a great start for managing complex software design. Designers can construct the Implementation Model with classes based on concepts from the Domain Model, making it easy for the Customizer to learn. However, OO design does not guarantee that a system will be easy to customize. Additional architectural ideas can be applied to the design of a software system to make it more manageable for the Customizer to perform customizations tasks. Three such complimentary software architecture elements are design patterns, components, and frameworks. As stated by Larsen, "Patterns and frameworks provide the medium for reusing proven solutions and elevating the abstractions where engineers communicate and produce solutions. Components provide the packaging for the patterns and frameworks" [31].

In the following three sections, we take a closer look at design patterns, components, and frameworks with a focus on how they can help make software more customizable.

4.2 Design Patterns

In "A Pattern Language", Christopher Alexander says: "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem in such a way that you can use this solution a million times over, without ever doing it the same way twice"[32]. Although Alexander was speaking about the patterns found in architecture his description is quite applicable to software design [33]. There are three reasons which make design patterns appropriate for customizable software: design patterns are well documented; design patterns provide proven solutions to making aspects of the design customizable; and lastly, design patterns can make it easier to change/add to the code to realize customizations.

4.2.1 Well Documented

Documentation is an undeniable resource in software systems where the software design is to be comprehended by programmers who did not participate in its original design or implementation. A core element of every design pattern is its documentation. This enables a Designer to use design patterns in their systems and let the publicly available documentation explain their complexity (and purpose, and how to use them, etc) to any future Customizers. Design pattern documentation generally includes the following details [33]:

- **Pattern Name and Classification:** Establishes a vocabulary for design
- **Intent:** Indicates what problem the pattern addresses
- **Also Known As:** Other names pattern is called
- **Motivation:** A scenario demonstrating how this pattern helped
- **Applicability:** General circumstances indicating when this pattern will apply.
- **Structure:** Graphical representation of the pattern
- **Participants:** Descriptions of the classes/objects and interfaces involved
- **Collaborations:** How the participants work together in the pattern
- **Consequences:** Positive and negative tradeoffs.
- **Implementation:** Language specific issues, and other hints and pitfalls.
- **Sample Code:** Code fragments showing skeleton of the design pattern.
- **Known Uses:** Real systems that use the pattern
- **Related Patterns:** Similar or complementary patterns

When we look at how documentation fits into our Software Customization Model, design pattern documentation plays a significant role. For the Customizer, the documentation can lead them to understand how the Implementation Model that the Designer built works. Secondly, this documentation may help the Customizer to add to his/her general knowledge about programming which will also help them better understand the Implementation Model (and future models).

Designers can seek out design patterns when they feel that an element of their design could be improved. It is likely that previous Designers have solved or designed a better solution for any given problem (which hopefully they have documented). In the next section, we describe how Design Patterns can be applied to construct customizable features in a software system.

4.2.2 Customizable Solutions

Not only are design patterns useful for helping programmers understand the system, they are in fact blueprints for designing customizable software features. A Designer with a particular need to make an element flexible in the design can look for a suitable method to support the required customization in the design pattern literature. In general, all patterns can be placed into one of three categories: creational, structural, and behavioural. Creational patterns offer alternative methods to object creation. Structural patterns (such as the Decorator Pattern) deal with different methods for the composition of classes or objects. Behavioural patterns illustrate different ways in which classes and objects can interact and share responsibility [33]. In Table 4.1, some example design patterns from these categories are listed with a description of how they can support customization.

Purpose	Design Pattern	Aspect(s) That Can Vary
Creational	Abstract Factory	Families of product objects
	Builder	How a composite object gets created
	Factory Method	Subclass of object that is instantiated

	Prototype	Class of object that is instantiated
	Singleton	The sole instance of a class
Structural	Adapter	Interface to an object
	Bridge	Implementation of an object
	Composite	Structure and composition of an object
	Decorator	Responsibilities of an object without subclassing
	Façade	Interface to a subsystem
	Flyweight	Storage costs of objects
Behavioural	Proxy	How an object is accessed; its location
	Chain of Responsibility	Object that can fulfill a request
	Command	When and how a request is fulfilled
	Interpreter	Grammar and interpretation of a language
	Iterator	How an aggregate's elements are accessed, traversed
	Mediator	How and which objects interact with each other
	Memento	What private information is stored outside an object and when
	Observer	Number of objects that depend on another object; how the dependent objects stay up to date
	State	States of an object
	Strategy	An algorithm
	Template Method	Steps of an algorithm
	Visitor	Operations that can be applied to object(s) without changing their class(es)

Table 4.1 Design Patterns with their respective varying aspects [33]

4.2.3 Designing for Customization

The third important characteristic of design patterns is their inherent tailorability. Many design patterns will permit a Customizer to perform customizations of the system through changing the source code. For an example, let us take a look at the Strategy pattern, which is shown in Figure 4.1. In this simple pattern, different ways (algorithms) of performing the same task are changed from being the methods of their parent class and placed into separate classes that implement a common interface. That way, as the context of the situation changes, the class can simply point to the appropriate strategy class to perform the task.

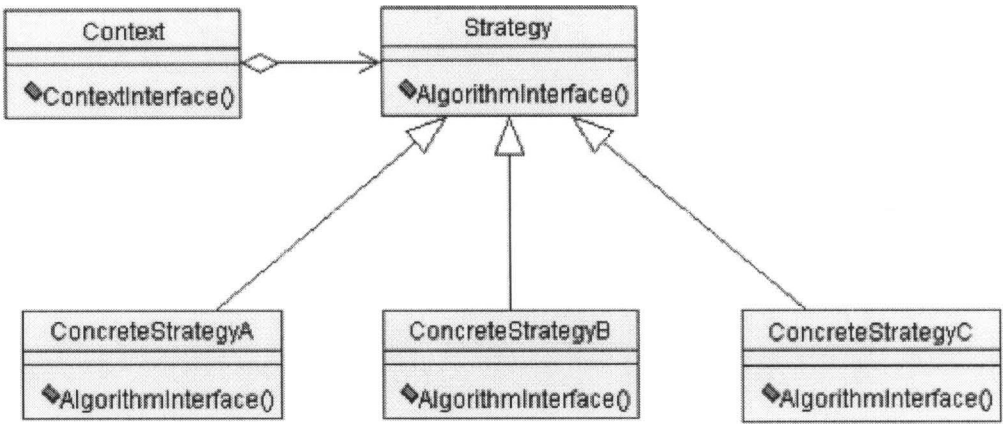

Figure 4.1 The Strategy Design Pattern

For example, a business application may use the Internet to transmit data to and from remotely located offices. For the developer, there are several different protocols to choose from to manage such transfers (HTTP, FTP, SMTP) and no single solution will be suitable for every office. Each office may have different infrastructures, security restrictions, and sets of standards that determine which protocol(s) can be used and which will likely change over time. The strategy pattern in this case would be quite useful in encapsulating each of the different methods of transfer that may be needed. The software system would meet the requirements of each remotely located office, as well as have in place an established mechanism for adding new methods of transferring files as the technology and businesses evolve.

4.2.4 Design Pattern Summary

Design patterns can be very useful tools for the Designer to make a system customizable yet understandable by the Customizer. The wealth of documentation, the applicability of many design patterns to software customization and the ease in which the design pattern can be tailored make design patterns a tool not to be overlooked. However, caution must

be taken to ensure that design patterns are not misused or overused. In general, if the abstractions provided by a design become too difficult to understand, their benefits are lost.

4.3 Components

Design patterns enable us to reuse the expertise of software Designers. To package implementations of design patterns (and other designs), we look to components. Components, like design patterns and object oriented design, are a widely applicable methodology for structuring the implementation of a software system.

4.3.1 Defining Components

The definition of a component varies depending on one's background and current point of view. Four definitions are presented here to provide a broad perspective on what a component is. David Sprott's definition is: "the component is a separate, encapsulated entity and by virtue of its separation is easier to manage, upgrade, [and] collaborate with" [4, p 64]. Grady Booch describes a component as "a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces" [34]. Margaretha Price *et al.* have a simpler definition: "A component is a set of classes that are expected to be reused as a group" [35, p 42]. Finally, D'Souza and Wills describe a component as a "coherent package of software artifacts that can be independently developed and delivered as a unit and that can be composed, unchanged, with other components to build something larger" [36, p. 386].

Components are a fundamental aspect of many programming schools. Most notable of the component implementations are Microsoft's COM [37], Sun's Java Beans [38], and CORBA [39]. Each of these implementations is different in terms of rules, usage and effect. However, they each share the following attributes [36]:

- A list of provided interfaces.
- A list of required interfaces.
- A specification of what the component does.
- The executable code.
- The validation code: code that is used to help decide whether a proposed connection between components is OK.
- Design documentation.

The composition of a component is important but so too are the resulting characteristics and benefits.

4.3.2 Characteristics and Benefits of a Component Architecture

The most important characteristic of a component is that it is a black box design. Customizers should be able to make use of a component without having to understand how it works [40]. "The only thing you need to know about the component is the published interface that specifies the contractual agreement necessary to use the services provided" [41, p. 64]. In fact, most of the benefits associated with components are due to the fact that their implementation details are hidden behind their interfaces (see Figure 4.2).

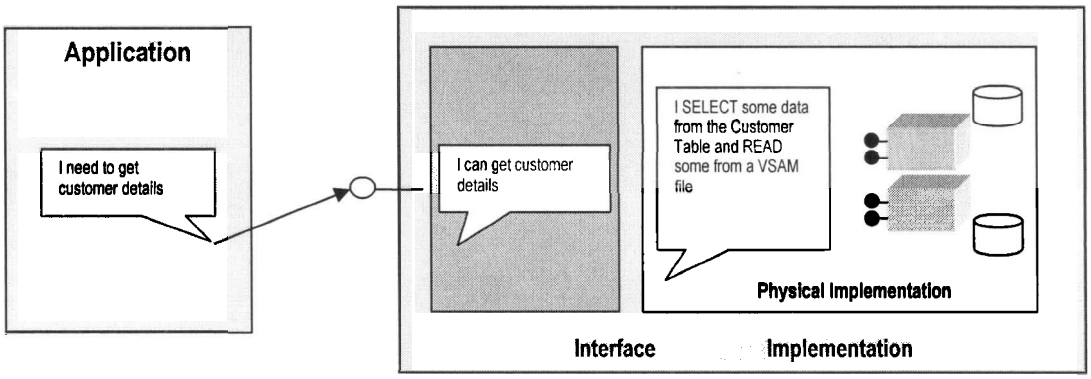

Figure 4.2 The component's interface isolates the details of the implementation from the application [41].

There are six attributes of component architecture that make it an ideal choice for any serious software system: applicability, simplified group development, easier integration, adaptability, upgradeability and scalability.

Applicability

Componentized application architectures present a high degree of applicability by offering the most choices in terms of acquiring the needed functionality in a software system. "The binary choice between 'build or buy' becomes 'build and buy and reuse'" [41, p. 67]. Applications can be assembled with components from three different groups: 1) core application providers (the Designers), 2) third party add-on component developers, and 3) in-house component developers (the Customizers). Balancing costs and time restrictions with the desired functionality in a software system becomes more manageable and achievable with these choices.

Simplified Group Development

Components also enable different groups of people to separately participate in the development of a software system. As long as each component's interface and behaviour has been agreed upon, parallel development of the components of the software can occur. This would enable teams of Customizers to work together to more quickly customize an application.

Easier Integration

Component technology also offers solutions in terms of integration. There are established methods to take legacy applications and wrap them to look and act like other components within the system. This offers the Customizer the ability to customize the software so that it works with other user applications. Components developed in Java for instance, can be wrapped and used as an Active X control in a Microsoft Visual Basic application [42].

Adaptability

With a componentized design provided by the Designer for the Customizer, the Customizer may determine that one or more of the components does not exactly reflect the needs of the User. To deal with this situation, the Customizer can apply one of two options. First, the components in question can be adapted. Components can be designed (by the Designer) with extra interfaces that enable them to accept plugins that specialize their behaviour. For example, a communication component may use plugins to specify

the encryption scheme to be used. If a component does not have this type of facility, the Customizer can adapt the component's functionality by changing its source code.

Upgradeability

The second option available to the Customizer to manage a component that does not precisely perform as needed is to perform a component upgrade and replace the existing component with a new one. "There should no longer be a need to upgrade entire 'systems'; instead, components get replaced or added as needed" [36, p 395]. As mentioned previously, the new component can be developed in-house by the Customizer, by the Designer, or by a third party.

Scalability

Lastly, by being interface-centric, components enable scalable and extensible architectures. "By letting each component have multiple interfaces, we reduce the dependency of any one component on irrelevant features of another component that it connects with" [36, p. 397]. These characteristics enable the Customizer to deliver a system with the required features and performance.

4.3.3 Component Based Design

Additional benefits from using components are apparent when one looks at how components can be combined through component-based design. Component-based design can be described as "the mind-set, science, and art of building with and for components and ensuring that the result of plugging components together has the expected effect" [36, p. 410]. To support component based design, the ports and

connectors that can be used are defined and documented. Ports are agreed upon plugs and sockets through which a component exposes itself for connection as well as those points where it accesses the services of others. Connectors connect two or more ports together to compose components into larger components or applications [36].

A set of highly compatible components designed according to a unified architecture and set of standards and services is called a component kit [36]. Such kits can be used to quickly combine and configure applications. A Customizer can more easily swap in and out kit components to customize the design for the User.

4.3.4 Summary of How Components Support Customization

Support for customizable software is apparent in almost every aspect of component technology. As with design patterns, components increase the comprehensibility of the system through their abstract nature. The black box concept allows the Customizer to make use of components without having to understand their implementation in most cases. In addition, applications can support customizations either through the addition, replacement, or configuration of the components in the system.

The flexibility available in connecting components also supports customization. A User's requirements may require a distributed architecture or integration with existing applications which may reside on servers or other computers. By taking advantage of the various methods of how components can be connected, these challenges can be efficiently met by the Customizer.

Lastly, component kits can offer the Customizers substantial freedom by supporting the designing, building and customization of solutions with a minimal amount of programming. New development environments such as Macromedia Flash MX [43] and IBM's Eclipse [3] support and encourage developers to make use of existing components (Components for Flash MX and Plugins for Eclipse) as well as develop and share new ones. New applications and/or tailored applications can be produced quickly with reduced effort.

4.4 Frameworks

Tremendous effort goes into designing software. When creating software from scratch, the following steps are typical before any programming can even begin [36]:

1) Generating the business model
2) Generating the requirements specification
3) Refining the design down into components and high level functionality
4) Applying object design to specify the inner workings of each component
5) Choosing and designing a component architecture

The result of this process, and others like it, is a design for building the software which is a vital part of a successful implementation. Unfortunately, in many cases the cost, time, and skills required to produce such a design often cause companies to skip or speed through phases of the process. As described in Section 3, the company could offset their development costs by purchasing components from other vendors. This can, however, still place the company in a situation where they do not have the time or skills to bridge the functionality from the various components into a comprehensive system. Frameworks are an alternative to component based design that offers a more holistic solution, while still providing the needed flexibility.

4.4.1 Frameworks Defined

Like design patterns, frameworks represent the refined results of the efforts of previous Designers. And like components, frameworks can be defined differently depending on one's context and thus, several definitions are presented here. The first definition we mention is by Grady Booch who describes frameworks as an "...architectural pattern that provides an extensible template for applications within a domain" [34]. The second is the longer definition of Ralph Johnson.

> "A framework describes the architecture of an object-oriented system; the kinds of objects in it and how they interact. It describes how a particular kind of program, such as a user interface or network communication software, is decomposed into objects. It is represented by a set of classes (usually abstract), one for each kind of object, but the interaction patterns between objects are just as much a part of the framework as the classes" [44, p 12].

The last definition we provide comes from D.F. D'Souza and A.C. Wills which describe a framework as follows, "A template package; a package that is designed to be imported with substitutions. It 'unfolds' to provide a version of its contents that is specialized based on the substitutions made" [36, p. 340].

Frameworks extend their model from design patterns as they are general solutions to a problem. Frameworks however tend to be on a larger scale. They are meant to be templates for entire applications and can consist of several components. Another key difference between a framework and a design pattern are the extension points. As the above definitions indicate, frameworks are designed with planned holes that the software Customizer is meant to fill in. Extension points represent planned opportunities for the Customizer to fill in the specific settings or functionality to complete the design (see

Figure 4.3). Lastly, frameworks, unlike Design Patterns, are an implementation that can be instantly leveraged and extended by the Customizers.

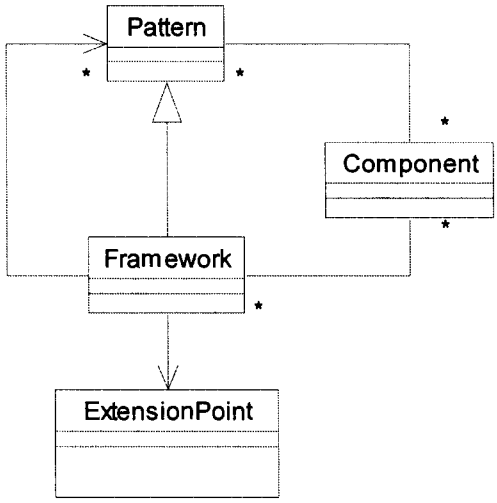

Figure 4.3 A class diagram showing the relationship between patterns, components, and frameworks as shown in [45].

An excellent example of a software framework is IBM's Eclipse. Eclipse is a platform for building integrated development environments that can be used to develop a diverse set of applications [3]. Its architecture, implemented with Java, strongly follows the paradigm of a software framework. Eclipse is composed of a core platform with extension points that can be extended by "plugins" (a type of component). A plugin typically includes compiled Java code, the code's required libraries and resources (such as images or data files), and a "manifest" file.

The manifest is an XML file that describes the plugin's interconnections with other plugins. This includes declaring which extension points the plugin is plugging into and which extension points are being made available by the plugin for other plugins to

extend. Eclipse also has effective support for the development of plugins with its Java Development Tools and the Plugin Development Environment; both of which are plugins themselves. An overview of the Eclipse framework is provided in Figure 4.4.

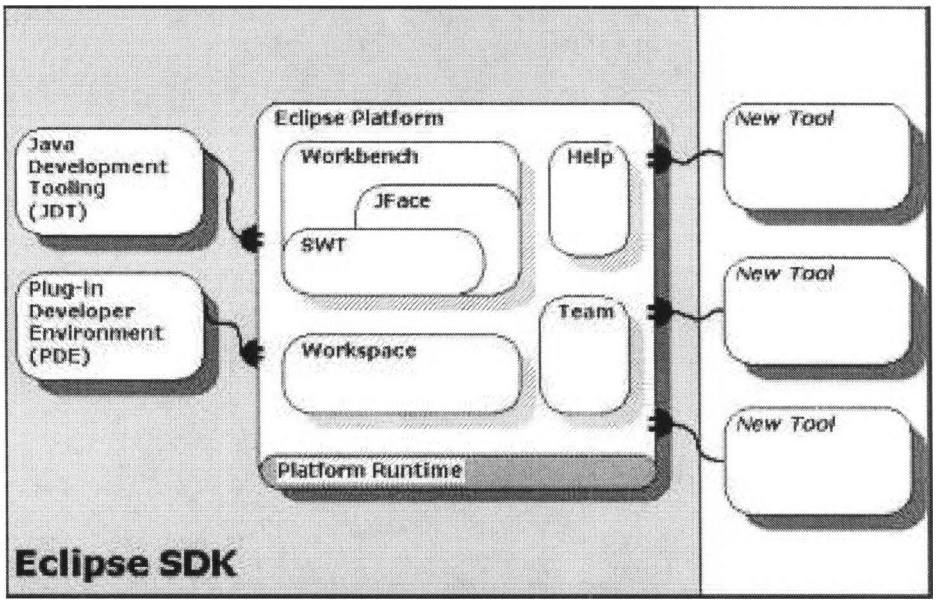

Figure 4.4 An overview of the Eclipse Software Framework

To further explain how frameworks can be used, it is necessary to know more details about the composition of a framework. Grant Larsen in [45] proposes that a framework should contain the following artifacts:

a) Architecture document
b) Components implementing the framework
c) Defined extension points
d) Framework characteristics
e) Framework code
f) Framework quality measurements
g) Requirements document and database
h) Models
i) Snapshots
j) Test cases
k) Test data and test drivers

The intention of providing these artifacts is to transfer the Designer's knowledge of the design into the minds of future application developers. A Designer can use this framework to build an application or part of an application. The Customizer can then make use of the accompanying artifacts to aid them in understanding how to customize the system for the User.

4.4.2 Using Frameworks

When using a framework, the general usage method will be determined by the style of the framework – white box or black box or somewhere in between. The difference between the two extremes can be identified through their extensibility mechanisms.

White box frameworks require the Designer or Customizer to use OO language features such as inheritance and polymorphism. The functionality and design is to be reused and extended by inheriting code from the base classes of the framework and/or by overriding the hook-methods. Requiring the use of inheritance and overriding methods leads to two disadvantages. Firstly, using inheritance requires a deep understanding of the frameworks structure and individual classes. Secondly, the resulting software will be tightly coupled to the hierarchies of the framework [46].

Black box frameworks are quite different. Black box frameworks support extensibility through the use of components that can be plugged in and removed from a design via the frameworks defined method for component composition. Users of black box frameworks should not have to understand the details of the implementation since components hide their implementation details behind interfaces. In addition, the resulting software tends to

be less coupled to the framework. The only major drawback of black box frameworks is that they are more difficult to design and build making them more unusual [46].

4.4.3 Frameworks and Support for Customization

Frameworks in general are not easy to develop, and are constructed through an iterative process of improving and generalizing a software system [47]. Designers build a framework for the sole purpose of supporting customization by future Designers and Customizers.

As with design patterns, the result of using a framework will be a modular, reusable, and extensible design that has proven itself in previous projects [46]. Frameworks are desirable as they combine object oriented technology with patterns and components and thus can inherit all the benefits associated with them (described in previous sections). In addition, since frameworks have identified extensibility points, the Designer and the Customizer are guided in their customizations and the nature of the framework will support them. These customizations will most likely take the shape of adding, replacing, or reconfiguring components or classes at these extension points.

4.5 How Software Technology Can Enable Customization

In Chapter 3, we created a model of software customization that guided us by identifying those aspects of an application that need to be customized by the Customizer. In this chapter we explored some architectural design elements that can be applied to achieve the flexibility needed in a software system. Here we redesign Table 3.1 as Table 4.2 and add examples of how the elements can be customized through the use of design patterns and components. For example, Feature Selection can be made possible by using the Strategy

Pattern (useful if the feature is an algorithm for doing an operation) or by using Component composition techniques (beneficial for larger scale features such as spell check).

What is Customized	Supportable with an Interaction Style (typically)	How: Frameworks		
		How: Design Patterns	How: Components	How: Extension Points (Eclipse)
Presentation: Information architecture	Yes	Bridge Pattern	Decouples GUI from implementation	Yes
Presentation: Graphical Design	Yes	Composite Pattern, Decorator Pattern, Command Pattern	Component Toolkit to Guide GUI building	Yes
Data: Format	No	Builder Pattern, Interpreter Pattern	Adapt/Upgrade Data Format Component	Yes
Data: Content	Yes	-	-	Yes
Control: Feature Selection	Yes	Strategy Pattern	Component Composition	No
Control: Option Specification	Yes	Template Method Pattern, State Pattern	Component Composition and Adaptation	Yes
Control: Feature Addition	No	Abstract Factory Pattern	Component Composition	Yes
Control: Feature Enhancement	No	Strategy Pattern, Decorator Pattern	Component Adaptation	Yes
Control: Feature Constraint	No	Decorator Pattern	Component Adaptation	No
Control: Feature Coordination	No	Mediator Pattern, Adapter Pattern	Component Composition	Yes

Table 4.2 Sample applications of Components and Design Patterns to 'what' can be customized.

This table also summarizes how Frameworks can be applied to software customization as Frameworks are composed of Design Patterns and Components. In addition, Frameworks also have extension points to enable customizations and can be used to achieve most types of customization. For example, in our experiences customizing the Eclipse Framework, extension points enabled us to customize the environment except

when it came to feature selection and feature constraint. Changing how existing features worked usually required component adaptation or component replacement. For full details of an evaluation of Eclipse's customization framework, see [48].

4.6 Summary

Designers who consider 'how' they are going to build a customizable software system should be encouraged to take advantage of the benefits offered by object oriented design, design patterns, components, and frameworks. By doing so, some of the difficulties associated with making elements of the design customizable become simpler as the nature of these design and implementation methodologies inherently, or deliberately, support customization. In addition, the Customizer will be able to take advantage of the higher level abstractions that are produced and more easily map their knowledge of the Domain to know what and how the software system can be customized. Incorporation of these ideas into our model is shown Figure 4.5.

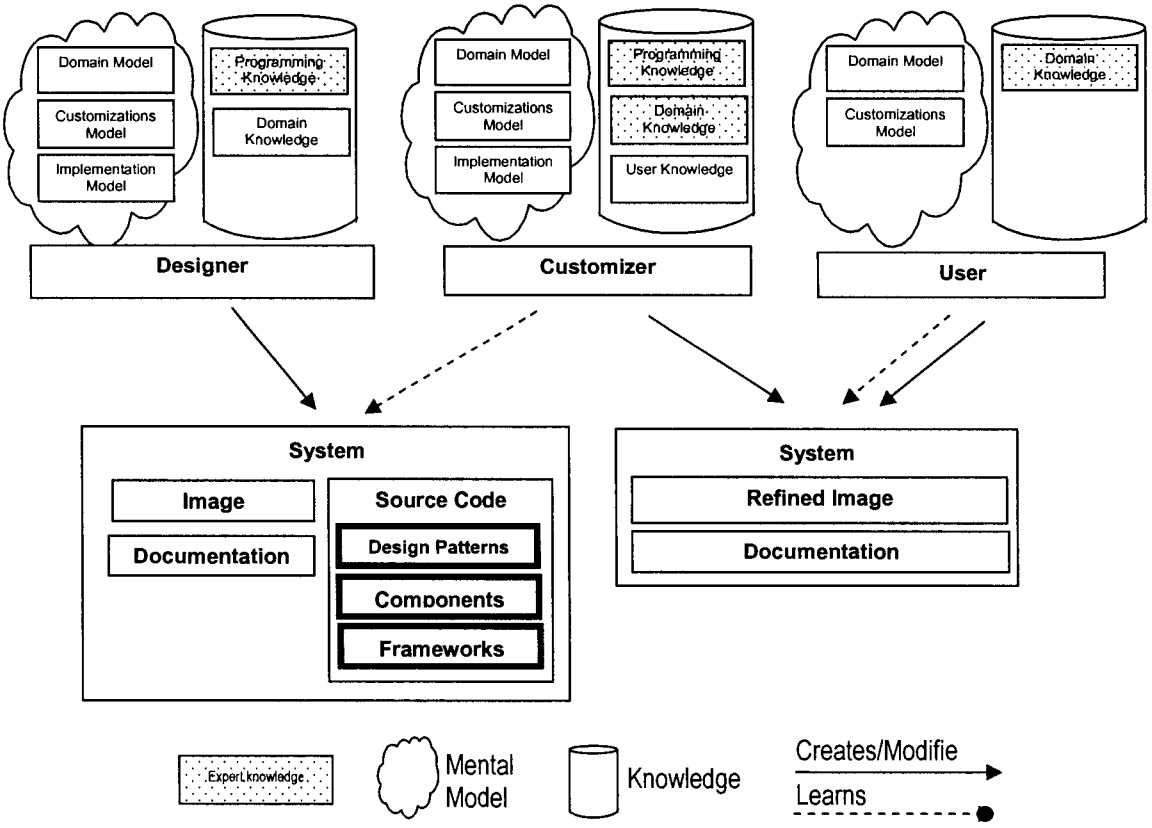

Figure 4.5 Design Patterns, Components and Frameworks are used by the Designer to enable the Customizer to customize the software.

In the next chapter we will take a look at ways for the system to be customized by methods that do not involve direct manipulation of the source code.

Chapter 5

Customization without Source Code

Working with a software application via the source code can be a daunting and difficult task. This may be true despite the best efforts of a Designer to incorporate comprehensible architectural design elements into the application such as those presented in Chapter 4. Some designs are inherently difficult to implement and it may be more effective for a Designer to build a method of customization that relieves the Customizer from having to work with the source code directly. This indirect method of customization is often referred to as end-user programming. In this chapter, we explore research related to end-user programming and determine an appropriate interaction style for the Customizer to specify customizations for the Users.

5.1 Previous Work on End User-Programming/Customization

A relatively small portion of research has focused on understanding end user programming. Researchers that have done significant work are Wendy Mackay and Bonnie Nardi. These researchers have studied end-user programming by observing how users of various experience and expertise levels interact with the programmable features of their software.

Even though the work of Nardi and Mackay does not focus directly upon individuals who would be considered a Customizer, the results are still useful and relevant to this group. Some results help to reinforce the necessity of having a Customizer involved in the customization design process, while other results help to determine the style in which the customizations should take place. In the next two subsections, we summarize the work of Mackay and Nardi. Note that the reference of 'users' is meant to be understood as referring to the Customizers as well as to Users in this chapter. This section is followed by a summary and discussion to explain how this work applies to our customization model and framework.

5.1.1 Wendy Mackay's Research

Mackay's user study [12] was aimed at determining the barriers to, and triggers for, user customizations. She studied 51 users whose technical skill varied from novice to expert. The most common barriers to performing a customization by the participants were the lack of time and the complexity of the task. The most common triggers for customization reported by the users were tasks they had to perform repeatedly, their annoyance with slow operations, and the desire to retrofit new software so that it worked like previous versions.

Not only did Mackay identify and rank a list of barriers and triggers; she also provided some recommendations to software Designers when building customizable systems. The first recommendation was that the Designer should enable users to customize the behaviour of their software as opposed to only supporting customization in the form of feature selection. This reinforces the classification we did of software customization types that not only includes feature selection, but many others including feature enhancement and feature constraint (see Chapter 2).

Second, Mackay stated the following: "Software [Designers] should develop designs that allow users to learn about effective usage patterns and modify them as such" [11, p 159]. In other words, the application should lead the user to investigating customization options and performing them.

Mackay also suggests that, in order to support a culture of customization, the software be architected to facilitate easy sharing of customization settings and code. Not only that, but customizations should be simple to find and be human readable. Lastly, Mackay noticed that users seem to only customize their applications at certain points in the software's lifecycle. To correct this, Mackay suggests that managers should create opportunities for their end users to reflect upon their usage patterns, determine what customizations they need, and adapt their software accordingly.

5.1.2 Bonnie Nardi's Research

Bonnie Nardi also performed extensive user studies and the results of her work are published in *A Small Matter of Programming* [2]. The goal of Nardi's work was to determine why users found it so difficult to perform end-user programming. By studying the users of two applications considered successful in supporting end-user programming (spreadsheet applications and CAD applications), Nardi was able to determine some key success factors in this area. These factors are summarized in the following sub sections.

5.1.2.1 Appropriateness of Formal Languages

Nardi found that end-user programming is successful when the users can express themselves in a formal language that is closely tied to their domain. Nardi describes formal systems and why they are useful for end user programming as follows:

1) People are accustomed to learning formal systems.

   "In everyday life, people learn and use a wide variety of formal languages and systems. These systems are so pervasive that we scarcely stop to notice them: the alphabet, numerals, games, methods for scoring games, arithmetic, algebra, shorthand, the conventions used in sewing, knitting and crochet patterns, copyediting marks, calendars, maps, money, musical notation, street and highway traffic control signs, and even the juvenile secret language 'Pig Latin' " [2, p 20].

2) People are more likely to learn formal languages as opposed to conventional programming languages because they are interested in them.

   "... it is only when people have a *particular interest in something*...that they readily learn the formal languages and notations ... It is the infusion of *interest* that enables a housewife who failed high school math to take a sewing pattern, alter it (which involves going from 2-space to 3-space and understanding the conventions of the sewing pattern to make the correct adjustments) and create a perfect garment" [2, p 35].

3) Task specific languages (formal languages) are useful.

> "Task-specificity has two advantages: (1) it affords users ready understanding of what the primitives of the language do ... and (2) it eases application development because users can directly express domain semantics in the high-level operations of the language – there is no need to string together lower-level operations to get the desired behaviour" [2, p 39].

5.1.2.2 Other Interaction Styles and their Issues

To further support the superiority of formal languages, Nardi explains the problems associated with the other common interaction styles being applied to software customization. These problematic styles include visual languages, forms, programming by example modification, programming by example, and automatic programming by informal program specification. Her arguments are summarized below.

Visual languages have grammars which use icons and symbols as opposed to the textual words of traditional programming languages. Nardi agrees with the proponents of this research that graphical representations are powerful at the high level where abstractions are better understood with a picture but finds that the empirical work does not support the claims that visual languages are more natural, easier to understand, or alleviate the need to learn syntax. Nardi also finds that *visual languages* are not compact enough to enable the user to view enough of the code on a single screen [2].

The next interaction style, *form based programming*, has a different set of issues. *Form based programming* requires users to enter values and select options in a series of forms or screens. The primary drawback of this approach is the narrow breadth of tasks that can be supported. Nardi does indicate that such a style has its merits but only in

situations where the customization task can be broken down into a set of variables that are known ahead of time [2].

Programming by example modification is a style where users are expected to modify existing segments of code. The heart of the problem with this style is the lack of code to present as examples. This style is inappropriate unless the exact sets of customizations that will occur are known and there exists the resources to provide working sample code [2].

Similar in name, but quite different in its approach, is the interaction style called *programming by example.* In this style users directly manipulate an interface in a manner that indicates what they wish to do. Then, the system interprets the action and produces the corresponding code or customization settings. This desirable, and seemingly magical approach to programming is hampered by the fact that the method lacks ... " the ability to clearly express terminating conditions and branching" [2, p. 72] as well as the extreme difficulty in performing accurate pattern matching to determine the true intentions of the user.

Last on the list of problematic interaction styles is *programming by informal specification* (also known as application generators, as discussed in Chapter 2). These systems are essentially programs that write programs when given a set of customization inputs (acquired usually with a set of forms). The main issue with these systems is their lack of applicability: one cannot possibly predict all of the features and variations that will be

wanted by the users. If it is possible to conceive of all the needed features, the cost of building such a system usually makes it impractical [2].

5.1.2.3 Issues with Programming Languages

Nardi identifies aspects of programming languages that non-programmers (this refers to the Users and not the Customizers) tend to find difficult, and that special attention should be paid to these features when designing the language and its implementation. Variables, the first of these, are difficult to use for the non-programmer, because it can be difficult to understand that a word in the code can represent information which is not visible on the screen.

Certain control constructs are also identified as being potentially difficult. Consideration has to be given on how to simplify the presentation of conditional logic, loops, and dependencies. Nardi describes how these issues are dealt with in spreadsheet programming. Nardi found that conditional logic was well understood by her spreadsheet users since the ability to reference cells was easy and well understood. Iterative structures were realized by performing an operation on a series of cells (such as a sum). Instead of having the user to write a for loop structure, the structure was implied by the users summation formula. This suggests that control structures can sometimes be specified and represented in an alternative (and perhaps more appropriate) manner.

5.1.2.4 Customization Community

Nardi, like Mackay, also identified the importance in creating a community of users. Her work describes three kinds of users in such communities: novice end users, expert users and professional programmers. This classification was part of the inspiration for our

Customization framework that identifies the Designer, the Customizer, and the User. Nardi suggests that an application should allow for users of all levels of customization expertise. Such a hierarchy of customization complexity allows all persons involved to participate at their level of ability as well as allow room for skill improvement by the less experienced in programming.

5.2 Applying the End-User Programming Research to our Customization Framework

5.2.1 Customizations and the Customizer

The work of Nardi and Mackay, although directed primarily at supporting the User and their customizations, lends much strength to our argument that the User should not be the one to perform the majority of the customizations directly. Instead, their work can be seen as providing motivation for placing the Customizer (as we have in the Customization framework) to be the one responsible for customizing as much of the application as possible. This can be seen for three reasons. Firstly, the User typically finds the process of customization too difficult. Mackay documented difficulty being among the primary barriers to customization. Users who find customization too difficult will likely continue to use their software as-is and not realize the benefits of customized software.

Secondly, Mackay identifies that even when customizations are made by users, the customizations rarely involve extensive changes. Annoying aspects of the design (such as slow or repetitive tasks) are customized to become less annoying. Although important, these types of customization seem to only be scratching the surface of what can be done to help Users make better use of their applications.

Finally, through analysis of empirical studies, Nardi identifies that the appropriate method for performing customization is a formal language – such as a scripting language. Only a formal language is truly capable of performing all of the possibly necessary customizations to the software. Other interaction styles have some positive characteristics, but in general lack some feature or have some limitation that make them unsuitable for being a general purpose customization interaction style. This leaves the User, who does not have programming abilities, unqualified and unable to make the customizations through this style. Only the Customizer is able to make the necessary changes to the system through a formal language.

We realized that this last reason seems to contradict what Nardi reports in her user studies of spreadsheet and CAD users where Users did make some customizations through formal languages. However, we believe that spreadsheet Users and CAD Users do not represent average application users. In these two domains, programming is an integral part of the application. Spreadsheets require the user to gain skills in simple formulas and logic in order to use the primary features of the application. CAD applications, like spreadsheets, make heavy use of math as well as logic. Therefore, analysis of customization and programming with these applications cannot provide untainted, general results for all domains. Many applications do not involve skills such as formula construction, math, and logic, and thus would not teach or encourage the User to learn them. Without these fundamental skills, a natural progression towards programming

seems unlikely. Further studies that include a more diverse set of applications and users are needed.

However, the research of Nardi and Mackay does contain many valuable points that can help a Designer in designing the interaction style that can be used by the Customizer for customizations. In Table 5.1, we present our synthesis of the important elements for an effective interaction style.

Lesson	**Description**
Simple	Managing complex tasks should be possible with simple steps that can be built upon another or in series.
Concrete, Tangible and Visible	Don't hide information the user will want to see. Let them easily see and change information when possible.
Enticing / Salient	Provide and suggest opportunities to customize.
Focused on behaviour	Let the user change how the application works as well as what it does.
Readable and traceable	Make customizations easily accessible and clear. Users should be able to see and compare different customizations.
Shareable	Allow groups of users to use and share customizations
Support various skill levels	Most customizations should be simple but leave room for complex settings for those with programming expertise.
Based in a formal language	Provide an expressive format.
Tightly coupled to the domain	Use words and expressions from the user's domain. Mimic the methods and procedures familiar to the users.

Table 5.1 Lessons learned from previous work on end user programming.

It is unlikely that a Designer can provide a customization system that incorporates all the lessons we have extracted and presented here. In fact, some lessons may contradict each other in some scenarios (for example it may be difficult to maintain simplicity while supporting various skill levels). Instead, we suggest that these lessons be something that the Designer be aware of in his/her design. Also, awareness of these lessons during requirements gathering may help to elicit which lessons are more applicable and appropriate for a project.

5.2.2 Customization and the User

With the Customizer now being able to customize the application through either the source code and/or a formal language, where does that leave the User? Although a detailed answer on how a User can perform customization is not within the scope of this thesis, we present some of our thoughts on this topic. First, we believe that judicious application of non-programming interaction styles such as forms and programming by demonstration, though not appropriate as a robust general purpose customization format, are appropriate for selected aspects of customization by the User. In addition, we believe that part of the role of the Customizer's duties when performing customizations should be to specify the aspects that are appropriate for customization by the User and then provide appropriate interaction styles for each. This is justified by the Customization Model where we identified that the Customizer has necessary knowledge of what the User needs (and usually not the Designer).

From our review of end user programming research, Users need to customize their software in two main ways. The first is personalization [9]. Users want to customize the look and feel of the application to suit their own visual preferences and sense of aesthetics. Secondly, Users need to be able to update the behaviour of their tools to accommodate changes in their job. For example, a business process may include distinct phases that need to be re-ordered for optimization from time to time. The User should be able to tailor their application to reflect such a re-ordering. It is important that these customizations not require programming, because as demonstrated in our model, Users do not have the necessary programming skills.

5.3 Summary

The ability to customize an application without interacting with the source code can be important. Complex designs are perhaps best customized by using an interaction style that operates in some way on top of the application. The work of Mackay and Nardi, who are researchers in the field of end-user programming, provide us with a list of features that are ideal in an interaction style for customization and that, in essence, depict a custom scripting language. The result of having a specialized programming language further supports our assignment of the task of customization to the Customizer as opposed to the User. The Users, without skills in programming, are perhaps best suited to other interaction styles, which should most likely be selected and designed by the Customizer.

In the preceding two chapters of the thesis we have established three levels of customization. The first level is source code. Designers can organize and structure the source code using object oriented design, design patterns, components and frameworks to make the implementation comprehensible for the Customizers. The next level of customization is the formal language, which is also provided by the Desinger for use by the Customizer. The formal language enables the Customizer to customize many features of the application without the need to learn (and then change) the low level details system's source code. The last level of customization is the interaction styles that are designed by the Customizer for use by the User.

These additional concepts are incorporated into the Customization Model (Figure 5.1). This completes the model, as we have considered all of the stakeholders and how they do

customization. In the next Chapter we present a case study where we apply the software customization framework by applying it to the Map Explorer project.

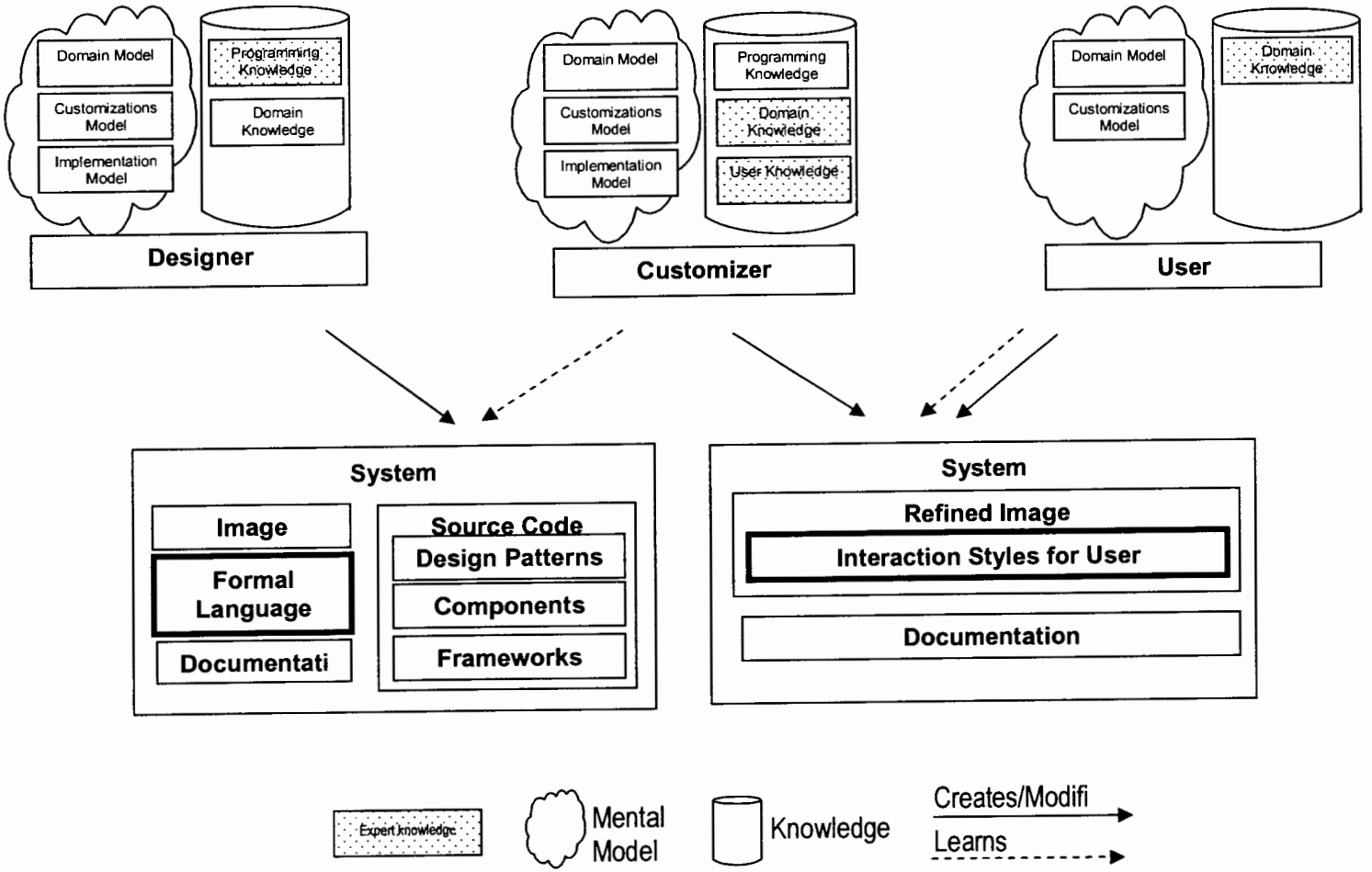

Figure 5.1 Expanded Customization Model incorporating ideas of a formal language for the Customizer and the Interaction Styles for user Customization

Chapter 6

Map Explorer Project

In this chapter we describe how our improved understanding of software customization helped considerably during the design of a customizable map explorer application. We first provide an overview of the Map Explorer project and describe the initial frustrations we encountered when trying to decide which features were to be customizable, how they should be built, and who would be responsible for developing the customizations for the Users. We then illustrate how we used the framework to help us answer these questions. First, we identify and describe the stakeholders involved in this project: The Designers, the Customizers and the Users. We then explain how the framework was used to guide the Designers during the design and implementation of the Map Explorer tool so that it effectively supported the customization needs of the Customizer. We provide details of

the architecture level support, as well as describe the formal end-user programming interaction style that was developed. We conclude the chapter with a summary of the customization facilities provided by the Map Explorer tool.

6.1 Introduction to the Case Study

The New Media Innovation Center (NewMIC) in Vancouver, British Columbia, Canada, is a new research center whose focus is to bring together academia with large and small businesses to work on novel projects that make use of the latest technologies. Our project team was composed of members from IBM's e-Commerce Pacific Development Centre and researchers from the University of Victoria. The focus of the collaboration was the construction of an Information Visualization Toolkit to be used in creating highly interactive web sites and applications. The first tool we developed for the toolkit was the Map Explorer, which is shown in Figure 6.1.

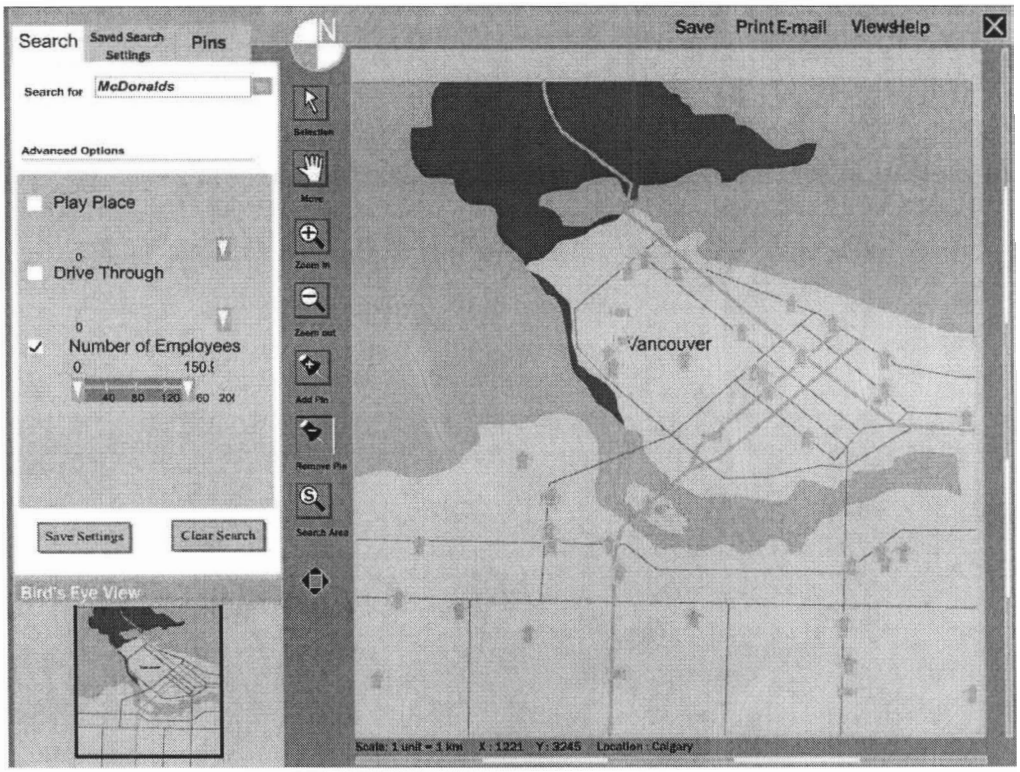

Figure 6.1 Map Explorer Tool

6.1.1 The Map Explorer

The Map Explorer tool provides an interactive environment for exploring a map with a set of locations superimposed on its surface. Different methods of interaction are available through a set of tools in a toolbox (along the left side of the map), a set of panels to perform searches (along the far left), menu options (along the top), as well as a secondary top level view of the map called the Bird's Eye View, located in the bottom left corner. Users can also mark up the map with tools such as the Pin tool from the toolbox.

Differentiating the Map Explorer from other similar tools is its ability to be used for different domains. The map, the locations of interest, the manner in which the tool supports searching, etc. can be adapted to support Users searching for locations in different domains. Examples of different domains include entertainment (restaurants, theatres), government (daycares, municipal offices), and telecommunication (coverage area, retail stores).

6.1.2 Why customization support was needed

In the beginning of the project, the team understood that the Map Explorer tool, if it was to be successful, would need to be highly configurable. This requirement was established for several reasons. First, the Map Explorer tool was intended to be deployable on multiple different platforms, such as websites, desktop applications, and potentially onto handheld devices; as well as for multiple domains, as previously described. The diverse set of installation situations establishes a wide variety of users with varying needs and abilities. Users may be novices or experts in computer use, have little or no experience with digital maps, and may or may not be familiar with the type and purpose of the

locations presented on the map. The Map Explorer tool can support this diversity if it is designed to be flexible, adaptable, and configurable.

Next, the resellers of the tools, IBM web development teams, had reasons for needing a sophisticated level of customization. They wanted tools, like the Map Explorer tool, that offer novel and enticing functionality for inclusion in e-Commerce projects. However, due to rigid time lines and limited resources, it is not feasible to build elaborate features, such as the Map Explorer tool, from scratch for most projects. Instead, they need to have a set of pre-built tools that can be fully customized and easily integrated into a project.

6.1.3 Why the customization framework was needed

Having a clear understanding of why customization was needed was only the first step. The team struggled to understand how this level of customizability was to be achieved in the Map Explorer tool. We understood our high level requirements but there seemed to be no easy way to translate these requirements into a design. In our discussions about customization, we had difficulties describing which features were to be customizable. The descriptions we used were often inconsistent and thus lead to misunderstandings amongst the team members.

We also felt that it was difficult to know when we had provided enough consideration for customization because we were unaware of all the different types of customization possible. The developers on the team also had difficulties envisioning a design for the system with the amount of customization support requested. Although the developers

had some experience making elements of a tool configurable, none had experience designing such a fully configurable and adaptable tool.

The struggles we experienced as a team made it apparent that we needed a better understanding of software customization. We needed to understand the challenges of building a highly configurable system and the implications it would have on our software process. We needed guidance in making these complex decisions so that our choices would produce a system with the level of configurability required. It was these needs that motivated us to spend the time to learn about software customization and develop our software customization framework. In the remainder of this chapter, we explain how we applied our software customization framework to the Map Explorer project.

6.2 The Designer, the Customizer, and the User

The first step in using our software customization framework is to realize how the stakeholders of the project fit within the roles of Designers, Customizers, and Users. In the following subsections we describe how we organized the Map Explorer team into the two groups of Designers and Customizers as well as provide some details about the Users.

6.2.1 Map Explorer Designers

In our software customization framework, we model the Designers as being responsible for designing and building the software system such that it can be fully customized by the Customizers for the Users. The framework also suggests that the Designers of the system should include members that have knowledge of the domain and expertise in programming. To satisfy these guidelines, the Designers of the system included a diverse

group. First, it included two researchers (including the author) from the University of Victoria who had knowledge of digital maps and information visualization. Next, it included two business analysts from IBM who were skilled at formalizing requirements in the format demanded by IBM projects.

The team also included three experts in programming: an IBM system's architect knowledgeable of IBM's backend systems, and two programmers proficient with the implementation technologies of Flash [43] and XML [49]. For the remainder of the chapter, the reference 'we' and 'Designers' is used interchangeably as the author was a member of the Designer group.

6.2.2 Map Explorer Customizers

As described in the software customization framework, the Customizers should be the individuals with the responsibility of tailoring a system provided by the Designers for different sets of users. In the Map Explorer project, the Customizers are the web Customizers on IBM's e-Commerce projects. These individuals are experts in graphic and information architecture and have sufficient skills in programming to manage the programming tasks associated with web development. On e-Commerce projects, Customizers are responsible for creating the needed web based tools and the site's interface. Their knowledge of their users is gained through either direct communication or through documentation that has been produced by business analysts on the team.

These Customizers will be the ones who decide to employ the Map Explorer tool on a client's website. They will be responsible for its customization as well as working with

programmers (Designers) in situations where the tool is integrated into any back-end systems.

6.2.3 Map Explorer Users

Initially, the Users will be the individuals visiting web sites designed by the IBM web development teams. Practically speaking, it is difficult to ascertain what their skills are going to be. It will be likely that they will have some domain knowledge about the website that they are visiting. Some of these users will be experts in that domain and some may just be casual browsers. Any other details will only become clear to the Customizers as they work with sample users. This lack of details regarding the Users reinforces the framework's recommendation that the Customizers must be the ones to do most of the customization. In our case, only the Customizers will have the necessary programming skills as well as an understanding of the Users needs to properly tailor the application.

6.3 Development Process of the Map Explorer

The development of the Map Explorer tool was split into several stages. In the first stage of the project, the Designers produced a set of documents that described the use-cases and the general features to be provided by the Map Explorer tool. These documents grew from the notes taken at Designer's brainstorming meetings and were evolved into formal documents by the business analysts among the Designers.

In the next stage, the Designers decided which features were to be customizable and how the tool was going to support their customizations. These decisions were guided by the software customization framework and by the opinions of the Customizers. It was

decided that customization support would enable the Customizers to customize the Map Explorer through its source code and with a formal language – as was recommended by the framework. The Designer's and the Customizer's agreement to follow the framework's recommendations provided us with some positive feedback on the framework's utility.

In the next stage, source code customization support was designed. Our design was based on several design patterns which accordingly led to a model composed of a collection of components and classes. With the design in hand, the implementation of the system began. Two of the Designers were responsible for coding the design. Regular meetings were held by the Designers (that occasionally included a Customizer) to ensure the development stayed on track and to update the design as needed. During these meetings, the formal language was developed and incorporated into the design.

Finally, when a prototype of the design was complete, the Customizers were given the opportunity to assess the tool. They customized the tool for a fictional domain. Details of the Map Explorer's design and its customization support are presented in the following sections.

6.4 Source Code Customizability

In the following sections we take a look at how source code customization is supported. The Designers followed the customization framework's recommendations and organized the implementation with levels of abstraction to provide better comprehensibility.

The Designer's first task in the development of the Map Explorer tool was to adapt the design of the Map Explorer to the development environment chosen: Macromedia Flash. We describe how the Designers developed some useful standards to ensure the Map Explorer's implementation's was not negatively affected by the somewhat awkward Flash development environment.

The next set of development choices was guided by our customization framework. The Designers of the system knew that object oriented design, design patterns and components were ideal choices to organize the implementation of the tool. However, the framework guided the use of those technologies to ensure that the Customizers could easily customize the tool. Emphasis was placed on simplicity and configurability. We show how the Designers structured the system with components, classes, and design patterns to enable the Customizer to quickly comprehend the implementation and customize it.

6.4.1 Implementation Language Issues

An important consideration in any project is the choice of implementation languages. Flash [43], a Macromedia tool, was chosen to be the primary language for the implementation of the Map Explorer. There were two primary reasons for selecting Flash. First, it has a powerful development environment that combines vector graphics, multiple time lines, and a programming language called ActionScript [50]. Second, Customizers at IBM had extensive experience working with Flash and believed that it would lead to quick results.

Unfortunately, the Flash environment can be very confusing to work with. First, entire projects are rarely implemented as separate files; all of the code and graphics are stored in one Flash file, which hinders separation of concerns. Second, since it is an animation tool, Flash implementations are structured according to two dimensions: time and space. Scenes are organized with a timeline dimension to direct how it changes over time. Each scene is also organized into layers where each major graphic is typically contained within its own layer. The result of this structure is a matrix of frames as shown in Figure 6.2.

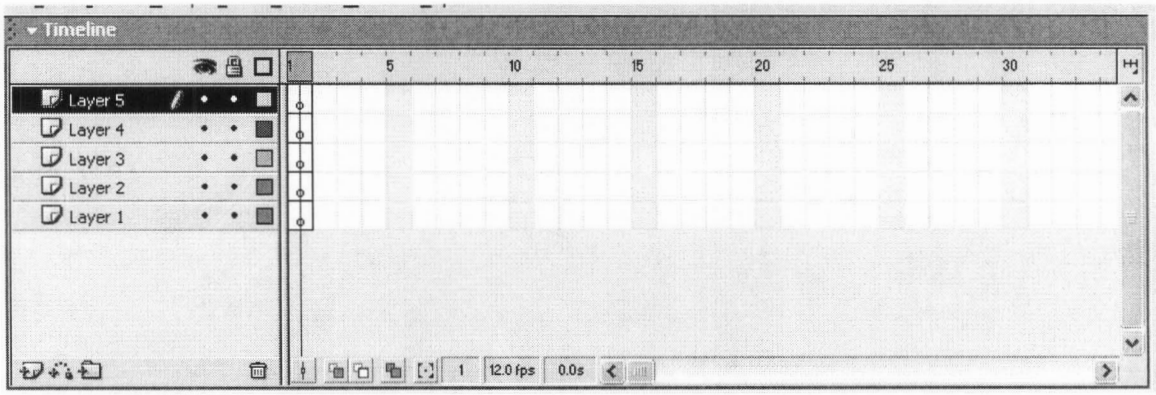

Figure 6.2 An empty matrix of frames (the cells in the above grid). Layers are organized along the left and time is measured along the top.

ActionScript, the language used in Flash, can be embedded into any frame within a Flash project. In addition, each graphic is like a scene; it contains a timeline and layers and therefore can contain ActionScript as well as other graphics. Although this structure enables Flash developers to create sophisticated interfaces, it also permits developers to create applications that are extremely hard to understand. Code can be dispersed throughout the project in both dimensions and can be hidden within layers of nested graphics.

To address this issue, the Designers were required to centralize the code as much as possible and to document all instances where code was placed outside this central location. The location chosen to centralize the code was the main scene of the Map Explorer Flash movie. This is shown in Figure 6.3 where each class has been placed on its own level, making the classes easy to organize, and then later, find.

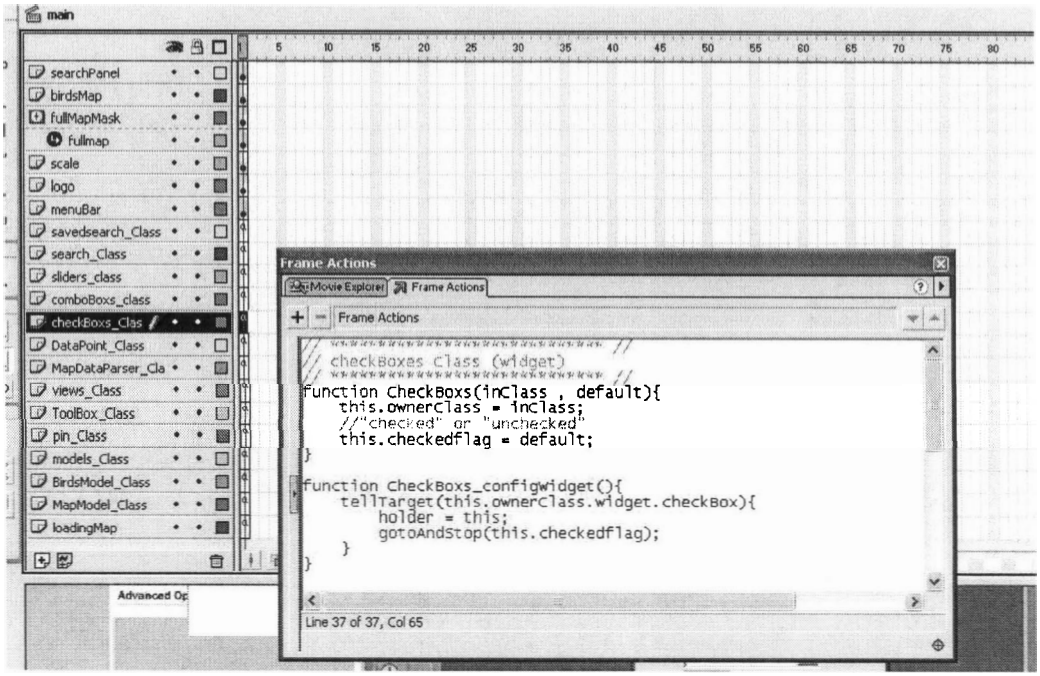

Figure 6.3 Flash 5 development environment with code centralized for better comprehension and customizability support.

These decisions contributed significantly to enabling **presentation customization.** Centralizing the code decoupled the interface from the graphical elements of the implementation to a large degree. The documentation provided was also useful. It detailed how any new graphical user interface elements could be added to the system and hooked into the existing code. Thus the methods to customize the information architecture or graphic design were well documented and enabled.

6.4.2 Component Design

Centralizing the code so that it could be read and easily found was only a start to properly structuring the implementation to support customization as well as comprehensibility. However, the most influential aspect of the design was to realize the features of the Map Explorer tool as a set of high level components. A diagram of this design (as seen in Figure 6.4) is very readable by a Customizer as each component can be mapped to concepts from their Domain Knowledge of the Map Explorer.

For example, the concept of the Birds Eye View describes the secondary navigation view located in the lower left corner of the Map Explorer tool. Customizers wanting to customize the code for this view would have to look no further than the "Birds Map Component". In essence, our component design establishes a top-down view that enables Customizers to easily begin navigating the software's architecture.

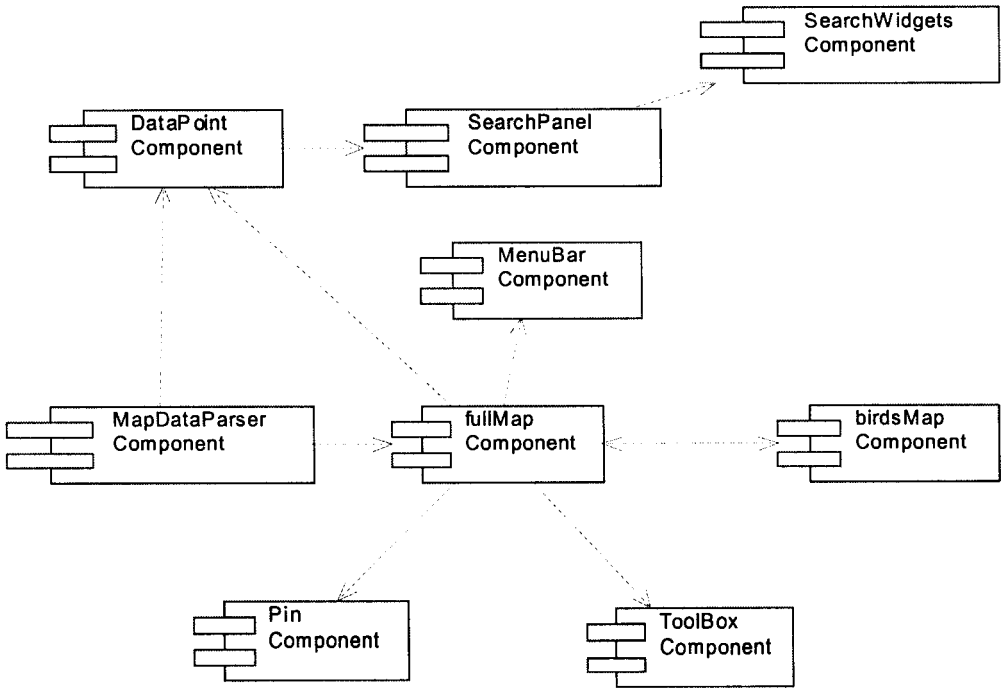

Figure 6.4 Map Explorer Component Diagram

Each of these components is then realized as a set of classes so that the Customizer can continue the use of a top-down approach for navigating the system. Once at a class level, the Customizer should be able to locate the segments of code that they will need to change to achieve their desired customization.

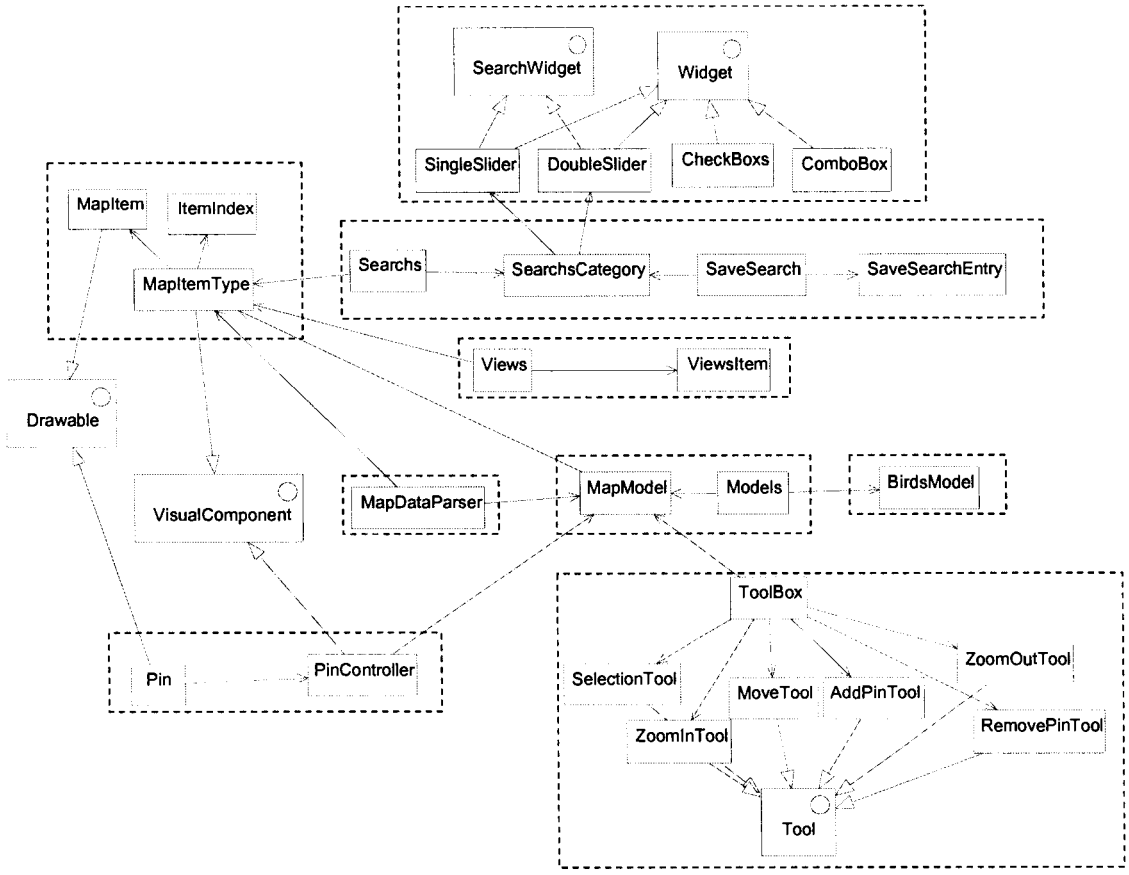

Figure 6.5 Map Explorer Class Diagram. Classes and Interfaces surrounded by boxes to identify the components described in Figure 6.4.

6.4.3 Components and Customization

In this section we take a look at how customization is supported more specifically in the Map Explorer tool by using components. Although a component architecture supports customization in almost all ways needed (see Table 4.2), here we describe how components best support direct customization (source code customization) in the Map

Explorer tool. With that in mind we will take a look at Feature Selection, Feature Addition, Feature Enhancement and Feature Constraint.

Feature Selection by the Customizer is ultimately achieved through the benefits of components. As each feature is encapsulated in a component, removing or adding features from an existing list of features involves enabling/disabling components within the system. Since the connections between components have been minimized, this can be managed with a small amount of effort. For instance, removing search facilities from the tool is possible by removing both the SearchPanel Component along with the SearchWidgets Component.

Feature Addition requires writing code and thus does require more effort. However, by taking a look at how other components are constructed and connected to the system (through the documentation as well as the source code), the effort involved in architecting the addition is simplified. In fact, it is hoped that providing a clear architecture that organizes each major feature into a component, that Customizers will do the same with any additions. For example, if the Customizer wished to enable the User to be able to place other kinds of markers onto the map, such as the Pin, that the Customizer would have the component or class implement the Drawable and VisualComponent interfaces.

To realize **Feature Enhancement** and **Feature Constraint**, the appropriate components within Map Explorer offer the flexibility of being specialized. In addition to the interfaces provided for its primary services, additional interfaces have been incorporated

for plug-ins that would enable certain services to be specialized. For instance within the ToolBox component there is an interface plug-in that enables the addition (or removal) of individual tools (such as Zoom In and Zoom Out). Each tool is realized as a class that conforms to the plug-in interface, enabling the component to treat all tools the same. This organization and structure allows the toolbox component to provide a vast number of tools and tool combinations but in such a manner that it is easy for the Customizer to program.

6.4.4 Design Patterns

Each component of the component design encapsulates a feature and thus provides excellent avenues for customizing that feature. However, when several features have to work together, design patterns are an effective means to customizing **Feature Coordination**. Here we will describe how a design pattern helps to structure how multiple components can be used together to administer a set of tasks.

To manage the storage and access of data (**Data Customization**) within the Map Explorer tool, a Model View Controller [51] design pattern was incorporated into its design. The Model is made up of collections of MapItem and ItemIndex classes which have database-like methods for data insertion and retrieval. The Controller is made up several instances of the MapItemType class, one for each different type of data, and as it name implies, controls access to the Model. The different Views, Main Map and Search Panel, are provided by their respective components and acquire the data they need from the Model by interacting with the Controller(s). This structure is represented in Figure 6.6.

Even though many components of the system are involved, once the design pattern is understood, customizing how these features work together becomes simplified. For instance, additional views can be added by simply registering them with the Controller. Synchronizing data related events among the Views can also be done by making the appropriate changes in the Controller. Improving the efficiency of data retrieval by applying better search algorithms in the Model and Controller is yet another example.

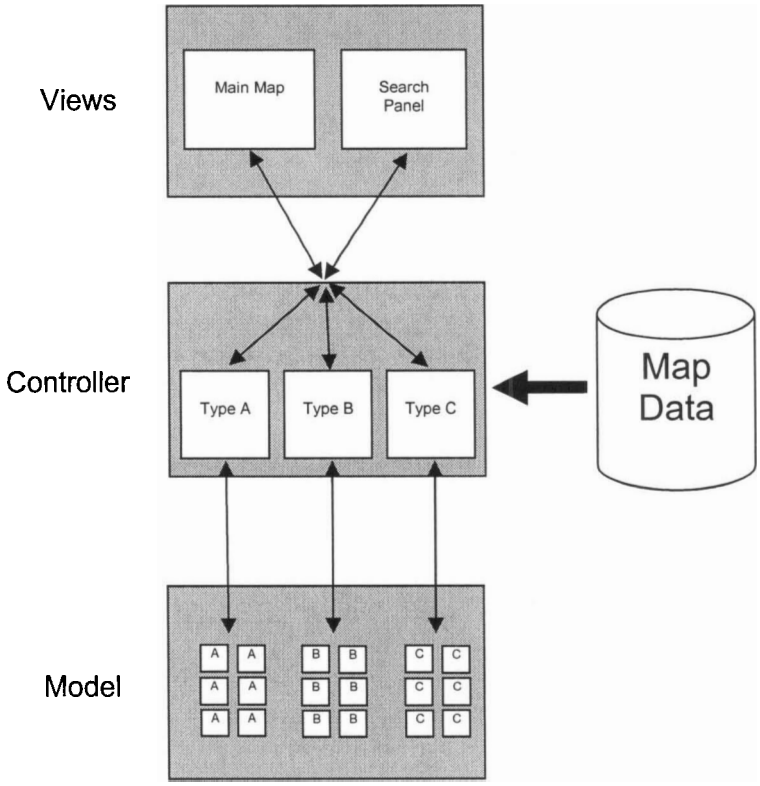

Figure 6.6 Use of a design pattern to manage data

6.4.5 Source Code Customization Summary

With the use of some coding conventions in addition to an application of architecture elements such as components and object oriented design, the source code of the Map Explorer tool is organized and structured. This enables our Customizers to understand how the tool works and to customize its operations to better suit their needs. Not only

that, but our use of components and a design pattern in this first iteration of the project makes extracting a framework from our work more likely in future iterations. As mentioned in Chapter 4, frameworks are difficult to produce as they are the result of continuous refinement and thus are not a realistic goal from the first implementation of a project.

We have also shown in this section how the source code can be customized in terms of data, control/behaviour, and presentation. Data customization is supported with the MVC design pattern; presentation customization is made possible due to our deliberate separation of the interface from the source code; and control customization is provided by our use of components, object oriented design, as well as the MVC design pattern.

However, it is not always necessary to make changes to source code in order to customize an application. In this next section we take a look at how Customizers can customize many aspects of the Map Explorer tool without having to interact directly with its implementation.

6.5 Non Source Code Customizability

The Customizers that we consulted with expressed a desire to have an interaction style to customize the Map Explorer tool. Due to their busy schedules, Customizers may not have the time to perform all the needed customizations via the source code of the Map Explorer. Some tasks are inherently complex resulting in source code that is difficult to read and understand regardless of the quality of the architecture. Also, the Customizers expressed a desire to have their customizations shareable and external to the tool.

Therefore, we decided to provide the Customizers with a formal language (as recommended by the framework) they could use to specify their customizations.

6.5.1 Embedding Customizations in XML

The interaction style chosen for the Map Explorer tool was to embed settings and customizations within an XML[49] format. This would mean encoding customizations into an XML file (making it a formal language) which would be read into the Map Explorer tool for interpretation. There were several reasons for choosing XML. First, the Customizers have existing skills in writing, reading, and exchanging XML. Second, XML was already in use in the project to exchange data for the map data points. Third, XML has high third party tool support and fourth, it allows for the customizations to evolve over time.

Finally, we chose XML because it helped us to fulfill some of the requirements for a successful interaction style (which were presented in Chapter 5). Namely, XML is generally readable, shareable, and simple (simple because the Customizers already knew XML). The other ideal characteristics for an interaction style were also built into the format and are discussed later.

This formal language is used to author the Map Explorer settings file. This document is split into two segments: one segment that describes how the map data is to be interpreted and searched, and a second segment that encodes customizations directed at presentation and control/behaviour features. In the following sections, details of these two segments will be provided as well as our initial evaluation of this approach to customization.

6.5.2 Data Customization

The first section of the settings file will enable the Customizer to quickly customize the data being presented in the Map Explorer tool. A sample of the first section of the settings file is displayed in Listing 6.1. The intent of this section of the XML is to customize how the Map Explorer tool will interpret a set of map data points. From the XML in Listing 6.1, we can see that it describes two types of map data points by noting the two itemType elements with the id values of **McDonalds** and **Pizza Hut**. Within the description of each of these types there are two types of elements that are used for encoding customizations. The first element is the attr child element which is used to describe the itemType in general. In the examples below, details regarding the title as well as the icon to be used for locations of the type on the map are described with attr elements.

Additional characteristics are specified with the itemAttr child elements. These elements allow the Customizer to describe the meta-data of the map data points by encoding the names, the default values, and the data types of these meta-data. In addition, the Customizer can specify the search widget (if any) associated with each meta-data and thus has control of how the search panel will appear when the user is searching for data points of that type. So far, these customizations are relatively simple and straight forward.

```
<itemType id="McDonalds">
 <attr name="title" value="McDonalds" />
 <attr name="icon" value="buildings.McDonalds" />
 <itemAttr name="Address" default="" type="String" search="none" />
 <itemAttr name="City" default="Vancouver" type="String" search="ComboBox" />
 <itemAttr name="Postal Code" default="" type="String" search="None" />
 <itemAttr name="Play Place" default="no" type="Number" search="SingleSlider"/>
 <itemAttr name="Drive Through" default="no" type="Number" search="SingleSlider" />
 <itemAttr name="Number of Employees" default="0" type="Number" search="DoubleSlider" />
 <itemAttr name="notes" default="Unknown" type="String" search="none" />
```

```
 <itemAttr name="x" default="-1000" type="Number" search="none" />
 <itemAttr name="y" default="-1000" type="Number" search="none" />
</itemType>

<itemType id="PizzaHut">
 <attr name="title" value="PizzaHut" />
 <attr name="icon" value="buildings.PizzaHut" />
 <itemAttr name="Address" default="" type="String" search="none" />
 <itemAttr name="City" default="Vancouver" type="String" search="none" />
 <itemAttr name="Postal Code" default="" type="String" search="None" />
 <itemAttr name="Play Place" default="no" type="Number" search="SingleSlider" />
 <itemAttr name="Drive Through" default="no" type="Number" search="SingleSlider" />
 <itemAttr name="Number of Employees" default="0" type="Number" search="DoubleSlider" />
 <itemAttr name="notes" default="Unknown" type="String" search="none" />
 <itemAttr name="x" default="-1000" type="Number" search="none" />
 <itemAttr name="y" default="-1000" type="Number" search="none" />
</itemType>
```

Listing 6.1 XML for customization of managing data in the Map Explorer

This simplicity should not be mistaken for a lack of capability. These simple settings empower the Customizer with the ability to customize how the Map Explorer (without source code modifications) will import, store, and present each element of map data.

6.5.3 Presentation and Control/Behaviour Customization

In the next section of the XML customization document, the focus is on customizing the features of the Map Explorer tool. In Listing 6.2, we see an example of the customizations for the **Toolbox, the Birds Eye View**, the **Search Panel** and the **Full Map** features. Each XML Feature element contains the settings for one particular feature of the tool. For example, in the **Toolbox** Feature element there are eight settings. The first setting, WhenToShow, contains its value within an element called Simple and has a value of **Always** indicating that this feature is to be always shown. The next seven elements are Setting elements and indicate the position of the **Toolbox** as well as which tools are to be made available in the **Toolbox**.

More dynamic settings are apparent when looking at the settings for the **Search Panel**. The WhenToShow element in the **Search Panel** settings has a Dynamic child element

(instead of a Simple one as described above) where a formula attribute is set to locations.count >10 which indicates that the search panel does not need to be provided if the data downloaded to the map contains 10 or less locations. This simple control logic provides for dynamic control, is easy to read, easy to change, and is easily incorporated amongst the simpler settings.

```
<Feature id="Toolbox">
 <WhenToShow>
  <Simple value="Always"/>
 </WhenToShow>
 <Setting name="Position" value="W"/>
 <Setting name="MoveTool" value="Yes"/>
 <Setting name="ZoomInTool" value="Yes"/>
 <Setting name="ZoomOutTool" value="Yes"/>
 <Setting name="AddPinTool" value="No"/>
 <Setting name="RemovePinTool" value="No"/>
 <Setting name="SelectionTool" value="Yes"/>
</Feature>

<Feature id="BirdsEye">
 <WhenToShow>
  <Simple value="Always"/>
 </WhenToShow>
 <Setting name="Position" value="SE"/>
 <Setting name="Size" value="Small"/>
</Feature>

<Feature id="SearchPanel">
 <WhenToShow>
   <Dynamic formula="locations.count >10"/>
 </WhenToShow>
 <Setting name="Position" value="E"/>
</Feature>

<Feature id="FullMap">
 <WhenToShow>
  <Simple value="Always"/>
 </WhenToShow>
 <Setting name="ZoomSpeed" value="Fast"/>
 <Setting name="PanSpeed" value="Slow"/>
 <ConditionalSetting name="DigitalMap">
  <Situation when="ZoomLevel<5" choice="VancouverGeneral">
  <Situation when="ZoomLevel>5" choice="VancouverDetailed">
 </ConditionalSetting>
</Feature>
```

Listing 6.2 XML for customization of how features behave.

Another example of a dynamic setting occurs in the **Full Map** Feature element. Here, the selection for the source of the digital map is contained with an element called ConditionalSetting. This element contains two child elements called Situation which

encodes the testing condition within a when attribute and the associated setting within a choice attribute. Reading this setting indicates that the digital map chosen for the Map Explorer is dependent on how far zoomed-in the user is. When the user is zoomed-in only as far as the 4th level, the user will see the **VancouverGeneral** map. When the user zooms in further, the Map Explorer tool will load the **VancouverDetailed** map to provide greater map detail.

6.5.4 Evaluation of the XML Interaction Style

This interaction style was designed with the intention of meeting the requirements we extracted and summarized from the work of Nardi and Mackay. We describe in Table 6.1 how our XML based formal language reflects, to varying degrees, the desirable characteristics of an interaction style. For instance the formal language can be described as enticing, as being concrete, and as supportive of various skill levels. However, a formal evaluation of the interaction style would be needed to accurately assess its usefulness, applicability, and how it exhibits the desirable characteristics.

Desirable Interaction Style Characteristic	Map Explorer's XML Customization Formal Language
Simple	XML is well known by the Customizers, has high tool support, and the designed schema is semantically rich.
Concrete, Tangible and Visible	Most aspects of the customization are visible: however dynamic sections refer to non visible items.
Enticing	All the customizations are organized by feature. As the Customizer customizes one characteristic of a feature, he/she will see the other characteristics available for customization.
Focused on behaviour	Customization is focused on how individual features behave.
Readable and traceable	XML is a readable format with high tool support to perform comparisons of different versions.
Shareable	Since the customization is not embedded within the tool, customizations are easy to share.
Support various skill levels	Simple and dynamic customizations available. More challenging customizations still possible by modifying the source code.
Based in a formal language	Format is flexible and formal
Tightly coupled to the domain	The tags in the XML format directly reflect the domain.

Table 6.1 Summary of how XML Customization format supports the requirements

6.6 Map Explorer Customization Summary

The Map Explorer project presented us with the motivation for developing, and an opportunity for applying, our customization framework. With the customization framework in hand, we organized the stakeholders of the Map Explorer project into Designers, Customizers (not Gurus in this case, since they were not users as well), and Users. The Designers of the system used the framework to ensure their design included the necessary support for customization. They used the framework as a checklist to ensure that the needed flexibility would be incorporated into the tool. Then, as suggested by our customization framework, the Designers constructed the Map Explorer implementation with technologies that produce useful high level abstractions. Components were built with objects, and where needed, coordinated with design patterns. The implementation was also specially structured to enable the source code to be more readable within the chosen development technology.

The Designers then used the lessons we learned from previous researchers as a guide and produced an XML-based formal language. In summary, the Map Explorer project proved to us that the Customization Framework was a definite help in designing and building a tool that is not only customizable from a technical standpoint, but a practical one as well. Summarized in Table 6.2 is the distribution of 'how' each of the types of customization we identified in Chapter 2 was achieved for the Map Explorer tool.

What	How			
	Design Patterns (MVC)	Components	Interaction Style (XML)	Other (Centralizing Source Code)
Presentation: Information architecture			*	*
Presentation: Graphical Design			*	*
Data: Format			*	

Data: Content				
Control: Feature Selection		*	*	
Control: Option Specification			*	
Control: Feature Addition		*		
Control: Feature Enhancement	*	*		
Control: Feature Constraint	*	*	*	
Control: Feature Coordination	*		*	

Table 6.2 Distribution of how the Map Explorer tool can be customized.

6.7 Customization by the End User

The focus of our customization framework was on giving guidance to the Designer to design and build a customizable system which was to be tailored by the Customizer for Users. Little work was done on understanding how to best create customization facilities for the Users by the Customizers. However, as explained in Chapter 5, the Customizer at minimum should enable the User to personalize their software. In addition, if it is known how the User's needs would evolve over time, the facilities to adapt to those changes should be provided.

Working with the Customizer, we decided the following facilities should be given to the user to customize their tool:

1. A set of skins to choose from to tailor the look and feel.
2. A function to select custom icons for the data points.
3. The ability to re-arrange the components around the window.
4. The ability to load and save multiple versions of a data set.

Further exploration of how to effectively design customizations tools for end users requires more study and is a topic for future work.

6.8 Summary

In this chapter we presented a case study to demonstrate the need for and utility of our software customization framework. We believe that this is only a partial assessment as there is still obvious work to be done in terms of a formal evaluation. However, we feel

strongly that our software customization framework was of great benefit in this case study. In the next chapter we will conclude with a discussion of why we felt this case study was successful and provide details of our current and future work.

Chapter 7

Conclusions and Future Work

We conclude the thesis with a discussion of how the customization framework was indeed a worthwhile pursuit as well as provide an overview of our future work.

7.1 Notes on Applying the Customization Framework

The application of our customization framework to the design and development of the Map Explorer tool was worthwhile. Decisions regarding design, architecture, and how the Map Explorer tool was to be deployed were guided by our framework; this made those tasks more manageable and less demanding. This contrasted with our initial work on the Map Explorer project where we were frustrated and impeded by our lack of understanding of customization. This floundering was useful, however, as it lead us to

the development of the software customization framework which enabled success in this project and in future projects.

Several elements of the framework contributed to this success. Particularly noteworthy is the classification presented in Chapter 2. It was of great help to have this breakdown of customization when designing and building the tool. The different types of customization acted as a checklist, ensuring that each of the elements was considered and accounted for in the design.

Next, the framework helped us organize the team from IBM and NewMIC into Designers and Customizers, and guided the Designers to tailor any customization facilities towards the Customizer more than for the User. The framework was also useful in providing a set of recommended methods to implement the chosen customizability. We believe that the framework was general enough that it did not constrain our efforts, while still providing some specific assistance in our design and implementation tasks.

Current applications of the framework are proving to be just as successful as the Map Explorer experience. We are applying the framework to help us increase the customizability of an information visualization tool called SHriMP [52]. Although domain independent, the users of SHriMP have struggled to tailor the tool to meet their needs and to work effectively with their data. We are using the framework to guide us as we include additional support for customization. For example, we are adding scripting support (in the form of a formal language) to the tool.

Another current project is the Gild [53] project, which is adapting the Eclipse IDE for use in first year classes. In this project we are using the framework to explain the aspects of Eclipse that enables our required customization as well as indicate where Eclipse's design impedes our progress. Furthermore, the framework has also helped us make the Gild project customizable by the instructors and TAs (the Gurus, in this case).\

7.2 Limitations

Our work in software customization will not be of use to all projects needing customization support. First, our framework does not address how to incorporate customization into existing software tools that currently do not support the required customization. Second, the framework does not suggest how it can be used in situations where there are special considerations concerning privacy as is the case in security software and critical systems. An understanding of customization unique to those domains may need to be uncovered to provide guidance to those types of projects when incorporation of customization is necessary.

7.3 Contributions

This work offers many contributions to the software engineering research community, to the businesses whose core tasks involve designing and developing customizable software, and to the organizations that use the resulting tools and applications.

First, this thesis has formally made clear the need for software engineering research that addresses large and difficult issues. We need to improve our understanding of major software concerns and develop models, strategies, and guidelines to help steer software

projects in the right direction. Detailed study and work on classical research topics, such as software architecture, tool design, documentation, and programming languages are very important. More important, however, is research that connects research areas to address issues that crosscut these disciplines and our software processes and lifecycles. Our work in software customization is an example of this. We have taken work from software architecture, program comprehension, requirements, and HCI to discover an understanding of customizable software. These disciplines and how they are connected are presented in Figure 7.1

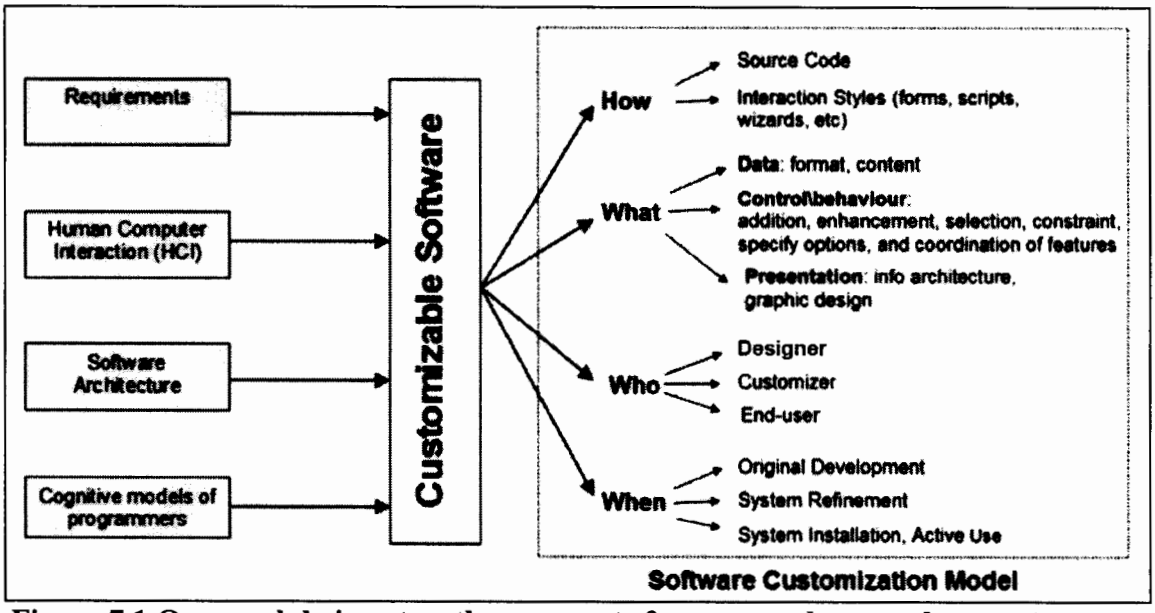

Figure 7.1 Our work brings together concepts from several research areas to create an understanding of software customization.

Also shown in Figure 7.1 is our description of software customization which organizes its issues according to how, what, who, and when. We believe this to be a particularly useful contribution to software engineers because it provides a vocabulary to discuss and document software customization issues and features. It helps software engineers to

create a mental model of software customization, which, according to our research in mental models, will help them make decisions and correct mistakes in their designs.

Also, in this work we provide guidance for not one issue, but for several issues related to software customization. We were able to understand these diverse issues by, once again, bringing together elements from several areas of research. We evolved a model for the software customization process from existing HCI and program comprehension models. The model we produced exposes the mental models and knowledge of each set of stakeholders as well as the interactions had with the software system.

From this holistic view of software customization we were able to produce some important recommendations for stakeholders in projects involving customizable software. More specifically, we directed the Designers to provide tailored customization support for a group known as the Customizers. We were also able to suggest that it should be the Customizers to develop customization facilities for the User as they will know their needs best.

We then elaborated on what this support could include by exploring the two 'hows' of customization: source code and interaction styles. In our research of source code technologies, we gained a better understanding of how object oriented design, components, design patterns, and software frameworks can be constructed and assembled to produce comprehensible software designs. We also explored the work on interaction styles and discovered their role in software customization, including a set of ideal

characteristics that an interaction style should be instilled with. Throughout these efforts we updated our software customization model to reflect what we have learned.

We suggest, once more, that the integration of the diverse research domains offers useful insights. Giving developers the ability to see and work with the 'big picture' will help improve designs and the success of projects. Being able to relate decisions from high level requirements to low level implementation details is another dimension of understanding that can be achieved with such holistic integrations.

7.4 Ongoing and Future Work

Currently we are applying the framework in helping us re-architect a domain independent visualization tool (called SHriMP [52]). The framework is proving very helpful in helping us decide how the architecture will enable or constrain customizations by a guru and by the end-users. We have struggled for the past few years on making such decisions, and having the framework has provided clarity and consistency in our current design approaches. We are, at this time, integrating scripting support (a type of formal language) to better support gurus to customize the tool for end users. We are confident that this will trigger future Shrimp users to customize the tool to better support their particular needs.

We have also been using our customization framework on a project called Gild. In this project we are customizing the Java development platform and software framework called Eclipse. The goal of the Gild project is to simplify Eclipse's Java development tools and make them more appropriate for first year students. We are also making use of

Eclipse's plugin architecture to integrate new features such as collaborative support and software visualization. We have used this project to evaluate Eclipse's customization abilities (identifying its strengths and weaknesses) and are sharing these results with IBM and the research community [48].

Further thought and research into documentation to support software customization is also needed. How should customizable systems be best documented by the Designer to ensure that future Customizers can determine the best way to customize the system? We have encountered this issue repeatedly while customizing Eclipse in the Gild project. Often Eclipse has the customization support we need but it takes significant effort on our part to locate and leverage the customizable elements. We also want to increase our understanding of how a Customizer can design customization facilities for Users. New approaches and best practices from HCI and related fields will need to be synthesized and incorporated into the framework.

Finally, we wish to gather feedback from these projects and endeavours to help guide the evolution of our customization framework. As the framework matures and evolves, formal evaluations will be needed to verify and amend aspects of this work.

7.5 Concluding Remarks

We believe that this framework begins to fill an important gap in software engineering. Best practices for designing, developing, deploying, and maintaining highly customizable systems needs to be researched, discovered, and documented. Our framework offers the research community a vocabulary to discuss and describe customization in their software

and software designs as well as a model that typifies how software is and or should be customized.

We believe that the need for customizable software that is feasible to build, possible to deploy, and successful in meeting the needs of users will continue to grow as time progresses. Finally, we hope that we have inspired others to research this topic and contribute to the knowledge of how to cope, manage, and tackle these difficult challenges.

References

[1] P. Dourish, "Developing a Reflective Model of Collaborative Systems," *ACM Transactions on Computer-Human Interaction*, vol. 2, pp. 40-63, 1995.

[2] B. A. Nardi, *A Small Matter of Programming*. Massachusetts: The MIT Press, 1993.

[3] "Eclipse Platform Technical Overview," IBM, http://www.eclipse.org/whitepapers/eclipse-overview.pdf, 2003.

[4] D. Sprott, "Enterprise resource planning: componentizing the enterprise application packages," *Communications of ACM*, vol. 43, pp. 63-69, 2000.

[5] K. C. Kang and P. J. L. Donohoe, "Feature-oriented product line engineering," *IEEE Software*, vol. 19, pp. 66-72, 2002.

[6] D. Sharon, "Meeting the Challenge of Software Maintenance," *IEEE Software*, vol. 13, pp. 125-127, 1996.

[7] G. Fischer and E. Scharff, "Meta-design: design for designers," presented at Conference on Designing interactive systems, processes, practices, methods and techniques 2000, New York, New, York, 2000.

[8] P. Dourish and W. K. Edwards, "A Tale of Two Toolkits: Relating Infrastructure and Use in Flexible CSCW Toolkits," *Computer Supported Cooperative Work: The Journal of Collaborative Computing*, vol. 9, pp. 33-51, 2000.

[9] J. Price, "The audience of one: making professional communication personal"," presented at Conference on Professional Communication (IPCC), 2001.

[10] N. Shedroff, "Information Interaction Design: A Unified Field Theory of Design," http://www.nathan.com/thoughts/unified/unified.pdf.

[11] W. E. Mackay, "Triggers and Barriers to Customizing Software," presented at Human factors in computing systems, New Orleans, Louisiana, 1991.

[12] W. E. Mackay, "Patterns of Sharing Customizable Software," presented at Computer-Supported Cooperative Work, Los Angeles, California, 1990.

[13] A. v. Lamsweerde, "Requirements Engineering in the Year 00: A Research Perspective," presented at 12th International Conference on Software Engineering (ICSE), Limerick, Ireland, 2000.

[14] A. Wasserman, "Tool Integration in Software Engineering Environments," presented at international workshop on environments on Software engineering environments, Chinon, France, 1990.

[15] J. Armitage, "From user interface to über-interface: a design discipline model for digital products," *interactions*, vol. 10, pp. 18-29, 2003.

[16] J. C. Cleaveland, "Building Application Generators," *IEEE Software*, vol. 5, pp. 22-33, 1988.

[17] "SAP," www.sap.com.

[18] N. Pennington, "Stimulus structures and mental representations in expert comphrehension of computer programs," *Cognitive Psychology*, vol. 19, pp. 295-341, 1987.

[19] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway, *Mental models and software maintenance*: Ablex Publishing Corporation, 1986.

[20] S. Letovsky, "Cognitive process in program comprehension," presented at Empirical Studies of Programmers, 1986.

[21] E. Soloway, J. Pinto, S. Letovsky, D. Littman, and R. Lampert, " Designing documentation to compensate for delocalized plans," *Communications of ACM*, vol. 31, November 1998.

[22] A. V. Mayrhauser and A. M. Vans, "Program comprehension during software maintenance and evolution," *Computer*, vol. 28, pp. 44-55, 1995.

[23] D. A. Norman, *The Design of Everyday Things*: Currency and Doubleday, 1988.

[24] R. Guindon, "Knowledge Exploited by Experts during Software Systems Design," *Int'l J. Man-Machine Studies*, vol. 33, pp. 279-182, 1990.

[25] V. Rajlich, N. Damaskinos, and P. Linos, "VIFOR: A tool for software maintenance," *Software--Practice and Experience*, vol. 20, pp. 67-77, January 1990.

[26] M. Steckel, K. Brade, M. Guzdial, and E. Soloway, "Whorf: A visualization tool for software maintenance," presented at 1992 IEEE Workshop on Visual Languages, Seattle,Washington, 1992.

[27] M.-A. Storey and H. A. Müller, "Manipulating and Documenting Software Structures using SHriMP Views," presented at 1995 International Conference on Software Maintenance (ICSM '95), Opio (Nice), France, 1995.

[28] H. A. Müller and K.~Klashinsky, "Rigi --- A system for programming-in-the-large," presented at 10th International Conference on Software Engineering (ICSE'10), Raffles City, Singapore, 1988.

[29] T. P. Vayda, "Lessons from the battlefield," presented at Tenth annual conference on Object-Oriented programming, systems, languages, and applications, Austin Texas, 1995.

[30] T. Korsen and D. G. McGregor, "Understanding Object-Oriented: A Unifying Paradigm," *Communications of the ACM*, vol. 33, 1990.

[31] G. Larsen, "Designing component-based frameworks using patterns in the UML," *Communications of the ACM*, vol. 42, pp. 38-45, 1999.

[32] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdal-King, and S. Angel, *A Pattern Language*. New York: Oxford University of Press, 1977.

[33] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Don Mills, Ontario: Addison Wesley, 1999.

[34] G. Booch, *The Unified Modeling Language User Guide*. MA: Addison Wesley, 1998.

[35] M. W. Price, D. M. Needham, and S. A. D. Sr, "Producing Reusable Object-Oriented Components: A Domain-and-Organization-Specific Perspective," presented at Symposium on Software Reusability, Toronto, Ontario, 2001.

[36] D. E. Souza and A. C. Wills, *Objects, Components, and Frameworks with UML*. Don Mills, Ontario: Addison-Wesley, 1998.

[37] "COM," http://www.microsoft.com/com/default.asp.

[38] "Java Beans," http://java.sun.com/products/javabeans/.

[39] "Corba," http://www.corba.org/.

[40] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*: Addison-Wesley, 1998.

[41] D. Sprott, "Enterprise resource planning: componentizing the enterprise application packages," *Communications of the ACM*, vol. 43, 2000.

[42] C. Verbowski, "Integrating Java and COM," http://air.knu.ac.kr/reference/COM/java_com.htm, 1999.
[43] "Macromedia Flash," http://www.macromedia.com/software/flash/.
[44] R. E. Johnson, "Components, frameworks, patterns," presented at 1997 symposium on software reusability, Boston, Massachusetts, 1997.
[45] G. Larsen, "Designing component-based frameworks using patterns in the UML," *Communications of the ACM*, vol. 42, pp. 38–45, 1999.
[46] M. Fayad and D. C. Schmidt, "Object-oriented application frameworks," *Communications of the ACM*, vol. 40, 1997.
[47] D. Roberts and R. Johnson, "Evolving Frameworks," http://st-www.cs.uiuc.edu/users/droberts/evolve.html.
[48] J. Michaud, M.-A. Storey, and R. Lintern, "A Customization Model for Evaluating Software Frameworks," University of Victoria, Victoria, BC 2003.
[49] "XML," http://www.w3.org/XML/.
[50] "Action Script," http://www.macromedia.com/support/flash/action_scripts.html.
[51] S.S.Adams, "MetaMethods: The MVC paradigm," *HOOPLA*, vol. 1, pp. 5-21, 1998.
[52] C. Best, M.-A. Storey, and J. Michaud, "Designing a Component-Based Framework for Visualization in Software Engineering and Knowledge Engineering," presented at Fourteenth International Conference on Software Engineering and Knowledge Engineering, Ischia, Italy, 2002.
[53] M.-A. Storey, M. Sanseverino, D. German, D. Damian, J. Michaud, R. Lintern, and J. Chisen, "Adopting GILD: An Integrated Learning and Development Enviornment for Programming," presented at Adoption-Centric Software Engineering (ACSE) at ICSE, Portland, OR, 2003.