APPLICATION OF PATTERNS TO REAL-TIME OBJECT-ORIENTED SOFTWARE DESIGN

by

ROSS ALBERT MCKEGNEY

A thesis submitted to the Department of Computing & Information Science in conformity with the requirements for the degree of Master of Science

Queen’s University,

Kingston, Ontario Canada

July 2000

ABSTRACT

The design and development of real-time software (i.e. software that must ensure timeliness while interacting with an external environment) is more difficult than for most other software. Modeling tools help deal with this complexity, allowing developers to view the system at various levels of abstraction, animate the models in a simulation environment, and even generate the code for a variety of target hardware/RTOS configurations. A natural extension to these tools is to provide support for design patterns (a method of documenting experience in the form of problem/context/solution triples for recurring problems). Such an extension provides yet another layer of abstraction to the models, and makes explicit the application of design patterns.

This thesis will extract from the patterns literature a set of patterns dealing with issues relevant to the design of real-time object-oriented software (in order to demonstrate their variety and quality) - then will propose an extension to Rational Rose-RT to support patterns as an abstraction layer.

ACKOWLEDGMENTS

I would like to acknowledge my supervisor, Dr. Terry Shepard, for his guidance and direction throughout the writing of this thesis.

I also acknowledge the members of my ‘hot-topics’ group at ChiliPLoP 2000: Mary-Lynn Manns (UNCS), Linda Rising (AGCS), Don Olson (AGCS), John Letourneau (Lucent Technologies), and Carol Stimmel (MediaOne Labs) for introducing me to pattern writing workshops.

Also Grant Larsen of Blueprint-Technologies for providing a free copy of Framework Studio, and Francis Bordeleau (Carleton University) for reading and commenting on drafts of this thesis.

Finally, I would like to thank my fiancé Sarah – for all her love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

C h a p t e r   1

CHAPTER 1: INTRODUCTION

This thesis will extract from the patterns literature a set of patterns dealing with issues relevant to the design of real-time object-oriented software (in order to demonstrate their variety and quality) - then will propose an extension to Rational Rose-RT to support patterns as an abstraction layer.

1.1 Motivation

Two recent trends provide the major motivation for this research:

1. Once a very specialized domain focused on hardware and low-level software programming, real-time software development is increasingly part of the mainstream, with well-established RT-specific software development tools (e.g. Rational Rose-RT [Rat00c], Objectime Developer 5.2 [Obj00], i-Logix Rhapsody [iLo00a] and Statemate Magnum [iLo00b], and Telelogic Tao [Tel00]) based on the object-oriented methodology.
2. The 'pattern' concept originated by Christopher Alexander (a building architect [AIS77, Ale79, AS+75]) in the 1970s has become very popular among computer scientists as a means of documenting recurring software design problem/context/solution triples.

Real-time developers often resort to hardware or low-level language programming because of efficiency requirements imposed by the target system's limited memory and processing power. As the cost of hardware decreases, developers must consider the tradeoff between hardware costs and software development costs, and decide whether to focus on optimizing hardware usage, or to apply software abstraction to reduce the costs associated with the software. Object-oriented programming languages (e.g. Ada95 [ISO95] and C++ [ISO98]), and design notations (e.g. ROOM [SGW94], UML [BRJ99]) make software easier to develop and understand - thereby decreasing initial software costs while improving maintainability and reusability. Real-time Object-oriented (RTOO) modeling tools (e.g. Rational Rose-RT [Rat00c], Objectime Developer 5.2 [Obj00], i-Logix

Rhapsody [iLo00a] and Statemate Magnum [iLo00b], and Telelogic Tao [Tel00]) allow developers to focus on software design by automating the generation of RTOO source code from models.

Unfortunately, object-oriented design is difficult, and significant experience is required to become proficient. The pattern form has recently become popular among OO designers as a means of capturing experience in terms of solutions to recurring design problems. A wealth of design experience documented as ‘patterns’ currently exists, describing many ways of dealing with the complex issues faced in the development of all types of object-oriented software – including RTOO software.

1.2 Problem

In general, the research problem addressed by this thesis can be described as follows:

> “Expertise distinguishes a novice from an expert, and it is difficult for experts to convey their expertise to novices. Capturing expertise is one challenge, communicating it is another, and assimilating it is yet another.” [BF+96]

This statement is particularly true for object-oriented design, and for the design of real-time software. Fortunately, there currently exists a method of documenting design experience (design patterns), and for transforming designs into source code (RTOO modeling tools). Thus, the specific research problem becomes: “How can we facilitate capturing, communicating and assimilating design patterns in the context of RTOO structure and behaviour models?”

1.3 Objectives

The objectives of this thesis are as follows:

- Provide thorough background for design patterns and RTOO modeling.
- Discuss costs and benefits of pattern use, based on available experience reports from the RT software domain.
- Summarize and classify the existing design patterns for RTOO software.
- Propose an extension to the Rose-RT toolset to support patterns as an abstraction layer.

1.4 Organization of Contributions

Background information on patterns (Chapter 2) and real-time object-oriented modeling (Chapter 3) put the research in context. In Chapter 4, the validity of the proposed research is discussed; an overview of existing case studies and experience reports from the RTOO domain demonstrate that design patterns are applicable to the domain, and that tool support is necessary to deal with the quantity and disorganization of the patterns literature. Next, a critical survey of design patterns for RTOO software (Chapter 5) describes the varied and extensive nature of the existing design and experience documentation available through patterns. Chapter 6 presents an overview of the variety of design patterns tools; followed by a description of the extensions necessary to make Rational Rose-RT support design patterns. Summary and conclusions are presented in Chapter 7.

C h a p t e r 2

CHAPTER 2: PATTERNS

One of the most interesting aspects of design patterns is that they are nothing new; instead, they describe proven solutions to recurring problems based on actual experience. Ralph Johnson describes the movement philosophically:

> "One of the distinguishing characteristics of computer people is the tendency to go 'meta' at the slightest provocation. Instead of writing programs, we want to invent programming languages. Instead of inventing programming languages, we want to create systems for specifying programming languages. There are many good reasons for this tendency, since a good theory makes it a lot easier to solve particular instances of the problem. But if you try to build a theory without having enough experience in the problem, you are unlikely to find a good solution. Moreover, much of the information in a design is not derived from first principles, but obtained by experience." [Joh94, p. 50]

This chapter presents first the philosophical origins of the 'pattern' idea, followed by a discussion of the application of patterns to software.

2.1 Pattern Origins – Christopher Alexander

The Pattern entity was originally defined by Christopher Alexander (a building architect [AS+75, AIS77, Ale79]) in the 1970s:

> "Each pattern describes a problem that 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." [AIS77, p. *x*]

Alexander sought to capture (as patterns) that which makes structures and spaces comfortable and beautiful – what he called the 'Quality Without A Name' (QWAN). He believed that given a set of patterns, individuals could design for themselves spaces that met their needs – functionally and aesthetically.

To illustrate Alexander's patterns, the full text of the Window Place pattern from A Pattern Language [AIS77] has been included in Appendix A.

2.1.1 Pattern Languages

Notice in the example pattern from Appendix A that Alexander explicitly refers to other patterns that could be applied either prior to, or immediately after the current pattern. This is because in Alexander's view patterns do not stand on their own; instead, they belong to cohesive collections called 'pattern languages'. The patterns in a pattern language each contribute to the resolution of forces in a given problem space, and are linked together in such a way that a designer can work through the language from coarser to finer grain patterns.

At the beginning of each pattern description, Alexander refers to the patterns that come above in the language and in which the pattern can be embedded. At the end of each pattern description, references are supplied to smaller patterns that help completing the pattern. The systematic application of patterns leads to something that Alexander calls *compression* (harmonizing multiple patterns in a single design – e.g. using the Window Place pattern in conjunction with the Light On Two Sides Of Every Room pattern). The notion of compression is in fact a very significant concept – one that makes patterns very powerful, while at the same time making the automation of their application difficult.

2.2 Software Patterns

In the decade that followed the publication of Alexander's trilogy of pattern books, object-oriented software designers began examining and applying his ideas. Most notable of the early efforts were Kent Beck and Ward Cunningham, who developed a simple five pattern language [BC87] for user interface design in SmallTalk; the patterns – Window Per Task, Few Panes, Standard Panes, Nouns and Verbs, and Short Menus – helped novice designers take advantage of Smalltalk's strengths and avoid its weaknesses. The benefits of the Alexandrian pattern form were beginning to be recognized in the object-oriented community, with sessions devoted to patterns at OO conferences becoming more frequent.

In the early 1990s, a variety of patterns related research was being conducted in parallel: Erich Gamma was working on *design patterns* (attempting to capture recurring design structures in the ET++ [WGM88] application framework) as part of his Ph.D. research; James Coplien was developing a set of C++ *idioms* [Cop92] – essentially low-level, or language specific, patterns; the Hillside group [Hil00] was formed (including Cunningham, Beck, Desmond DeSouza, Norm Kerth, Doug Lea, Wolfgang Pree, and others) and coordinated the first annual Pattern Languages of Programming (PLoP) conference in 1994; finally, Gamma, along with Richard Helm, Ralph Johnson, and John Vlissides (thereafter known as the Gang of Four, or GoF) collaborated to write the now famous book Design Patterns: Elements of Reusable Object-oriented Software [GH+95]. Certainly the most significant of these early efforts was the GoF book, which has been credited with the current popularity of the movement.

2.2.1 Design Patterns

The GoF define their design pattern as:

> "A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. The design pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities." [GH+95, p. 3-4]

Philosophically, the GoF patterns borrowed significantly from Alexander: context, problem and solution are explicitly stated; the patterns are carefully categorized, by Purpose (creational, structural, or behavioural) and Scope (applies to objects, classes, or both); and the inter-relationships are clearly identified. However, the GoF used a much more formal pattern template than had Alexander, allowing the reader to clearly see the different components that made up the pattern. This template follows:

“Pattern Name and Classification: The pattern’s name conveys the essence of the pattern succinctly. A good name is vital, because it will become part of your design vocabulary. The pattern’s classification reflects the scheme introduced above.

Intent: A short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

Also Known As: Other well-known names for the pattern, if any.

Motivation: A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help you understand the more abstract description of the pattern that follows.

Applicability: What are the situations in which the design pattern can be applied? What are examples of poor designs that the patterns can address? How can you recognize these situations?

Structure: A graphical representation of the classes in the pattern using a notation based on the Object Modeling Technique (OMT) [RB+91]. We also use interaction diagrams [JC+92, Boo94] to illustrate sequences of requests and collaborations between objects.

Participants: The classes and/or objects participating in the design pattern and their responsibilities.

Collaborations: How the participants collaborate to carry out their responsibilities.

Consequences: How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What aspect of system structure does it let you vary independently?

Implementation: What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues?

Sample Code: Code fragments that illustrate how you might implement the pattern in C++ [Str91] or Smalltalk [Win97].

Known Uses: Examples of the pattern found in real systems. We include at least two examples from different domains.

Related Patterns: What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?” [GH+95, pp. 6-7]

To illustrate the GoF design patterns, the full description of Decorator is provided in Appendix B. This pattern is typical of the GoF catalog; its description consists of diagrams and text describing in detail the problem, potential solutions, tradeoffs between solutions, and related patterns.

2.2.2 Other Pattern Types

Since the mid 1990s, the annual PLoP conference has expanded to Europe (European Pattern Language of Programming conference), to Arizona (ChiliPLoP) and to Australia (KoalaPLoP). These conferences provide a forum for pattern discussion and facilitate the iterative pattern development process. By pairing the pattern writer with a mentor (often referred to as a 'shepherd') from the pattern community, the pattern is refined and integrated with the existing patterns. Many of the best patterns arising from these conferences have been published in a series of books [CS95,VCK96, MRB97, HFR99]. While design patterns (particularly object-oriented design patterns) remain the most popular of the software patterns, researchers and practitioners have proposed a variety of other types. These include: Organizational Patterns, Analysis Patterns, Process Patterns, Idioms, and Anti-Patterns.

2.2.1 Organizational Patterns

Organizational patterns [Cop00], promoted chiefly by James Coplien of the Systems and Software Research Center at Bell Laboratories [Sys00], describe problem/context/solution triples for dealing with organizational issues such as team structure and collaborations. Examples of Coplien's organizational patterns include Self-selecting Team and Apprentice.

2.2.2 Analysis Patterns

Analysis patterns have been proposed by a variety of authors, principally Martin Fowler [Fow97, Fow99a], Coad *et. al.* [CNM95], and David Hay [Hay96]. Each author approaches analysis patterns from a slightly different perspective; the common thread is that they all describe models whose semantics make them applicable to very specific domains or applications. For example, Fowler's patterns use UML to describe recurring analysis models for object-oriented systems, whereas Hay's patterns focus on modeling the system data.

2.2.3 Process Patterns

Software process patterns [Amb99, Amb98] can be viewed either as a pattern type, or as a subtype of organizational patterns. Ambler uses these patterns to describe his Object-oriented Software Process (OOSP), which details how to deal with the following issues (in the context of a medium-large scale organizations actively using object-oriented technology): “successfully deliver large applications using object technology; develop applications that are truly easy to maintain and enhance; manage these projects; and ensure that your development efforts are of high quality” [Amb00].

The software process patterns are categorized as: *Task process patterns,* the detailed steps to perform a specific task, such as the Technical Review and Reuse First process patterns; *Stage process patterns*, depicting the steps, which are often performed iteratively, of a single project stage. A stage process pattern is presented for each project stage such as the Program and Rework stages; *Phase process patterns*, the interactions between the stage process patterns for a single project phase, such as the Initiate and Delivery phases.

2.2.4 Language Specific Patterns (Idioms)

Language-specific patterns, also known as Idioms, have been created for C++ [Cop92] and Smalltalk [Bec96].

2.2.5 Anti-Patterns

Anti-patterns document solutions to commonly used, yet misguided, attempts at resolving software problems. Thus, they focus on *refactoring* existing software. Brown *et. al.*, in their text <u>AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis</u> [BM+98] have identified the relationship between design patterns and AntiPatterns:

> “The essence of an AntiPattern is two solutions, instead of a problem and a solution for ordinary design patterns. The first solution is *problematic*. It is a commonly occurring solution that generates overwhelmingly negative consequences. The second solution is called the *refactored solution*. The refactored solution is a commonly occurring method in which the AntiPattern can be resolved and reengineered into a more beneficial form.” [BM+98, pp. 16]

Like patterns, anti-patterns can be applied to virtually any conceivable aspect of the software development process. Brown *et. al.* identify three perspectives from which anti-patterns may be particularly relevant: Software Development AntiPatterns comprise technical problems and solutions encountered by programmers (e.g. Spaghetti Code, Cut-and-Paste Programming); Software Architecture AntiPatterns identify and resolve common problems in how systems are structured (e.g. Stovepipe System, Reinvent the Wheel); and Software Project Management AntiPatterns address common problems in software processes and development organizations (e.g. Death by Planning, Blowhard Jamboree).

2.3 Related Research

Presented in this section are two research areas closely related to software patterns, frameworks and refactoring.

2.3.1 Frameworks

Frameworks provide generic and extendible architectures for a family of applications. Knowing where and how to extend a framework can be very complex; this is exactly the type of problem/solution pair to which patterns are suited. Frameworks and patterns are thus highly synergistic: a pattern can be used to describe a framework, and a framework can be written as a concrete implementation of a pattern. The typical structure of a framework is presented diagrammatically below:

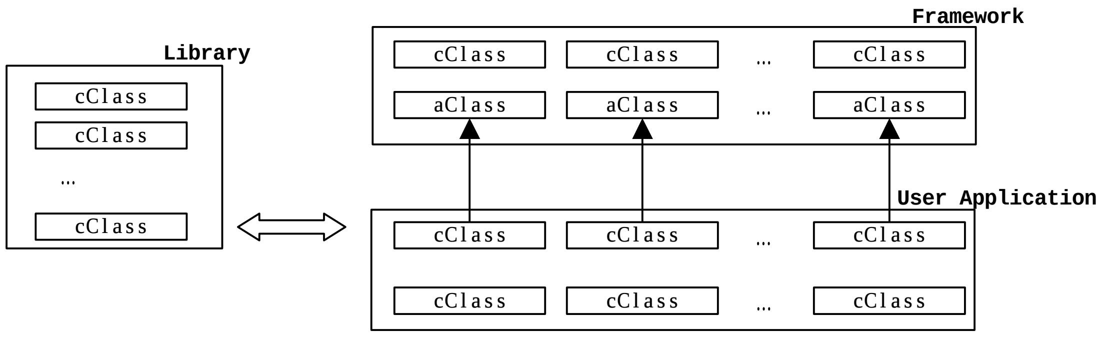


Figure 2.1: Structure of a typical system derived from a Framework

This diagram has three components: the framework consists of *concrete* classes (denoted 'cClass' in the diagram) and *abstract* classes (denoted 'aClass' in the diagram); the user application consists of concrete classes that extend the abstract classes in the Framework, as well as any supporting classes that the application needs; and the library serves as a repository for previously implemented framework extension classes. Designing a system as an extendible framework has the potential to make it reusable by other related applications; this is particularly useful when the shared code is very complex – since the framework can be designed to completely hide this complexity.

2.3.2 Refactoring

Fowler provides the following definition of a *refactoring*:

> "**Refactoring** (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour." [Fow99b, p. 53]

Over time, system architectures tend to degrade – refactoring provides a means of recapturing the initial design, or evolving the design to make it better suited to changing requirements. Because refactorings are always confined to a small scope (to ensure that any bugs introduced will be easy to find), they can be automated [Ref00].

2.4 Summary

Christopher Alexander's pattern concept has proven useful for documenting recurring problems in software design (at the architectural and code levels), analysis models, organizational structures, and other areas of interest to the software community. However, this is a relatively new area in computer science, and suffers from a lack of standards as to how patterns should be described. This is complicated by the philosophical nature of Alexander's work, to which software patterns writers often turn for inspiration and clarification. As the movement matures, definitions of key terms like 'pattern', 'design pattern', etc. will be made concrete, and standardized pattern templates, design notations, and classifications mechanisms should naturally evolve.

Chapter 3

CHAPTER 3: REAL-TIME OBJECT-ORIENTED MODELING

Real-time systems must not only be functionally correct, but also timely (i.e. return responses within a specified time interval). This additional dimension, coupled with the safety-critical nature of many such systems, makes the design and development of real-time software more difficult than for most other software. Fortunately, modeling can help:

> “Through modeling, we achieve four aims:
> 1. Models help us visualize a system as it is or as we want it to be.
> 2. Models permit us to specify the structure or behaviour of a system.
> 3. Models give us a template that guides us in constructing a system.
> 4. Models document the decisions we have made.” [BRJ99, p. 6]

Models simplify reality; they allow developers to narrow in on the aspects of the system that they want, or to back away and view the context. Using currently available RTOO modeling tools (e.g. Rational Rose-RT [Rat00c], Objectime Developer 5.2 [Obj00], i-Logix Rhapsody [iLo00a] and Statemate Magnum [iLo00b], and Telelogic Tao [Tel00]), models can be given enough detail to express complete real-time systems – and even generate their source code for a variety of target hardware/RTOS configurations.

3.1 Real-time Systems

Buttazzo gives a succinct definition of real-time systems: “computing systems that must react within precise time constraints to events in the environment” [But97]. The two components of this definition (timely response, and interaction with an external environment) conspire to make the design and implementation of such systems particularly difficult. Figure 3.1 shows the relationships between the basic elements of a real-time system; these elements are described in the sections that follow.

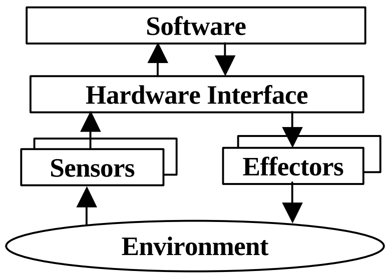


Figure 3.1: Basic Elements of a Real-time System

3.1.1 Environment

By definition, real-time systems must interact with their *external environment* (i.e. the computer hardware and software are ‘embedded’ within a larger system, hence the term ‘embedded system’). For example, the embedded computer controlling an automobile’s ABS braking system must monitor each wheel’s speed, and apply varying levels of brake pressure as required (including pulsing the brakes if they approach lock-up). In this example, the real-time system’s external environment consists of the automobile’s four wheels – it must constantly monitor this environment, and effect change in a timely manner when required. This example also illustrates the concurrent and distributed nature of the external environment; the ABS system must monitor four wheels at the same time – and must potentially respond to four changes occurring simultaneously. The system has no way of knowing what event (or events) will occur next – but it must be prepared to deal with any situation in a timely fashion.

3.1.2 Effectors & Sensors

Real-time systems can collect information about their environment via *sensors*, and can effect change in the environment via *effectors*. Hence, the set of sensors and effectors in the system constrain what environmental changes can be detected, and what actions can be taken to change the environment. Examples of sensors include temperature sensors and altimeters; examples of effectors include pressure release valves and ailerons.

3.1.3 Hardware Interface

The layer between the real-time application software and the effectors and sensors in the external system is known as the *hardware interface*. For example, analog to digital converters are often used to translate sensor signals to a format readable by the RT-software.

3.1.4 Software

Real-time software must perform three functions:

1. Sample sensor data.
2. Calculate new state based on sensor data and previous state information.
3. Effect change in the environment via effectors.

There are two schemes for gathering sensor data: time triggered or event triggered. In a time-triggered system, a real-time clock is used to coordinate activities. The system polls sensors at given time intervals to detect state changes (thus the *latency* – or the interval between the time when the environment changes and the time when the change starts to be serviced - is at most the polling interval). Event-triggered systems wait for the environment to provide notification of changes in state (making the latency dependent on how many events occur simultaneously).

The processing portion of the real-time software often involves processor intensive calculations to determine exactly what the external system is doing, and what action needs to be taken to keep the system running smoothly. The amount of time required to process an input is known as the *service time* for that input. Embedded systems are often required to run on processor and memory limited target hardware; these resources must often be used optimally to keep service time short.

The interval between a change in the environment and effecting corrective action is known as the *reaction time* (equivalent to the sum of the latency and the service time) for that change. Ideally, reaction times will always be less than their pre-defined *deadlines* – however, this may not always be required. Real-time systems can thus be classified as either soft or hard real-time, depending on the required relationship between reaction times and deadlines. Soft real-time systems must respond in a timely fashion; however, some deadlines may be occasionally missed without serious negative consequence (indeed, there may not even be deadlines). Systems of this type include e-commerce applications, where transactions must be processed in a timely fashion – but where intermittent network delays may be tolerated. By contrast, hard real-time systems are those where any missed deadline can lead to severe adverse consequences (in *safety-critical systems*, these consequences can include loss of life). Systems of this type include nuclear power plant control systems, manufacturing control, and military applications.

3.2 Object Orientation in Real-time Software

Despite its popularity within the software development community at large, object-orientation has been slow to catch on in the real-time domain. There appear to be three fundamental reasons for this:

Perceived shortcomings: Real-time software developers have traditionally tried to get as close to the underlying hardware as possible (despite the additional development costs and resulting loss of portability). Consequently, the level of abstraction imposed by OO software constructs is perceived as being too inefficient.

Resistance to change: Because RT software developers often have the added burden of being potentially responsible for loss of life if their software malfunctions, they are typically very resistant to change. Consequently, OO technology has been slow to catch on in hard real-time applications.

Programmer vs. Software Engineer Mindset: Many RT domain specialists are doing RT coding, and not paying attention to software engineering issues.

Despite these factors, object-orientation is increasingly being used for real-time software. The benefits, including increased portability, reusability, and maintainability, mean that long-term development costs of OO software should be much lower than for assembly or procedural code. For many systems, this means that it may be cost-effective to purchase the processor power required to handle the slightly less efficient software. Additionally, there are ways of alleviating some of the overhead associated with object-orientation:

Use a profiler to determine where the system should be optimized: A profiler can be used to determine which methods are used most heavily and changes can be made to improve performance. In most cases (by Pareto's Law, also known as the 80/20 rule) programs spend a majority of their running time in a small portion of the code, so there is often an opportunity for dramatic performance increase for a low cost and with minimal disruption to design integrity.

Transform OO code at compile/link time: In theory, OO overhead can be taken out by using a pre-compiler [FP88]. Such a pre-compiler could transform OO code into structured code - then optimize it. The downside of this technique is the difficulty associated with recovering the system structure for debugging purposes.

Use a subset of an OO programming language: Subsets of both C++ [Emb00] and Ada95 [Pra00a] have been proposed – which exclude the features most responsible for boosting memory requirements, reducing efficiency, and making verification complex.

3.3 Real-time Object-oriented Programming Languages

Two of the most widely used real-time object-oriented programming languages are: Ada95 [ISO95] (an extension of Ada83 [ISO87]) and C++ [ISO98] (a variant of the widely used C [ISO99] language). For soft real-time applications, non-RT programming languages have also been used (including Java [Sun95], which has an RT Java specification under development [RTJ00]).

3.3.1 Ada 95

Ada95 [ISO95] emerged from the Ada83 [ISO87] standard; it consists of the core language, and six annexes (Systems Programming, Real-time Systems, Distributed Systems, Information Systems, Numeric Algorithms, Safety and Security). Ada83 already offered encapsulation in the form of packages, but Ada95 added additional object-oriented features (including polymorphism and inheritance). Programs written in Ada95 are composed of one or more ‘program units’, which can be: subprograms (which define executable algorithms), packages (which define collections of entities), task units (which define concurrent computations), protected units (which define operations for the coordinated sharing of data between tasks), or generic units (which define parameterized forms of packages and subprograms). Each program unit normally consists of two parts: a specification, containing the information that must be visible to other units, and a body, containing the implementation details, which need not be visible to other units. Most program units can be compiled separately.

Real-time facilities supported by Ada95 include built-in support for concurrency (tasks), synchronization and communication (shared variables, protected types, and rendezvous), assigning priorities to tasks, various dispatching policies (including FIFO queuing and priority queuing, but also allowing for user-defined policies), interrupts, and access to the computer’s hardware clock (supporting time intervals in terms of *Durations* in days, hours, minutes, seconds, etc., or sub-millisecond *ticks*).

A subset of Ada95 has been developed (called SPARK [Pra00a]) for use in high integrity applications – where it is essential that programs exhibit logical soundness, formal definition, expressive power, security, verifiability, bounded space and time requirements, and minimal runtime system requirements. The features included in the subset are listed below:

> “It includes Ada constructs regarded as essential for the construction of complex software, such as packages, private types, typed constants, functions with structured values, and the library system. It excludes tasks, exceptions, generic units, access types, use clauses, type aliasing, anonymous types, default values in record declarations, default subprogram parameters, goto statements, and declare statements.” [Pra00a]

SPARK programs must include mandatory formal annotations (including descriptions of variables used in procedures or functions, and any dependency relations that exist between imported and exported variables in procedures), and may include optional formal annotations (pre-conditions, post-conditions, and assertions). The consistency between SPARK code and annotations can then be checked using automated tools (such as the SPARK Examiner [Pra00b]).

3.3.2 C++

C++ evolved from ANSI C [ISO99] and Simula67 [Poo87]; the designers of C++ wanted to add Simula67's object-oriented mechanisms (including classes and inheritance) without compromising the efficiency and power that made ANSI C so popular. Any program written in ANSI C is also a valid C++ program; C++ does nothing to enforce OOP. Many of the features required by real-time developers are not supported by C++ directly; for example, concurrency in C++ relies on external APIs (like POSIX [IEE96]).

A subset of C++ has been proposed for embedded systems, called Embedded C++ (or EC++) [Emb00]. This subset includes most of the object-oriented concepts of C++, but excludes the features that are most responsible for boosting memory requirements and reducing efficiency (i.e. multiple inheritance, virtual base classes, templates, and exceptions). While each of the excluded language features can be very useful, the penalty for including it in the language (even if it is never actually used) can significantly affect the performance and determinism of embedded programs. Compilers (i.e. [Gre00]) designed specifically for EC++ can ensure that excluded features are not used, and can provide better code optimization.

3.3.3 Java

Despite being initially developed by Sun for embedded applications [Sun95], the Java language has been most widely used for Internet applications. Java borrows its syntax from C/C++, but resolves many of the faults of C++: it is not bloated with the entire C language as a subset, pointer arithmetic is prohibited, and garbage collection is automated. Java is an interpreted language; programs are compiled to byte code, then run on a Java Virtual Machine (JVM) layered over the target hardware. This means that a program can be written once then run on any hardware platform that has a JVM.

Java has the potential to be very useful for soft real-time systems: development tools are available and inexpensive; it is an easy language to learn and use; and the "write once, run everywhere" promise is very appealing. However, there are issues that must be addressed before the language will see widespread use in this domain: the JVM has a relatively large footprint (the minimal configuration requires 256-512k of ROM and 256-512k of RAM [Wal98]); and interpreted Java code is slow (interpreted Java code runs between 10 and 40 times slower than equivalent C code [Wal98]). As a result, embedded systems using Java must be configured with more memory and a faster processor. Often these costs cannot be justified by Java's benefits.

There are areas where Java as it is currently specified has great potential: in systems where memory and processor limitations are not an issue; and in embedded systems configured as web servers transmitting applets to their host(s). The latter configuration does not require that the embedded system have a JVM.

There currently exist at least two ways of dealing with Java's efficiency limitations: C code can be embedded in areas where a faster response is required; and a Just-In-Time (JIT) compiler can be used (which compiles the class files just before they are needed). A real-time specification for Java is currently under development [RTJ00]; this specification aims to address the following issues: thread scheduling and dispatching; memory management; synchronization and resource sharing; asynchronous event handling; asynchronous transfer of control; thread termination; and physical memory access.

3.4 RTOO Modeling Notations & Tools

Modeling tools help deal with the complexity inherent in the design and development of real-time software. They allow developers to view the system at various levels of abstraction, and simulate running the software on a variety of target hardware/RTOS configurations.

3.4.1 SDL

SDL (Specification and Description Language) [ITU93] is a formal and standardized graphical language intended for the description of complex, event-driven, real-time systems. System structure is captured as hierarchical partitions (called 'blocks'); behaviour is captured as processes (described using extended finite state machines) running simultaneously, and communicating with each other via signals. Figure 3.2 shows the four levels of a typical SDL diagram:

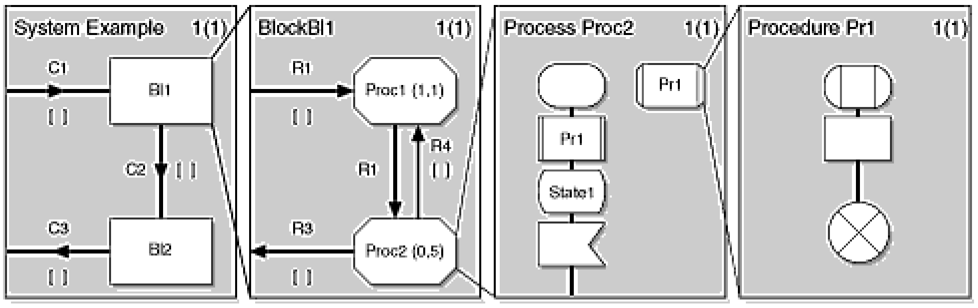


Figure 3.2: SDL Diagram (from [Web00])

The system is partitioned into blocks (Bl1 and Bl2), whose behaviour is implemented by processes (Proc1 and Proc2), which are described using hierarchical finite state machines consisting of states (State1) and procedures (Pr1). Two types of communication are supported: asynchronous signals and synchronous remote procedure calls; communications channels (e.g. C1, C2, C3) and signal routes (e.g. R1, R2, R3, R4) must be explicitly documented.

3.4.1.1 TeleLogic Tau SDL Suite

The Telelogic Tau SDL suite [Tel00] supports the specification and design of systems using SDL [ITU93] notation. It consists of a graphical editor and syntax analyzer, a simulator and validator and several optimized code generators (for C/C++ or CHILL [CHI00] on a variety of targets). It also integrates with the Telelogic Tau UML Suite [Tel00] (for requirements capture and use case description), and with the Telelogic Tau TTCN Suite [Tel00] for testing.

3.4.2 ROOM

Models in Real-time Object-oriented Modeling (ROOM) format are composed of elements of the following types: Actors (classes that define independent, concurrent objects); Protocols (classes that define the messages to be used for inter-actor and actor to runtime system communication); Ports (actor interfaces, used to accept or relay messages); and Data (additional classes used to define the data stored by actors or carried in messages). ROOMCharts (essentially an extended state machine notation) and structure diagrams are used to describe the behaviour and structure of the system. Figure 3.3 shows a simple ROOM diagram (Structure Diagram and ROOMChart) for a Dyeing Run Controller Actor [from SGW94]:

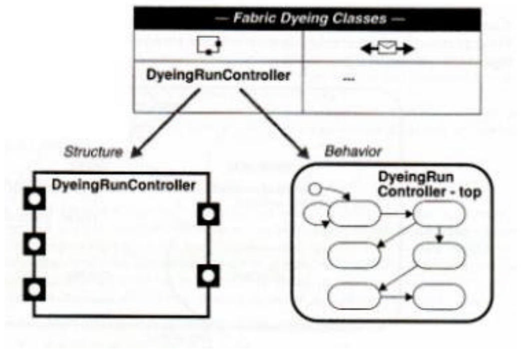


Figure 3.3: ROOM Diagram

The Structure diagram shows that the actor has five ports (in this case they are endPorts – meaning that DyeingRunController can receive messages on them) and no internal actors. The ROOMChart shows the top state and six legal substates for the actor; given more detail it would be possible to examine each state transition to see the triggers, and to examine states and transitions to see the associated code.

3.4.2.1 Objectime Developer 5.2

Objectime Developer 5.2 [Obj00] is a software development tool that enables software developers to generate complete applications from ROOM models. Developer 5.2 also supports: Message Sequence Charts (MSCs) [ITU96], for expressing the intended behaviour of a system and for automatically verifying this behaviour at run-time; model level debugging, including state-machine animation and breakpoints; and integration with third party source debuggers, compilers, configuration management and operating systems.

3.4.3 Statecharts

Statecharts [Har87] were originally developed by David Harel as an extension to state diagrams (shown in Figure 3.4), for expressing the reactive behaviour of systems.

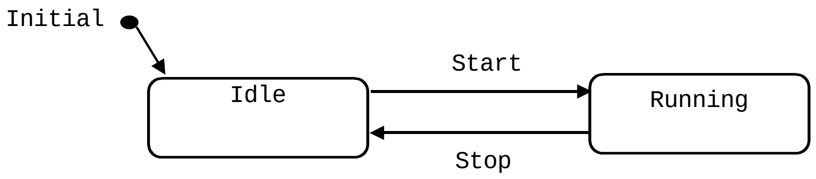


Figure 3.4: A Simple State Diagram

The state diagram in Figure 3.4 shows a state machine with three *states* (Initial, 'Idle', and 'Running'), and three *transitions* (initial, 'Start', and 'Stop'). When the object is in one of the states, it is performing some *activity* (processing) and waiting for some *event* (occurrence of a stimulus that can trigger a state transition). When the right combination of events and conditions occurs, flow of control follows a transition from one state to the next.

The base notation for Harel's statechart is the same as for a state diagram (states, transitions, events, conditions, etc.), but is extended with AND/OR decompositions, inter-level transitions, and a broadcast mechanism for communicating between concurrent components. The statechart in Figure 3.5 shows many of these features:

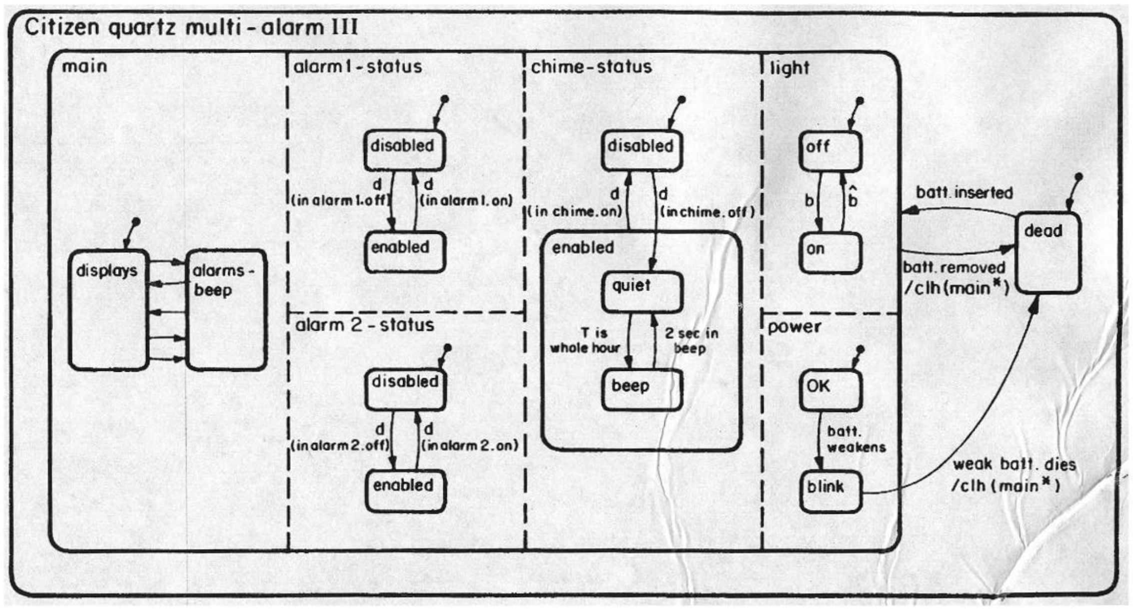


Figure 3.5: Statechart for a multi-function watch (from [Har87])

States can encapsulate other states, representing abstractions for the state diagram encapsulated and allowing the description of common properties. Typically, only one substate can be active at a time (XOR decomposition); however, a dashed line can be used to represent orthogonality, meaning that for each of the subsystems one of the states is active (AND decomposition). Finally, 'H-entrances' substates can be defined, that keep a history of the most recently active substate, so that when flow of control returns to the state, it will be redirected to that substate.

3.4.3.1 i-Logix Statemate MAGNUM

The core of the Statemate MAGNUM [iLo00b] modeling language is the statechart (used to specify reactive behaviour of objects); system structure is modeled using Activity-charts (flow of data between classes) and Module-charts (physical structure). The i-Logix Statemate MAGUM toolset [iLo00b] is based on this language – and allows developers to graphically model, simulate, analyze, verify, and prototype the functionality and behaviour of complex embedded systems.

The major features of the Statemate MAGNUM toolset are: (1) Visual requirements validation using scenarios; (2) Automatic generation of Ada, C, VHDL, or Verilog code from models; (3) Visual animation of the running model; (4) Rapid prototype development; and (5) Complete mathematical specification of the design is generated as the system models are developed, that can be checked against the requirements specification.

3.4.4 UML

The Unified Modelling Language (UML) [BRJ99] was developed by combining features from the following methods: Booch, Jacobson's OOSE (Object-oriented Software Engineering), Rumbaugh's OMT (Object Modeling Technology), Fusion, Schlaer-Mellor, and Coad-Yourdon. The result is a modeling language with Structural diagrams (Class diagram, Object diagram, Component diagram, and Deployment diagram) and Behavioural diagrams (Use case diagram, Sequence diagram, Collaboration diagram, Statechart diagram, Activity diagram). These nine diagram types provide tremendous flexibility, and can be used throughout the software development process.

There have been at least two proposals for real-time extensions to UML: the first [SR98] by Bran Selic and Jim Rumbaugh - the second [Dou97] by Bruce Douglass.

3.4.4.1 Rational Rose Realtime 6.1

Rational Rose Realtime [Rat00c] extends the Rational Rose [Rat00b] visual modeling tool with model execution and code generation capabilities from Objectime Limited [Obj00]. Models are built using active objects (called capsules) that interact with each other through signal-based boundary objects called ‘ports’. Each capsule has an associated state machine (which specifies its functionality); complex capsules can also contain sub-capsules. Complete C or C++ based executables can be generated directly from the UML models – for a variety of hardware/RTOS targets.

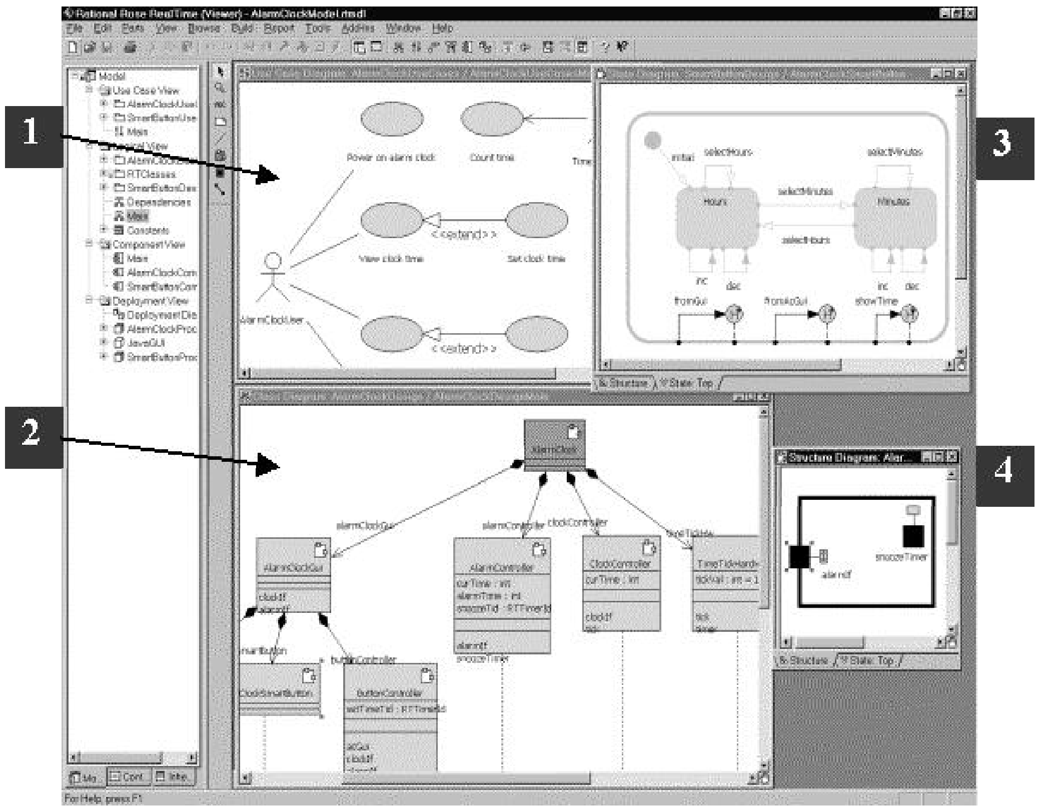


Figure 3.6: Rational Rose-RT 6.1 Screen Capture

Figure 3.6 shows how a system can be specified in Rose-RT: (1) system specifications are described as use case diagrams; (2) class diagrams are used to describe classes and their associations; (3) hierarchical state machines are used to capture the behaviour of capsule classes; and (4) capsule structure is captured using structure diagrams. Other supported UML diagrams include collaboration diagrams, sequence diagrams, component diagrams, and deployment diagrams.

3.4.4.2 i-Logix Rhapsody

i-Logix Rhapsody [iLo00a] is a UML-compliant visual programming environment. Functional requirements are captured as use cases, and design as UML models (Class diagrams capture static structure; Sequence and Collaboration diagrams capture interactions among objects; and Statecharts capture reactive behaviour of classes). Models can be animated; translation from models to C, C++ or Java code is completely automated.

3.5 Summary

This chapter has discussed the wide range of OO programming languages (e.g. Ada95 [ISO95], C++ [ISO98], and Java [Sun95]), and design notations (e.g. ROOM [SGW94], and UML [BRJ99]) that can be used for the design and development of real-time software. These technologies have the potential to assist in the design of real-time software that is more portable, reusable, and maintainable.

Object-oriented technologies are not appropriate for every software project, in part because the abstractions introduce memory and processing overhead. However, these extra resource needs can be minimized with good design and by using optimization techniques. In particular, using an OO programming subset (e.g. SPARK Ada [Pra00a] or Embedded C++ [Emb00]) ensures that the most inefficient and unpredictable language features will not be used, and helps the compiler with its optimization.

For those systems where OO is applicable, RTOO modeling tools (e.g. Rational Rose-RT [Rat00c], Objectime Developer 5.2 [Obj00], i-Logix Rhapsody [iLo00a] and Statemate Magnum [iLo00b], and Telelogic Tao [Tel00]) can aid the development process by helping to abstract away low-level details – allowing users to focus on design. Developers use the tool to create ‘executable models’, typically including some form of hierarchical state machine to specify system behaviour, and structure diagrams that allow individual FSMs to relate to one another, support scaling up, decomposition and separation of concerns. These models can then be animated, simulated on the host, and ported to a variety of target hardware/RTOS configurations.

Chapter 4

CHAPTER 4: EXPERIENCES WITH RTOO DESIGN PATTERNS

Speaking at OOPSLA ’99, Alexander asked the following question about the value of patterns in software development:

> “I understand that the software patterns, insofar as they refer to objects and programs, and so on, can make a program better. That isn't the same thing, because in that sentence ‘better’ could mean merely technically efficient, not actually ‘good.’ Again, if I'm translating from my experience, I would ask that the use of pattern language in software has the tendency to make the program or the thing that is being created morally profound - actually has the capacity to play a more significant role in human life. A deeper role in human life. Will it actually make human life better as a result of its injection into a software system?” [Ale99]

Software pattern writers tend to be more pragmatic, and enjoy the superficial benefits of the pattern form. This chapter will discuss some of these perceived benefits; serving both as background for the design patterns presented in the next chapter, and as validation of the proposed research into tool support for RTOO design patterns.

4.1 General Design Patterns Experiences

This chapter is organized in 6 parts: Industrial experience with design patterns; Patterns in practice; Pros and cons of adopting and applying design patterns; Using design patterns to develop reusable OO communication software; Using design patterns against a moving target; and How to preserve the benefits of design patterns. The experiences discussed are not all RTOO specific, but they discuss some general costs and benefits of using patterns.

4.1.1 Industrial Experience with Design Patterns

Kent Beck (First Class Software), James O. Coplien (AT&T), Ron Crocker (Motorola, Inc.), Lutz Dominick (Siemens, AG), Gerard Meszaros (Bell Northern Research), Frances Paulisch (Siemens, AG), and John Vlissides (IBM Research) collaborated to present their experiences with design patterns in [BC+96]. This is a widely cited article on the costs and benefits of pattern integration by some of the major proponents of the approach.

Kent Beck's contribution revolves around his work on a set of Smalltalk Best Practice Patterns (SBPP) [Bec96]; these are language-specific patterns meant to describe the 'habits' of expert Smalltalk programmers. Even in a fairly rough form, the SBPP was used successfully in two projects: the first with a team of five developers at Hewitt Associates, the second with 25-30 developers at Orient Overseas Container Limited (OOCL). Beck suggests that the positive effects of using his patterns include better communication within teams, improved code quality and increased quantity of produced code. This is not to say that patterns are a silver bullet [Bro87]: "Patterns solve a limited (but critically important) set of communication problems with team development, and make individuals more productive. They cannot substitute for effective project management." [BC+96, p. 105]

James Coplien describes three of the ways in which patterns have been applied at AT&T: Design patterns have been used to describe the architecture of high-availability fault-tolerant communications systems; organizational patterns have been used to document effective software development processes and organization; idioms (language-specific patterns) have been used to capture low-level language-specific programming techniques. Observed benefits of pattern use at AT&T include: Pattern mining (extracting patterns from existing code) has proven a useful method for documenting design architectures; mined patterns can then be applied to new projects, facilitating the design of new architectures; the resource base provided by patterns facilitates the analysis of new project requirements, which helps to determine feasibility and assign deadlines; and Process patterns have been used to assess the 'health' of development groups.

Ron Crocker describes experiences by Motorola's Cellular Infrastructure Group (CIG) with design patterns as a means of achieving 'large-grain' reuse. Previous reuse efforts within the group had been relatively unsuccessful, due to strong coupling of OO artifacts within products, and near-sightedness. The reuse effort was further complicated by communication problems between developers and software architects (architects used a different terminology, and were not available for reference). The solution: "…use design patterns to capture problem-domain-specific entities in an implementation-independent way for sharing across projects (and products)" [BC+96, p. 107].

Gerard Meszaros describes two types of patterns that were in use at Bell Northern Research (BNR): Process/method patterns, which describe design methodology and patterns of behaviour which may lead to good architectures; and technical patterns, used for defining the technical aspects of communications system architecture. Meszaros reflects on the impact of patterns at BNR as follows: "…communication between people with a 'shared space' of patterns is quicker, more complete, and less likely to be misunderstood." [BC+96, p. 109] Further, he observed three personality types (with respect to patterns): "…those who see patterns everywhere and can describe them, those who can recognize patterns but can not describe them easily, and those who are oblivious to the patterns surrounding them." [BC+96, p. 109] Meszaros suggests that the majority of individuals are of the second type; only a very small percentage of developers belong to one of the two extremes. These observations suggest that most developers can successfully apply patterns, and once they do, a significant improvement in communication (with many resulting benefits) will follow.

Francis Paulisch & Lutz Dominick describe the results of a project for Siemens, AG investigating the effectiveness of applying patterns to technology oriented applications like the process control of steel mills. Their experiment involved three steps: First, a team consisting of two domain experts and two patterns experts met to develop an initial set of patterns; Second, these patterns were made available online – allowing all developers to access the patterns, and providing a forum for suggesting refinements and adding new patterns; third, pattern usage was analyzed. The creation of the initial set of patterns took approximately three meetings; yielding a set of patterns that met the criteria that they correctly represent the problem solution pair, and that they be a useful representation of knowledge demanded by their projects. These patterns were made available as an online catalog, which facilitated searching and made it possible to represent the set of patterns graphically in several formats. Initial analysis suggested that the patterns were well received, but that tool support became necessary for more than 30 patterns.

John Vlissides' consulting experiences on software projects with a variety of companies lead him to the realization that most face similar problems. Two 'irritants' repeatedly surfaced: The first irritant was the difficulty associated with getting a proper picture of the system's architecture. Faced with one spaghetti class diagram after another, Vlissides was forced to resort to 'interrogation' to get an understanding of the relationships between components in the system. A related irritant was the lack of documentation for design changes. Without explanations for why a design change (large or small) was made, Vlissides repeatedly found himself in the position of blindly reverse engineering design choices. Design patterns inherently address these two fundamental problems. They provide a mechanism for communicating design, while explicitly documenting its rationale.

The six experience reports presented in this paper describe applications of patterns in a variety of contexts. Despite a lack of quantitative data on the benefits of patterns, experiences from members of the community suggest that their impact can be significant – and positive. The perspectives presented are those of patterns advocates; however, their discussion on the benefits of patterns is balanced with common-sense advice on how to properly use patterns, and how to avoid potential pitfalls. Particularly interesting are the ways in which the various organizations have dealt with the inherent complexity and variety of patterns. Each has developed internal mechanisms for classifying patterns, and many have developed tools for making pattern access easier.

4.1.2 Patterns in Practice

Richard Helm (DMR Group Inc.) presented "Patterns in Practice" [Hel95] at OOPSLA ‘95. Data for this report comes from two years experience with CEE - a cellular network management and engineering system developed by DMR in collaboration with Ericsson Communications. Helm makes several observations:

- Patterns are applicable throughout the system and across domains.
- Patterns cover a large portion of a system’s design and architecture.
- Patterns can be used to describe relationships/couplings between sub-systems.
- The proportion of pattern specific code is very small.
- Designs based on patterns seem more robust to requirements changes, are more reusable, and lessen the need for class refactoring and re-design.
- Patterns provide a shared vocabulary.
- Patterns provide a ready resource for less experienced team members to rapidly produce effective designs.
- There are dangers in naively applying patterns to the design of systems.
- There is a danger in viewing the GoF patterns as “the gospel according to the Gang of Four”.
- OO methodologies and tools need to adapt and take into account patterns during the transition from analysis to design.
- Patterns influence analysis object models.
- Reconciling pattern-based approach with OO methodologies (which say little about reuse) is difficult.
- Knowing when to use patterns is difficult.
- When an appropriate pattern is found, the design tends to fall into place.

Helm describes some of the benefits to be gained from pattern integration – while also exposing some of the potential pitfalls. This balanced discussion brings forward several issues ripe for future research, particularly in the intersection between OO methodology and design patterns.

4.1.3 Pros and Cons of Adopting and Applying Design Patterns

Marshall P. Cline (Paradigm Shift, Inc.) discusses some of the costs and benefits of integrating patterns in "The Pros and Cons of Adopting and Applying Design Patterns in the Real World" [Cli96]. Having successfully employed design patterns for a three-year development project (150 OO/ C++ developers), Cline recognized several practical benefits of patterns:

- Design patterns provide a shared vocabulary between developers, designers, and project management – thereby coordinating the entire process and community.
- The uniform and widespread application of design patterns facilitates passing off code to new developers or a new team. Design decisions and rationale are captured explicitly in the code, rather than staying in the heads of the original team.
- Design pattern use improves code quality. Design tradeoffs are explicitly documented, resulting in architectures that are more robust.
- Design patterns provide means of designing code with ‘hinges’ or ‘hot-spots’ where future extension can be made. Thus, architectures are more adaptable and extendible.
- Design patterns may even provide solutions that will enable seemingly mutually exclusive qualities simultaneously.
- Design degradation over time is less likely with code designed using design patterns. Since the code’s structure is explicitly documented, the probability that a maintenance programmer will make a change that damages the design is decreased.

Cline also recognizes some inhibitors to pattern application:

- Design patterns have been overly hyped; they are not a ‘silver bullet’, and the consequences of their use must be understood (since in most cases their benefits do not come without cost).
- Some design patterns are unnecessarily difficult to learn; patterns with unclear names and/or problem statements can be more of a distraction than a benefit.
- The GoF design pattern categorization mechanism (along purpose and scope axes) does not reflect the breadth of available patterns.

4.1.4 Using Design Patterns to Develop Reusable OO Communication Software

Douglas C. Schmidt (Washington University) documented his experiences with design patterns for large-scale commercial distributed systems (projects at Ericsson, Motorola, and Kodak) in [Sch95b]. A long list of lessons learned is provided – including benefits gained from pattern use, and workarounds for problems encountered. These include:

Rewards should be institutionalized for developing patterns: Schmidt suggests that corporate policies are needed for rewarding the creators of useful patterns. He found that in many cases developers saw their knowledge as a competitive advantage over their peers – and needed some form of incentive (i.e. take the pattern writing into consideration at the next performance review) to encourage them to share their knowledge.

Patterns may lead developers to think they know more about the solution to a problem than they actually do: Patterns by their nature address complex problems in an intuitive fashion – this may lead developers to a false sense of security. In most cases, implementing the structure described in the pattern in an efficient and portable fashion is very difficult.

The focus should be on developing patterns that are strategic to the domain and reusing tactical patterns: Development organizations should focus their pattern writing on the development of domain specific patterns; general purpose (or tactical) patterns should be reused – not reinvented.

Integrating patterns into a software development process is a human-intensive activity: Learning to use patterns effectively cannot be done individually; it is most successful through interactive sessions like pattern reviews or pattern mining exercises.

Implementing patterns efficiently requires careful selection of language features: Design patterns abstract the architecture of the application, leaving the implementation details up to the developer. The Gang of Four used inheritance and polymorphism to describe the structure of their patterns; however, more efficient language features (such as parameterized types) may be used when high performance is required.

Schmidt's comments suggest that developers should not blindly use patterns; patterns are a tool that can facilitate design and improve communication – but they must be used correctly to see these benefits.

4.1.5 Using Design Patterns Against a Moving Target

Kim G. Woodward (DCS Corporation) presented "Heading off Tragedy: Using Design Patterns Against a Moving Target" [Woo96] at the Second World Conference on Integrated Design and Process Technology. She suggests that most designs are not static; instead, they must adapt to changing requirements. Systems must be designed for a 'moving target', and patterns can help.

Woodward begins by describing the characteristics of bad design: they are rigid (it is difficult to make the changes that customers demand); they are fragile (changes made to one part of the application adversely affect other components); and they are not reusable (coupling is so high that desirable components cannot be extracted for reuse). Good designs can then be said to be set in 'mushy concrete' – resilient enough to meet the system's demands, but with the flexibility to change as required. She proposes an eight-step process intended to result in better designs:

"Identify the minimal set of requirements first": Extract from the requirements definition those that are not likely to change.

"Design a flexible framework second": Use the subset of static requirements to build a framework for the application.

"Search for answers in patterns last": Design patterns should be used to assist in choosing implementation methods. Using standard approaches to solving problems helps reuse, and makes the framework easier to understand.

"Encapsulate, Encapsulate, Encapsulate": Components which are likely to change should be encapsulated.

"Write it down": The system's design should be well documented, including: rejected and accepted components of the design, design patterns used, and the rationale for design decisions.

"Use your tools": Tools should be used throughout the development lifecycle – not only during the design phase.

"Keep the design current": The design should be periodically refactored to reflect the true architecture of the system and preventing design degradation over time.

"Force changes to follow a rulebook": A rulebook documenting how and when the design should be changed will help limit changes that damage the original intentions of the design.

Woodward's proposed eight-step design process relies heavily on patterns: assisting the design of an original framework based on initial system requirements; providing mechanisms for decoupling components – thereby facilitating low-level changes; and documenting the purpose and structure of the original design, so that future changes do not degrade the design.

4.1.6 How to Preserve the Benefits of Design Patterns

Ellen Agerbo and Aino Cornilis (University of Aarhus, Denmark) presented "How to preserve the benefits of Design Patterns" [AC98] at OOPSLA '98. The authors recognize three fundamental benefits to be gained from design patterns: They encapsulate experience; they provide a common vocabulary for computer scientists across domain barriers; and they enhance the documentation of software designs. Unfortunately, as the number of patterns increases, these benefits may be lost: "…an overdose of Design Patterns will eliminate two of the three benefits that Design Patterns offer; they will make it too laborious to find and use the encapsulated experience, and they will make the common vocabulary too large to be easily comprehended" [AC98, p. 134].

Agerbo & Cornilis' propose categorizing patterns as: Fundamental Design Patterns (FDP) - language-independent design patterns solving recurring design problems using original ideas; Language Dependent Design Patterns (LDDP) - programming language specific design patterns; or Related Design Patterns (RDP): applications or variations of existing design patterns. Once patterns of different types have been differentiated, FDPs can be studied for their contribution to the literature, RDPs can be scanned to assess the particularities of the pattern application, and LDDPs can be used in a pattern library. As an example, the categorization was applied to the 23 Gang of Four patterns: twelve of the patterns were categorized as FDPs, seven as LDDPs, two as RDPs, and two uncategorized.

4.2 Design Patterns and Frameworks

This section discusses experiences using design patterns with three frameworks; for speech recognition, telecommunications, and vessel control.

4.2.1 Speech Recognition Framework

Savitha Srinivasan (IBM) describes the application of design patterns to an object-oriented framework for speech recognition applications using IBM's ViaVoice speech recognition technology [Sri99]. The goal of the framework was to hide the complexity of the speech recognition technology, thereby facilitating the development of a family of applications that used the technology. The framework can be represented diagrammatically as follows:

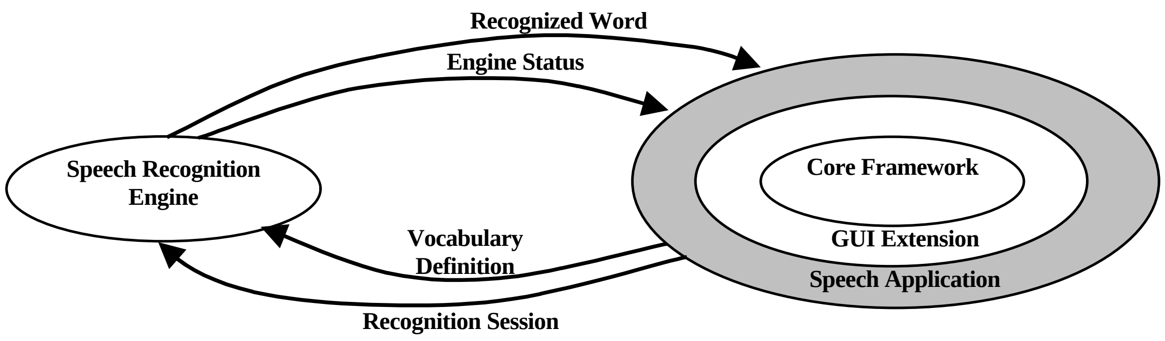


Figure 4.1: Speech Recognition Framework

Figure 4.1 represents the relationship between an application built using the speech recognition framework, and IBM's ViaVoice speech recognition engine. The speech application and its associated GUI are built on top of a core framework, which encapsulates the methods required to modify the current active vocabulary (the set of words that the speech recognition engine is listening for) and to retrieve a recognized word from the speech recognition engine. This framework has been successfully used to build four applications: MedSpeak/Radiology [LV97], MedSpeak/Pathology [LV97], the VRAD (visual rapid application development) environment, and a video cataloging tool with speech recognition functionality. The framework was refined in parallel with application development, such that by the fourth application almost all of the speech functionality was in the core framework (whereas earlier applications had up to 25% of the speech functionality in the GUI extensions).

Design patterns were used extensively in the design of the core framework – particularly in the design of the 'hotspots'. The patterns used were classified as 'Object-oriented' (Adapter, Façade, Observer, and Singleton) and 'Distributed or Concurrent' (Active Object, Asynchronous Completion Token, and Service Configurator). Srinivasan suggests that design patterns made the evolution of the framework easier, and simplified the documentation of framework extensions.

4.2.2 Communication Software Framework

Douglas C. Schmidt (Washington University) and Paul Stephenson (Ericsson, Inc.) discuss the applicability of design patterns to large-scale communications software in [SS95]. They use as a case study the evolution of an object-oriented telecommunications framework from UNIX to Windows NT OS platforms.

The porting of Ericsson's ADAPTIVE Service eXecutive (ASX) framework from UNIX to Windows NT was complicated by the different mechanisms used for event demultiplexing and I/O. Despite not being able to reuse many of the modules in the framework, the underlying architecture (developed using design patterns) could be reused. Two patterns were discussed in detail: Reactor – used to decouple event demultiplexing from event handler dispatching; and Acceptor – for decoupling connection establishment from the resulting service to be performed. Despite differences in the specifics of the required low-level functionality on the two OS platforms, the basic structure of these two patterns could be reused effectively.

Schmidt & Stephenson describe mechanisms for maximizing patterns benefits:

Expectation management: Design patterns are no silver bullet; for patterns to be used effectively, the hype currently associated with them must be dispelled.

Wide-spectrum pattern exemplars: Patterns should provide a variety of sample implementations; ideally, this would be done using a hypertext-enabled browser, so that the user could simply follow links to sample code.

Integrate patterns with object-oriented frameworks: An extension to the solution above would be to reference 'mini-frameworks' that provide concrete and extendible implementations of the pattern. Developers could follow the link to the implementation that interests them, copy the code to the project that they are working on, and extend it.

This case study, and the resulting conclusions, explicitly describes the effects that target hardware/OS can have on software design. For communication software (complex large scale software systems typically ported to multiple hardware/OS platforms) the architectural reuse made possible by correct use of design patterns can be extremely useful.

4.2.3 Vessel Control Systems Framework

Per Dagermo and Jonas Knutsson co-wrote [DK96] as part a report on the DOVER (Development of an Object-oriented Framework for Vessel Control Systems) project. It was expected that having an object-oriented framework would lead to reduced software development cost, shorter software development time, better estimation of development time, and higher quality in future vessel control systems.

Vessel control is an example of a real-time process control system:

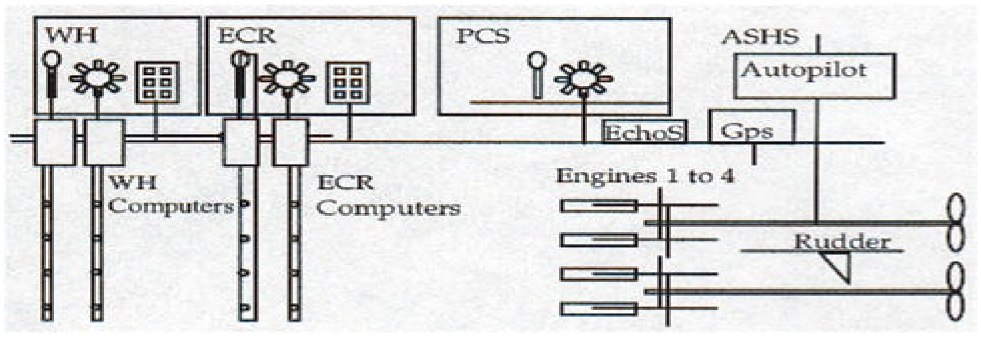


Figure 4.2: Schematic picture of a typical vessel control system

Inputs come from controls in the wheelhouse (WH) and engine control room (ECR), and from external systems such as the autopilot, GPS and echo sounder; processing determines if and where faults are located, and displays meta-data; and actuators maneuver the ship. Redundancy is built-in to ensure that the system continues to function even if a component fails (using exceptions), and timeliness of processing and effecting change is guaranteed (time-triggered mechanism).

The architecture of the framework is not discussed in detail; however, the patterns used in its implementation were described. The Observer pattern was used to update dependent values in the system as inputs change; Singleton was used to ensure that a class had only one globally accessible instance (e.g. to keep a single network manager); Proxy was used to make it transparent that value objects are located on different nodes; and State to change an object's behaviour at runtime depending on current state (e.g. the engine class behaves differently when it is in running state vs. stopped state).

Dagermo & Knutsson list several observations based on their experience with patterns and frameworks:

- Migrating from structured C to object-oriented C++ was more difficult than expected
- Design patterns are best learned through experience
- Framework development requires expert knowledge of OO techniques, and of the specific domain
- Research projects such as DOVER with no short-term profit have lower priority; management support is needed to keep from having developers pulled in and out of the project.

Preliminary analysis of the DOVER framework suggests a significant reduction in code size, and significant increases in reuse and code quality. Design patterns proved beneficial to the framework's design.

4.3 Summary

The experiences presented in this chapter were from developers in many different domains; each had their own conclusions about the benefits of patterns – and suggestions for avoiding pitfalls in their application. Despite a lack of quantitative proof (a difficult but very important area for future research), some general observations can be made based on the consensus of this group of authors:

- Understanding a shared set of patterns can improve communication among team members and within organizations.
- Design patterns are good at documenting design tradeoffs, and framework extensions.
- Many organizations are developing their own classification mechanisms (often with tool support) for dealing with the volume and complexity in the pattern literature.

Documenting experience in patterns form has many benefits; unfortunately, the volume and disorganization of the current literature is a significant impediment to their use. Organizations have independently developed their own mechanisms for dealing with this issue; this is an area where standards could be very beneficial and tool support is necessary.

C h a p t e r   5

CHAPTER 5: SURVEY OF REAL-TIME OBJECT-ORIENTED DESIGN PATTERNS

The design of real-time object-oriented software requires dealing with complex issues, which may include: concurrency, distribution, limited memory and processing power, schedulability, timeliness, availability, scalability, predictability, maintainability, and portability. This chapter presents summaries and discussion of design patterns addressing many of these issues. Timeliness and schedulability issues are notably absent from the collection of patterns, since to the best of our knowledge they are not addressed in the literature.

The patterns were extracted from the design patterns literature, and organized according to a simple classification. The top-level classification is by domain specificity – widely applicable RTOO design patterns are presented first, followed by domain specific RTOO design patterns. The widely applicable RTOO design patterns are sub-classified by primary author; while the domain specific RTOO design patterns are sub-classified by the domain to which they apply.

Following the survey of patterns is an analysis of the collection, with discussion of how they form a pattern language for real-time object-oriented software design. The relationships between the patterns are made explicit, as are the more incomplete areas of the collection.

5.1 Widely Applicable RTOO Design Patterns

This section presents two sets of patterns and four stand-alone patterns; these are more specialized than the GoF design patterns, but are generally applicable to real-time software. The first set of patterns comes from Bruce Douglass of i-Logix, the second set of patterns is a pattern language for concurrent, parallel and distributed systems – developed by a variety of authors, principally Doug Schmidt of Washington University.

5.1.1 Douglass' Real-time Design Patterns

Bruce Douglass presents many design patterns for real-time software in [Dou99]. The wide scope of this text makes it a good starting point for this chapter.

At the highest level, Douglass classifies his patterns as Architectural (design patterns) vs. Mechanistic (language specific patterns). The architectural patterns are further classified as: Architectural Support Patterns; Collaboration and Distribution Patterns; Safety and Reliability Patterns; and State Behaviour Patterns.

5.1.1.1 Architectural Support Patterns

Microkernel Architecture Pattern: Systems can be made more portable by separating responsibilities into layers (by *level of abstraction*). Two implementation alternatives are discussed: open architecture subpattern - in which layers can invoke services of any layer below; closed architecture subpattern - in which layers can invoke services of only the layer immediately below.

Six-Tier Microkernel Architecture Pattern: Alternatively, systems can be separated into layers based on subject areas of interest (or domain). The Six-Tier Microkernel Architecture pattern divides systems into the following domains: Application, Distribution, User Interface, Protocol, OS, and Hardware. An open or closed architecture can be used with this pattern.

5.1.1.2 Collaboration and Distribution Patterns

The first three Collaboration and Distribution Patterns (Container-Iterator, Observer, and Proxy) come directly from [GH+95]. The Broker pattern is an extension of Proxy:

Broker Pattern: The Broker pattern is applicable in cases when a proxy could not know the location of objects. An object broker is used to connect object requests with object services.

5.1.1.3 Safety and Reliability Patterns

Watchdog Pattern: A watchdog can be used to monitor the behaviour of objects and provide error correction in case one of them stops responding.

Safety Executive Pattern: A more sophisticated version of the watchdog that provides central monitoring of objects. The Safety executive typically tracks: watchdog timeouts, software error assertions, and faults identified by monitors in the Monitor-Actuator pattern.

5.1.1.4 State Behaviour Patterns

Latch State: Used to guard against certain behavior before the required pre-conditions have been met.

Persistent Latch State: Provide a Latch State pattern that persists in a larger context by remembering the latch sub-state.

Polling State: Periodically perform an action.

Latched Data: Combine Polling and Latch patterns to retain data even if the system isn't ready to process it.

Device Mode State: Model independent device modes of operation

Transaction State: Model different quality of service transaction styles, such as for remote communications.

Component Synchronization State: General synchronization based on a set of ANDed pre-conditions.

Waiting Rendezvous: Implement a waiting rendezvous with the Component Synchronization State pattern.

Timed Rendezvous: Implement a timed rendezvous with the Component Synchronization State pattern.

Balking Rendezvous: Implement a balking rendezvous with the Component Synchronization State pattern.

Thread Barrier State: Implements a barrier for n components or threads.

Event Hierarchy State: Perform a general action on a class of events, as well as a specific action.

Unexpected Event: A specialized sub-pattern of the Event Hierarchy Pattern used when it is a semantic error to ignore an unexpected event.

Random State: Enter a new state using random selection.

Null State: Use the results of a transition action in a guard by inserting a null state.

Watchdog State: Execute a periodic liveness check to ensure the application is continuing properly by generating a stroke event for the watchdog orthogonal state component.

Keyed Watchdog State: A more elaborate watchdog, which checks for a proper sequence of events rather than a single stroke event.

Retriggerable Counter State: Decode pulse-modulated information with a retriggerable counter.

Several of Douglass' patterns were omitted from the classification above, including: Multi-Channel Voting; Homogeneous Redundancy; Diverse Redundancy; Monitor-Actuator; Rendezvous; Semaphore. In most cases UML documents were provided; however, the supporting documentation was not sufficient. For example, the Semaphore pattern consisted of a reference to a later chapter – then was never mentioned further.

This text is useful as a reference manual for real-time software since it discusses such a wide range of topics. However, the patterns are not useful unless they are used in conjunction with an external classification such as was provided above.

5.1.2 Schmidt et. al.'s Patterns for Concurrent, Parallel, and Distributed Systems

Doug Schmidt (Washington University), in collaboration with other researchers at Washington University's Center for Distributed Object Computing [Cen00], has developed several patterns for concurrent, parallel and distributed (CPD) systems. These patterns have been applied to large-scale distributed telecommunications, medical imaging, and financial services projects. Examples of these patterns in practice can be found in the open-source ACE communications framework

[Cen00]. Schmidt divides his patterns into four categories: Concurrency, Event, Initialization, and Synchronization.

5.1.2.1 Concurrency Patterns

These patterns address concurrency issues in multi-threaded systems:

Thread-per-request: The Thread-per-Request pattern [LS96] describes a concurrency technique whereby a new thread is allocated for every incoming request to a server.

Active Object (Thread-per-Object): The Active Object pattern [LS96] facilitates synchronized access to an object. When a request arrives at a server, it is assigned a thread associated with the data object it wishes to access. This simplifies the handling of critical regions – and reduces the probability of having a thread blocked for an extended period while waiting for an object to be released (since the thread will not be assigned until the object is available).

Thread Pool: The Thread Pool pattern [LS96] suggests pre-allocating a pool of threads, which can then be associated to incoming requests. May be used in conjunction with Active Object or Thread-per-Request.

Half-Sync/ Half-Async: The Half-Sync/Half-Async pattern [SC95] specifies a mechanism for improving efficiency of I/O in concurrent systems. Higher-level tasks use synchronous I/O, while lower-level tasks use asynchronous I/O.

Thread Specific Storage: The Thread Specific Storage pattern [SHP97] allows multiple threads to use one logically global (but physically specific) access point to retrieve thread-specific data. Using this pattern results in no locking overhead for each data access, thereby improving performance and simplifying multi-threaded applications.

5.1.2.1.1 Synchronizer

The Synchronizer pattern [Gra97] was developed by Ennio Grasso of the Centro Studi e Laboratori Telecomunicazioni (Torino, Italy) as a specialization of the Active Object pattern [LS96]. The Active Object pattern stipulated that each shared object should have one associated thread – Grasso suggests that this technique limits concurrency.

Synchronizer: complements the thread-per-request pattern with the synchronization model of the thread-per-object (Active Object) pattern to avoid unnecessarily waiting on synchronization constraints. Essentially, the Scheduler selects a request that satisfies synchronization constraints, and assigns it to a thread. This technique allows increased concurrency over Thread-per-object (since multiple requests can concurrently operate on the same object), and reduced probability of having threads suspended until synchronization constraints are met (unlike Thread-per-request, threads are not assigned until synchronization constraints are met). This pattern uses Proxy and Chain of Responsibility from [GH+95].

Transactional Synchronizer: the Synchronizer pattern can easily be extended to handle the two-phased locking policy (growing phase: transaction progressively acquires and retains locks while active; shrinking phase: locks released when the transaction completes) for a transaction service. This functionality can be obtained by a simple modification to the running queue – rather than remove a thread from the running queue when it finishes processing its request, do not remove it until the entire transaction is complete.

5.1.2.2 Event Patterns

The following patterns describe techniques for handling the occurrence of synchronous and asynchronous events in event-driven systems.

Reactor: The Reactor pattern [Sch95d] provides a means of implementing concurrency without using multiple threads. The premise is that the application waits in an event loop until an event occurs, at which time an event handler is allocated and put into a priority queue. At given intervals a dispatcher notifies the event handler at the head of the queue that it can initiate its operation.

Proactor: The Proactor pattern [PH+97] associates with each event handler the handlers for events that are to immediately follow it. Upon event completion, the associated handlers are put in the queue.

Asynchronous Completion Token: The premise of the Asynchronous Completion Token (ACT) pattern [HSP96] is that many RT systems perform long-running operations asynchronously to avoid blocking the processing of other, higher priority, operations. When these asynchronous operations complete, they should notify the calling application – not only that they have finished, but also with some state information. The ACT is used to associate this state information.

5.1.2.3 Initialization Patterns

These patterns document techniques for creation and deletion of objects.

Acceptor-Connector: The Acceptor and Connector patterns [Sch95c] decouple connection establishment and service initialization in a distributed system from the processing performed once a service is initialized. A Connector is used to establish a connection with a remote Acceptor component, and initialize a Service Handler to process data exchanged on the connection.

Service Configurator: The Service Configurator pattern [SJ97] decouples the instantiation of services from the time when they are configured. This allows services to be reconfigured at run-time – particularly useful for reconfiguring communication systems in distributed environments.

Object Lifetime Manager: The Object Lifetime Manager pattern [Sch99a] is complementary to creational patterns like Singleton and Factory Method (from [GH+95]). The pattern defines an Object Lifetime Manager component, which governs the entire lifecycle of managed objects, from creation to ensuring proper destruction.

5.1.2.4 Synchronization Patterns

The synchronization patterns discuss techniques for improving the efficient synchronization of objects in concurrent systems:

Double Checked Locking: The Double Checked Locking pattern [SH96] discusses a technique for reducing synchronization overhead whenever critical sections need to acquire locks just once. A lightweight conditional test is made before entering the method where locking will be necessary.

Strategized Locking: The Strategized Locking pattern [Sch99b] 'strategizes' (i.e. Strategy pattern [GH+95]) a component's synchronization to increase flexibility and reusability without degrading performance or maintainability. Synchronization strategies (e.g. single-threaded, multi-threaded using a thread mutex, multi-threaded using a readers/writers lock) can then be dynamically changed.

Thread-Safe Interface (Also known as Thread-safe Decorator): The Thread-Safe Interface pattern [Sch99b] is used to avoid *self-deadlock* (one component acquires a lock, then calls another method that tries to reacquire the same lock) and minimize unnecessary *locking* *overhead* (if a recursive component lock is used, then significant overhead may be incurred acquiring and releasing locks multiple times across intra-component method calls). Components should be structured so that locks will only be acquired/released at their 'border' (i.e. implementation methods should trust their calling methods, and not acquire or release locks).

5.1.3 Real-time Constraints as Strategies Pattern

The Real-time Constraints as Strategies pattern [Bus98] was developed by Frank Buschmann (Siemens AG). It is a special version of the Strategy pattern [GH+95], allowing the decoupling of real-time specific constraints and behaviour from the service to which they apply. Service methods are coded as strategies (ignoring timing requirements), while real-time behaviour and constraints are implemented as hook methods [Pre95]. For example, the service class might call a hook to deal with deadline misses for the system, or for a particular task.

By decoupling real-time behaviour from processing, RT applications implemented using this pattern are more portable, easier to modify, and potentially more efficient (as they allow runtime re-configuration of timing behaviour). The Real-time Constraints as Strategies pattern has been successfully applied to several real-time applications: including TAO [Cen00] (The ACE ORB, a real-time object request broker), REFORM [Sch97] (a framework for hot steel rolling mills), and REBOOT [Kar95] (for its flow control system).

5.1.4 Multithreaded Rendezvous Pattern

The Multithreaded Rendezvous pattern [JPA99] was developed by Ricardo Jiménez-Peris, Marta Patiño-Martínez, and Sergio Arévalo (Universidad Politecnica de Madrid); it proposes a variant of the Ada95 [ISO95] Rendezvous synchronization primitive for distributed transactions. The traditional Rendezvous primitive consists of a single-threaded server that provides a variety of services, and a set of clients that synchronize (or rendezvous) with the server as part of the calling mechanism. The proposed variant uses a multi-threaded server (one thread associated with each client), and a forwarder object that forwards calls to the corresponding server thread.

The advantage of using the multithreaded rendezvous pattern is that requests arriving from a variety of clients do not interfere with one another. The Multithreaded Rendezvous pattern has been applied successfully to the implementation of the language Transactional Drago [PJA98], an Ada extension to build fault-tolerant distributed applications that offers rendezvous as a synchronization mechanism for distributed transactions.

5.1.5 Recursive Control Pattern

The Recursive Control pattern [Sel96] was developed by Bran Selic (ObjecTime Ltd.). It describes a systematic method for dealing with support functions. Selic explains that these functions (including system startup/shutdown, failure detection and discovery, preventive maintenance, performance monitoring and statistics gathering, synchronization with external control systems, and online loading of new hardware and software) represent as much as 80% of all code written – but are typically given low priority at design time.

The Recursive Control pattern separates the control and service-providing aspects of a real-time program - allowing each to be defined and modified independently.

5.1.6 Recoverable Distributor Pattern

The Recoverable Distributor Pattern [ID96], developed by Nayeem Islam & Murthy Devarakonda (IBM T. J. Watson Research Center), assists developers of fault-tolerant, state-sharing distributed programs. This pattern focuses on the issues of performance and fault tolerance in distributed systems, and its usefulness is demonstrated through its successful implementation in two IBM projects – the Octopus distributed scheduler [IPS96] and a distributed lock manager for the Calypso file system [MDK91].

The Recoverable Distributor is used in systems where state sharing between multiple distributed processes is required. It provides a facility to manage the sharing in an efficient manner, and provides recovery facilities in case any of the processors fail. The actual methods used to deal with data consistency and fault tolerance are abstracted away – meaning that protocols for each can be set according to the requirements of a particular system.

5.2 Domain Specific RTOO Design Patterns

The patterns in the previous section were generally applicable to real-time software. This section presents more specialized patterns – applicable to a specific domain. The patterns are classified according to the type of real-time software to which they apply: Process Control; Communications, Command, and Control; or Embedded Computer.

5.2.1 Process Control Patterns

Process Control systems involve monitoring sensors, processing to calculate the current state (based on sensor inputs and previous state), and effecting change in the external system via effectors. Process control systems must react very quickly to changes in their environment.

5.2.1.1 Design Patterns for Avionics Control Systems

Doug Lea (SUNY Oswego and NY CASE Center) has collected a set of patterns [Lea00] for Avionics Control Systems (ACS), the main navigation system of an aircraft. These patterns were developed based on experience with the Avionics Domain Application Generation Environment (ADAGE) portion of the Domain-Specific Software Architecture (DSSA) project [DSS00].

There are 13 patterns in this collection:

Navigation Models: Represent estimates about the actual state of the aircraft and other real-world objects (updated continuously based on sensor data).

Objective Models: Represent the desired state of the aircraft.

Error Models: Represent differences between actual and desired state.

Action Models: Represent desired effector and display settings, in a form suitable for manipulating aircraft hardware.

Staging: Helper models interposed for estimation purposes.

Sensors: Inputs to the system.

Effectors: Transduce desired settings into mechanical actions.

Filters: Transform data on its way from component to component.

Guidance Modes: Components that compute objective model values.

Flow: Policies that govern the mechanics of information flow through the system.

Monitoring and Control: Facilities to dynamically track, enable, and alter system components.

Updates: Protocol allowing components to update each other.

Configuration: Strategies and tools for putting together an instance of the system.

5.2.1.2 Design Patterns for Fire Alarm Control Systems

The Design Patterns for Fire Alarm Control Systems [Mol96] were developed by Peter Molin and Lennart Ohlsson (University College of Karlskrona/Ronneby, Sweden), for microprocessor based fire alarm systems. There are five patterns in this collection:

Point pattern: Each sensor/actuator is logically modeled as a set of points whose binary state is determined internally. This decouples the fire alarm system from the specifics of how a sensor/actuator determines its state.

Deviation pattern: Sensors and actuators can be in normal state, or in some 'deviated' state (triggered by a fire, low-battery, faulty, etc.). Rather than store the state of each sensor/actuator at every microprocessor (which would use up valuable memory resources), store only information on ‘deviated’ objects and assume everything else valid.

Ticked Object pattern: The role of the fire-alarm system is to sample sensors, do some processing, then actuate alarm bells/close fire doors/call fire department/ etc. in case of fire. This functionality could be achieved using multi-threading but at too high a cost. Instead, use a simple event loop iterating through a list of TickedObject (one for each sensor). Because each sensor type has potentially different sampling/processing requirements, make a different child of TickedObject for each. This scheme uses the Bridge and Composite patterns from [GH+95].

Pool pattern: The pool pattern provides a mechanism for keeping a virtual container for all deviations in the system (accessible from any node in the system). Such a mechanism can use either remote iteration or the Proxy pattern [GH+95] - allowing querying for a list of all alarm signals, all faulty system components, etc. An additional benefit of this mechanism is that, in case of fire (which may destroy portions of the fire alarm communications network) the unaffected nodes can store physical copies of the disrupted nodes last state.

Lazy State pattern: Fire alarm systems, like most systems, allow user commands to over-ride the system logic (for testing purposes, to shut off part of the system, etc.). Since the system's behaviour is modeled using state charts, use the State pattern [GH+95] to dynamically compute state each time state-dependent behaviour is invoked. This solution allows new user commands to be added without requiring significant changes to the state machine.

5.2.2 Communication, Command and Control Patterns

Telecommunications software developers have been quick to embrace the design patterns movement, and to publish their patterns and experiences with them. Communication, Command and Control systems gather information from a variety of terminals and sensors, and effect change through effectors automatically or in response commands from the command post.

5.2.2.1 Pattern Language for Improving the Capacity of Reactive Systems

The Pattern Language for Improving the Capacity of Reactive Systems [Mes96] was developed by Gerard Meszaros (Bell Northern Research). The ten patterns in the language address issues relevant to distributed systems operating under heavy load:

Capacity Bottleneck: Customers frequently want to add terminals/users or otherwise allow an increasing number of requests without buying new hardware. A possible solution is to determine the bottlenecks for the system - their nature (memory, processing, messaging, etc.) and location (client, server, network, etc.) - then use optimization techniques on those elements that truly affect throughput.

Processing Capacity: There are three potential solutions when the bottleneck is a lack of processing power at one of the system components: optimize handlers for frequently recurring requests; add more processors; reduce the size of each processor's safety buffer. By understanding the effects of each alternative on system throughput, an appropriate compromise can be devised.

Optimize High-Runner Cases: When available processor power has already been maximized, but desired throughput has not been achieved, optimize the handlers for the most costly requests. Optimizing all handler code is infeasible, so use a profiler to determine where the system spends most of its resources, and optimize appropriately.

Shed Load: In order to prevent thrashing (wasting resources dealing with requests that it can't handle - occurs when the number of requests exceeds the maximum which can be successfully handled) the system should 'shed' requests when throughput approaches capacity.

Finish Work in Progress: Rather than shed requests arbitrarily, choose to shed new requests over continuations of work in progress. This requires an understanding of the relationship between requests.

Fresh Work Before Stale: Although a violation of the fairness doctrine, using a LIFO queue for requests ensures that all handled requests are done promptly. For example, in a telecommunications system operating at capacity, the oldest requests in the queue are probably stale anyway (e.g. user has given up waiting, and probably tried again).

Match Progress Work with New: If relationships between requests are known, then it may be possible to discard multiple waiting requests without processing them. For example, consider the following scenario from a telecom system at capacity: a user picks up the receiver, waits for a dial tone, gives up, and then hangs up the telephone. Since the system is at capacity, it may not get to the request for dial tone until after it processes the 'hang-up' event. If the two events are associated, the system can discard the 'wait for dial-tone' event at the same time as the 'hang-up' - without processing it.

Share the Load: If other optimization techniques do not provide the required increase in throughput, a solution might be to divide the load between more processors. Unfortunately, adding more processors does not provide a linear increase in processing power (due to inter-processor communication costs incurred). Ideally, move clearly partitioned processing to the new processor(s) to minimize synchronization costs.

Work Shed at Periphery: In cases of high load, requests may be moving through the system, only to be shed when they reach a capacity bottleneck. A solution is to inform processing nodes at the system's periphery of the available processing capacity of the system - thereby allowing requests to be shed early.

Leaky Bucket of Credits: In order for peripheral nodes to be able to shed requests effectively, they must be able to recognize when the system is overloaded. A solution is to use a 'leaky bucket' counter which is kept full by the central processor when the system is not at capacity. In this way, peripheral nodes can accept requests as long as they have credits in their bucket (which cause the counter to decrement) and shed them when the bucket is empty.

5.2.2.2 Pattern Language for Improving the Capacity of Layered Client/Server Systems with Multi-threaded Servers

Dorina Petriu and Gurudas Somadder (Carleton University) extended Meszaros' pattern language [Mes96] with [PS97], introducing the software server as a potential system bottleneck. Distributed systems are represented using directed acyclic graphs whose nodes represent clients or servers; by comparing these graphs with performance analysis results from the system under heavy load, bottleneck locations (client or intermediate server) can be uncovered and addressed. The patterns explain how to diagram the layered system, how to conduct performance analysis, and how uncovered software server bottlenecks should be dealt with.

Intermediate Server Bottleneck: This pattern describes a technique for determining if the bottleneck in an intermediate server is with the software, or the hardware. Such a bottleneck can occur if the request input queue builds up too rapidly, preventing the underlying hardware from being used effectively.

Server Multi-threading: If the bottleneck is at a software server using blocking I/O and communication primitives, then multi-threading should be used. The Thread Per Request, Thread Per Session, and Thread Pool patterns describe valid approaches.

Thread Per Request: Same as the Thread-per-Request pattern from [LS96], described in section 5.1.2.1.

Thread Per Session: Solution for a small number of sessions: server spawns off a thread for each incoming session.

Thread Pool: Same as the Thread Pool pattern from [LS96], described in section 5.1.2.1.

Minimized Serial Thread Management: Given a system that implements a Thread Pool, improve capacity by moving as much of the thread pool management from the main thread to each of the worker threads.

Multiprocessor: Same as the Share the Load pattern [Mes96], described in section 5.2.2.1.

5.2.2.3 Fault-tolerant Telecommunications System Patterns

The Fault-tolerant Telecommunications System Patterns [AC+96] were developed by Michael Adams, James Coplien, Robert Gamoke, Robert Hammer, Fred Keeve, and Keith Nicodemus (AT&T Bell Labs). These patterns are a subset of the pattern language used at AT&T, and focus on the ‘human factors’ (such systems must provide facilities to assist maintenance personnel) and ‘reliability’ (downtime of no more than two hours in forty years) characteristics of switching systems developed for the telecommunications industry.

Minimizing human intervention: History has shown that people cause the majority of problems in continuously running systems (taking incorrect actions, misinterpreting error messages, etc.). This pattern suggests designing systems to do almost everything themselves, deferring to the human only as a last resort.

People know best: However, there are cases when the human operator is better able to make correct decisions. Therefore design the system so that a knowledgeable user can override the automatic controls.

Five minutes of no escalation messages: When errors occur in continuously running systems, the human-machine interface can become saturated with error reports that may be rolling off the screen, or consuming resources just for the intense displaying activity. Rather than displaying a message for every change in state, display a periodic update message until the system is back to normal.

Riding over transients: Design the system to make sure that potentially transient error conditions really exist (either by checking several times, or using Leaky Bucket Counters to detect a critical number of occurrences). This way faults will be detected, but the system will not waste resources trying to deal with conditions that will go away naturally.

Leaky bucket counters: Same as Leaky Bucket of Credits pattern from [Mes96], described in 5.2.2.1.

SICO first and always: In systems where the System Integrity Control Program (SICO) coordinates system integrity, be sure to give it the ability and power to reinitialize the system whenever system sanity is threatened by error conditions. SICO should always be the highest-level function.

Try all hardware combos: In highly fault-tolerant systems the central controller (CC) must select a valid configuration (or path through the subsystems), even when some of the subsystems are faulty. This pattern suggests using a hardware counter representing which of the configurations is currently being attempted; if the initialization fails, then the counter is incremented, and the system reinitialized.

Fool me once: In systems where the Try All Hardware Combos pattern is used, the following scenario may occur: a valid configuration is found, the hardware counter is reset, application software starts, a software fault causes the system to reboot. To resolve this fault, the system may loop in reboots ("roll in recovery"), each time returning to the same configuration. This pattern suggests setting a time interval (using a Leaky Bucket Counter), then acknowledging only the first counter reset request received during the interval.

5.2.2.4 Telecommunications Distributed Processing Patterns

Another subset of the patterns used at AT&T Bell Labs is Dennis DeBruler's Telecommunications Distributed Processing Patterns. These patterns are a work in progress, but have been presented at PLoP conferences, and are available online [DeB00]. There are 16 patterns:

Define the Data Structure: The initial design of a distributed real-time system can be overwhelming. A good start is with a data-analysis exercise while ignoring processor boundaries.

Identify the Nouns: Begin the design stage with a brainstorming session - identifying the values, attributes, and roles of the system (and giving them precise names).

Factor Out Common Attributes: Design for reuse – in order to prevent cutting and pasting code at the coding stage. Identify relationships between objects, and diagram the inheritance structure.

Normalize the Roles: Many of the roles derived earlier should be split into multiple roles. Use normalizing techniques (the removal of undesirable insertion, update, and deletion dependencies; and the reduction of the need to restructure relations as additional information is added to the model).

Identify Problem Domain Relationships: Convert many-to-many relationships to two functional mappings and a new role.

Introduce Virtual Attributes: One problem with a network implementation is that quite a bit of navigation can be necessary to access an attribute. A solution to this problem is to provide 'accessor functions' for attributes (i.e. pass a pointer to attributes down the inheritance chain so that all children can easily access its value).

Animate the Data: The important events in the system all relate to the data – i.e. sampling inputs, processing based on input and state information, effecting change in the external world. The most generic way of meeting these functionality requirements is to assign each role to an actor, and map each actor to a processor.

Time Thread Analysis: Time thread analysis involves: (1) list the problem domain events/transactions that the real-world generates; (2) for each event, trace the causality flow through the roles of the data structure and note actions performed by each role. The result is that the actions needed by roles to animate the data are identified.

Determine the Actors: Use objects, callbacks, and Finite State Machines (FSM) to determine what lines of code get executed when an event happens.

Processes Considered Harmful (Draft): There are three problems with processes: context switching overhead, wasted stack space, and unnecessary message buffering. Use finite state machines instead.

Time Granularities (Draft): Determine the granularity required for responses (microsecond, millisecond, etc.). If granularity is extremely small (and related to processing), dedicate a processor to the task. Otherwise, use interrupts or queues.

Run to Completion (Draft): Avoid interrupts and preemptive scheduling whenever possible. Instead, allow handlers to run to completion (or voluntarily give up control). Avoid error handler interrupts by designing hardware with error counters that can be periodically polled.

Triage (Draft): The importance of an alarm is relative – significant system faults can become eclipsed by a greater fault. Likewise, a minor fault can sometimes be an indicator of something more serious.

System-Wide Pressure Gauges (Draft): Put some of the "computing" into the message transport sub-system. For example, this processing could bundle messages heading for the same destination, or delete low-priority error messages that may be obscuring (or slowing the delivery of) high-priority error messages.

Strategy vs. Tactics (Draft): Traditionally, algorithms have been either centralized or distributed. In many cases (such as with the use of the System-wide Pressure Gauges pattern for error messages), a hybrid approach would be beneficial.

Don't Retransmit at the Link Level (Draft): Because data will have to be validated at the higher levels of the protocol stack anyway, don't bother wasting time and resources re-transmitting lost data at the link layer. Instead, keep the link layer as fast and efficient as possible.

5.2.3 Embedded Computer Systems Patterns

Embedded computer systems reside within and are dedicated to monitoring, some physical equipment. Typically, these systems have limited processor and memory resources, no secondary storage, and must operate without user intervention. These and other issues make the design of software for embedded systems difficult.

5.2.3.1 Patterns for Managing Limited Memory

The Patterns for Managing Limited Memory [NW98] were developed by James Noble and Charles Weir. The patterns presented address issues relevant primarily to programmers of embedded commercial products such as mobile phones and palmtop computers where the cost of memory is still a significant development consideration. The patterns are categorized as "high-level and process”, “choose suitable data structures”, “compression”, “secondary storage”, and “read-only storage”.

The *High-level and process* patterns are a set of organizational patterns that encourage a memory-efficient mindset.

Think Small: Imagine the system as smaller than it really is, and encourage every team member to keep tight control on memory use. Use design and code reviews to get rid of wasteful features, habits, and techniques.

Memory Budget: Develop a memory budget up-front, allocating resources for the system and for each component.

Memory Overdraft: Include some flexibility in the memory budget to account for unforeseen needs.

Make the User Worry: Memory allocations for interactive systems can be difficult to budget; often it makes more sense to leave memory as overdraft, and let the user manage it.

Partial Failure: When programs run out of memory, choose partial failure or degraded performance over complete failure.

Do the Decent Thing: When a system runs out of memory, choose to free up memory from less vital components rather than fail the most important tasks. As the system approaches memory capacity, notify all running tasks that they should relinquish unnecessary resources. In extreme cases, less vital components should be told to close down.

Test Small: Use testing techniques that simulate out of memory conditions, to ensure that Partial Failure and Do the Decent Thing work properly.

Memory Performance Assessment: An alternative approach for developing software for systems with limited memory is to design the software considering memory constraints only where they have a significant impact on design, then (if the memory requirements exceed available memory) locate and optimize wasteful areas.

The *Choose suitable data structures* patterns encourage the use of data structures that will scale down to memory-constrained systems:

Variable Sized Data Structure: Most data structures in limited memory systems should be variable sized. As memory requirements change, the data structure should either request more memory, or relinquish the memory that it no longer needs.

Fixed Size Data Structure: Fixed size data structures should be used for operations for which it is crucial that memory requirements do not exceed memory available. Aside from improving the predictability of memory usage, this scheme also simplifies memory accesses, and reduces the overhead associated with garbage collection and memory management. Although fixed size data structures will always use more memory than they need, the overall memory requirements may be less – since the memory management overhead is not required.

Multiple Representations: Since there is often no one best representation for a data structure, a single interface should be provided – with multiple concrete implementations. This way the programmer can choose which implementation is most appropriate for a particular task. Variants of this pattern include: **Dynamic Multiple Representations** – which allows dynamically changing between implementations at runtime; and **Basic Type with Object Wrapper** – which suggests using a compressed representation for storage, and wrapping it in an object for processing.

Once the system has been coded as efficiently as possible, techniques described in the *Compression* patterns can be used to shrink the footprint even further. Compression techniques introduce additional complexity, and possibly reduced performance - these tradeoffs must be carefully considered:

Sharing: When the same data occurs many times in a program, store it once and allow it to be shared.

Byte Coding: To reduce the space occupied by program code, compile it to an intermediate form (byte code) optimized to take up as little space as possible, then use an interpreter to convert it back to full machine language as required.

String Compression: Use a standard string compression technique to reduce the storage space required for strings (e.g. strings displayed to the user, database keys, etc.).

File Compression: To reduce the space occupied by data files, use a standard compression algorithm.

Short Names: A simple technique for reducing the amount of memory needed to store source code or scripting code is to shorten the length of variable names.

Secondary storage (although not available in most embedded systems) gives the programmer access to lots of cheap memory but at the cost of management complexity and performance. The *Secondary storage* patterns provide details of these tradeoffs, and provide useful recommendations:

Program Chaining: If a large program can be broken up into independent 'pieces', then only one piece needs to be kept in primary memory at a time – the rest moved to secondary storage. The pieces can be chained together so that when one finishes, the next is loaded to memory and executed.

Data Chaining: The same process can be applied to data; in cases where there is a large amount of data to be processed, break it up into several smaller pieces, then process them one at a time.

Autoloading: Divide the system into a main program, and a set of extensions. Move the extensions from secondary storage to primary memory only when required.

Resource Files: An efficient mechanism for handling static data that will only be needed for short periods of time is to store it in secondary storage as resource files. Programs can then request a resource files from a resource manager, which will load it if is not already in memory. When finished processing the resource file, it is simply unloaded. For example, a resource file containing the information required to properly display a window in a GUI system can be loaded and processed when the window is opened, then immediately unloaded.

Segmentation: Segmentation is an alternative to autoloading. Programs are divided into segments, which can be loaded and unloaded as required. The division of code into segments, and the loading and unloading of segments are typically the responsibility of the programmer.

Paging: Demand paging can be used to give the illusion of more primary memory. Given a page size, memory is automatically divided into pages, and shuffled between main memory and secondary storage as required.

Another method of conserving main memory is to store static code and data in read-only memory. The *Read-only storage* provide strategies for dividing programs into modifiable and static modules, and for efficiently using read-only memory:

Copy-on-Write: Sometimes code or data stored in read-only memory needs to be changed. A solution is to copy the code or data to memory, make the change, and then use the copy in the future.

Hooks: Link read-only code and data together using hooks in writable memory. This way, if a section of read-only code or data needs to be extended or replaced, all that is needed is to change the hook.

Pre-initialize Data: Pre-initialize data in the read-only store, then load it directly. This technique saves the processing time that would be otherwise required to initialize the data structure.

5.2.3.2 Pattern Language for Simple Embedded Systems

The Pattern Language for Simple Embedded Systems [Bot99] was developed by Mark Bottomley (Computing Devices Canada); it consists of a framework called *The Carousel*, and a set of patterns. The carousel framework relies on the use of a fixed foreground/background cyclic executive (foreground cycles through I/O and determines new state while background does the data processing). The following patterns assist in fine-tuning the framework to suite a particular project.

The Carousel: For a system designed around a fixed cycle time, processing time must be allocated for the Foreground (synchronous I/O, Process, and Update) and Background (asynchronous) activities. This pattern describes how to classify the various activities in the system.

The Carousel Rate: Once the system activities have been identified and classified, the carousel rate must be determined. This pattern describes how to calculate the speed at which the carousel should ‘spin’.

Too Fast for Me: Many activities will not need to be performed at the Carousel Rate; instead, keep a ‘step’ counter, and have the slower activities run every N cycles.

Asynchronous Activities: Some activities cannot be polled, or require a response time that is much faster than can be met at the Carousel Rate; these must be handled using interrupts.

Background Activities: The processing time remaining after the foreground activities of the Carousel are finished can be used to process background activities. This pattern suggests using a priority queue to select which activity to process next.

Big Calculation: Some calculations may require processing time equivalent to many cycles. These calculations must be examined to determine their total processing requirement, latency and update frequency – then broken down into foreground, background, and interrupt activities.

Nap Time: For systems with power saving requirements, the background processing time remaining after all activities have been processed can be used to put the micro-controller and possibly associated peripherals into an idle (keep the peripherals running) or power-down mode.

Maintenance Check: Sanity checking for safety and/or functionality should be conducted regularly. These activities should be assigned to the Carousel as appropriate.

Sharing: Often inputs and outputs will share common hardware (e.g. sensors, digital I/O ports, and communication busses). As a way of than multiplexing or de-multiplexing the data on these shared resources, use virtual I/O. This makes I/O handling simple and consistent.

Sensor Filter: Since many input signals will not be clean, some processing and filtering will be required to prepare them for use in the rest of the system. This pattern suggests passing inputs through the Noisy World and the Perfect World filters consecutively.

Noisy World: The Noisy World uses five filters to extract the signal from the noise: (1) sensor/converter health to verify that the converter believes that the signal conversion is valid; (2) *anomaly clipping* to filter large noise spikes; (3) *averaging* filter takes the average of several samples; (4) *dead banding* filter which ignores low-level noise; (5) *result re-scaling* filter to convert the scale used by the sensor to a more useful unit (e.g. to convert a binary sensor value to p.s.i. or kPa).

Perfect World: Once the signal has been extracted from the noise, the Perfect World filter is used to condition it for use by the remainder of the system. Three filters are used: (1) *cross-coupling* filter which combines multiple inputs to create or calibrate a sensor; (2) *soft limiting* filter to restrict the range of acceptable values (e.g. to report zero rather than a negative value); (3) *default/override* filter to decide how to handle faults reported by Noisy World.

5.2.3.3 Patterns to Ease the Port of Micro-kernels in Embedded Systems

The Patterns to Ease the Port of Microkernels in Embedded Systems [deC96] were developed by Michel de Champlain (University of Canterbury, New Zealand). The purpose of these patterns is to facilitate the design of portable micro-kernels.

Object Manager: The Object Manager pattern encapsulates the implementation and management of a set of simple and predictable abstractions (descriptor stack, priority queue, and delta list). This pattern is very similar to Schmidt's Object Lifetime Manager pattern [Sch99a], but differs in that the allowable abstractions (objects with methods and data) are explicitly specified.

Timer: The Timer pattern decouples timing from scheduling functionality. The simple mechanism periodically updates delta list objects with the current time, and detects the presence of expired objects.

Scheduler: The Scheduler pattern allows dynamic scheduling policy selection (i.e. by priority, by earliest deadline, or by rate monotonic analysis). This pattern is essentially a specialization of the GoF Strategy pattern [GH+95].

Portability is a significant concern for real-time developers; systems are often coded in assembly language (and/or make use of other hardware specific features) that improve performance and predictability but significantly increase development cost and reduce portability.

5.3 RTOO design pattern language

The 'pattern language' concept comes from Alexander, who believed that the inter-pattern relationships were as important as the patterns themselves – and should be explicitly documented. Unfortunately, most of the RTOO patterns discussed in this chapter – even those claiming to belong to a pattern language – documented independent solutions with little mention of how they could be used in conjunction with other patterns.

This chapter has attempted to present design patterns for RTOO software in pattern language form, by using a simple classification (along ‘applicability’ axes) and discussing some of the inter-pattern relationships. The following sections will make these details explicit, and will discuss some of the issues to be overcome before this collection of patterns can be called a ‘pattern language’. These issues include the high level classification and organization of the language, inter-pattern relationships, pattern language hierarchies, and the completeness of the language.

5.3.1 Organization/Classification

The RTOO design patterns in this chapter are organized/classified according to applicability axes: the top-level classification is by domain specificity (widely applicable RTOO design patterns vs. patterns that are domain specific); the widely applicable RTOO design patterns are sub-classified by primary author, while the domain specific RTOO design patterns are sub-classified by the domain to which they apply.

This is a simple classification mechanism that seems to be appropriate for this collection of patterns. Other potential classifications include the GoF classification (by purpose and scope), by pattern type (e.g. design patterns, organizational patterns, AntiPatterns, etc.), and by problem solved (e.g. improve efficiency, address synchronization issues, object creation/destruction, etc.).

5.3.2 Inter-pattern Relationships

The systematic application of sets of related patterns requires that inter-pattern relationships be explicitly documented. Although many authors document these relationships within their own pattern languages or collections, very few consider the relationships with patterns outside their authorship. This section will discuss some of the more common relationship types: ‘alias’, ‘uses’, ‘specializes’, and ‘alternative’. It will also discuss some of the relationships present within the RTOO design patterns presented earlier in the chapter.

5.3.2.1 Alias

Many pattern templates (including GoF and Alexandrian) include a section for 'aliases' – allowing authors to document other names by which the pattern may be known. However, it may be that these aliases are not just other names for the same pattern, but rather distinct patterns that duplicate the proposed problem, context and solution. This is the first relationship type, called the 'alias' relationship.

Because patterns document recurring solutions to problems, and because of the large number of existing patterns, some duplication is inevitable. This is not necessarily a problem, since authors will present their patterns differently, and contribute examples from their own experiences. However, when this relationship is not explicitly documented, readers can become frustrated by the duplication in the literature. Within repositories, it should be decided whether to retain all duplicates of a particular pattern, or to keep one and use proxies for the other instances. The latter alternative was used in this chapter for the following patterns exhibiting the alias relationship:

RTOO Design Pattern		**Alias Of**	
Author	**Pattern**	**Author**	**Pattern**
Adams *et. al.*	Leaky Bucket Counters	Meszaros	Leaky Bucket of Credits
Douglass	Container-Iterator	GoF	Iterator
Douglass	Observer	GoF	Observer
Douglass	Proxy	GoF	Proxy
Grasso	Thread-per-object	Schmidt	Active Object
Petriu & Somadder	Thread-per-request	Schmidt	Thread-per-request
Petriu & Somadder	Thread Pool	Schmidt	Thread Pool

Table 5.1: RTOO Patterns Exhibiting the Alias Relationship

Note that this relationship is bi-directional; there is no implication that one author 'copied' another's work. There are always subtle differences in how the patterns are describes, in the usage proposed, and in the examples given; it is therefore assumed that each author discovered the pattern independently.

5.3.2.2 Uses

The 'uses' relationship comes directly from Alexander, although he did not document it as such. Alexander organized his patterns according to their relative position in a pattern hierarchy (discussed in the next section). Higher level patterns represented coarser solutions, which could be 'filled in' using finer grained patterns. This process of 'filling in' or completing a given pattern represents the 'uses' relationship.

All of the pattern languages described in this chapter have examples of the 'uses' relationship; since these are explicitly documented in the full texts of the patterns they will not be listed here. However, this relationship may exist beyond the scope of a single pattern language. These examples of the 'uses' relationship are not so easy to find, and are listed below:

RTOO Design Pattern		**Uses**	
Author	**Pattern**	**Author**	**Pattern**
Buschmann	Real-time Constraints as Strategies	GoF	Strategy
De Champlain	Scheduler	GoF	Strategy
Molin & Ohlsson	Ticked Object	GoF	Bridge, Composite
Molin & Ohlsson	Lazy State	GoF	State
Molin & Ohlsson	Pool	GoF	Proxy
Schmidt	Strategized Locking	GoF	Strategy
Schmidt	Object Lifetime Manager	GoF	Singleton, Factory

Table 5.2: RTOO Patterns Exhibiting the Uses Relationship

5.3.2.3 Specializes

The 'specializes' relationship exists when an author takes an existing pattern and applies it to a different problem space. The result is two distinct patterns that share a similar problem, but because the context is different the solution must be adapted. The patterns in this chapter that exhibit this relationship include:

RTOO Design Pattern		**Specializes**	
Author	**Pattern**	**Author**	**Pattern**
De Champlain	Object Manager	Schmidt	Object Lifetime Manager
Douglass	Broker	GoF	Proxy
Douglass	Safety Executive	Douglass	Watchdog
Douglass	Latch State	Douglass	Persistent Latch State
Douglass	Microkernel Architecture	Douglass	Six-Tier Microkernel Architecture
Grasso	Synchronizer	Schmidt	Active Object
Grasso	Transactional Synchronizer	Grasso	Synchronizer
Noble & Weir	Multiple Representations	Noble & Weir	Dynamic Multiple Representations
Noble & Weir	Multiple Representations	Noble & Weir	Basic Types with Object Wrapper
Petriu & Somadder	Intermediate Server Bottleneck	Meszaros	Capacity Bottleneck
Schmidt	Thread-safe Interface	GoF	Decorator

Table 5.3: RTOO Patterns Exhibiting the Specializes Relationship

5.3.2.4 Alternative

The ‘alternative’ relationship exists between two patterns sharing the same problem and context, but whose solutions are different. Hence the patterns represent two alternative means of resolving the same forces. The patterns from this chapter that exhibit the ‘alternative’ relationship include:

RTOO Design Pattern		**Alternative**	
Author	**Pattern**	**Author**	**Pattern**
Schmidt	Reactor	Schmidt	Active Object
Noble & Weir	Autoloading	Noble & Weir	Segmentation, Paging
Petriu & Somadder	Thread per Request	Petriu & Somadder	Thread per session

Table 5.4: RTOO Patterns Exhibiting the Alternative Relationship

Since pattern authors document only what they know to work, the ‘alternatives’ relationship provides a mechanism for other authors to present other ways to resolve the problem. This can add to the completeness of a language, and help to further specify arbitrary solutions.

5.3.3 Pattern Language Hierarchies

Alexander’s pattern language form suggests that there will always be a hierarchy of patterns in a pattern language; that a pattern can be either higher or lower in relation to another pattern. However, there is currently no standard way of documenting these hierarchies either textually or diagrammatically.

The following sub-languages from this chapter included hierarchy information (either implicitly in the text, or explicitly using some type of diagram):

- Petriu & Somadder’s Pattern Language for Improving the Capacity of Layered Client/Server Systems with Multi-threaded Servers [PS97] (which extends Meszaros’ Pattern Language for Improving the Capacity of Reactive Systems [Mes96]). Figure 5.1 illustrates the hierarchy (Meszaros’ patterns in boxes).

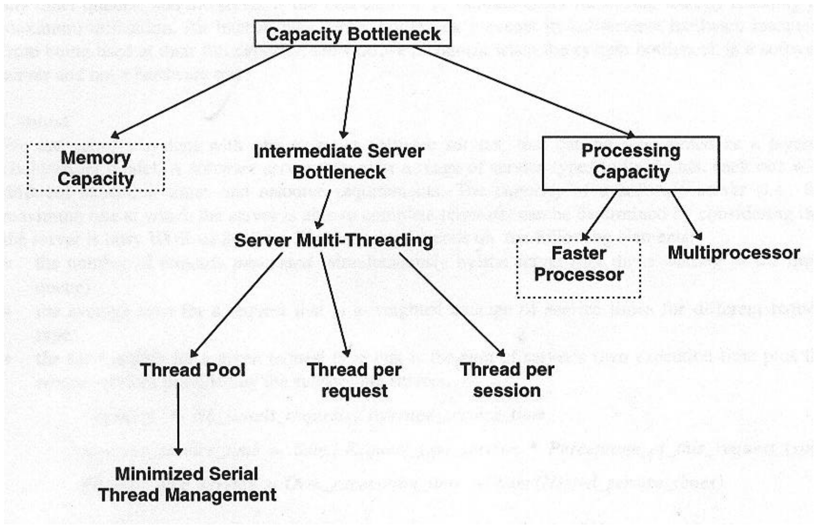


Figure 5.1: Example of Pattern Hierarchy Relationships (from [PS97])

- Mark Bottomley's Pattern language for simple embedded systems [Bot99].
- Though not completely documented at present, Doug Schmidt is working to integrate his patterns for concurrent, parallel and distributed systems [Sch00] into a hierarchical pattern language.

Typically, the relationship represented by the arrows in a hierarchy diagram is the 'uses' relationship; higher-level patterns use lower-level patterns to complete their solutions. However, the other relationship types (alias, specializes, and alternative) can sometimes be detected. For example, in Figure 5.1 the Thread per Request, and Thread per session are alternatives to each other.

5.3.4 Completeness

For a collection of patterns to be called a pattern language it should not only explicitly document the inter-pattern relationships, there should also be a sense of 'completeness' (that all combinations of forces in the problem space can be resolved using the constituent patterns). In practice, it is unlikely that any pattern language will ever be complete – since completing (or even determining completeness of) a pattern language is extremely difficult.

As an RTOO pattern language, the collection described in this chapter is very incomplete. There are important issues that are not addressed in the language, and although some sub-languages cover many of the forces in their problem spaces, others address only a very small subset.

Notably absent from the collection of patterns is discussion of timing and schedulability issues. These can be significant issues in the design of RTOO software, for which there are certainly patterns to be found. However, their absence from the design patterns literature may be indicative of the software process currently being employed – to neglect these issues until the implementation phase. For example, although formal methods may be able to determine in advance that time constraints would not be met, it is often less costly to do some rough up-front calculations for feasibility, build the system, and then take corrective action (e.g. hack the code, buy a faster processor) if the system is not running fast enough.

Despite the incompleteness of the collection as a whole, there are some sub-languages that can subjectively be judged more complete than others. For example, Doug Schmidt's patterns [Sch00] represent a sub-language that can be used to resolve many of forces present in the design of concurrent, parallel, and distributed systems, particularly when the extensions from Grasso [Gra97] are incorporated. The communications, command, and control patterns discussed in section 5.2 also form a fairly complete pattern language.

The less complete sub-languages are those where the patterns themselves do not fully describe their context, problem, and solution (e.g. Bruce Douglass' RT Patterns [Dou99] and Doug Lea's Patterns for Avionics Control Systems [Lea00]) and those whose solutions are too simplistic or that otherwise do not resolve their forces (e.g. de Champlain's patterns to ease the port of micro-kernels [deC96], which addresses only a very small subset of the issues related to porting micro-kernels). Sometimes, pattern languages can be made more complete by adding patterns. This is particularly useful in cases where the author's choice of solution seems arbitrary (e.g. Mark Bottomley in his Pattern Language for Simple Embedded Systems [Bot99] discusses only a few of the many filtering techniques available to prepare input signals for processing). The addition of ‘alternative’ patterns can very easily account for these types of discrepancies – increasing the completeness of the sub-language and its value to the developer.

5.4 Summary

This chapter has extracted and discussed many design patterns addressing issues related to the design of real-time object-oriented software. Unfortunately, locating these patterns was difficult due to the current disorganization of the pattern literature. Patterns are spread through books, journal articles, conference proceedings, and on the web – with very little structure. This lack of cohesion is noticeable at the pattern level as well; although the relationships between patterns in languages or collections are often described, the relationships between patterns across collections/languages are rarely mentioned. This is unfortunate, since one of the strengths of the pattern form is the way paths through a set of patterns can be used to resolve complete sets of forces in a problem space. Most pattern authors are currently focusing on creating new patterns, rather than examining how the existing patterns should work together.

The simple classification used in this chapter helps resolve these issues by arranging a set of patterns for RTOO design in a logical ordering, then documenting explicitly the relationships present in the collection. However, it is far from a final presentation and categorization – this may happen more completely as the field matures, not just for RT patterns but for the patterns literature as a whole.

C h a p t e r 6

CHAPTER 6: TOOL SUPPORT FOR PATTERNS

There currently exists a wealth and variety of documented experience in pattern form; however, finding the right pattern when it is needed – then choosing the appropriate implementation method - remains difficult. One of the reasons for the popularity of the Gang of Four text is that it presents a relatively small set of very good quality patterns in an organized fashion. With only 23 patterns to learn, it is not unreasonable to expect developers to remember the basics of the patterns and recognize instances where they might be applicable. As the number of patterns increases (for example, the 150+ RTOO design patterns identified in chapter 5 of this thesis) tool support is needed to make the patterns more accessible.

This chapter will first present an overview of available patterns tools then will discuss the potential for convergence between patterns tools and RTOO modeling tools. As a concrete example, an extension to Rational Rose-RT [Rat00c] is proposed, adding the functionality of design pattern tools as a layer of abstraction over its structure and behaviour model layer.

6.1 Types of Pattern Tools

Many researchers, both from industry and academia, have attempted to develop tools to support the application of patterns. These tools can loosely be classified as tools that generate code from patterns, tools that detect patterns in code or design diagrams, tools that automatically refactor code, and tools that classify and index patterns. Often the tools provide more than one of these functions.

Tools that automatically generate code from design patterns (e.g. [KB95, Mei96]) must consider the tradeoffs between the expressiveness of the informal pattern notation, and the ability to easily generate code. The approach that has typically been taken is to formalize patterns as *contracts* (a mechanism for explicitly specifying behavioural compositions [HHG90, Hol92]), allowing patterns to be integrated in a design as a set of components, or micro-frameworks. This mechanism allows for the association of multiple concrete implementations with each abstract pattern.

Tools that detect patterns in code or design diagrams (e.g. [Bro96, Mad00]) are faced with another dilemma: the concept of ‘compression’ (i.e. multiple patterns may be combined into a single solution). Not only must the tool recognize a potentially infinite variety of implementations for the patterns, but also must extract a variety of intertwined patterns from a compressed solution.

Refactoring tools (e.g. [Ref00, Opd92]) have seen some success, particularly with SmallTalk code. A refactoring is a related but distinct entity from patterns; these are small-scale mechanisms for making code easier to understand and maintain without changing its behaviour. For example, a refactoring might provide a mechanism for moving code from a bloated class to a subclass, or for abstracting a class up a hierarchy. Because it is operating at such a small granularity, this process can be semi-automatic.

Finally, there are tools that classify and index patterns. These tools range from simple web-based applications that store patterns by name and provide full text search (e.g. the Portland Pattern Repository [Por00], the GoF book [GH+95] on CDROM), to collaborative pattern writing environments (e.g. the Wiki Wiki Web [Wik00]), to basic pattern catalogs (e.g. Pattern Depot [Pat00]), to more sophisticated tools that store patterns classified according to various criteria along with sample source code (e.g. Blueprint Technologies Framework Studio [Blu00]). Most experience reports from chapter 4 describe some type of online catalog – since it is infeasible to expect developers to search through volumes of patterns every time they have a problem. This type of tool meets an existing need, and has the potential to integrate with the other tool types (for example, Budinsky *et. al.* has developed a pattern cataloging tool that has a section immediately following each pattern to automatically generate sample code [BF+96]). Pattern repository tools are described in more detail in the following section.

6.2 Examples of Pattern Repository Tools

The tools described in this section provide varying levels of functionality; the first few tools are very simple, more powerful tools are described later in the section.

6.2.1 Portland Pattern Repository

The Portland Pattern Repository [Por00] is an example of a website that stores patterns and related material, as well as links to other patterns sites on the Internet. The advantage of this type of repository is that it makes sets of patterns widely available; the disadvantage is that updates typically are not frequent enough, so content may be out-of-date.

6.2.2 Wiki Wiki Web

The Wiki Wiki Web [Wik00] is an online tool that allows multiple users to collaboratively edit web pages via a web browser. The underlying engine consists of a database and a set of CGI scripts used to retrieve or update the data. When the pages are retrieved from the database certain keywords appearing in the saved content are translated to the appropriate HTML tags. Access control is managed by the author, based on the list of authors and editors specified at the bottom of each page.

The beauty of Wiki is that it allows content to grow in a piecemeal fashion. Unfortunately, the cost of this freedom for the authors is that there is very little structure to the site – making it sometimes difficult to navigate.

6.2.3 Pattern Depot

Pattern Depot [Pat00] is an online pattern catalog developed and sponsored by Addison-Wesley. It uses a standard template to store patterns, classified according to domain (miscellaneous, language-specific, architectural, design, analysis, organizational, pedagogical, telecommunication, concurrent/ distributed, user interface, or business) and status (published, published draft, hidden from others, or deleted). The actual pattern can then be attached along with supporting files such as images, models, or source code.

There are some strong features in this tool: the standard template greatly simplifies finding relevant patterns, and hyperlink capability allows linking to author homepages, relevant patterns, conference proceedings, and other online content. However, there are also some significant problems:

- The classification by domain mechanism should be multi-dimensional, since the current scheme does not adequately reflect the domains to which a pattern might be applicable. For example, most architectural patterns are also design patterns, and most telecommunications patterns are also concurrent/ distributed.
- Authors must enter content twice, in the classification and as an attachment.

6.2.4 Budinsky et. al.

Recognizing the need for tool support to assist developers in finding and implementing patterns, Budinsky *et. al.* developed a tool [BF+96] that provides the GoF patterns in HTML format, and supports the generating design pattern code automatically from a small amount of user-supplied information. Patterns are organized with 'sections' (i.e. Intent, Context, Problem, etc.) on separate web pages, followed by a code generation page. The code generation page prompts the user for information like names for the participants and choices for design tradeoffs - then uses Perl scripts to automatically create C++ class declarations and definitions that implement the pattern.

The fundamental problem with the solution discussed by Budinsky *et. al.* is that it requires too much work to enter new patterns. New scripts must be written for every pattern added to the tool – and must be modified if new tradeoffs or design choices are added to existing patterns.

6.2.5 Blueprint Technologies Framework Studio

Framework Studio allows developers to store patterns, frameworks, components, or other 'artifacts' for later reuse. It integrates with Rational Rose [Rat00b] to capture the design diagrams, and with Microsoft Visual Studio [Mic00] for the concrete implementations.

Framework Studio provides the following features:

- The tool supports storage of four types of potentially reusable *elements*: *Artifacts* (any product of the development process); *Components* (encapsulated part of a system); *Frameworks*; or *Patterns*. The elements are stored in databases called *repositories*, either locally or shared over a network.
- Pattern classification and description has a concrete structure (Name, Author(s), Intent, Motivation, Known Uses, See Also) but also provides flexibility in what additional content will be associated under a variety of titles (Business Domains, Problems Solved, Benefits, Liabilities, Implementation details, etc.).
- Ability to search the repositories for elements. Searches can be brute force (using the full-text indexes of the Repositories), or more targetted (making use of the classification mechanisms provided).
- Ability to implement a captured component, framework, or pattern. The developer has the option of saving associated source files to disk, or to generate into Microsoft Visual Studio [Mic00].
- Ability to apply a framework or pattern to a Rational Rose [Rat00b] model. This allows the developer to associate roles in a pattern with classes in an existing model, and to create the other necessary classes.
- Ability to create HTML files from an element.
- The tool ships with the GoF design patterns (from [GH+95]) and the Buschmann design patterns (from [BM+96]); Blueprint Technologies sells other pattern repositories, including the Fowler analysis patterns [Fow97], and David Hay's data model patterns [Hay96].
- Framework Studio's *Object Miner* provides the ability to parse documents written in Microsoft Word [Mic00] or Rational RequisitePro [Rat00b]. This parsing process mines documents looking for possible classes and methods then saves them in Rational Rose.

However, there are some problems with Framework Studio:

- There is no explicit support for pattern languages – although a partial work-around would be to have a separate repository for each pattern language, and link the patterns together using the "See Also" field. Patterns are rarely meant to be used in isolation; the relationships between patterns are extremely important, and could be better documented in the tool.
- It does not differentiate between pattern types (design patterns vs. analysis patterns for example).

- Associating many implementations with a single pattern is possible, but makes using the automatic implementation feature difficult, and is very messy with complex patterns that require multiple files.

6.3 Tool Convergence

There is strong potential for convergence between pattern tools and RTOO modeling tools. One approach is to add the functionality of design pattern tools as a layer of abstraction over the structure and behaviour model layers in an RTOO modeling tool. This will allow design patterns to be more easily integrated into the modeling process.

As a concrete example, this section proposes an extension to Rose-RT [Rat00c]. Figure 6.1 demonstrates the concept underlying the proposal: an abstraction layer for patterns (incorporating features from a variety of pattern tools) will be added above the Rose-RT models (consisting of class and structure diagrams, and statecharts). The intent is that this additional abstraction layer will facilitate the design of Rose-RT executable models that can then be compiled and ported to target. The result should be to improve the quality of RTOO software designed using Rose-RT, facilitate the use of design patterns (thereby increasing the level of design reuse), and make explicit the patterns used in designs.

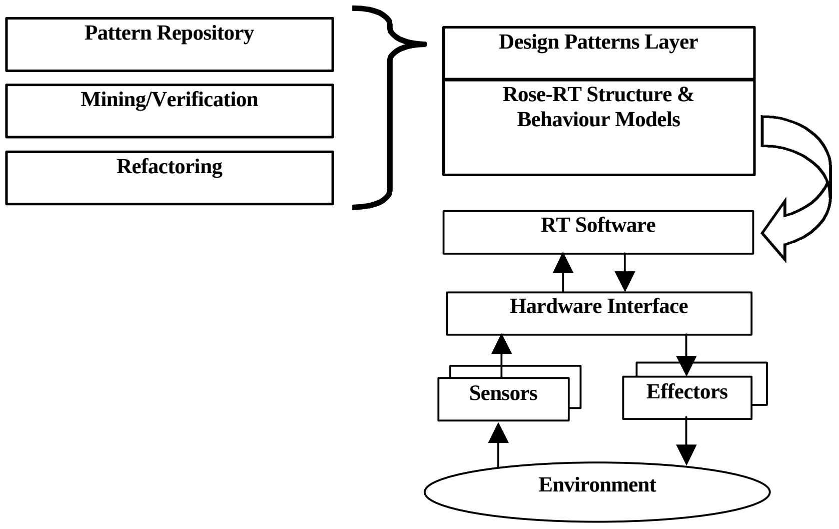


Figure 6.1: Tool Convergence

6.3.1 Desirable Functionality

This section describes a list of desirable features. The sections that follow will outline some of the issues to be reconciled before implementation is possible, then will extract from this list a set of features that are feasible, and make a proposal for the Rose-RT extension.

- **Multiple pattern repositories:** Provide support for multiple pattern repositories, with the possibility to share repositories over a network.
- **Improved classification mechanism:** An improved mechanism for structuring the potentially large sets of patterns to facilitate finding applicable patterns.
- **Ease of Expansion:** Relatively easy to add or extend patterns.
- **Explicit support for pattern languages:** Cross-referencing between patterns in languages, and the ability to specify relationships, including integration issues, between patterns.
- **Multiple pattern implementations**: Associate multiple concrete implementations with each abstract pattern specification.
- **Multiple implementation types:** Allow pattern implementations to consist of Rose-RT behaviour models and/or structure models, or files of another type.
- **Highlight patterns in models:** Keep track of where patterns have been used in a model, and make it easy for a developer to see where and how these patterns have been applied.
- **Pattern verification:** Automatically verify whether changes to the model have 'broken' any of the patterns.
- **Pattern mining:** Automatically detect occurrences of patterns in a model.
- **Three methods of pattern implementation:** Support top-down, bottom-up, or mixed pattern implementation (i.e. create Capsules/ Protocols/ Data from one of the stored implementations for the pattern; associate Capsules/ Protocols/ Data from an existing model with those of a stored implementation for the pattern; or use some combination).
- **HTML publishing:** Publish the set of patterns to an Internet or Intranet website so they can be shared with others who do not have access to the tool.
- **Allow patterns and modeling tool to exist independently:** Use the patterns tool without Rose-RT, or use Rose-RT without the patterns extension.

6.3.2 Issues

Convergence of design patterns tools with RTOO modeling tools has great potential; however, there are some significant issues to be addressed. The proposal presented in the next section suggests particular ways of resolving the following issues:

- **Compression:** A single class may participate in multiple patterns; this issue will complicate the pattern detection, verification, and highlighting functionalities.
- **Rose-RT Classes:** Rose-RT models consist of capsule, protocol and data classes; not all patterns will have structure/behaviour that can reconcile with these class types. Developers may have other patterns that they want included in the repository, but that don't map well to Rose-RT models (e.g. process/organizational patterns, or design patterns/idioms for components of systems not implemented in a language supported by Rose-RT).
- **Abstract nature of patterns:** The mining and verification functions will be very difficult if flexibility of implementation is allowed.
- **Programming language issues:** Although design patterns should ideally be language independent, the reality is that many work only (or at least work best) in the context of a particular language or language type.

6.3.3 Proposal for Extension

Based on the previous two sections, what appears to be a feasible and appropriate extension to Rose-RT is proposed. This proposal represents a prototype implementation, and requires minimal change to the Rose-RT toolset. It is presented in terms of the major design decisions that must be made.

6.3.3.1 Where should the patterns functionality reside?

The first important decision to be made is whether to add design pattern functionality to Rose-RT or to keep the pattern tool a distinct entity, with the ability to interact with Rose-RT. The advantages of having the pattern layer fully integrated into Rose-RT are that the pattern mining and verification processes, as well as the ability to keep track of where patterns are used in models, are greatly simplified. However, this is potentially problematic for users who want to use Rose-RT but not patterns, or who want to use patterns without Rose-RT. Further, developers may expend significant energy developing pattern repositories, and should not risk losing these repositories if they change modeling tools. It is thus proposed that the pattern tool remain distinct from Rose-RT, but that the models developed in Rose-RT can be used as implementations of the patterns. For full benefit, some modifications to Rose-RT may be required (for example, to support highlighting patterns in models); however, these should be in the form of optional plug-ins rather than a required feature of the tool.

6.3.3.2 What is the recommended initial set of features?

The following are the recommended initial features of the tool. Choosing to implement these features raises other design decisions that will be discussed in the sections that follow.

- **Multiple pattern repositories**
- **Improved classification mechanism**
- **Ease of expansion**
- **Explicit support for pattern languages**
- **Multiple pattern implementations**
- **Multiple implementation types**
- **Highlight patterns in models**
- **Three methods of pattern implementation**
- **HTML publishing**

These features all appear to be feasible in the first phase of the toolset, and their implementation should represent a significant contribution to the Rose-RT toolset.

6.3.3.3 What other features should be considered?

These are features that would be desirable, but that are infeasible in a prototype due to their complexity. They will not be discussed in detail.

- **Refactoring**
- **Pattern verification**

- **Pattern mining**

Refactorings could be used to modify a model to make the application of a pattern easier during a design; however, this is beyond the current scope of the proposal. Similarly, pattern mining and verification functionalities would be very useful for discovering patterns in existing models, and for ensuring that patterns used remain valid – but the complexity required to implement these features automatically is beyond the scope of the proposed prototype.

6.3.3.4 What pattern template should be used?

Choosing a pattern template is not trivial because the majority of patterns that will be added to repositories will be existing patterns, that typically are not constrained to any one pattern template. As a result, a standard template should not be enforced; instead, a list of possible section titles will be provided, from which authors can choose which ones to incorporate, and their ordering. A sample list of section titles is provided below; as well, authors may create their own section titles.

- Intent
- Problem
- Context
- Example
- Resulting Context
- Solution
- Structure
- Behaviour
- Benefits
- Known Uses
- Liabilities
- See Also
- References
- Forces

Each of these sections will be implemented as rich text fields, allowing pattern authors to include diagrams, images, hyperlinks, and other content.

6.3.3.5 How should patterns be classified?

The informal pattern notation is at once a strength and weakness. The lack of constraints on pattern form means that authors can express their patterns the best way they know how; however, this also makes the automated application of patterns difficult. A possible solution to this dilemma is to leave the pattern template informal (see previous subsection), but to require standard meta-data. The following meta-data classification is proposed:

- **Name:** The name of the pattern.
- **Alias(es):** Other names by which the pattern is known, with links if other very similar (or identical) patterns exist in other repositories.
- **Source:** The original source of the pattern (e.g. book, conference proceedings, website, etc.)
- **Pattern Type(s):** e.g. Design, Idiom, Organizational, Process, etc.
- **Repository:** The repository (or repositories) to which the pattern belongs; this is also the name of the pattern language/collection.
- **Next higher-level patterns:** If the pattern is a member of a language, then this entry lists links to the next higher-level patterns (coarser grain patterns that may be implemented immediately before it) in the hierarchy.
- **Next lower-level patterns:** If the pattern is a member of a language, then this entry lists links to the next lower-level patterns (finer grain patterns that may be implemented immediately after it) in the hierarchy.
- **Rose-RT compatible?** If the pattern is Rose-RT compatible, indicate whether it is Structural, Behavioural, or both.
- **Domain(s):** The domains to which the pattern is applicable.
- **Extends:** If this pattern extends (or is a specialization of) another pattern, provide a link to that pattern.
- **Uses:** If the pattern 'uses' other patterns as part of its implementation, provide links to these patterns.
- **Alternatives:** If there are other patterns that provide solutions to a similar problem or set of forces, provide links to these patterns.

These criteria allow applicability and pattern inter-relationships to be explicitly documented.

6.3.3.6 How should pattern implementations be represented?

The tool should place as few constraints on pattern implementation as possible; implementations can be Rose-RT behaviour models and/or structure models, models from other tools with associated source code, or files of any other type. Regardless of the specific components of the implementation, there are some details that must be included:

- **Name:** A unique handle for the implementation.

- **Brief description:** Describe the solution's overall purpose and applicability.
- **List of Components:** List the classes, states, use-cases, etc. included in the implementation.
- **Roles/collaborations:** Describe the roles and collaborations of the classes from each associated structure model or class diagram.
- **Benefits and liabilities:** Discuss the effects this implementation has on the benefits and liabilities discussed in the general pattern text, and any new benefits or liabilities.

From this starting point, Rose-RT structure and/or behaviour models (or files of other types) can be added.

6.3.3.7 How will pattern language members be linked together?

Pattern languages and collections of patterns will be handled the same way in the tool. All patterns belonging to a particular collection/ language will belong to a common repository (specified in the pattern classification) – that may be nested within a larger repository if desired. Pattern relationships (e.g. hierarchy, alias, extends, uses, alternatives) described in the classification section will be implemented as hyperlinks, allowing easy navigation within and between repositories. Thus patterns will be linked hierarchically based on domain relationships and hierarchical inter-pattern relationships, and arbitrarily based on other inter-pattern relationships.

6.3.3.8 How will adding patterns to a model in top-down, bottom-up and mixed fashions be implemented?

Drag-and-drop functionality between the implementation section of the tool and Rose-RT diagrams will facilitate top-down (add complete pattern from repository to a model), bottom-up (assign roles to classes in an existing model), and mixed design using patterns. Developers should be able to drag-and-drop classes or roles from a Rose-RT implementation of a pattern to their model.

6.3.3.9 How can the tool facilitate keeping track of where patterns are used in a model?

When a pattern is added to a model, some meta-data will be added to the documentation section of the classes affected – specifying the roles played in each of the associated patterns. This meta-data can then be used by to assist developers in keeping track of where patterns are used in their models.

This is one area where changes to Rose-RT will be required – so that patterns can be highlighted in a model, rather than requiring that the developer look for the meta-data in the class documentation.

6.3.3.10 What will be published to HTML?

There are two reasons for HTML publishing: (1) to get a printable version of a single pattern and the associated implementations; (2) to make pattern repositories less dependent on particular tools. The HTML documentation for a single pattern includes a title page, the classification information, the body of the pattern, and the associated implementations. Scaling to the entire repository (or set of repositories) in use by an organization requires also generating HTML for index and search pages, and the hyperlinks between patterns.

6.4 Summary

RTOO modeling tools abstract away the low-level details of systems, allowing developers to focus on the design model – precisely the area where design patterns can be useful. The extension to Rose-RT proposed is meant as a basis for a prototype; the complexity of the problem is such that more than one iteration of prototypes will be needed. This proposal is a starting point for that experimentation.

C h a p t e r   7

CHAPTER 7: SUMMARY & DISCUSSION

This thesis has met all of the objectives outlined in Chapter 1: thorough background was presented for design patterns and real-time object-oriented modeling; experience reports from RTOO software practitioners using design patterns were examined; a survey of RTOO design patterns was provided; and a feasible and useful design patterns extension to Rose-RT was proposed. The available documentation shows that patterns have significant potential to assist the design process – yet the nature of the current literature is acting to the movement's detriment. Tool support for patterns is needed to make them more usable.

7.1 Costs & Benefits of Patterns

Chapter 4 discussed practical experiences with patterns; from these experiences, the fundamental benefits, costs, and risks associated with pattern use were extracted and presented here in a consolidated form.

7.1.1 Benefits

The most widely cited benefits gained by using patterns are:

Patterns enable widespread reuse of software architecture: The fundamental benefit to be gained from design patterns is the reuse of ideas, concepts, and strategies. Patterns may also facilitate other types of reuse (for example, patterns can be used to document framework extensions - thereby facilitating code reuse).

Facilitated architectural design: Design patterns make available a set of solutions that work. Rather than designing the required system from scratch, developers can think in terms of an aggregation of existing abstract design components.

Improved communication between (and within) project teams: The improvements in communication come at two levels: (1) when discussing designs, developers can speak in terms of the patterns they wish to use, rather than in terms of the specifics of the implementation; (2) because the patterns are language-independent, experience gained by one project team can potentially be reused by another in a different context. Thus, an organization that has integrated design patterns into its development strategy can expect improved communication because of shared vocabulary and concepts.

Better design documentation: Patterns facilitate design documentation by providing a high level of abstraction.

Since pattern descriptions explicitly enumerate consequences, they serve to record engineering tradeoffs and design alternatives: One of the intrinsic benefits of design patterns is that they not only document good designs (or in the case of AntiPatterns, poor designs), but also why the designs are good (or poor) - and explicitly what consequences result from their implementations.

Patterns explicitly capture knowledge that experienced developers understand implicitly: Because patterns are written by developers knowledgeable in a given area, they provide a means whereby the knowledge gained can be disseminated to others. This leads to many other advantages, including aiding new developers to get up to speed, and eliminating some of the loss of knowledge as team members leave the group.

Documentation of successful software: Design patterns have the potential to accomplish much more than merely document how to design software. Because they are the result of actual experiences, they serve as documentation of successful software projects. While studying patterns trains developers to design quality software, merely reading them has an equally significant benefit - providing an interesting perspective into a successful software project.

Patterns facilitate training of new developers: New team members can be given the set of patterns in use, which they can read to help them get up to speed on the techniques and strategies used in the developed code.

7.1.2 Costs

The costs associated with using patterns include:

Developing and maintaining a list of patterns: Many of the advantages discussed in the previous section (including shared vocabulary, design documentation, and training for new developers in a group) require that all developers in a project team (or organization) share a common set of patterns. This entails that someone (or some group of people) be responsible for selecting which patterns to start with - and over time maintain this list by adding and removing patterns as needs change.

Training users to develop and use these patterns: Once a set of patterns has been chosen, it becomes important to train developers in the proper use of the patterns. Most good patterns are self-explanatory - making the patterns available (hardcopy of online) is thus the fundamental first-step. Ongoing support via email, Intranet, CBTs, or in-person training adds to the cost, but also increases the likelihood of having the patterns understood and applied consistently.

7.1.3 Risks

The risks that could keep pattern benefits out of reach include:

Developers resist the change: As with any change in strategy, the introduction of design patterns may see some resistance from those people expected to learn and use them. In practice, it is developers who see the value of this medium and who have been promoting its use.

Design patterns are not used consistently: The crux of this issue is that all developers must understand the patterns in use by their development team or organization. Without this shared pattern subset, the potential advantages of having a shared design vocabulary and consistent solutions to recurring problems are not gained.

Pattern quality is not maintained: With at least four major conferences a year attracting pattern authors, and a steady stream of publications, the number of design patterns is growing rapidly. The original GoF patterns were of a high quality; there is short term potential for this quality to be lost as the patterns community is flooded with new additions. Regardless the value of the patterns, keeping up with all of the additions is very time-consuming.

The benefits to be gained from patterns are significant (particularly for project teams or organizations developing large-scale software); however, the costs and risks can significantly reduce these benefits – possibly even negating them.

7.2 Available RTOO Patterns

Chapter 5 described design patterns dealing with a variety of issues relevant to developers of RTOO software. By extracting and organizing the patterns, they were made easier to learn and use.

7.2.1 Current State of the Literature

The process of discovering, organizing, and presenting the patterns in Chapter 5 led to two observations about the current state of the literature:

- **Lack of Organization:** Finding relevant patterns can be difficult, since there are no central pattern listings. The patterns from Chapter 5 came from books, journal articles, conference proceedings, and the Internet.
- **Rapid Growth:** The volume of available patterns is growing very rapidly; many are duplicates of others, because it is difficult to keep up with all the published and work-in-progress patterns.

Tool support for patterns attempts to resolve these problems – by collecting patterns and logically linking them together. Another effort that may help is Linda Rising's forthcoming book The Patterns Almanac 2000 [Ris00], which lists and describes all available software patterns as of fall 1999.

7.2.2 Pattern Quality

Quality varies considerably between patterns; though difficult to measure empirically, there are characteristics that 'good' patterns should possess (determined subjectively):

- **Should explicitly describe context, problem, solution, and resulting context:** Although there exist many pattern templates, the minimal set of information required to make something a 'pattern' is generally agreed (based on every available pattern definition and template) to be context, problem, solution, and resulting context. These sections should be explicitly documented.

- **Should provide a good solution to the problem:** The pattern should describe a good solution to the problem described, and should describe the benefits and liabilities associated with choosing to implement this solution.
- **Should be clear and easy to understand:** The context and problem should be described in sufficient detail to make it easy to understand when the pattern is applicable and when it is not, and the solution description sufficiently clear to implement.
- **Should explicitly describe relationships with other patterns:** Though useful individually, the real strength of patterns comes when they are used together. These relationships should be described explicitly.
- **Should be the result of experience:** Many pattern definitions include a *rule of three* clause - that something should not be called a 'pattern' unless it has been seen three times, in three different applications. This helps to ensure that the forces being resolved by the pattern are real and that the solution is valid.
- **Should provide examples:** The pattern should describe implementation examples in detail, including diagrams and source code.

The current model for achieving pattern quality is the shepherding process (matching the author of a new pattern with a seasoned expert in the patterns field) coupled with pattern writers' workshops (borrowed from poetry, where groups of authors get together to discuss new works). These processes work well, but are not applied consistently. A more sophisticated scheme has been proposed by the Object Technology Group at Clemson University, who are examining 'Pattern Juries' [Mil00] - which provide guidelines for selecting jurors, and for the subjective determination of pattern quality.

7.3 Convergence of Tools for RTOO Modeling and Design Patterns

Chapter 6 discussed the potential for convergence between RTOO modeling tools (e.g. [Rat00c, Obj00, iLo00a, iLo00b, Tel00) and Design Pattern tools (e.g. [KB95, Mei96, Bro96, Wik00, Pat00, Blu00, BF+96]). As a concrete example, an extension to Rational Rose-RT was proposed to support patterns.

Initially, the goal of the proposed tool was to extend pattern repository functionality to support Rose-RT models (that can be used to capture both the structure and the underlying source code for pattern implementations). However, as the research progressed it became increasingly apparent that tool support can also be beneficial for describing the inter-relationships between patterns. The proposal thus included repository functionality, and pattern grouping, organization and relationship functionality. The result is a proposal for a tool that not only makes the application of individual patterns easier and more explicit, but also encourages the systematic application of multiple related patterns.

7.4 Future Research

Although the objectives of this thesis were met, the magnitude of the problem tackled means that many issues remain outstanding.

7.4.1 Tool Implementation

This thesis has demonstrated the applicability of patterns to RTOO software design, and has recommended extensions to Rational Rose-RT to support design patterns as an abstraction layer. Though a significant challenge, the refinement and implementation of this proposal is the logical next step.

7.4.2 Other Pattern Types

This thesis has focused on design patterns; however, there are other pattern types that can be equally useful to the design of RTOO software. The foundations proposed are equally suitable for organizational, analysis, process, and other pattern types.

7.4.3 Inter-pattern Relationships

Though useful individually, the real strength of patterns comes when multiple patterns are used together to resolve forces in a problem space. This systematic application requires that authors explicitly document the relationships between their patterns and other related patterns. Although many authors document these relationships within their own pattern languages or collections – very few consider the relationships with patterns outside their authorship. The explicit documentation of inter-pattern relationships represents an open opportunity for the patterns community.

7.5 Concluding Remarks

This thesis has presented at least three significant contributions: First, the survey of experience reports from the RT domain has shown the potential benefits, costs, and risks associated with pattern use; Second, the survey of RTOO design patterns has demonstrated the breadth and depth of available patterns, and has also exposed the disorganization in the current literature; Third, an extension of Rational Rose-RT to support design patterns was proposed, describing how patterns may be used as an abstraction layer in design models.

BIBLIOGRAPHY

[AC+96] Adams, M., J. Coplien, R. Gamoke, R. Hanmer, F. Keeve, and K. Nicodemus. "Fault-tolerant Telecommunications Patterns" in Vlissides, J.M., J.O. Coplien, and N.L. Kerth. (eds.) *Pattern Languages of Program Design 2*. Addison-Wesley, Reading, Mass. 1996.

[AC98] Agerbo, E. and A. Cornils. "How to Preserve the Benefits of Design Patterns." In *Object-Oriented Programming, Systems, Languages, and Applications Conference Proceedings*. 1998

[Ale99] Alexander, C.A. "The Origins of Pattern Theory: The Future of the Theory, and the Generation of a Living World." *IEEE Software Special Issue on Architecture Design*, 16(5): 71--82, September 1999.

[AS+75] Alexander, C., M. Silverstein, S. Angel, & D. Abrams. *The Oregon Experiment*. Oxford University Press, New York, 1975.

[AIS77] Alexander, C., S. Ishikawa, and M. Silverstein. *A Pattern Language*. Oxford University Press, New York, 1977.

[Ale79] Alexander, Christopher. *The Timeless Way of Building*. Oxford University Press, New York, 1979.

[Amb00] Ambler, S.W. "The Process Patterns Resource Page." http://www.ambysoft.com/processPatternsPage.html. Available 2000.

[Amb98] Ambler, S.W. *Process Patterns: Building Large-Scale Systems Using Object Technology*. Cambridge University Press/SIGS Books, 1998.

[Amb99] Ambler, S.W. *More Process Patterns: Delivering Large-scale Systems Using Object Technology*. Cambridge University Press/SIGS Books, 1999.

[App89] Apple Computer, Inc. *Macintosh Programmers Workshop Pascal 3.0 Reference*. 1989.

[Bec96] Beck, K. *Smalltalk Best Practice Patterns*. Prentice Hall. 1996.

[BC87] Beck, K., and W. Cunningham. "Using Pattern Languages for Object-oriented Programs." In *Object-Oriented Programming, Systems, Languages, and Applications Conference Proceedings*. 1987.

[BC+96] Beck, K., R. Crocker, G. Meszaros, J. Vlissides, J.O. Coplien, L. Dominick and F. Paulisch. "Industrial experience with design patterns," In *Proceedings of the 18th International Conference on Software Engineering*. 103 – 114. 1996.

[Blu00] Blueprint Technologies Framework Studio Toolset. http://blueprint-technologies.com. Available 2000.

[Boo94] Booch, G. *Object-oriented Analysis and Design with Applications (Second Edition)*. Benjamin/Cummings, Redwood City, CA, 1994.

[BRJ99] Booch, G., J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, Mass. 1999.

[Bot99] Bottomley, M. "A Pattern Language for Simple Embedded Systems". In *Proceedings of the Sixth Annual Pattern Languages of Programming Conference*. 1999.

[Bro87] Brooks, F.P. Jr. "No Silver Bullet: Essence and Accidents of Software Engineering." *IEEE Computer*, 20, 10-19, April 1987.

[Bro96] Brown, K.G. "Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk." *MSc. Thesis*. North Carolina State University, Department of Computer Science. 1996.

[BM+98] Brown, W. J., R. C. Malveau, H. W. McCormick III, and T. J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc. New York, 1998.

[BF+96] Budinsky, F.J., M.A. Finnie, J.M. Vlissides, and P.S. Yu. "Automatic code generation from design patterns." *IBM Systems Journal*. 35, 2, 1996.

[BM+96] Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System Of Patterns*. John Wiley & Sons Ltd. Sussex, England. 1996.

[Bus98] Buschmann, F. "Real-time Constraints as Strategies," In *Proceedings of the Third Annual European Pattern Languages of Programming Conference*. 1998.

[But97] Buttazzo, G.C. *Hard Real-time Computing Systems, Predictable Scheduling Algorithms and Applications*. Kluwer, USA. 1997.

[Cen00] Center for Distributed Object Computing at Washington University Home Page. http://www.cs.wustl.edu/~schmidt/doc-center.html. Available 2000.

[CHI00] CHILL Home Page. http://www1.informatik.uni-jena.de/languages/chill/chill.htm. Available 2000.

[Cli96] Cline, M.P. "The Pros and Cons of Adopting and Applying Design Patterns in the Real World." *Communications of the ACM*, October 1996, 39, 10, 47-49.

[CNM95] Coad, P., D. North and M. Mayfield. *Object Models: strategies, patterns and applications*, Prentice Hall, Englewood Cliffs, 1995.

[Cop92] Coplien, J.O. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, Reading, Mass. 1992.

[Cop00] Coplien, J.O. "Organizational Patterns Home Page." http://www.bell-labs.com/cgi-user/OrgPatterns/OrgPatterns?OrganizationalPatterns. Available 2000.

[CS95] Coplien, J.O., D.C. Schmidt (eds.). *Pattern Languages of Program Design*. Addison-Wesley, Reading, Mass. 1995.

[DK96] Dagermo, P. and J. Knutsson. "Development of an Object-oriented Framework for Vessel Control Systems." Technical Report, ESPRIT III/ ESSI/ DOVER #10496. 1996.

[DeB00] DeBruler, D.L. "Telecommunications distributed processing patterns." http://www.bell-labs.com/people/cope/Patterns/DistributedProcessing/DeBruler/index.html. Available 2000.

[deC96] de Champlain, M. "Patterns to ease the port of micro-kernels in embedded systems." In *Proceedings of the Third Annual Pattern Languages of Programming Conference*. 1996.

[Dou97] Douglass, B.P. *Real-Time UML: Efficient Objects for Embedded Systems*. Addison-Wesley, Reading, Mass. 1997.

[Dou99] Douglass, B.P. *Doing Hard Time: Developing Real-time Systems with UML, Objects, Frameworks, and Patterns*. Addison-Wesley, Reading, Mass. 1999.

[DSS00] DSSA ADAGE Home Page. http://www.owego.com/dssa. Available 2000.

[Emb00] Embedded C++ Home Page. http://www.caravan.net/ec2plus. Available 2000.

[FP88] Faulk, S.R., Parnas, D.L., "On Synchronization in Hard-Real-Time Systems", *Communications of the ACM, 31, 3*, March 1988, pp. 274-287.

[Fow97] Fowler, M. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, Reading, Mass. 1997.

[Fow99a] Fowler, M. "Is there such a thing as object-oriented analysis?" *Distributed Computing*, October 1999.

[Fow99b] Fowler, M. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Reading, Mass. 1999.

[GH+95] Gamma, E., R. Helm, R. Johnson & J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995.

[Gra97] Grasso, E. "Synchronizer: an Object Behavioural Pattern for Concurrent Programming." In *Proceedings of the Second European Conference on Pattern Languages of Programming*, Kloster Irsee, Germany, 10-12/07/1997. 1997.

[Gre00] Green Hills Software Home Page. http://www.ghs.com. Available 2000.

[Har87] Harel, D. "Statecharts: A visual formalism for complex systems." *Science of Computer Programming, 8*, 231-274. 1987.

[HSP96] Harrison, T.H., D.C. Schmidt, and I. Pyarali. “Asynchronous Completion Token: An Object Behavioural Pattern for Efficient Asynchronous Event Handling.” In *Proceedings of the Third Annual Pattern Languages of Programming Conference*. 1996.

[HFR99] Harrison, N., B. Foote, H. Ronnert (eds.). *Pattern Languages of Program Design 4*. Addison Wesley, Reading, Mass. 1999.

[Hay96] Hay, D. *Data model patterns: conventions of thought*. Dorset House, New York, NY, 1996.

[Hel95] Helm, R. “Patterns in Practice.” In *Object-Oriented Programming, Systems, Languages, and Applications Conference Proceedings*. 1995.

[HHG90] Helm, R., I.M. Holland, and D. Gangopadhyay. “Contracts: Specifying Behavioural Compositions in Object-oriented Software.” In *Object-Oriented Programming, Systems, Languages, and Applications Conference Proceedings*. Ottawa, Canada, 1990.

[Hil00] Hillside Group. The Patterns Home Page. http://www.hillside.net/patterns. Available 2000.

[Hol92] Holland, I.M. “Specifying Reusable Components Using Contracts,” in *Proceedings of ECOOP ’92*, Lecture Notes in Computer Science #615, Springer-Verlag, 1992.

[iLo00a] i-Logix Rhapsody Toolset.. http://www.i-Logix.com. Available 2000.

[iLo00b] i-Logix Statemate MAGNUM Toolset.. http://www.i-Logix.com. Available 2000.

[IEE96] IEEE/ANSI. *Std 1003.1, Portable Operating System Interface (POSIX)*. 1996.

[IPS96] Islam, N., A. Prodromidis, and M.S. Squillante. Dynamic partitioning in different distributed-memory environments. In *Proceedings of the 2nd Workshop on Job Scheduling Strategies for Parallel Processing*. April 1996. pp. 26-49.

[ID96] Islam, N., and M. Devarakonda. “An Essential Design Pattern for Fault-Tolerant Distributed State Sharing.” *Communications of the ACM*, 39, 10 (October 1996), pp. 65-74.

[ISO87] ISO/IEC 8652 :1987. *Ada Reference Manual : Language and Standard Libraries*, Intermetrics, Inc. 1987.

[ISO95] ISO/IEC 8652 :1995. *Ada Reference Manual : Language and Standard Libraries, Version 6.0*, Intermetrics, Inc. 1995.

[ISO99] ISO/IEC 9899:1999. *Programming Languages – C*. 1999.

[ISO98] ISO/IEC 14882:1998. *Programming Languages – C++*. 1998.

[ITU93] ITU-T. *Recommendation Z.100: Specification and Description Language (SDL)*, 3/93.

[ITU96] ITU-T. *Recommendation Z.120: Message sequence chart (MSC)*, 10/96.

[JC+92] Jacobson, I., M. Christerson, P. Jonsson, and G. Overgaard. *Object-oriented Software Engineering - A Use Case Driven Approach*. Addison-Wesley, Wokingham, England, 1992.

[JPA99] Jiménez-Peris, R. M. Patiño-Martínez, and S. Arévalo. “Multithreaded Rendezvous: A Design Pattern for Distributed Rendezvous.” In *Proceedings of the Symposium on Applied Computing*. 1999.

[Joh94] Johnson, R.E. “Why a conference on pattern languages?” *Software Engineering Notes*, 19(1): 50-52, January 1994.

[Kar95] Karlsson, E.A. (ed.). *Software Reuse - A Holistic Approach*. John Wiley & Sons. 1995.

[KB95] Kim, J.J., and K.M. Benner. “A Design Patterns Experience: Lessons Learned and Tool Support.” In *Proceedings of ECOOP ‘95*. 1995.

[LV97] Lai, J. and J. Vergo. “MedSpeak: Report Creation with Continuous Speech Recognition.” *Proceedings of CHI ‘97*, ACM Press, New York, 1997, pp. 431-438.

[LS96] Lavender, R.G. and D.C. Schmidt. “Active Object: An Object Behavioural Pattern for Concurrent Programming.” in Vlissides, J.M., J.O. Coplien, and N.L. Kerth. (eds.) *Pattern Languages of Program Design 2*. Addison-Wesley, Reading, Mass. 1996.

[Lea00] Lea, D. “Design Patterns for Avionics Control Systems.” http://g.oswego.edu/dl/acs/acs/acs.html. Available 2000.

[LC+92] Linton, M.A., P. Calder, J. Interrante, S. Tang, and J. Vlissides. *Interviews Reference Manual*. CSL, Stanford University, 3.1 Edition, 1992.

[LVC89] Linton, M.A., J.M. Vlissides, and P.R. Calder. "Composing User Interface with InterViews." *IEEE Computer*, 22(2): 8-22, February 1989.

[Mad00] Madapusi, Bhadrinarayanan. "Automated Identification of Design Patterns", *MSc. Thesis*. Department of Computing & Information Sciences, Queen's University. To Appear 2000.

[MRB97] Martin, R.C., D. Riehle, and F. Buschmann (eds.). *Pattern Languages of Program Design 3*. Addison-Wesley, Reading, Mass. 1997.

[Mei96] Meijers, M. "Tool Support for Object-oriented Design Patterns." *M.Sc. Thesis*. Department of Computer Science, Ultrecht University, The Netherlands. 1996.

[Mes96] Meszaros, G. "A Pattern Language for Improving the Capacity of Reactive Systems" in Vlissides, J.M., J.O. Coplien, and N.L. Kerth. (eds.) *Pattern Languages of Program Design 2*. Addison-Wesley, USA, 1996.

[Mic00] Microsoft Corporation Home Page. http://www.microsoft.com. Available 2000.

[Mil00] Miller, T. "Juries for Design Pattern Validation Home Page". http://www.cs.clemson.edu/~tmiller/jury. Available 2000.

[MDK91] Mohindra, A., M. Devarakonda, and B. Kish. "Recovery in the Calypso file system." *ACM Transactions on Computing Systems. 14, 3* (August 1991), 52-60.

[Mol96] Molin, P. and L. Ohlsson. "Points & Deviations - A pattern language for fire alarm systems." In *Proceedings of the Third Annual Pattern Languages of Programming Conference*. 1996.

[NW98] Noble, J., & C. Weir. "Proceedings of the Memory Preservation Society." In *Proceedings of the Third Annual European Pattern Languages of Programming Conference*. 1998.

[Obj00] Objectime Developer 5.2 Toolset. http://www.objectime.com. Available 2000.

[Opd92] Opdyke, W.F. "Refactoring Object-oriented Frameworks." *Ph.D. Thesis*. Department of Computer Science, University of Illinois, Urbana, Illinois, 1992.

[Par90] ParcPlace Systems, Mountain View, CA. *ObjectWorks/Smalltalk Release 4 Users Guide*, 1990.

[PJA98] Patiño-Martínez, M., R. Jiménez-Peris, and S. Arévalo. "Integrating Groups and Transactions: A Fault-Tolerant Extension of Ada." In Asplund, L. (ed.), *Proceedings of the International Conference on Reliable Software Technologies, Ada-Europe '98*, volume LNCS 1411, pp 78-89. Springer, June 1998.

[Pat00] Pattern Depot. http://www.patterndepot.com. Available 2000.

[Por00] Portland Pattern Repository. http://c2.com/ppr. Available 2000.

[Pra00a] Praxis Critical Systems Ltd. "SPARK Home Page." http://www.praxis-cs.co.uk/products/sparkhome.html. Available 2000.

[Pra00b] Praxis Critical Systems Ltd. "SPARK Examiner Home Page." http://www.praxis-cs.co.uk/products/examiner.html. Available 2000.

[Pre95] Pree, W. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, Reading, Mass. 1995.

[PS97] Petriu, D. & G. Somadder. "A Pattern Language for Improving the Capacity of Layered Client/Server Systems with Multi-Threaded Servers." In *Proceedings of the Second Annual European Pattern Languages of Programming Conference*. 1997.

[Poo87] Pooley, R.J. *An Introduction to Programming in SIMULA*. Blackwell Scientific Publications, Oxford, 1987.

[PH+97] Pyarali, I., T. Harrison, D.C. Schmidt, T.D. Jordan. "Proactor: An Object Behavioural Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events." In *Proceedings of the Fourth Annual Pattern Languages of Programming Conference*. 1997.

[Rat00a] Rational RequisitePro Toolset. http://www.rational.com/products/reqpro. Available 2000.

[Rat00b] Rational Rose Toolset. http://www.rational.com/products/rose. Available 2000.

[Rat00c] Rational Rose-Realtime Toolset. http://www.rational.com/products/rosert. Available 2000.

[RTJ00] Real-time Java Specification Home Page. http://www.rtj.org. Available 2000.

[Ref00] Refactoring Browser. http://st-www.cs.uiuc.edu/users/brant/Refactory/RefactoringBrowser.html. Available 2000.

[Ris00] Rising, L. *The Patterns Almanac 2000*. Addison-Wesley, Reading, Mass. To Appear 2000.

[RB+91] Rumbaugh, J. "The Life of an object model: How the object model changes during development." *Journal of Object-oriented Programming*, 7(1): 24-32, March/April 1994.

[Sch00] Schmidt, D.C. "Patterns for Concurrent, Parallel, and Distributed Systems Home Page". http://www.cs.wustl.edu/~schmidt/patterns-ace.html. Available 2000.

[Sch95a] Schmidt, D.C. "An Object Behavioural Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching." In Coplien, J.O. and D.C. Schmidt (eds.) *Pattern Languages of Program Design*. Addison-Wesley, Reading, Mass. 1995.

[Sch95b] Schmidt, D.C. "Using Design Patterns to Develop Reusable Object-oriented Communication Software." *Communications of the ACM*, October 1995, 38, 10, 65-74.

[Sch95c] Schmidt, D.C. "Acceptor and Connector: Design patterns for active and passive establishment of network connections." In *Proceedings of ECOOP '95*, (Aarhus, Denmark), August 1995.

[Sch99a] Schmidt, D.C. "Object Lifetime Manager – A Complementary Pattern for Controlling Object Creation and Destruction." In *Proceedings of the Fifth Pattern Languages of Programming Conference*, Allerton Park, Illinois, USA, August 1999.

[Sch95d] Schmidt, D.C. "Reactor: An Object Behavioural Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching." In Coplien, J.O., D.C. Schmidt (eds.). *Pattern Languages of Program Design*. Addison-Wesley, Reading, Mass. 1995.

[Sch99b] Schmidt, D.C. "Strategized Locking, Thread-Safe Decorator, and Scoped Locking: Patterns and Idioms for Simplifying Multi-Threaded C++ Components", *C++ Report*, SIGS, 11, 9, September 1999.

[SC95] Schmidt, D.C., and C.D. Cranor. "Half-Sync/Half-Async: An architectural pattern for efficient and well-structured concurrent I/O." In *Proceedings of the Second Annual Conference on the Pattern Languages of Programs*, (Monticello, Illinois, Sept. 1995), pp. 1-10.

[SH96] Schmidt, D.C. and T. Harrison. "Double-checked Locking – An Optimization Pattern for Efficiently Initializing and Accessing Thread-safe Objects." In *Proceedings of the Third Annual Pattern Languages of Programming Conference*, Allerton Park, Illinois, September 4-6, 1996.

[SHP97] Schmidt, D.C., T. Harrison, and N. Pryce. "Thread-Specific Storage – An Object Behavioural Pattern for Accessing per-Thread State Efficiently." In *Proceedings of the Fourth Annual Pattern Languages of Programming Conference*, Allerton Park, Illinois, September 2-5, 1997.

[SJ97] Schmidt, D.C. and P. Jain. "A Pattern for Dynamic Configuration of Services." *C++ Report, SIGS*, 9, 6, June 1997.

[SS95] Schmidt, D.C., and P. Stephenson. "Experience Using Design Patterns to Evolve Communication Software Across Diverse OS Platforms." *Proceedings of the Ninth European Conference on Object-oriented Programming*, Aarhus, Denmark, 1995.

[Sch97] Schuderer, C. "REFORM Entwurfsdokumentation." Siemens in-house design document. 1997.

[Sel96] Selic, B. "An Architectural Pattern for Real-time Control Software." In Vlissides, J.M., J.O. Coplien, and N.L. Kerth. (eds.) *Pattern Languages of Program Design 2*. Addison-Wesley, USA, 1996.

[SGW94] Selic, B., G. Gullekson, and P.T. Ward. *Real-time Object-oriented Modeling*. John Wiley & Sons Inc., USA, 1994.

[SR98] Selic, B., and J. Rumbaugh. "Using UML for Modeling Complex Real-Time Systems." http://www.objectime.com/otl/technical/umlrt.pdf. 1998. (Available 2000).

[Sri99] Srinivasan, S. "Design Patterns in Object-Oriented Frameworks." *IEEE Computer*. Feb. 1999, pp.24-32.

[Str91] Stroustroup, B. *The C++ Programming Language: 2nd Edition*. Addison-Wesley. 1991.

[SHH91] Stoyenko, A.D., C. Hamacher, and R.C. Holt. "Analyzing Hard-Real-Time Programs For Guaranteed Schedulability." *IEEE Transactions on Software Engineering*, 17, 8 (Aug 1991), 737-750.

[Sun95] Sun Microsystems Inc. "The Java Language Environment: A White Paper." 1995.

[Sym93] Symantec Corporation. *Bedrock Developer's Architecture Kit*. 1993.

[Sys00] Systems and Software Research Center Home Page at Bell Laboratories. http://www.bell-labs.com/org/1135. Available 2000.

[Tel00] Telelogic Tao Toolset. http://www.telelogic.com. Available 2000.

[VCK96] Vlissides, J.M., J.O. Coplien, N.L. Kerth (eds.). *Pattern Languages of Program Design 2*. Addison Wesley, Reading, Mass. 1996.

[Wal98] Walls, C. "Java's Role in the Embedded World". *Real-Time Magazine*, 1, 1998.

[Web00] Web ProForum. Specification and Description Language (SDL) Tutorial. http://www.webproforum.com/sdl/. Available 2000.

[WGM88] Weinard, A., E. Gamma, and R. Marty. "ET++ - An object-oriented application framework in C++". In *Object-Oriented Programming, Systems, Languages, and Applications Conference Proceedings*, pp. 46-57, San Diego, CA, Sept. 1988.

[Wik00] Wiki Wiki Web. http://c2.com/cgi/wiki?WikiWikiWeb. Available 2000.

[Win97] Winston, P.H. *On to Smalltalk*. Addison-Wesley. Reading, Mass. 1997.

[Woo96] Woodward, K.G. "Heading off Tragedy: Using Design Patterns Against a Moving Target." *Proceedings of the Second World Conference on Integrated Design and Process Technology*, 1996.

APPENDIX A: ALEXANDER'S WINDOW PLACE PATTERN

The full text of Christopher Alexander's Window Place pattern is provided below (from [AIS77]):

180 WINDOW PLACE **

This pattern helps complete the arrangement of the windows given by ENTRANCE ROOM (130), ZEN VIEW (134), LIGHT ON TWO SIDES OF EVERY ROOM (159), STREET WINDOWS (164). According to the pattern, at least one of the windows in each room needs to be shaped in such a way as to increase its usefulness as a space.

Everybody loves window seats, bay windows, and big windows with low sills and comfortable chairs drawn up to them.

It is easy to think of these kinds of places as luxuries, which can no longer be built, and which we are no longer lucky enough to be able to afford.

In fact, the matter is more urgent. These kinds of windows which create "places" next to them are not simply luxuries; they are *necessary*. A room which does not have a place like this seldom allows you to feel fully comfortable or perfectly at ease. Indeed, a room without a window place may keep you in a state of perpetual unresolved conflict and tension – slightly, perhaps, but definite.

This conflict takes the following form. If the room contains no window which is a "place", a person in the room will be torn between two forces:

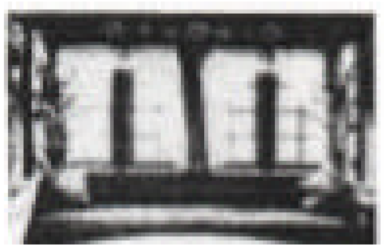
1. He wants to sit down and be comfortable.

2. He is drawn toward the light.

Obviously, if the comfortable places – those places in the room where you most want to sit – are away from the windows, there is no way of overcoming this conflict. You see, then, that our love for window "places" is not a luxury but an organic intuition, based on the natural desire a person has to let the forces he experiences run free. A room where you feel truly comfortable will always contain some kind of window place.

Now, of course, it is hard to give an exact definition of a "place". Essentially a "place" is partly enclosed, distinctly identifiable spot within a room. All of the following can function as "places" in this sense: bay windows, window seats, a low window sill where there is an obvious position for a comfortable armchair, and deep alcoves with windows all around them. To make the concept of a window place more precise, here are some examples of each of these types, together with discussion of the critical features which make each of them work.

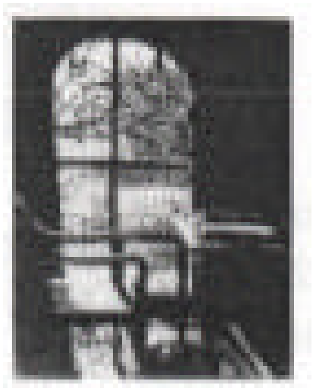
A bay window. A shallow bulge at one end of a room, with windows wrapped around it. It works as a window place because of the greater intensity of light, the views through the side windows, and the fact that you can pull chairs or a sofa up to the bay.


Bay window

A window seat. More modest. A niche, just deep enough for the seat. It works best for one person, sitting parallel to the window, back to the window frame, or for two people facing each other in this position.

Window Seat

A low sill. The most modest of all. The right sill height for a window place, with acomfortable chair, is very low: 12 to 14 inches. The feeling of enclosure comes from the armchair – best of all, one with a high back and sides.

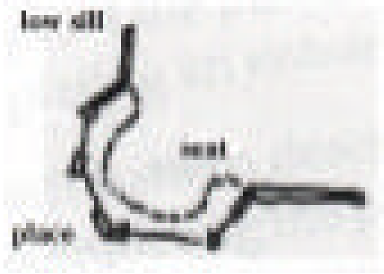

Low Sill

A glazed alcove. The most elaborate kind of window place: almost like a gazebo or a conservatory, windows all around it, a small room, almost part of the garden.

Glazed alcove

And, of course, there are other possible versions too. In principle, any window with a reasonably pleasant view can be a window place, provided that it is taken seriously as a space, a volume, not merely treated as a hole in the wall. Any room that people use often should have a window place. And window places should even be considered for waiting rooms or as special places along the length of hallways. Therefore:

In every room where you spend any length of time during the day, make at least one window into a "window place".



Make it low and self-contained if there is room for that – ALCOVES (179); keep the sill low- LOW SILL (222); put in the exact positions of frames, and mullions, and seats after the window place is framed, according to the view outside – BUILT-IN SEATS (202), NATURAL DOORS AND WINDOWS (221). And set the window deep into the wall to soften light around the edges – DEEP REVEALS (223). Under a sloping roof, use DORMER WINDOWS (231).

APPENDIX B: GANG OF FOUR DECORATOR PATTERN

The full text of the Gang of Four Decorator pattern is provided below (from [GH+95]):

DECORATOR (Object Structural)

Intent

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Also Known As

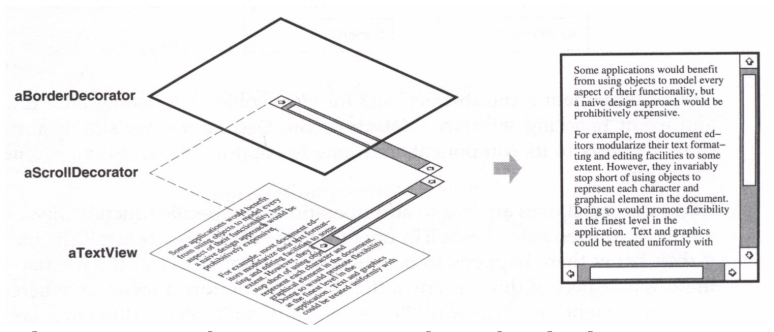
Wrapper

Motivation

Sometimes we want to add responsibilities to individual objects, not to an entire class. A graphical user interface toolkit, for example, should let you add properties like borders or behaviours like scrolling to any user interface component.

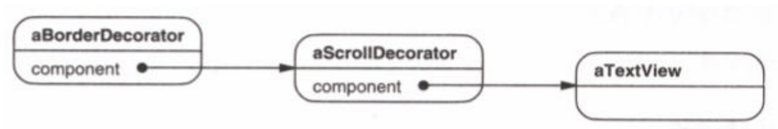
One way to add responsibilities is with inheritance. Inheriting a border from another class puts a border around every subclass instance. This is inflexible, however, because the choice of border is made statically. A client can't control how and when to decorate the component with a border.

A more flexible approach is to enclose the component in another object that adds the border. The enclosing object is called a **decorator**. The decorator conforms to the interface of the component it decorates so that its presence is transparent to the component's clients. The decorator forwards requests to the component and may perform additional actions (such as drawing a border) before or after forwarding. Transparency lets you nest decorators recursively, thereby allowing an unlimited number of added responsibilities.

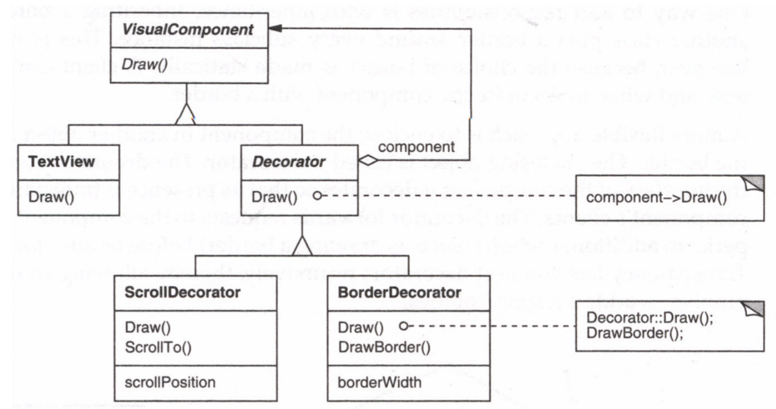

aBorderDecorator
aScrollDecorator
aTextView
Some applications would benefit from using objects to model every aspect of their functionality, but a naive design approach would be prohibitively expensive.
For example, most document editors modularize their text formatting and editing facilities to some extent. However, they invariably stop short of using objects to represent each character and graphical element in the document. Doing so would promote flexibility at the finest level in the application. Text and graphics could be treated uniformly with

For example, suppose we have a TextView object that displays text in a window. TextView has no scroll bars by default, because we might not always need them. When we do, we can use a ScrollDecorator to add them. Suppose we also want to add a thick black border around the TextView. We can use a BorderDecorator to add this as well. We simply compose the decorators with the TextView to produce the desired result.

The following object diagram shows how to compose a TextView object with BorderDecorator and ScrollDecorator objects to produce a bordered, scrollable text view:



The ScrollDecorator and BorderDecorator classes are subclasses of Decorator, an abstract class for visual components that decorate other visual components.



VisualComponent is the abstract class for visual objects. It defines their drawing and event handling interface. Note how the Decorator class simply forwards draw requests to its component, and how Decorator subclasses can extend this operation.

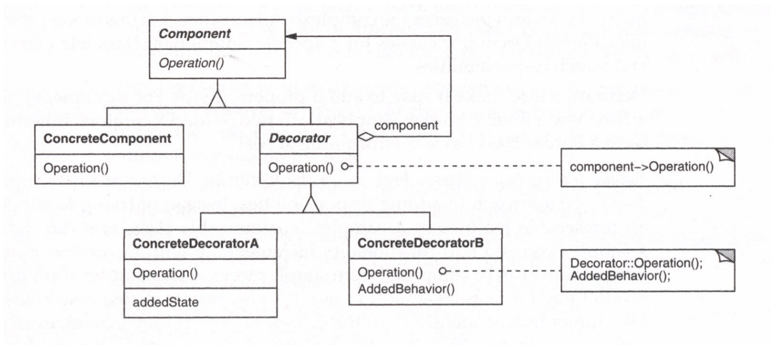
Decorator subclasses are free to add operations for specific functionality. For example, ScrollDecorator's ScrollTo operation lets other objects scroll the interface if they know there happens to be a ScrollDecorator object in the interface. The important aspect of this pattern is that it lets decorators appear anywhere a VisualComponent can. That way clients generally can't tell the difference between a decorated component and an undecorated one, and so they don't depend at all on the decoration.

Applicability

Use decorator

- To add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
- For responsibilities that can be withdrawn.
- When extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination. Or a class definition may be hidden or otherwise unavailable for subclassing.

Structure



Participants

- **Component** (VisualComponent): Defines the interface for objects that can have responsibilities added to them dynamically.
- **ConcreteComponent** (TextView): Defines an object to which additional responsibilities can be attached.
- **Decorator:** Maintains a reference to a Component object and defines an interface that conforms to the Component's interface.
- **ConcreteDecorator** (BorderDecorator, ScrollDecorator): Adds responsibilities to the component.

Collaborations

Decorator forwards requests to its Component object. It may optionally perform additional operations before and after forwarding the request.

Consequences

The Decorator pattern has at least two key benefits and two liabilities:

1. *More flexibility than static inheritance*. The Decorator pattern provides a more flexible way to add responsibilities to objects than can be had with static (multiple) inheritance. With decorators, responsibilities can be added and removed at run-time simply by attaching and

detaching them. In contrast, inheritance requires creating a new class for each additional responsibility (e.g., BorderedScrollableTextView, BorderedTextView). This gives rise to many classes and increases the complexity of a system. Furthermore, providing different Decorator classes for a specific Component class lets you mix and match responsibilities. Decorators also make it easy to add a property twice. For example, to give a TextView a double border, simply attach two BorderDecorators. Inheriting from a Border class twice is error-prone at best.

2. *Avoids feature-laden classes high up in the hierarchy*. Decorator offers a pay-as-you-go approach to adding responsibilities. Instead of trying to support all foreseeable features in a complex, customizable class, you can define a simple class and add functionality incrementally with Decorator objects. Functionality can be composed from simple pieces. As a result, an application needn't pay for features it doesn't use. It's also easy to define new kinds of Decorators independently from the classes of objects they extend, even for unforeseen extensions. Extending a complex class tends to expose details unrelated to the responsibilities you're adding.

3. *A decorator and its components aren't identical*. A decorator acts as a transparent enclosure. But from an object identity point of view, a decorated component is not identical to the component itself. Hence you shouldn't rely on object identity when you use decorators.

4. *Lots of little objects*. A design that uses Decorator often results in systems composed of lots of little objects that all look alike. The objects differ only in the way they are interconnected, not in their class or in the value of their variables. Although these systems are easy to customize by those who understand them, they can be hard to learn and debug.

Implementation

Several issues should be considered when applying the Decorator pattern:

1. *Interface conformance*. A decorator object's interface must conform to the interface of the component it decorates. ConcreteDecorator classes must therefore inherit from a common class (at least in C++).

2. *Omitting the abstract Decorator class*. There's no need to define an abstract Decorator class when you only need to add one responsibility. That's often the case when you're dealing with an existing class hierarchy rather than designing a new one. In that case, you can merge Decorator's responsibility for forwarding requests to the component into the ConcreteDecorator.

3. *Keeping Component classes lightweight*. To ensure a conforming interface, components and decorators must descend from a common Component class. It's important to keep this common class lightweight; that is, it should focus on defining an interface, not on storing data. The definition of the data representation should be deferred to subclasses; otherwise the complexity of the Component class might make the decorators too heavyweight to use in quantity. Putting a lot of functionality into Component also increases the probability that concrete subclasses will pay for features they don't need.

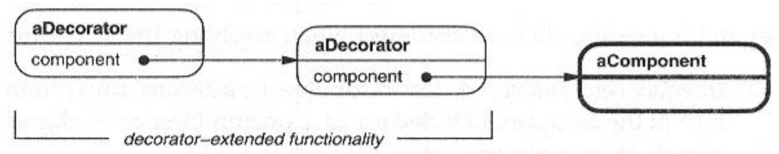
4. *Changing the skin of an object versus changing its guts*. We can think of a decorator as a skin over an object that changes its behaviour. An alternative is to change the object's guts. The Strategy pattern is a good example of a pattern for changing the guts. Strategies are a better choice in situations where the Component class is intrinsically Heavyweight, thereby making the Decorator pattern too costly to apply. In the Strategy pattern, the component forwards some of its behaviour to a separate strategy object. The Strategy pattern lets us alter or extend the component's functionality by replacing the strategy object.

Strategies are a better choice in situations where the Component class is intrinsically heavyweight, thereby making the Decorator pattern too costly to apply. In the Strategy pattern, the component forwards some of its behavior to a separate strategy object. The Strategy pattern lets us alter or extend the component's functionality by replacing the Strategy object.

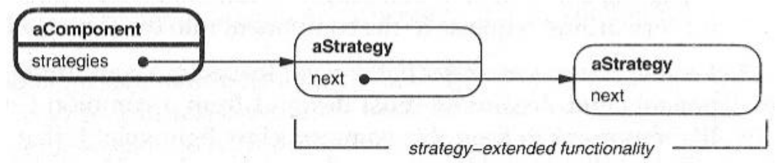
For example, we can support different border styles by having the component defer border-drawing to a separate Border object. The Border object is a Strategy object that encapsulates a border-drawing strategy. By extending the number of strategies from just one to an open-ended list, we achieve the same effect as nesting decorators recursively.

In MacApp 3.0 [App89] and Bedrock [Sym93], for example, graphical components (called "views") maintain a list of "adorner" objects that can attach additional adornments like borders to a view component. If a view has any adorners attached, then it gives them a chance to draw additional embellishments. MacApp and Bedrock must use this approach because the View class is heavyweight. It would be too expensive to use a full-fledged View just to add a border.

Since the Decorator pattern only changes a component from the outside, the component doesn't have to know anything about its decorators; that is, the decorators are transparent to the component:



With strategies, the component itself knows about possible extensions. So it has to reference and maintain the corresponding strategies:



The Strategy-based approach might require modifying the component to accommodate new extensions. On the other hand, a strategy can have its own specialized interface, whereas a decorator's interface must conform to the component's. A strategy for rendering a border, for example, need only define the interface for rendering a border (DrawBorder, GetWidth, etc.), which means that the strategy can be lightweight even if the Component class is heavyweight.

MacApp and Bedrock use this approach for more than just adorning views. They also use it to augment the event-handling behavior of objects. In both systems, a view maintains a list of “behavior” objects that can modify and intercept events. The view gives each of the registered behavior a chance to handle the event before nonregistered behaviors, effectively overriding them. You can decorate a view with special keyboard-handling support, for example, by registering a behavior object that intercepts and handles key events.

Sample Code

The following code shows how to implement user interface decorators in C++. We’ll assume there’s a Component class called `VisualComponent`.

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

  virtual void Draw;
  virtual void Resize();
  //…
};
```

We define a subclass of `VisualComponent` called `Decorator`, which we’ll subclass to obtain different decorations.

```
class Decorator : public VisualComponent {
public:
  Decorator(VisualComponent*);

  virtual void Draw();
  virtual void Resize();
  //…

private:
  VisualComponent* _component;
};
```

`Decorator` decorates the `VisualComponent` referenced by the `_component` instance variable, which is initialized in the constructor. For each operation in `VisualComponent`’s interface, Decorator defines a default implementation that passes the request on to `_component`:

```
Void Decorator::Draw() {
  _component->Draw();
}
void Decorator::Resize() {
  _component->Resize();
}
```

Subclasses of `Decorator` define specific decorations. For example, the class `BorderDecorator` adds a border to its enclosing component. `BorderDecorator` is a subclass of `Decorator` that overrides the `Draw` operation to draw the border.

`BorderDecorator` also defines a private `DrawBorder` helper operation that does the drawing. The subclass inherits all other operation implementations from `Decorator`.

```
class BorderDecorator : public Decorator {
public:
 BorderDecorator(VisualComponent*, int borderWidth);
 virtual void Draw();

private:
  void BrawBorder(int);

private:
  int _width;
};

void BorderDecorator::Draw() {
 Decorator::Draw();
 DrawBorder(_width);
}
```

A similar implementation would follow for `ScrollDecorator` and `DropShadowDecorator`, which would add scrolling and drop shadow capabilities to a visual component.

Now we can compose instances of these classes to provide different decorations. The following code illustrates how we can use decorators to create a bordered scrollable `TextView`.

First, we need a way to put a visual component into a window object. We'll assume our Window class provides a `SetContents` operation for this purpose:

```
Void Window::SetContents (VisualComponent* contents) {
  //…
}
```

Now we can create the text view and a window to put it in:

```
Window* window = new Window;
TextView* textView = new TextView;
```

`TextView` is a `VisualComponent`, which lets us put it into the window:

```
Window->SetContents(textView);
```

But we want a bordered and scrollable `TextView`. So we decorate it accordingly before putting it in the window.

```
Window->SetContents(
  new BorderDecorator(
    new ScrollDecorator(textView), 1
  )
);
```

Because `Window` accesses its contents through the `VisualComponent` interface, it's unaware of the decorator's presence. You, as the client, can still keep track of the text view if you have to interact with it directly, for example, when you need to invoke operations that aren't part of the `VisualComponent` interface. Clients that rely on the component's identity should refer to it directly as well.

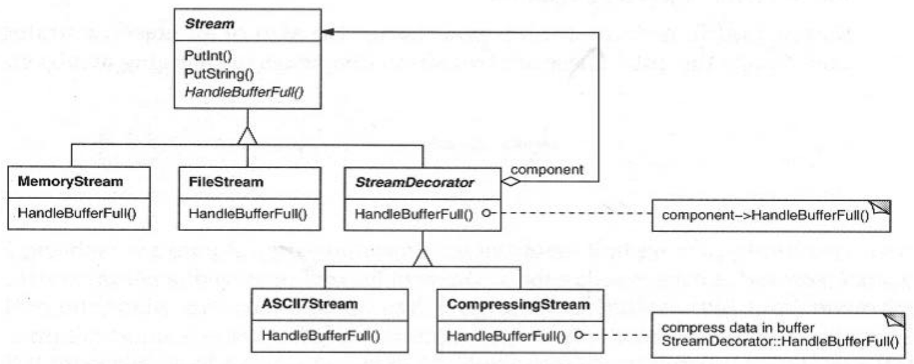
Known Uses

Many object-oriented user interface toolkits use decorators to add graphical embellishments to widgets. Examples include Interviews [LVC89, LC+92], ET++ [WGM88], and the ObjectWorks\Smalltalk class library [Par90]. More exotic applications of Decorator are the DebuggingGlyph from InterViews and the PassivityWrapper from ParcPlace Smalltalk. A DebuggingGlyph prints out debugging information before and after it forwards a layout request to its component. This trace information can be used to analyze and debug the layout behaviour of objects in a complex composition. The PassivityWrapper can enable or disable user interactions with the component.

But the Decorator pattern is by no means limited to graphical user interfaces, as the following example (based on the ET++ streaming classes [WGM88]) illustrates.

Streams are a fundamental abstraction in most I/O facilities. A stream can provide an interface for converting objects into a sequence of bytes or characters. That lets us transcribe an object to a file or to a string in memory for retrieval later. A straightforward way to do this is to define an abstract Stream class with subclasses MemoryStream and FileStream. But suppose we also want to be able to do the following:

- Compress the stream data using different compression algorithms (run-length encoding, Lempel-Ziv, etc.).
- Reduce the stream data to 7-bit ASCII characters so that it can be transmitted over an ASCII communication channel.



The Decorator pattern gives us an elegant way to add these responsibilities to streams. The diagram below shows one solution to the problem:

The Stream abstract class maintains an internal buffer and provides operations for storing data onto the stream (PutInt, PutString). Whenever the buffer is full, Stream calls the abstract operation HandleBufferFull, which does the actual data transfer. The FileStream version of this operation overrides this operation to transfer the buffer to a file.

The key class here is StreamDecorator, which maintains a reference to a component stream and forwards requests to it. StreamDecorator subclasses override HandleBufferFull and perform additional actions before calling StreamDecorator's HandleBufferFull operation.

For example, the CompressingStream subclass compresses the data, and the ASCII7Stream converts the data into 7-bit ASCII. Now, to create a FileStream that compresses its data *and* converts the compressed binary data to 7-bit ASCII, we decorate a FileStream with a CompressingStream and an ASCII7Stream:

```
Stream* aStream = new CompressingStream(
  new ASCII7Stream(
    new FileStream(“aFileName”)
  )
);
aStream->PutInt(12);
aStream->PutString(“aString”);
```

Related Patterns

Adapter: A decorator is different from an adapter in that a decorator only changes an object's responsibilities, not its interface; an adapter will give an object a completely new interface.

Composite: A decorator can be viewed as a degenerate composite with only one component. However, a decorator adds additional responsibilities – it isn't intended for object aggregation.

Strategy: A decorator lets you change the skin of an object; a strategy lets you change the guts. These are two alternative ways of changing an object.

VITA

Name:	Ross Albert McKegney
Place and Year of Birth:	Regina, Saskatchewan, Canada, 1976
Education:	Fredericton High School, 1991-1994
	High School Diploma (with Honours), 1994
	University of New Brunswick, 1994-1999
	Bachelor of Arts with Distinction (Majors in History and Sociology), 1999
	Bachelor of Computer Science 1st Division, 1999
Experience:	Research Assistant, Queen's University, Summer 1999 and Summer 2000
	Teaching Assistant, Queen's University, Fall and Winter 1999/2000
	Computer Programmer, NBTel Inc., 05/1998-04/1999
	Marker, University of New Brunswick, Fall and Winter 1998-1999
	Software Developer, Formal Systems Inc., 09/1996-08/1997
	Respiratory Technician's Assistant, Dr. Everett Chalmers Hospital, 08/1993-04/1997
Awards:	Ontario Graduate Scholarship in Science and Technology, 1999
	UNB Dean's Scholar, 1999
	St. George Prize in History, 1999
	N. Myles Brown Undergraduate Scholarship, 1998
	UNB Awards Office Bursary, 1997
	NBAAC Scholarship, 1997
	UNB Undergraduate scholarship, 1996
	Chester Martin Prize for History, 1996
	Beaverbrook Scholarship, 1995