Software Engineering Institute

Modifiability Tactics

Felix Bachmann
Len Bass
Robert Nord

September 2007

TECHNICAL REPORT
CMU/SEI-2007-TR-002
ESC-TR-2007-002

Software Architecture Technology Initiative

Carnegie Mellon

This report was prepared for the

SEI Administrative Agent
ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2100

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Table of Contents

List of Figures

List of Tables

Abstract

An architectural tactic is a design decision that affects how well a software architecture addresses a particular quality attribute. This report describes how tactics are based on the parameters of quality attribute models. Tactics provide an architectural means of adjusting those parameters, which, in turn, can improve the quality-attribute-specific behavior of the resulting system.

This report justifies the tactics for modifiability, using established concepts of coupling, cohesion, and cost motivations as the means of identifying parameters of interest. Various tactics are then described based on their ability to control these parameters. The report also describes a standard set of architectural patterns and their variants in terms of the use of these tactics.

1 Introduction

Architects use architectural patterns and tactics to aid them in the design process. Both are intended to improve software architectural decisions and simplify the design process. Patterns are known solutions to common problems, whereas tactics focus on specific quality attributes. To more effectively apply both tactics and patterns, architects need to understand how architectural tactics and patterns relate and how to use them effectively. In this report, we explore the relationships of tactics to architectural patterns through the lens of one quality attribute—modifiability.

Modifiability is one of the properties of a software system that has been recognized by software engineers as being important for many years. The basic concepts of coupling and cohesion and information hiding originated in the 1970s [Stevens 1974, Parnas 1972]. A large number of architectural patterns are also intended to support the modifiability of a system [Buschmann 1996]. In 2003, Bass and colleagues introduced the concept of architectural tactics for modifiability to identify architecture transformations that support the achievement of modifiability [Bass 2003]. In this report, we relate coupling and cohesion to tactics, and tactics to patterns. We motivate our collection of modifiability tactics[1] in terms of coupling and cohesion and in terms of cost models. We explain the patterns described by Buschmann and colleagues in terms of these tactics [Buschmann 1996]. We also explain how the variants of these patterns result from the application (or removal) of an architectural tactic to an existing pattern.

Designers using this report will have a choice of techniques to apply to achieve modifiability. They could use tactics while understanding why each one improves the modifiability of a design, they could use patterns while understanding how they combine a number of tactics, or they could use tactics to produce a variant of an existing pattern.

We begin this report with a motivation for the use of tactics as architecture transformations and move into a discussion of prior work on which we build or from which we differ. Then we discuss modifiability, describe several classes of analytic models for modifiability, and describe the architectural tactics for modifiability. Finally in this report, we relate tactics to patterns and the variants of these patterns. In subsequent reports, we will examine achieving the qualities of performance, security, and availability.

In Section 2, we briefly discuss the use of *architectural tactics*, a group of design decisions that influence quality attributes, to transform a software design's behavior [Bass 2003]. Through identifying the parameters of a quality attribute model, we can identify architectural tactics that allow us to control the model's behavior. We make several observations about the relationship between architectural tactics and parameters. In Section 3, we contextualize our examination of modifiabil-

[1] The tactics we define in this report differ slightly from the tactics introduced previously by Bass and colleagues. The differences result from our increased understanding of the sources of the tactics, the omission of tactics from Bass and colleagues that are not architecture transformations, and the revision of the names for clarity [Bass 2003].

ity and architectural tactics within three categories: (1) coupling and cohesion, (2) architectural patterns, (3) techniques for achieving modifiability. Section 4 defines modifiability, examines modification costs, and places modifiability in the context of a design fragment's life cycle. We also discuss modifiability and portability, clarifying how we use portability in the context of this examination. We then, in Section 5, elaborate on the relationships among the responsibilities, or computations of the responsibilities, in a specific module. Responsibilities can be combined, separated, or moved among modules; and the use of modifiability tactics can reduce the strength of coupling of two responsibilities, thus reducing the cost of making a modification.

In Section 6, we enumerate tactics for modifiability. These tactics are defined in terms of the parameters of the coupling and cohesion models—reducing the cost of modifying a single responsibility, increasing cohesion, and reducing coupling. Each of these parameters is developed and illustrated. We identify the deferred bindings enabled by each tactic. In Section 7, we connect the design decisions intended to improve modifiability with the architectural tactics discussed earlier. These design decisions are described in terms of the eight software architectural patterns discussed by Buschmann and colleagues in *Pattern-Oriented Software Architecture: A System of Patterns* [Buschmann 1996]. The section includes an overview of the correspondence between patterns and tactics, as well as a table that details the specific correspondences.

Sections 8 through 15 describe each of the architectural patterns in detail. In these sections, we also discuss the tactics implemented by the particular pattern being examined in terms of responsibilities and dependencies. In each of these sections, we define the modifiability implications of the pattern, followed by the variants of the pattern. Finally, Section 16 summarizes our conclusions.

2 Tactics as Transformations

In this section, we motivate tactics as the transformations that are available to affect the behavior of a design with respect to a particular quality attribute. Figure 1 is a basic queuing-theory model. Events arrive, are queued, are served by the server, are sent to another queue, are served by another server, and then exit the system.

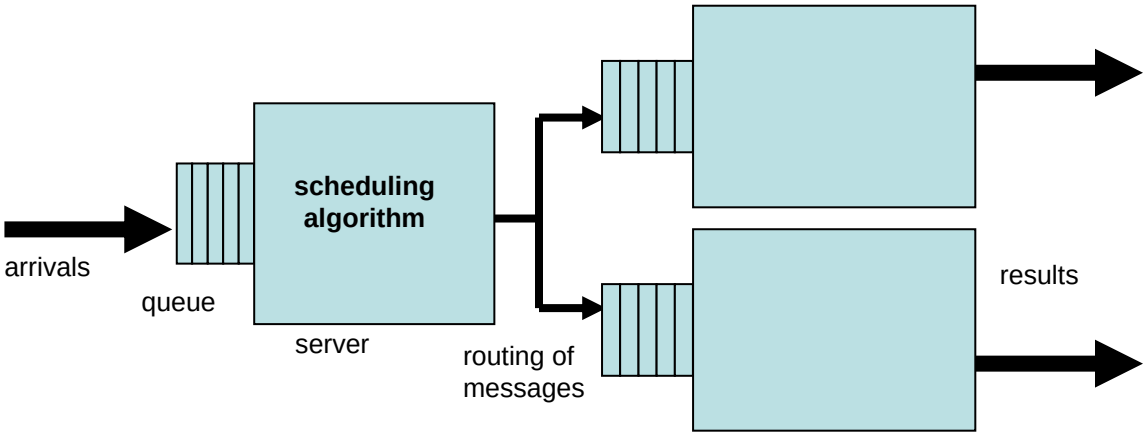

Figure 1: Basic Queuing-Theory Model

This type of model is used to predict the average response time or throughput for jobs arriving at the system.

This model has a number of parameters including the average arrival rate of requests, queue size, queuing discipline, average service time for a request, scheduling algorithm for the server, topology of the network, routing of requests through the network, network bandwidth, and size of the messages being routed through the network. These parameters represent the means to control the behavior of the model. They can be adjusted to control latency or throughput.

An architectural tactic is an architecture transformation that affects the parameters of an underlying quality attribute model, which, in our example, is the basic queuing-theory model. In our example, tactics include transformations to reduce average service time, such as improving an algorithm, pre-allocating dynamic resources such as threads or database connections, or reducing interprocess communication costs by collocating computations within a single process. They also include transformations to affect the routing of messages or the scheduling of the processor.

Three observations that can be made about architectural tactics are

1. The identification of tactics depends on identifying the parameters of the model. We do not need specific values for the parameters or the resulting values for latency and throughput in order to identify tactics. The model must be analyzed sufficiently to determine the direction of the response's change based on the direction of the parameter's change. For example, re-

ducing average service times will result in a reduction of latency and an increase in throughput. To make this assertion about the *direction* of the change, there is no need to predict the *amount* of the change.

2. The model does not need to be a detailed description of a system. Adding more detail to the model adds additional parameters so that our list of parameters above is not complete. Given a model, we can identify all of its parameters. Making the model more detailed adds more parameters. For example, one might criticize Figure 1 on the grounds that it does not reflect classes of jobs or queues for the network. These criticisms are based on the argument that the model does not capture sufficient detail for realistic predictions. However, we are not using the model for predictions but rather only for the identification of parameters. Adding details as to classes of jobs or queues for the network will add parameters and enable us to identify additional tactics, but it does not contradict the identification of the parameters.

3. We do not claim that the tactics in this report are complete, only that they represent transformations for changing parameters. The parameters of the model represent the *opportunities* for changing parameters, but the enumerated tactics are not necessarily all of the transformations for changing them. For example, there are several performance tactics for reducing the average service time, but additional transformations for reducing that time are possible too. Our expectation is that our lists of tactics include the most common techniques.

The argument that we make for modifiability is similar. We define several quality attribute models for modifiability and identify the parameters for them. Once the parameters are identified, we describe some architectural tactics that will affect those parameters and motivate them in terms of the relation between the parameters they affect and the predictions of the model.

3 Context

Three categories of work provide the context for this report: (1) coupling and cohesion, (2) patterns, and (3) techniques for achieving modifiability.

3.1 COUPLING AND COHESION

Stevens, Myers, and Constantine introduced the concepts of coupling and cohesion in 1974 [Stevens 1974]. “Coupling is the measure of the strength of association established by a connection from one module to another,” and “coupling is reduced when the relationships among elements not in the same module are minimized” [Stevens 1974, p. 117 and 121]. Cohesion is a measure of the relationship among the responsibilities of a specific module. We will not attempt to summarize the work based on coupling and cohesion that’s been done since then, except to note that these concepts have been very influential. Our enumeration of tactics is justified based on coupling and cohesion.

3.2 PATTERNS

Architectural patterns have also been very influential since their introduction by Buschmann and colleagues in 1996 [Buschmann 1996]. The collection of architectural patterns given by those authors is the most widely cited. They define an architectural pattern as “expressing a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them” [Buschmann 1996, p. 12]. We will demonstrate how those patterns and their variants can be expressed in terms of architectural tactics.

3.3 TECHNIQUES FOR ACHIEVING MODIFIABILITY

Kazman and Bass introduced several “unit operations” as a first attempt to define the primitives used to achieve various qualities [Kazman 1994]. Separation is a sample unit operation. The set of unit operations was never grounded in theory but rather was derived from expert practice. Subsequently, architectural tactics were introduced for a variety of different quality attributes [Bass 2003, Ch. 5]. The modifiability tactics were also derived from expert practice and never grounded in a theoretical framework for modifiability. In this report, we provide that theoretical grounding for the quality of modifiability.

Other researchers such as Baldwin and Clark [Baldwin 2002] and Chung, Nixon, and Yu [Chung 1999] introduced primitive operations for modifiability, but these operations were never grounded in a theoretical framework for modifiability either.

4 What Do We Mean by Modifiability?

Modifiability is a quality attribute of the software architecture that relates to "the cost of change and refers to the ease with which a software system can accommodate changes" [Northrop 2004, p. 28]. It brings up four concerns: 1) Who makes the change? 2) When is the change made? 3) What can change? and 4) How is the cost of change measured?

In this report, we focus on modifiability as follows:

- Who makes the change? We refer to modifiability as changes made by the architect and developer, though in other situations modifications may be made by others involved in the software system.
- When is the change made? Figure 2 shows several times in a design fragment's life that are important for our discussion. A portion of the system is designed and, at some later time, modified. The modified portion of the system is then deployed and executed. In other situations, modifications may be made after deployment, but that is not represented. Note that this representation is independent of the development process. In a waterfall process, the times refer to the whole system, and in an iterative process, the times refer to the portion of the system being added or modified in the current iteration.

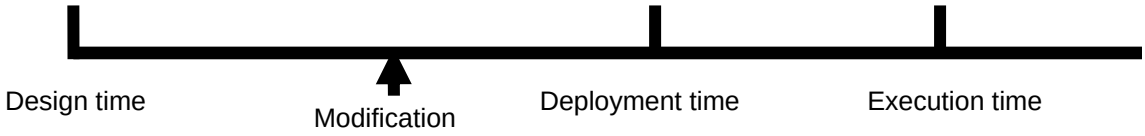

Figure 2: Different Times in the Life of a Design Fragment That Are Important in This Report

- What can change? Determining which elements should be modifiable is an important part of the specification process and is often overlooked. We are examining how the modifiability potential of elements may influence overall costs. Determining specific design elements' needs for modifiability early in the architecture process reduces costs and saves time overall, particularly if the modifiability needs to be flexible. We are concerned in this report about the decisions made about a design fragment at design time. These decisions will affect the difficulty of making a particular modification to that fragment.
- How is the cost of change measured? Once a change has been specified, the new implementation must be designed, implemented, tested, and deployed. All of these actions have an associated cost that can be measured. Our measure of a modification is the cost to make a specific modification given a particular design fragment. Any tactic based on coupling and cohesion will be used during design time to lower the cost of a subsequent change at modification time. Other factors that we might consider include the cost of implementing a tactic, the opportunity cost of not being able to use the money spent on implementing the tactic for other purposes during the period between design time and modification time, the probability

that a particular modification will actually be made, and the number of times that a modification of a particular type is made. The decision to apply a particular tactic is a complicated function of these and other factors. See the work of Ozkaya, Kazman, and Klein for a discussion of one method of treating these factors [Ozkaya 2007]. For this report, we ignore these additional factors and are interested only in the cost of making a particular modification once a tactic has been implemented.

As an example, an architect may choose to apply the encapsulation tactic to a module during design time in anticipation of modifications needed inside the module at a later time. The cost of implementing that tactic for a module includes defining the interface and restricting outside access to the module to use only that interface. There is also the cost of making a modification inside the module once it has been encapsulated. This cost, presumably, will be lower than it would have been had we not created the interface. This second cost—the cost of making the modification given that a tactic has been applied—is the only cost we focus on in this report.

ISO/IEC 9126-1, *Software Engineering – Product Quality – Part I: Quality Model*, has two categories having to do with change: maintainability and portability [ISO 2001]. It defines *maintainability* as the capability of the software product to be modified and *portability* as the capability of the software product to be transferred from one environment to another.

In our usage, portability

- would be a form of modifiability if transferring a product to another environment requires a change to the product
- would not be a form of modifiability if the capability to execute in the different environment already existed within the product

Maintainability is a subset of what we are calling modifiability.

Modifiability, as a concept, is not sufficiently specific to aid designers. They must know what specific modifications are expected in order to apply the appropriate tactics that will allow for those modifications. In our discussion of each tactic, we identify the modification for which the tactic is useful.

5 Parameters to Control Modifiability Based on Coupling and Cohesion

A modifiability tactic is an architecture transformation that improves the modifiability of that architecture. We are applying the approach we sketched for performance to modifiability. That is, we begin with an abstract model of the modifiability problem based on coupling and cohesion and then identify the parameters of this abstract model. The model we begin with requires us to specify somewhat more precisely what is meant by coupling and cohesion and to use this understanding to identify the appropriate parameters.

First, we discuss coupling and cohesion and their implications. Their original definitions are in terms of modules, but our argument involves changing the functions computed within a module and uses the concept of responsibilities to describe the computations of a module. In particular, we are interested in the relations among the responsibilities. We introduce the concept of the "strength" of the coupling of two responsibilities to enable us to argue that modifiability tactics will reduce the strength of a coupling and, hence, reduce the cost of making a modification.

5.1 RESPONSIBILITIES

We define coupling and cohesion in terms of responsibility, which is a concept in a software context that comes from object-oriented design [Wirfs-Brock 2003]. A responsibility is an action, knowledge to be maintained, or a decision to be carried out by a software system or an element of that system. One characterization of a computer system is in terms of the responsibilities of the system, their assignment to modules, and the relationships among the responsibilities [Clements 2003].

We assume that each responsibility has a modification cost associated with it—that of modifying that single responsibility. This is clearly a simplification, since some modifications to a particular responsibility could cost more than others. The variance in costs based on the type of modification could be modeled by introducing probabilistic functions. Our argument does not depend on the particular values assigned but rather on the idea that making certain transformations will not increase the overall cost and will reduce the cost of certain modifications.

The potential to reorganize responsibilities is an important aspect of our discussion. They could be decomposed; for example, the responsibility to "manage bank accounts" might decompose into "authenticate user," "view accounts," and "transfer funds." The responsibilities could be merged; as an example, simply reverse the prior example. They could also be moved from one module to another. Any reorganization could be viewed as a decomposition into more fine-grained responsibilities followed by mergers to yield the new set of responsibilities. Some of our tactics decompose responsibilities, some merge responsibilities, and others move responsibilities among modules.

We use one relation between two responsibilities—the "strength" of the coupling. This relation is not defined formally by Steven, Myers, and Constantine but is implicit in their discussion of coupling [Stevens 1974]. We define the strength of the coupling of two responsibilities as the probability that a modification to one responsibility will propagate to the other. Intuitively, two responsibilities have a high strength of coupling if one of them has an intimate knowledge of the other. This knowledge suggests that changes to one responsibility in one of the modules might have to be reflected in the other. Using our bank example, a modification to "authenticate user" might propagate to "transfer funds." We use strength of coupling to provide a likelihood that this propagation will occur. This is a special case of the UML concept of dependence; although in UML, a dependence either exists or does not exist [OMG 2005]. There is no concept of the strength of a dependence in UML.

The average cost of modifying a responsibility is the average cost of directly modifying that responsibility plus the average cost of modifying all responsibilities that have additional modifications propagated to them. The assumption behind the definition of high coupling is that modifications will propagate to those responsibilities that are strongly coupled to the responsibility being modified.

Coupling is an asymmetric relation: the strength of coupling between responsibility A and responsibility B is not necessarily the same as the strength of the coupling between responsibility B and responsibility A.

We can now identify two of the parameters that we will use to motivate modifiability tactics. Note that these parameters are independent of any particular cost prediction model:

- *average cost of modifying a single responsibility*: Reducing the average cost of modifying a single responsibility A and making no other changes will decrease the average cost of any modification that affects A. Tactics that split responsibilities will reduce the average cost of making a modification to the responsibility that is being split.
- *coupling*: Reducing the strength of the coupling between two responsibilities A and B and making no other changes will decrease the cost of any modification that affects A. Tactics that reduce coupling are those that place barriers of various sorts between responsibilities A and B.

5.2 MODULES

We have defined coupling (although not yet cohesion) in terms of responsibilities. Next, we discuss what happens when modules are introduced. In the Module view type [Clements 2003], responsibilities are assigned to modules. Modules can be decomposed and form a hierarchy. When we refer to responsibilities A and B as being in the same module, we are referring to the module deepest in the hierarchy that contains both responsibilities.

"Coupling is reduced when the relationships among elements *not* in the same module are minimized. There are two ways of achieving this—minimizing the relationships among modules and

maximizing relationships among elements in the same module" [Stevens 1974, p. 121]. Maximizing the relationships among elements in the same module is the definition of cohesion.

We turn cohesion into our third parameter:

- *cohesion*: If responsibility A has a high strength of coupling with responsibility B, the cost of changing responsibility A is less if A and B are collocated in the same module than it would be if they were located in different modules.

In our use of the parameters, we are not explicit about the "high strength of coupling." We assume that the strength of coupling is known to designers through their understanding of the solution's specifics.

5.3 COST-BASED IDENTIFICATION OF PARAMETERS

An additional parameter is based on when in the life of a system a particular modification occurs. Coupling and cohesion are easily translated into architectural terms, since they deal with structural concepts. The tactics that we deal with in this section, called "deferred binding tactics," are motivated by cost considerations. Most cost models are not based on structural concepts but rather on some measurement of the amount of functionality and the skill of the developers. The consequence, for us, is that our tactics are based on one fundamental assumption: as long as there is suitable preparation, the later in the life of a system a modification occurs, the less the modification will cost. The keys are the "suitable preparation" and the fact that we are not considering the cost of preparing for the modification (similar to the coupling and cohesion arguments). The "suitable preparation" is what distinguishes a planned change from an unplanned repair.

Figure 2 shows several different times during the life of a design fragment. The suitable preparation occurs during design time and enables a deferred binding tactic to be applied during modification time, deployment time, start of execution time, or the execution of the system. Furthermore, we shall argue that the suitable preparation is based on the application of a coupling and cohesion tactic. That is, preparation for a deferred decision is based on a tactic derived from coupling and cohesion considerations. Deferring a particular decision is based on cost considerations.

Every activity whose time is indicated in Figure 2 has a cost. If an activity occurs later in Figure 2 than the time of the modification, there is a cost associated with it. If it occurs earlier, the cost is not considered. If a modification is made at execution time—for example through the use of a name server—the modification, *per se*, does not need to be tested or deployed; consequently, the cost of testing and deploying is zero for that modification.

The fourth parameter we identify as a result of cost considerations is:

- *life cycle time of modification*: A suitably prepared modification made late in the life cycle will cost less than the same modification made earlier, because the preparedness of the system means that some of the life cycle costs will be zero for late life cycle modifications.

6 Modifiability Tactics

We are now ready to enumerate tactics for modifiability. The key points for consideration are

- A modifiability tactic is an architectural design decision that affects parameters based on coupling and cohesion or on the ability to omit particular life-cycle steps for a design fragment.
- We wish to control three parameters based on cohesion and coupling through architectural means. We will use those parameters as the means of organizing the tactics.
- One parameter is based on the cost of omitting steps in the life cycle of a design fragment. The tactics that follow from this parameter are made possible through the application of one or more tactics based on coupling and cohesion.
- We must be specific about the expected modifications whose cost will be reduced by the use of tactics.

The tactics we will define are organized based on the parameters of the coupling and cohesion models:

- reducing the cost of modifying a single responsibility
  - Split a Responsibility.
- increasing cohesion
  - Maintain Semantic Coherence.
  - Abstract Common Services.
- reducing coupling
  - Use Encapsulation.
  - Use a Wrapper.
  - Raise the Abstraction Level.
  - Use an Intermediary.
  - Restrict Communication Paths.

Furthermore, we will argue that the coupling- and cohesion-based tactics enable the deferred binding tactics. The deferred bindings that we consider are organized based on the system's life cycle or runtime:

- coding time
  - Use Aspect-Oriented Programming.
  - Use Polymorphism.
  - Parameterize Modules.
- build time
  - Use Component Replacement.

- deployment time
  - Use Configuration-Time Binding.
- initialization time
  - Use Resource Files.
- runtime
  - Use Runtime Registration.
  - Interpret Parameters.
  - Use Start-Up Time Binding.
  - Use Runtime Binding.
  - Use Name Servers.
  - Use Plug-Ins.
  - Use Publisher-Subscriber.

6.1 REDUCING THE COST OF MODIFYING A SINGLE RESPONSIBILITY

The basic approach to reducing the cost of modifying responsibilities is to transform the architecture by splitting the responsibility into two portions based on the specific change to be made. If the split involves more than one responsibility, the tactic is applied as many times as necessary.

6.1.1 Split a Responsibility

We are assuming that a modification has been specified in enough detail for particular responsibilities to be identified as the target of the modification.

If the responsibility being modified includes a great deal of capability, the modification costs might be high. Refining the responsibility into several smaller responsibilities and placing them in distinct modules reduces the cost of particular changes. The aggregate of the responsibilities is the initial responsibility, but for changes that only affect a portion of the initial responsibility, splitting the responsibility reduces the cost.

Splitting a responsibility transforms the architecture in the following fashion:

- The responsibility that was split is replaced by its two children.
- The responsibility being split is deleted from its module. Each of the new children is assigned to a new module. The old module is deleted if it no longer contains any other responsibilities and is retained if responsibilities are still assigned to it.
- Any responsibility that is coupled to the responsibility being split is now coupled to the new responsibilities with a strength no greater than the original strength (i.e., the strength of external coupling is not increased). The coupling of the new responsibilities must be specified. This tactic does not prejudge the strength of coupling the new responsibilities.

Figure 3 shows Responsibility A before and after it is split. For simplicity in the graphics, we show the coupling as symmetric; however, recall that the coupling could have different strengths depending on its direction.

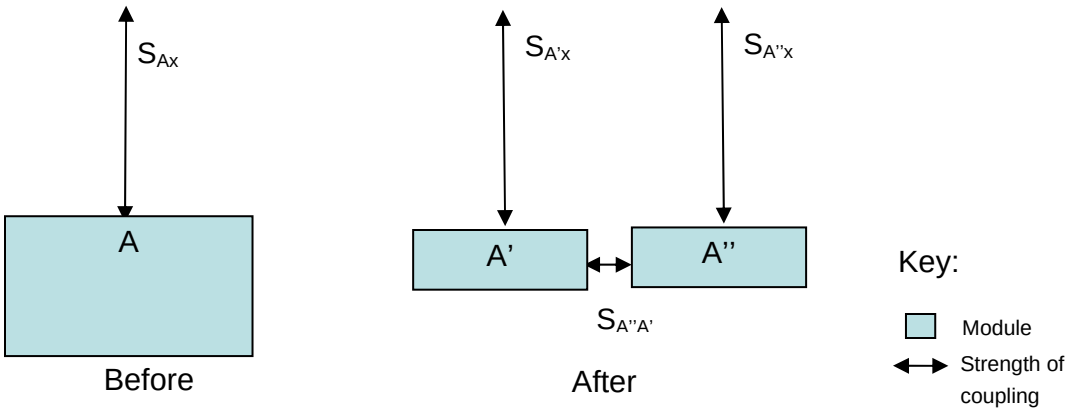

Figure 3: Responsibility A is Divided into Two Portions A' and A'' [2]

One common criterion for splitting a responsibility is that the children of the responsibility can be modified independently. Splitting a responsibility using this criterion reduces the cost of a modification, since only a portion of the initial responsibility has to be modified in response to a modification request.

Deferred bindings enabled by this tactic: Consider the following situation in which a modification specification affects a particular responsibility, and the designer decides to allow the modification at runtime. The responsibility to be modified is then typically split into two responsibilities: one performs the activity being split and the other manages the runtime activities to support a request for the modification. Many deferred binding tactics that are bound prior to deployment time assume that the correct responsibilities have been split from the overall responsibilities. More specifically, the following list identifies the types of responsibilities that have been separated from the overall responsibilities to enable the particular deferred binding. Those tactics that reflect increased generality, such as parameterizing modules or using configuration time bindings, assume that additional responsibilities are added to the initial set of responsibilities and will be considered later.

The bindings enabled by this tactic include

- Use Aspect-Oriented Programming. Aspect-Oriented Programming depends on identifying *aspects* that are woven into the remainder of the code. These aspects must be split from the remaining responsibilities.

[2] Each portion is assigned to a new module and has its own strength of coupling to other responsibilties.

- Use Polymorphism. Polymorphism depends on identifying the responsibilities that are common to a set of services and splitting them from the total set of responsibilities for a system.
- Use Component Replacement. Replacing components at bind time depends on the common set of responsibilities being split from the responsibilities of each component. The component responsibilities are those that are specific for the particular system being built.

6.2 INCREASING COHESION

Several tactics involve moving responsibilities from one module to another. As we discussed when we described cohesion, the purpose of moving a responsibility from one module to another is to reduce the likelihood of side effects to other responsibilities in the original module.

We are assuming that a modification is specified in enough detail to allow the identification of responsibilities A and B as the responsibilities that will be affected by the modification.

In general, the actions of tactics that move responsibilities follow Figure 4**.** The various tactics that we will discuss operate on responsibilities A and B in the figure. A is split into A' and A'', and B is split into B' and B''. A' and B' are then collocated in a single module. Specifically, the tactics make the following transformations to the architecture:

- First, they split responsibilities A and B into the portions to be moved (A' and B') and the portions to remain (A'' and B'').
- Then they create a new module and assign A' and B' to it.

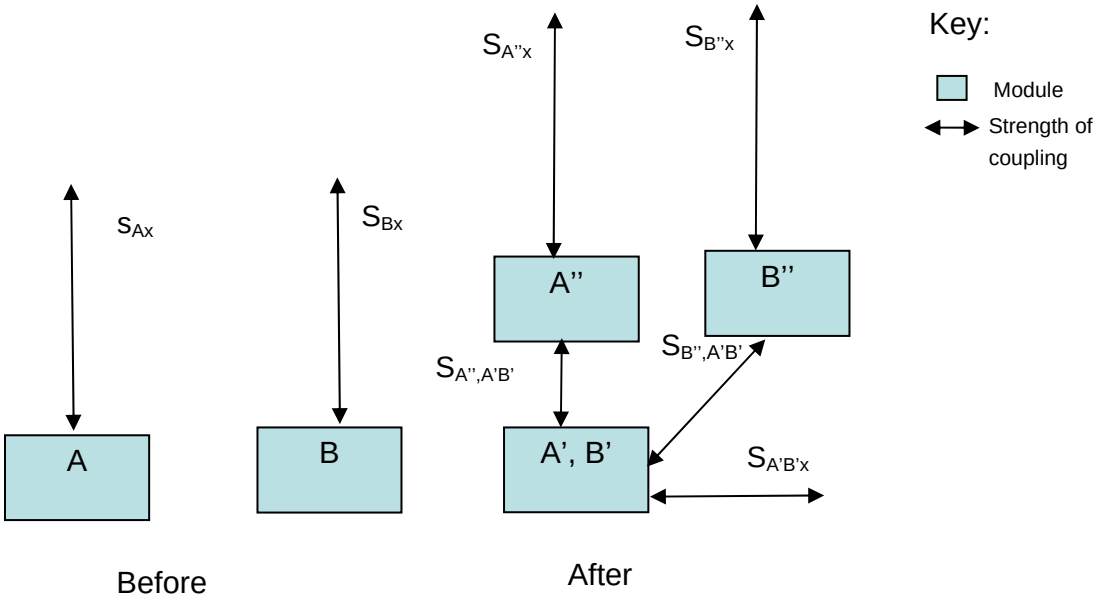

Figure 4: Application of a Tactic to Increase Cohesion[3]

[3] Responsibility A is split into A' and A'', responsibility B is split into B' and B'', and A' and B' are placed into a new module. The interface for the module containing A' and B' is not shown.

The strength of coupling must be assigned to the new responsibilities from other responsibilities. Figure 4 shows responsibilities A and B before and after these tactics are applied.

The particular basis for determining the split depends on the tactic chosen. Next, we discuss the possible tactics.

6.2.1 Maintain Semantic Coherence

A' and B' represent those portions of responsibilities A and B that are simultaneously affected by expected modifications. Those modifications may result from historical knowledge, from specific knowledge of future modifications, or from a logical coherence of the responsibilities. By splitting the responsibilities into two portions—one portion affected by the modification and one not affected by it—the architect enables the collocation of the affected portions, A' and B'. Collocating those portions of various responsibilities that are affected by a single modification will reduce the total cost of a modification if

- The cost of modifying A' (B') is less than the cost of modifying A (B).
- A'' and B'' are unaffected by the modification.

6.2.2 Abstract Common Services

In this case, A' and B' represent similar services. A' provides a variant of this service to A'' to support the achievement of responsibility A. B' provides a variant of the service to B''. Since A' and B' provide essentially the same service, they could be implemented only once in a slightly more general form. Any modification to that service would then only need to occur once, reducing the cost of modifiability if the modification under consideration affects the common service.

Refactoring is an example of this tactic. One refactoring rule involves examining existing responsibilities and factoring out the similar elements. That is exactly what is done by the Abstract Common Services tactic.

The deferred bindings enabled by these tactics are

- Use Aspect-Oriented Programming. A common service is assigned to the module that contains the "advice," and this advice is then woven into the remainder of the system. This weaving occurs at compile or build time, although some research is underway to defer the weaving to runtime [Baker 2002].
- Use Polymorphism. Polymorphism is an instance of refactoring, which, as mentioned, is an application of the Abstract Common Services tactic.
- Use Component Replacement. The replacement components typically use services that are common across all components. Component replacement is an application of the Abstract Common Services tactic.
- Use Configuration-Time Binding. In order to specify bindings at configuration time, the services to be bound must have been abstracted, and mechanisms must exist to bind the configuration time specifications to the actions they control.

- Use Start-Up Time Binding. In order to specify bindings at start-up time, the services to be bound must have been abstracted, and mechanisms must exist to bind the start-up time specifications to the actions they control.
- Use Runtime Binding. In order to specify bindings at runtime, the services to be bound must have been abstracted, and mechanisms must exist to bind the runtime specifications to the actions they control.

6.3 REDUCING COUPLING

We now discuss four tactics intended to reduce coupling. Some of the tactics reduce the coupling from one responsibility to another, and others reduce the coupling from one module to another.

We are assuming that a modification has been specified in enough detail for particular responsibilities to be identified as being affected by the modification.

6.3.1 Use Encapsulation

Encapsulation is used as a portion of virtually all the other tactics, but it also has an independent existence. In this report, we do not explore the concept of the interdependence of various tactics, but others have been investigating these relationships [Kumar 2007].[4] The purpose of encapsulation is to reduce the probability that a change to one module propagates to other modules by introducing an explicit interface. Figure 5 shows module A being accessed by modules B, C, D, and E. Encapsulating module A places an interface in front of A and reduces the strength of coupling between A and B, C, D, and E. Encapsulation acts by enforcing information hiding behind an interface. With such an interface, S_{Ax} is reduced, decreasing the probability that the modification to A will propagate to B, C, D, or E. Encapsulating responsibility A transforms the architecture in the following fashion:

- An explicit interface is added to a module. The interface has its own responsibility. It includes an application programming interface (API) and its associated responsibilities, such as “perform a syntactic transformation on an input parameter to an internal representation.”
- The strengths of coupling that previously went to responsibility A now go to the interface for A. They are reduced from their previous values, because the interface acts to limit the ways in which external responsibilities can interact with A. The external responsibilities can now only directly interact with A through the exposed interface. Indirect interactions, such as dependence on quality of service, will likely remain unchanged; however, limiting direct interactions reduces the coupling between external responsibilities and the internals of A.
- There is strong coupling between A and the interface of A and low coupling between the interface of A and A. This is one case where we make the asymmetry of coupling explicit.

[4] Kumar, K. & Prabhakar, T.V. *Towards Formalizing Architecture Tactic Knowledge*, submitted for publication.

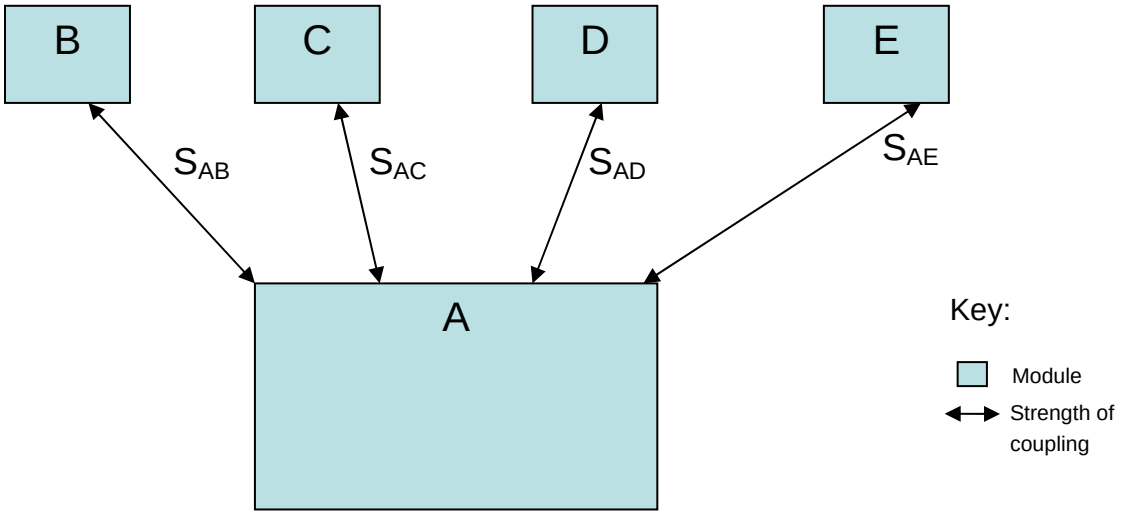

Figure 5: Modules Accessing Module A Prior to Encapsulating A

Figure 6 shows the situation after A has been encapsulated.

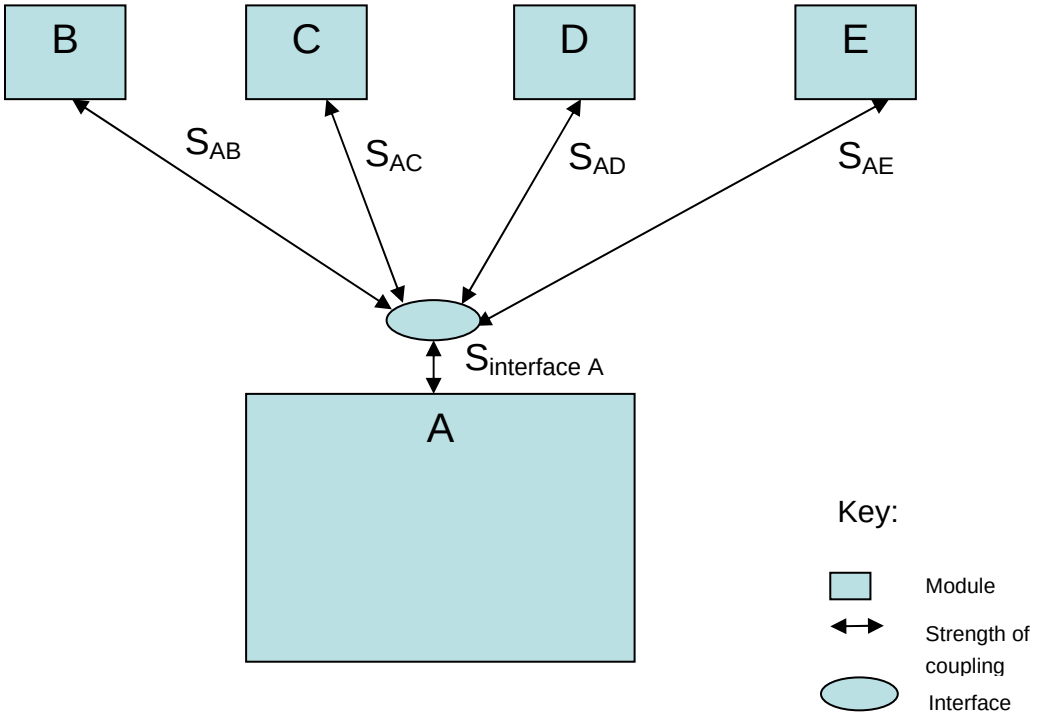

Figure 6: Encapsulating A Introduces an Interface.[5]

[5] The couplings from outside to the interface are unchanged, but the couplings to A are reduced. There is also strong coupling between A and the interface of A and low coupling in the other direction.

Deferred bindings enabled by this tactic: It is common practice for the deferred bindings to have an explicit interface that encapsulates some responsibilities. It is possible to conceive of some runtime bindings being done without an explicit interface, but that is unlikely since encapsulation behind an interface is so pervasive in normal development processes.

6.3.2 Use a Wrapper

A wrapper is a form of encapsulation. It is an interface for a module that transforms the data or control information for the module. The distinction between a wrapper and encapsulation is fairly subtle. Encapsulation is intended to hide information, and transformations may be used as a portion of the hiding strategy. A wrapper is intended to transform invocations, and encapsulation is a portion of the transformation strategy. The point at which an encapsulation becomes a wrapper is vague. Application of the Use a Wrapper tactic is intended to keep outside modifications from propagating into the module. The cost of changing the wrapper is assumed to be less than the cost of changing A. If it's not, the addition of a wrapper makes no sense. Specifically, the Use a Wrapper tactic makes the following transformations:

- An explicit interface is added to the module with its own responsibility named "wrapper for module A."
- Couplings that previously went to responsibility A now go to the wrapper for A.
- The wrapper responsibility has a high strength of coupling in relation to A.
- A has a low strength of coupling in relation to the wrapper responsibility.

These transformations reduce the total cost of a change to a responsibility external to A by reducing the costs associated with propagation. Changes to the external responsibility may propagate to the wrapper responsibility of A, but the wrapper will keep most changes from propagating to A itself. Since the cost of modifying the wrapper responsibility is less than the cost of modifying A, the overall cost of making a modification external to A is reduced.

Figure 5 and Figure 6 (encapsulation) are appropriate for this tactic as well.

6.3.3 Raise the Abstraction Level

Raising the level of abstraction for a responsibility involves parameterizing its activities. The parameters can be as simple as values for variables or as complex as statements in a specialized language that are subsequently interpreted. In the interface, the responsibility typically converts the parameters into an internal form for execution by the original responsibility. Raising the level of abstraction transforms the architecture in the same fashion as the Split a Responsibility tactic. Interpret Figure 3 in the following fashion: A' has the responsibility "interpret parameters," and A" has the responsibility "implement A."

Raising the abstraction level of A will reduce the total cost of changing an external responsibility B by making changes to B less likely to propagate to A. Changes to B can be reflected in different parameters being passed to A that A is already equipped to handle.

Deferred bindings enabled by this tactic: Those deferred bindings that allow a variety of parameters or alternative inputs are enabled by this tactic:

- Parameterize Modules. A module must be general enough to handle a variety of parameter values. The binding of the value to an algorithm is deferred from initial coding time to module use time.
- Use Configuration-Time Binding. The configuration time bindings are typically parameters to a module. Consequently, the module must be general enough to handle a variety of configuration time settings.
- Use Resource Files. The resource file bindings are typically parameters to a module. Consequently, the module must be general enough to handle a variety of resource time settings.
- Use Runtime Registration. The registration service has parameters to control its behavior, allowing it to deal with a variety of different registrants.
- Interpret Parameters. In order to interpret the parameters, the module must be general enough to have parameters.
- Use Start-Up Time Binding. The start-up time bindings are typically parameters to a module. Consequently, the module must be general enough to handle a variety of start-up time settings.
- Use Runtime Binding. The runtime bindings are typically parameters to a module. Consequently, the module must be general enough to handle a variety of runtime settings.
- Use Name Servers. The name server has parameters to control its behavior, allowing it to deal with a variety of different registrants.
- Use Plug-Ins. The service that determines the correct plug-in to invoke is parameterized by the type of the plug-in.
- Use Publisher-Subscriber. The Publisher-Subscriber intermediary differentiates among the various publishers and subscribers by type information. The type information is a parameter that affects the Publisher-Subscriber behavior.

Note that many of these deferred-binding time tactics are enabled by other tactics and thus could be considered to be a pattern, as discussed in the next section.

6.3.4 Use an Intermediary and Restrict Communication Paths

An intermediary breaks a dependency. Given a dependency between responsibility A and responsibility B, it can be broken by the insertion of an intermediary. The type of intermediary depends on the type of dependency. For example, if A is a data producer and B is a data consumer, a Publisher-Subscriber intermediary will remove A's knowledge that B is a consumer. Other data producers and consumers may be able to use the same intermediary to break their dependencies, but that does not affect our discussion. A special case of using an intermediary is to remove a dependency caused by communication needs and then funnel this communication through an intermediary. This is the Restrict Communication Paths tactic. The Use an Intermediary tactic makes the following transformations to modules A and B:

- Create a new responsibility to act as the intermediary, and create a new module for it.
- Delete the strength of coupling between A and B. Replace it with a strength of coupling between A and the intermediate module and a strength of coupling between the intermediate module and B.

Figure 7 shows modules A and B before and after the Use an Intermediary tactic is applied.

Deferred bindings enabled by this tactic: This tactic allows several different forms of deferred binding, which all involve breaking different forms of dependencies:

- Use Name Servers. The location of a service can be determined by using an intermediary for registering the service and allowing this location to be discovered.
- Use Plug-Ins. The interpretation of a bit stream can be bound at runtime by having an intermediary that determines the type of the bit stream and binds it to a component that can interpret the bit stream.
- Use Publisher-Subscriber. Producers can be coupled to consumers at link time or at runtime through different versions of Publisher-Subscriber intermediaries.

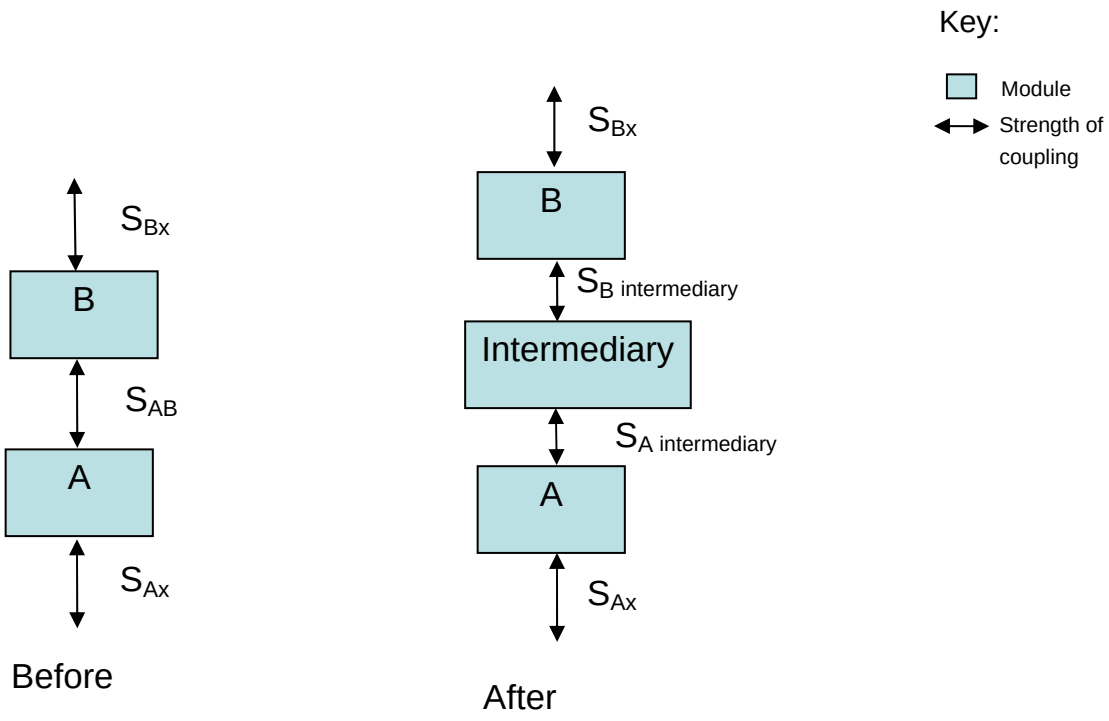

Figure 7: Modules A and B Before and After Applying an Intermediary Tactic

7 Understanding Architectural Patterns in Terms of Tactics and Models

In this section, we draw the connection between potential design decisions intended to improve modifiability (as exemplified by eight software architectural patterns identified in *Pattern-Oriented Software Architecture: A System of Patterns* [Buschmann 1996]) and the architectural tactics we described previously. Table 1 shows the correspondence between these patterns and the tactics they implement.

Table 1: Architectural Patterns and Corresponding Tactics

Pattern	Modifiability									
	Increase Cohesion		Reduce coupling					Defer Binding Time		
	Maintain Semantic Coherence	Abstract Common Services	Use Encapsulation	Use a Wrapper	Restrict Comm. Paths	Use an Intermediary	Raise the Abstraction Level	Use Runtime Registration	Use Start-Up Time Binding	Use Runtime Binding
Layers	X	X	X		X	X	X			
Pipe-and-Filter	X		X		X	X			X	
Blackboard	X	X			X	X	X	X		X
Broker	X	X	X		X	X	X	X		
Model-View-Controller	X		X			X				X
Presentation-Abstraction-Control	X		X			X	X			
Microkernel	X	X	X		X	X				
Reflection	X		X							

Each pattern is described in more detail in the sections that follow. First, the description, problem, and solution of each pattern are described, using text excerpted from the pattern definitions provided by Buschmann and colleagues [Buschmann 1996]. Second, the tactics implemented by the pattern are discussed in terms of responsibilities and dependencies. Third, the modifiability implications of the pattern are discussed in terms of the coupling, cohesion, and deferred binding. Finally, the variants of the pattern are discussed in terms of the same models.

Architects think about their software in terms of its structure both as a set of implementation units and as a set of elements that have runtime behavior and interactions [Clements 2003]. Patterns may be represented in terms of one of these structures or as a hybrid of both of them. The modifiability claims of these architectural patterns rely on a set of assumptions about how the runtime

elements relate to the implementation units (e.g., components are “an encapsulated part of a software system … [and] may be represented as modules, classes, objects or a set of related functions” [Buschmann 1996, p. 434]). In order to reason about the cost of change with respect to the patterns expressed as runtime elements, the corresponding implementation units need to be identified and analyzed.

8 Layers Pattern

The Layers architectural pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.[6]

8.1 PROBLEM

The system specification provided to you describes the high-level tasks to some extent, and specifies the target platform. Portability to other platforms is desired. The mapping of high-level tasks onto the platform is not straightforward, mostly because they are too complex to be implemented directly using services provided by the platform.

The following forces influence the solution:

- *Late source code changes should not ripple through the system…*
- *Interfaces should be stable, and may even be prescribed by a standards body.*
- *Parts of the system should be exchangeable…*
- *It may be necessary to build other systems at a later date with the same low-level issues…*
- *Similar responsibilities should be grouped…*
- *There is no 'standard' component granularity.*
- *Complex components need further decomposition.*
- *Crossing component boundaries may impede performance…*
- *The system will be built by a team of programmers, and work has to be subdivided along clear boundaries…*

8.2 SOLUTION

The pattern consists of a number of layers. Layers are partially ordered with respect to the uses relationship. A layer may only use lower level layers in the partial order.

[6] The italicized portions of Sections 8-15 are taken from the work of Buschmann and colleagues [Buschmann 1996]. These portions are used verbatim from the original publication and have not been edited.

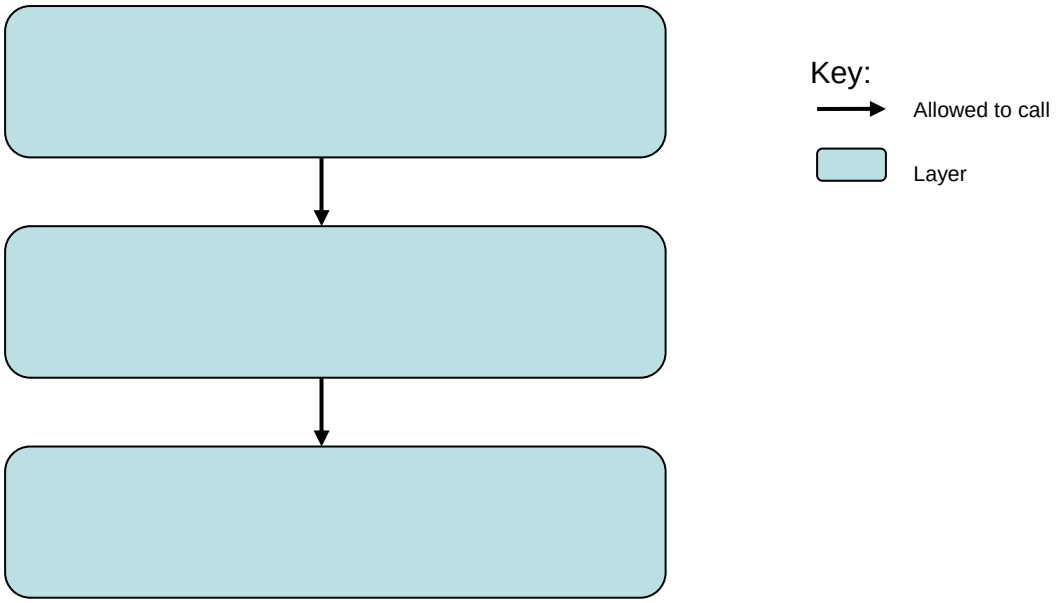

Figure 8: Layers Pattern Structure

The lowest level abstraction is called Layer 1. Services of Layer J are only used by Layer J+1– there are no further direct dependencies between layers.

8.3 THE PATTERN UNDERSTOOD IN TERMS OF TACTICS

We explain the relation of the Layers pattern to modifiability by explaining how the pattern implements various tactics. Each tactic has been explained in terms of its impact on responsibilities, coupling, cohesion, and cost of change. Thus, explaining the Layers pattern in terms of tactics provides a means of relating it to the underlying model.

The Layers pattern can be understood in terms of the following tactics:

- **Maintain Semantic Coherence.** The goal of ensuring that a layer's responsibilities all work together without excessive reliance on other layers is achieved by choosing responsibilities that have some sort of semantic coherence. Doing so binds responsibilities that are likely to be affected by a change. For example, responsibilities that deal with hardware should be allocated to the hardware layer and not to the application layer. A hardware responsibility typically does not have semantic coherence with the application responsibilities.
- **Raise the Abstraction Level.** Layers represent an abstract ladder of services. Modules in lower layers may have an abstract (or more general) representation in the upper layers. For example, there may be concrete device drivers in the lower layer and a more general notion of device driver in an upper layer. Modules in the lower layers are generalized in higher layers to form the abstraction ladder. In our presentation of raising the abstraction level, we identified a responsibility in the interface of a module that translated the parameters into an internal form for subsequent execution. In the Layers pattern, this responsibility exists in the next higher layer.

- **Abstract Common Services.** Typically the responsibilities of a layer are grouped together into services. For example, a layer that has the processor-dependent responsibility may group responsibilities that deal with memory management together to provide a memory management service. Sometimes, a service is replicated in multiple modules. Abstract Common Services would place a single copy of the service in a distinct module and have it accessed by the consumers of the replicated service. In the Layers pattern, the common services are abstracted and located in a layer below the consumers of the services.
- **Use Encapsulation.** There are two design considerations of the Layers pattern with respect to interfaces: (1) each layer may have its own interface and (2) particular layers may act as an interface (e.g., API, façade) for another layer. The first consideration is an instance of the Use Encapsulation tactic; the second is an instance of the Use an Intermediary tactic. The Use Encapsulation tactic divides a layer's responsibilities into two categories: public and private. The public responsibilities are made visible to the outside through an interface. Changing public responsibilities will change the interface through which they are accessed and therefore require adaptations in any layers that use that interface. Changes to the private responsibilities have no effect on other layers.
- **Restrict Communication Paths.** Layers define an ordering and only allow a layer to use the services of its adjacent lower layer. The possible communication paths are reduced to the number of layers minus one. This limitation has a great influence on the dependencies between the layers and makes it much easier to limit the side effects of replacing a layer.
- **Use an Intermediary.** Particular layers may act as an interface (e.g., API, façade) for another layer. The layer above contains the public responsibilities that need to be visible to higher layers and acts as the interface to the layer below in which the private responsibilities are hidden. Changing public responsibilities will change the interface through which they are accessed and therefore require adaptations in any layers that use that interface. Changes to the private responsibilities have no effect on other layers.

8.4 VARIANTS

Relaxed layered system. A relaxed layered system is one in which layer N can invoke any layer below it rather than exclusively layer N-1, which is achieved by removing the Restrict Communication Paths tactic (i.e., removing an intermediary). The intermediary transformation in Figure 7 implements a tactic. The reverse transformation must be used to remove the tactic. Removal of this tactic increases the coupling between layer N and layers N-2 and below.

Layering through inheritance. This variant refers to how the layers are packaged and, consequently, the binding time between them. If lower layers are implemented as base classes, higher layers would inherit the lower layers and could subsequently modify them as desired, resulting in the code-based binding time of the layers and increased coupling between them. The encapsulation of the lower layers will be broken, because those layers become extensions of the code being inherited. The Use Encapsulation tactic of the lower layers is removed. Again, the reversal of the transformation we presented in Figure 5 and Figure 6 can be used to create this variant.

9 Pipes and Filters Pattern

The Pipes and Filters architectural pattern provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters allows you to build families of related systems.

9.1 PROBLEM

Imagine you are building a system that must process or transform a stream of input data. Implementing such a system as a single component may not be feasible for several reasons: the system has to be built by several developers, the global system task decomposes naturally into several processing stages, and the requirements are likely to change.

The following forces influence the solution:

- *Future system enhancements should be possible by exchanging processing steps or by recombining steps, even by users.*
- *Small processing steps are easier to reuse in different context than large components.*
- *Non-adjacent processing steps do not share information.*
- *Different sources of input data exist, such as a network connection or a hardware sensor providing temperature readings.*
- *It should be possible to present or store final results in various ways.*
- *Explicit storage of intermediate results for further processing in files clutters directories and is error-prone, if done by users.*
- *You may not want to rule out multi-processing the steps, for example running them in parallel or quasi-parallel.*

9.2 SOLUTION

The Pipes and Filters architectural pattern divides the task of a system into several sequential processing steps. Each processing step is implemented by a filter. The input to the system is provided by a data source. The output flows into a data sink. The data sources, filters, and the data sink are connected sequentially by pipes. Each pipe implements the data flow between adjacent processing steps.

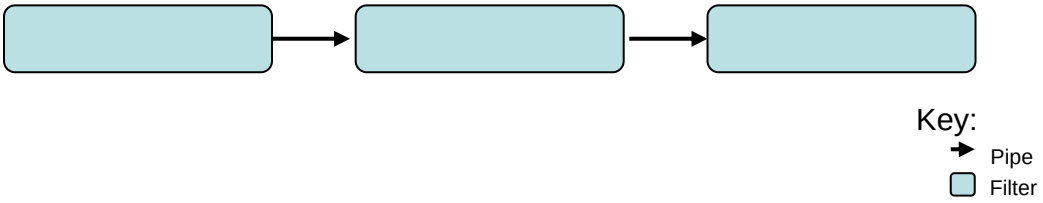

Figure 9: Pipes-and-Filters Pattern Structure

9.3 THE PATTERN UNDERSTOOD IN TERMS OF TACTICS

The Pipes-and-Filters pattern implements the following tactics:

- **Maintain Semantic Coherence.** Filters are defined as the processing steps. A filter's responsibilities work together to enrich, refine, or transform its input data. Semantic coherence groups the responsibilities that are allocated to a filter by how they manipulate data or information. The responsibilities work together to enrich data by computing and adding information, to refine data by concentrating or extracting information, and to transform data by delivering data in some other representation. These responsibilities operate independently and do not depend on other processing steps.
- **Use Encapsulation.** Filters hide information about the processing and conform to an interface for processing data. Some systems use the same data interface for all filters; others define specialized interfaces. The filters that public responsibilities make visible to the outside include getting input data and supplying output data. The private responsibilities that are hidden are those involved with performing a function on the input data.
- **Restrict Communication Paths.** The sequence of filters combined with pipes is called a processing pipeline. The pattern restricts communication paths with constraints that limit filters to be single input and single output. Variants relax this tactic by allowing filters to have more than one input and/or more than one output with constraints on the topology (e.g., directed acyclic graphs, tee-and-join pipeline systems). This limitation influences the number of dependencies between filters and, hence, the number of responsibilities to which a modification could propagate.
- **Use an Intermediary.** Pipes break some, but not all, dependencies between adjacent filters by providing buffering and synchronization. However, they do not break dependencies on the syntax of data. An intermediary could protect filters from changes in data syntax.
- **Use Start-Up Time Binding.** The binding between pipes and filters may be created when the filters are invoked. UNIX uses start-up time binding in its use of pipes and filters.

9.4 VARIANTS

Tee and Join Pipeline Systems. Tee is the copying of an input stream to create two pipes with the same content. This process results in the removal the Restrict Communication Paths tactic,

because it creates a new communication path. The reverse transformation from Figure 8 can be used to remove the tactic. Join is the merger of two input streams into a single filter. Join is also a removal of the Restrict Communication Paths tactic, because it adds a new communication path to the filter managing the join.

10 Blackboard Pattern

The Blackboard architectural pattern is useful for problems for which no deterministic solution strategies are known. In Blackboard several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution.

10.1 PROBLEM

The Blackboard pattern tackles problems that do not have a feasible deterministic solution for the transformation of raw data into high-level data structures.

The following forces influence the solution:

- *A complete search of the solution space is not feasible in a reasonable time.*
- *Since the domain is immature, you may need to experiment with different algorithms for the same subtasks. For this reason, individual models should be easily exchangeable.*
- *There are different algorithms that solve partial problems.*
- *Input, as well as intermediate and final result, have different representations, and the algorithms are implemented according to different paradigms.*
- *An algorithm usually works on the results of other algorithms.*
- *Uncertain data and approximate solutions are involved.*
- *Employing disjoint algorithms induces potential parallelism. If possible you should avoid a strictly sequential solution.*

10.2 SOLUTION

The idea behind the Blackboard architecture is a collection of independent programs (knowledge sources) that work cooperatively on a common data structure. A central control component evaluates the current state of processing and coordinates the specialized programs.

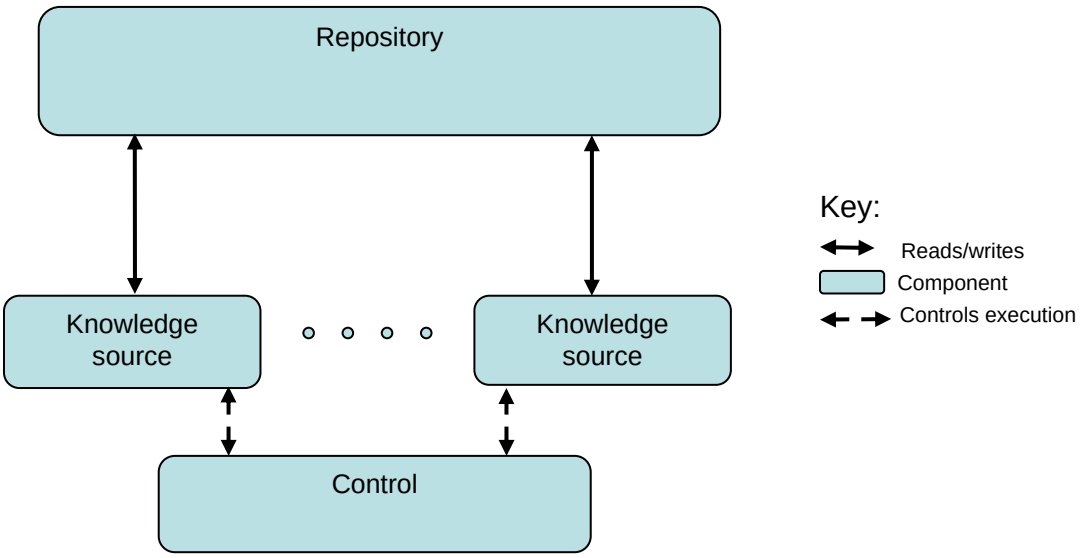

Figure 10: Blackboard Pattern Structure

10.3 THE PATTERN UNDERSTOOD IN TERMS OF TACTICS

The Blackboard pattern implements the following tactics:

- **Maintain Semantic Coherence.** Knowledge sources are organized by semantic coherence. They are separate, independent subsystems that solve aspects of the overall problem. Semantic coherence groups responsibilities within a knowledge source according to the specialized part of the problem they have knowledge to solve.
- **Use an Intermediary and Restrict Communication Paths.** Knowledge sources do not communicate with each other directly. Rather, they use an intermediary in the form of the blackboard. The pattern doesn't say anything about the binding time, though the structure of the patterns lends itself to use with other patterns to defer the binding time of the knowledge sources.
- **Use Runtime Registration.** Typically, knowledge sources and sinks register for particular data items at runtime. The control structure is such that knowledge sources are triggered by changes in data items.

10.4 VARIANTS

Production system. In this variant, the knowledge sources are represented as a set of rules in a higher order language. This representation is the result of applying the Raise the Abstraction Level, Use Runtime Binding, and Abstract Common Services tactics. Each knowledge source is expressed as statements in the higher level language. Raising the level of abstraction of each knowledge source so that the source contains an interpreter of the higher level language and then abstracting this interpreter as a common service yields the infrastructure for the production system language. Binding the language statements to the interpreter at runtime yields a production system.

Repository. A repository is a blackboard that acts only on data and does not manage the control aspects of data, thereby removing one aspect of the intermediary.

11 Broker Pattern

The Broker architectural pattern can be used to structure distributed software systems with decoupled components that interact by remote service invocations. A broker component is responsible for coordinatng communication, such as forwarding requests, as well as for transmitting results and exceptions.

11.1 PROBLEM

Building a complex software system as a set of decoupled and interoperating components, rather than as a monolithic application, results in greater flexibility, maintainability, and changeability. By partitioning functionality into independent components the system becomes potentially distributable and scalable.

The following forces influence the solution:

- *Components should be able to access services provided by others through remote, location-transparent service invocations.*
- *You need to exchange, add, or remove components at run-time.*
- *The architecture should hide system- and implementation-specific details from the users of components and services.*

11.2 SOLUTION

Introduce a broker component to achieve better decoupling of clients and servers. Servers register themselves with the broker, and make their services available to clients through method interfaces. Clients access the functionality of servers by sending requests via the broker. A broker's tasks include locating the appropriate server, forwarding the request to the server, and transmitting results and exceptions back to the client.

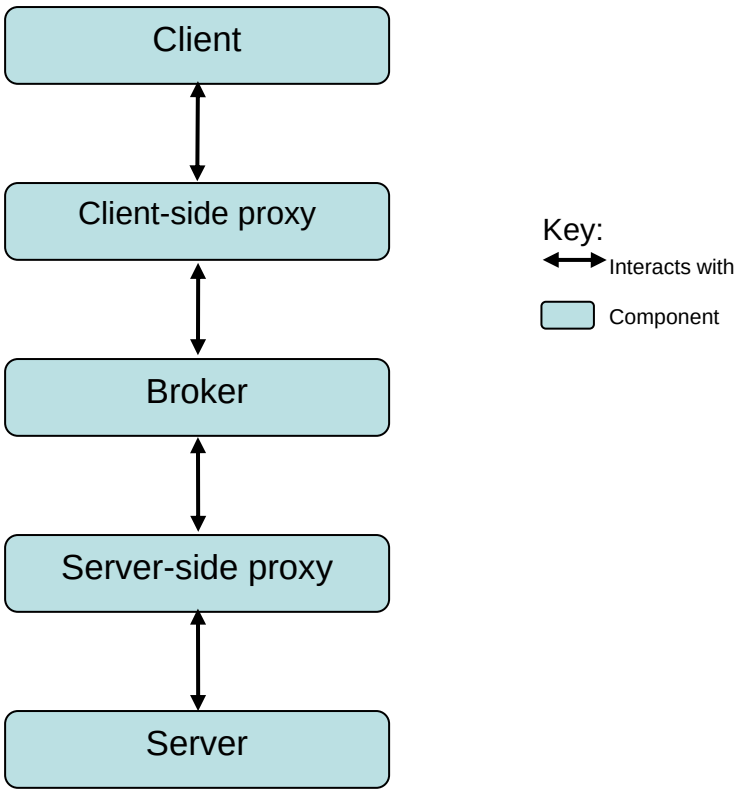

Figure 11: Broker Pattern Structure

The Broker architectural pattern comprises six types of participating components: clients, servers, brokers, bridges, client-side proxies, and server-side proxies.

11.3 THE PATTERN UNDERSTOOD IN TERMS OF TACTICS

The Broker pattern implements the following tactics:

- **Maintain Semantic Coherence.** The description of the Broker pattern states that servers group "semantically-related functionality."
- **Use Encapsulation.** Servers expose their public responsibilities to provide services to clients through interfaces and hide their private responsibilities that represent implementation details. Brokers are responsible for transmitting requests from clients to servers and for transmitting responses back to the client. The brokers expose their public responsibilities through APIs to clients and servers that include operations for registering servers and invoking server methods. These interfaces hide system- and implementation-specific details and allow clients to access services provided by others through remote, location-transparent service invocations.
- **Use an Intermediary.** The Broker pattern acts as an intermediary between clients and servers to provide location-transparent service invocations. Client-side proxies represent an intermediary between clients and the broker, so remote objects appear to the client as local. The proxy hides implementation details from the client, such as the interprocess communication mechanism used for message transfers between clients and brokers; the creation and de-

letion of memory blocks; and the marshaling of parameters and results. Server-side proxies represent an intermediary between servers and the broker. The proxy is responsible for receiving requests, unpacking incoming messages, unmarshalling the parameters, and calling the appropriate service.

- **Restrict Communication Paths.** The clients communicate with the proxy rather than directly with the servers. This restriction allows for changes in the servers for the purposes of reliability or load balancing.
- **Use Runtime Registration.** Servers register themselves with the broker at runtime.

11.4 VARIANTS

Direct communication. In this variant, clients communicate with servers directly and bypass the broker. Brokers serve two functions: (1) making the connection between the client and the server and (2) managing the communication between them. In this variant, the communication management function is removed. This variant is a removal of one application of the Use an Intermediary tactic.

Message-passing broker system. In this variant, servers depend on the type of message to determine their action rather than offering specific services. This variant is an application of the Raise the Abstraction Level tactic, since the servers interpret the incoming message to determine their action.

Trader system. In this variant, requests are made for *services* rather than *servers*. The broker is responsible for locating a server to respond to the service request. This variant is an application of the Raise the Abstraction Level tactic to the broker. Rather than having the server's identity hardwired, the broker must interpret the service request and determine the correct server.

Adapter broker system. This variant adds a layer to the broker on the server side that registers servers and interacts with them. By having multiple adapters, different strategies for assigning servers can be used. This application of the Use an Intermediary tactic interposes the adapter between the server and the broker, which also restricts communication paths. Also an application of the Abstract Common Services tactic, this variant puts all server communication and registration services in one location.

Callback broker system. In this variant, there is no distinction between the client and server. Each server registers to receive events of a particular type, and the broker calls the appropriate server when events arrive. In this sense, it is similar to the message-passing variant. This variant is an application of the Raise the Abstraction Level tactic to the broker, since it now must distinguish between different types of events rather than reacting to the identity of the server in a request.

12 Model-View-Controller Pattern

The Model-View-Controller architectural pattern (MVC) divides an interactive application into three components. The model contains the core functionality and data. Views display information to the user. Controllers handle user input. Views and controllers together comprise the user interface. A change-propagation mechanism ensures consistency between the user interface and the model.

12.1 PROBLEM

User interfaces are especially prone to change requests. Building a system with the required flexibility is expensive and error prone if the user interface is tightly interwoven with the functional core.

The following forces influence the solution:

- *The same information is presented differently in different windows.*
- *The display and behavior of the application must reflect data manipulations immediately.*
- *Changes to the user interface should be easy, and even possible at runtime.*
- *Supporting different 'look and feel' standards or porting the user interface should not affect code in the core of the application.*

12.2 SOLUTION

MVC divides an interactive application into the three areas: processing, output, and input. The model component encapsulates core data and functionality. View components display information to the user. Each view has an associated controller component. Controllers receive input.

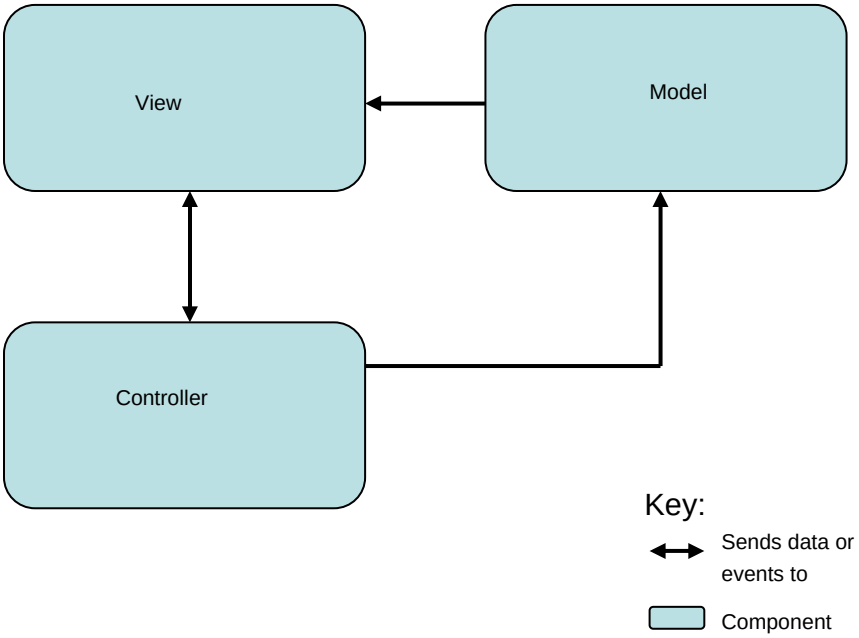

Figure 12: Model-View-Controller Pattern Structure

12.3 THE PATTERN UNDERSTOOD IN TERMS OF TACTICS

The Model-View-Controller pattern implements the following tactics:

- **Maintain Semantic Coherence.** According to the definition, the model component contains the functional core of the application, requiring all the necessary responsibilities for those concepts to be located within the model.
- **Use Encapsulation.** The model component encapsulates the functional core data and functionality.
- **Use an Intermediary.** The controller acts as an intermediary between the input device and the model. The view acts as an intermediary between the model and the output device.
- **Use Runtime Binding.** Views can be opened and closed dynamically, and different views can be bound to the data at different times during execution.

12.4 VARIANTS

Document view. In this variant, the view and the controller are merged. This merger is the application of the Maintain Semantic Coherence tactic, because it allows all window events to be managed in a consistent fashion. This merger is also the removal of the Use an Intermediary tactic, because the controller no longer acts as an intermediary between the view and the model.

13 Presentation-Abstraction-Control Pattern

The Presentation-Abstraction-Control architectural pattern (PAC) defines a structure for interactive software systems in the form of a hierarchy of cooperating agents. Every agent is responsible for a specific aspect of the application's functionality and consists of three components: presentation, abstraction, and control. This subdivision separates the human-computer interaction aspects of the agent from its functional core and its communication with other agents.

13.1 PROBLEM

The following forces influence the solution:

- *Agents often maintain their own state and data.*
- *Interactive agents provide their own user interface.*
- *Systems evolve over time.*

13.2 SOLUTION

Structure the interactive application as a tree-like hierarchy of PAC agents. Every agent is responsible for a specific aspect of the application's functionality, and consists of three components: presentation, abstraction, and control. The presentation provides the visible behavior. The abstraction maintains the data model that underlies the agent, and provides functions that operate on this data. Its control connects the present and abstraction components and provides functionality that allows the agent to communicate with other PAC agents.

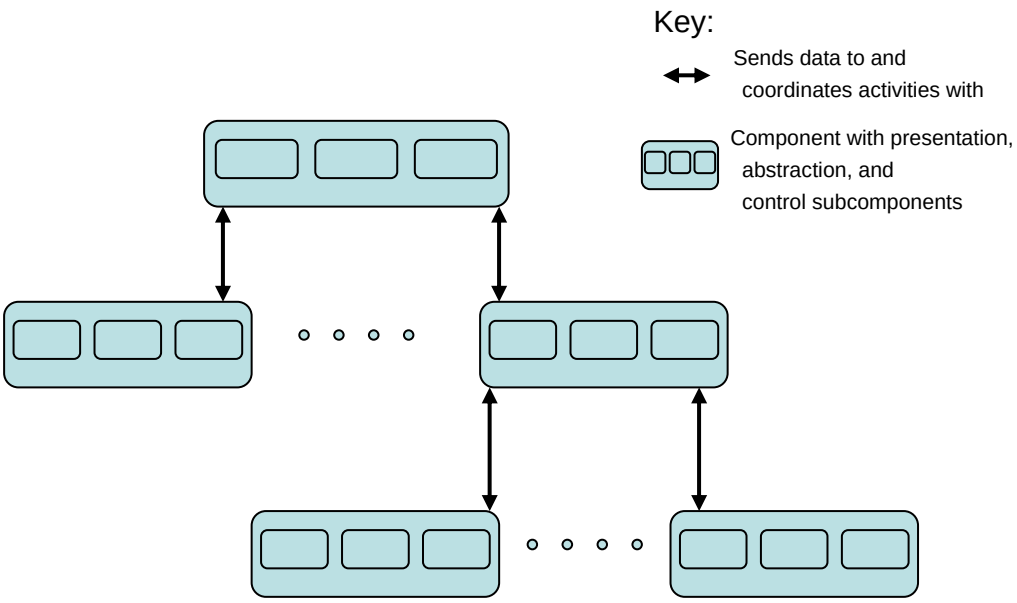

Figure 13: Presentation-Abstraction-Control Pattern Structure

13.3 THE PATTERN UNDERSTOOD IN TERMS OF TACTICS

The Presentation-Abstraction-Control pattern implements the following tactics:

- **Maintain Semantic Coherence.** According to the definition, a PAC agent implements an important concept in the system. Implementing a concept within an agent requires locating responsibilities for those concepts within the components of an agent.
- **Raise the Abstraction Level.** Data that flows down the hierarchy is specialized at each agent. Thus, much like layers, each agent acts as an interface to specialize information before passing it on to the agents below it.
- **Use Encapsulation.** Information and behavior that define the concept are made visible. How an agent computes that concept is kept hidden.
- **Use an Intermediary.** PAC is a hierarchy of agents. Each parent acts as an intermediary between its parent and its children. Using this hierarchy allows the modification of the children without affecting the upper levels of the hierarchy.

13.4 VARIANTS

PAC agents as active objects. In this variant, each PAC agent is active and lives in its own thread of control. An agent's thread model is not the result of the application (or removal) of a modifiability tactic. Threads of control are not a concept in the modifiability model.

PAC agents as processes. In this variant, each PAC agent may be located in a different process or on remote machines. Proxies are used to manage the communication. The use of a proxy is an application of the Use an Intermediary tactic.

14 Microkernel Pattern

The Microkernel architectural pattern applies to a software system that must be able to adapt to changing system requirements. It separates a minimal functional core from extended functionality and customer-specific parts. The microkernel also serves as a socket for plugging in these extensions and coordinating their collaboration.

14.1 PROBLEM

The following forces influence the solution:

- *The application platform must cope with continuous hardware and software evolution.*
- *The application platform should be portable, extensible, and adaptable to allow easy integration or emerging technologies.*

14.2 SOLUTION

The microkernel patterns defined five kinds of participating components: internal servers, external servers, adaptors, clients, and microkernel.

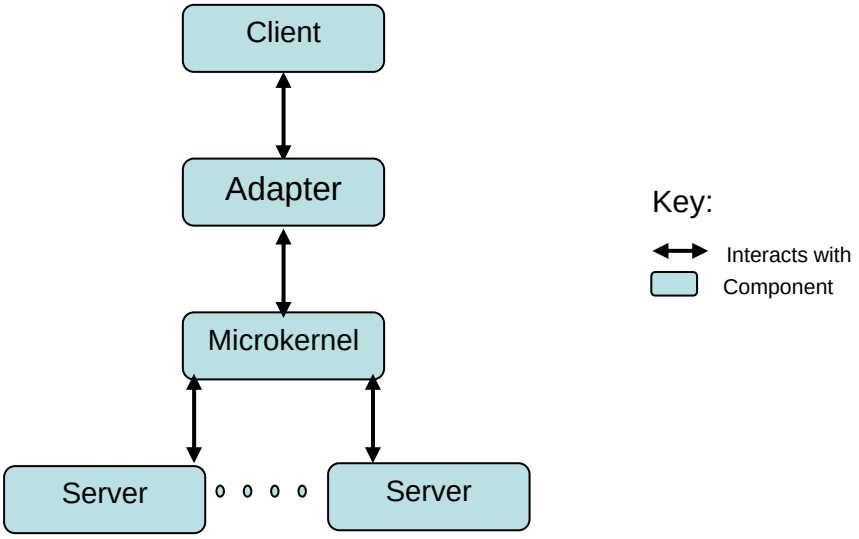

Figure 14: Microkernel Pattern Structure

14.3 THE PATTERN UNDERSTOOD IN TERMS OF TACTICS

The Microkernel pattern implements the following tactics:

- **Maintain Semantic Coherence.** The microkernel implements central services for resource handling. Internal and external servers extend the functionality provided by the microkernel.

- **Abstract Common Services.** The microkernel implements atomic services or mechanisms that serve as a fundamental base on which more complex functionality is constructed. These services are abstracted from any consumer of the services.
- **Use Encapsulation.** System-specific dependencies are encapsulated within the microkernel.
- **Restrict Communication Paths.** Internal servers are only accessible by the microkernel component. The microkernel can only be accessed by external servers and adaptors. Clients can only access the system by way of adaptors.
- **Use an Intermediary.** Adaptors (also known as emulators) serve as interfaces between clients and their external servers to protect clients from direct dependencies.

14.4 VARIANTS

Microkernel system with indirect client-server connections. In this variant, all communication paths are established through the microkernel. The communication itself is done directly. This direct communication is an application of the Abstract Common Services tactic that establishes communication in one location and of the Restrict Communication Paths tactic that requires certain communications to go through the microkernel.

Distributed microkernel system. In this variant, the microkernel is distributed but appears uniform to the client. The client does not need to know on which machine a particular service is located. This distribution is an application of the Use an Intermediary tactic that disguises the location of the actual service.

15 Reflection Pattern

The Reflection architectural pattern provides a mechanism for changing the structure and behavior of a software system dynamically. It supports the modification of fundamental aspects, such as type structures and function call mechanisms. In this pattern, an application is split into two parts. A meta level provides information about selected system properties and makes the software self-aware. A base level includes the application logic. Its implementation builds on the meta level. Changes to information kept in the meta level affect subsequent base-level behavior.

15.1 PROBLEM

The following forces influence the solution:

- *Changing software is tedious, error prone, and often expensive.*
- *Adaptable software systems usually have a complex inner structure.*
- *The more techniques that are necessary for keeping a system changeable, the more awkward and complex its modification becomes.*
- *Changes can be of any scale.*
- *Even fundamental aspects of the software system can change.*

15.2 SOLUTION

Make the software self-aware and make selected aspects of its structure and behavior accessible for adaptation and change. This leads to a structure of two parts, a meta level and a base level. The meta level consists of meta-objects that represent information about the software. The base level defines the application logic. Its implementation uses the meta-objects to remain independent of those aspects that are likely to change.

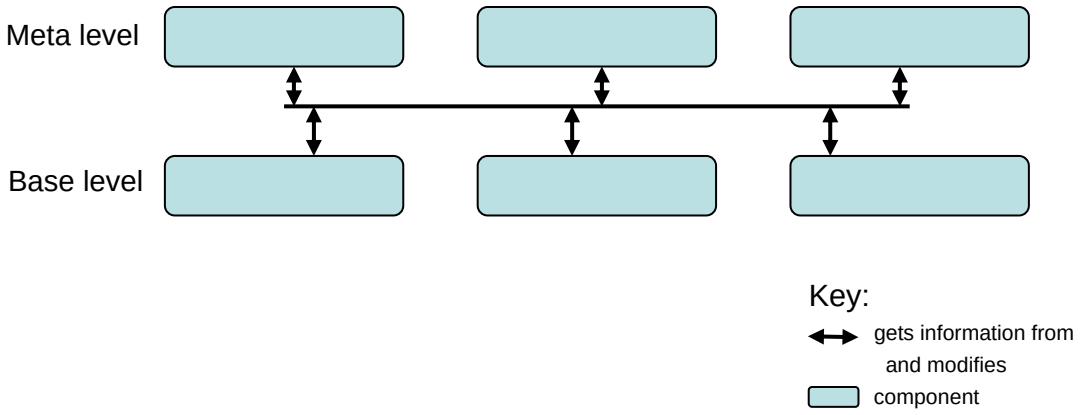

Figure 15: Reflection Pattern Structure

15.3 THE PATTERN UNDERSTOOD IN TERMS OF TACTICS

The Reflection pattern implements the following tactics:

- **Maintain Semantic Coherence.** Meta-objects represent aspects of the software that are likely to change, and they control that change. The base level defines the application logic and uses the meta-objects to remain independent of those aspects that are likely to change.
- **Use Encapsulation.** An interface is specified for manipulating the meta-objects.

15.4 VARIANTS

Reflection with several meta levels. In this variant, several meta levels may depend on each other. This variant is another application of the Maintain Semantic Coherence tactic, in which the tactic is applied to the meta-objects themselves.

16 Conclusions

The general problem of improving an architecture's capabilities and properties is one that practitioners face daily. What we have presented here is a first approach to a principled basis for performing transformations to improve the modifiability characteristics of an architecture.

Beginning with two classes of models of modifiability—one based on coupling and cohesion and one based on cost models—we derived a collection of architectural tactics to improve modifiability. Each tactic represents a transformation of the architecture.

Because of the importance of architectural patterns in the design process, we also presented an explanation of architectural patterns that relate to modifiability and the patterns' variants in terms of the architectural tactics. These tactics allow the transformation of architectures based on patterns or the transformation of existing architectures into ones that embody the use of particular patterns. The designer has two options for using modifiability tactics:

1. to understand patterns and how they support modifiability
2. to adjust patterns in order to improve their properties with respect to modifiability

In either case, the designer can use architectural tactics as a basis for making principled design decisions.

Modifiability is one quality that is important for architectures, but it is not the only one. We anticipate providing similar analyses for other important quality attributes.

17 Bibliography

[Baldwin 2000]
Baldwin, C. & Clark, K. *Design Rules: The Power of Modularity*. Cambridge, MA: MIT Press, 2000 (ISBN: 978-0-262-02466-2).

[Baker 2002]
Baker, J. & Hsieh, W. "Runtime Aspect Weaving Through Metaprogramming," 86-95. *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*. Enschede, The Netherlands, April 2002. New York, NY: ACM Press, 2002 (ISBN:1-58113-469-X).

[Bass 2003]
Bass, L.; Clements, P.; & Kazman, R. *Software Architecture in Practice*, 2nd edition. Boston, MA: Addison-Wesley, 2003 (ISBN: 978-0-321-15495-8).

[Bosch 2000]
Bosch, J. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach.* Boston, MA: Addison-Wesley, 2000 (ISBN: 0-201-67494-7).

[Buschmann 1996]
Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; & Stal, M. *Pattern-Oriented Software Architecture: A System of Patterns*. Chichester, NY: Wiley, 1996 (ISBN: 978-0-471-95869-7).

[Chung 1999]
Chung, L.; Nixon, B.; & Yu, E. *Non-Functional Requirements in Software Engineering*. Boston, MA: Kluwer Academic, 1999 (ISBN: 978-0-792-38666-7).

[Clements 2003]
Clements, P.; Bachmann, F.; Bass, L.; Garlan, D.; Ivers, J.; Little, R.; Nord, R.; & Stafford, J. *Documenting Software Architectures: Views and Beyond.* Boston, MA: Addison-Wesley, 2003 (ISBN: 978-0-201-70372-6).

[ISO 2001]
International Organization for Standardization/International Electrotechnical Commission. ISO/IEC 9126-1: 2001: *Software Engineering - Product Quality - Part 1: Quality Model*. Geneva, Switzerland: International Organization for Standardization, 2001.

[Kazman 1994]
Kazman, R. & Bass, L. *Toward Deriving Software Architectures from Quality Attributes* (CMU/SEI-94-TR-010, ADA283827). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1994.
http://www.sei.cmu.edu/publications/documents/94.reports/94.tr.010.html.

[Kiczales 1996]
Kiczales, G. "Aspect-Oriented Programming." *ACM Computing Surveys (CSUR) 28*, 4es (December 1996): 154.

[Northrop 2004]
Northrop, L. *Achieving Product Qualities Through Software Architecture Practices*. http://www.sei.cmu.edu/architecture/cseet04.pdf (2004).

[OMG 2005]
Object Management Group. *Unified Modeling Language: Superstructure Version 2.0, formal/05-07-04*. http://www.omg.org/cgi-bin/doc?formal/05-07-04 (August 2005).

[Ozkaya 2007]
Ozkaya, I.; Kazman, R.; & Klein, M. *Quality-Attribute-Based Economic Valuation of Architectural Patterns* (CMU/SEI-2007-TR-003, ADA468620). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2007. http://www.sei.cmu.edu/publications/documents/07.reports/07tr003.html.

[Parnas 1972]
Parnas, D. "On the Criteria to Be Used in Decomposing Systems into Modules." *Communications of the ACM 15*, 12 (December 1972): 1053 – 1058.

[Stevens 1974]
Stevens, W. P.; Myers, G. J.; & Constantine, L. L. "Structured Design." *IBM Systems Journal 13*, 2 (1974): 115-139. Reprinted in *IBM Systems Journal 38*, 2/3 (1999): 231-257.

[Wirfs-Brock 2003]
Wirfs-Brock, R. & McKean, A. *Object Design: Roles, Responsibilities, and Collaborations*. Boston, MA: Addison-Wesley, 2003 (ISBN: 978-0-201-37943-3).

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	September 2007	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Modifiability Tactics	FA8721-05-C-0003

6. AUTHOR(S)

Felix Bachmann, Len Bass, & Robert Nord

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2007-TR-002

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	ESC-TR-2007-002

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

An architectural tactic is a design decision that affects how well a software architecture addresses a particular quality attribute. This report describes how tactics are based on the parameters of quality attribute models. Tactics provide an architectural means of adjusting those parameters, which, in turn, can improve the quality-attribute-specific behavior of the resulting system.

This report justifies the tactics for modifiability, using established concepts of coupling, cohesion, and cost motivations as the means of identifying parameters of interest. Various tactics are then described based on their ability to control these parameters. The report also describes a standard set of architectural patterns and their variants in terms of the use of these tactics.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Software architecture, architectural tactics, architectural design, architectural analysis, architectural patterns, modifiability	62

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL