7. Modifiability

Adapt or perish, now as ever, is nature’s inexorable imperative.
—H.G. Wells

Change happens.

Study after study shows that most of the cost of the typical software system occurs after it has been initially released. If change is the only constant in the universe, then software change is not only constant but ubiquitous. Changes happen to add new features, to change or even retire old ones. Changes happen to fix defects, tighten security, or improve performance. Changes happen to enhance the user’s experience. Changes happen to embrace new technology, new platforms, new protocols, new standards. Changes happen to make systems work together, even if they were never designed to do so.

Modifiability is about change, and our interest in it centers on the cost and risk of making changes. To plan for modifiability, an architect has to consider four questions:

- **What can change?** A change can occur to any aspect of a system: the functions that the system computes, the platform (the hardware, operating system, middleware), the environment in which the system operates (the systems with which it must interoperate, the protocols it uses to communicate with the rest of the world), the qualities the system exhibits (its performance, its reliability, and even its future modifications), and its capacity (number of users supported, number of simultaneous operations).

- **What is the likelihood of the change?** One cannot plan a system for all potential changes—the system would never be done, or if it was done it would be far too expensive and would likely suffer quality attribute problems in other dimensions. Although anything might change, the architect has to make the tough decisions about which changes are likely, and hence which changes are to be supported, and which are not.

- **When is the change made and who makes it?** Most commonly in the past, a change was made to source code. That is, a developer had to make the change, which was tested and then deployed in a new release. Now, however, the question of when a change is made is intertwined with the question of who makes it. An end user changing the screen saver is clearly
making a change to one of the aspects of the system. Equally clear, it is not in the same category as changing the system so that it can be used over the web rather than on a single machine. Changes can be made to the implementation (by modifying the source code), during compile (using compile-time switches), during build (by choice of libraries), during configuration setup (by a range of techniques, including parameter setting), or during execution (by parameter settings, plugins, etc.). A change can also be made by a developer, an end user, or a system administrator.

- **What is the cost of the change?** Making a system more modifiable involves two types of cost:
  - The cost of introducing the mechanism(s) to make the system more modifiable
  - The cost of making the modification using the mechanism(s)

For example, the simplest mechanism for making a change is to wait for a change request to come in, then change the source code to accommodate the request. The cost of introducing the mechanism is zero; the cost of exercising it is the cost of changing the source code and revalidating the system. At the other end of the spectrum is an application generator, such as a user interface builder. The builder takes as input a description of the designer user interface produced through direct manipulation techniques and produces (usually) source code. The cost of introducing the mechanism is the cost of constructing the UI builder, which can be substantial. The cost of using the mechanism is the cost of producing the input to feed the builder (cost can be substantial or negligible), the cost of running the builder (approximately zero), and then the cost of whatever testing is performed on the result (usually much less than usual).

For \( N \) similar modifications, a simplified justification for a change mechanism is that

\[
N \times \text{Cost of making the change without the mechanism} \leq \text{Cost of installing the mechanism} + (N \times \text{Cost of making the change using the mechanism}).
\]

\( N \) is the anticipated number of modifications that will use the modifiability mechanism, but \( N \) is a prediction. If fewer changes than expected come in, then an expensive modification mechanism may not be warranted. In addition, the cost of creating the modifiability mechanism could be applied elsewhere—in adding functionality, in improving the performance, or even in nonsoftware investments such as buying tech stocks. Also, the equation does not take time
into account. It might be cheaper in the long run to build a sophisticated change-handling mechanism, but you might not be able to wait for that.

7.1. Modifiability General Scenario

From these considerations, we can see the portions of the modifiability general scenario:

• **Source of stimulus.** This portion specifies who makes the change: the developer, a system administrator, or an end user.

• **Stimulus.** This portion specifies the change to be made. A change can be the addition of a function, the modification of an existing function, or the deletion of a function. (For this categorization, we regard fixing a defect as changing a function, which presumably wasn’t working correctly as a result of the defect.) A change can also be made to the qualities of the system: making it more responsive, increasing its availability, and so forth. The capacity of the system may also change. Accommodating an increasing number of simultaneous users is a frequent requirement. Finally, changes may happen to accommodate new technology of some sort, the most common of which is porting the system to a different type of computer or communication network.

• **Artifact.** This portion specifies what is to be changed: specific components or modules, the system’s platform, its user interface, its environment, or another system with which it interoperates.

• **Environment.** This portion specifies when the change can be made: design time, compile time, build time, initiation time, or runtime.

• **Response.** Make the change, test it, and deploy it.

• **Response measure.** All of the possible responses take time and cost money; time and money are the most common response measures. Although both sound simple to measure, they aren’t. You can measure calendar time or staff time. But do you measure the time it takes for the change to wind its way through configuration control boards and approval authorities (some of whom may be outside your organization), or merely the time it takes your engineers to make the change? Cost usually means direct outlay, but it might also include opportunity cost of having your staff work on changes instead of other tasks. Other measures include the extent of the change (number of modules or other artifacts affected) or the number of new defects introduced by the change, or the effect on other quality attributes. If the change is being made by a user, you may wish to
measure the efficacy of the change mechanisms provided, which somewhat overlaps with measures of usability (see Chapter 11).

Figure 7.1 illustrates a concrete modifiability scenario: The developer wishes to change the user interface by modifying the code at design time. The modifications are made with no side effects within three hours.

Figure 7.1. Sample concrete modifiability scenario

Table 7.1 enumerates the elements of the general scenario that characterize modifiability.

Table 7.1. Modifiability General Scenario
7.2. Tactics for Modifiability

Tactics to control modifiability have as their goal controlling the complexity of making changes, as well as the time and cost to make changes. Figure 7.2 shows this relationship.

To understand modifiability, we begin with coupling and cohesion. Modules have responsibilities. When a change causes a module to be modified, its responsibilities are changed in some way. Generally, a change that affects one module is easier and less expensive than if it changes more than one
module. However, if two modules’ responsibilities overlap in some way, then a single change may well affect them both. We can measure this overlap by measuring the probability that a modification to one module will propagate to the other. This is called coupling, and high coupling is an enemy of modifiability.

Cohesion measures how strongly the responsibilities of a module are related. Informally, it measures the module’s “unity of purpose.” Unity of purpose can be measured by the change scenarios that affect a module. The cohesion of a module is the probability that a change scenario that affects a responsibility will also affect other (different) responsibilities. The higher the cohesion, the lower the probability that a given change will affect multiple responsibilities. High cohesion is good; low cohesion is bad. The definition allows for two modules with similar purposes each to be cohesive.

Given this framework, we can now identify the parameters that we will use to motivate modifiability tactics:

• **Size of a module.** Tactics that split modules will reduce the cost of making a modification to the module that is being split as long as the split is chosen to reflect the type of change that is likely to be made.

• **Coupling.** Reducing the strength of the coupling between two modules A and B will decrease the expected cost of any modification that affects A. Tactics that reduce coupling are those that place intermediaries of various sorts between modules A and B.

• **Cohesion.** If module A has a low cohesion, then cohesion can be improved by removing responsibilities unaffected by anticipated changes.

Finally we need to be concerned with when in the software development life cycle a change occurs. If we ignore the cost of preparing the architecture for the modification, we prefer that a change is bound as late as possible. Changes can only be successfully made (that is, quickly and at lowest cost) late in the life cycle if the architecture is suitably prepared to accommodate them. Thus the fourth and final parameter in a model of modifiability is this:

• **Binding time of modification.** An architecture that is suitably equipped to accommodate modifications late in the life cycle will, on average, cost less than an architecture that forces the same modification to be made earlier. The preparedness of the system means that some costs will be zero, or very low, for late life-cycle modifications. This, however, neglects the cost of preparing the architecture for the late binding.

Now we may understand tactics and their consequences as affecting one or
more of the previous parameters: reducing the size of a module, increasing cohesion, reducing coupling, and deferring binding time. These tactics are shown in Figure 7.3.

![Modifiability Tactics Diagram]

**Figure 7.3. Modifiability tactics**

**Reduce the Size of a Module**
- *Split module.* If the module being modified includes a great deal of capability, the modification costs will likely be high. Refining the module into several smaller modules should reduce the average cost of future changes.

**Increase Cohesion**
Several tactics involve moving responsibilities from one module to another. The purpose of moving a responsibility from one module to another is to reduce the likelihood of side effects affecting other responsibilities in the original module.
- *Increase semantic coherence.* If the responsibilities A and B in a module do not serve the same purpose, they should be placed in different modules. This may involve creating a new module or it may involve moving a responsibility to an existing module. One method for identifying responsibilities to be moved is to hypothesize likely changes that affect a module. If some responsibilities are not affected by these changes, then
those responsibilities should probably be removed.

Reduce Coupling

We now turn to tactics that reduce the coupling between modules.

- **Encapsulate.** Encapsulation introduces an explicit interface to a module. This interface includes an application programming interface (API) and its associated responsibilities, such as “perform a syntactic transformation on an input parameter to an internal representation.” Perhaps the most common modifiability tactic, encapsulation reduces the probability that a change to one module propagates to other modules. The strengths of coupling that previously went to the module now go to the interface for the module. These strengths are, however, reduced because the interface limits the ways in which external responsibilities can interact with the module (perhaps through a wrapper). The external responsibilities can now only directly interact with the module through the exposed interface (indirect interactions, however, such as dependence on quality of service, will likely remain unchanged). Interfaces designed to increase modifiability should be abstract with respect to the details of the module that are likely to change—that is, they should hide those details.

- **Use an intermediary** breaks a dependency. Given a dependency between responsibility A and responsibility B (for example, carrying out A first requires carrying out B), the dependency can be broken by using an intermediary. The type of intermediary depends on the type of dependency. For example, a publish-subscribe intermediary will remove the data producer’s knowledge of its consumers. So will a shared data repository, which separates readers of a piece of data from writers of that data. In a service-oriented architecture in which services discover each other by dynamic lookup, the directory service is an intermediary.

- **Restrict dependencies** is a tactic that restricts the modules that a given module interacts with or depends on. In practice this tactic is achieved by restricting a module’s visibility (when developers cannot see an interface, they cannot employ it) and by authorization (restricting access to only authorized modules). This tactic is seen in layered architectures, in which a layer is only allowed to use lower layers (sometimes only the next lower layer) and in the use of wrappers, where external entities can only see (and hence depend on) the wrapper and not the internal functionality that it wraps.

- **Refactor** is a tactic undertaken when two modules are affected by the
same change because they are (at least partial) duplicates of each other. Code refactoring is a mainstay practice of Agile development projects, as a cleanup step to make sure that teams have not produced duplicative or overly complex code; however, the concept applies to architectural elements as well. Common responsibilities (and the code that implements them) are “factored out” of the modules where they exist and assigned an appropriate home of their own. By co-locating common responsibilities—that is, making them submodules of the same parent module—the architect can reduce coupling.

- **Abstract common services.** In the case where two modules provide not- quite-the-same but similar services, it may be cost-effective to implement the services just once in a more general (abstract) form. Any modification to the (common) service would then need to occur just in one place, reducing modification costs. A common way to introduce an abstraction is by parameterizing the description (and implementation) of a module’s activities. The parameters can be as simple as values for key variables or as complex as statements in a specialized language that are subsequently interpreted.

**Defer Binding**

Because the work of people is almost always more expensive than the work of computers, letting computers handle a change as much as possible will almost always reduce the cost of making that change. If we design artifacts with built-in flexibility, then exercising that flexibility is usually cheaper than hand-coding a specific change.

Parameters are perhaps the best-known mechanism for introducing flexibility, and that is reminiscent of the abstract common services tactic. A parameterized function \( f(a, b) \) is more general than the similar function \( f(a) \) that assumes \( b = 0 \). When we bind the value of some parameters at a different phase in the life cycle than the one in which we defined the parameters, we are applying the defer binding tactic.

In general, the later in the life cycle we can bind values, the better. However, putting the mechanisms in place to facilitate that late binding tends to be more expensive—yet another tradeoff. And so the equation on page 118 comes into play. We want to bind as late as possible, as long as the mechanism that allows it is cost-effective.

Tactics to bind values at compile time or build time include these:
• Component replacement (for example, in a build script or makefile)
• Compile-time parameterization
• Aspects

Tactics to bind values at deployment time include this:
• Configuration-time binding

Tactics to bind values at startup or initialization time include this:
• Resource files

Tactics to bind values at runtime include these:
• Runtime registration
• Dynamic lookup (e.g., for services)
• Interpret parameters
• Startup time binding
• Name servers
• Plug-ins
• Publish-subscribe
• Shared repositories
• Polymorphism

Separating building a mechanism for modifiability from using the mechanism to make a modification admits the possibility of different stakeholders being involved—one stakeholder (usually a developer) to provide the mechanism and another stakeholder (an installer, for example, or a user) to exercise it later, possibly in a completely different life-cycle phase. Installing a mechanism so that someone else can make a change to the system without having to change any code is sometimes called *externalizing* the change.

### 7.3. A Design Checklist for Modifiability

*Table 7.2* is a checklist to support the design and analysis process for modifiability.

*Table 7.2. Checklist to Support the Design and Analysis Process for Modifiability*
<table>
<thead>
<tr>
<th>Category</th>
<th>Checklist</th>
</tr>
</thead>
</table>
| Allocation of Responsibilities| Determine which changes or categories of changes are likely to occur through consideration of changes in technical, legal, social, business, and customer forces. For each potential change or category of changes:  
  - Determine the responsibilities that would need to be added, modified, or deleted to make the change.  
  - Determine what responsibilities are impacted by the change.  
  - Determine an allocation of responsibilities to modules that places, as much as possible, responsibilities that will be changed (or impacted by the change) together in the same module, and places responsibilities that will be changed at different times in separate modules. |
| Coordination Model            | Determine which functionality or quality attribute can change at runtime and how this affects coordination; for example, will the information being communicated change at runtime, or will the communication protocol change at runtime? If so, ensure that such changes affect a small number set of modules.  
  Determine which devices, protocols, and communication paths used for coordination are likely to change. For those devices, protocols, and communication paths, ensure that the impact of changes will be limited to a small set of modules.  
  For those elements for which modifiability is a concern, use a coordination model that reduces coupling such as publish-subscribe, defers bindings such as enterprise service bus, or restricts dependencies such as broadcast. |
| Data Model                    | Determine which changes (or categories of changes) to the data abstractions, their operations, or their properties are likely to occur. Also determine which changes or categories of changes to these data abstractions will involve their creation, initialization, persistence, manipulation, translation, or destruction. |
For each change or category of change, determine if the changes will be made by an end user, a system administrator, or a developer. For those changes to be made by an end user or system administrator, ensure that the necessary attributes are visible to that user and that the user has the correct privileges to modify the data, its operations, or its properties.

For each potential change or category of change:

- Determine which data abstractions would need to be added, modified, or deleted to make the change.
- Determine whether there would be any changes to the creation, initialization, persistence, manipulation, translation, or destruction of these data abstractions.
- Determine which other data abstractions are impacted by the change. For these additional data abstractions, determine whether the impact would be on the operations, their properties, their creation, initialization, persistence, manipulation, translation, or destruction.
- Ensure an allocation of data abstractions that minimizes the number and severity of modifications to the abstractions by the potential changes.

Design your data model so that items allocated to each element of the data model are likely to change together.

Mapping among Architectural Elements

Determine if it is desirable to change the way in which functionality is mapped to computational elements (e.g., processes, threads, processors) at runtime, compile time, design time, or build time. Determine the extent of modifications necessary to accommodate the addition, deletion, or modification of a function or a quality attribute. This might involve a determination of the following, for example:

- Execution dependencies
- Assignment of data to databases
- Assignment of runtime elements to processes, threads, or processors
7.4. Summary

Modifiability deals with change and the cost in time or money of making a change, including the extent to which this modification affects other functions or quality attributes.

Changes can be made by developers, installers, or end users, and these changes need to be prepared for. There is a cost of preparing for change as well as a cost of making a change. The modifiability tactics are designed to prepare for subsequent changes.

Tactics to reduce the cost of making a change include making modules smaller, increasing cohesion, and reducing coupling. Deferring binding will also
reduce the cost of making a change.

Reducing coupling is a standard category of tactics that includes encapsulating, using an intermediary, restricting dependencies, co-locating related responsibilities, refactoring, and abstracting common services.

Increasing cohesion is another standard tactic that involves separating responsibilities that do not serve the same purpose.

Defer binding is a category of tactics that affect build time, load time, initialization time, or runtime.

7.5. For Further Reading

Serious students of software engineering should read two early papers about designing for modifiability. The first is Edsger Dijkstra’s 1968 paper about the T.H.E. operating system [Dijkstra 68], which is the first paper that talks about designing systems to be layered, and the modifiability benefits it brings. The second is David Parnas’s 1972 paper that introduced the concept of information hiding [Parnas 72]. Parnas prescribed defining modules not by their functionality but by their ability to internalize the effects of changes.

The tactics that we have presented in this chapter are a variant on those introduced by [Bachmann 07].

Additional tactics for modifiability within the avionics domain can be found in [EOSAN 07], published by the European Organization for the Safety of Air Navigation.

7.6. Discussion Questions

1. Modifiability comes in many flavors and is known by many names. Find one of the IEEE or ISO standards dealing with quality attributes and compile a list of quality attributes that refer to some form of modifiability. Discuss the differences.

2. For each quality attribute that you discovered as a result of the previous question, write a modifiability scenario that expresses it.

3. In a certain metropolitan subway system, the ticket machines accept cash but do not give change. There is a separate machine that dispenses change but does not sell tickets. In an average station there are six or eight ticket machines for every change machine. What modifiability tactics do you see at work in this arrangement? What can you say about availability?

4. For the subway system in the previous question, describe the specific form
of modifiability (using a modifiability scenario) that seems to be the aim of arranging the ticket and change machines as described.

5. A *wrapper* is a common aid to modifiability. A wrapper for a component is the only element allowed to use that component; every other piece of software uses the component’s services by going through the wrapper. The wrapper transforms the data or control information for the component it wraps. For example, a component may expect input using English measures but find itself in a system in which all of the other components produce metric measures. A wrapper could be employed to translate. What modifiability tactics does a wrapper embody?

6. Once an intermediary has been introduced into an architecture, some modules may attempt to circumvent it, either inadvertently (because they are not aware of the intermediary) or intentionally (for performance, for convenience, or out of habit). Discuss some architectural means to prevent inadvertent circumvention of an intermediary.

7. In some projects, *deployability* is an important quality attribute that measures how easy it is to get a new version of the system into the hands of its users. This might mean a trip to your auto dealer or transmitting updates over the Internet. It also includes the time it takes to install the update once it arrives. In projects that measure deployability separately, should the cost of a modification stop when the new version is ready to ship? Justify your answer.

8. The abstract common services tactic is intended to reduce coupling, but it also might reduce cohesion. Discuss.

9. Identify particular change scenarios for an automatic teller machine. What modifications would you make to your automatic teller machine design to accommodate these changes?