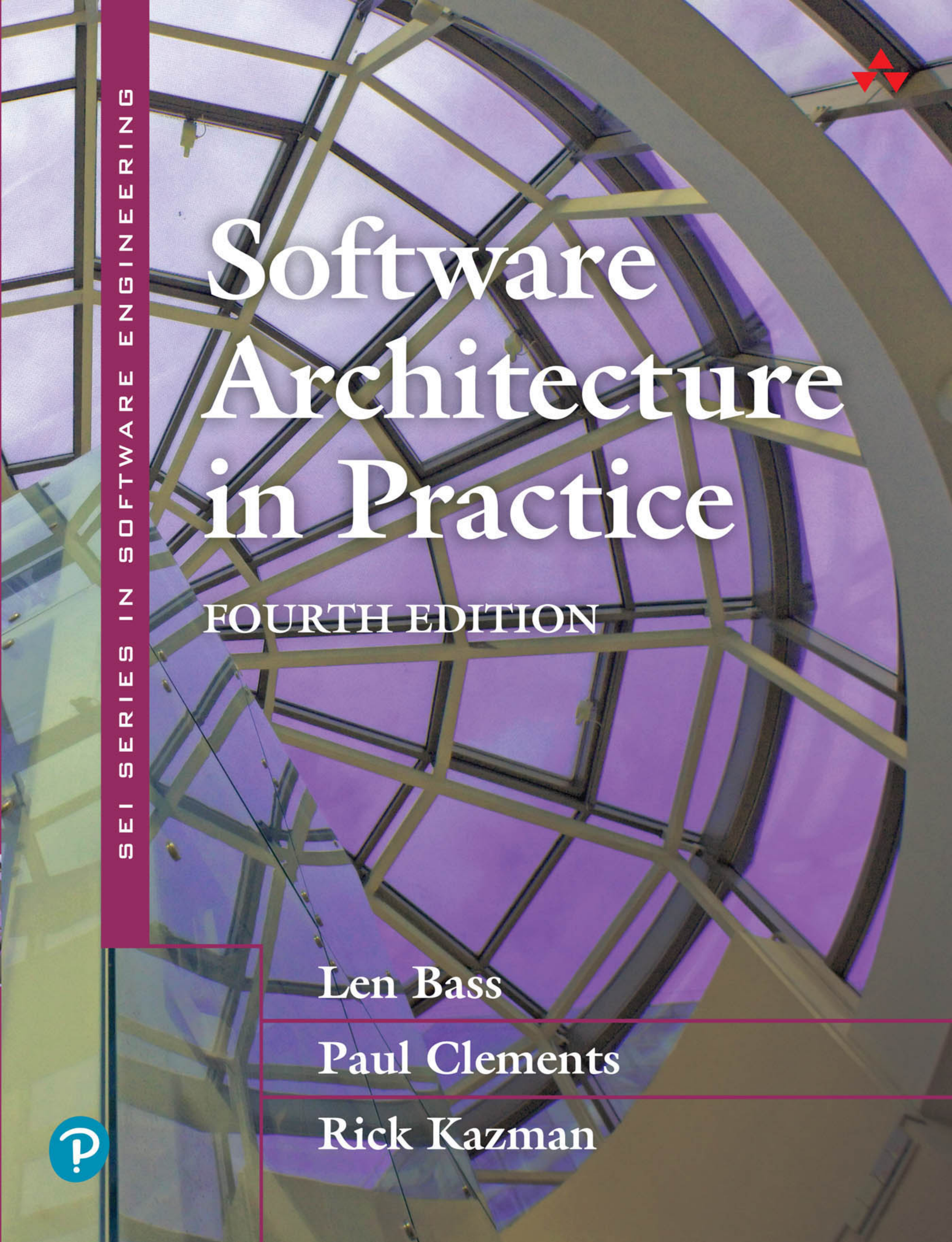

SEI SERIES IN SOFTWARE ENGINEERING

Software Architecture in Practice

FOURTH EDITION

Len Bass
Paul Clements
Rick Kazman

Software Architecture in Practice

Fourth Edition

Change is so prevalent in the life of software systems that special names have been given to specific flavors of modifiability. Some of the common ones are highlighted here:

- *Scalability* is about accommodating more of something. In terms of performance, scalability means adding more resources. Two kinds of performance scalability are horizontal scalability and vertical scalability. Horizontal scalability (scaling out) refers to adding more resources to logical units, such as adding another server to a cluster of servers. Vertical scalability (scaling-up) refers to adding more resources to a physical unit, such as adding more memory to a single computer. The problem that arises with either type of scaling is how to effectively utilize the additional resources. Being *effective* means that the additional resources result in a measurable improvement of some system quality, did not require undue effort to add, and did not unduly disrupt operations. In cloud-based environments, horizontal scalability is called *elasticity*. Elasticity is a property that enables a customer to add or remove virtual machines from the resource pool (see Chapter 17 for further discussion of such environments).
- *Variability* refers to the ability of a system and its supporting artifacts, such as code, requirements, test plans, and documentation, to support the production of a set of variants that differ from each other in a preplanned fashion. Variability is an especially important quality attribute in a product line, which is a family of systems that are similar but vary in features and functions. If the engineering assets associated with these systems can be shared among members of the family, then the overall cost of the product line plummets. This is achieved by introducing mechanisms that allow the artifacts to be selected and/or adapt to usages in the different product contexts that are within the product line's scope. The goal of variability in a software product line is to make it easy to build and maintain products in that family over a period of time.
- *Portability* refers to the ease with which software that was built to run on one platform can be changed to run on a different platform. Portability is achieved by minimizing platform dependencies in the software, isolating dependencies to well-identified locations, and writing the software to run on a "virtual machine" (for example, a Java Virtual Machine) that encapsulates all the platform dependencies. Scenarios describing portability deal with moving software to a new platform by expending no more than a certain level of effort or by counting the number of places in the software that would have to change. Architectural approaches to dealing with portability are intertwined with those for *deployability*, a topic addressed in Chapter 5.
- *Location independence* refers to the case where two pieces of distributed software interact and the location of one or both of the pieces is not known prior to runtime. Alternatively, the location of these pieces may change during runtime. In distributed systems, services are often deployed to arbitrary locations, and clients of those services must discover their location dynamically. In addition, services in a distributed system must often make their location discoverable once they have been deployed to a location. Designing the system for location independence means that the location will be easy to modify with minimal impact on the rest of the system.

8.1 Modifiability General Scenario

From these considerations, we can construct the general scenario for modifiability. Table 8.1 summarizes this scenario.

TABLE 8.1 General Scenario for Modifiability

Portion of Scenario	Description	Possible Values
Source	The agent that causes a change to be made. Most are human actors, but the system might be one that learns or self-modifies, in which case the source is the system itself.	End user, developer, system administrator, product line owner, the system itself
Stimulus	The change that the system needs to accommodate. (For this categorization, we regard fixing a defect as a change, to something that presumably wasn't working correctly.)	A directive to add/delete/modify functionality, or change a quality attribute, capacity, platform, or technology; a directive to add a new product to a product line; a directive to change the location of a service to another location
Artifacts	The artifacts that are modified. Specific components or modules, the system's platform, its user interface, its environment, or another system with which it interoperates.	Code, data, interfaces, components, resources, test cases, configurations, documentation
Environment	The time or stage at which the change is made.	Runtime, compile time, build time, initiation time, design time
Response	Make the change and incorporate it into the system.	One or more of the following: ▪ Make modification ▪ Test modification ▪ Deploy modification ▪ Self-modify
Response measure	The resources that were expended to make the change.	Cost in terms of: ▪ Number, size, complexity of affected artifacts ▪ Effort ▪ Elapsed time ▪ Money (direct outlay or opportunity cost) ▪ Extent to which this modification affects other functions or quality attributes ▪ New defects introduced ▪ How long it took the system to adapt

Figure 8.1 illustrates a concrete modifiability scenario: *A developer wishes to change the user interface. This change will be made to the code at design time, it will take less than three hours to make and test the change, and no side effects will occur.*

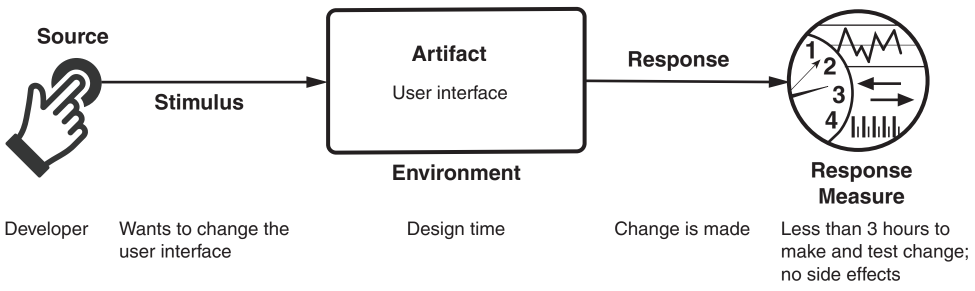

FIGURE 8.1 Sample concrete modifiability scenario

8.2 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 8.2 shows this relationship.

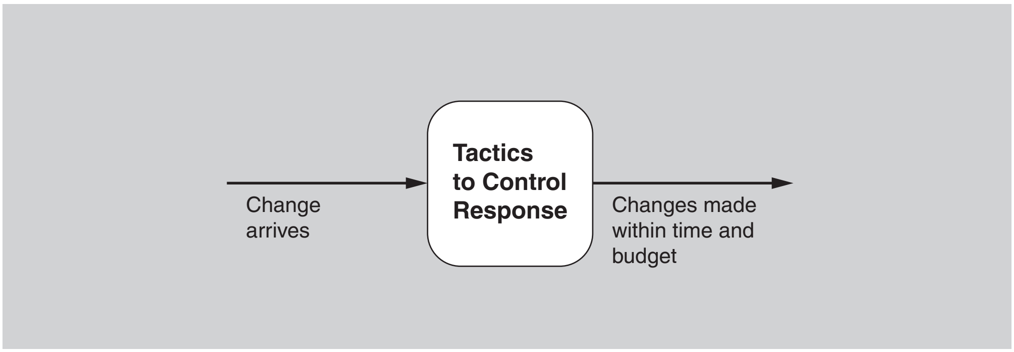
Change arrives
Tactics to Control Response
Changes made within time and budget

FIGURE 8.2 Goal of modifiability tactics

To understand modifiability, we begin with some of the earliest and most fundamental complexity measures of software design—coupling and cohesion—which were first described in the 1960s.

Generally, a change that affects one module is easier and less expensive than a change that affects more than one module. However, if two modules' responsibilities overlap in some way, then a single change may well affect them both. We can quantify this overlap by measuring the probability that a modification to one module will propagate to the other. This relationship is called *coupling*, and high coupling is an enemy of modifiability. Reducing the coupling between two modules will decrease the expected cost of any modification that affects either one. Tactics that reduce coupling are those that place intermediaries of various sorts between the two otherwise highly coupled modules.

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 modules. High cohesion is good for modifiability; low cohesion is bad for it. If module A has a low cohesion, then cohesion can be improved by removing responsibilities unaffected by anticipated changes.

A third characteristic that affects the cost and complexity of a change is the *size of a module*. All other things being equal, larger modules are more difficult and more costly to change, and are more prone to have bugs.

Finally, we need to be concerned with the point in the software development life cycle where 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 be successfully made (i.e., quickly and at low cost) late in the life cycle only if the architecture is suitably prepared to accommodate them. Thus the fourth and final parameter in a model of modifiability is *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 modifications that occur late in the life cycle.

Now we can understand tactics and their consequences as affecting one or more of these parameters: reducing size, increasing cohesion, reducing coupling, and deferring binding time. These tactics are shown in Figure 8.3.

Increase Cohesion

Several tactics involve redistributing responsibilities among modules. This step is taken to reduce the likelihood that a single change will affect multiple modules.

- *Split module*. If the module being modified includes responsibilities that are not cohesive, the modification costs will likely be high. Refactoring the module into several more cohesive modules should reduce the average cost of future changes. Splitting a module should not simply consist of placing half of the lines of code into each submodule; instead, it should sensibly and appropriately result in a series of submodules that are cohesive on their own.
- *Redistribute responsibilities*. If responsibilities A, A′, and A″ (all similar responsibilities) are sprinkled across several distinct modules, they should be placed together. This

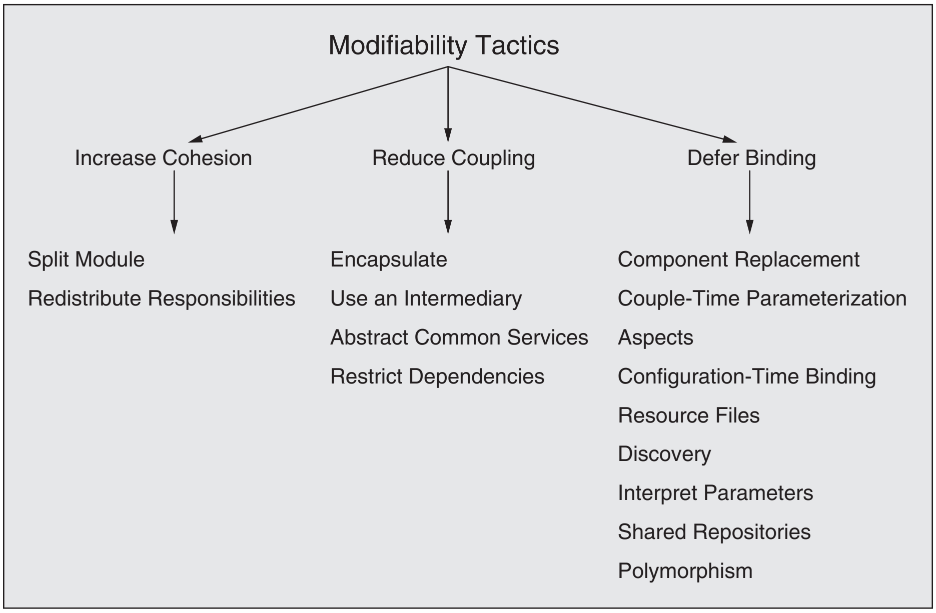

FIGURE 8.3 Modifiability tactics

refactoring may involve creating a new module, or it may involve moving responsibilities to existing modules. One method for identifying responsibilities to be moved is to hypothesize a set of likely changes as scenarios. If the scenarios consistently affect just one part of a module, then perhaps the other parts have separate responsibilities and should be moved. Alternatively, if some scenarios require modifications to multiple modules, then perhaps the responsibilities affected should be grouped together into a new module.

Reduce Coupling

We now turn to tactics that reduce the coupling between modules. These tactics overlap with the integrability tactics described in Chapter 7, because reducing dependencies among independent components (for integrability) is similar to reducing coupling among modules (for modifiability).

- *Encapsulate*. See the discussion in Chapter 7.
- *Use an intermediary*. See the discussion in Chapter 7.
- *Abstract common services*. See the discussion in Chapter 7.

- *Restrict dependencies.* This tactic restricts which modules a given module interacts with or depends on. In practice, this tactic is implemented 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). The restrict dependencies tactic is seen in layered architectures, in which a layer is allowed to use only lower layers (sometimes only the next lower layer), and with the use of wrappers, where external entities can see (and hence depend on) only the wrapper, and not the internal functionality that it wraps.

Defer Binding

Because the work of people is almost always more expensive error-prone 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 their use 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 deferring binding.

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—a well-known tradeoff. And so the equation given earlier in the chapter comes into play. We want to bind as late as possible, as long as the mechanism that allows it is cost-effective.

The following tactics can be used to bind values at compile time or build time:

- Component replacement (for example, in a build script or makefile)
- Compile-time parameterization
- Aspects

The following tactics are available to bind values at deployment, startup time, or initialization time:

- Configuration-time binding
- Resource files

Tactics to bind values at runtime include the following:

- Discovery (see Chapter 7)
- Interpret parameters
- Shared repositories
- Polymorphism

Separating the building of a mechanism for modifiability from the use of that 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 administrator or installer) to exercise it later, possibly in a completely different life-cycle