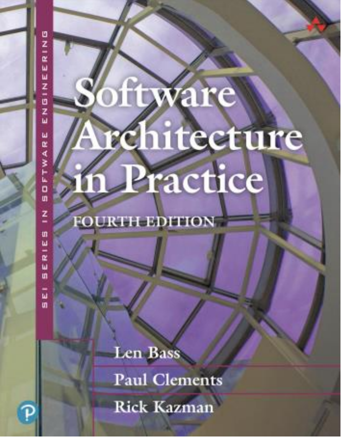
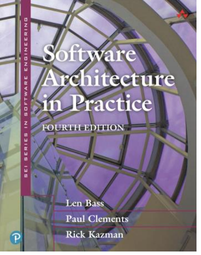

Software Architecture in Practice
FOURTH EDITION
SEI SERIES IN SOFTWARE ENGINEERING
Len Bass
Paul Clements
Rick Kazman

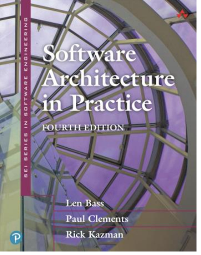
Chapter 8: Modifiability

It is not the strongest of the species that survive, nor the most intelligent, but the one most responsive to change.
—Charles Darwin



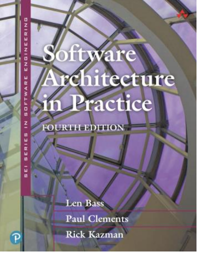
Chapter Outline

- What is Modifiability?
- Modifiability General Scenario
- Tactics for Modifiability
- Tactics-Based Questionnaire for Modifiability
- Patterns for Modifiability
- Summary



What is Modifiability?

- Modifiability is about change and our interest in it is in the cost and risk of making changes.
- To plan for modifiability, an architect has to consider three questions:
  - What can change?
  - What is the likelihood of the change?
  - When is the change made and who makes it?

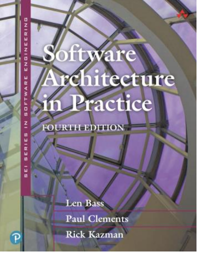


What is Modifiability?

- For *N* similar modifications, a simplified justification for a change mechanism is that

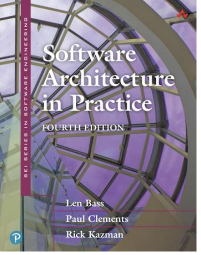
  N * Cost of making change without the mechanism ≤

  Cost of creating the mechanism + (*N* * cost of making the change using the mechanism)



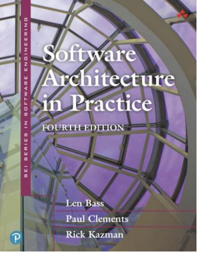
What is Modifiability?

- 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:
  - *Scalability*
  - *Variability*
  - *Portability*
  - *Location independence*



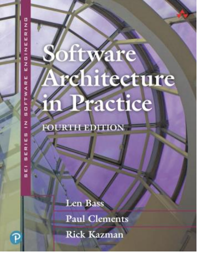
Modifiability General Scenario

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



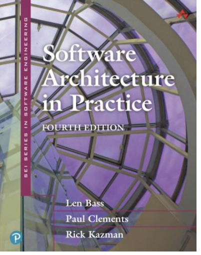
Modifiability General Scenario

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

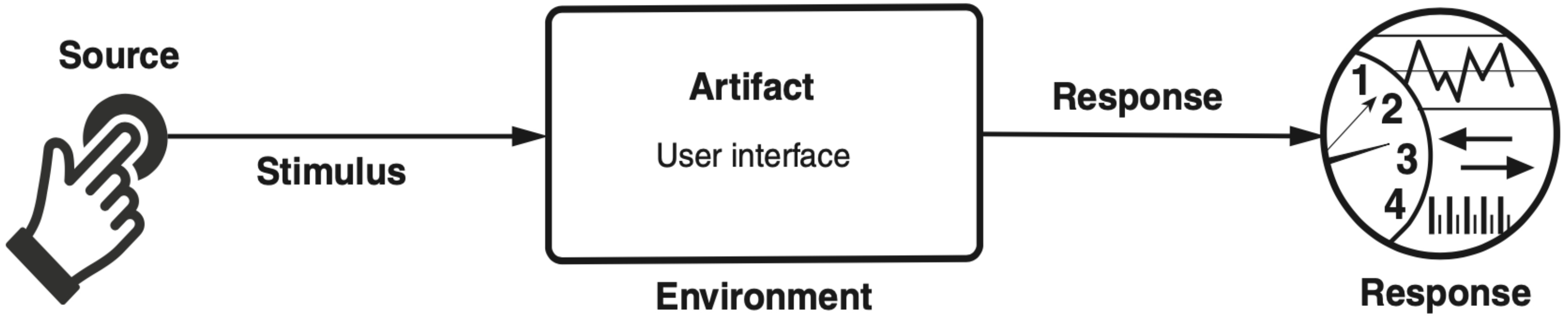


Sample Concrete Modifiability Scenario

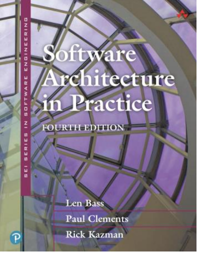
- *A developer wishes to change the user interface. This change will be made to the code at design time, it will take less than three hours to make and test the change, and no side effects will occur.*



Sample Concrete Modifiability Scenario

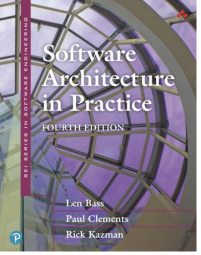


Developer	Wants to change the user interface	Design time	Change is made	Less than 3 hours to make and test change; no side effects

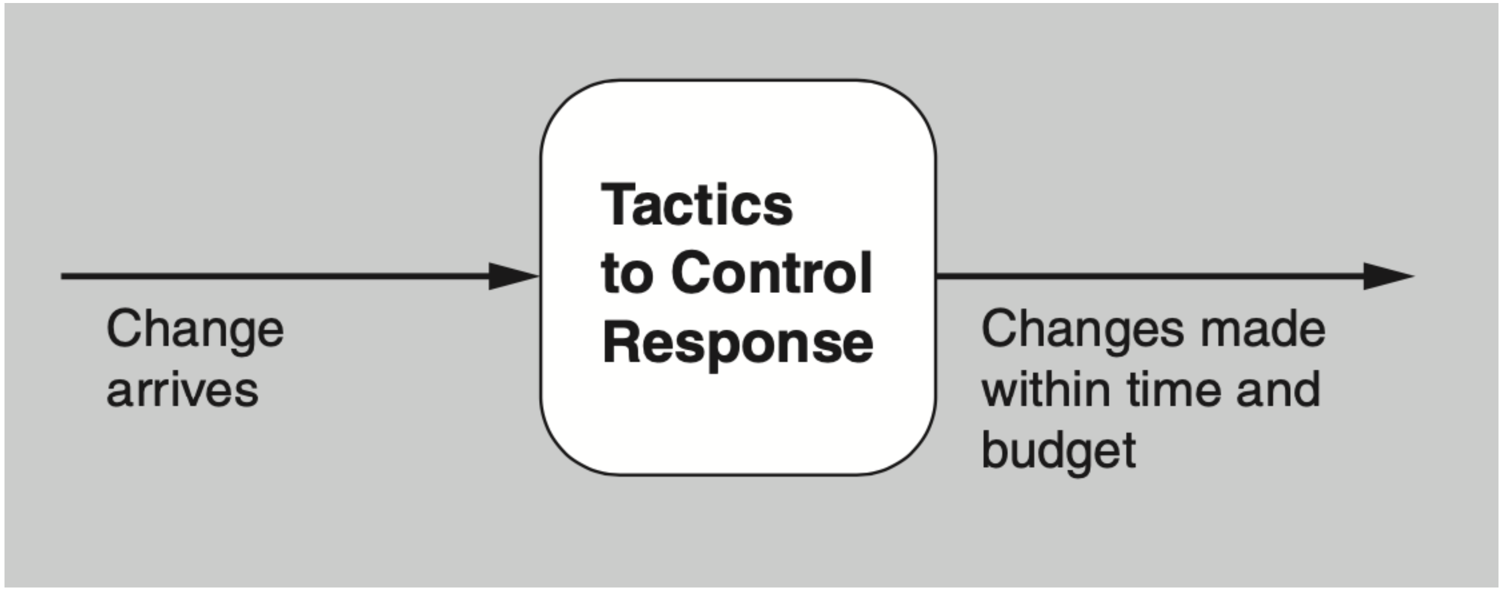


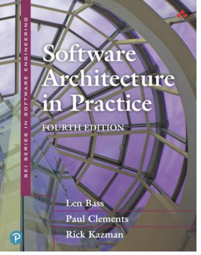
Goal of Modifiability Tactics

- Tactics to control modifiability have as their goal controlling the complexity of making changes, as well as the time and cost to make changes.

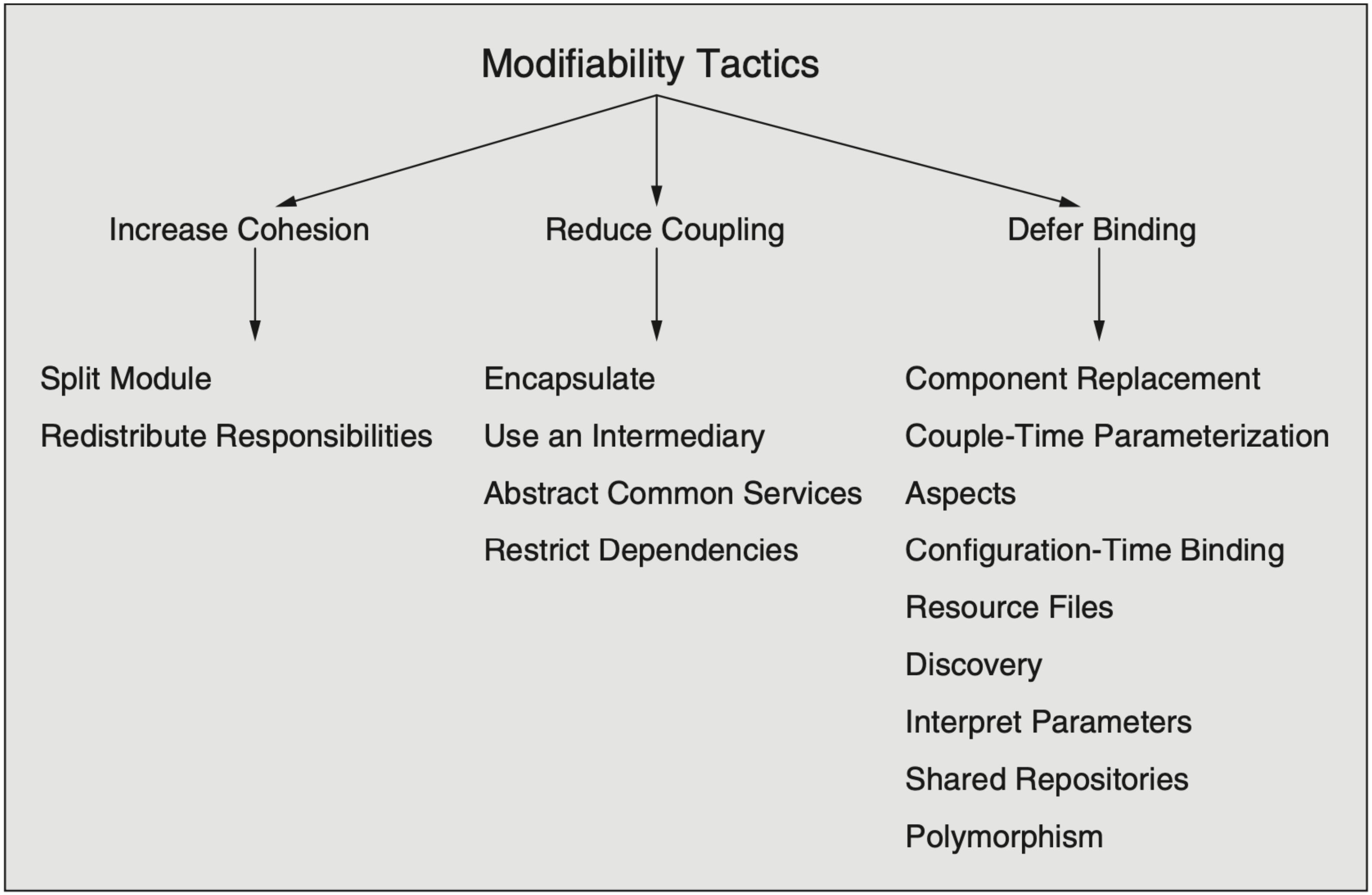


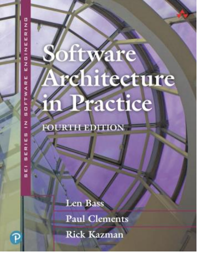
Goal of Modifiability Tactics


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



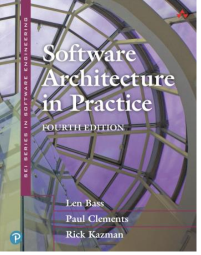
Modifiability Tactics





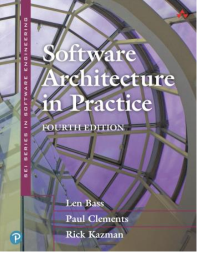
Increase Cohesion

- *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.
- *Redistribute responsibilities*. If responsibilities A, A', and A'' are sprinkled across several distinct modules, they should be placed together. This refactoring may involve creating a new module, or it may involve moving responsibilities to existing modules.



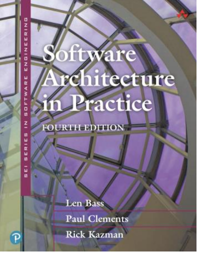
Reduce Coupling

- *Encapsulate*. See Chapter 7.
- *Use an intermediary*. See Chapter 7.
- *Abstract common services*. See 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).



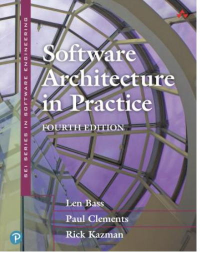
Defer Binding

- The following tactics can be used to bind values at compile time or build time:
  - Component replacement
  - Compile-time parameterization
  - Aspects
- The following tactics 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



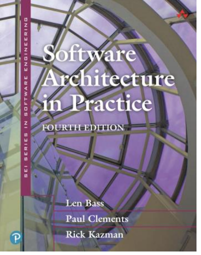
Tactics-Based Questionnaire for Modifiability

Tactics Group	Tactics Question	Supported? (Y/N)	Risk?	Design Decisions and Location	Rationale and Assumptions
Increase Cohesion	Do you make modules more cohesive by **splitting the module**? For example, if you have a large, complex module, can you split it into two (or more) more cohesive modules?				
	Do you make modules more cohesive by **redistributing responsibilities**? For example, if responsibilities in a module do not serve the same purpose, they should be placed in other modules.				



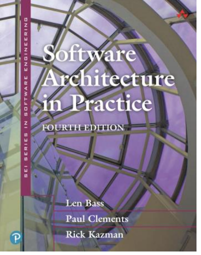
Tactics-Based Questionnaire for Modifiability

Reduce Coupling	Do you consistently **encapsulate** functionality? This typically involves isolating the functionality under scrutiny and introducing an explicit interface to it.
	Do you consistently **use an intermediary** to keep modules from being too tightly coupled? For example, if A calls concrete functionality C, you might introduce an abstraction B that mediates between A and C.
	Do you **restrict dependencies** between modules in a systematic way? Or is any system module free to interact with any other module?
	Do you **abstract common services**, in cases where you are providing several similar services? For example, this technique is often used when you want your system to be portable across operating systems, hardware, or other environmental variations.



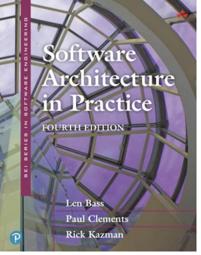
Tactics-Based Questionnaire for Modifiability

Defer Binding	Does the system regularly **defer binding** of important functionality so that it can be replaced later in the life cycle? For example, are there plug-ins, add-ons, resource files, or configuration files that can extend the functionality of the system?



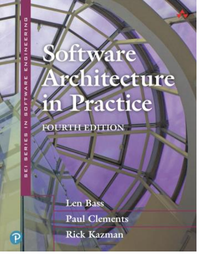
Client-Server Pattern for Modifiability

- The client-server pattern consists of a server providing services simultaneously to multiple distributed clients. The most common example is a web server providing information to multiple simultaneous users of a website.



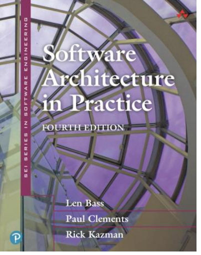
Client-Server Pattern Benefits

- Benefits:
  - The connection between a server and its clients is established dynamically. The server has no a priori knowledge of its clients—that is, there is low coupling between the server and its clients.
  - There is no coupling among the clients.
  - The number of clients can easily scale and is constrained only by the capacity of the server. The server functionality can also scale if its capacity is exceeded.
  - Clients and servers can evolve independently.
  - Common services can be shared among multiple clients.
  - The interaction with a user is isolated to the client.



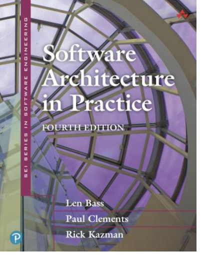
Client-Server Pattern Tradeoffs

- Tradeoffs:
  - This pattern is implemented such that communication occurs over a network, perhaps even the Internet. Thus messages may be delayed by network congestion, leading to degradation (or at least unpredictability) of performance.
  - For clients that communicate with servers over a network shared by other applications, special provisions must be made for achieving security (especially confidentiality) and maintaining integrity.



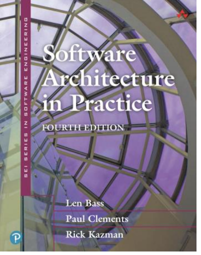
Plug-in (Microkernel) Pattern for Modifiability

- The plug-in pattern has two types of elements:
  - core elements, that provide a core set of functionality
  - plug-ins, that add functionality to the core via a fixed set of interfaces.
- The two types are typically bound together at build time or later.



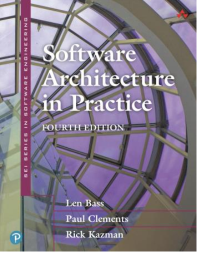
Plug-in (Microkernel) Pattern Benefits

- Plug-ins provide a controlled mechanism to extend a core product and make it useful in a variety of contexts.
- The plug-ins can be developed by different teams or organizations than the developers of the microkernel. This allows for the development of two different markets: for the core product and for the plug-ins.
- The plug-ins can evolve independently from the microkernel (as long as the interfaces do not change).



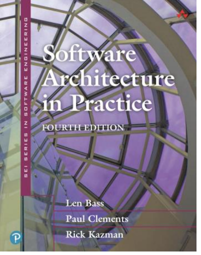
Plug-in (Microkernel) Pattern Tradeoffs

- Because plug-ins can be developed by different organizations, it is easier to introduce security vulnerabilities and privacy threats.



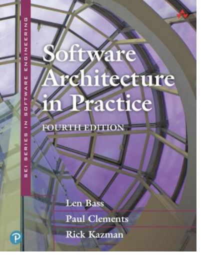
Layers Pattern for Modifiability

- The layers pattern divides the system in such a way that the modules can be developed and evolved separately with little interaction among the parts, which supports portability, modifi- ability, and reuse.
- To achieve this separation of concerns, the layers pattern divides the software into units called layers. Each layer is a grouping of modules that offers a cohesive set of services.
- The *allowed-to-use* relationship among the layers is subject to a key constraint: The relations must be unidirectional.
- Layers completely partition a set of software, and each partition is exposed through a public interface.



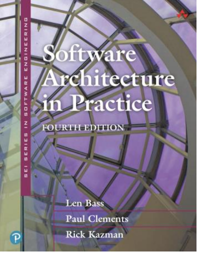
Layers Pattern Benefits

- Because a layer is constrained to use only lower layers, software in lower layers can be changed (as long as the interface does not change) without affecting the upper layers.
- Lower-level layers may be reused across different applications.
- Because the *allowed-to-use* relations are constrained, the number of interfaces that any team must understand is reduced.



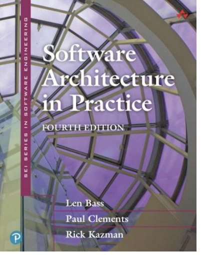
Layers Pattern Tradeoffs

- If the layering is not designed correctly, it may actually get in the way, by not providing the lower-level abstractions that programmers at the higher levels need.
- Layering often adds a performance penalty to a system. If a call is made from a function in the top-most layer, it may have to traverse many lower layers before being executed by the hardware.
- If many instances of layer bridging occur, the system may not meet its portability and modifiability goals, which strict layering helps to achieve.



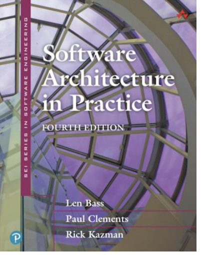
Publish-Subscribe Pattern for Modifiability

- Publish-subscribe is an architectural pattern in which components communicate primarily through asynchronous messages, sometimes referred to as “events” or “topics.”
- The publishers have no knowledge of the subscribers, and subscribers are only aware of message types.
- At runtime, when a message is published, the publish–subscribe (or event) bus notifies all of the elements that registered an interest in the event or topic.
- The result is loose coupling between the publishers and the subscribers.
- The publish-subscribe pattern has three types of elements:
  - *Publishers*
  - *Subscribers*
  - *Event bus*



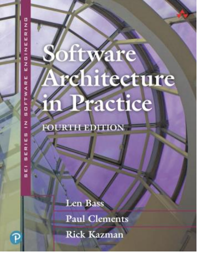
Publish-Subscribe Pattern Benefits

- Publishers and subscribers are independent and hence loosely coupled. Adding or changing subscribers requires only registering for an event and causes no changes to the publisher.
- System behavior can be easily changed by changing the event or topic of a message being published, and consequently which subscribers might receive and act on this message.
- Events can be logged easily to allow for record and playback.



Publish-Subscribe Pattern Tradeoffs

- Some implementations of the publish-subscribe pattern can negatively impact performance (latency). Use of a distributed coordination mechanism will ameliorate the performance degradation.
- In some cases, a component cannot be sure how long it will take to receive a published message. In general, system performance and resource management are more difficult to reason about in publish-subscribe systems.
- Use of this pattern can negatively impact determinism. The order in which methods are invoked, as a result of an event, may vary.
- Use of the publish-subscribe pattern can negatively impact testability. Seemingly small changes in the event bus can have a wide impact on system behavior and quality of service.
- Some publish-subscribe implementations limit the mechanisms available to implement security (integrity). Since publishers do not know the identity of subscribers, and vice versa, end-to-end encryption is limited. Messages from a publisher to the event bus can be encrypted, and messages from the event bus to a subscriber can be encrypted; but end-to-end encrypted communication requires all publishers and subscribers to share the same key.



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.
- 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.