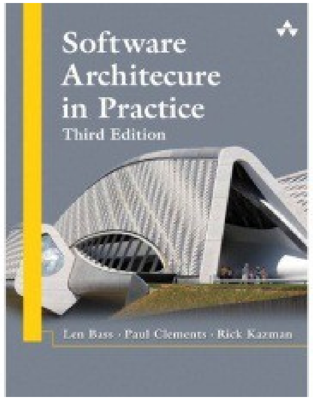
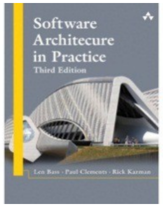


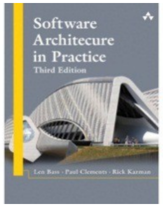
Chapter 13: Patterns and Tactics

Rob Pettit
SWE 443
Spring 2015



Chapter Outline

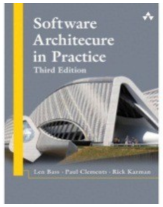
- What is a Pattern?
- Pattern Catalogue
  - Module patterns
  - Component and Connector Patterns
  - Allocation Patterns
- Relation Between Tactics and Patterns
- Using tactics together
- Summary



What is a Pattern?

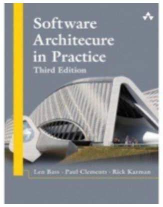
An architectural pattern establishes a relationship between:

- *A context*. A recurring, common situation in the world that gives rise to a problem.
- *A problem.* The problem, appropriately generalized, that arises in the given context.
- *A solution.* A successful architectural resolution to the problem, appropriately abstracted. The solution for a pattern is determined and described by:
  - A set of element types (for example, data repositories, processes, and objects)
  - A set of interaction mechanisms or connectors (for example, method calls, events, or message bus)
  - A topological layout of the components
  - A set of semantic constraints covering topology, element behavior, and interaction mechanisms



Relationships Between Tactics and Patterns

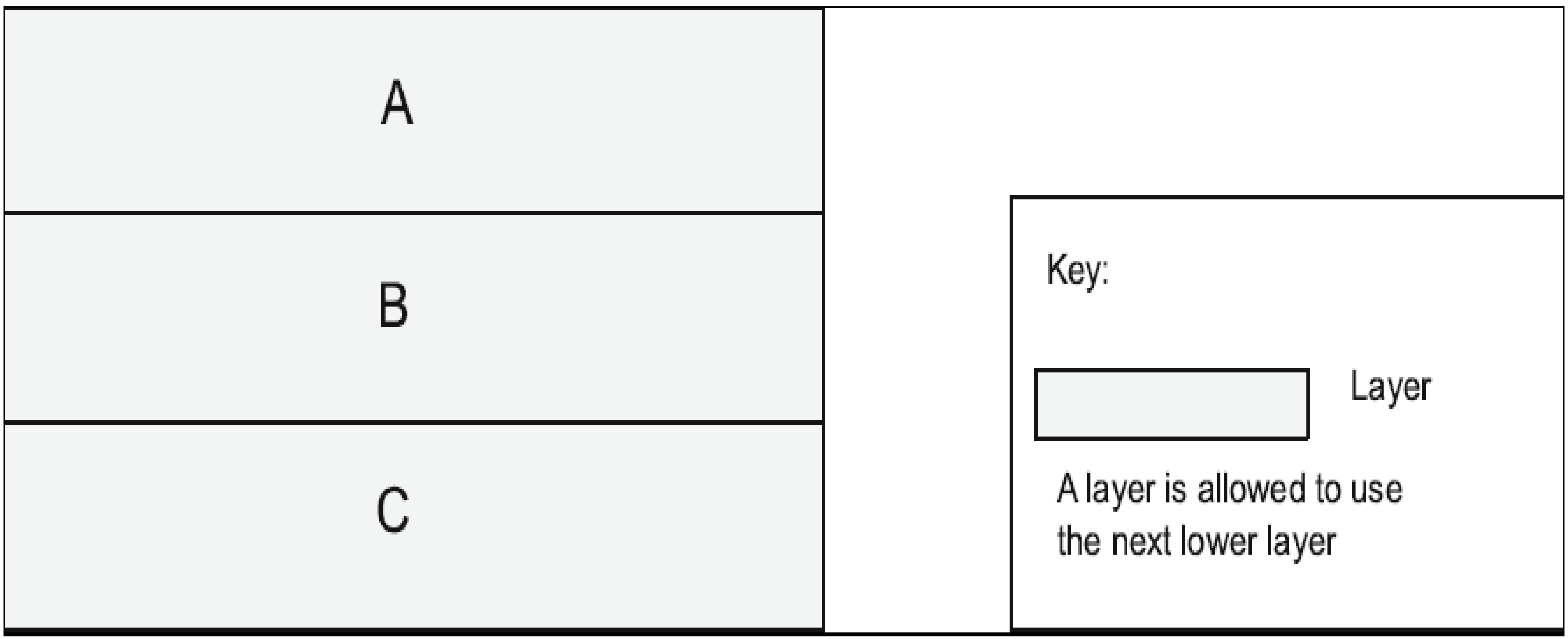
- Patterns are built from tactics; if a pattern is a molecule, a tactic is an atom.
- Model View Controller, for example utilizes the tactics:
  - Increase semantic coherence
  - Encapsulation
  - Use an intermediary
  - Use run time binding
- Tactics help to fine tune patterns
  - Address specific quality attributes and tradeoff decisions

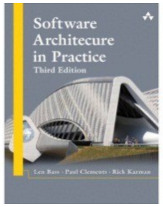


Layer Pattern

- **Context:** All complex systems experience the need to develop and evolve portions of the system independently. For this reason the developers of the system need a clear and well-documented separation of concerns, so that modules of the system may be independently developed and maintained.
- **Problem:** The software needs to be segmented in such a way that the modules can be developed and evolved separately with little interaction among the parts, supporting portability, modifiability, and reuse.
- **Solution:** To achieve this separation of concerns, the layered pattern divides the software into units called layers. Each layer is a grouping of modules that offers a cohesive set of services. The usage must be unidirectional. Layers completely partition a set of software, and each partition is exposed through a public interface.

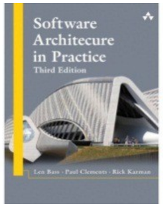
Layer Pattern Example





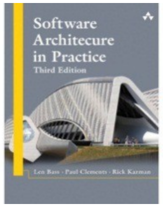
Layer Pattern Solution

- Overview: The layered pattern defines layers (groupings of modules that offer a cohesive set of services) and a unidirectional *allowed-to-use* relation among the layers.
- Elements: *Layer,* a kind of module. The description of a layer should define what modules the layer contains.
- Relations: *Allowed to use.* The design should define what the layer usage rules are and any allowable exceptions.
- Constraints:
  – Every piece of software is allocated to exactly one layer.
  – There are at least two layers (but usually there are three or more).
  – The *allowed-to-use* relations should not be circular (i.e., a lower layer cannot use a layer above).
- Weaknesses:
  – The addition of layers adds up-front cost and complexity to a system.
  – Layers contribute a performance penalty.

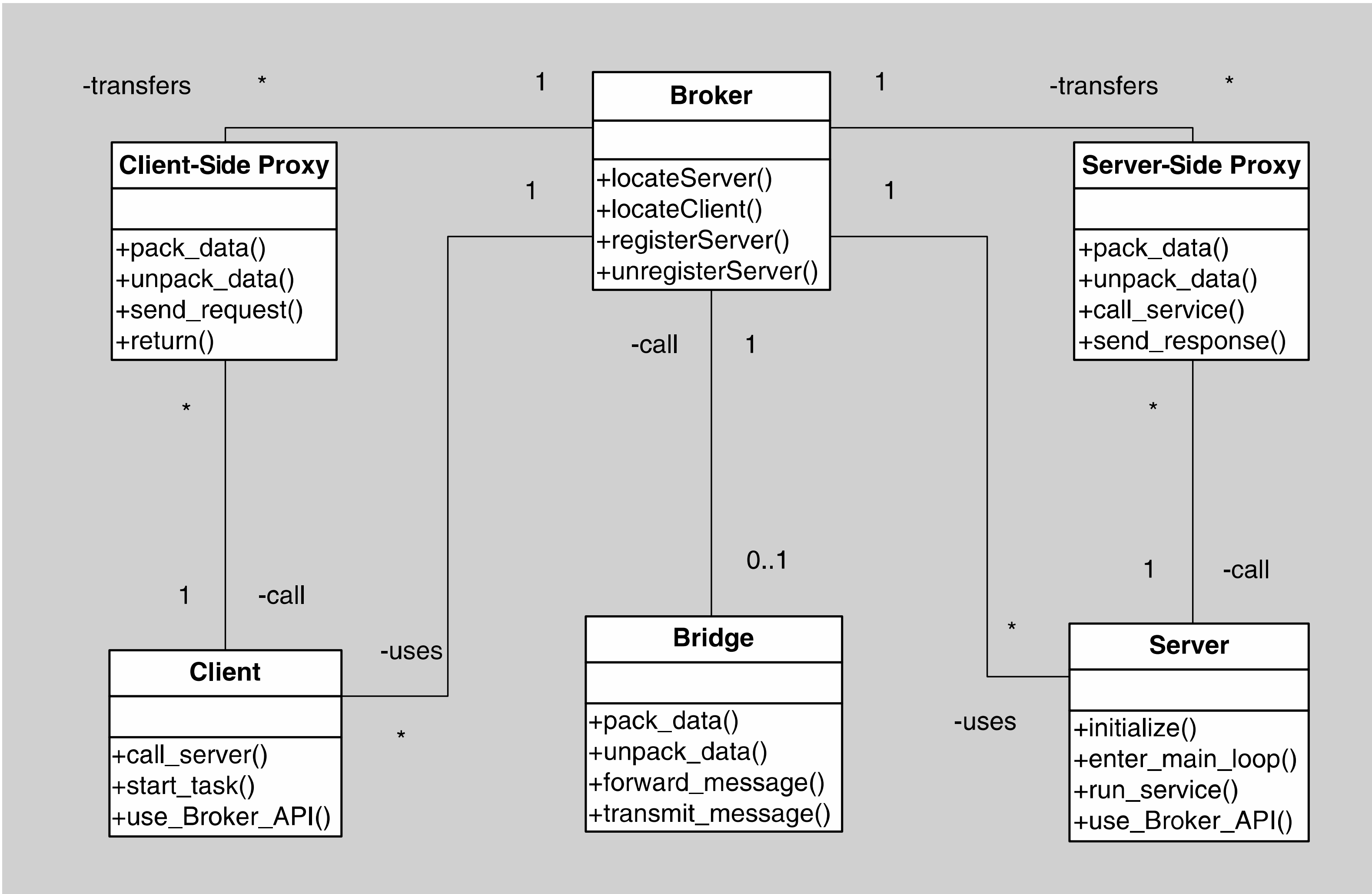


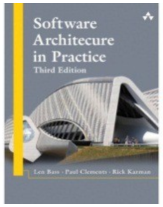
Broker Pattern

- **Context**: Many systems are constructed from a collection of services distributed across multiple servers. Implementing these systems is complex because you need to worry about how the systems will interoperate—how they will connect to each other and how they will exchange information—as well as the availability of the component services.
- **Problem**: How do we structure distributed software so that service users do not need to know the nature and location of service providers, making it easy to dynamically change the bindings between users and providers?
- **Solution**: The broker pattern separates users of services (clients) from providers of services (servers) by inserting an intermediary, called a broker. When a client needs a service, it queries a broker via a service interface. The broker then forwards the client's service request to a server, which processes the request.



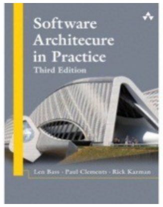
Broker Example





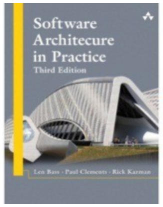
Broker Solution – 1

- Overview: The broker pattern defines a runtime component, called a broker, that mediates the communication between a number of clients and servers.
- Elements:
  - *Client,* a requester of services
  - *Server,* a provider of services
  - *Broker,* an intermediary that locates an appropriate server to fulfill a client's request, forwards the request to the server, and returns the results to the client
  - *Client-side proxy,* an intermediary that manages the actual communication with the broker, including marshaling, sending, and unmarshaling of messages
  - *Server-side proxy,* an intermediary that manages the actual communication with the broker, including marshaling, sending, and unmarshaling of messages



Broker Solution - 2

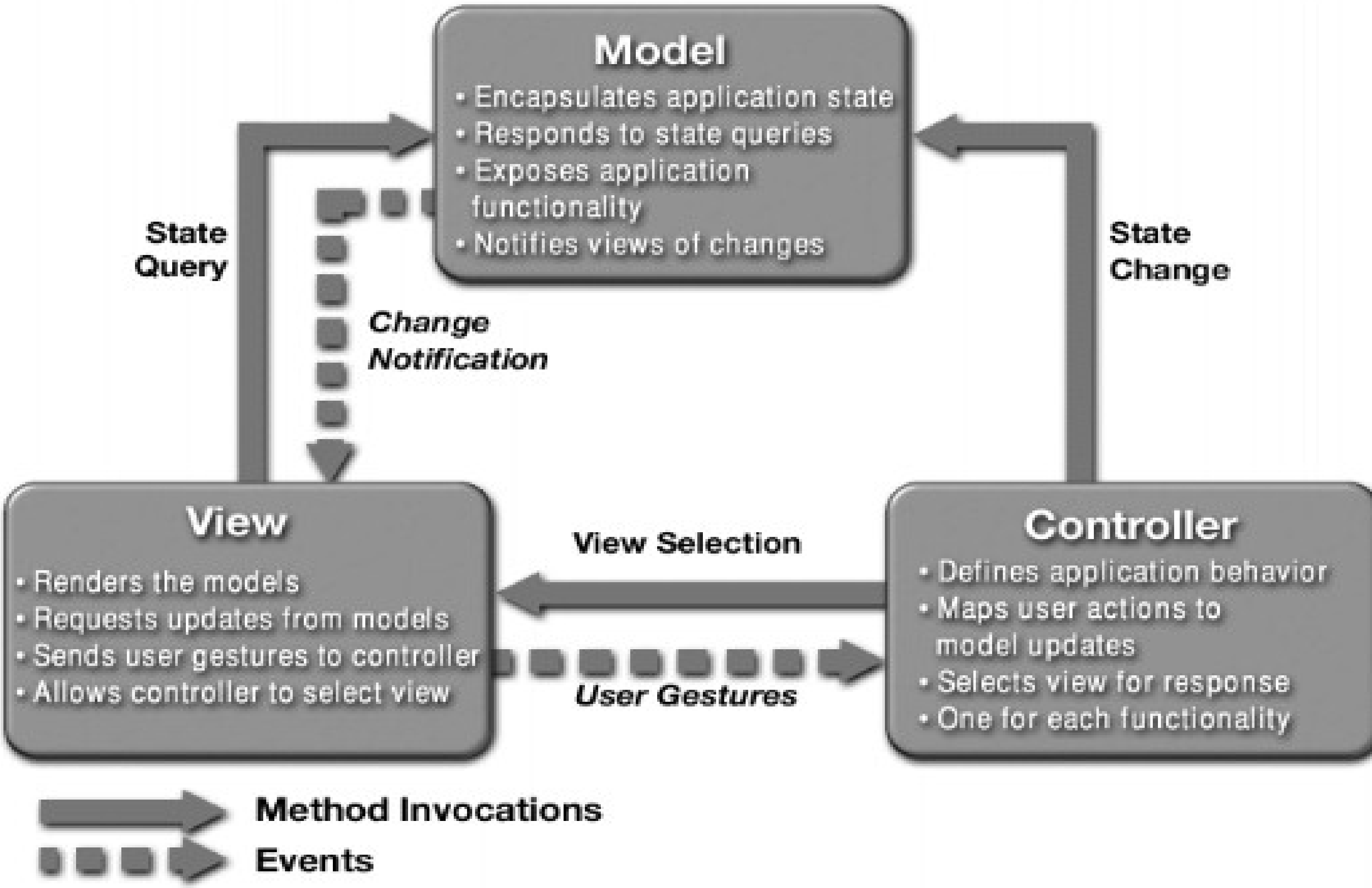
- Relations: The *attachment* relation associates clients (and, optionally, client-side proxies) and servers (and, optionally, server-side proxies) with brokers.
- Constraints: The client can only attach to a broker (potentially via a client-side proxy). The server can only attach to a broker (potentially via a server-side proxy).
- Weaknesses:
  – Brokers add a layer of indirection, and hence latency, between clients and servers, and that layer may be a communication bottleneck.
  – The broker can be a single point of failure.
  – A broker adds up-front complexity.
  – A broker may be a target for security attacks.
  – A broker may be difficult to test.

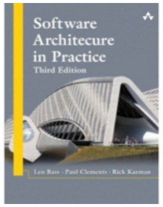


Model-View-Controller Pattern

- **Context**: User interface software is typically the most frequently modified portion of an interactive application. Users often wish to look at data from different perspectives, such as a bar graph or a pie chart. These representations should both reflect the current state of the data.
- **Problem**: How can user interface functionality be kept separate from application functionality and yet still be responsive to user input, or to changes in the underlying application's data? And how can multiple views of the user interface be created, maintained, and coordinated when the underlying application data changes?
- **Solution**: The model-view-controller (MVC) pattern separates application functionality into three kinds of components:
  - A model, which contains the application's data
  - A view, which displays some portion of the underlying data and interacts with the user
  - A controller, which mediates between the model and the view and manages the notifications of state changes

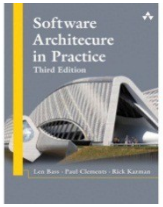
MVC Example


Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
Change Notification
State Change
View
• Renders the models
• Requests updates from models
• Sends user gestures to controller
• Allows controller to select view
View Selection
User Gestures
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• One for each functionality
Method Invocations
Events



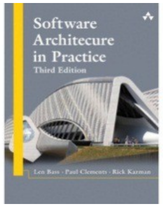
MVC Solution - 1

- Overview: The MVC pattern breaks system functionality into three components: a model, a view, and a controller that mediates between the model and the view.
- Elements:
  - The *model* is a representation of the application data or state, and it contains (or provides an interface to) application logic.
  - The *view* is a user interface component that either produces a representation of the model for the user or allows for some form of user input, or both.
  - The *controller* manages the interaction between the model and the view, translating user actions into changes to the model or changes to the view.



MVC Solution - 2

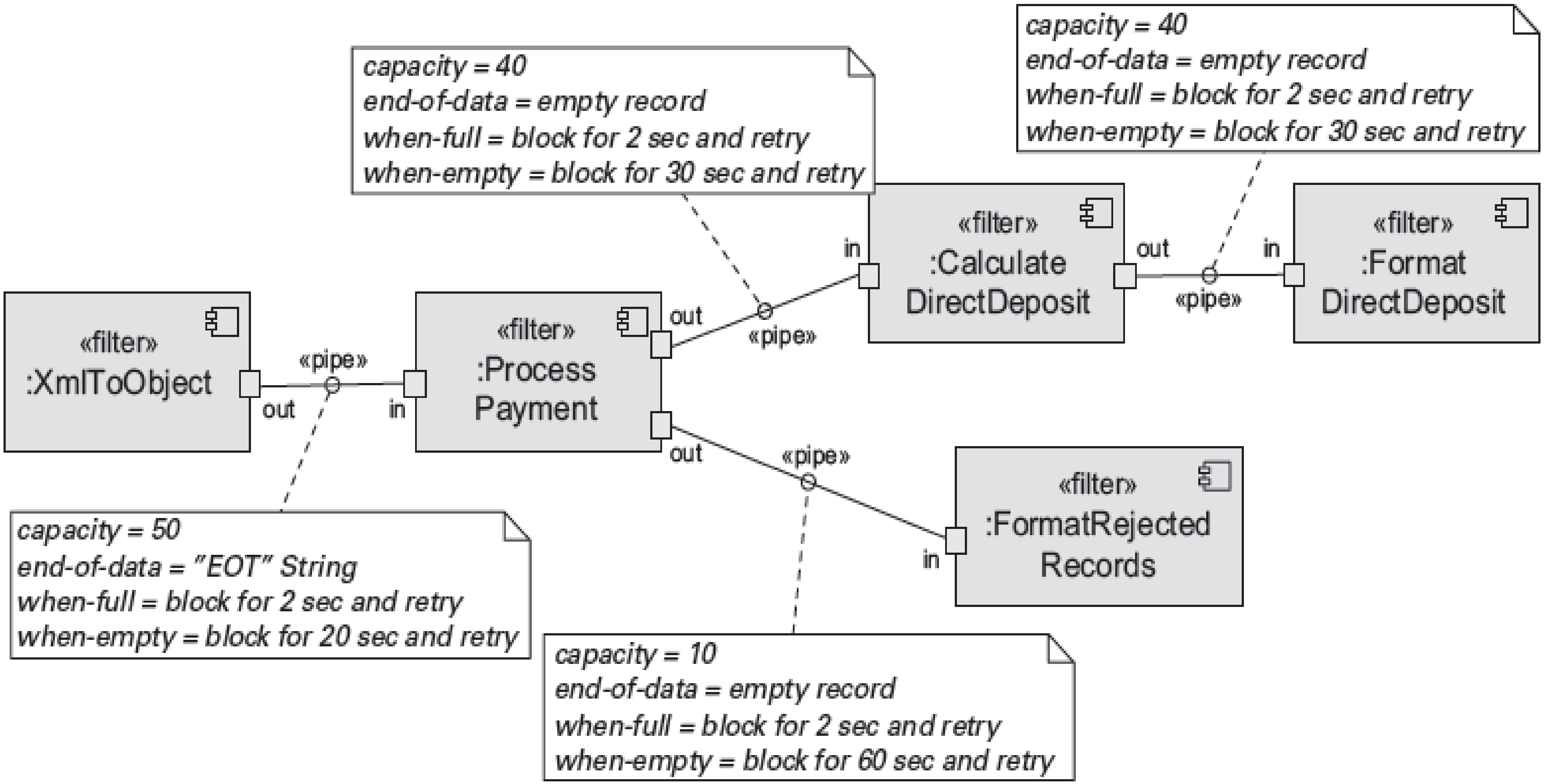
- Relations: The *notifies* relation connects instances of model, view, and controller, notifying elements of relevant state changes.
- Constraints:
  - There must be at least one instance each of model, view, and controller.
  - The model component should not interact directly with the controller.
- Weaknesses:
  - The complexity may not be worth it for simple user interfaces.
  - The model, view, and controller abstractions may not be good fits for some user interface toolkits.

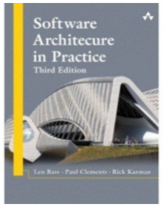


Pipe and Filter Pattern

- **Context:** Many systems are required to transform streams of discrete data items, from input to output. Many types of transformations occur repeatedly in practice, and so it is desirable to create these as independent, reusable parts.
- **Problem:** Such systems need to be divided into reusable, loosely coupled components with simple, generic interaction mechanisms. In this way they can be flexibly combined with each other. The components, being generic and loosely coupled, are easily reused. The components, being independent, can execute in parallel.
- **Solution:** The pattern of interaction in the pipe-and-filter pattern is characterized by successive transformations of streams of data. Data arrives at a filter's input port(s), is transformed, and then is passed via its output port(s) through a pipe to the next filter. A single filter can consume data from, or produce data to, one or more ports.

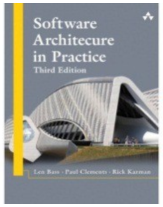
Pipe and Filter Example





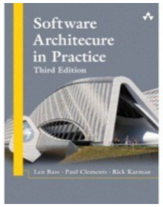
Pipe and Filter Solution

- Overview: Data is transformed from a system's external inputs to its external outputs through a series of transformations performed by its filters connected by pipes.
- Elements:
  - *Filter,* which is a component that transforms data read on its input port(s) to data written on its output port(s).
  - *Pipe,* which is a connector that conveys data from a filter's output port(s) to another filter's input port(s). A pipe has a single source for its input and a single target for its output. A pipe preserves the sequence of data items, and it does not alter the data passing through.
- Relations: The *attachment* relation associates the output of filters with the input of pipes and vice versa.
- Constraints:
  - Pipes connect filter output ports to filter input ports.
  - Connected filters must agree on the type of data being passed along the connecting pipe.

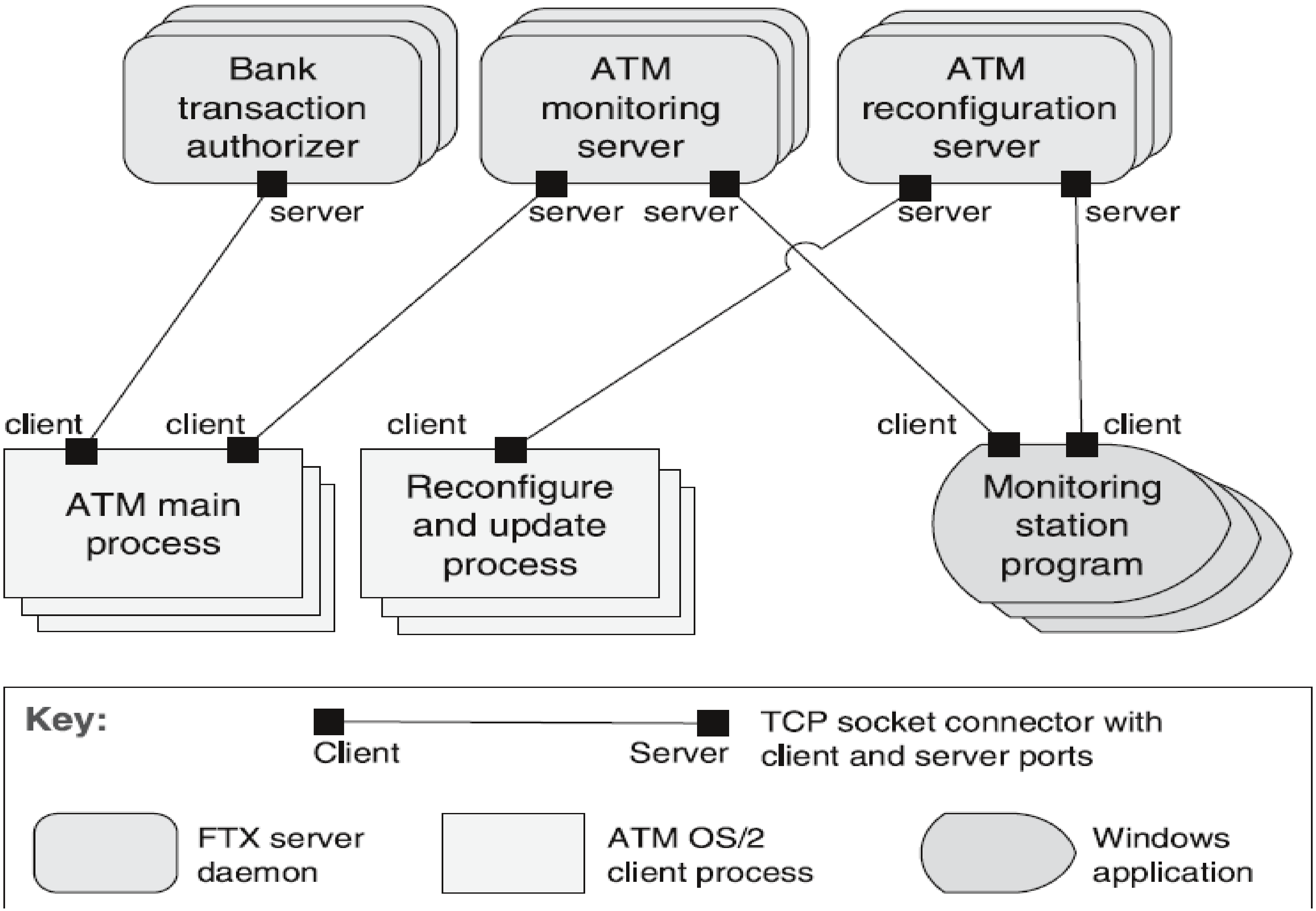


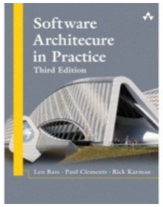
Client-Server Pattern

- **Context:** There are shared resources and services that large numbers of distributed clients wish to access, and for which we wish to control access or quality of service.
- **Problem:** By managing a set of shared resources and services, we can promote modifiability and reuse, by factoring out common services and having to modify these in a single location, or a small number of locations. We want to improve scalability and availability by centralizing the control of these resources and services, while distributing the resources themselves across multiple physical servers.
- **Solution:** Clients interact by requesting services of servers, which provide a set of services. Some components may act as both clients and servers. There may be one central server or multiple distributed ones.



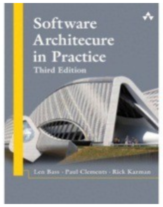
Client-Server Example





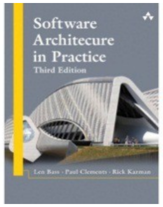
Client-Server Solution - 1

- Overview: Clients initiate interactions with servers, invoking services as needed from those servers and waiting for the results of those requests.
- Elements:
  - *Client,* a component that invokes services of a server component. Clients have ports that describe the services they require.
  - *Server:* a component that provides services to clients. Servers have ports that describe the services they provide.
- *Request/reply connector:* a data connector employing a request/reply protocol, used by a client to invoke services on a server. Important characteristics include whether the calls are local or remote, and whether data is encrypted.



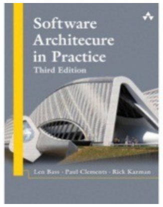
Client-Server Solution- 2

- Relations: The *attachment* relation associates clients with servers.
- Constraints:
  - Clients are connected to servers through request/reply connectors.
  - Server components can be clients to other servers.
- Weaknesses:
  - Server can be a performance bottleneck.
  - Server can be a single point of failure.
  - Decisions about where to locate functionality (in the client or in the server) are often complex and costly to change after a system has been built.

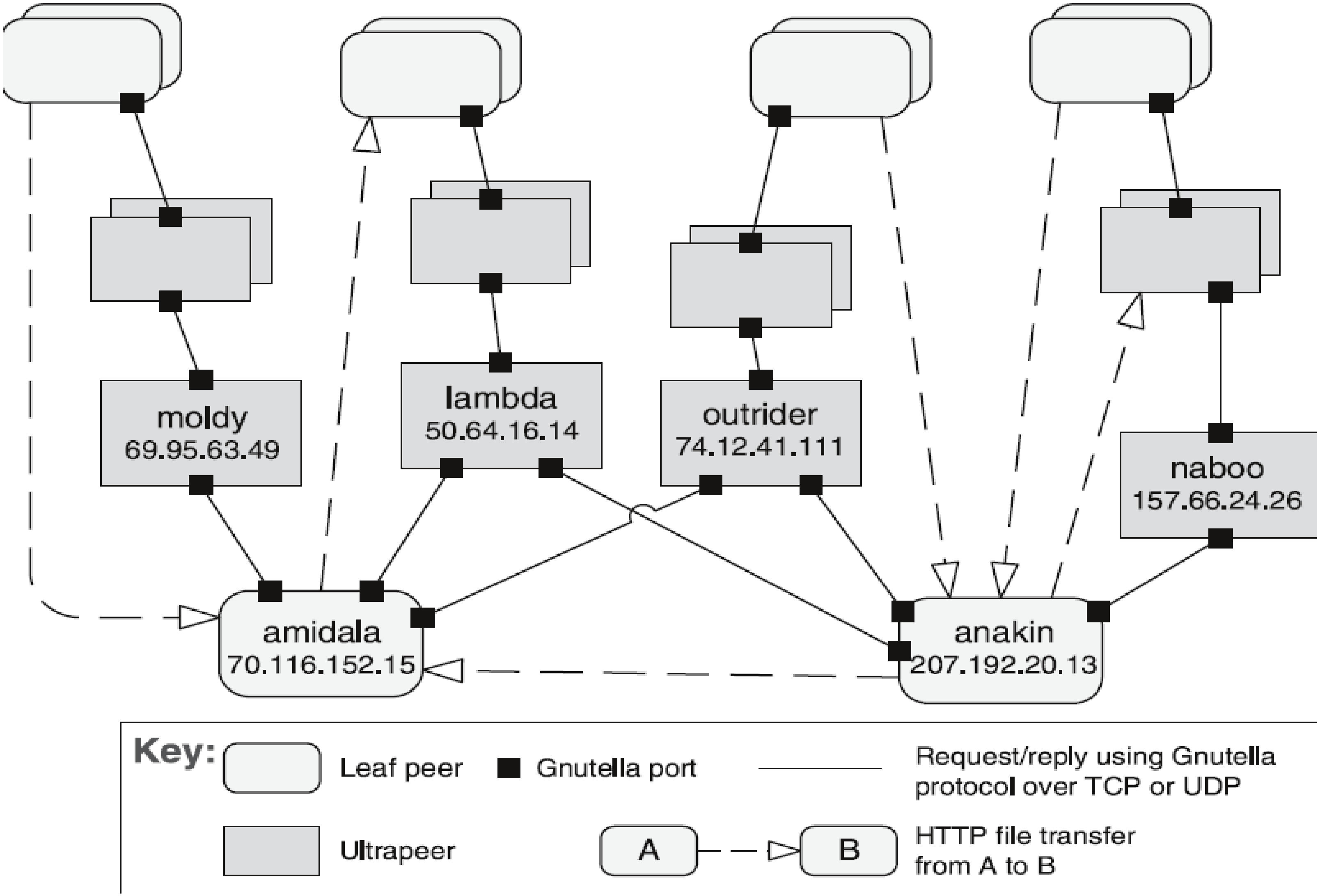


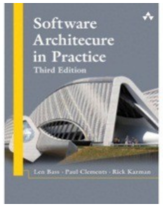
Peer-to-Peer Pattern

- **Context:** Distributed computational entities—each of which is considered equally important in terms of initiating an interaction and each of which provides its own resources—need to cooperate and collaborate to provide a service to a distributed community of users.
- **Problem:** How can a set of “equal” distributed computational entities be connected to each other via a common protocol so that they can organize and share their services with high availability and scalability?
- **Solution:** In the peer-to-peer (P2P) pattern, components directly interact as peers. All peers are “equal” and no peer or group of peers can be critical for the health of the system. Peer-to-peer communication is typically a request/reply interaction without the asymmetry found in the client-server pattern.



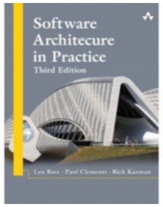
Peer-to-Peer Example





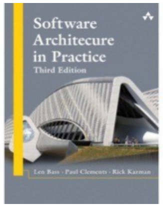
Peer-to-Peer Solution - 1

- **Overview**: Computation is achieved by cooperating peers that request service from and provide services to one another across a network.
- **Elements**:
  - *Peer,* which is an independent component running on a network node. Special peer components can provide routing, indexing, and peer search capability.
  - *Request/reply connector,* which is used to connect to the peer network, search for other peers, and invoke services from other peers. In some cases, the need for a reply is done away with.
- **Relations**: The relation associates peers with their connectors. Attachments may change at runtime.



Peer-to-Peer Solution - 2

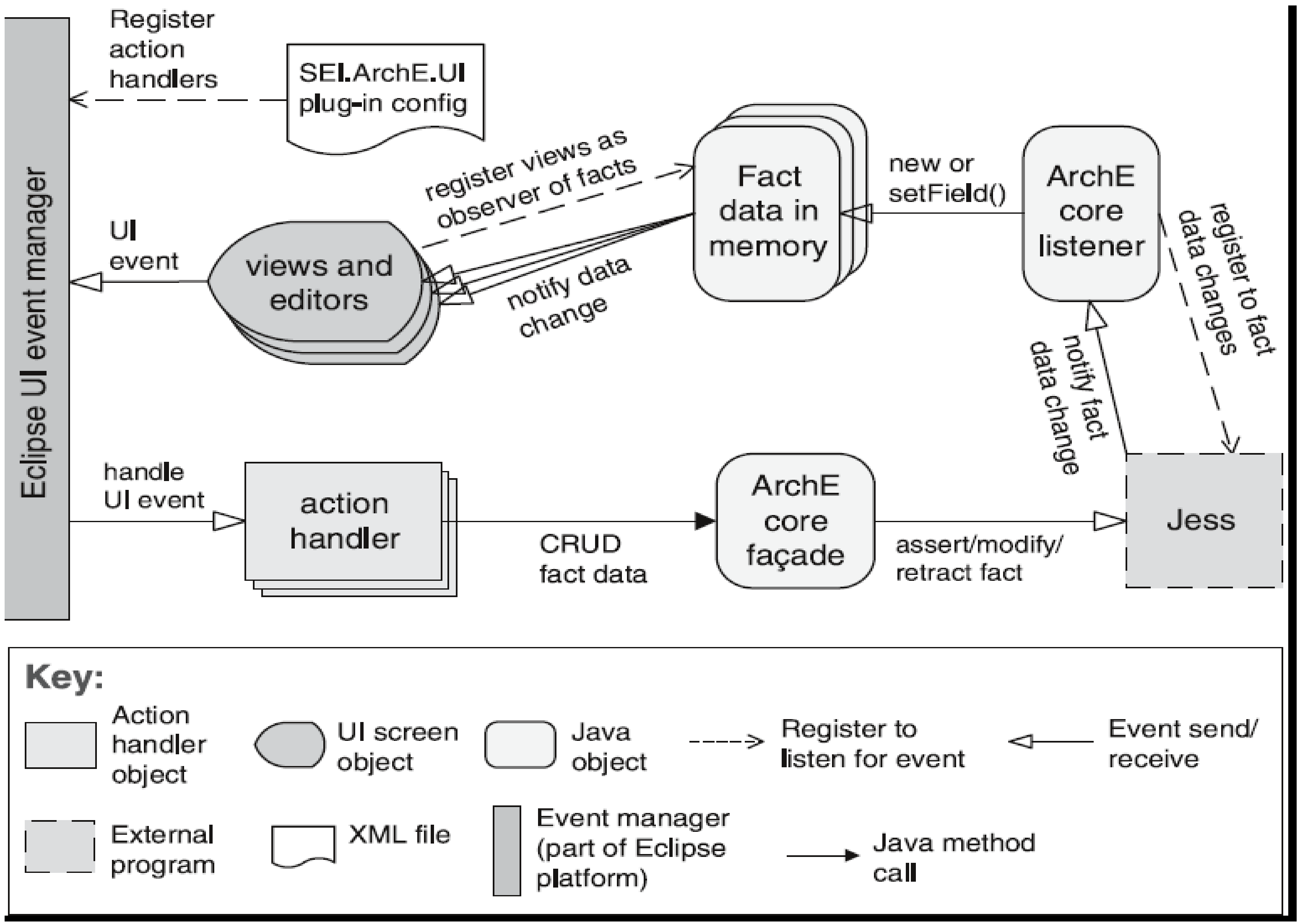
- Constraints: Restrictions may be placed on the following:
  - The number of allowable attachments to any given peer
  - The number of hops used for searching for a peer
  - Which peers know about which other peers
  - Some P2P networks are organized with star topologies, in which peers only connect to supernodes.
- Weaknesses:
  - Managing security, data consistency, data/service availability, backup, and recovery are all more complex.
  - Small peer-to-peer systems may not be able to consistently achieve quality goals such as performance and availability.

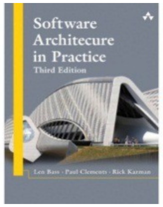


Publish-Subscribe Pattern

- **Context:** There are a number of independent producers and consumers of data that must interact. The precise number and nature of the data producers and consumers are not predetermined or fixed, nor is the data that they share.
- **Problem:** How can we create integration mechanisms that support the ability to transmit messages among the producers and consumers so they are unaware of each other's identity, or potentially even their existence?
- **Solution:** In the publish-subscribe pattern, components interact via announced messages, or events. Components may subscribe to a set of events. Publisher components place events on the bus by announcing them; the connector then delivers those events to the subscriber components that have registered an interest in those events.

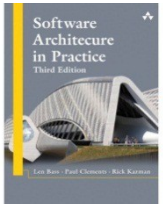
Publish-Subscribe Example





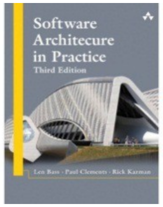
Publish-Subscribe Solution – 1

- Overview: Components publish and subscribe to events. When an event is announced by a component, the connector infrastructure dispatches the event to all registered subscribers.
- Elements:
  - *Any C&C component* with at least one publish or subscribe port.
  - *The publish-subscribe connector*, which will have *announce* and *listen* roles for components that wish to publish and subscribe to events.
- Relations: The *attachment* relation associates components with the publish-subscribe connector by prescribing which components announce events and which components are registered to receive events.



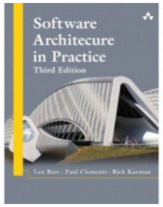
Publish-Subscribe Solution - 2

- Constraints: All components are connected to an event distributor that may be viewed as either a bus—connector—or a component. Publish ports are attached to announce roles and subscribe ports are attached to listen roles.
- Weaknesses:
  - Typically increases latency and has a negative effect on scalability and predictability of message delivery time.
  - Less control over ordering of messages, and delivery of messages is not guaranteed.

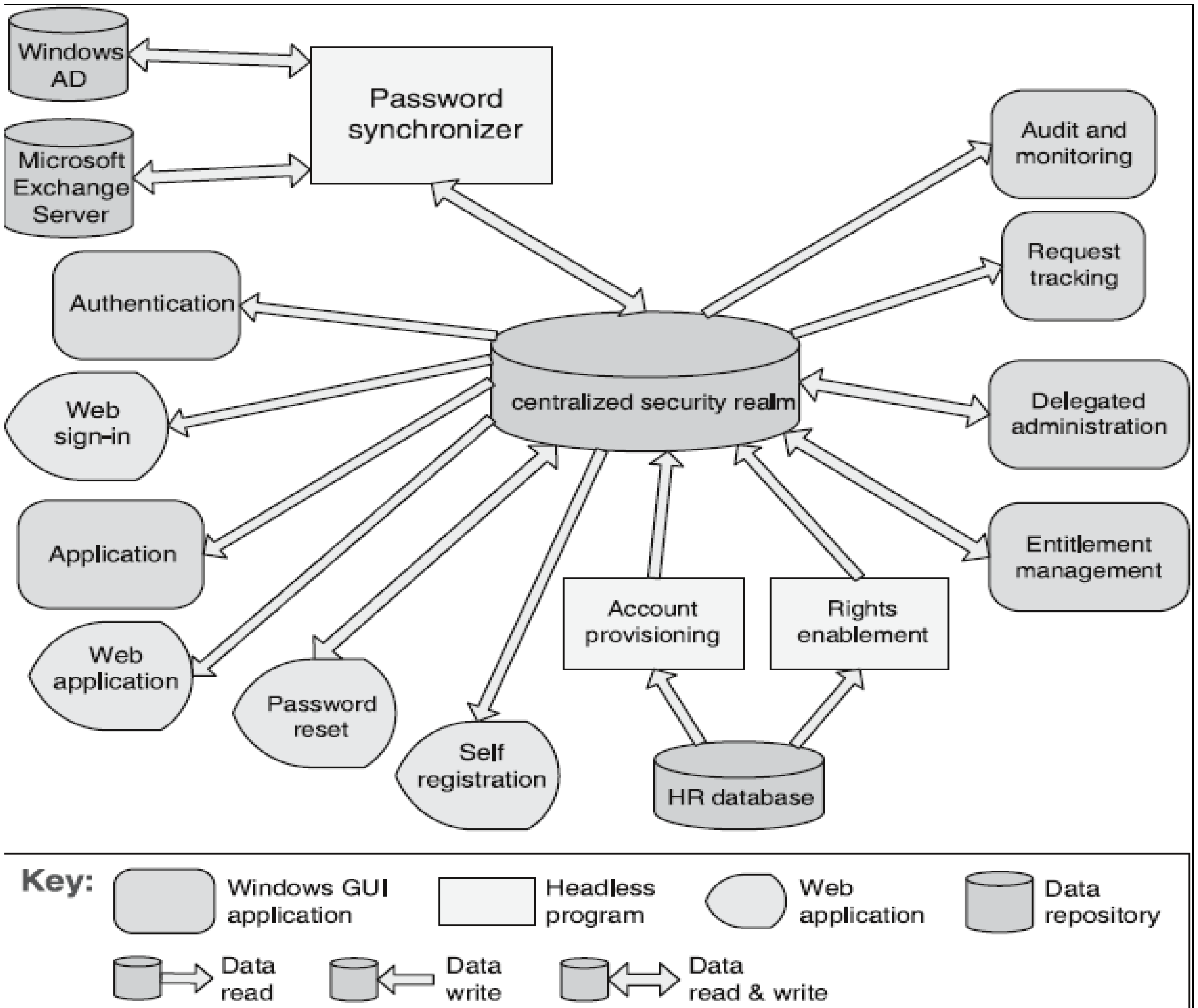


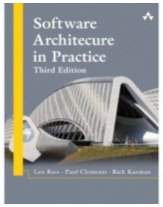
Shared-Data Pattern

- **Context:** Various computational components need to share and manipulate large amounts of data. This data does not belong solely to any one of those components.
- **Problem:** How can systems store and manipulate persistent data that is accessed by multiple independent components?
- **Solution:** In the shared-data pattern, interaction is dominated by the exchange of persistent data between multiple *data accessors* and at least one *shared-data store*. Exchange may be initiated by the accessors or the data store. The connector type is *data reading and writing*.



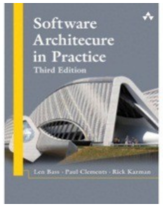
Shared Data Example





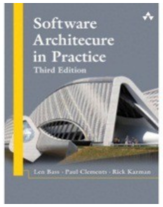
Shared Data Solution - 1

- Overview: Communication between data accessors is mediated by a shared data store. Control may be initiated by the data accessors or the data store. Data is made persistent by the data store.
- Elements:
  - *Shared-data store*. Concerns include types of data stored, data performance-oriented properties, data distribution, and number of accessors permitted.
  - *Data accessor component*.
  - *Data reading and writing connector*.



Shared Data Solution - 2

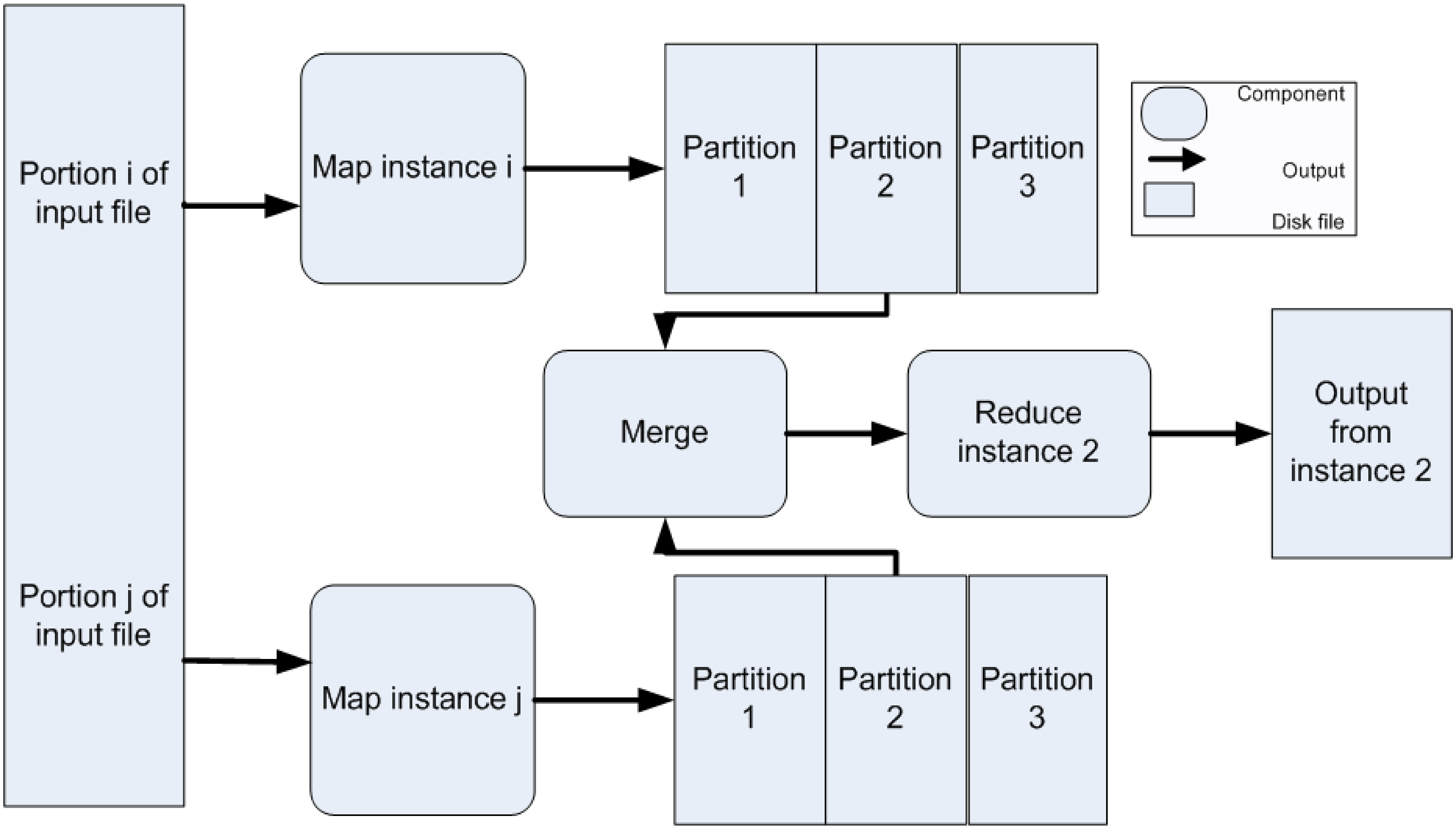
- Relations: *Attachment* relation determines which data accessors are connected to which data stores.
- Constraints: Data accessors interact only with the data store(s).
- Weaknesses:
  - The shared-data store may be a performance bottleneck.
  - The shared-data store may be a single point of failure.
  - Producers and consumers of data may be tightly coupled.

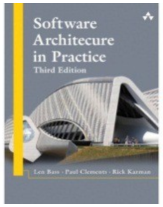


Map-Reduce Pattern

- **Context:** Businesses have a pressing need to quickly analyze enormous volumes of data they generate or access, at petabyte scale.
- **Problem:** For many applications with ultra-large data sets, sorting the data and then analyzing the grouped data is sufficient. The problem the map-reduce pattern solves is to efficiently perform a distributed and parallel sort of a large data set and provide a simple means for the programmer to specify the analysis to be done.
- **Solution:** The map-reduce pattern requires three parts:
  - A specialized infrastructure takes care of allocating software to the hardware nodes in a massively parallel computing environment and handles sorting the data as needed.
  - A programmer specified component called the map which filters the data to retrieve those items to be combined.
  - A programmer specified component called reduce which combines the results of the map

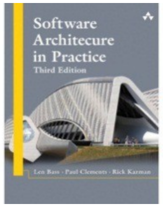
Map-Reduce Example





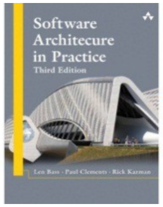
Map-Reduce Solution - 1

- Overview: The map-reduce pattern provides a framework for analyzing a large distributed set of data that will execute in parallel, on a set of processors. This parallelization allows for low latency and high availability. The map performs the extract and transform portions of the analysis and the reduce performs the loading of the results.
- Elements:
  - Map is a function with multiple instances deployed across multiple processors that performs the extract and transformation portions of the analysis.
  - Reduce is a function that may be deployed as a single instance or as multiple instances across processors to perform the load portion of extract-transform-load.
  - The infrastructure is the framework responsible for deploying map and reduce instances, shepherding the data between them, and detecting and recovering from failure.



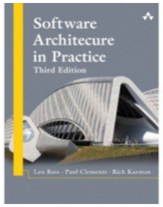
Map-Reduce Solution - 2

- Relations:
  - Deploy on is the relation between an instance of a map or reduce function and the processor onto which it is installed.
  - Instantiate, monitor, and control is the relation between the infrastructure and the instances of map and reduce.
- Constraints:
  - The data to be analyzed must exist as a set of files.
  - Map functions are stateless and do not communicate with each other.
  - The only communication between map reduce instances is the data emitted from the map instances as <key, value> pairs.
- Weaknesses:
  - If you do not have large data sets, the overhead of map-reduce is not justified.
  - If you cannot divide your data set into similar sized subsets, the advantages of parallelism are lost.
  - Operations that require multiple reduces are complex to orchestrate.

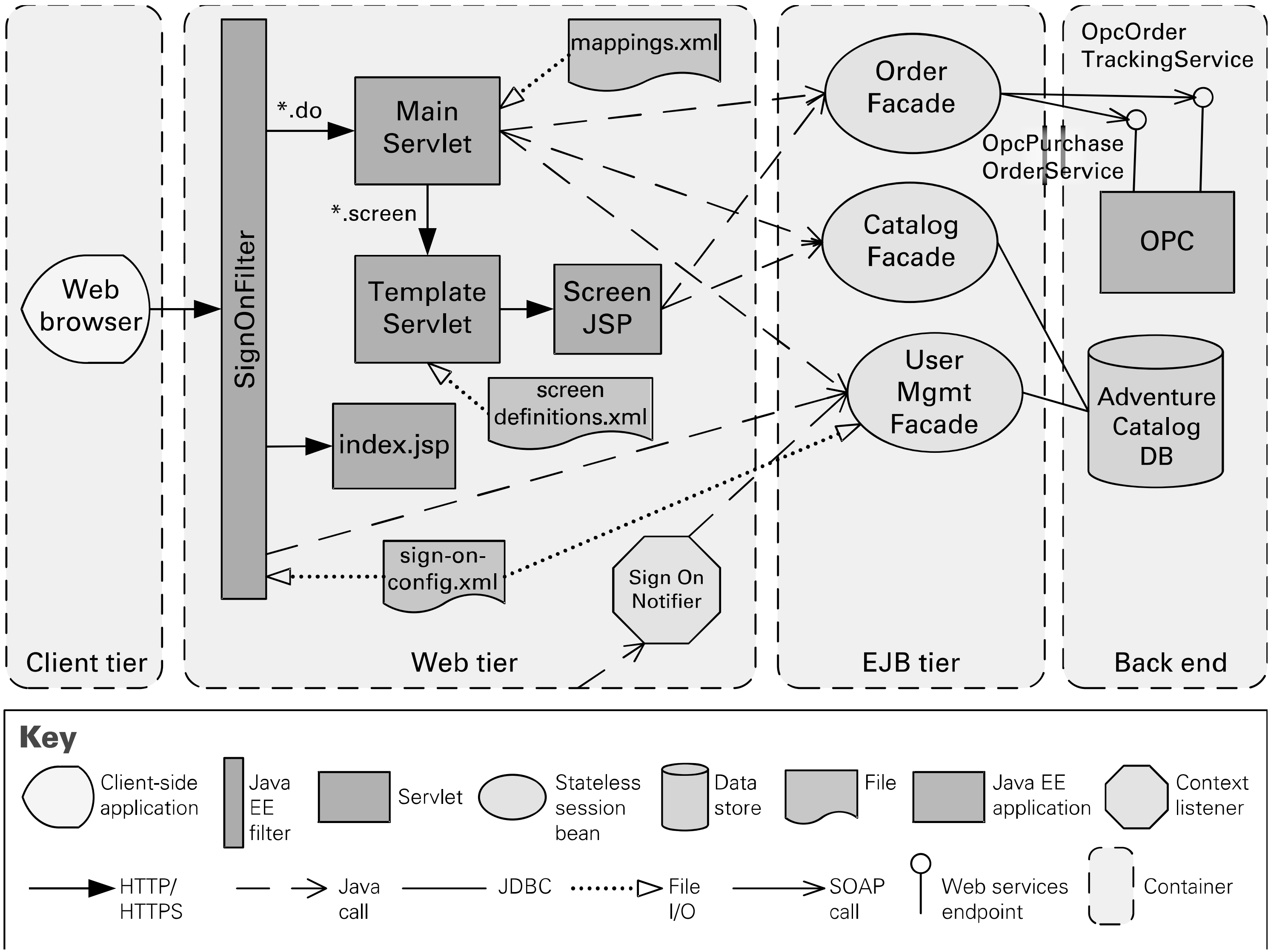


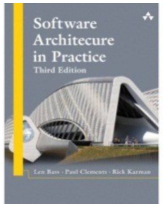
Multi-Tier Pattern

- **Context**: In a distributed deployment, there is often a need to distribute a system's infrastructure into distinct subsets.
- **Problem**: How can we split the system into a number of computationally independent execution structures—groups of software and hardware—connected by some communications media?
- **Solution**: The execution structures of many systems are organized as a set of logical groupings of components. Each grouping is termed a tier.



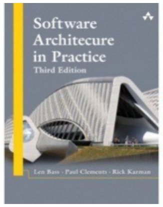
Multi-Tier Example





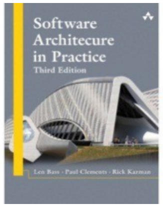
Multi-Tier Solution

- Overview: The execution structures of many systems are organized as a set of logical groupings of components. Each grouping is termed a *tier*.
- Elements:
  - *Tier,* which is a logical grouping of software components.
- Relations:
  - *Is part of,* to group components into tiers.
  - *Communicates with,* to show how tiers and the components they contain interact with each other.
  - *Allocated to,* in the case that tiers map to computing platforms.
- Constraints: A software component belongs to exactly one tier.
- Weaknesses: Substantial up-front cost and complexity.



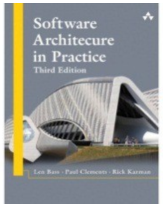
Relationships Between Tactics and Patterns

- Patterns are built from tactics; if a pattern is a molecule, a tactic is an atom.
- MVC, for example utilizes the tactics:
  - Increase semantic coherence
  - Encapsulation
  - Use an intermediary
  - Use run time binding



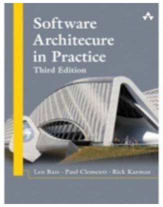
Tactics Augment Patterns

- Patterns solve a specific problem but are neutral or have weaknesses with respect to other qualities.
- Consider the broker pattern
  - May have performance bottlenecks
  - May have a single point of failure
- Using tactics such as
  - Increase resources will help performance
  - Maintain multiple copies will help availability



Tactics and Interactions

- Each tactic has pluses (its reason for being) and minuses – side effects.
- Use of tactics can help alleviate the minuses.
- But nothing is free…

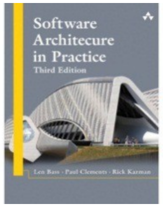


Tactics and Interactions - 2

A common tactic for detecting faults is Ping/Echo.

Common side-effects of Ping/Echo are:

- security: how to prevent a ping flood attack?
- performance: how to ensure that the performance overhead of ping/echo is small?
- modifiability: how to add ping/echo to the existing architecture?

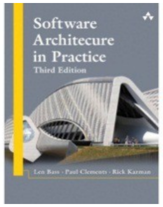


Tactics and Interactions - 4

A tactic to address the performance side-effect is “Increase Available Resources”.

Common side effects of Increase Available Resources are:

- cost: increased resources cost more
- performance: how to utilize the increase resources efficiently?

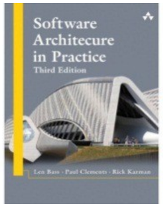


Tactics and Interactions - 6

A tactic to address the efficient use of resources side-effect is “Scheduling Policy”.

Common side effects of Scheduling Policy are:

- modifiability: how to add the scheduling policy to the existing architecture
- modifiability: how to change the scheduling policy in the future?

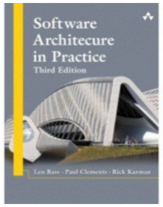


Tactics and Interactions - 8

A tactic to address the addition of the scheduler to the system is “Use an Intermediary”.

Common side effects of Use an Intermediary are:

- modifiability: how to ensure that all communication passes through the intermediary?



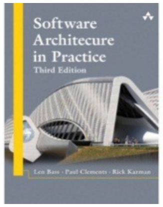
Tactics and Interactions – 10.

A tactic to address the concern that all communication passes through the intermediary is “Restrict Communication Paths”.

Common side effects of Restrict Communication Paths are:

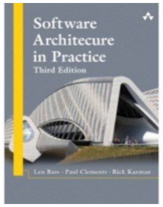
- performance: how to ensure that the performance overhead of the intermediary are not excessive?

Note: this design problem has now become recursive!



How Does This Process End?

- Each use of tactic introduces new concerns.
- Each new concern causes new tactics to be added.
- Are we in an infinite progression?
- No. Eventually the side-effects of each tactic become small enough to ignore.



Summary

- An architectural pattern
  - is a package of design decisions that is found repeatedly in practice,
  - has known properties that permit reuse, and
  - describes a *class* of architectures.
- Tactics are simpler than patterns
- Patterns are underspecified with respect to real systems so they have to be augmented with tactics.
  - Augmentation ends when requirements for a specific system are satisfied.