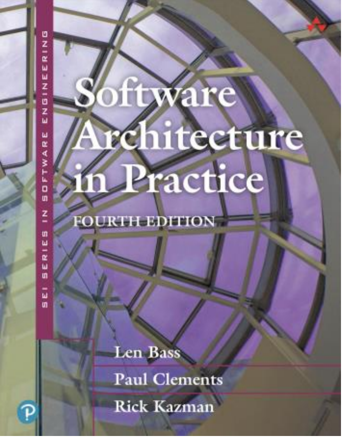
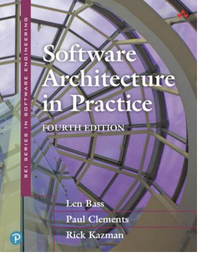

SEI SERIES IN SOFTWARE ENGINEERING
Software Architecture in Practice
FOURTH EDITION
Len Bass
Paul Clements
Rick Kazman

Chapter 22: Documenting an Architecture

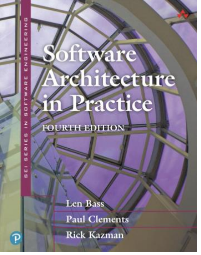
Documentation is a love letter that you write to your future self.

—Damian Conway


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

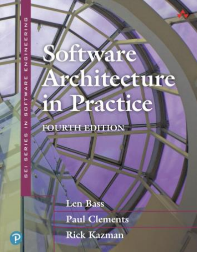
Chapter Outline

- Uses and Audiences for Architecture Documentation
- Notations
- Views
- Combining Views
- Documenting Behavior
- Beyond Views
- Documenting the Rationale
- Architecture Stakeholders
- Practical Considerations
- Summary



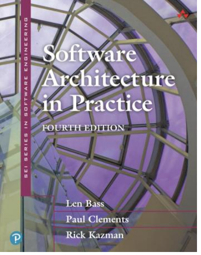
Architecture Documentation

- An architecture has to be communicated to let its stakeholders use it properly to do their jobs.
- If you go to the trouble of creating a strong architecture, you *must* go to the trouble of describing it.
- Documentation speaks for the architect today, when the architect should be doing other things besides answering questions, and tomorrow, when that person has left the project.



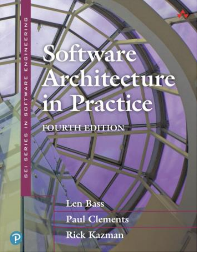
Architecture Documentation

- Even the best architecture will be useless if the people who need it
  - do not know what it is;
  - cannot understand it well enough to use, build, or modify it;
  - misunderstand it and apply it incorrectly.
- All of the effort, analysis, hard work, and insightful design on the part of the architecture team will have been wasted.



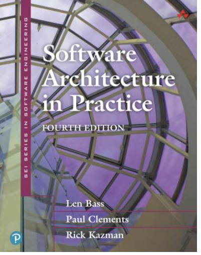
Uses and Audience for Architecture Documentation

- Architecture documentation must
  - be sufficiently transparent and accessible to be quickly understood by new employees
  - be sufficiently concrete to serve as a blueprint for construction
  - have enough information to serve as a basis for analysis.
- Architecture documentation is both prescriptive and descriptive.
  - For some audiences, it prescribes what *should* be true, placing constraints on decisions yet to be made.
  - For other audiences, it describes what *is* true, recounting decisions already made about a system’s design.
- Understanding stakeholder uses of architecture documentation is essential
- Those uses determine the information to capture.



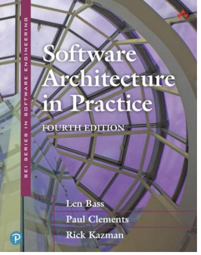
Uses and Audiences for Architecture Documentation

- Education
  - Introducing people to the system
    - New members of the team
    - External analysts or evaluators
    - New architect
- Primary vehicle for communication among stakeholders
  - Especially architect to developers
  - Especially architect to future architect!
- Basis for system analysis and construction
  - Architecture tells implementers what to implement.
  - Each module has interfaces that must be provided and uses interfaces from other modules.
- Basis for forensics when an incident occurs.
  - When an incident occurs, someone is responsible for tracking down the underlying causes.
  - Information about the flow of control immediately prior to the incident will provide the "as executed" architecture.



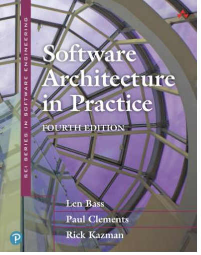
Notations

- *Informal notations*
  - Views are depicted (often graphically) using general-purpose diagramming and editing tools
  - The semantics of the description are characterized in natural language
  - They cannot be formally analyzed
- *Semiformal notations*
  - Standardized notation that prescribes graphical elements and rules of construction
  - Lacks a complete semantic treatment of the meaning of those elements
  - Rudimentary analysis can be applied
  - UML is a semiformal notation in this sense.
- *Formal notations*
  - Views are described in a notation that has a precise (usually mathematically based) semantics.
  - Formal analysis of both syntax and semantics is possible.
  - Architecture description languages (ADLs)
  - Support automation through associated tools.


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

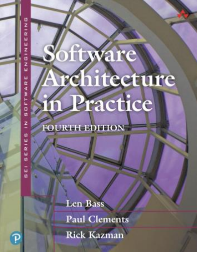
Choosing a Notation

- Tradeoffs
  - Typically, more formal notations take more time and effort to create and understand, but offer reduced ambiguity and more opportunities for analysis.
  - Conversely, more informal notations are easier to create, but they provide fewer guarantees.
- Different notations are better (or worse) for expressing different kinds of information.
  - UML class diagram will not help you reason about schedulability, nor will a sequence chart tell you very much about the system’s likelihood of being delivered on time.
  - Choose your notations and representation languages knowing the important issues you need to capture and reason about.



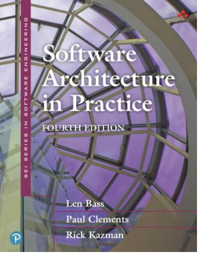
Views

- Views let us divide a software architecture into a number of (we hope) interesting and manageable representations of the system.
- Principle of architecture documentation:
  - *Documenting an architecture is a matter of documenting the relevant views and then adding documentation that applies to more than one view.*



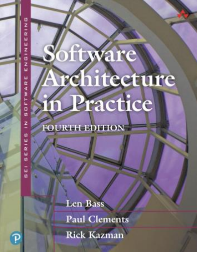
Which Views? The Ones You Need!

- Different views support different goals and uses.
- We do not advocate a particular view or collection of views.
- The views you should document depend on the uses you expect to make of the documentation.
- Each view has a cost and a benefit; you should ensure that the benefits of maintaining a view outweigh its costs.



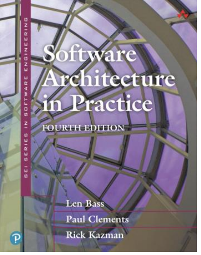
Overview of Module Views

- Elements
  - Modules, which are implementation units of software that provide a coherent set of responsibilities.
- Relations
  - *Is part of*, which defines a part/whole relationship between the submodule—the part—and the aggregate module—the whole.
  - *Depends on*, which defines a dependency relationship between two modules. Specific module views elaborate what dependency is meant.
  - *Is a*, which defines a generalization/specialization relationship between a more specific module—the child—and a more general module—the parent.



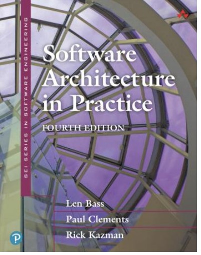
Overview of Module Views

- Constraints
  - Different module views may impose specific topological constraints, such as limitations on the visibility between modules.
- Usage
  - Blueprint for construction of the code
  - Change-impact analysis
  - Planning incremental development
  - Requirements traceability analysis
  - Communicating the functionality of a system and the structure of its code base
  - Supporting the definition of work assignments, implementation schedules, and budget information
  - Showing the structure of information that the system needs to manage



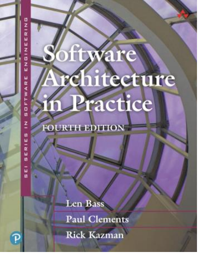
Module Views

- It is unlikely that the documentation of any software architecture can be complete without at least one module view.


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

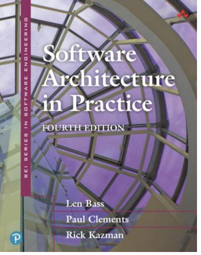
Overview of C&C Views

- Elements
  - *Components.* Principal processing units and data stores. A component has a set of *ports* through which it interacts with other components (via connectors).
  - *Connectors*. Pathways of interaction between components. Connectors have a set of roles (interfaces) that indicate how components may use a connector in interactions.
- Relations
  - *Attachments*. Component ports are associated with connector roles to yield a graph of components and connectors*.*
  - *Interface delegation.* In some situations component ports are associated with one or more ports in an "internal" subarchitecture. The case is similar for the roles of a connector



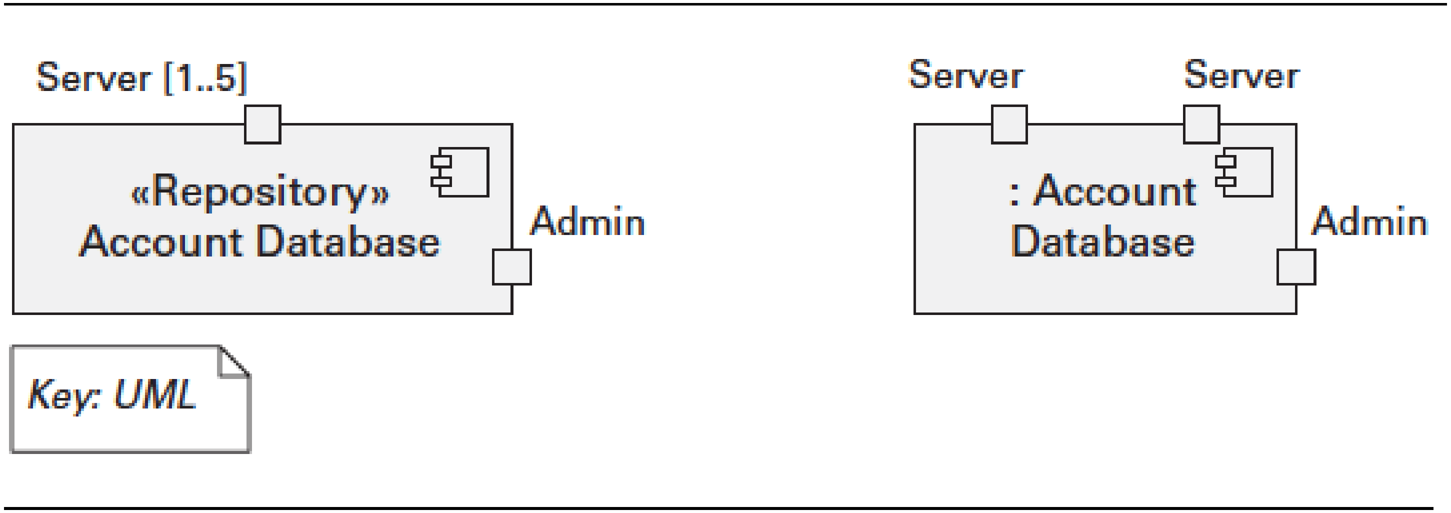
Overview of C&C Views

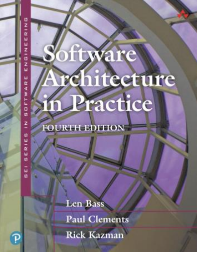
- Constraints
  - Components can only be attached to connectors, not directly to other components.
  - Connectors can only be attached to components, not directly to other connectors.
  - Attachments can only be made between compatible ports and roles.
  - Interface delegation can only be defined between two compatible ports (or two compatible roles).
  - Connectors cannot appear in isolation; a connector must be attached to a component.
- Usage
  - Show how the system works.
  - Guide development by specifying structure and behavior of runtime elements.
  - Help reason about runtime system quality attributes, such as performance and availability.



Notations for C&C Views

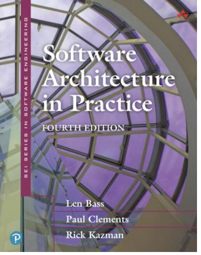
- UML components are good match for C&C components.





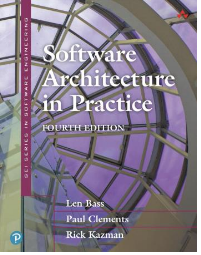
Overview of Allocation Views

- Elements
  - *Software element* and *environmental element.*
  - A software element has properties that are *required* of the environment.
  - An environmental element has properties that are *provided* to the software.
- Relations
  - *Allocated to*. A software element is mapped (allocated to) an environmental element. Properties are dependent on the particular view.



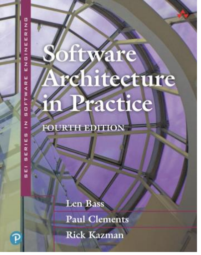
Overview of Allocation Views

- Constraints
  - Varies by view
- Usage
  - Reasoning about performance, availability, security, and safety.
  - Reasoning about distributed development and allocation of work to teams.
  - Reasoning about concurrent access to software versions.
  - Reasoning about the form and mechanisms of system installation.



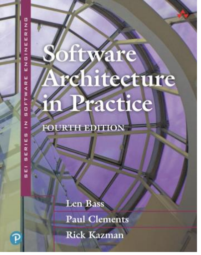
Quality Views

- A *quality view* can be tailored for specific stakeholders or to address specific concerns.
- A quality views is formed by extracting the relevant pieces of structural views and packaging them together.



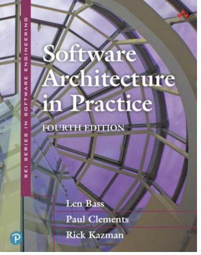
Quality Views: Examples

- *Security view*
  - Show the components that have some security role or responsibility, how those components communicate, any data repositories for security information, and repositories that are of security interest.
  - The view’s context information would show other security measures (such as physical security) in the system’s environment.
  - The behavior part of a security view
    - Show how the operation of security protocols and where and how humans interact with the security elements.
    - Capture how the system would respond to specific threats and vulnerabilities.
- *Communications view*
  - Especially helpful for systems that are globally dispersed and heterogeneous.
  - Show all of the component-to-component channels, the various network channels, quality-of-service parameter values, and areas of concurrency.
  - Used to analyze certain kinds of performance and reliability (such as deadlock or race condition detection).
  - The behavior part of this view could show (for example) how network bandwidth is dynamically allocated.



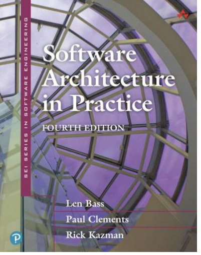
Quality Views: Examples

- *Exception* or *error-handling view*
  - Could help illuminate and draw attention to error reporting and resolution mechanisms.
  - Show how components detect, report, and resolve faults or errors.
  - It would help identify the sources of errors and appropriate corrective actions for each.
- *Reliability* view
  - Models mechanisms such as replication and switchover.
  - Depicts timing issues and transaction integrity.
- *Performance* view
  - Shows those aspects of the architecture useful for inferring the system's performance.
  - Show network traffic models, maximum latencies for operations, and so forth.



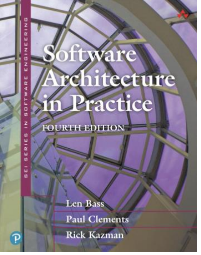
Choosing the Views

- At a minimum, expect to have at least one module view, at least one C&C view, and for larger systems, at least one allocation view in your architecture document.



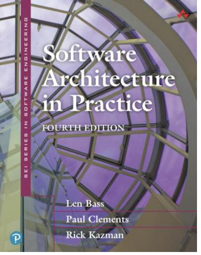
Combining Views

- Sometimes it is convenient to show a *combined view* with elements and relations that come from two or more other views.
- Such views can be very useful as long as you do not try to overload them.



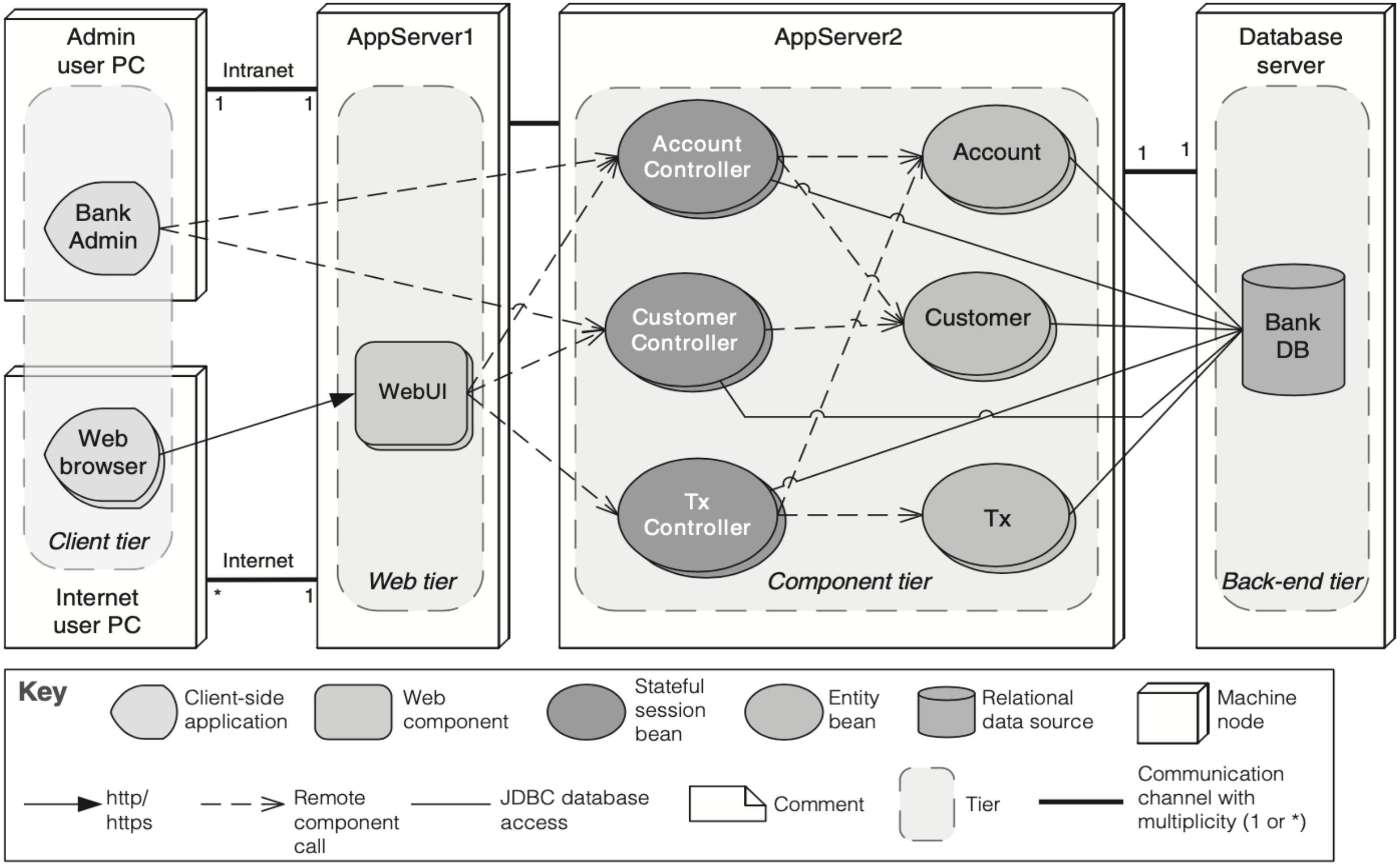
Combining Views

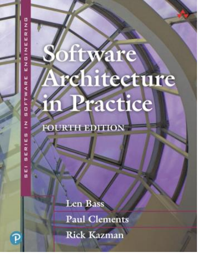
- The following combinations of views often occur quite naturally:
  - – *C&C views with each other*
  - – *Deployment view with any C&C view that shows processes*
  - – *Decomposition view and any work assignment, implementation, uses, or layered views*



Combining Views

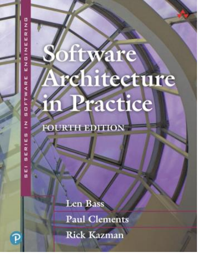
- A combined view (example)





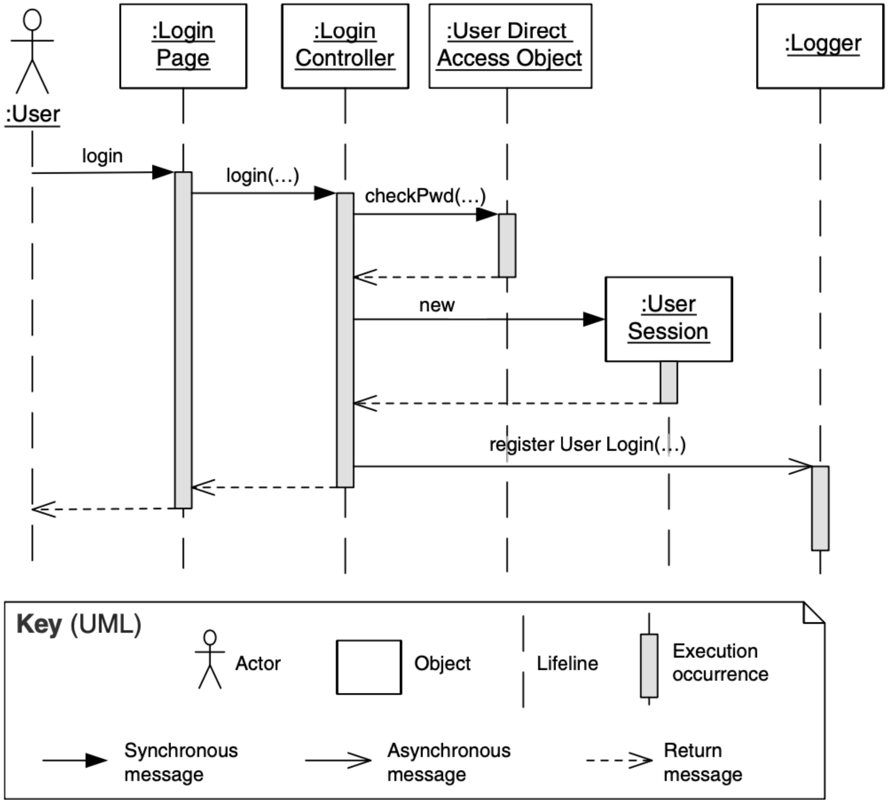
Documenting Behavior

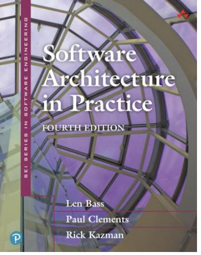
- An architecture may require behavior documentation describing how elements interact with each other.
- Two kinds of notations are available for documenting behavior: trace-oriented and comprehensive.



Documenting Behavior

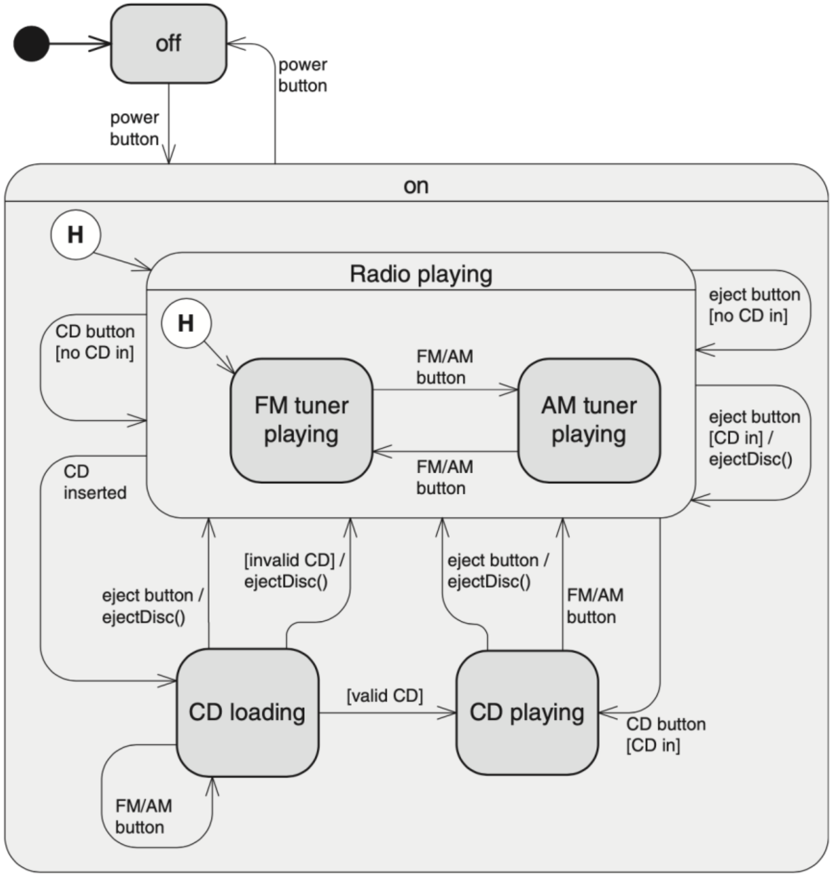
- *Traces* are sequences of activities or interactions that describe the system's response to a specific stimulus when the system is in a specific state.
- UML Examples:
  - – use case diagrams
  - – sequence diagrams
  - – communication diagrams
  - – activity diagrams.

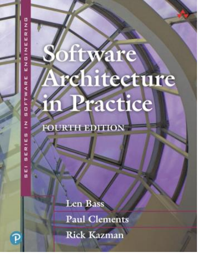




Documenting Behavior

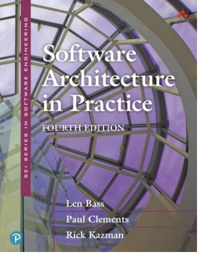
- *Comprehensive* notations show the complete behavior of structural elements. Given such documentation, it is possible to infer all possible paths through the system.
- UML Example:
  - state machine diagrams





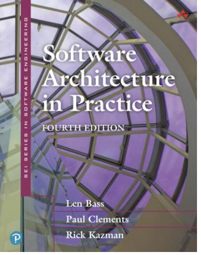
Building a Documentation Package

- A documentation package consists of
  - Views
  - Documentation beyond views


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

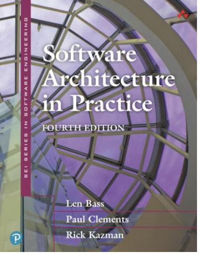
Beyond Views

- In addition to views and behavior, comprehensive information about an architecture will include the following items:
  - *Mapping between views*
  - *Documenting patterns*
  - *One or more context diagrams*
  - *Variability guide*
  - *Rationale*
  - *Glossary and acronym list*
  - *Document control information*



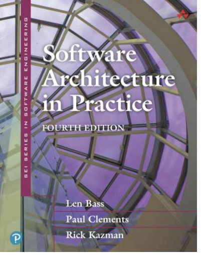
Documenting Rationale

- When designing, you make decisions to achieve your iteration goals. These decisions include:
  - Selecting a design concept from several alternatives
  - Creating structures by instantiating the selected design concept
  - Establishing relationships between elements and defining interfaces
  - Allocating resources (e.g., people, hardware, computation)
- When you study an architecture diagram, you only see the end product of a thought process.



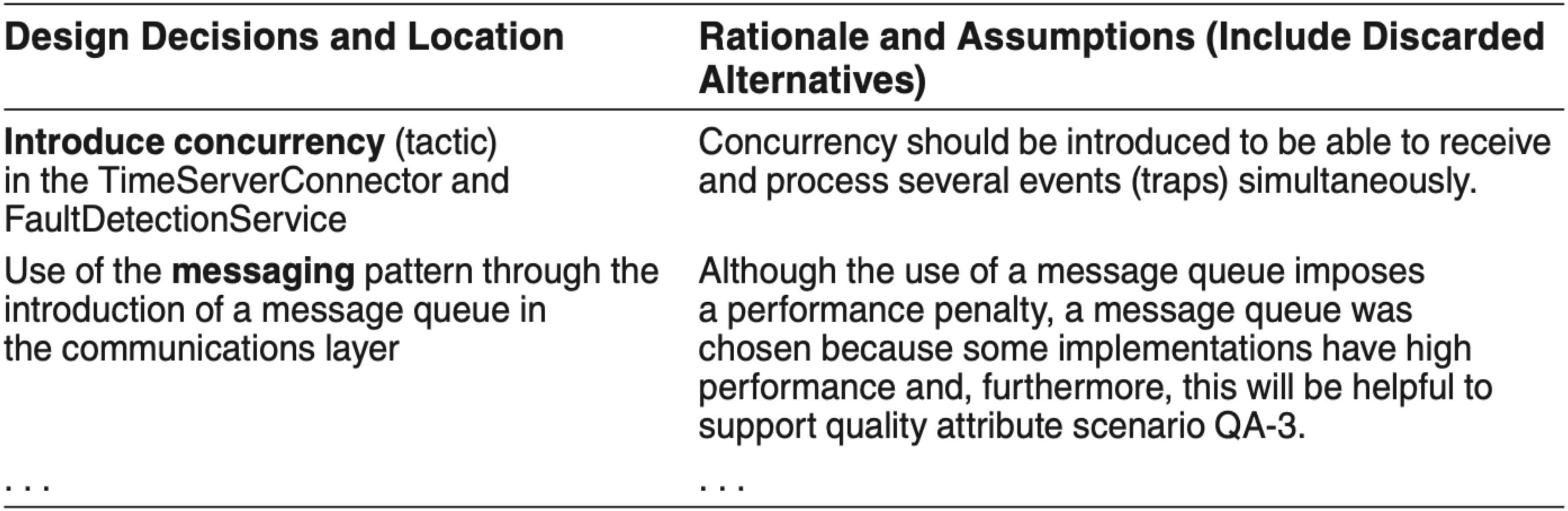
Documenting Rationale

- Recording design decisions *beyond* the chosen elements, relationships, and properties is fundamental to help understand how you arrived at the result—the design rationale.
- For example, you might record:
  - What evidence was produced to justify decisions?
  - Who did what?
  - Why were shortcuts taken?
  - Why were tradeoffs made?
  - What assumptions did you make?

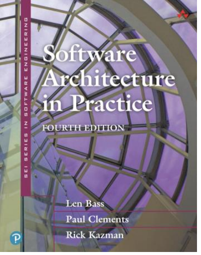


Documenting Rationale

- Example:

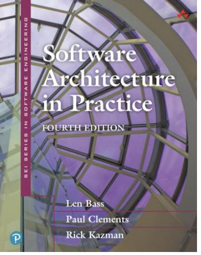


Design Decisions and Location	Rationale and Assumptions (Include Discarded Alternatives)
Introduce concurrency (tactic) in the TimeServerConnector and FaultDetectionService	Concurrency should be introduced to be able to receive and process several events (traps) simultaneously.
Use of the **messaging** pattern through the introduction of a message queue in the communications layer	Although the use of a message queue imposes a performance penalty, a message queue was chosen because some implementations have high performance and, furthermore, this will be helpful to support quality attribute scenario QA-3.
. . .	. . .



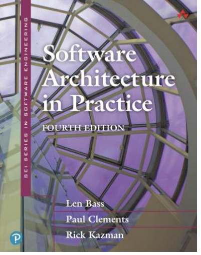
Architecture Stakeholders

- Key stakeholders of an architecture, and the views that care about include:
  - Project managers:
    - care about schedule, resource assignments, etc.
    - To create a schedule, they need information about the modules to be implemented and in what sequence, with information about their complexity, such as the list of responsibilities, and dependencies on other modules



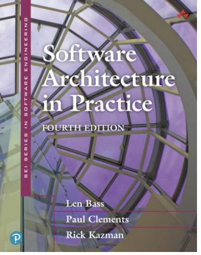
Architecture Stakeholders

– Members of the development team:
  - will want to know the general idea behind the system.
  - which elements they have been assigned for implementation.
  - the details of the assigned element, including the data model with which it must operate.
  - the elements with which the assigned part interfaces and what those interfaces are.
  - the code assets that the developer can utilize.
  - the constraints, such as quality attributes, legacy system interfaces, and budgets that must be met.



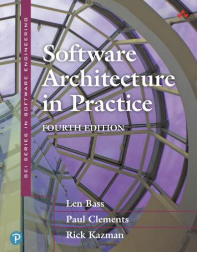
Architecture Stakeholders

- Testers and integrators:
  - a black-box tester will need to access the interface documentation for the element.
  - integrators and system testers need to see collections of interfaces, behavior specifications, and a uses view so they can work with incremental subsets.
- *Maintainers:*
  - will want to see a decomposition view that allows them to pinpoint the locations where a change will need to be carried out, and perhaps a uses view to help them build an impact analysis to fully scope out the effects of the change.
  - in addition, they will want to see the design rationale.



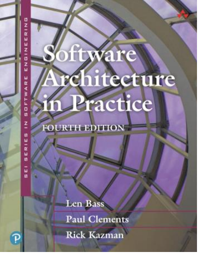
Architecture Stakeholders

- *Designers of other systems:*
  - these stakeholders will want to see interface documentation for elements their system will interact with, as found in module and C&C views
  - the data model for the system they will interact with
  - top-level context diagrams from various views showing the interactions
- *End users:*
  - can often gain useful insights into the system, what it does, and how they can use it effectively by examining the architecture.



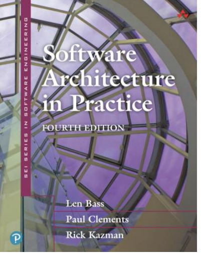
Architecture Stakeholders

- *Analysts:*
  - are interested in whether the design meets the system's quality objectives.
  - they require architectural information necessary to evaluate quality attributes.
- *Infrastructure support personnel:*
  - set up the infrastructure that supports the development, integration, staging, and production environments.
  - a variability guide is particularly useful to help set up the software configuration management environment.
- *Future architects:*
  - are the most avid readers of architecture documentation, with a vested interest in everything.



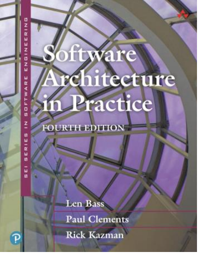
Practical Considerations

- Modeling tools:
  - Tools offer features aimed at large-scale use in industrial settings:
    - interfaces that support multiple users
    - version control
    - syntactic and semantic consistency checking of the models
    - traceability between models and requirements or models and tests
    - automatic generation of executable source code that implements the models.



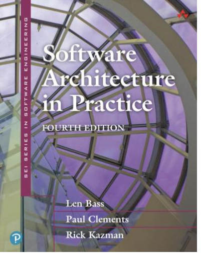
Practical Considerations

- Online documentation, hypertext, and wikis:
  - Documentation for a system can be structured as linked web pages.
  - Using tools such as wikis, it’s possible to create a *shared* document to which many stakeholders can contribute.



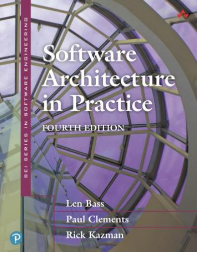
Practical Considerations

- Release strategy:
  - Your project’s development plan should specify the process for keeping the important documentation, including the architecture documentation, current.
  - Document artifacts should be subject to version control, as with any other important project artifact.
  - The architect should plan releases of the documentation to support major project milestones.



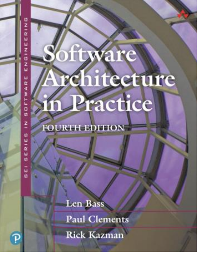
Practical Considerations

- Documenting architectures that change dynamically.
  - If your architecture changes rapidly (at runtime, or as a result of frequent releases):
    - *Document what is true about all versions of your system*.
    - *Document the ways the architecture is allowed to change*.
    - *Generate interface documentation automatically*.



Practical Considerations

- Traceability:
  - Traceability means linking design decisions to the requirements that led to them; those links should be captured in the documentation.
  - We seek to account for all ASRs in the architecture’s trace links.
  - Trace links may be represented informally—a table, for instance—or may be supported technologically in the project’s tool environment.



Summary

- Architectural documentation supports communication among various stakeholders, up the management chain, down to the developers, and across to peers.
- You must understand the uses to which the documentation is to be put and its audience.
- An architecture is a complicated artifact, best expressed by focusing on views.
- You must choose the views to document, the notations, and a set of views that is both minimal and adequate.
- There are other practical considerations, such as choosing a release strategy, choosing a dissemination tool, and creating documentation for architectures that change dynamically.