COIS 3040 Lecture 14

Designing an Architecture

1. Design Strategy
   - Decomposition
   - Designing to Architecturally Significant Requirements

     An architecturally significant requirement (ASR) is a requirement that will have a profound effect on the architecture, that is , the architecture might well be dramatically different in the absence of such a requirement.
   - Generate and Test

     This generate-and test approach views a particular design as a hypothesis: namely the design satisfies the requirements. Testing is the process of determining whether the design hypothesis is correct.
     - Creating the initial hypothesis
       - Existing systems
       - Frameworks
       - Patterns and tactics
       - Domain decomposition
       - Design checklists
     - Choose the tests
       - The analysis techniques
       - The design checklists
       - The architecturally significant requirements
     - Generating the next hypothesis
     - Terminating the process

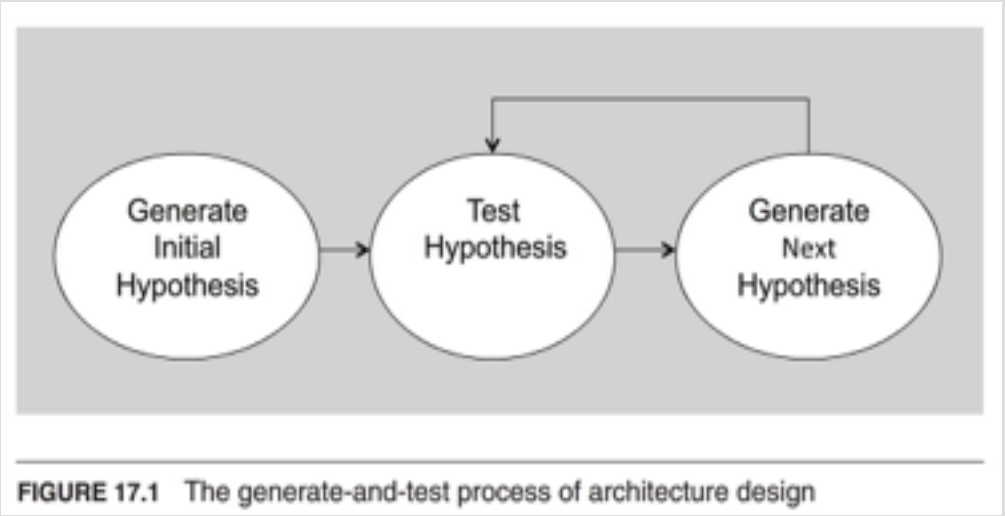


FIGURE 17.1 The generate-and-test process of architecture design

Designing an Architecture

2. The Attribute-Driven Design Method
   - Choose a part of the system to design
   - Marshal all the architecturally significant requirements for that part
   - Create and test a design for that part
   - Inputs to ADD: ASRs, contexts (system boundaries, external systems, underlying environments)
   - Output of ADD:

3. The Steps of ADD
   1) Step 1: Choose an element of the system to design
   1) Step 2: Identify ASRs for this element
   2) Step 3: Generate a design solution for the chosen element

   Step 4: Verify and refine requirements and create input for the next iteration

   Step 5: Repeat steps 1-4 until done

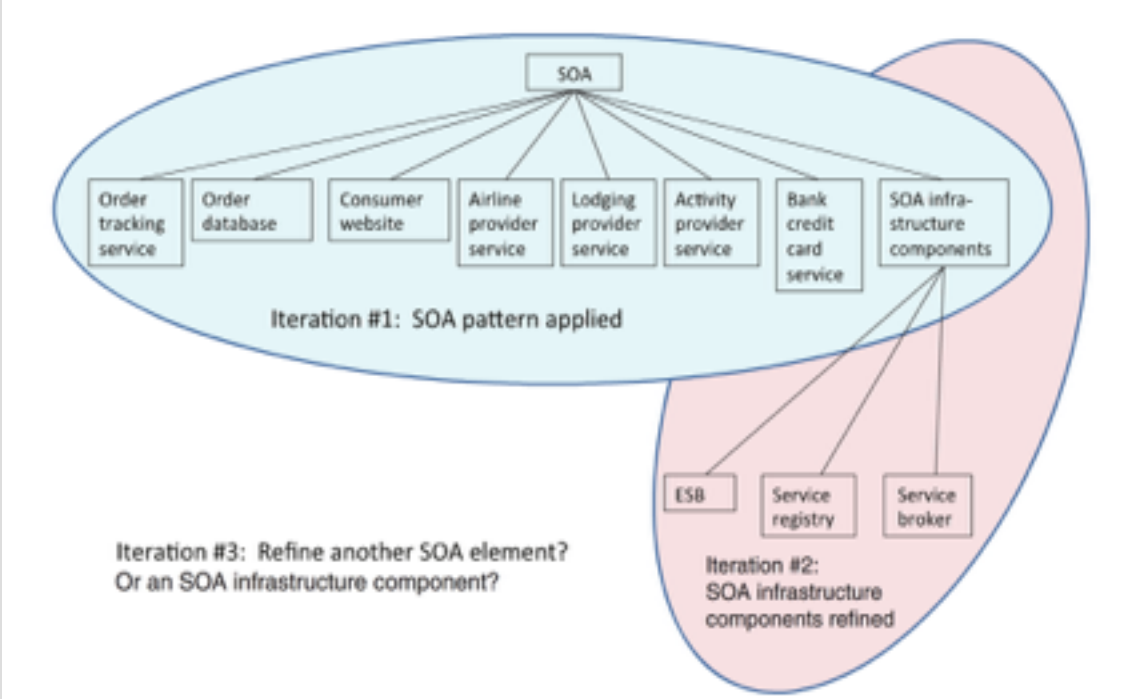


FIGURE 17.2 Iteration 1 applied the SOA pattern. Iteration 2 refined the infrastructure components. Where will iteration 3 take you?

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.

Three Uses 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.
- Documentation can serve as a receptacle for registering and communicating unresolved issues.
- Architecture documentation serves as the basis for architecture evaluation.

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.

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.

Overview of C&C Views

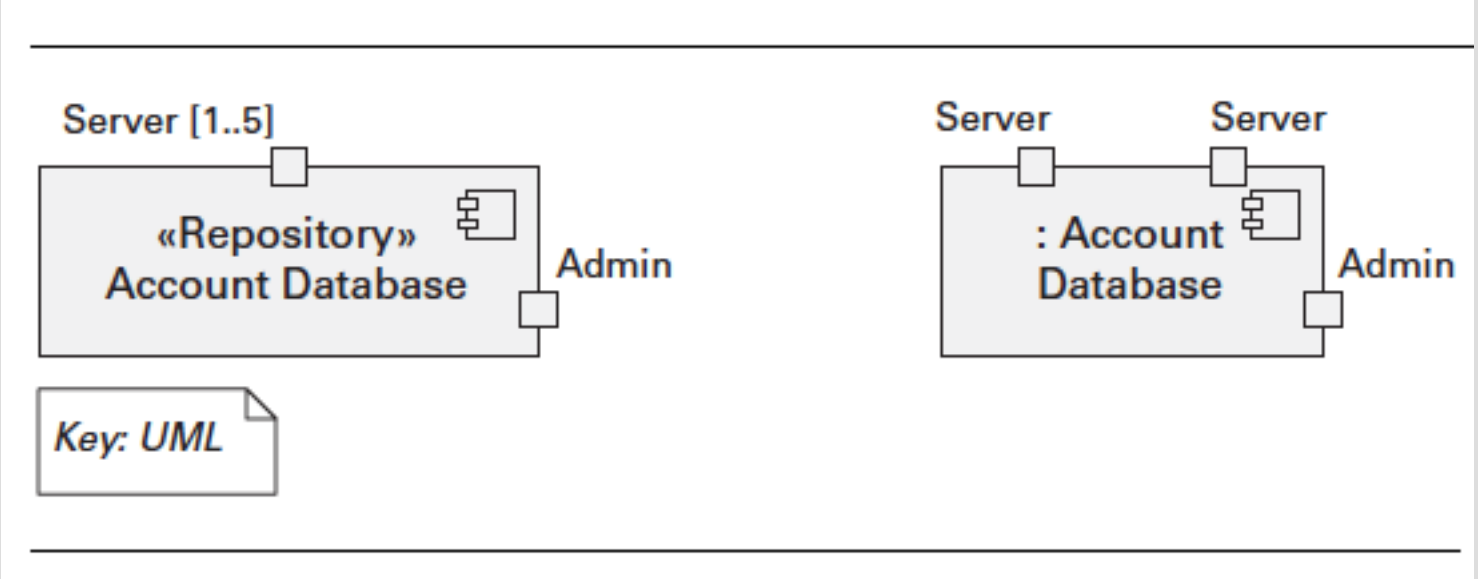
- Elements
  - *Components*. Principal processing units and data stores. A component has a set of *ports* through which it interacts with other components (via connectors).
  - *Connectors*. Pathways of interaction between components. Connectors have a set of roles (interfaces) that indicate how components 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.



Notations for C&C Views

- UML connectors are not rich enough to represent many C&C connectors.
  - UML connectors cannot have substructure, attributes, or behavioral descriptions.
- Represent a “simple” C&C connector using a UML connector—a line.
  - Many commonly used C&C connectors have well-known, application-independent semantics and implementations, such as function calls or data read operations.
  - You can use a stereotype to denote the type of connector.
- Connector roles cannot be explicitly represented with a UML connector.
  - The UML connector element does not allow the inclusion of interfaces.
  - Label the connector ends and use these labels to identify role descriptions that must be documented elsewhere.
- Represent a “rich” C&C connector using a UML component, or by annotating a line UML connector with a tag that explains the meaning of the complex connector.

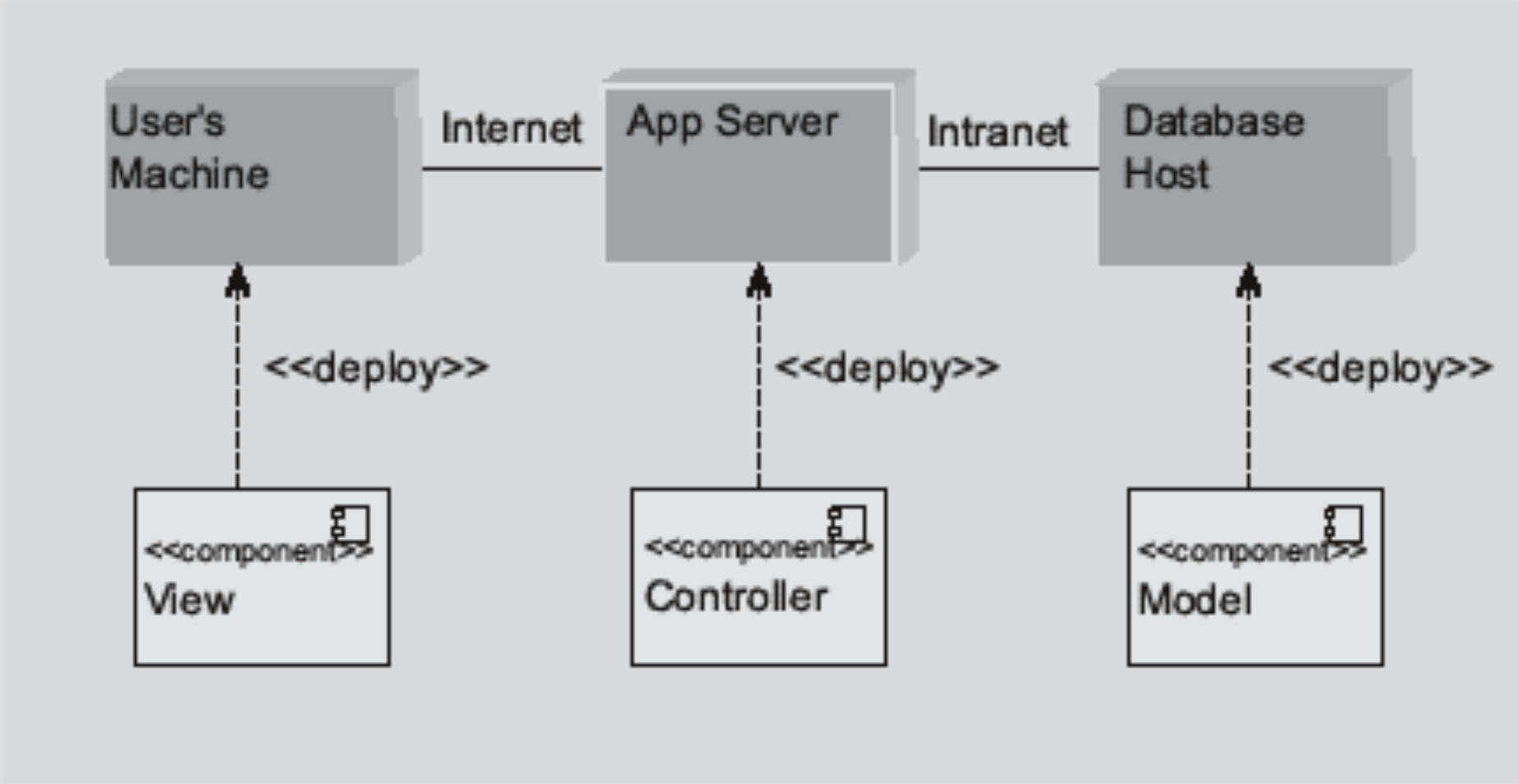
Overview of Allocation Views

- 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.

Allocation Model for MVC



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.

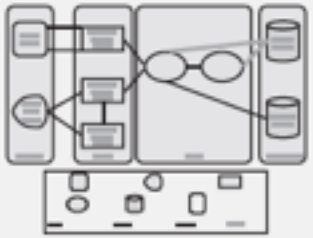
Quality Views: Examples

- *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.

View Template

Template for a View

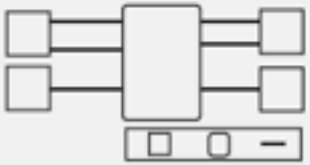
Section 1. Primary Presentation



Section 2. Element Catalog

- Section 2.A. Elements and Their Properties
- Section 2.B. Relations and Their Properties
- Section 2.C. Element Interfaces
- Section 2.D. Element Behavior

Section 3. Context Diagram



Section 4. Variability Guide

Section 5. Rationale

Documenting Information Beyond Views

- **Section 1: Documentation Roadmap.** The documentation map tells the reader what information is in the documentation and where to find it.
  - *Scope and summary*. Explain the purpose of the document and briefly summarize what is covered.
  - *How the documentation is organized*. For each section in the documentation, give a short synopsis of the information that can be found there.
  - *View overview*. Describes the views that the architect has included in the package. For each view::
    - The name of the view and what pattern it instantiates, if any.
    - A description of the view's element types, relation types, and property types.
    - A description of language, modeling techniques, or analytical methods used in constructing the view.
  - *How stakeholders can use the documentation*.
    - This section shows how various stakeholders might use the documentation to help address their concerns.
    - Include short scenarios, such as "A maintainer wishes to know the units of software that are likely to be changed by a proposed modification."
    - To be compliant with ISO/IEC 42010-2007, you must consider the concerns of at least users, acquirers, developers, and maintainers.

Documenting Information Beyond Views

- **Section 2: How a View Is Documented.**
  - Explain the standard organization you're using to document views—either the one described in this chapter or one of your own.
- **Section 3: System Overview.**
  - Short prose description of the system's function, its users, and any important background or constraints.
  - Provides your readers with a consistent mental model of the system and its purpose.
  - This might be a pointer to your project's concept-of-operations document for the system.

Documenting Information Beyond Views

- **Section 4: Mapping Between Views.**
  - Helping a reader understand the associations between views will help that reader gain a powerful insight into how the architecture works as a unified conceptual whole.
  - The associations between elements across views in an architecture are, in general, many-to-many.
  - View-to-view associations can be captured as tables.
    - The table should name the correspondence between the elements across the two views.
    - Examples
      - "is implemented by" for mapping from a component-and-connector view to a module view
      - "implements" for mapping from a module view to a component-and-connector view
      - "included in" for mapping from a decomposition view to a layered view

Documenting Information Beyond Views

- **Section 5: Rationale.**
  - Documents the architectural decisions that apply to more than one view.
    - Documentation of background or organizational constraints or major requirements that led to decisions of system-wide import.
    - Decisions about which fundamental architecture patterns are used.
- **Section 6: Directory.**
  - Set of reference material that helps readers find more information quickly.
    - Index of terms
    - Glossary
    - Acronym list.

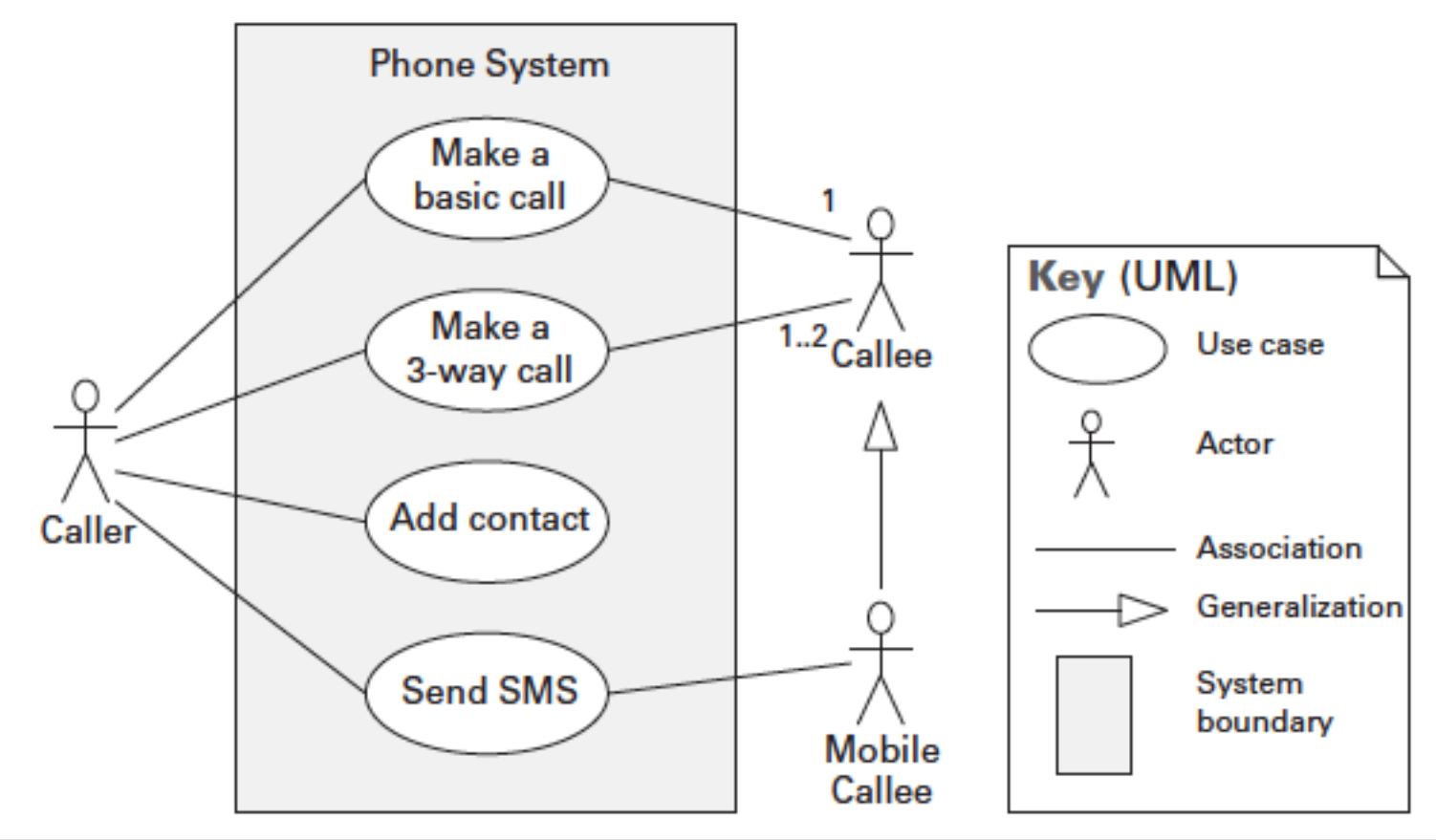
Documenting Behavior

- Behavior documentation complements each views by describing how architecture elements in that view interact with each other.
- Behavior documentation enables reasoning about
  - a system's potential to deadlock
  - a system's ability to complete a task in the desired amount of time
  - maximum memory consumption
  - and more
- Behavior has its own section in our view template's element catalog.

Notations for Documenting Behavior

- Trace-oriented languages
  - *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.
  - A trace describes a particular sequence of activities or interactions between structural elements of the system.
  - Examples
    - use cases
    - sequence diagrams
    - communication diagrams
    - activity diagrams
    - message sequence charts
    - timing diagrams
    - Business Process Execution Language

Use Case Diagram



From *Documenting Software Architectures: Views and Beyond*, 2nd edition, Addison Wesley, 2010

Use Case Description

Name: Make a basic call

Description: Making a point-to-point connection between two phones.

Primary actors: Caller

Secondary actors: Callee

Flow of events:

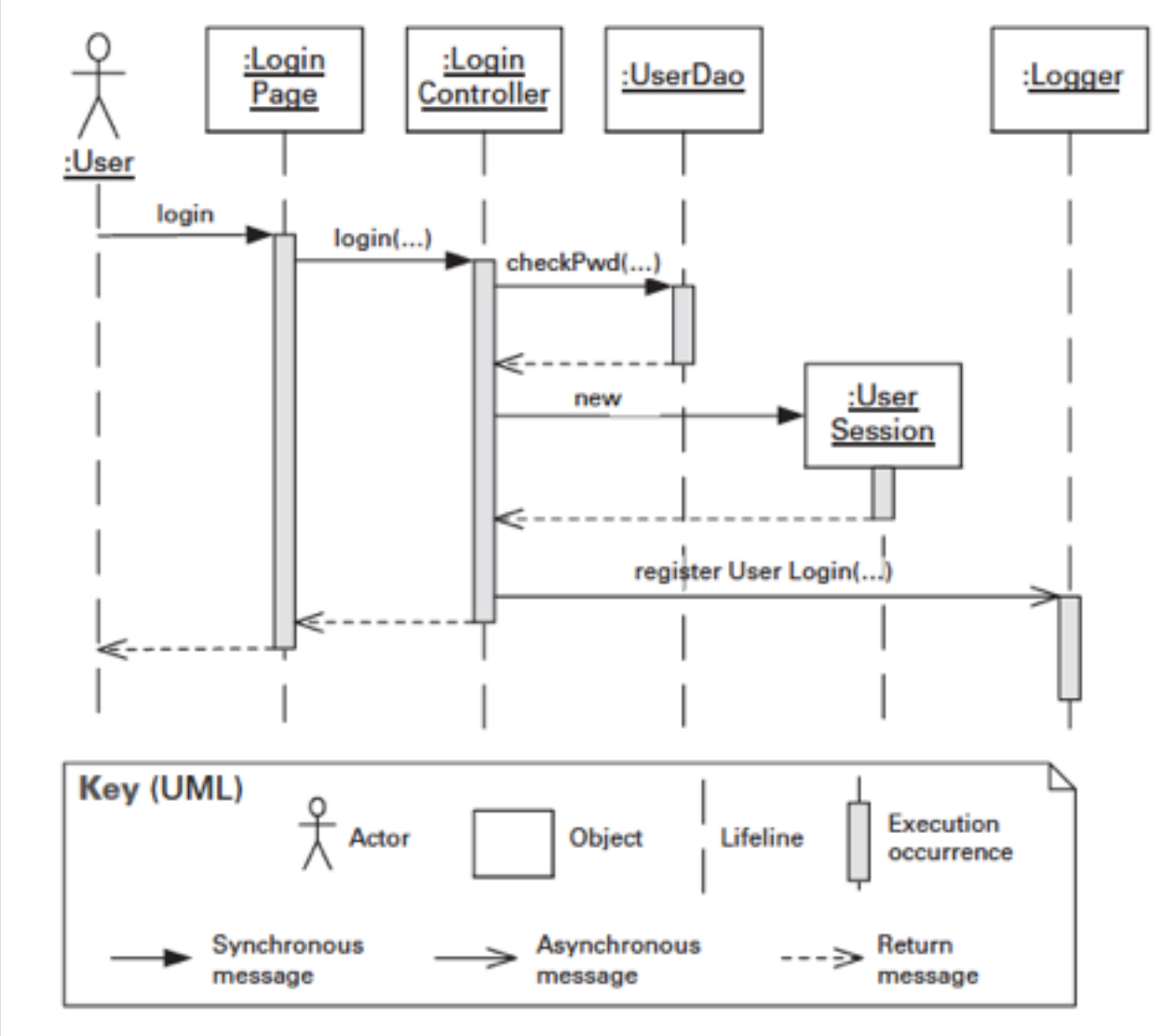
The use case starts when a caller places a call via a terminal, such as a cell phone. All terminals to which the call should be routed then begin ringing. When one of the terminals is answered, all others stop ringing and a connection is made between the caller's terminal and the terminal that was answered. When either terminal is disconnected—someone hangs up—the other terminal is also disconnected. The call is now terminated, and the use case is ended.

Exceptional flow of events:

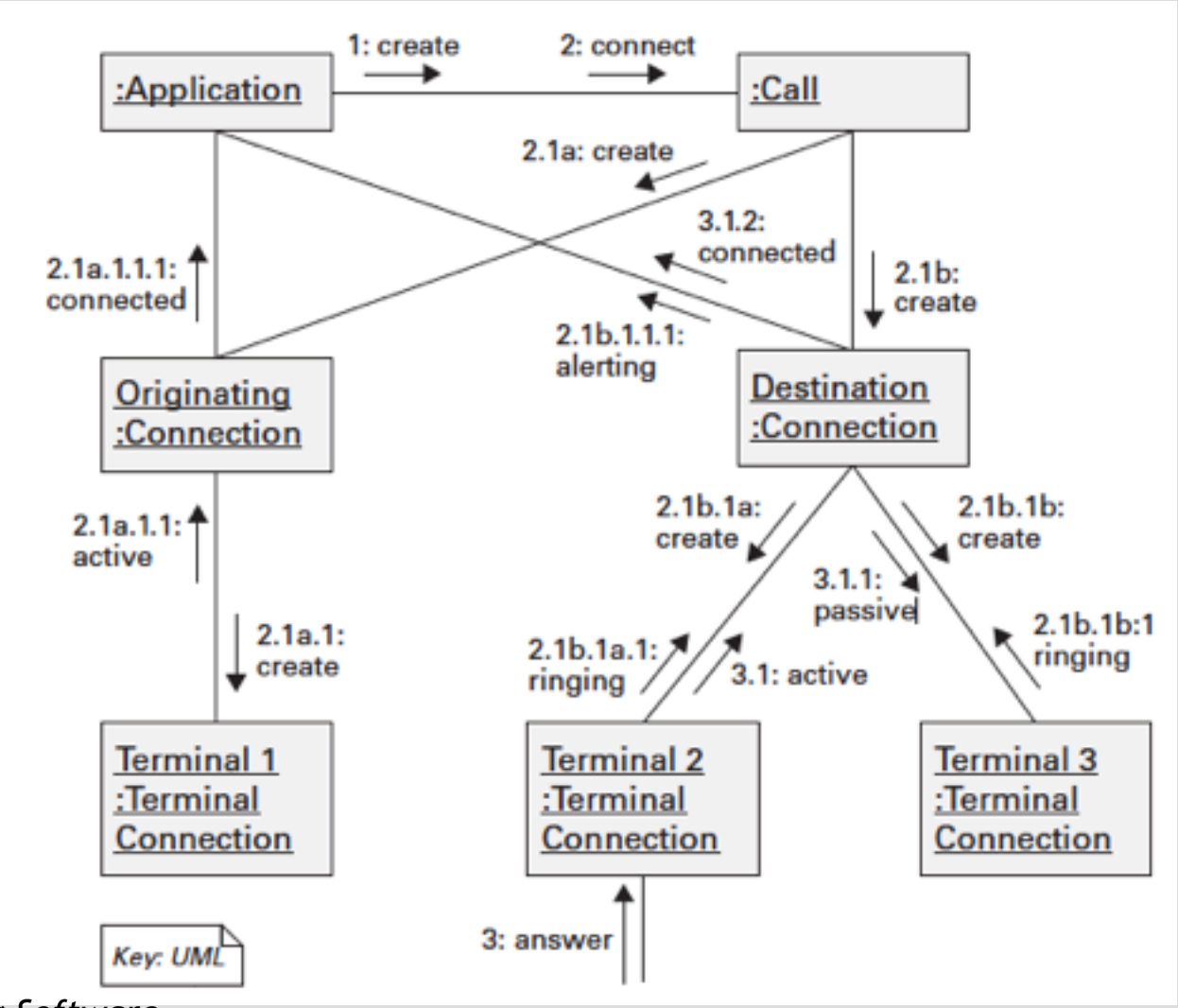
The caller can disconnect, or hang up, before any of the ringing terminals has been answered. If this happens, all ringing terminals stop ringing and are disconnected, ending the use case.

From *Documenting Software Architectures: Views and Beyond*, 2nd edition, Addison Wesley, 2010

Sequence Diagram

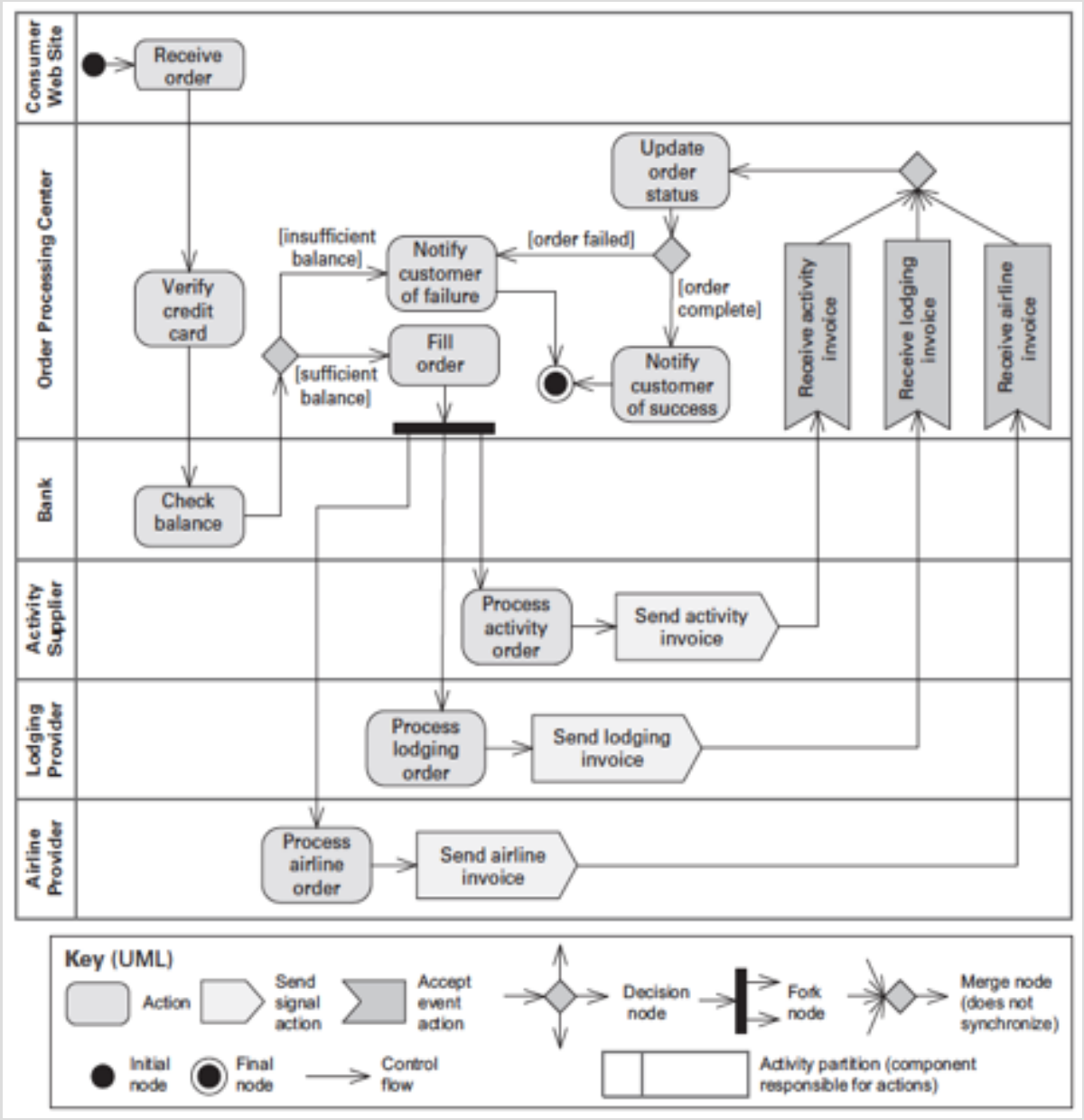


Communication Diagram



From *Documenting Software Architectures: Views and Beyond*, 2nd edition, Addison Wesley, 2010

Activity Diagram

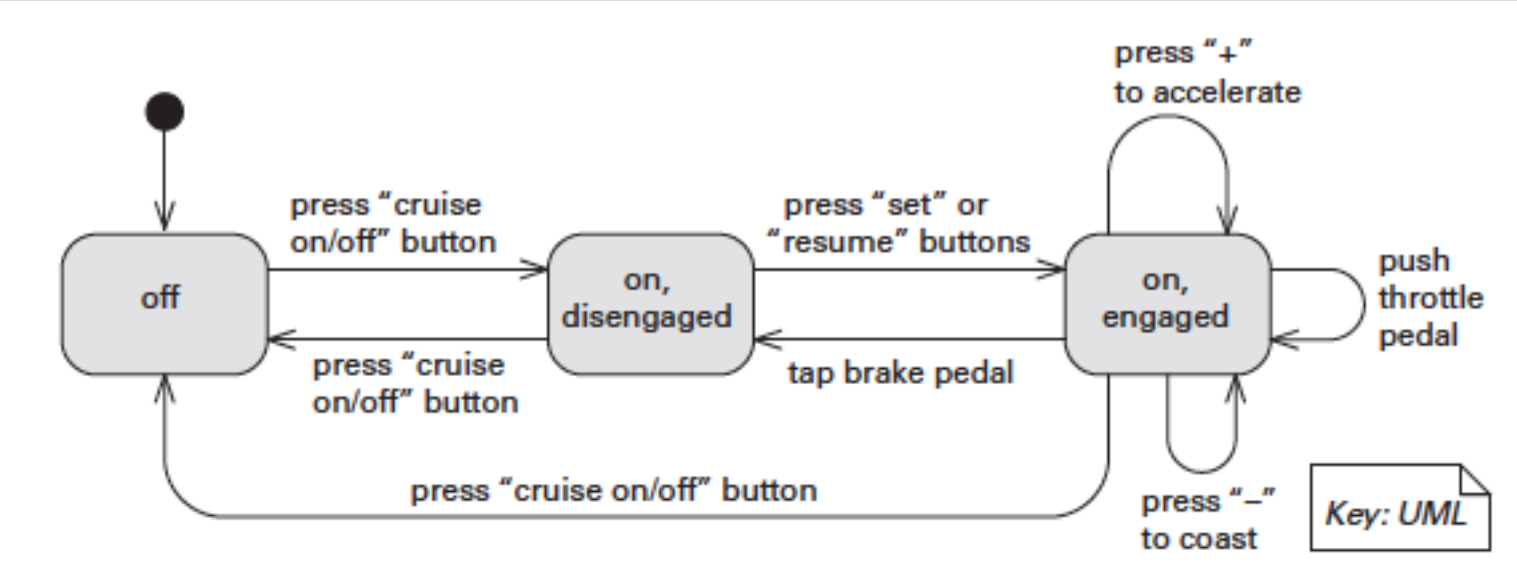


From *Documenting Software Architectures: Views and Beyond*, 2nd edition, Addison Wesley, 2010

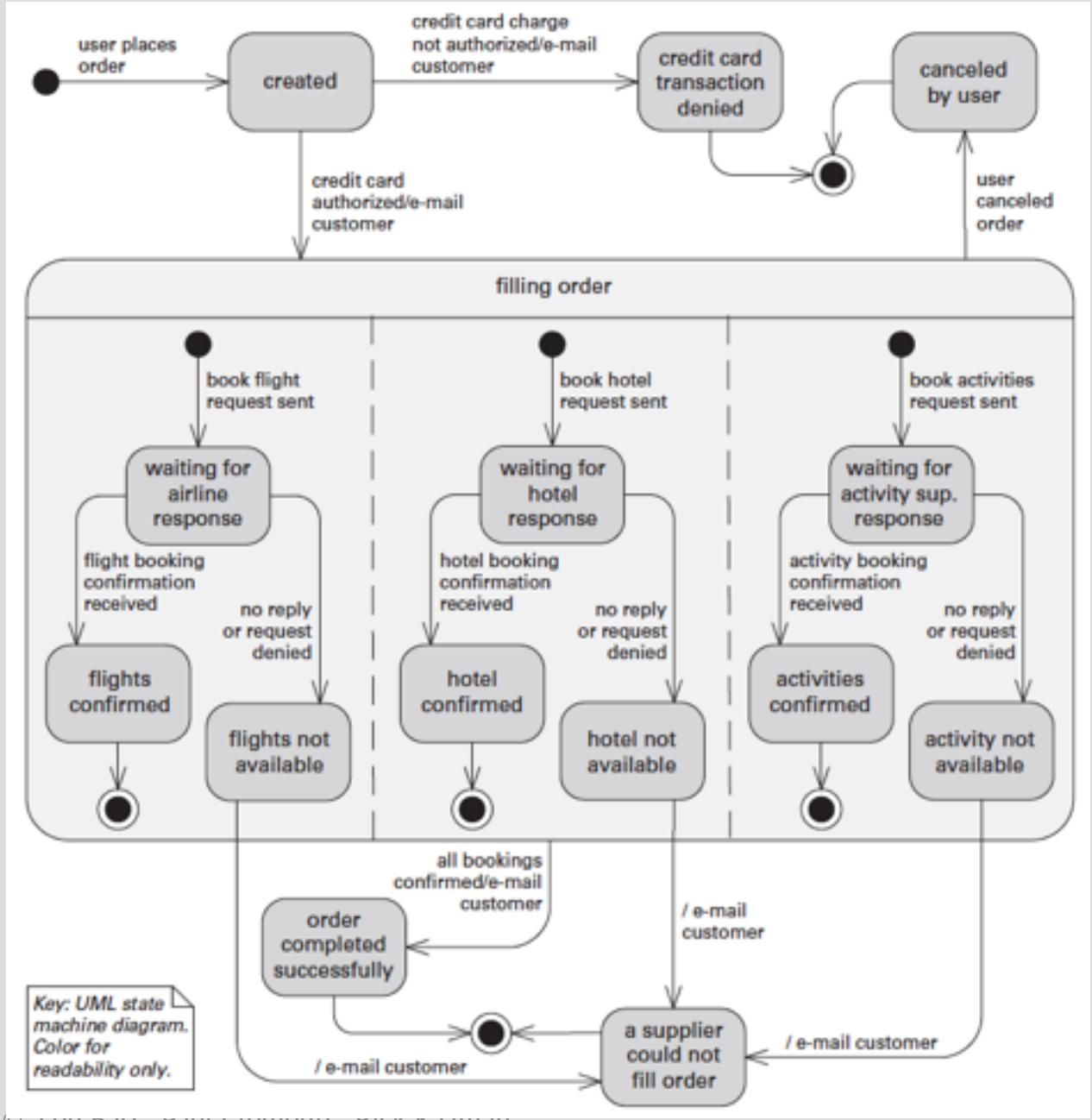
Notations for Documenting Behavior

- Comprehensive languages
  - *Comprehensive models* show the complete behavior of structural elements.
  - Given this type of documentation, it is possible to infer all possible paths from initial state to final state.
  - The state machine formalism represents the behavior of architecture elements because each state is an abstraction of all possible histories that could lead to that state.
  - State machine languages allow you to complement a structural description of the elements of the system with constraints on interactions and timed reactions to both internal and environmental stimuli.

State Machine



State Machine



From *Documenting Software Architectures: Views and Beyond*, 2nd edition, Addison Wesley, 2010

Documenting Quality Attributes

- Where do quality attributes show up in the documentation? There are five major ways:
  - Rationale that explains the choice of design approach should include a discussion about the quality attribute requirements and tradeoffs.
  - Architectural elements providing a service often have quality attribute bounds assigned to them, defined in the interface documentation for the elements, or recorded as *properties* that the elements exhibit.
  - Quality attributes often impart a "language" of things that you would look for. Someone fluent in the "language" of a quality attribute can search for the kinds of architectural elements) put in place to satisfy that quality attribute requirement.
  - Architecture documentation often contains a *mapping to requirements* that shows how requirements (including quality attribute requirements) are satisfied.
  - Every quality attribute requirement will have a constituency of stakeholders who want to know that it is going to be satisfied. For these stakeholders, the roadmap tells the stakeholder where in the document to find it.

Documenting Architectures That Change Faster Than You Can Document Them

- An architecture that changes at runtime, or as a result of a high-frequency release-and-deploy cycle, change much faster than the documentation cycle.
- Nobody will wait until a new architecture document is produced, reviewed, and released.
- In this case:
  - *Document what is true about all versions of your system*. Record those invariants as you would for any architecture. This may make your documented architecture more a description of constraints or guidelines that any compliant version of the system must follow.
  - *Document the ways the architecture is allowed to change*. This will usually mean adding new components and replacing components with new implementations. The place to do this is called the variability guide.

Documenting Architecture in an Agile Development Project

- Adopt a template or standard organization to capture your design decisions.
- Plan to document a view if (but only if) it has a strongly identified stakeholder constituency.
- Fill in the sections of the template for a view, and for information beyond views, when (and in whatever order) the information becomes available. But only do this if writing down this information will make it easier (or cheaper or make success more likely) for someone downstream doing their job.
- Don't worry about creating an architectural design document and then a finer-grained design document. Produce just enough design information to allow you to move on to code.
- Don't feel obliged to fill up all sections of the template, and certainly not all at once. Write "N/A" for the sections for which you don't need to record the information (perhaps because you will convey it orally).
- Agile teams sometimes make models in brief discussions by the whiteboard. Take a picture and use it as the primary presentation.