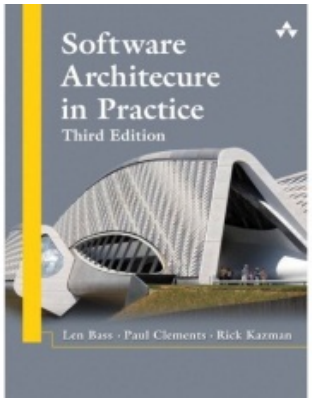
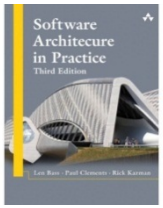
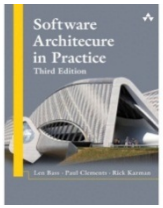


Chapter 1:
What is Software Architecture?



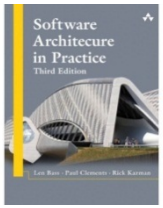
Chapter Outline

- What Software Architecture Is and What It Isn't
- Architectural Structures and Views
- Architectural Patterns
- What Makes a "Good" Architecture?
- Summary



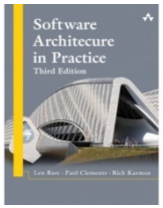
What is Software Architecture?

The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.

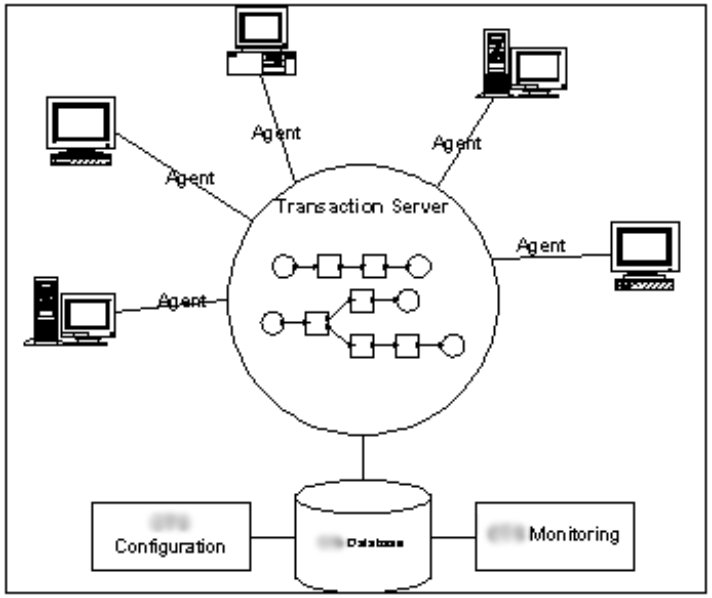


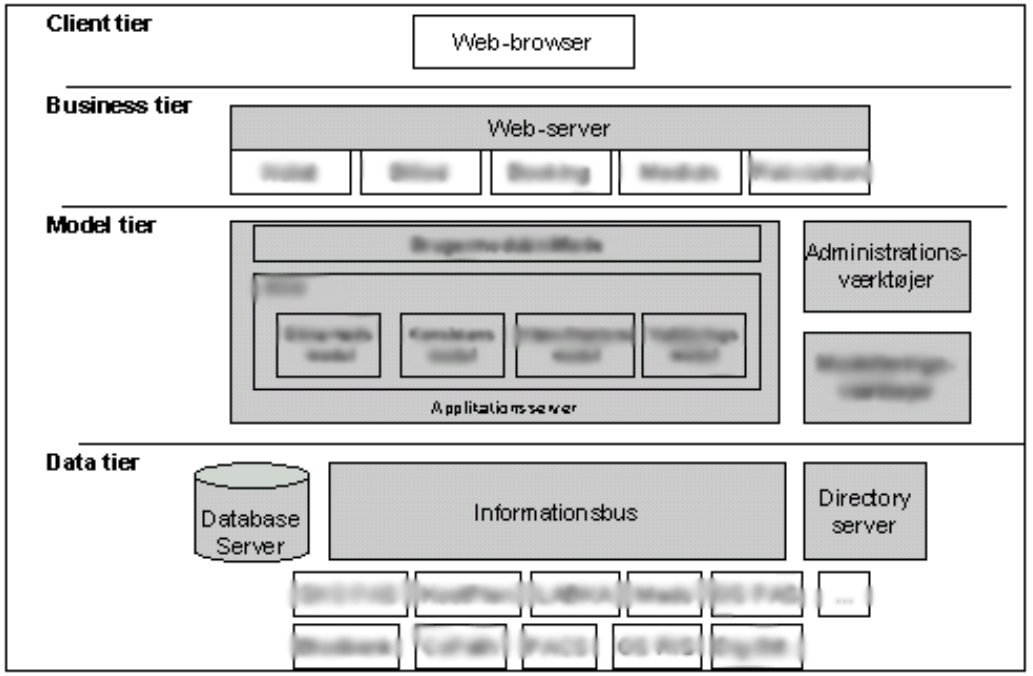
Definition

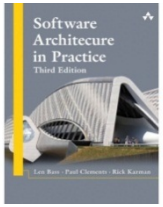
- This definition stands in contrast to other definitions that talk about the system's "early" or "major" design decisions.
  - Many architectural decisions are made early, but not all are.
  - Many decisions are made early that are not architectural.
  - It's hard to look at a decision and tell whether or not it's "major."
- Structures, on the other hand, are fairly easy to identify in software, and they form a powerful tool for system design.



Software Architecture?

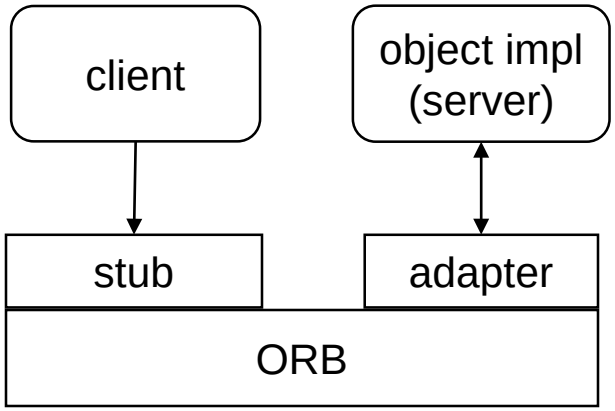




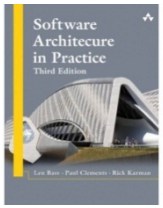


What is Software Architecture?

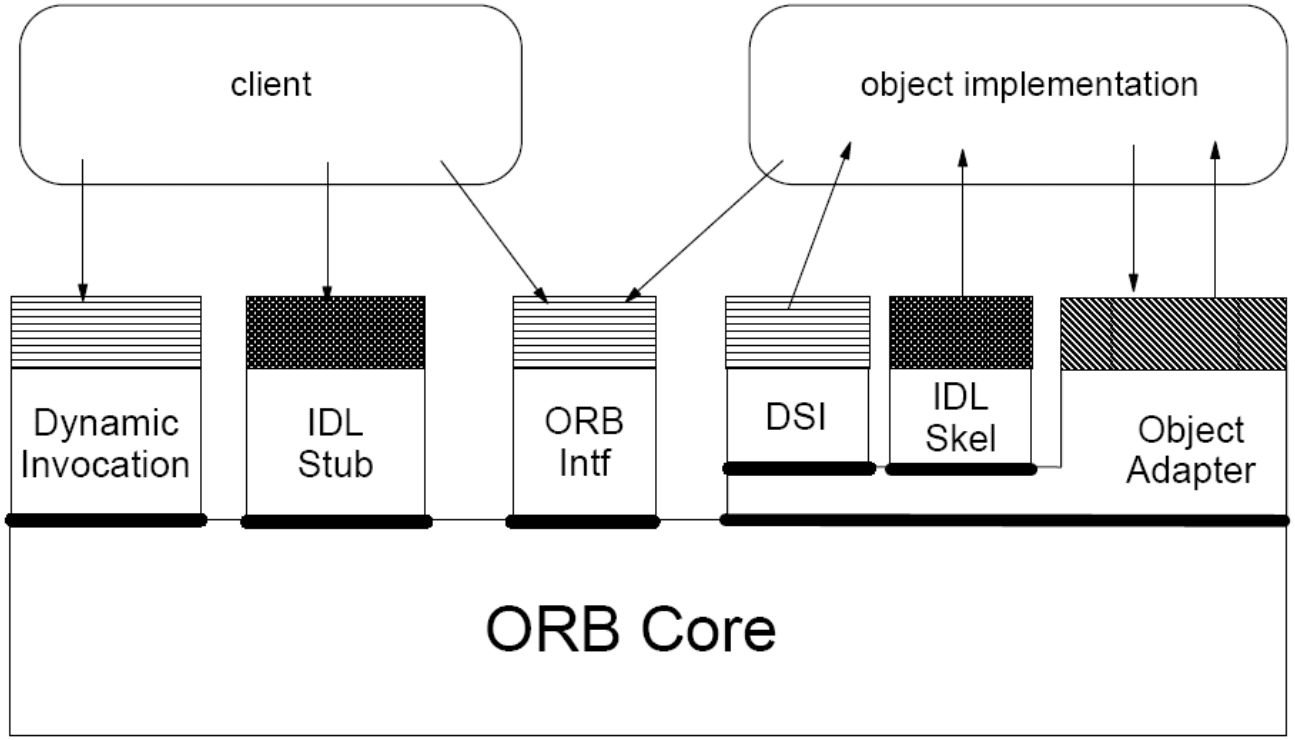
- *The architecture of CORBA:*
- *Common Object Request Broker Architecture*

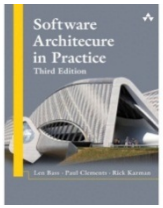


?



A Better Description?

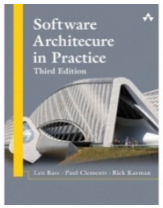




Box-n-line drawings

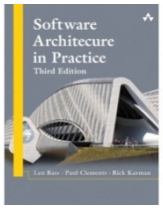
Hence, most of the box-and-line drawings that are passed off as architectures are in fact not architectures at all. They are simply box-and-line drawings.

[Bass et al.]



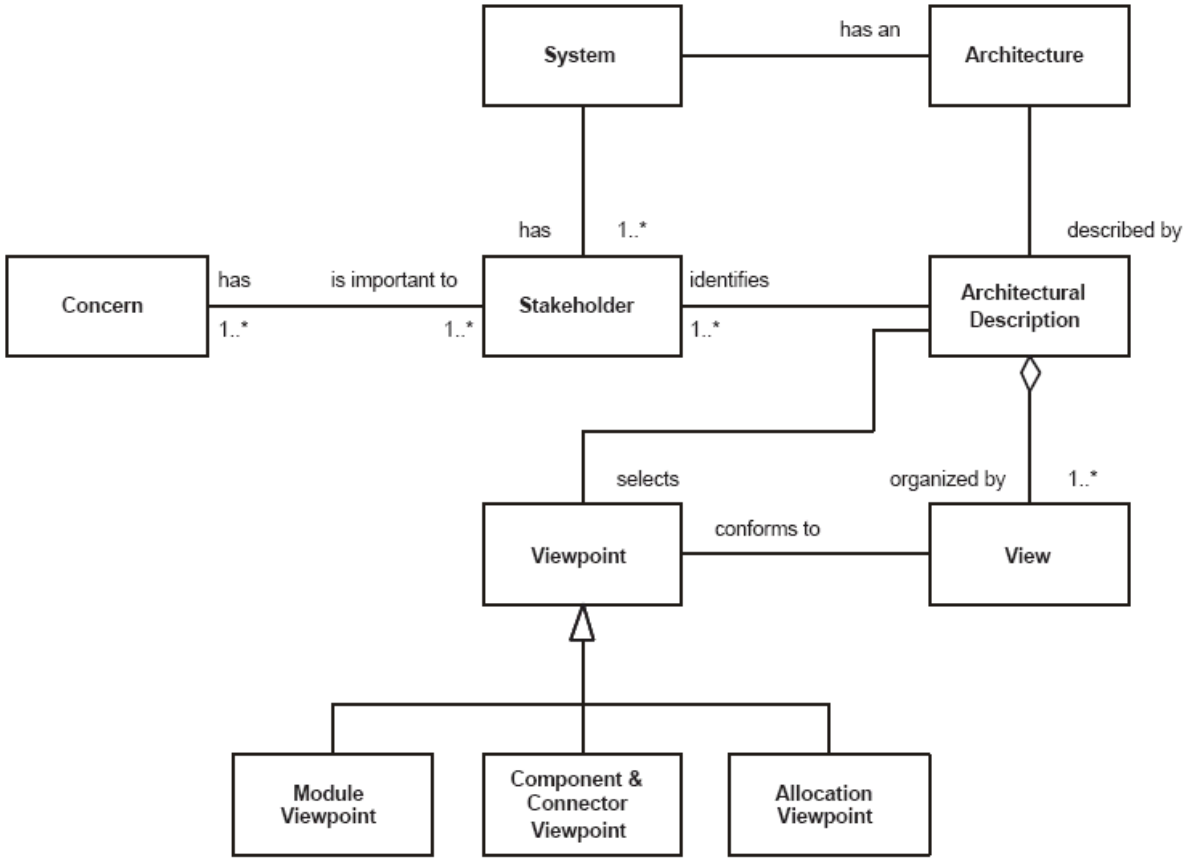
Architecture Is a Set of Software Structures

- A structure is a set of elements held together by a relation.
- Software systems are composed of many structures, and no single structure holds claim to being the architecture.
- There are three important categories of architectural structures.
  1. Module
  2. Component and Connector
  3. Allocation

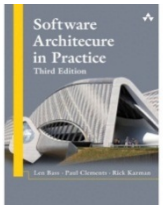


An Ontology of Architectural Descriptions

- Here: ontology = model of concepts

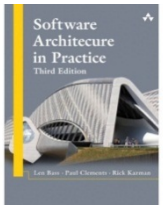


- Developed from [IEEE 1471, 2000]



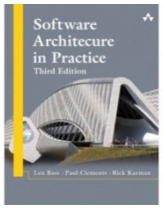
Module Structures

- Some structures partition systems into implementation units, which we call modules.
- Modules are assigned specific computational responsibilities, and are the basis of work assignments for programming teams.
- In large projects, these elements (modules) are subdivided for assignment to sub-teams.



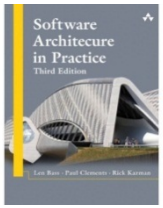
Component-and-connector Structures

- Other structures focus on the way the elements interact with each other at runtime to carry out the system's functions.
- We call runtime structures *component-and-connector (C&C) structures*.
- In our use, a component is always a runtime entity.
  - Suppose the system is to be built as a set of services.
  - The services, the infrastructure they interact with, and the synchronization and interaction relations among them form another kind of structure often used to describe a system.
  - These services are made up of (compiled from) the programs in the various implementation units – modules.



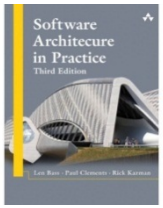
Allocation Structures

- Allocation structures describe the mapping from software structures to the system's environments
  - organizational
  - developmental
  - installation
  - Execution
- For example
  - Modules are assigned to teams to develop, and assigned to places in a file structure for implementation, integration, and testing.
  - Components are deployed onto hardware in order to execute.



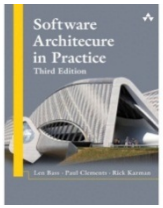
Which Structures are Architectural?

- A structure is architectural if it supports reasoning about the system and the system's properties.
- The reasoning should be about an attribute of the system that is important to some stakeholder.
- These include
  - functionality achieved by the system
  - the availability of the system in the face of faults
  - the difficulty of making specific changes to the system
  - the responsiveness of the system to user requests,
  - many others.



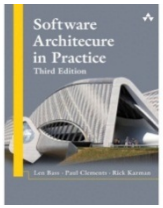
Architecture is an Abstraction

- An architecture comprises software elements and how the elements relate to each other.
  - An architecture specifically omits certain information about elements that is not useful for reasoning about the system.
  - It omits information that has no ramifications outside of a single element.
  - An architecture selects certain details and suppresses others.
  - Private details of elements—details having to do solely with internal implementation—are not architectural.
- The architectural abstraction lets us look at the system in terms of its elements, how they are arranged, how they interact, how they are composed, what their properties are that support our system reasoning, and so forth.
- This abstraction is essential to taming the complexity of an architecture.
- We simply cannot, and do not want to, deal with all of the complexity all of the time.



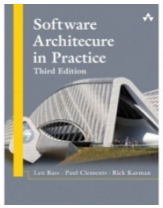
Every System has a Software Architecture

- Every system comprises elements and relations among them to support some type of reasoning.
- But the architecture may not be known to anyone.
  - Perhaps all of the people who designed the system are long gone
  - Perhaps the documentation has vanished (or was never produced)
  - Perhaps the source code has been lost (or was never delivered)
- An architecture can exist independently of its description or specification.
- Documentation is critical.

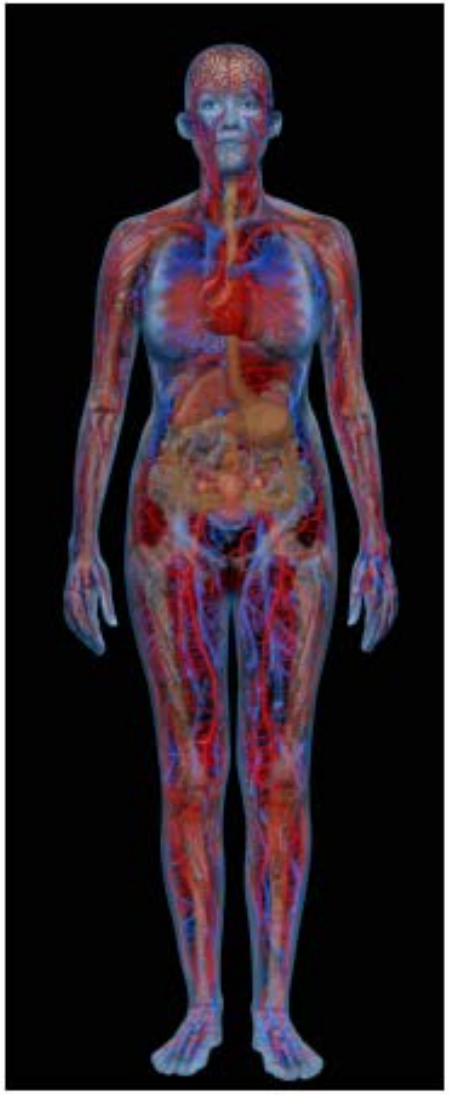


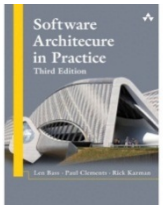
Architecture Includes Behavior

- The behavior of each element is part of the architecture insofar as that behavior can be used to reason about the system.
- This behavior embodies how elements interact with each other, which is clearly part of the definition of architecture.
- Box-and-line drawings that are passed off as architectures are not architectures at all.
  - When looking at the names of the a reader may well imagine the functionality and behavior of the corresponding elements.
  - But it relies on information that is not present – and could be wrong!
- This does not mean that the exact behavior and performance of every element must be documented in all circumstances.
  - Some aspects of behavior are fine-grained and below the architect's level of concern.
- To the extent that an element's behavior influences another element or influences the acceptability of the system as a whole, this behavior must be considered, and should be documented, as part of the software architecture.



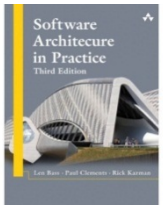
Physiological Structures





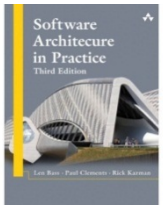
Structures and Views

- A *view* is a representation of a coherent set of architectural elements, as written by and read by system stakeholders.
  - A view consists of a representation of a set of elements and the relations among them.
- A *structure* is the set of elements itself, as they exist in software or hardware.
- In short, **a view is a representation of a structure**.
  - For example, a module *structure* is the set of the system's modules and their organization.
  - A module *view* is the representation of that structure, documented according to a template in a chosen notation, and used by some system stakeholders.
- Architects design structures. They document views of those structures.



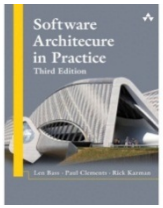
Module Structures

- Module structures embody decisions as to how the system is to be structured as a set of code or data units that have to be constructed or procured.
- In any module structure, the elements are modules of some kind (perhaps classes, or layers, or merely divisions of functionality, all of which are units of implementation).
- Modules are assigned areas of functional responsibility; there is less emphasis in these structures on how the software manifests at runtime.
- Module structures allow us to answer questions such as these:
  - What is the primary functional responsibility assigned to each module?
  - What other software elements is a module allowed to use?
  - What other software does it actually use and depend on?
  - What modules are related to other modules by generalization or specialization (i.e., inheritance) relationships?



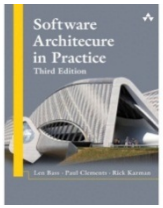
Component-and-connector Structures

- Component-and-connector structures embody decisions as to how the system is to be structured as a set of elements that have runtime behavior (components) and interactions (connectors).
- Elements are runtime comopnents such as services, peers, clients, servers, filters, or many other types of runtime element)
- Connectors are the communication vehicles among components, such as call-return, process synchronization operators, pipes, or others.
- Component-and-connector views help us answer questions such as these:
  - What are the major executing components and how do they interact at runtime?
  - What are the major shared data stores?
  - Which parts of the system are replicated?
  - How does data progress through the system?
  - What parts of the system can run in parallel?
  - Can the system's structure change as it executes and, if so, how?
- Component-and-connector views are crucially important for asking questions about the system's runtime properties such as performance, security, availability, and more.



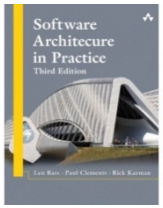
Allocation structures

- Allocation structures show the relationship between the software elements and elements in one or more external environments in which the software is created and executed.
- Allocation views help us answer questions such as these:
  - What processor does each software element execute on?
  - In what directories or files is each element stored during development, testing, and system building?
  - What is the assignment of each software element to development teams?



Structures Provide Insight

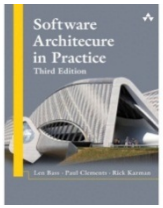
- Structures play such an important role in our perspective on software architecture because of the analytical and engineering power they hold.
- Each structure provides a perspective for reasoning about some of the relevant quality attributes.
- For example:
  - The module structure, which embodies what modules use what other modules, is strongly tied to the ease with which a system can be extended or contracted.
  - The concurrency structure, which embodies parallelism within the system, is strongly tied to the ease with which a system can be made free of deadlock and performance bottlenecks.
  - The deployment structure is strongly tied to the achievement of performance, availability, and security goals.
  - And so forth.



Some Useful Module Structures

Decomposition structure

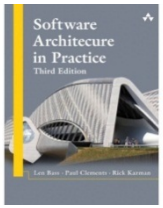
- The units are modules that are related to each other by the *is-a-submodule-of* relation.
- It shows how modules are decomposed into smaller modules recursively until the modules are small enough to be easily understood.
- Modules often have products (such as interface specifications, code, test plans, etc.) associated with them.
- The decomposition structure determines, to a large degree, the system's modifiability, by assuring that likely changes are localized.
- This structure is often used as the basis for the development project's organization, including the structure of the documentation, and the project's integration and test plans.
- The units in this structure tend to have names that are organization-specific such as "segment" or "subsystem."



Some Useful Module Structures

Uses structure.

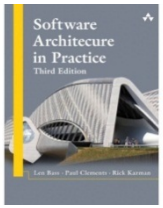
- The units here are also modules, perhaps classes.
- The units are related by the *uses* relation, a specialized form of dependency.
- A unit of software *uses* another if the correctness of the first requires the presence of a correctly functioning version (as opposed to a stub) of the second.
- The uses structure is used to engineer systems that can be extended to add functionality, or from which useful functional subsets can be extracted.
- The ability to easily create a subset of a system allows for incremental development.



Some Useful Module Structures

Layer structure

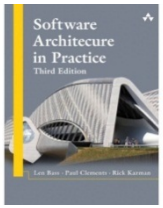
- The modules in this structure are called *layers*.
- A layer is an abstract "virtual machine" that provides a cohesive set of services through a managed interface.
- Layers are *allowed to use* other layers in a strictly managed fashion.
  - In strictly layered systems, a layer is only allowed to use a single other layer.
- This structure is imbues a system with portability, the ability to change the underlying computing platform.



Some Useful Module Structures

Class (or generalization) structure

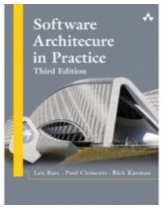
- The module units in this structure are called *classes*.
- The relation is *inherits from* or *is an instance of*.
- This view supports reasoning about collections of similar behavior or capability
  - e.g.,the classes that other classes inherit from and parameterized differences
- The class structure allows one to reason about reuse and the incremental addition of functionality.
- If any documentation exists for a project that has followed an object-oriented analysis and design process, it is typically this structure.



Some Useful Module Structures

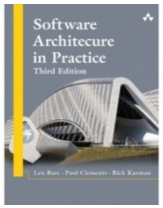
Data model

- The data model describes the static information structure in terms of data entities and their relationships.
  - For example, in a banking system, entities will typically include Account, Customer, and Loan.
  - Account has several attributes, such as account number, type (savings or checking), status, and current balance.



Some Useful C&C Structures

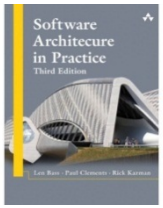
- The relation in all component-and-connector structures is attachment, showing how the components and the connectors are hooked together.
- The connectors can be familiar constructs such as “invokes.”
- Useful C&C structures include:
  - Service structure
    - The units are services that interoperate with each other by service coordination mechanisms such as SOAP.
    - The service structure helps to engineer a system composed of components that may have been developed anonymously and independently of each other.
  - Concurrency structure
    - This structure helps determine opportunities for parallelism and the locations where resource contention may occur.
    - The units are components
    - The connectors are their communication mechanisms.
    - The components are arranged into logical threads.



Some Useful Allocation Structures

Deployment structure

- The deployment structure shows how software is assigned to hardware processing and communication elements.
- The elements are software elements (usually a process from a C&C view), hardware entities (processors), and communication pathways.
- Relations are allocated-to, showing on which physical units the software elements reside, and migrates-to if the allocation is dynamic.
- This structure can be used to reason about performance, data integrity, security, and availability.
- It is of particular interest in distributed and parallel systems.



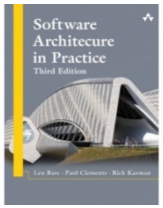
Some Useful Allocation Structures

Implementation structure

- This structure shows how software elements (usually modules) are mapped to the file structure(s) in the system's development, integration, or configuration control environments.
- This is critical for the management of development activities and build processes.

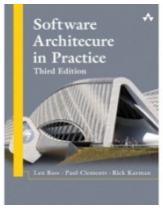
Work assignment structure

- This structure assigns responsibility for implementing and integrating the modules to the teams who will carry it out.
- Having a work assignment structure be part of the architecture makes it clear that the decision about who does the work has architectural as well as management implications.
- The architect will know the expertise required on each team.
- This structure will also determine the major communication pathways among the teams: regular teleconferences, wikis, email lists, and so forth.



Relating Structures to Each Other

- Elements of one structure will be related to elements of other structures, and we need to reason about these relations.
  - A module in a decomposition structure may be manifested as one, part of one, or several components in one of the component-and-connector structures.
- In general, mappings between structures are many to many.



Modules vs. Components

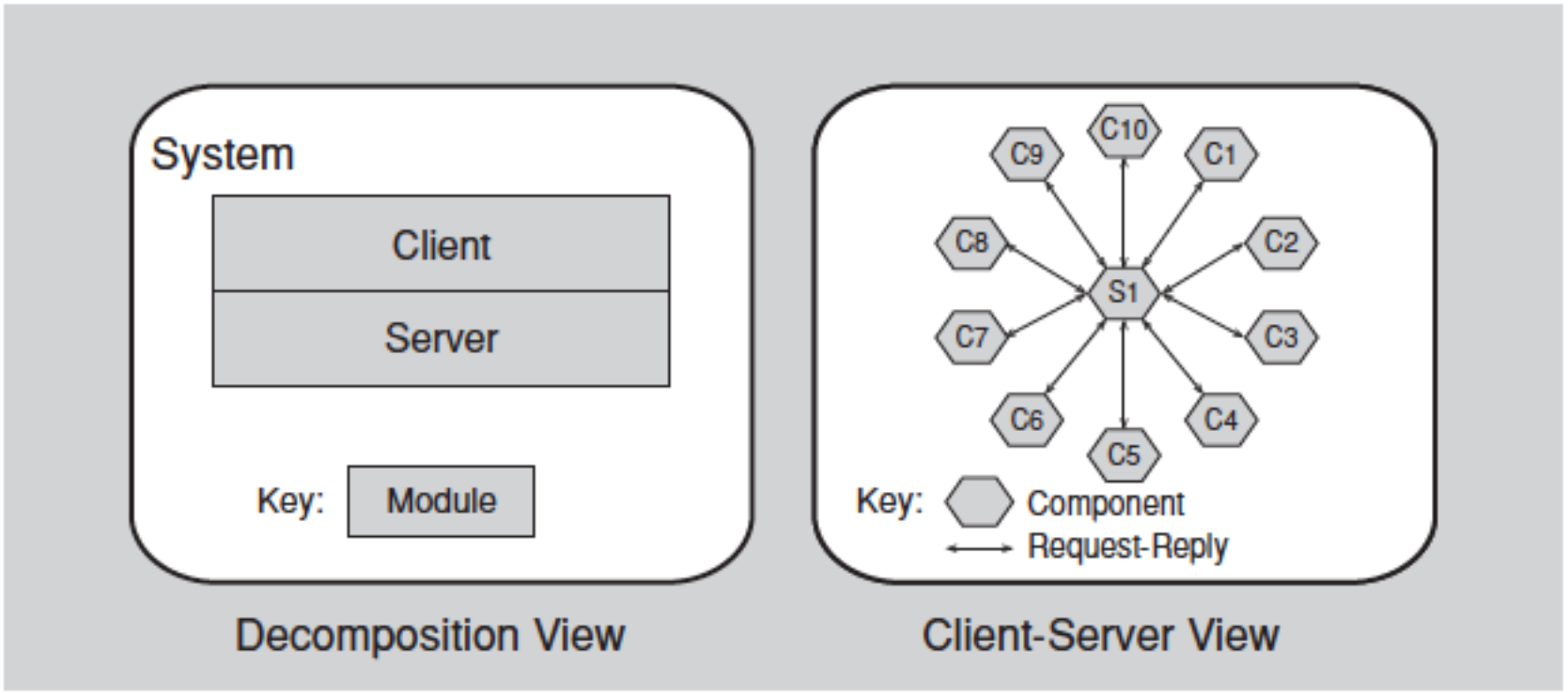
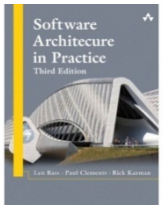


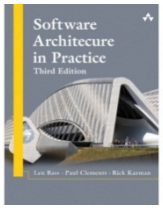
FIGURE 1.2 Two views of a client-server system



What Makes a “Good” Architecture?

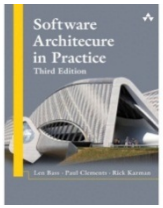
- There is no such thing as an inherently good or bad architecture.
- Architectures are either more or less fit for some purpose
- Architectures can be evaluated but only in the context of specific stated goals.

- There are, however, good rules of thumb.



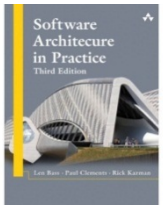
Process "Rules of Thumb"

- The architecture should be the product of a single architect or a small group of architects with an identified technical leader.
  - This approach gives the architecture its conceptual integrity and technical consistency.
  - This recommendation holds for Agile and open source projects as well as "traditional" ones.
  - There should be a strong connection between the architect(s) and the development team.
- The architect (or architecture team) should base the architecture on a prioritized list of well-specified quality attribute requirements.
- The architecture should be documented using views. The views should address the concerns of the most important stakeholders in support of the project timeline.
- The architecture should be evaluated for its ability to deliver the system's important quality attributes.
  - This should occur early in the life cycle and repeated as appropriate.
- The architecture should lend itself to incremental implementation,
  - Create a "skeletal" system in which the communication paths are exercised but which at first has minimal functionality.



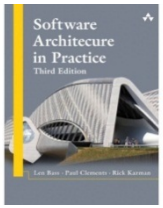
Structural “Rules of Thumb”

- The architecture should feature well-defined modules whose functional responsibilities are assigned on the principles of information hiding and separation of concerns.
  - The information-hiding modules should encapsulate things likely to change
  - Each module should have a well-defined interface that encapsulates or “hides” the changeable aspects from other software
- Unless your requirements are unprecedented your quality attributes should be achieved using well-known architectural patterns and tactics specific to each attribute.
- The architecture should never depend on a particular version of a commercial product or tool. If it must, it should be structured so that changing to a different version is straightforward and inexpensive.
- Modules that produce data should be separate from modules that consume data.
  - This tends to increase modifiability
  - Changes are frequently confined to either the production or the consumption side of data.



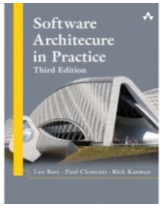
Structural “Rules of Thumb”

- Don’t expect a one-to-one correspondence between modules and components.
- Every process should be written so that its assignment to a specific processor can be easily changed, perhaps even at runtime.
- The architecture should feature a small number of ways for components to interact.
  - The system should do the same things in the same way throughout.
  - This will aid in understandability, reduce development time, increase reliability, and enhance modifiability.
- The architecture should contain a specific (and small) set of resource contention areas, the resolution of which is clearly specified and maintained.



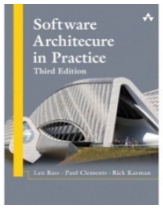
A Simple Case

- NextGen Point-Of-Sales (POS) System
  - Record sales and handle payments
    - Typically used in retail stores
  - Hardware
    - Terminal
    - Barcode scanner
  - Interfaces with external systems
    - Inventory
    - Accounting
    - …
- From [Larman, 2001]
  - See also note on architecture description using UML, [Christensen et al., 2004]

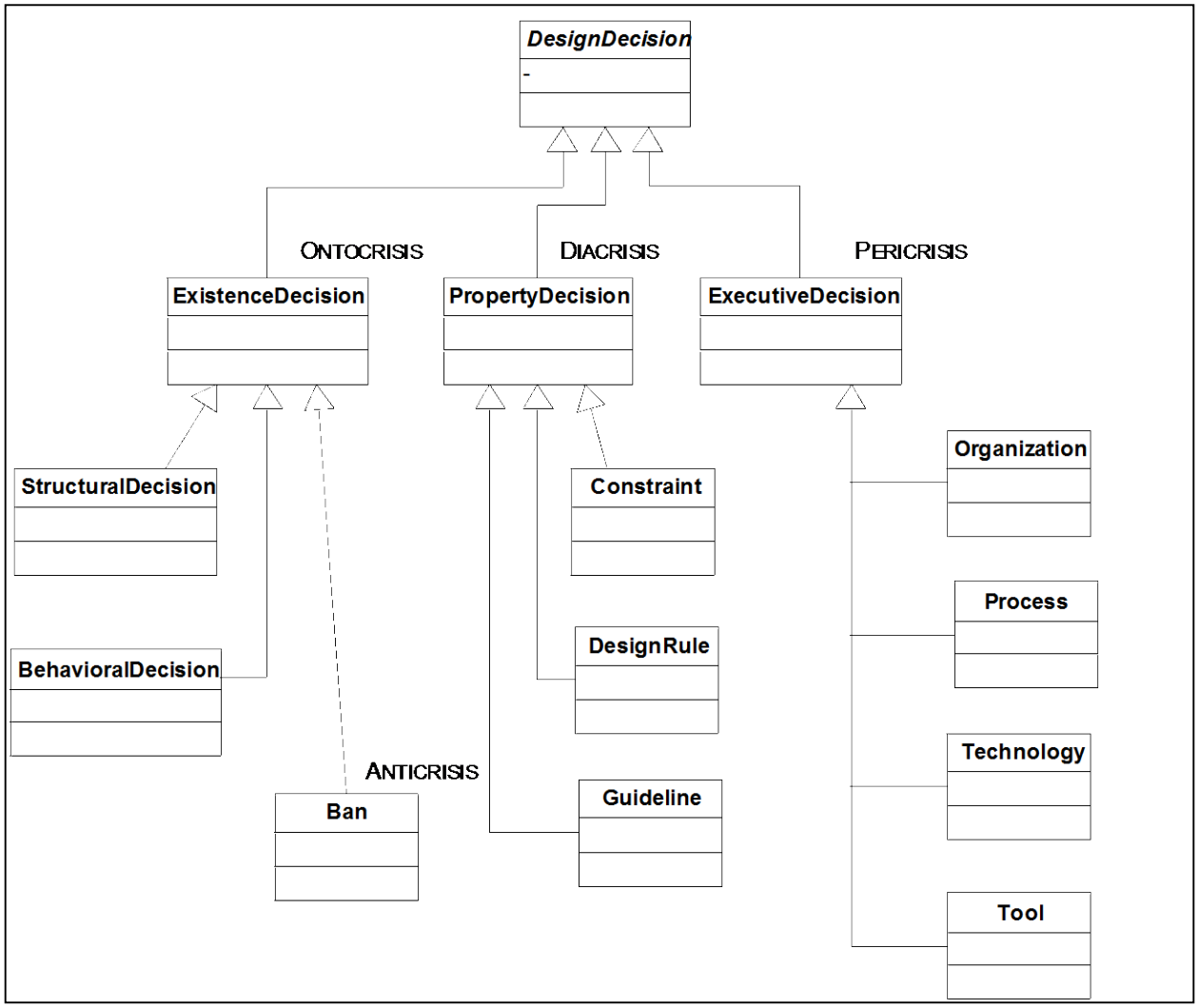


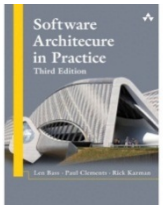
Architectural Requirements: POS Scenarios

Process Sale: A customer arrives at a checkout with items to purchase. The cashier uses the POS system to record each purchased item. The system presents a running total and line-item details. The customer enters payment information, which the system validates and records. The system updates inventory. The customer receives a receipt from the system and then leaves with the item



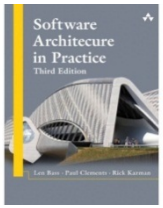
Types of Design Decisions [Kruchten, 2004]





Architectural Requirements: POS Qualities

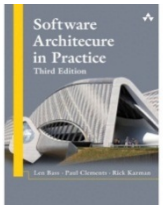
- Architectural drivers
  - Availability
    - The system shall be highly available since the effectiveness of sales depends on its availability
  - Portability
    - The system shall be portable to a range of different platforms to support a product line of POS systems
  - Usability
    - The system shall be usable by clerks with a minimum of training and with a high degree of efficiency
- This is not operational!
  - More later on quality attribute scenarios…



Component & Connector Viewpoint

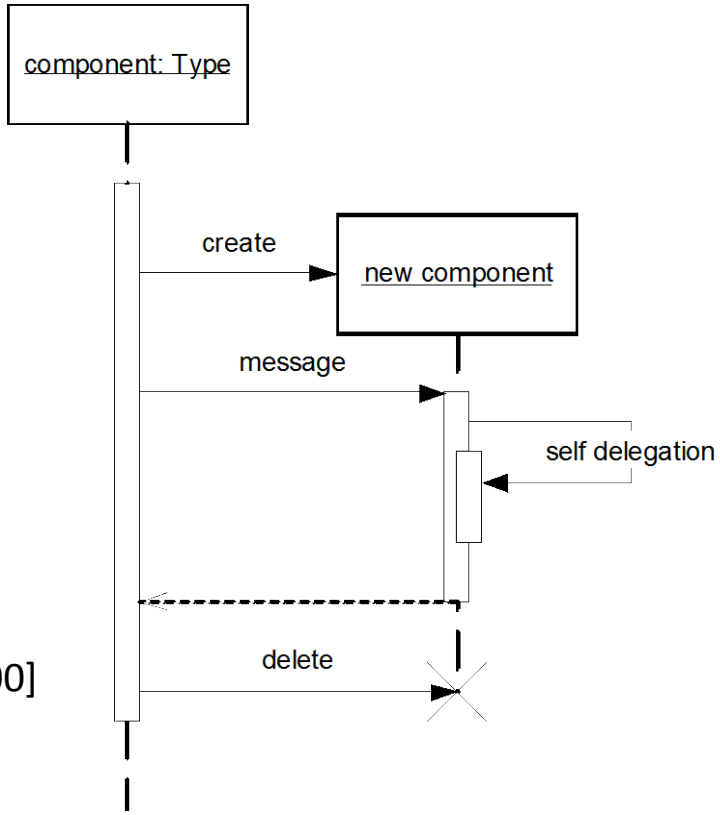
- Elements
  - Components
    - Functional behaviour
    - What part of the system is doing what?
- Relations
  - Connectors
    - Control and communication aspects
    - Define protocols for control and data exchange
      - Incoming and outgoing operations
      - Mandates ordering of operations
      - Define roles for attached components
- Mapping to UML
  - Object diagrams, interaction diagrams
  - Components = active objects
  - Connectors = links + annotations, messages
- + textual description of responsibilities

The software architecture of a computing system is the structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them [Bass et al., 2003]

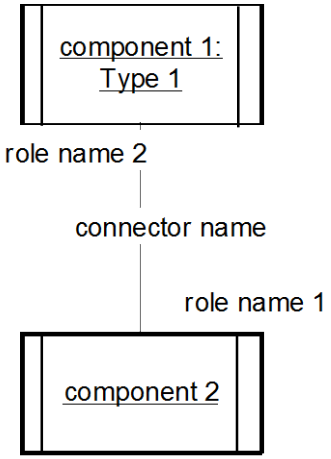


Basic C&C Elements

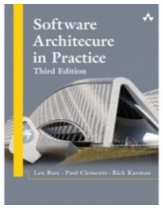
Sequence Diagrams



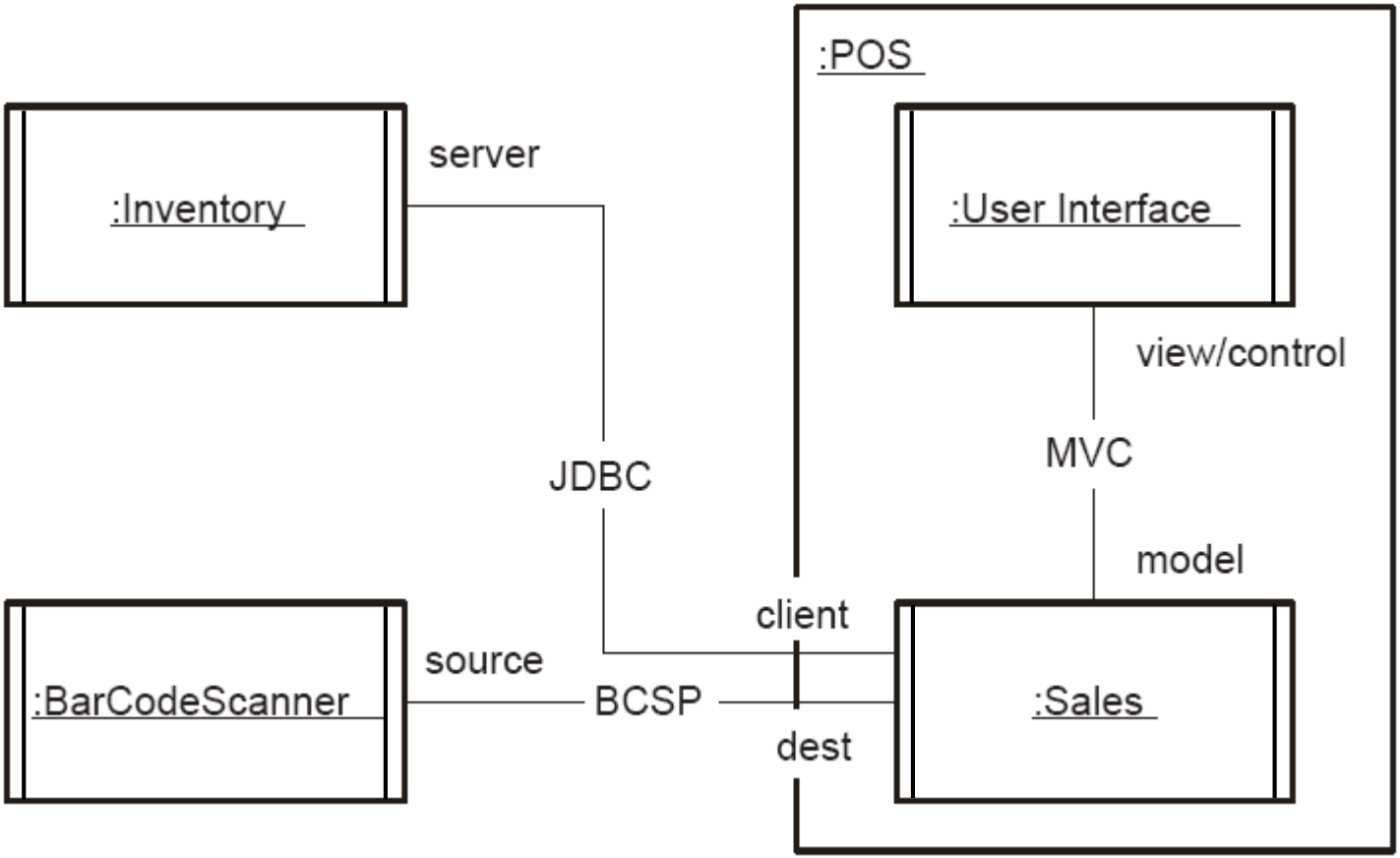
Active Objects

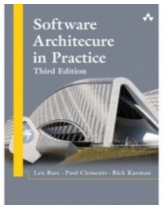


- [Fowler, 2000]

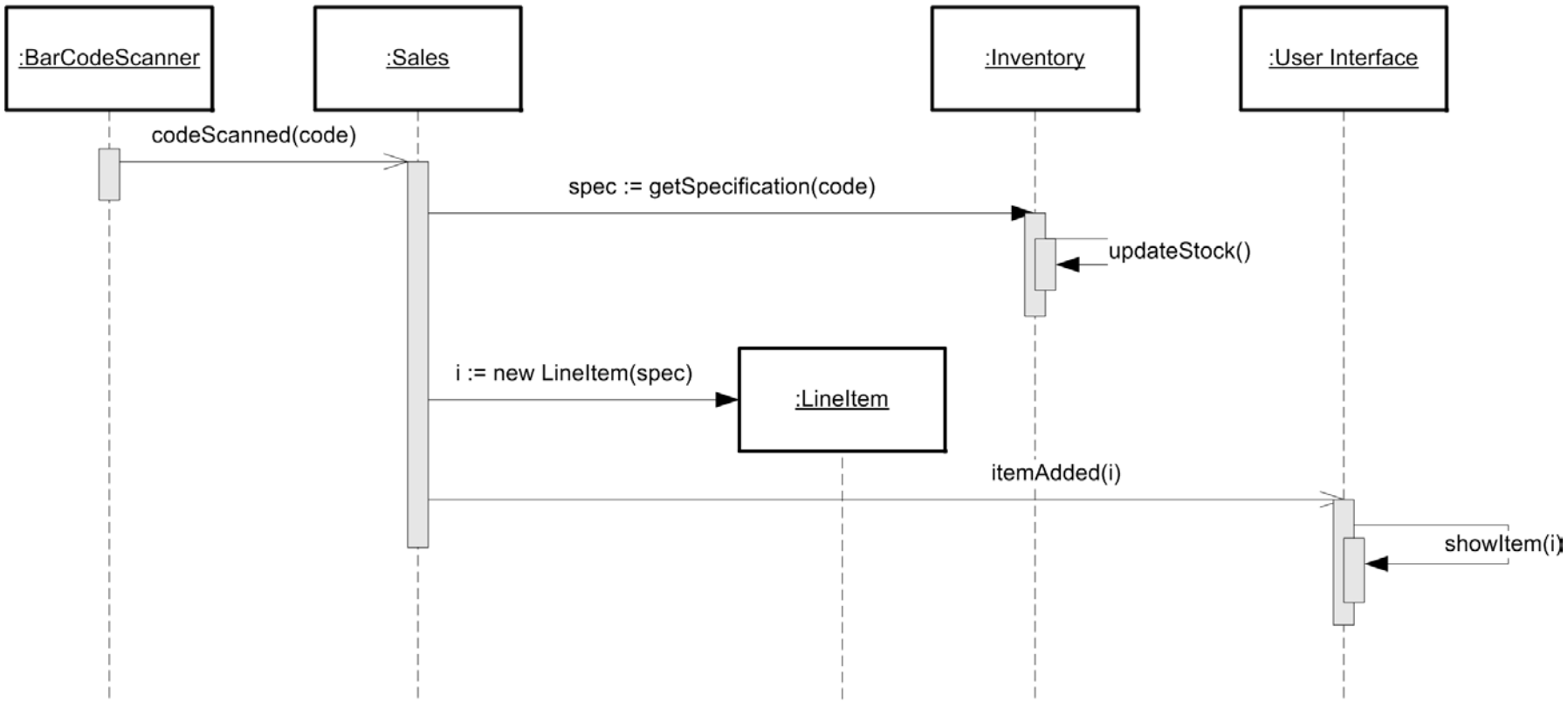


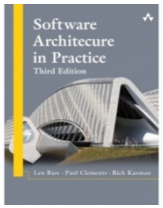
POS Example: C&C View





POS Example: C&C View (2)

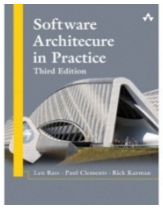




Module Viewpoint

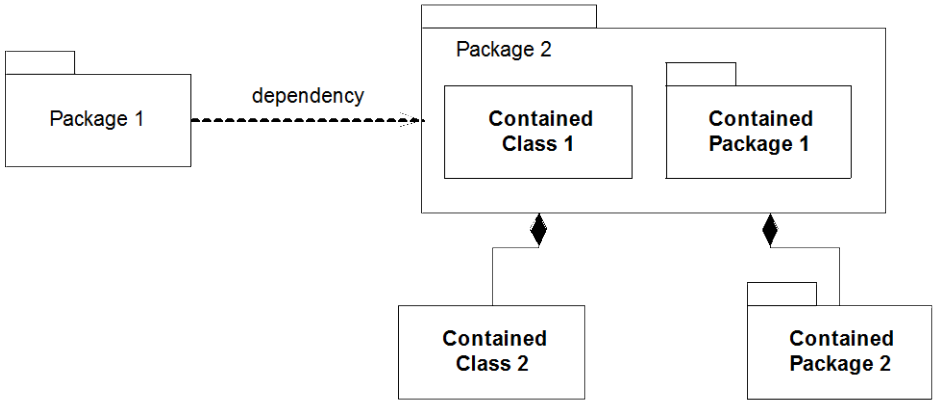
- Elements
  - Classes, packages, interfaces
- Relations
  - Associations, generalizations, realizations, dependencies
- Mapping to UML
  - Class diagrams…

> The software architecture of a computing system is the structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them [Bass et al., 2003]

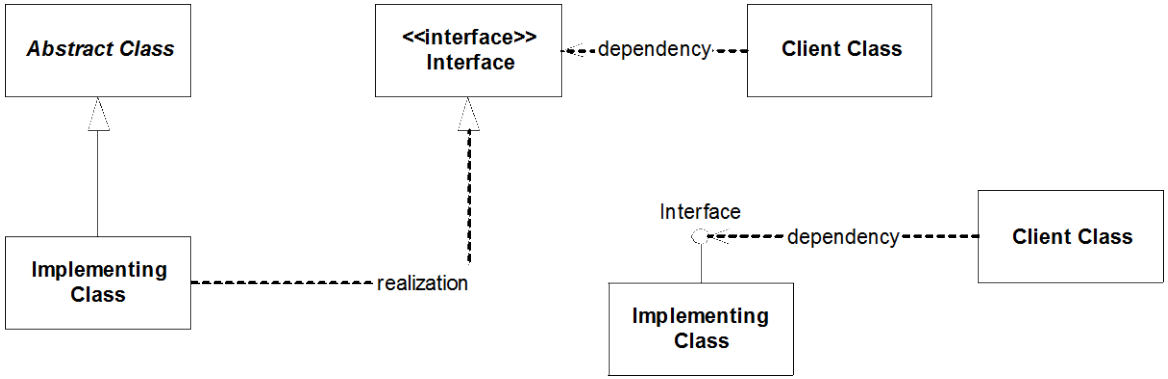


Basic Module Elements (1)

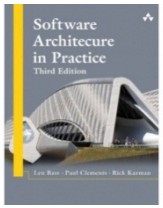
Packages



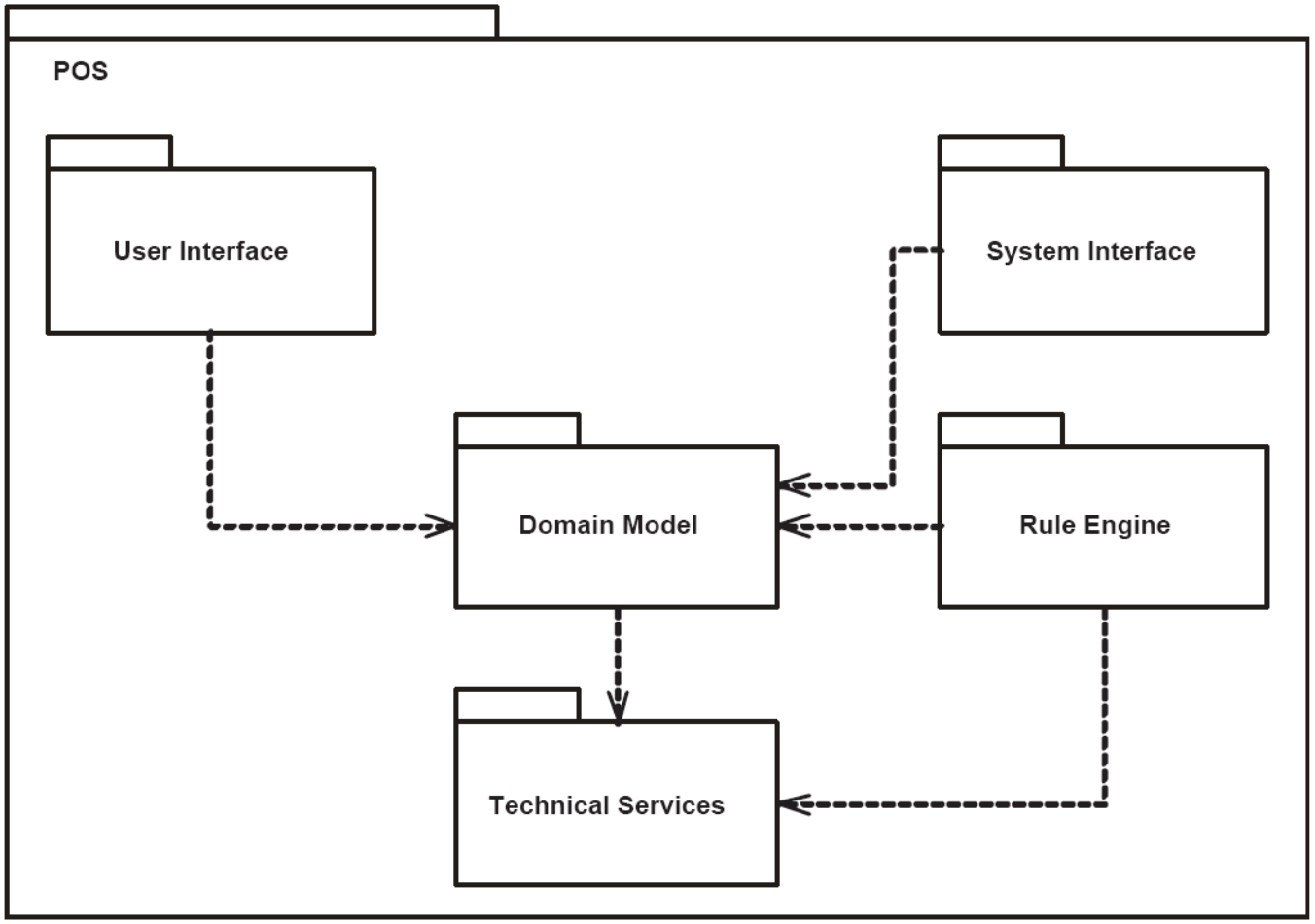
Interfaces

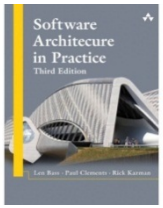


- [Fowler, 2000]

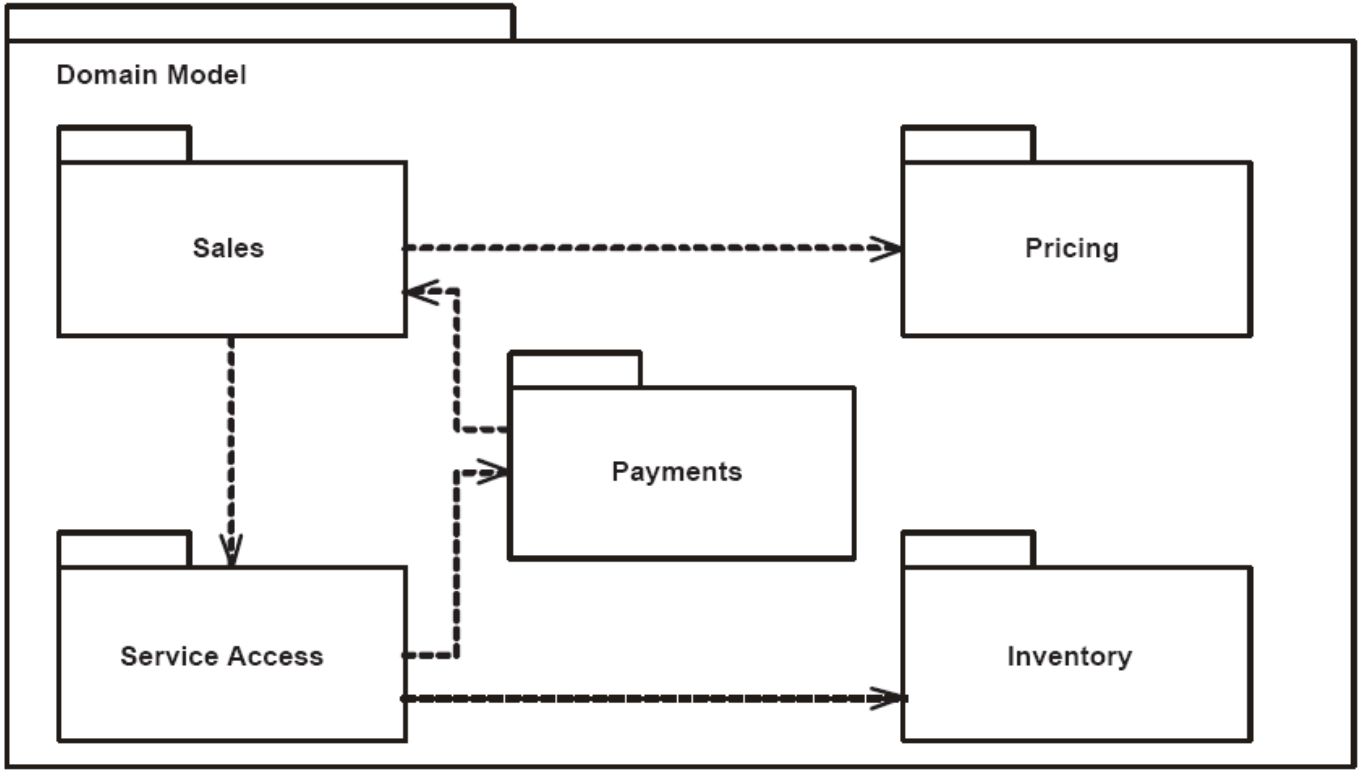


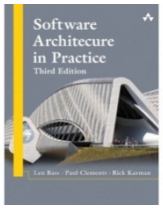
POS Example: Module View (1)



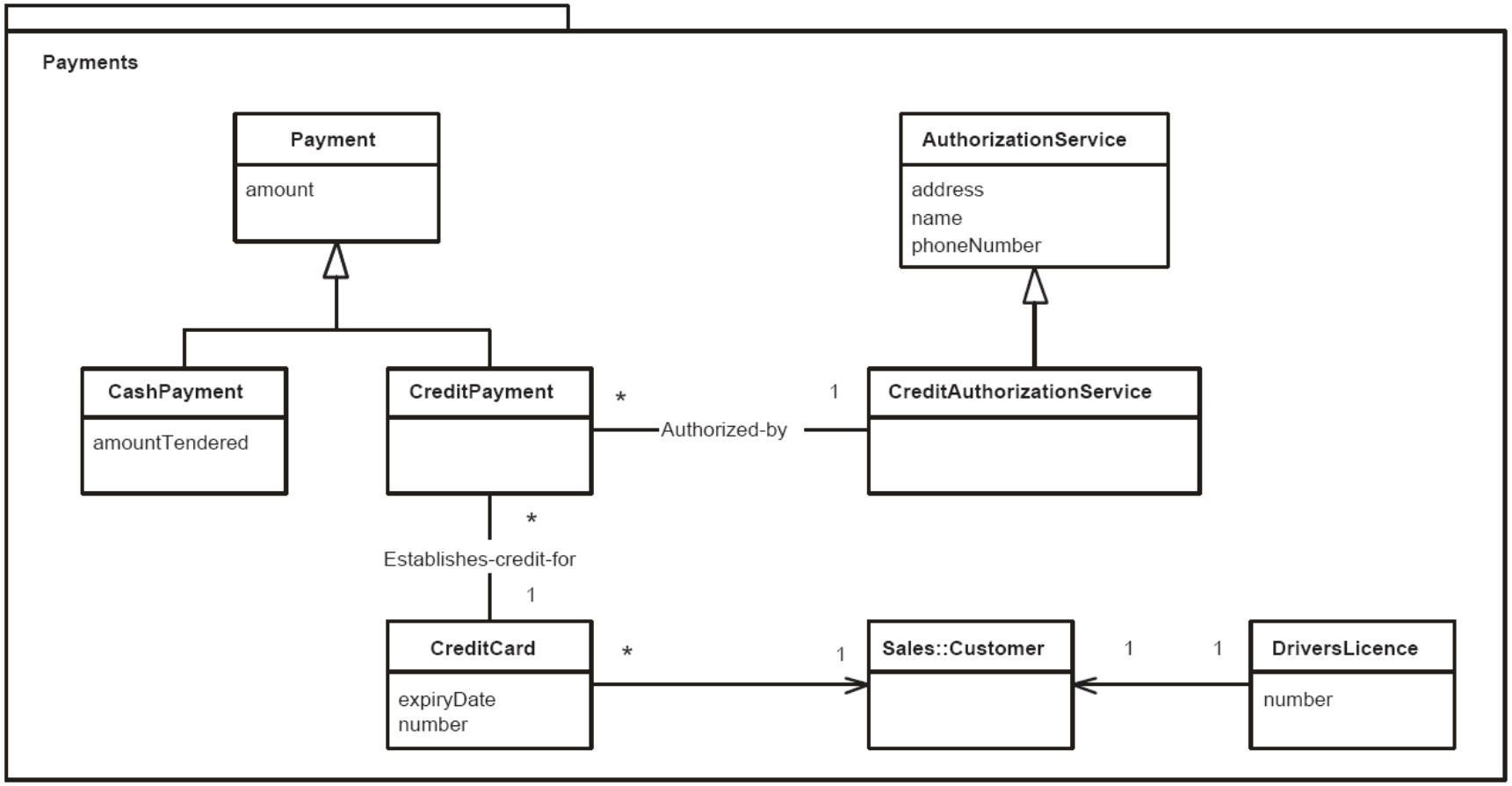


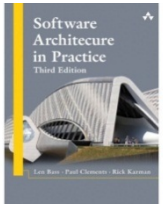
POS Example: Module View (2)





POS Example: Module View (3)





Allocation Viewpoint

Elements

- Software elements: Components, objects
- Environmental elements: Nodes

Relations

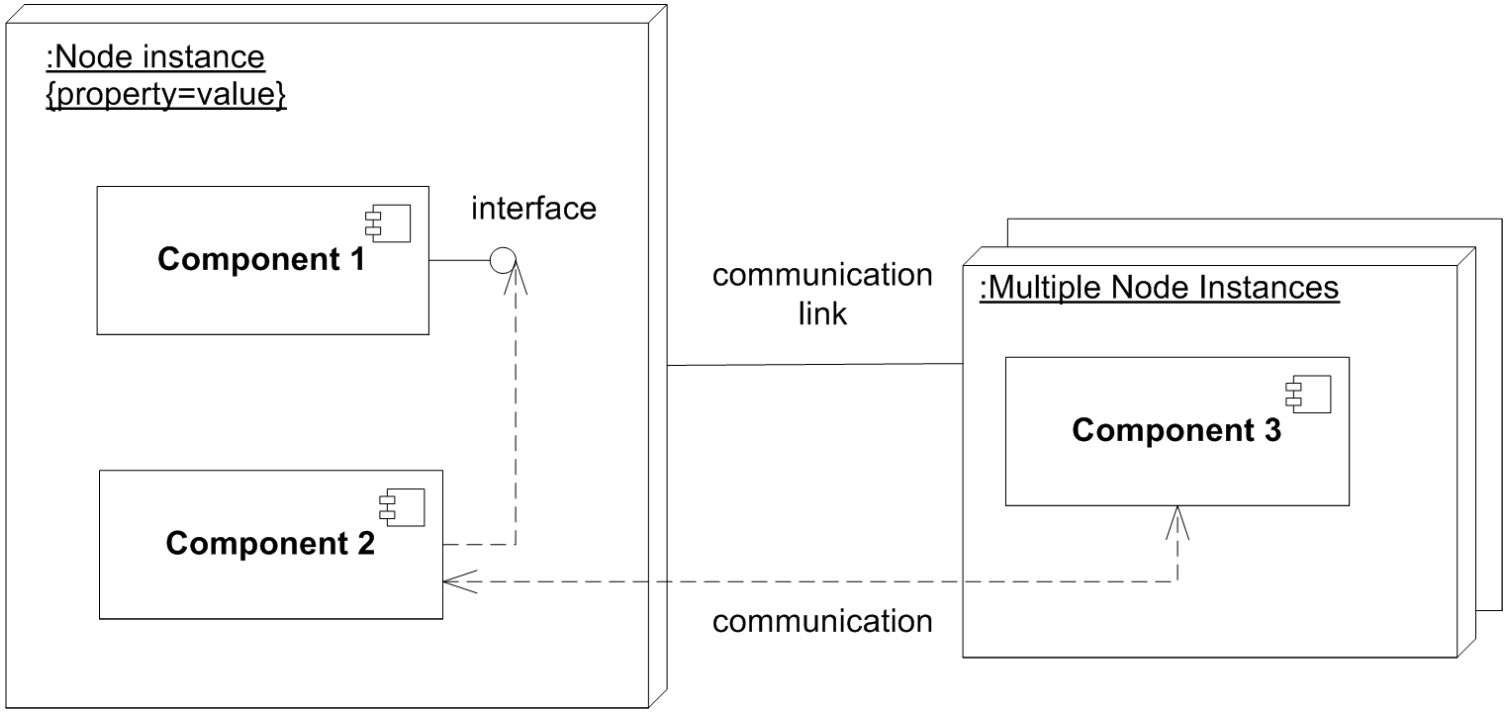
- Allocated-to
- Dependencies
- Connections (communication paths)

Mapping to UML

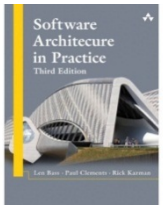
- Here: focus on deployment
  - Deployment and component diagrams

The software architecture of a computing system is the structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them [Bass et al., 2003]

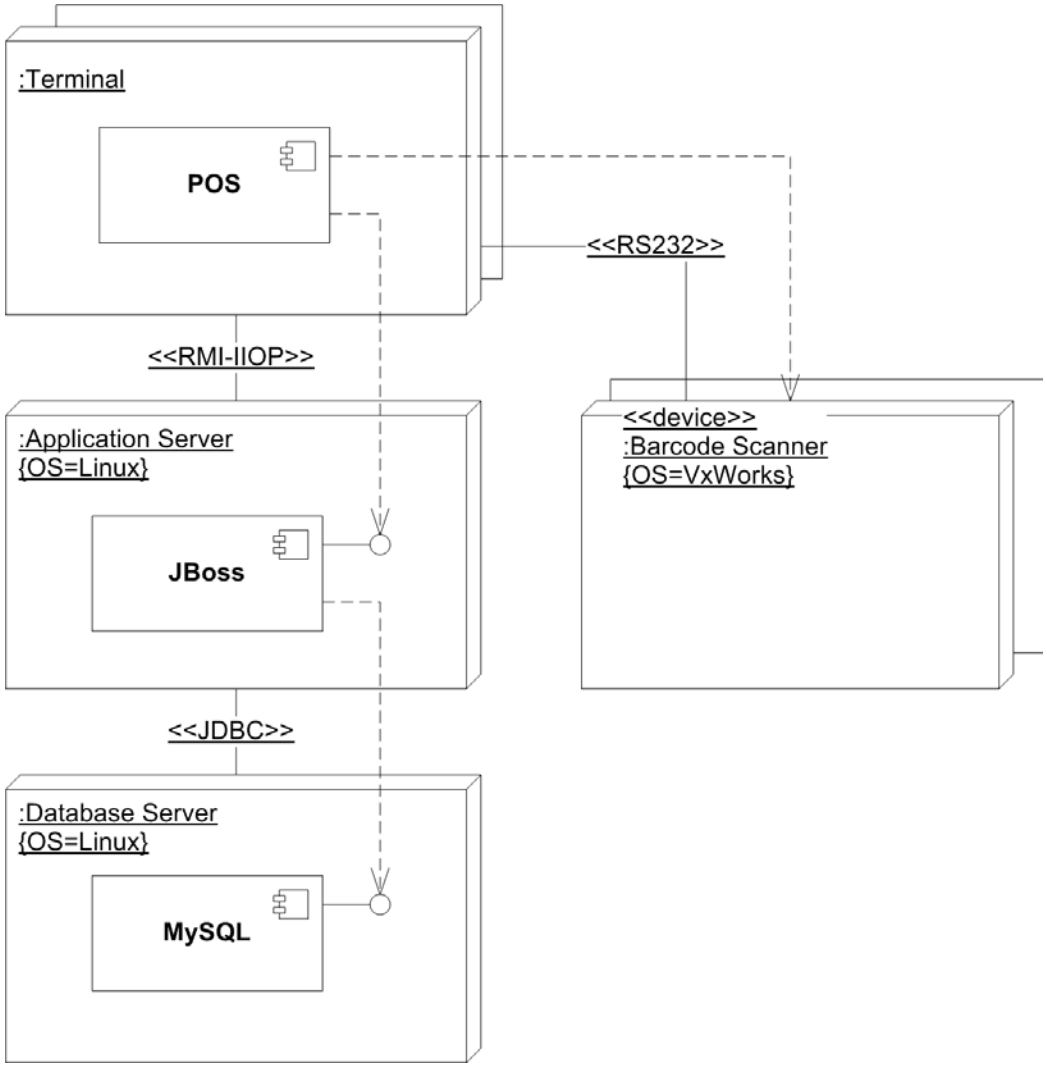
Basic Allocation Elements

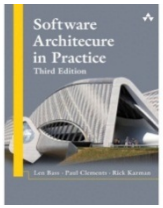


- (Note: *UML* components are used here)
- [Ambler: *Agile modelling]*



POS Example: Allocation View



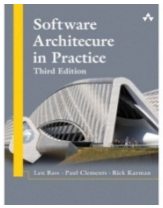


Formal Description Languages

- Architecture-specific [Medvidovic & Taylor, 2000]

ADL	ACME	Aesop	C2	Darwin	MetaH	Rapide	SADL	UniCon	Weaves	Wright
Focus	Architectural interchange, predominantly at the structural level	Specification of architectures in specific styles	Architectures of highly-distributed, evolvable, and dynamic systems	Architectures of highly-distributed systems whose dynamism is guided by strict formal underpinnings	Architectures in the guidance, navigation, and control (GN&C) domain	Modeling and simulation of the dynamic behavior described by an architecture	Formal refinement of architectures across levels of detail	Glue code generation for interconnecting existing components using common interaction protocols	Data-flow architectures, characterized by high-volume of data and real-time requirements on its processing	Modeling and analysis (specifically, deadlock analysis) of the dynamic behavior of concurrent systems

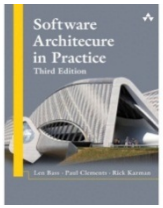
- Other
  - E.g., rate-monotonic analysis on architectural components
  - E.g., statechart-based analyses



Discussion

- Component & Connector viewpoint does not map well to implementation?
  - Few languages have interaction as first-class construct
  - Neither to the UML – bound tightly to OO implementation
  - On the other hand central in many approaches to architectural design
- UML is not designed specifically for software architecture description?
  - Many (irrelevant) modelling constructs
  - But very widely used and supported
- Not precise enough for formal analysis (?)

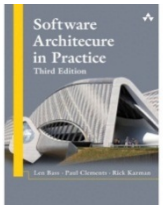
- Take a look at the structures of a systemarchitecture structures of a system.pdf



Architectural Patterns

- Architectural elements can be composed in ways that solve particular problems.
  - The compositions have been found useful over time, and over many different domains
  - They have been documented and disseminated.
  - These compositions of architectural elements, called architectural patterns.
  - Patterns provide packaged strategies for solving some of the problems facing a system.
- An architectural pattern delineates the element types and their forms of interaction used in solving the problem.

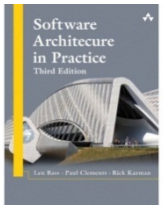
- A common module type pattern is the Layered pattern.
  - When the uses relation among software elements is strictly unidirectional, a system of layers emerges.
  - A layer is a coherent set of related functionality.
  - Many variations of this pattern, lessening the structural restriction, occur in practice.



Architectural Patterns

Common component-and-connector type patterns:

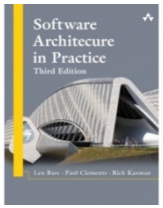
- Shared-data (or repository) pattern.
  - This pattern comprises components and connectors that create, store, and access persistent data.
  - The repository usually takes the form of a (commercial) database.
  - The connectors are protocols for managing the data, such as SQL.
- Client-server pattern.
  - The components are the clients and the servers.
  - The connectors are protocols and messages they share among each other to carry out the system's work.



Architectural Patterns

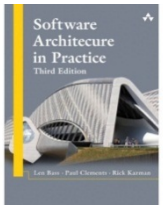
Common allocation patterns:

- Multi-tier pattern
  - Describes how to distribute and allocate the components of a system in distinct subsets of hardware and software, connected by some communication medium.
  - This pattern specializes the generic deployment (software-to-hardware allocation) structure.
- Competence center pattern and platform pattern
  - These patterns specialize a software system's work assignment structure.
  - In competence center, work is allocated to sites depending on the technical or domain expertise located at a site.
  - In platform, one site is tasked with developing reusable core assets of a software product line, and other sites develop applications that use the core assets.



Summary

- The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.
- A structure is a set of elements and the relations among them.
- A view is a representation of a coherent set of architectural elements. A view is a representation of one or more structures.



Summary

- There are three categories of structures:
  - Module structures show how a system is to be structured as a set of code or data units that have to be constructed or procured.
  - Component-and-connector structures show how the system is to be structured as a set of elements that have runtime behavior (components) and interactions (connectors).
  - Allocation structures show how the system will relate to nonsoftware structures in its environment (such as CPUs, file systems, networks, development teams, etc.).
- Structures represent the primary engineering leverage points of an architecture.
- Every system has a software architecture, but this architecture may be documented and disseminated, or it may not be.
- There is no such thing as an inherently good or bad architecture. Architectures are either more or less fit for some purpose.