



Module 1

Examining Object-Oriented Concepts and Terminology



Objectives

Upon completion of this module, you should be able to:

- Describe the important object-oriented (OO) concepts
- Describe the fundamental OO terminology



Examining Object Orientation

OO concepts affect the whole development process:

- Humans think in terms of nouns (objects) and verbs (behaviors of objects).
- With OOSD, both problem and solution domains are modeled using OO concepts.
- The *Unified Modeling Language* (UML) is a de facto standard for modeling OO software.
- OO languages bring the implementation closer to the language of mental models. The UML is a good bridge between mental models and implementation.



Examining Object Orientation

“Software systems perform certain actions on objects of certain types; to obtain flexible and reusable systems, it is better to base their structure on the objects types than on the actions.” (Meyer page vi)

OO concepts affect the following issues:

- Software complexity
- Software decomposition
- Software costs



Software Complexity

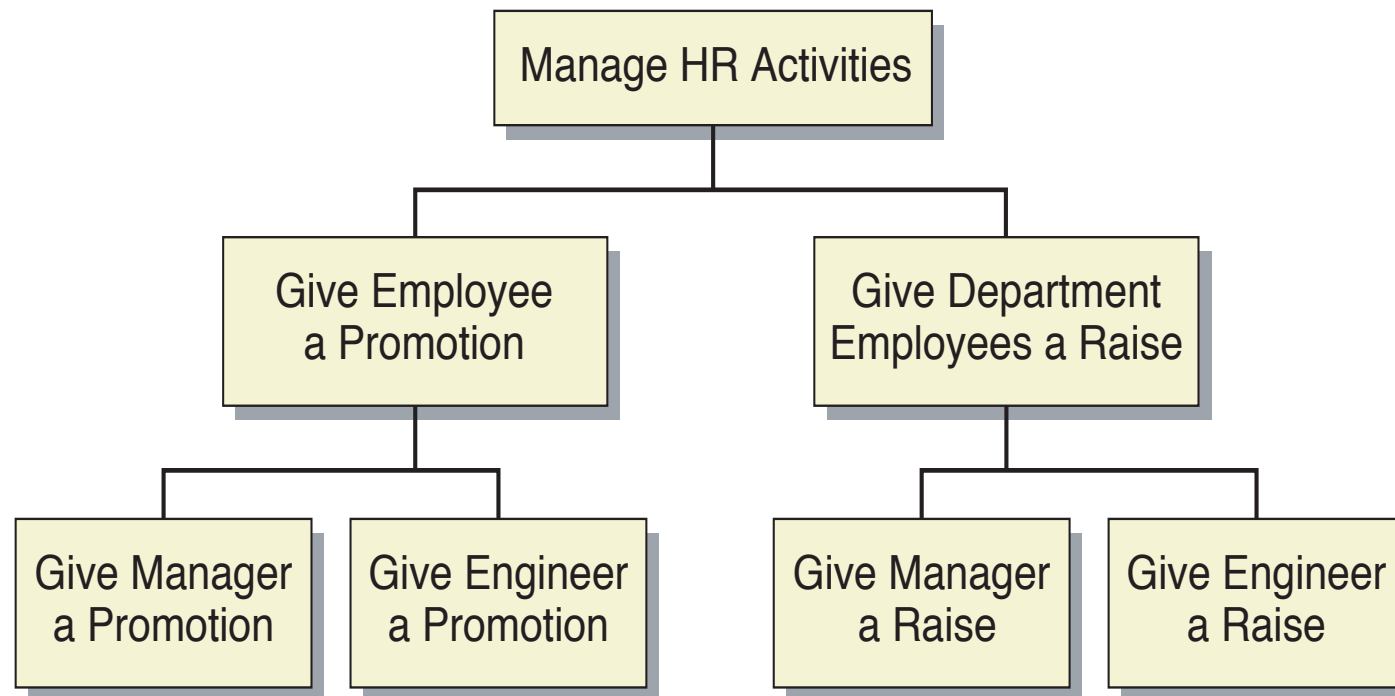
Complex systems have the following characteristics:

- They have a *hierarchical structure*.
- The choice of *which components are primitive* in the system is arbitrary.
- A system can be split by intra- and inter-component relationships. This *separation of concerns* enables you to study each part in relative isolation.
- Complex systems are usually composed of only a *few types of components in various combinations*.
- A successful, complex system invariably *evolves from a simple working system*.



Software Decomposition

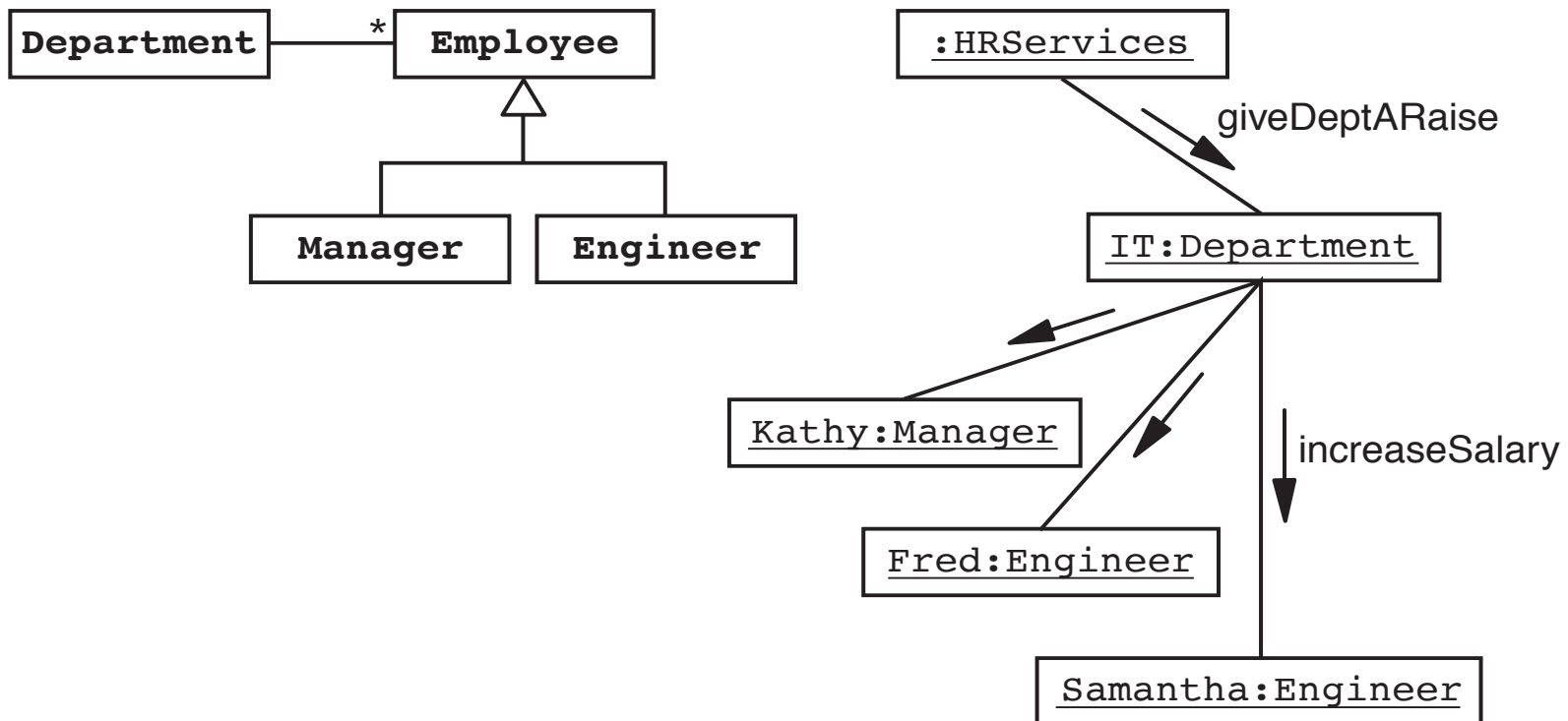
In the Procedural paradigm, software is decomposed into a hierarchy of procedures or tasks.





Software Decomposition

In the OO paradigm, software is decomposed into a hierarchy of interacting components (usually objects).





Software Costs

Development:

- OO principles provide a natural technique for modeling business entities and processes from the early stages of a project.
- OO-modeled business entities and processes are easier to implement in an OO language.

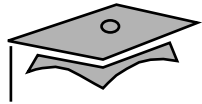
Maintenance:

- Changeability, flexibility, and adaptability of software is important to keep software running for a long time.
- OO-modeled business entities and processes can be adapted to new functional requirements.



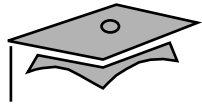
Comparing the Procedural and OO Paradigms

	Procedural Paradigm	OO Paradigm
Organizational structure	<p>Focuses on hierarchy of procedures and subprocedures</p> <p>Data is separate from procedures</p>	<p>Network of collaborating objects</p> <p>Methods (processes) are often bound together with the state (data) of the object</p>
Protection against modification or access	<p>Data is difficult to protect against inappropriate modifications or access when it is passed to or referenced by many different procedures.</p>	<p>The data and internal methods of objects can be protected against inappropriate modifications or access by using encapsulation.</p>



Comparing the Procedural and OO Paradigms

	Procedural Paradigm	OO Paradigm
Ability to modify software	Can be expensive and difficult to make software that is easy to change, resulting in many “Brittle” systems	Robust software that is easy to change, if written using good OO principles and patterns
Reuse	Reuse of methods is often achieved by copy-and-paste or 1001 parameters.	Reuse of code by using generic components (one or more objects) with well-defined interfaces. This is achieved by extension of classes (or interfaces) or by composition of objects.



Comparing the Procedural and OO Paradigms

	Procedural Paradigm	OO Paradigm
Configuration of special cases	Often requires if or switch statements. Modification is risky because it often requires altering existing code. So, modifications must be done with extreme care apart from requiring extensive regression testing. These factors make even minor changes costly to implement.	Polymorphic behavior can facilitate the possibility of modifications being primarily additive, subtractive, or substitution of whole components (one or more objects); thereby, reducing the associated risks and costs.



Surveying the Fundamental OO Concepts

- Objects
- Classes
- Abstraction
- Encapsulation
- Inheritance
- Interfaces
- Polymorphism
- Cohesion
- Coupling
- Class associations and object links
- Delegation



Objects

object = state + behavior

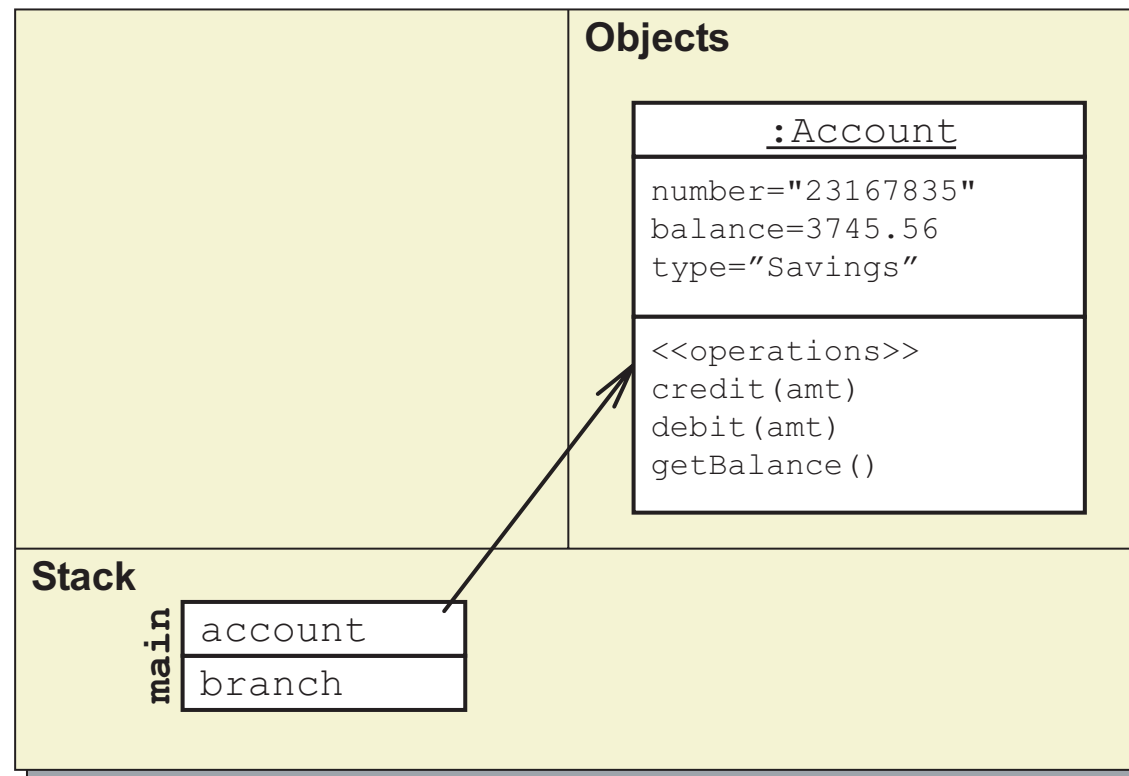
“An object has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class.”
(Booch Object Solutions page 305)

Objects:

- Have identity
- Are an instance of only one class
- Have attribute values that are unique to that object
- Have methods that are common to the class



Objects: Example





Classes

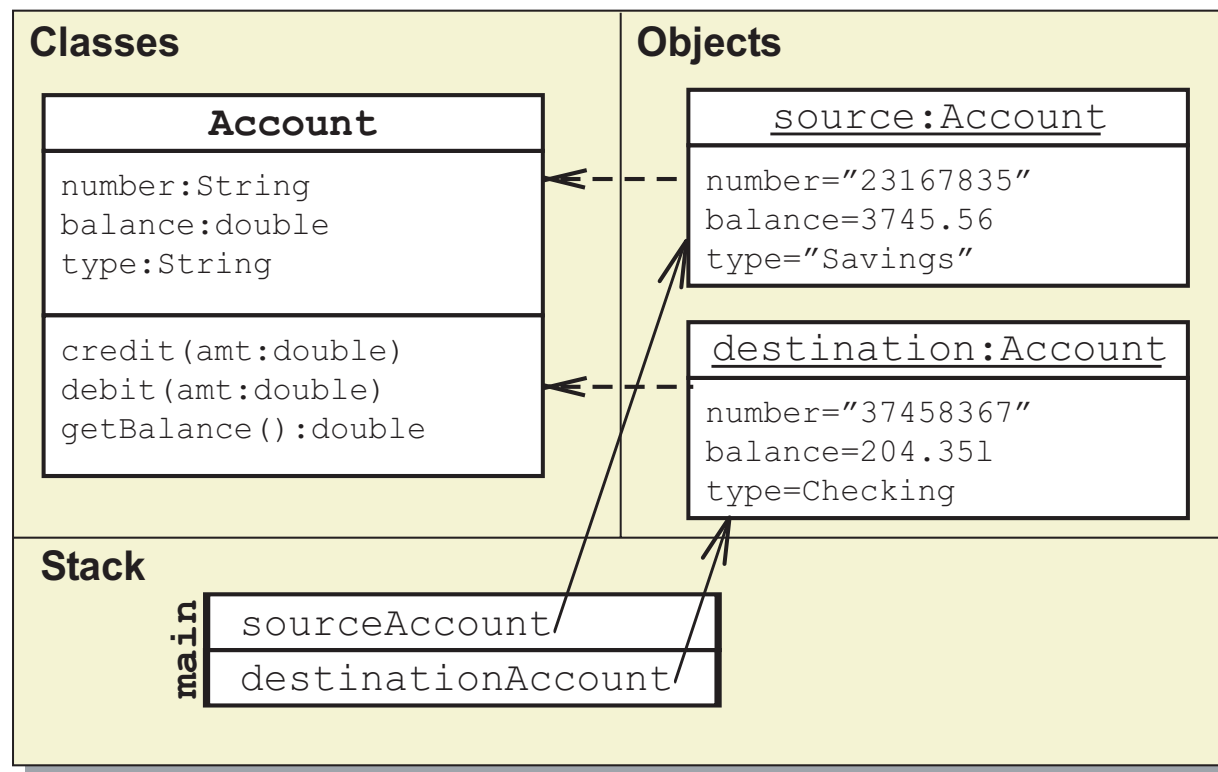
A class is a blueprint or prototype from which objects are created. (The Java™ Tutorials)

Classes provide:

- The metadata for attributes
- The signature for methods
- The implementation of the methods (usually)
- The constructors to initialize attributes at creation time



Classes: Example





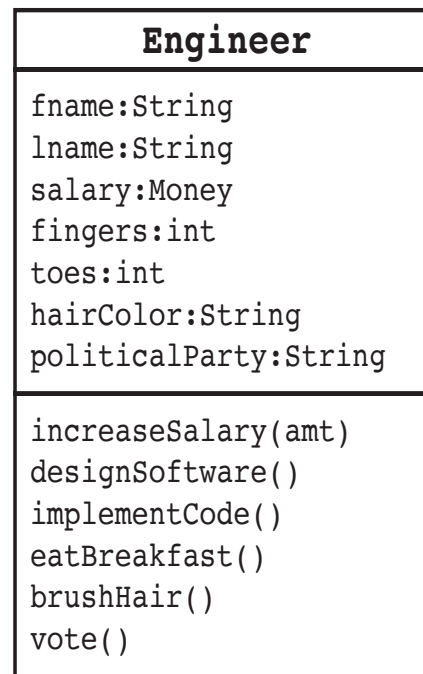
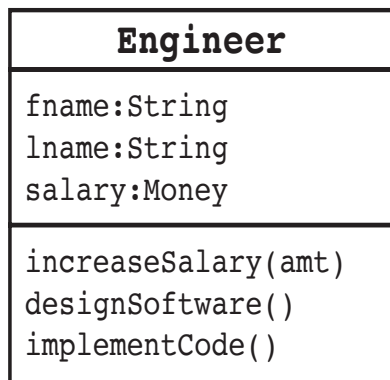
Abstraction

In OO software, the concept of abstraction enables you to create a simplified, but relevant view of a real world object within the context of the problem and solution domains.

- The abstraction object is a representation of the real world object with irrelevant (within the context of the system) behavior and data removed.
- The abstraction object is a representation of the real world object with currently irrelevant (within the context of the view) behavior and data hidden.



Abstraction: Example





Encapsulation

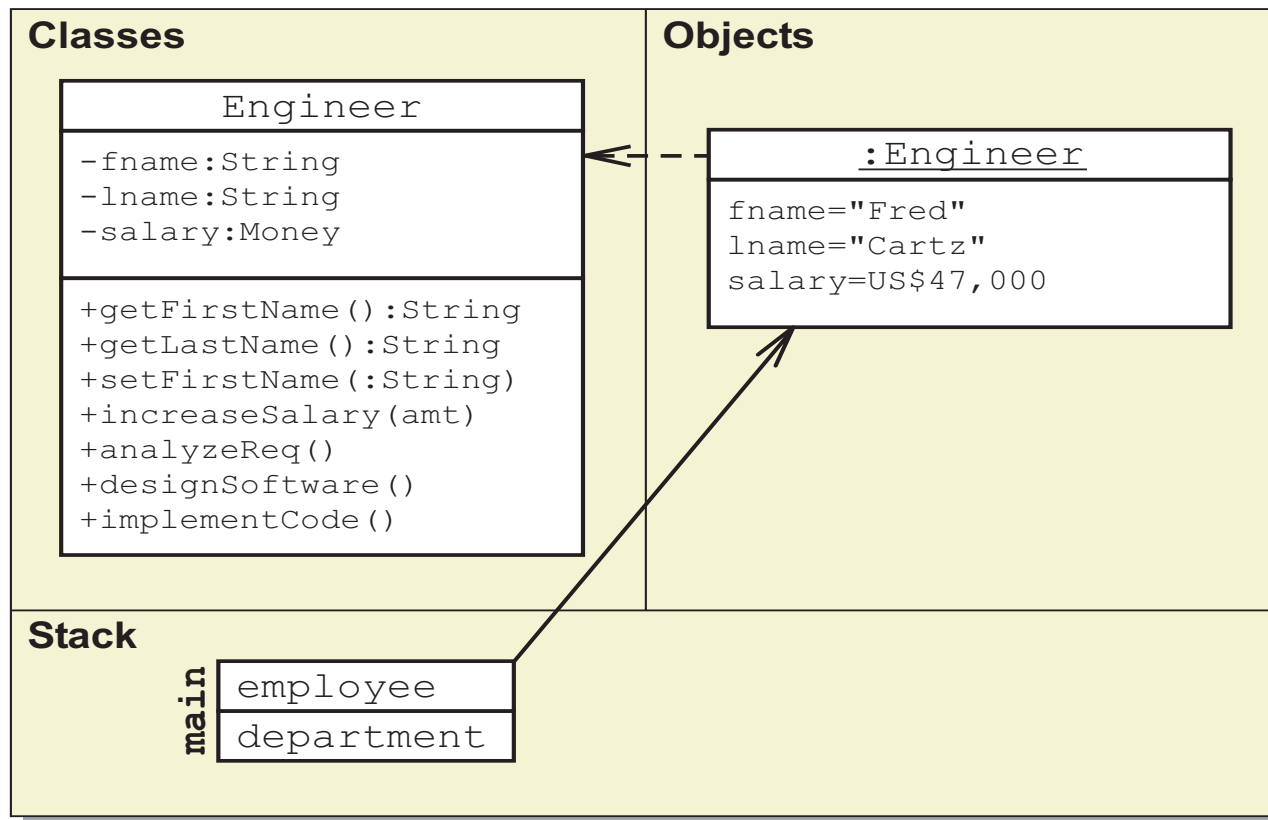
Encapsulation means “to enclose in or as if in a capsule”
(Webster New Collegiate Dictionary)

Encapsulation is essential to an object. An object is a capsule that holds the object’s internal state within its boundary.

In most OO languages, the term encapsulation also includes *information hiding*, which can be defined as: “hide implementation details behind a set of non-private methods”.



Encapsulation: Example



✗ `name = employee.fname;`

✗ `employee.fname = "Samantha";`

✓ `name = employee.getFirstName();`

✓ `employee.setFirstName("Samantha");`



Inheritance

Inheritance is “a mechanism whereby a class is defined in reference to others, adding all their features to its own.” (Meyer page 1197)

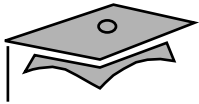
Features of inheritance:

- Attributes and methods from the superclass are included in the subclass.
- Subclass methods can override superclass methods.
- The following conditions must be true for the inheritance relationship to be plausible:
 - A subclass object *is a (is a kind of)* the superclass object.
 - Inheritance should conform to Liskov’s Substitution Principle (LSP).

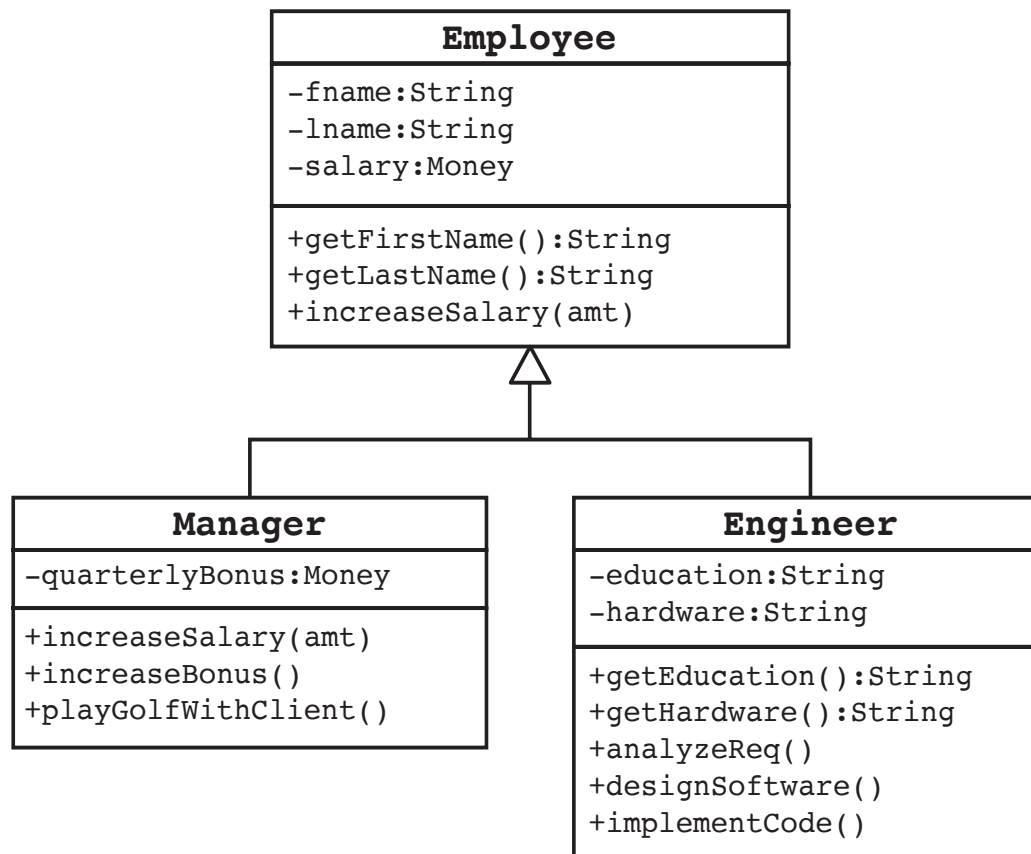


Inheritance

- Specific OO languages allow either of the following:
 - Single inheritance, which allows a class to directly inherit from only one superclass (for example, Java).
 - Multiple inheritance, which allows a class to directly inherit from one or more superclasses (for example, C++).



Inheritance: Example





Abstract Classes

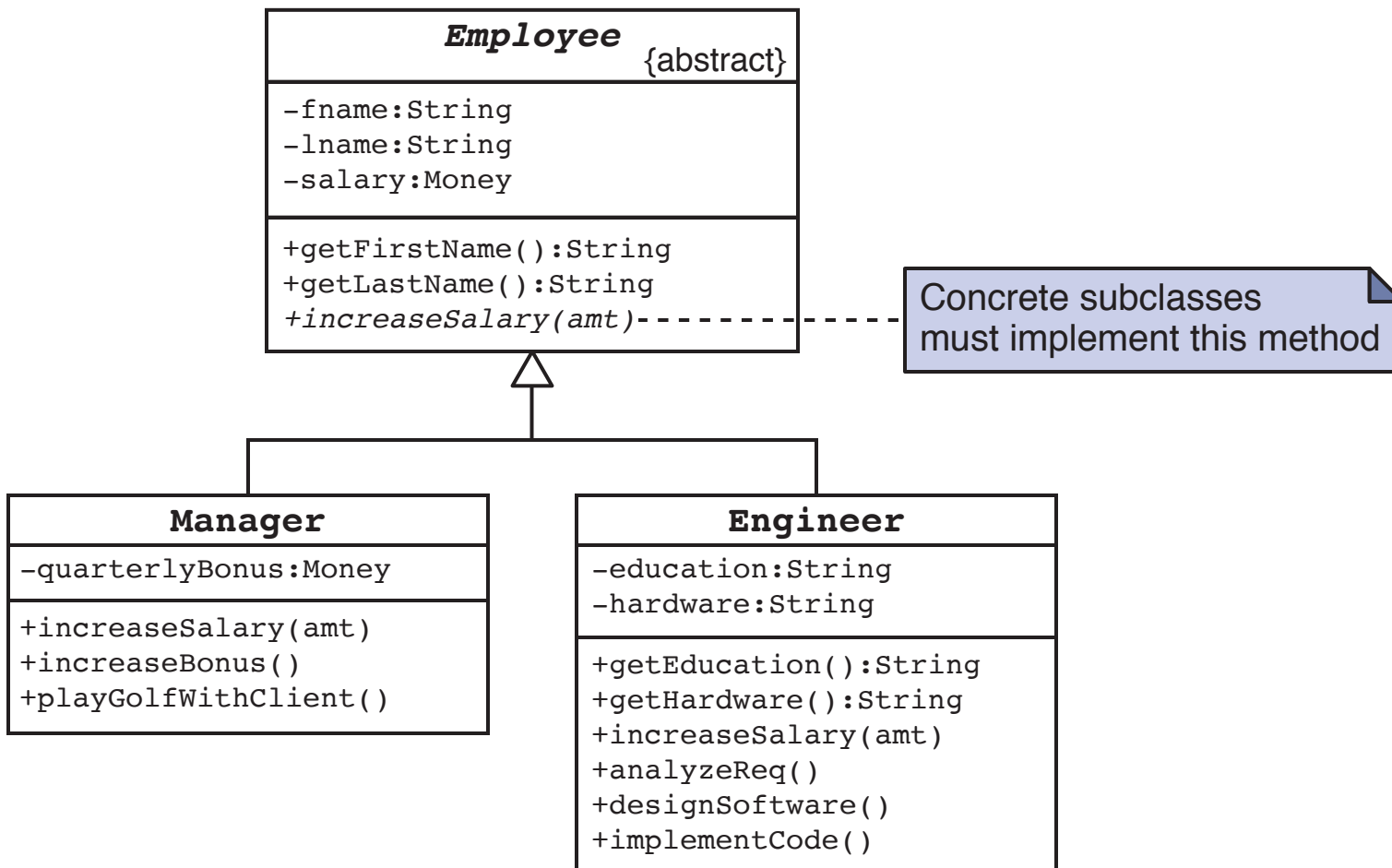
A class that contains one or more abstract methods, and therefore can never be instantiated. (Sun Glossary)

Features of an abstract class:

- Attributes are permitted.
- Methods are permitted and some might be declared abstract.
- Constructors are permitted, but no client may directly instantiate an abstract class.
- Subclasses of abstract classes must provide implementations of all abstract methods; otherwise, the subclass must also be declared abstract.
- In the UML, a method or a class is denoted as abstract by using italics, or by appending the method name or class name with `{abstract}`.



Abstract Classes: Example





Interfaces

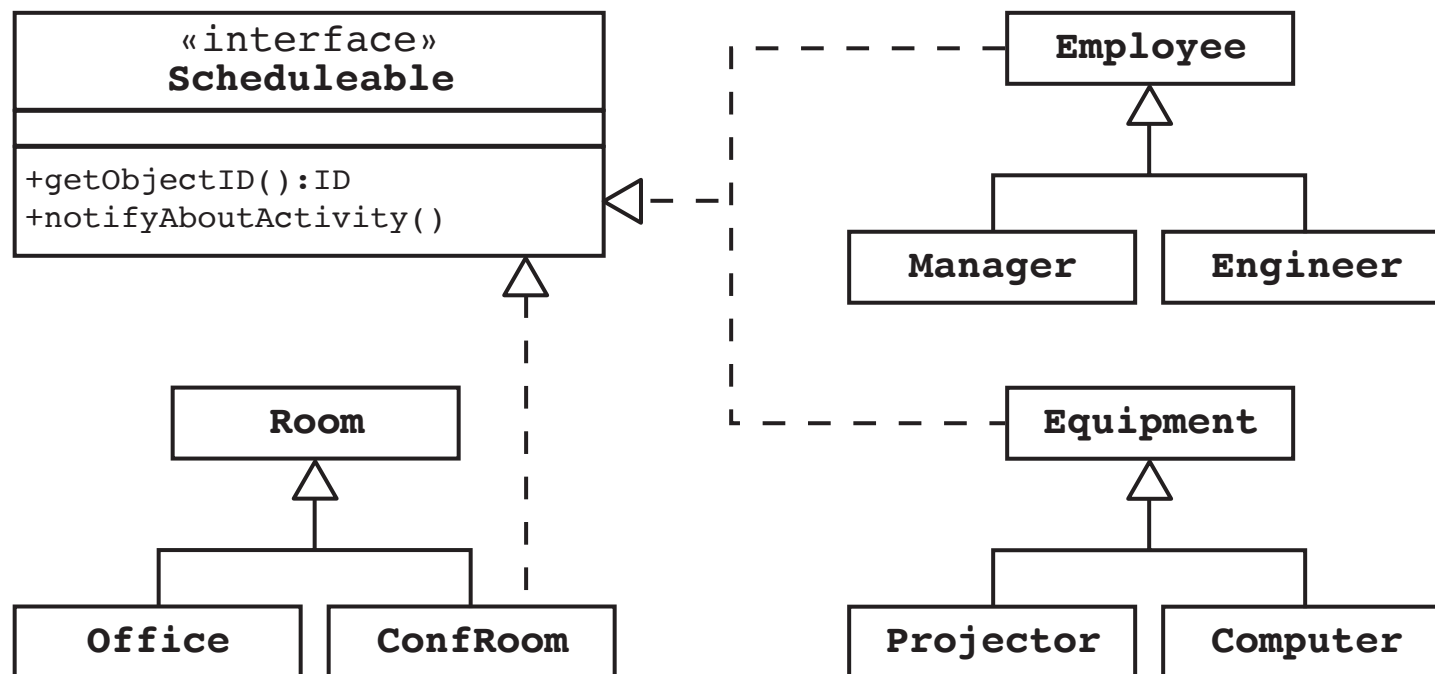
Features of Java technology interfaces:

- Attributes are not permitted (except constants).
- Methods are permitted, but they must be abstract.
- Constructors are not permitted.
- Subinterfaces may be defined, forming an inheritance hierarchy of interfaces.

A class may implement one or more interfaces.



Interfaces: Example





Polymorphism

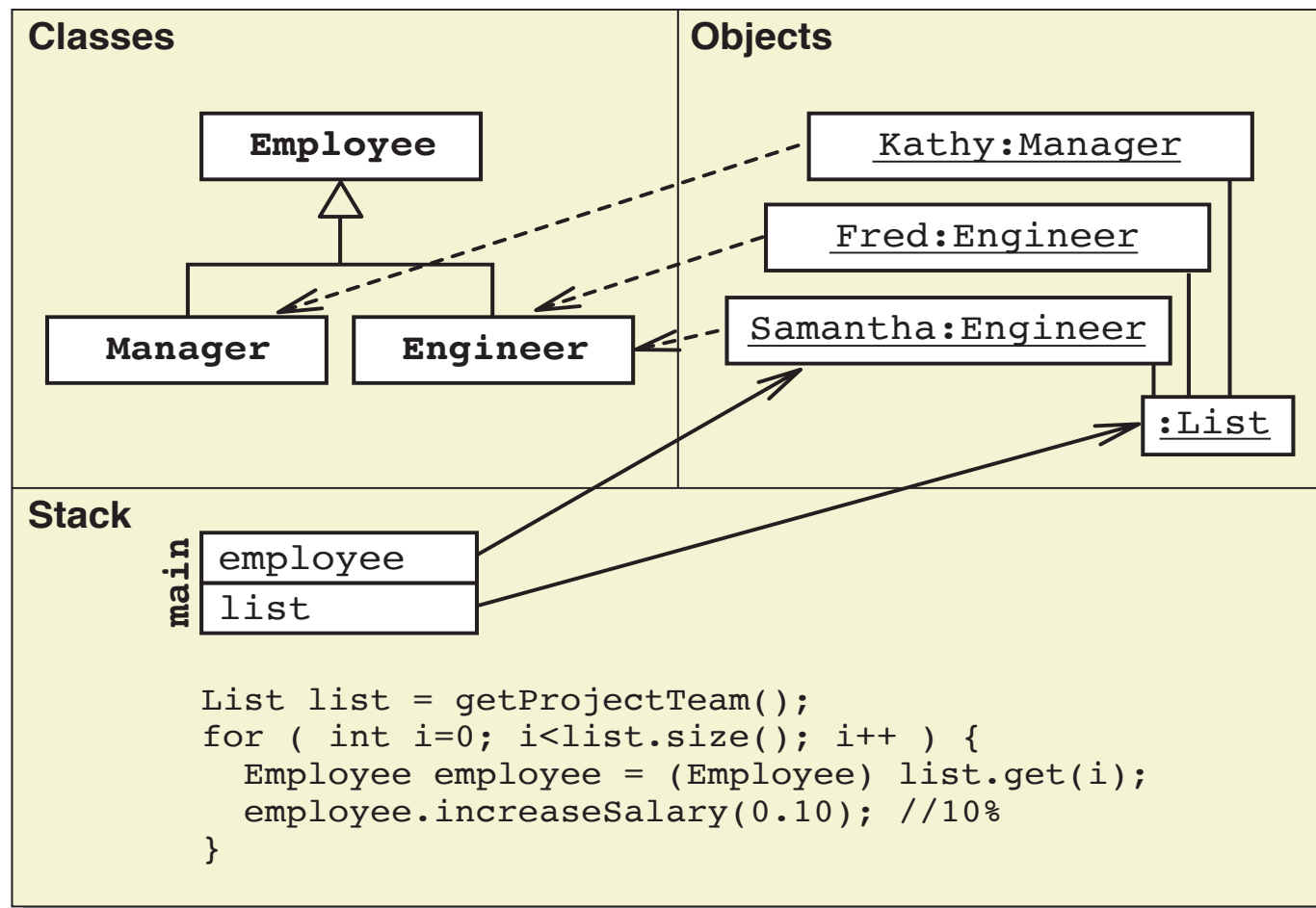
Polymorphism is “a concept in type theory, according to which a name (such as a variable declaration) may denote objects of many different classes that are related by some common superclass [type].” (Booch OOAD page 517)

Aspects of polymorphism:

- A variable can be assigned different types of objects at runtime provided they are a subtype of the variable's type.
- Method implementation is determined by the type of object, not the type of the declaration (dynamic binding).
- Only method signatures defined by the variable type can be called without casting.

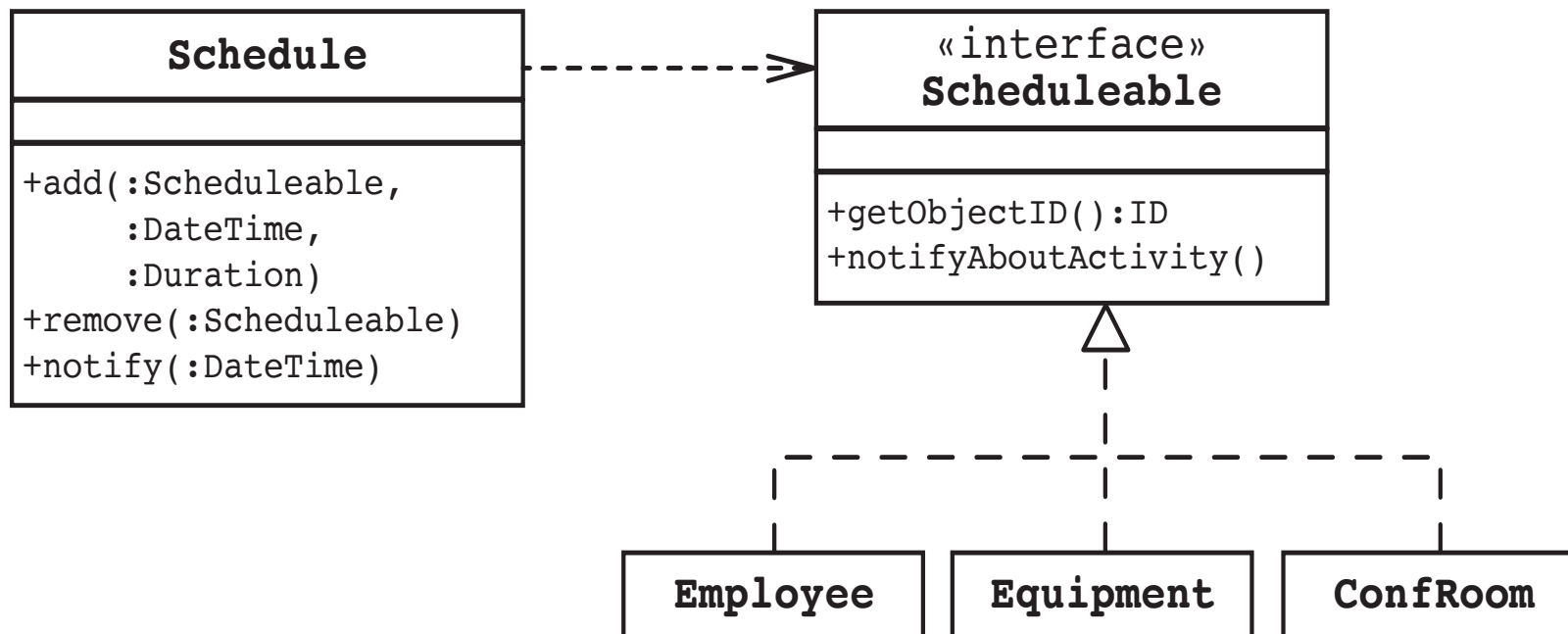


Polymorphism: Example





Polymorphism: Example





Cohesion

In software, the concept of cohesion refers to how well a given component or method supports a single purpose.

- Low cohesion occurs when a component is responsible for many unrelated features.
- High cohesion occurs when a component is responsible for only one set of related features.
- A component includes one or more classes. Therefore, cohesion applies to a class, a subsystem, and a system.
- Cohesion also applies to other aspects including methods and packages.
- Components that do everything are often described with the Anti-Pattern term of Blob components.



Cohesion: Example

Low Cohesion

SystemServices
makeEmployee makeDepartment login logout deleteEmployee deleteDepartment retrieveEmpByName retrieveDeptByID

High Cohesion

LoginService
login logout

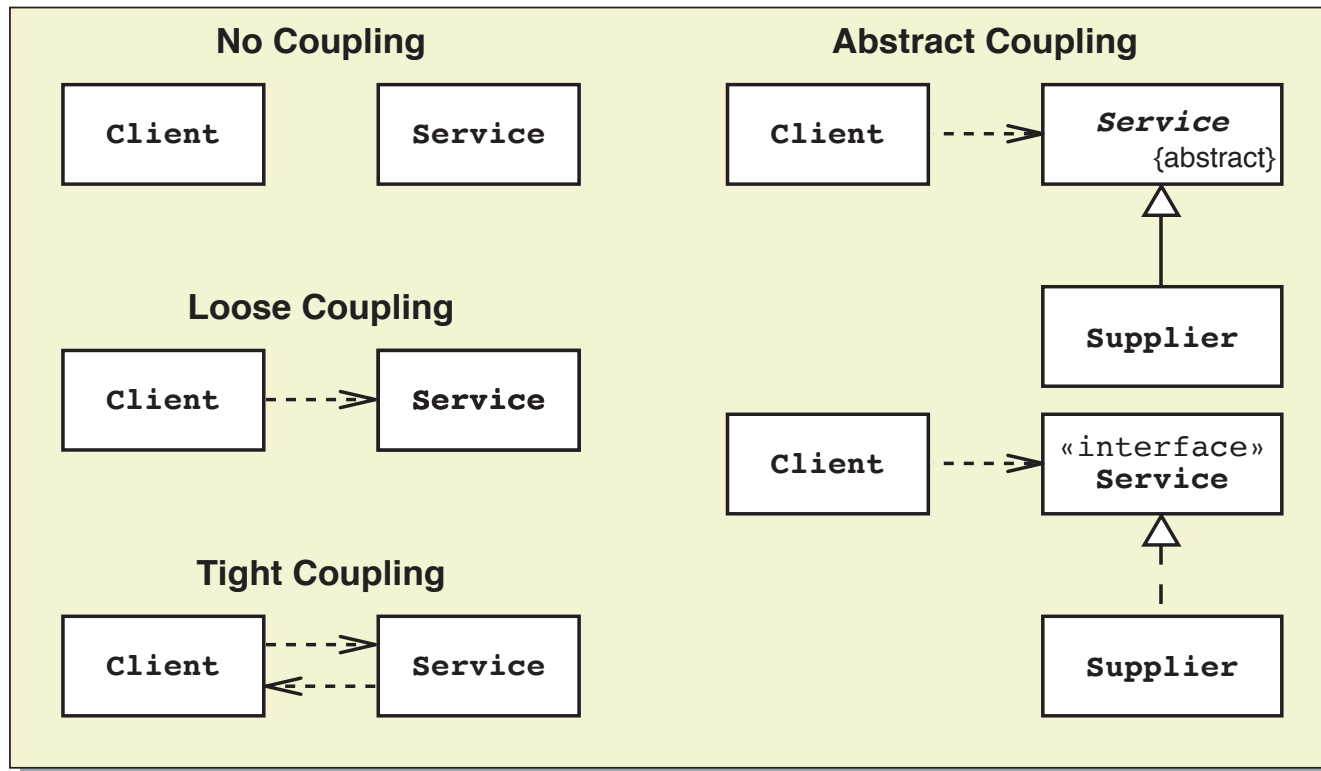
EmployeeService
makeEmployee deleteEmployee retrieveEmpByName

DepartmentService
makeDepartment deleteDepartment retrieveDeptByID



Coupling

Coupling is “the degree to which classes within our system are dependent on each other.” (Knoernschild page 174)





Class Associations and Object Links

Dimensions of associations include:

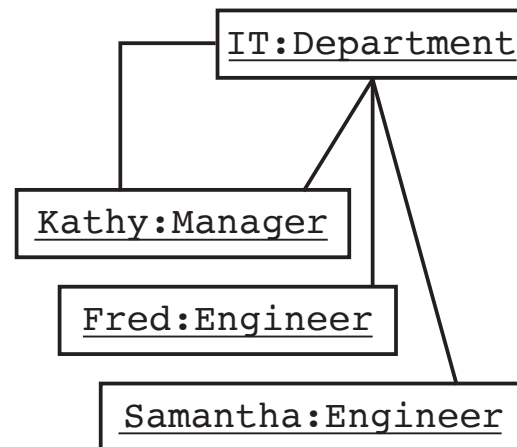
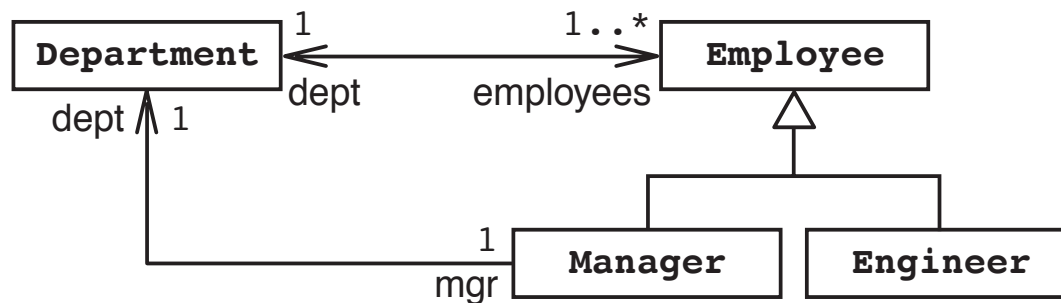
- The roles that each class plays
- The multiplicity of each role
 - 1 denotes exactly one
 - 1..* denotes one or more
 - 0..* or * denotes zero or more
- The direction (or navigability) of the association

Object links:

- Are instances of the class association
- Are one-to-one relationships



Class Associations and Object Links: Example





Delegation

Many computing problems can be easily solved by delegation to a more cohesive component (one or more classes) or method.

- Delegation is similar to how we humans behave.
 - A manager often delegates tasks to an employee with the appropriate skills.
 - You often delegate plumbing problems to a plumber.
 - A car delegates accelerate, brake, and steer messages to its subcomponents, who in turn delegate messages to their subcomponents. This delegation of messages eventually affects the engine, brakes, and wheel direction respectively.
- OO paradigm frequently mimics the real world.

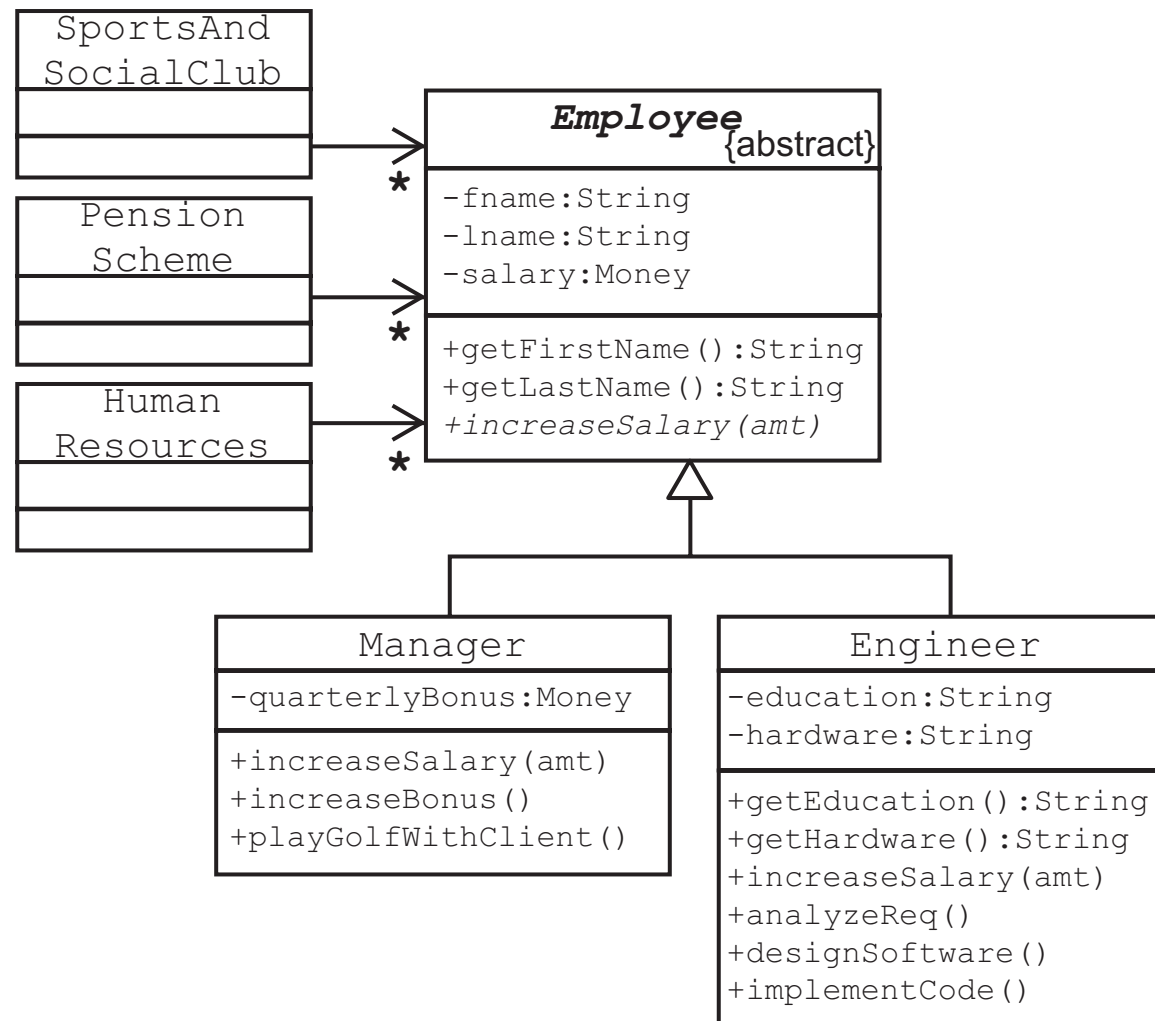


Delegation

- The ways you delegate in OO paradigm include delegating to:
 - A more cohesive linked object
 - A collection of cohesive linked objects
 - A method in a subclass
 - A method in a superclass
 - A method in the same class

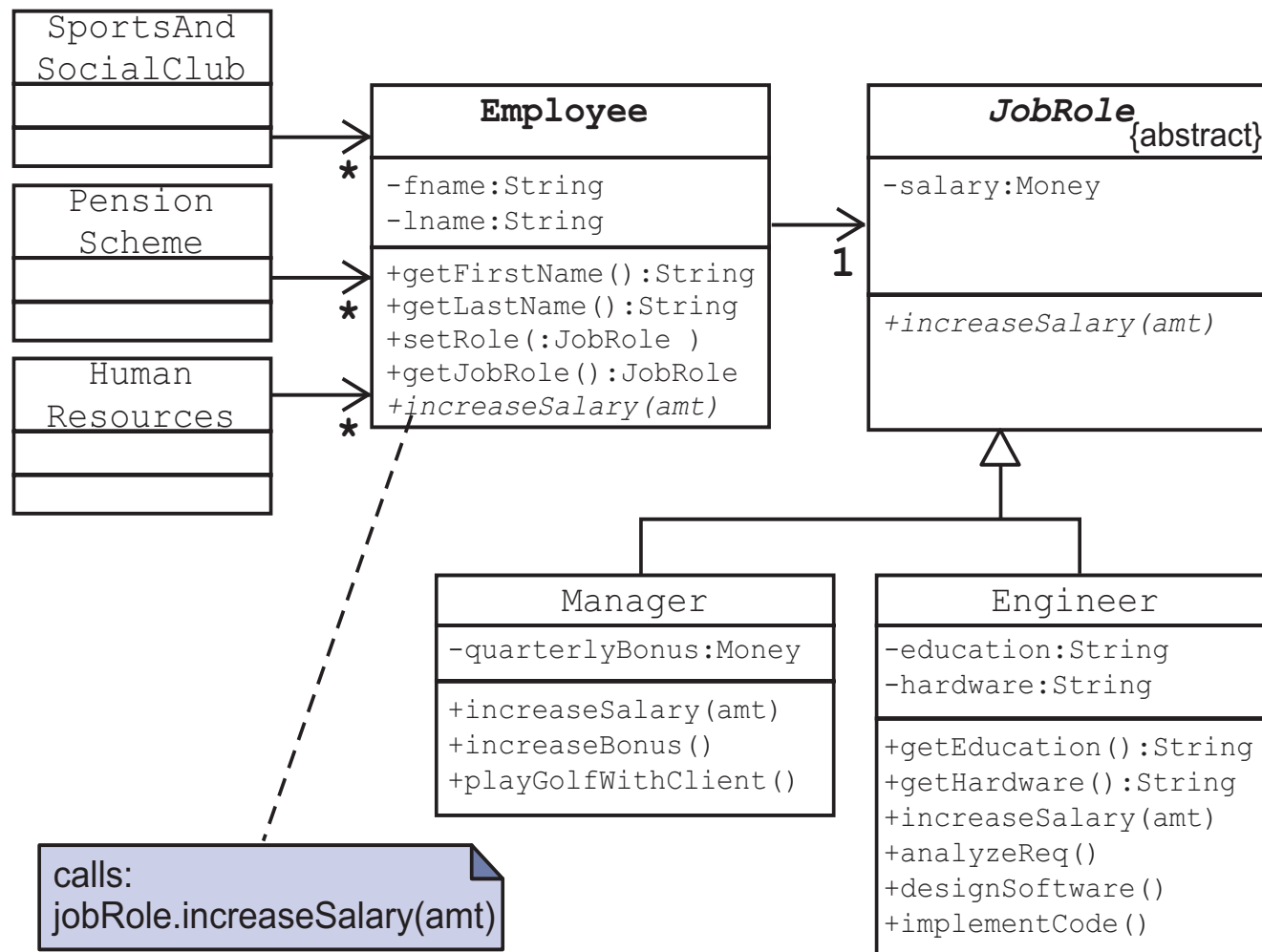


Delegation: Example Problem





Delegation: Example Solution





Summary

- Object orientation is a model of computation that is closer to how humans think about problems.
- OO paradigm provides a set of useful concepts.



Module 2

Introducing Modeling and the Software Development Process



Objectives

Upon completion of this module, you should be able to:

- Describe the Object-Oriented Software Development (OOSD) process
- Describe how modeling supports the OOSD process
- Describe the benefits of modeling software
- Explain the purpose, activities, and artifacts of the following OOSD workflows: Requirements Gathering, Requirements Analysis, Architecture, Design, Implementation, Testing, and Deployment



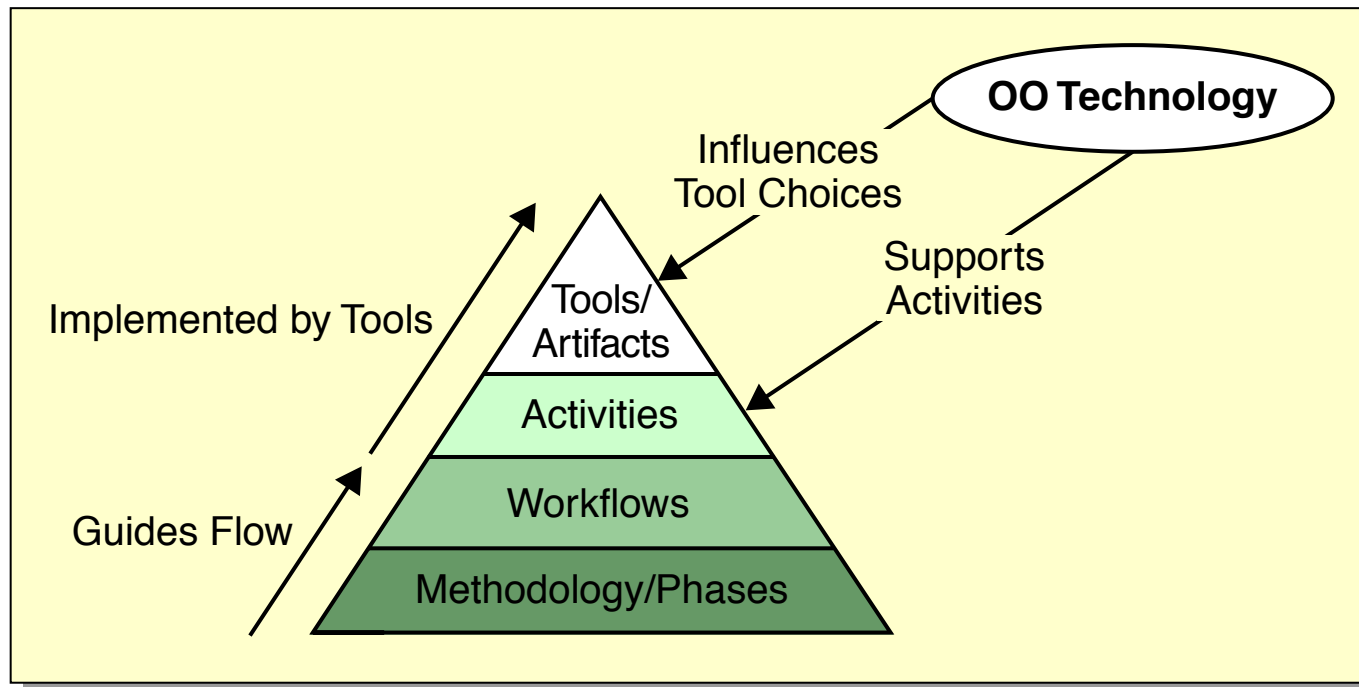
Describing Software Methodology

A methodology is “a body of methods, rules, and postulates employed by a discipline” [Webster New Collegiate Dictionary]

- In OOSD, methodology refers to the highest-level organization of a software project.
- This organization can be decomposed into medium-level phases. Phases are decomposed into workflows (disciplines). Workflows are decomposed into activities.
- Activities transform the artifacts from one workflow to another. The output of one workflow becomes the input into the next.
- The final artifact is a working software system that satisfies the initial artifacts: the system requirements.



The OOSD Hierarchy





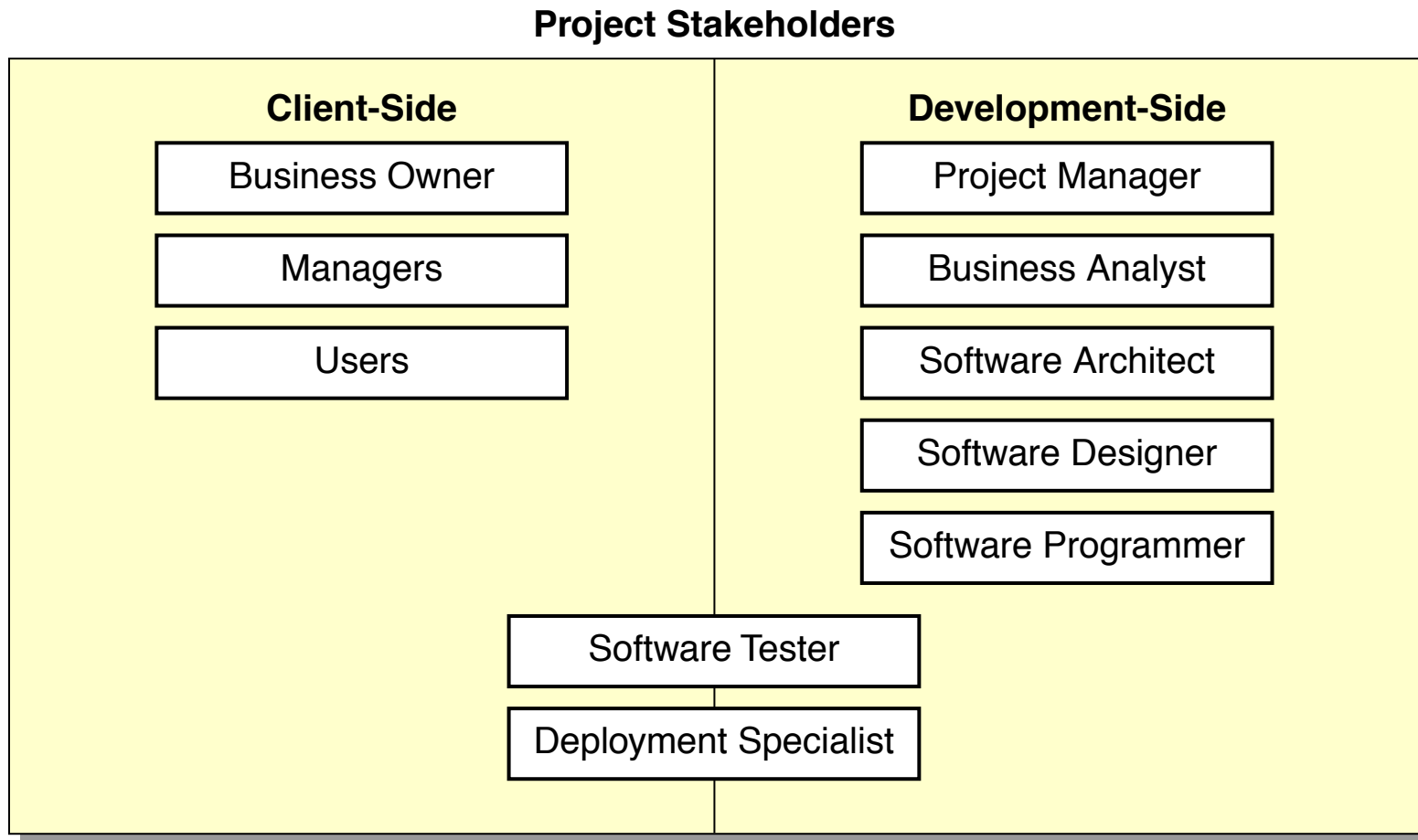
Listing the Workflows of the OOSD Process

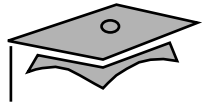
Software development has traditionally encompassed the following workflows:

- Requirements Gathering
- Requirements Analysis
- Architecture
- Design
- Implementation
- Testing
- Deployment



Describing the Software Team Job Roles



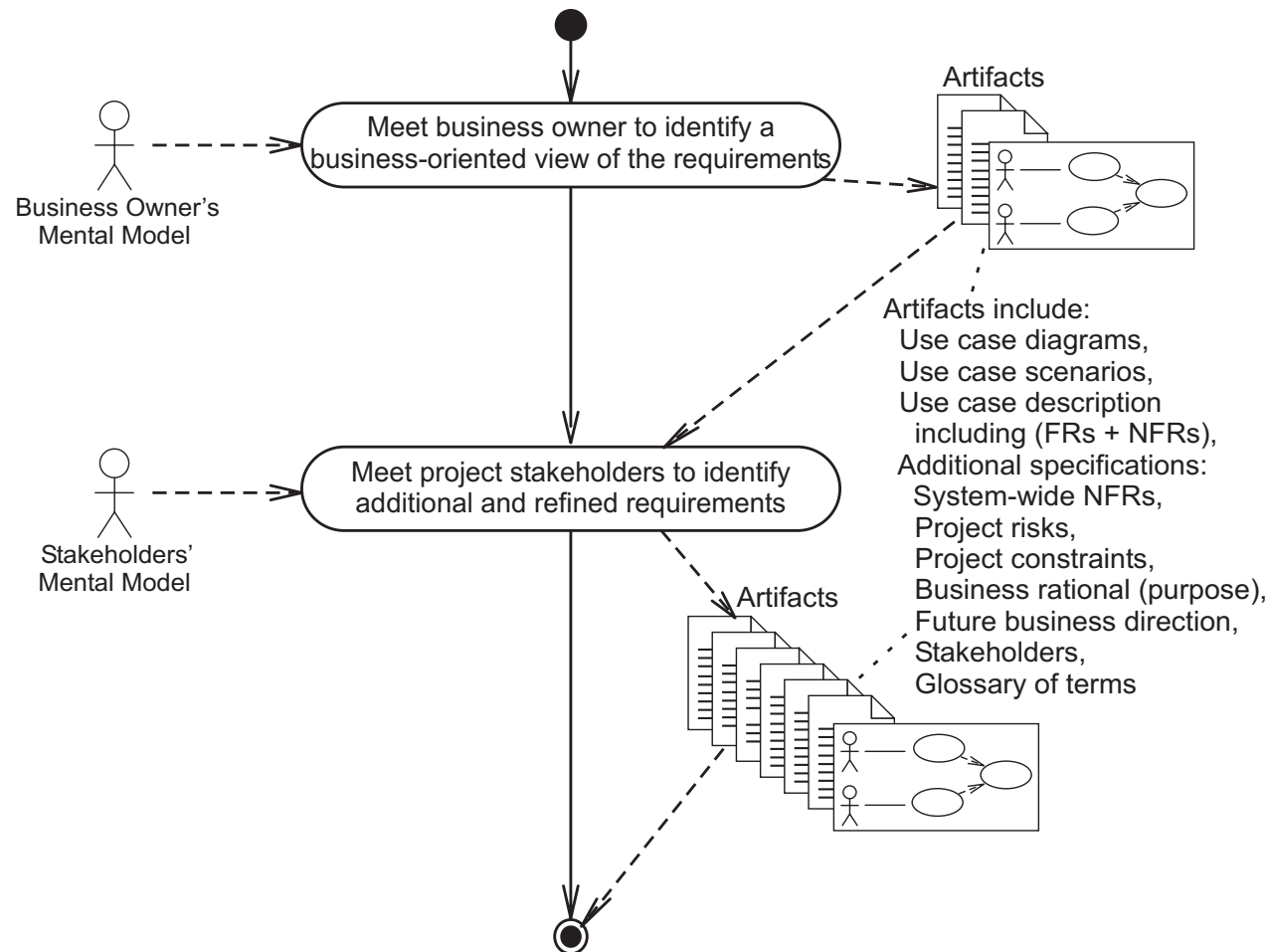


Exploring the Requirements Gathering Workflow

Workflow	Purpose	Description
Requirements Gathering	Determine <i>what</i> the system must do	Determine: <ul style="list-style-type: none">• With whom the system interacts (actor)• What behaviors (called use cases) that the system must support• Detailed behavior of each use case, which includes the low-level functional requirements (FRs)• Non-functional requirements (NFRs)



Activities and Artifacts of the Requirements Gathering Workflow



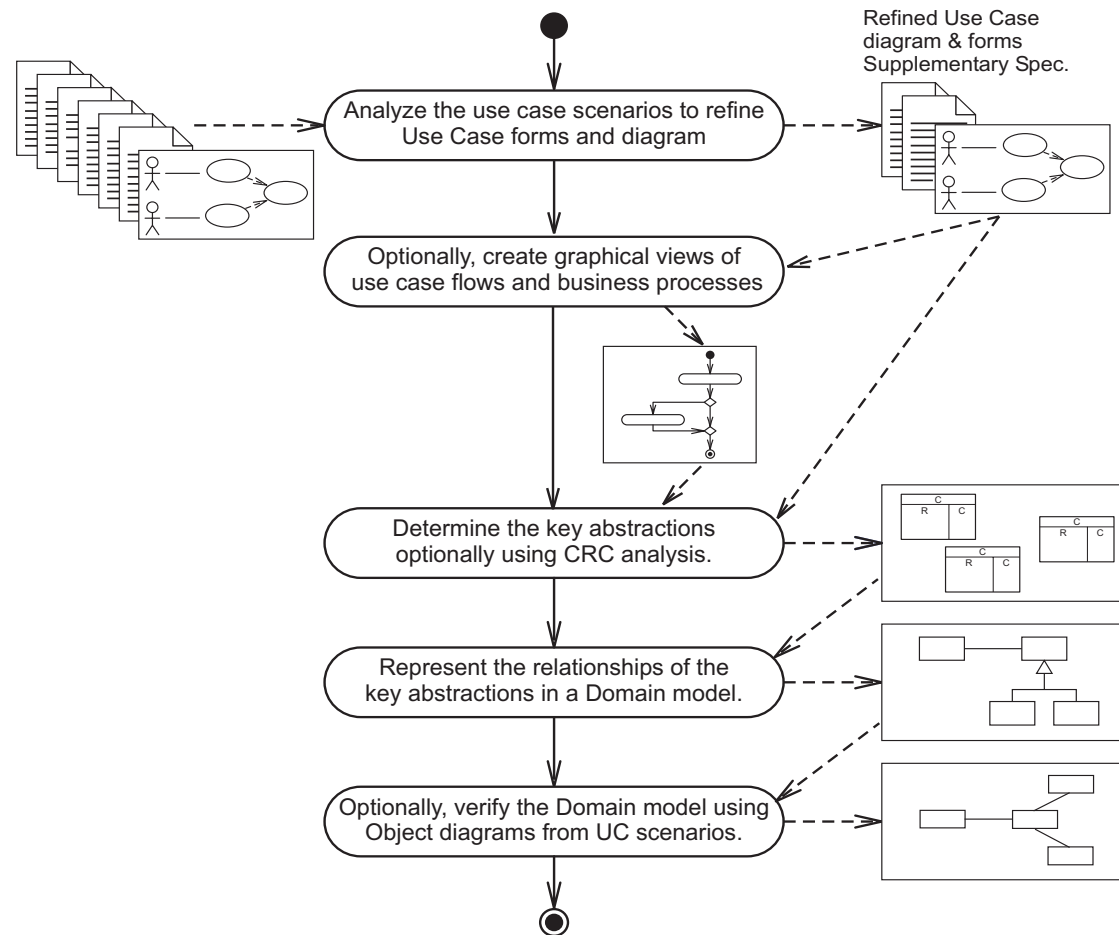


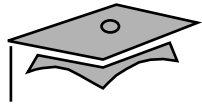
Exploring the Requirements Analysis Workflow

Workflow	Purpose	Description
Requirements Gathering	Determine <i>what</i> the system must do	
Requirements Analysis	Model the existing business processes	<p>Determine:</p> <ul style="list-style-type: none">• The detailed behavior of each use case• Supplementary use cases• The key abstractions that exist in the current increment of the problem domain• A business domain class diagram



Activities and Artifacts of the Requirements Analysis Workflow



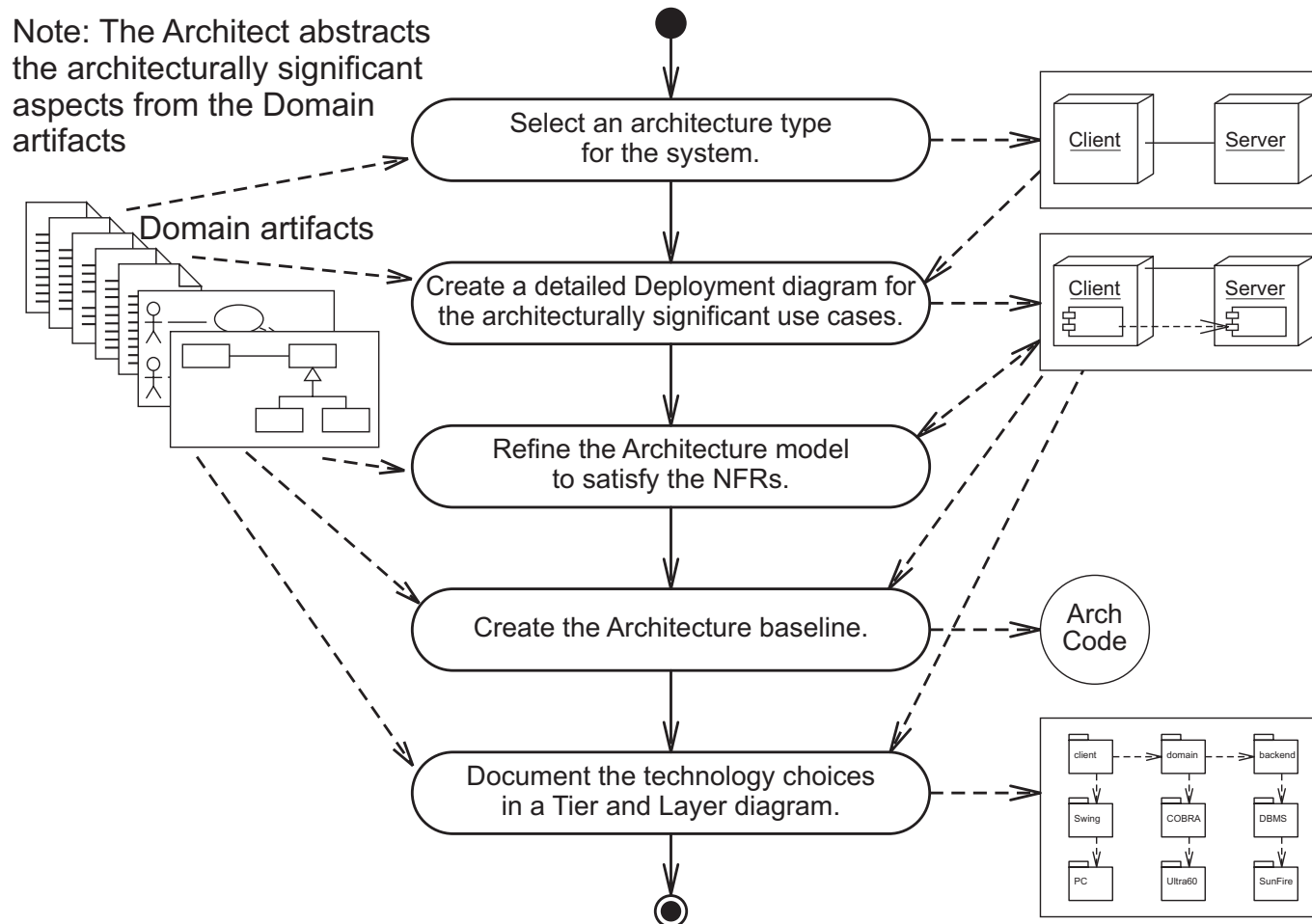


Exploring the Architecture Workflow

Workflow	Purpose	Description
Requirements Gathering	Determine <i>what</i> the system must do	
Requirements Analysis	Model the existing business processes	
Architecture	Model the high-level system structure to satisfy the NFRs	<ul style="list-style-type: none">• Develop the highest-level structure of the software solution• Identify the technologies that will support the Architecture model• Elaborate the Architecture model with Architectural patterns to satisfy NFRs



Activities and Artifacts of the Architecture Workflow





Exploring the Design Workflow

Workflow	Purpose	Description
Requirements Gathering	Determine <i>what</i> the system must do	
Requirements Analysis	Model the existing business processes	
Architecture	Model the high-level system structure to satisfy the NFRs	

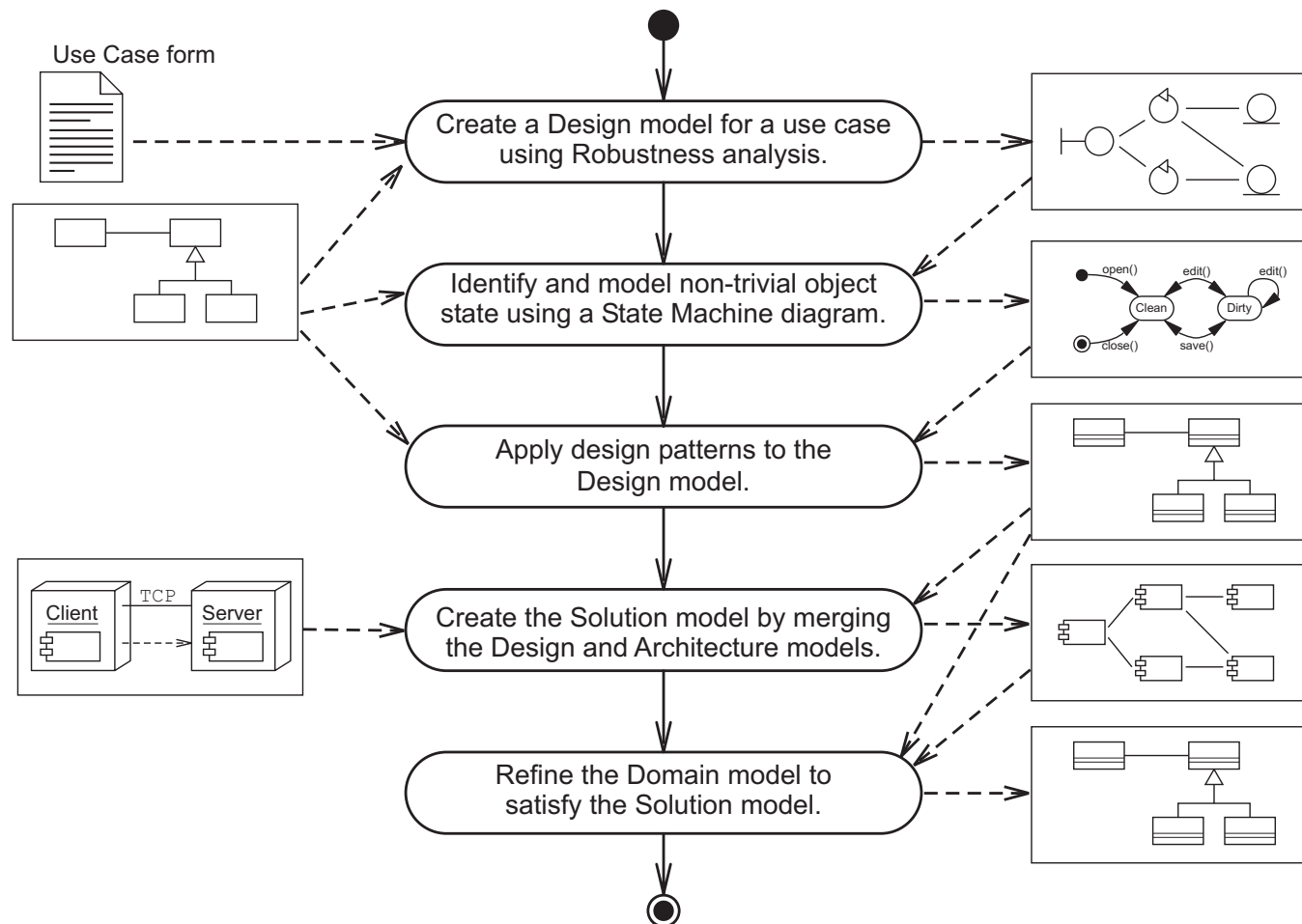


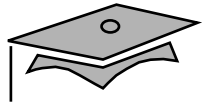
Exploring the Design Workflow

Workflow	Purpose	Description
Design	Model <i>how</i> the system will support the use cases	<ul style="list-style-type: none">• Create a Design model for a use case using Interaction diagrams• Identify and model objects with non-trivial states using a State Machine diagram• Apply design patterns to the Design model• Create a Solution model by merging the Design and Architecture models• Refine the Domain model



Activities and Artifacts of the Design Workflow





Exploring the Implementation, Testing, and Deployment Workflows

Workflow	Purpose	Description
Requirements Gathering	Determine <i>what</i> the system must do	
Requirements Analysis	Model the existing business processes	
Architecture	Model the high-level system structure to satisfy the NFRs	
Design	Model <i>how</i> the system will support the use cases	

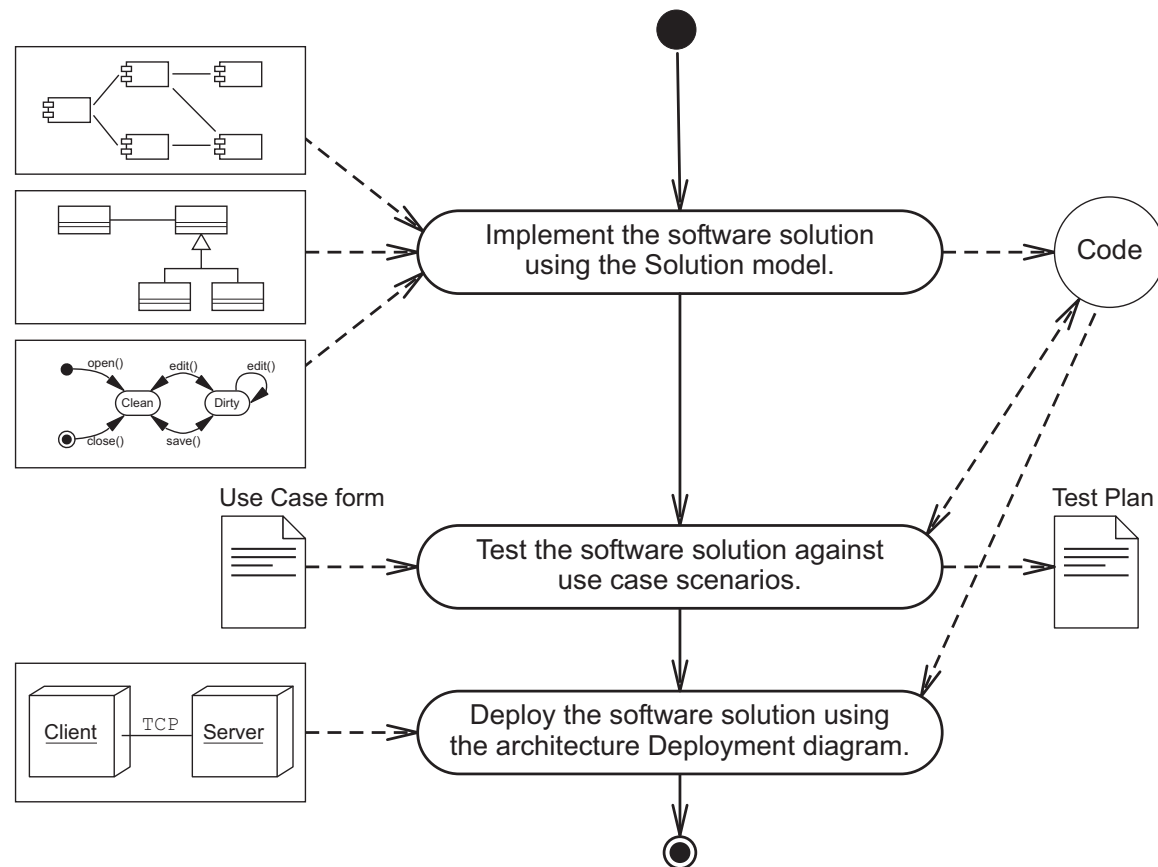


Exploring the Implementation, Testing, and Deployment Workflows

Workflow	Purpose	Description
Implementation, Testing, and Deployment	Implement, test, and deploy the system	<ul style="list-style-type: none">• Implement the software• Perform testing• Deploy the software to the production environment



Activities and Artifacts of the Implementation, Testing, and Deployment Workflows





Exploring the Benefits of Modeling Software

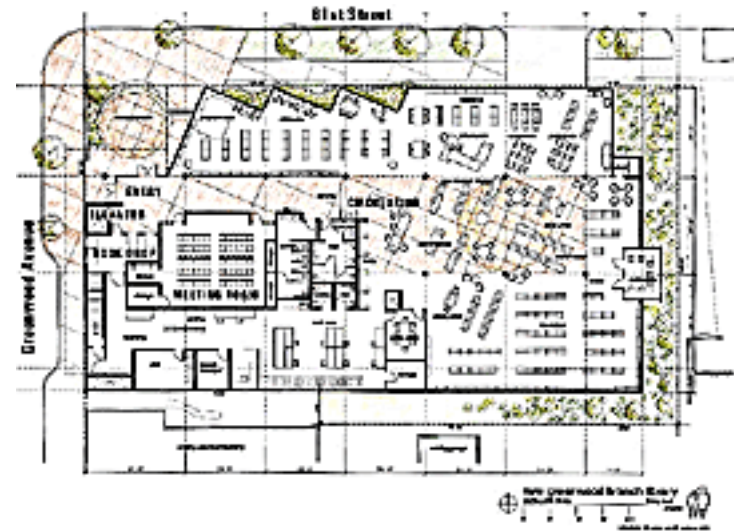
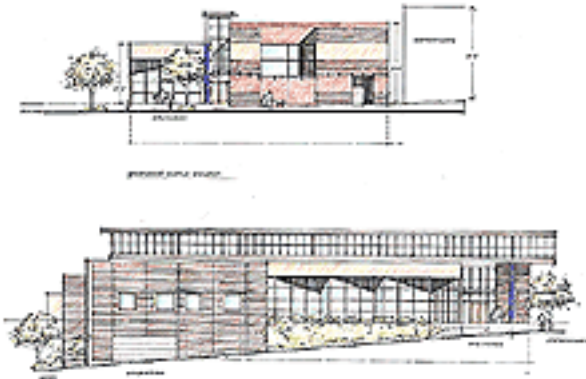
The inception of every software project starts as an idea in someone's mind.

To construct a realization of that idea, the development team must create a series of conceptual models that transform the idea into a production system.



What is a Model?

“A model is a simplification of reality.” (Booch UML User Guide page 6)



(Buffalo Design © 2002. Images used with permission.)

- A model is an abstract conceptualization of some entity (such as a building) or a system (such as software).
- Different views show the model from different perspectives.



Why Model Software?

“We build models so that we can better understand the system we are developing.” (Booch UML User Guide page 6)

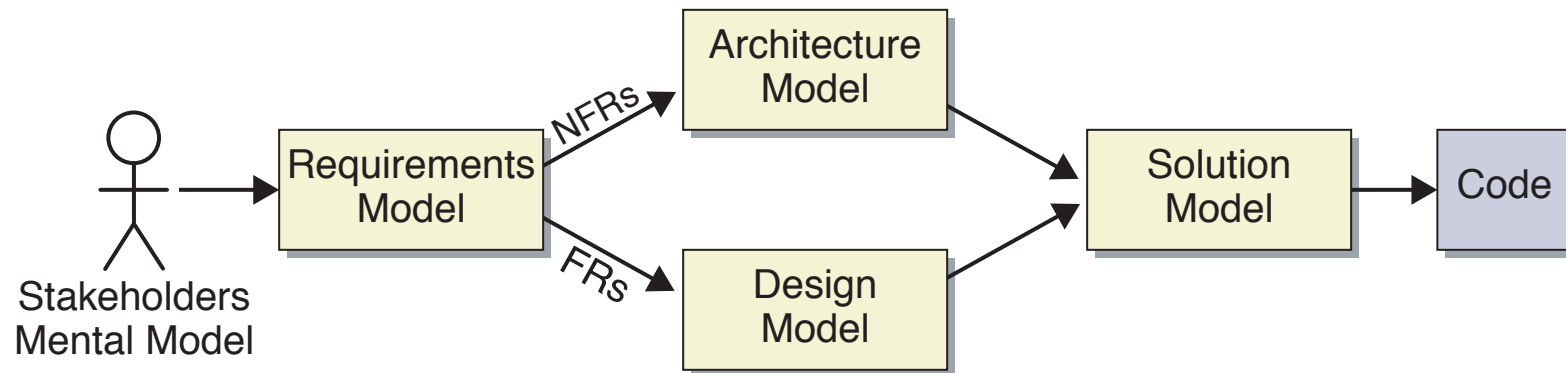
Specifically, modeling enables us to:

- Visualize new or existing systems
- Communicate decisions to the project stakeholders
- Document the decisions made in each OOSD workflow
- Specify the structure (static) and behavior (dynamic) elements of a system
- Use a template for constructing the software solution



OOSD as Model Transformations

Software development can be viewed as a series of transformations from the Stakeholder's mental model to the actual code:





Defining the UML

“The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system.”
(UML v1.4 page xix)

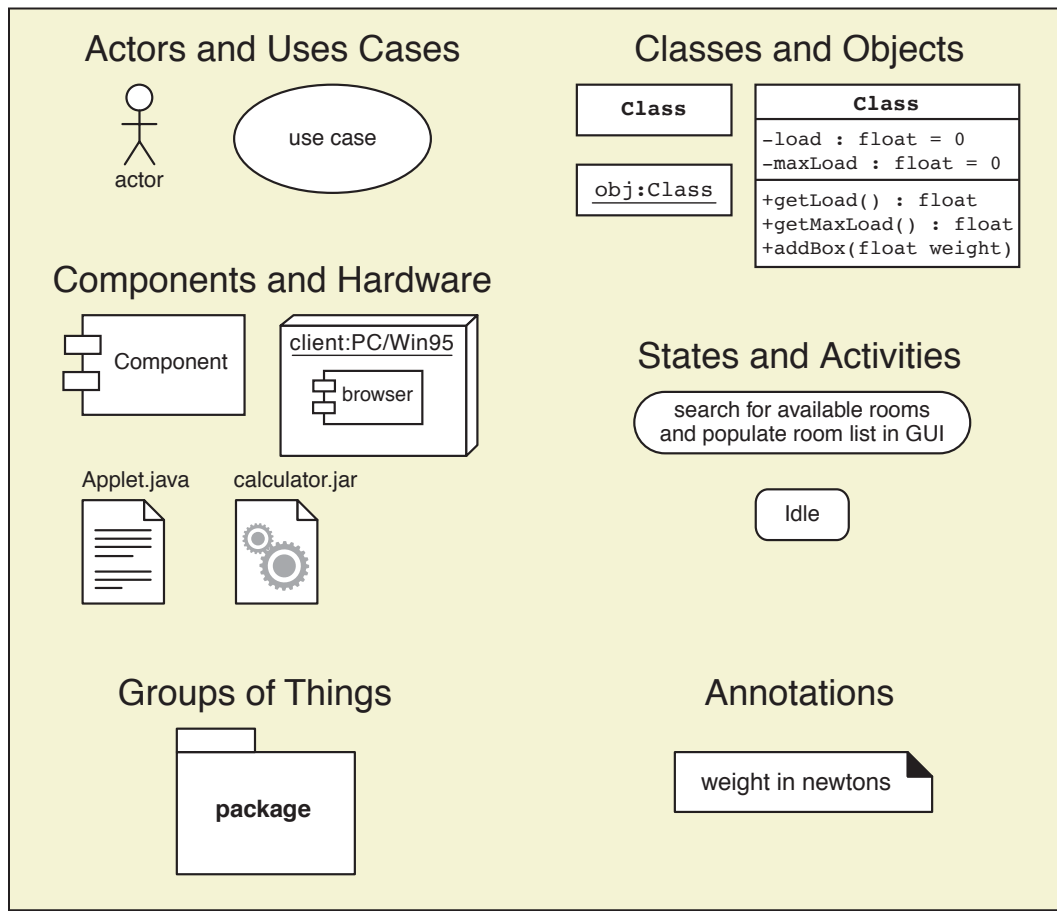
Using the UML, a model is composed of:

- Elements (things and relationships)
- Diagrams (built from elements)
- Views (diagrams showing different perspectives of a model)

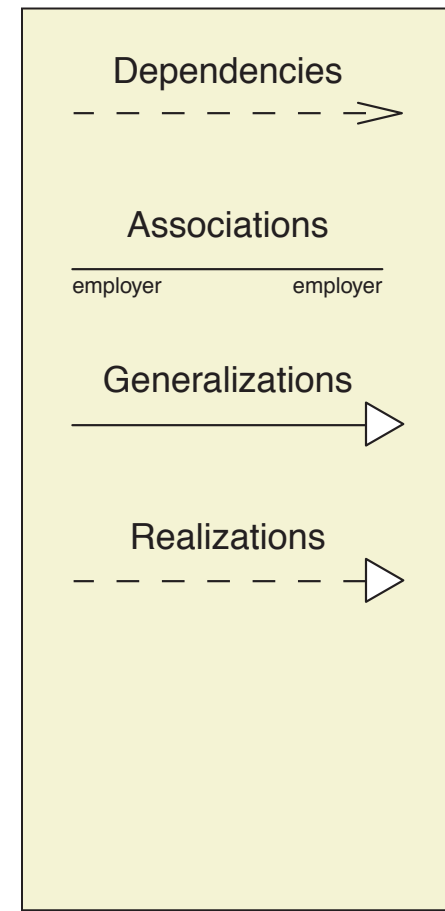


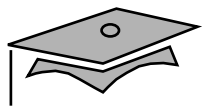
UML Elements

Things

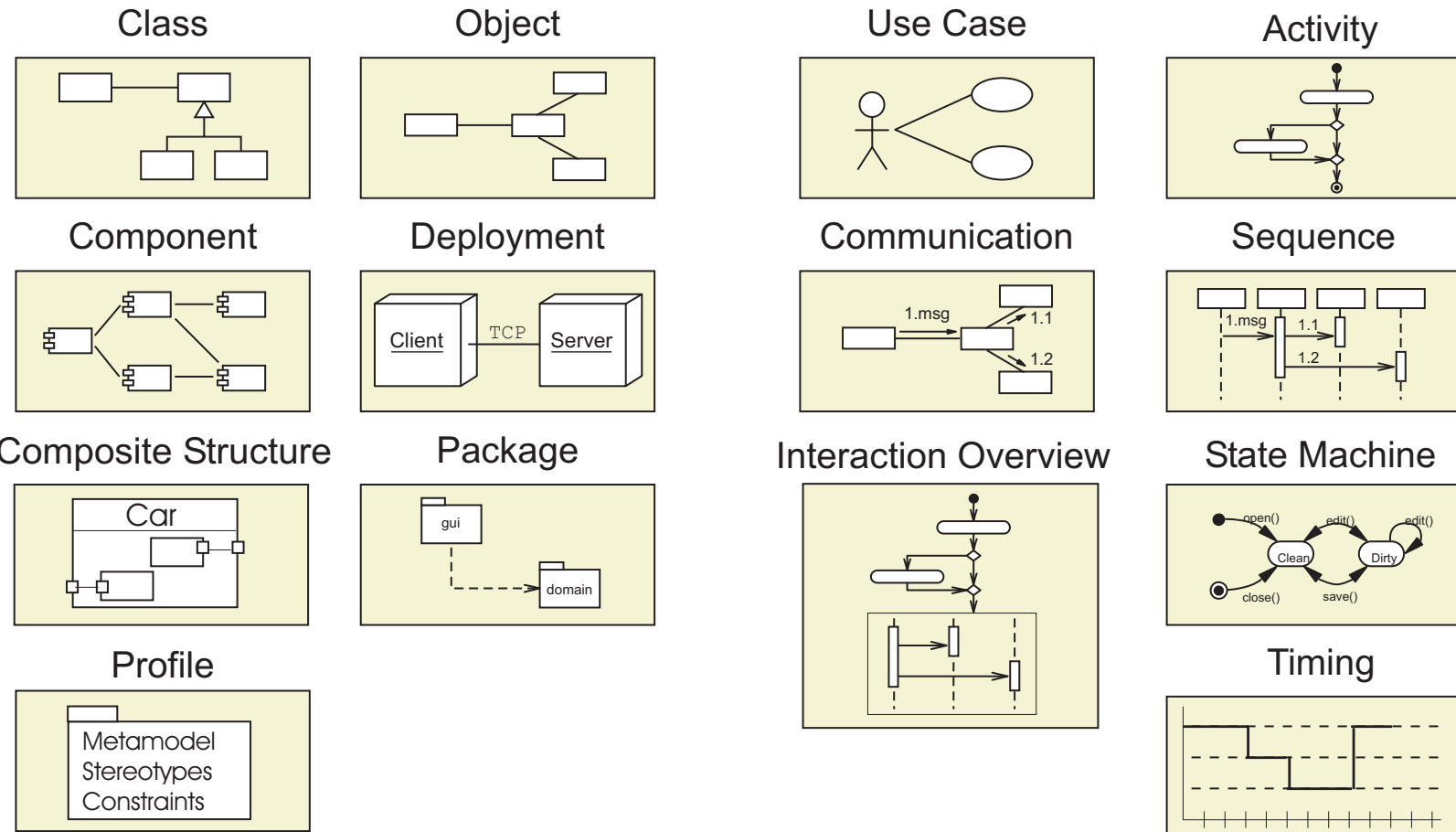


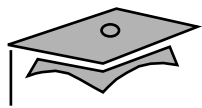
Relationships





UML Diagrams

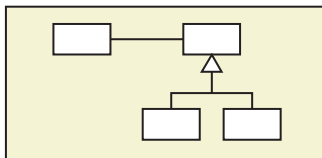




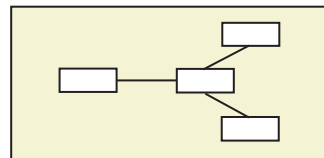
UML Diagram Categories

Structural

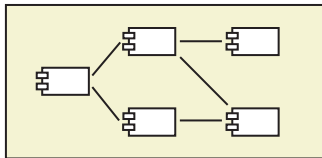
Class



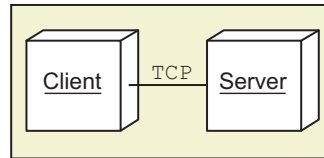
Object



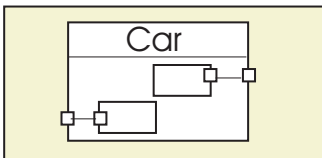
Component



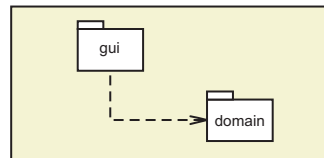
Deployment



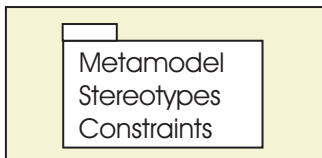
Composite Structure



Package

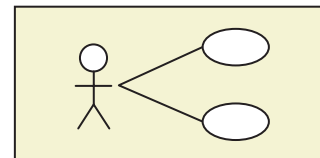


Profile

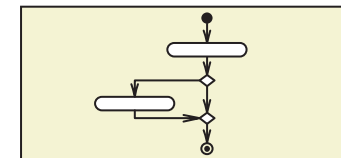


Behavioral

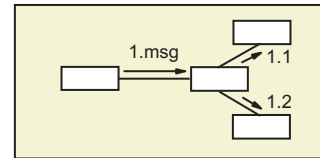
Use Case



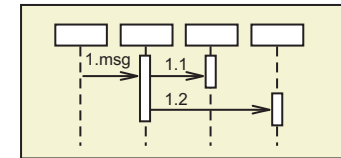
Activity



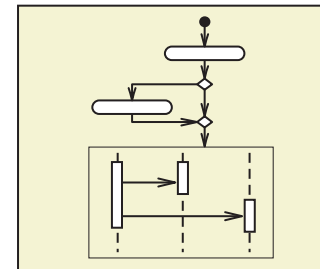
Communication



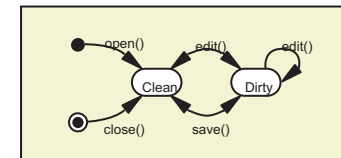
Sequence



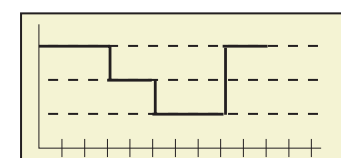
Interaction Overview



State Machine



Timing





Common UML Elements and Connectors

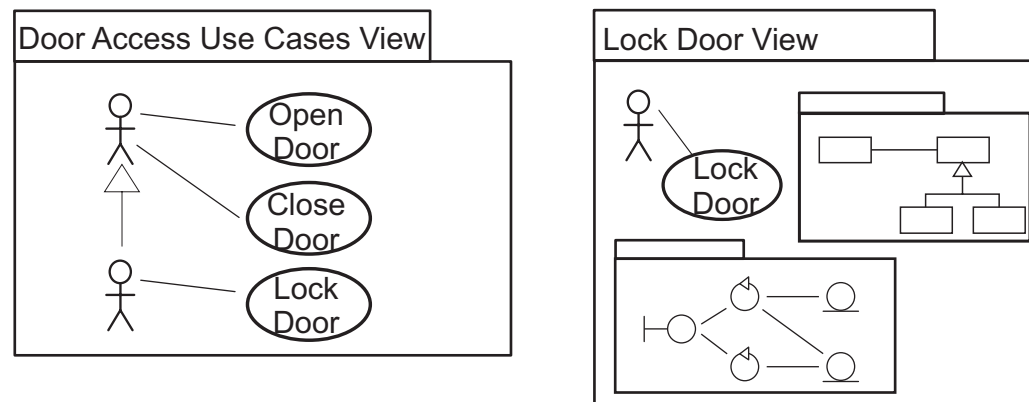
UML has a few elements and connectors that are common across UML diagrams. These include:

- Package
- Note
- Dependency
- Stereotypes

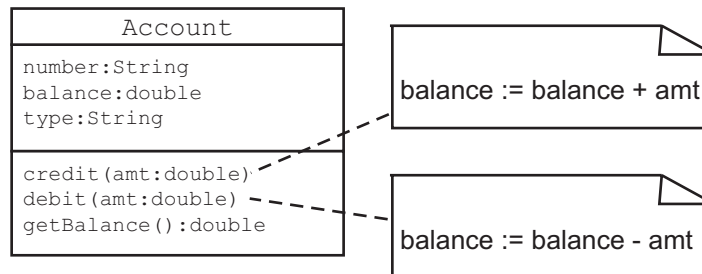


Packages and Notes

Package

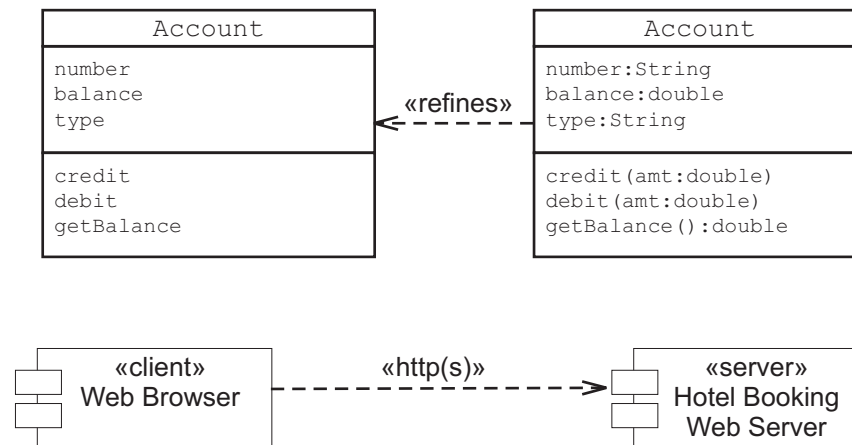


Notes





Dependency and Stereotype





What UML Is and Is Not

UML is not:	But it:
Used to create an executable model	Can be used to generate code skeletons
A programming language	Maps to most OO languages
A methodology	Can be used as a tool within the activities of a methodology



UML Tools

UML itself is a tool. You can create UML diagrams on paper or a white board. However, software tools are available to:

- Provide computer-aided drawing of UML diagrams
- Support (or enforce) semantic verification of diagrams
- Provide support for a specific methodology
- Generate code skeletons from the UML diagrams
- Organize all of the diagrams for a project
- Automatic generation of modeling elements for design patterns, Java™ Platform, Enterprise Edition (Java™ EE platform) components, and so on



Summary

- The OOSD process starts with gathering the system requirements and ends with deploying a working system.
- Workflows define the activities that transform the artifacts of the project from the requirements model to the implementation code (the final artifact).
- The UML supports the creation of visual artifacts that represent views of your models.



Module 3

Creating Use Case Diagrams



Objectives

Upon completion of this module, you should be able to:

- Justify the need for a Use Case diagram
- Identify and describe the essential elements in a UML Use Case diagram
- Develop a Use Case diagram for a software system based on the goals of the business owner
- Develop elaborated Use Case diagrams based on the goals of all the stakeholders
- Recognize and document use case dependencies using UML notation for extends, includes, and generalization
- Describe how to manage the complexity of Use Case diagrams by creating UML packaged views



Justifying the Need for a Use Case Diagram

Following are reasons a Use Case diagram is necessary:

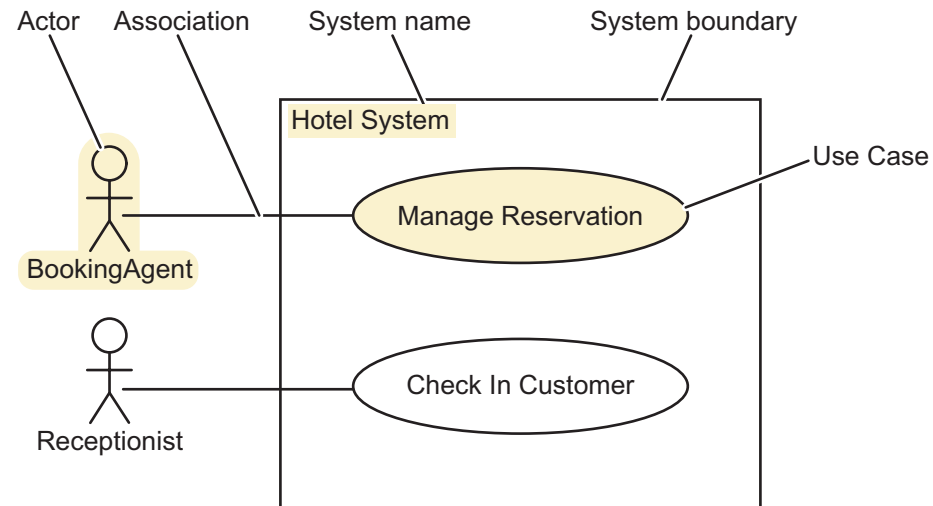
- A Use Case diagram enables you to identify—by modeling—the high-level functional requirements (FRs) that are required to satisfy each user's goals.
- The client-side stakeholders need a big picture view of the system.
- The use cases form the basis from which the detailed FRs are developed.
- Use cases can be prioritized and developed in order of priority.
- Use cases often have minimal dependencies, which enables a degree of independent development.



Identifying the Elements of a Use Case Diagram

A Use Case diagram shows the relationships between actors (roles) and the goals they wish to achieve.

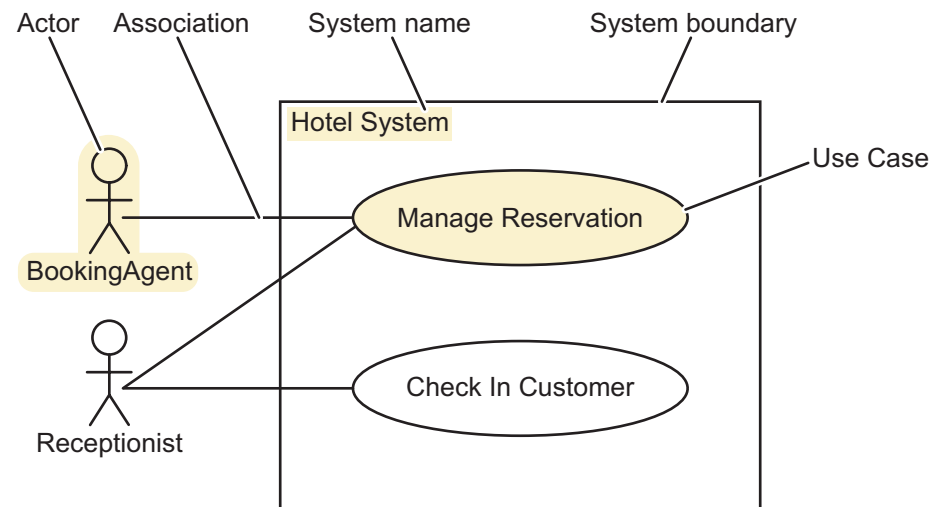
A physical job title can assume multiple actors (roles).





Identifying the Elements of a Use Case Diagram

This diagram illustrates an alternate style that explicitly shows an association between the Receptionist actor (role) and the Manage Reservation use case.





Actors

An actor:

- Models a type of role that is external to the system and interacts with that system
- Can be a human, a device, another system, or time
- Can be primary or secondary
 - Primary: Initiates and controls the whole use case
 - Secondary: Participates only for part of the use case

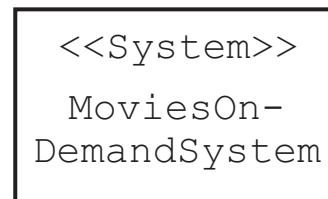
A single physical instance of a human, a device, or a system may play the role of several different actors.



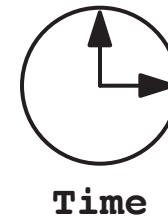
Actors



This icon represents a human actor (user) of the system.



This icon can represent any actor, but is usually used to represent external systems, devices, or time.

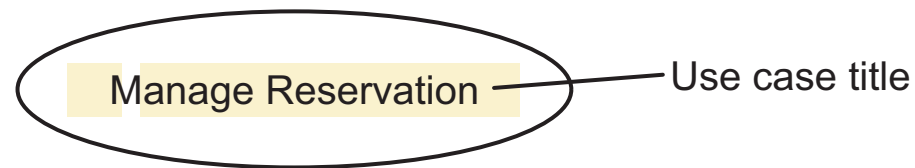


This icon represents a time-trigger mechanism that activates a use case.



Use Cases

A use case describes an interaction between an actor and the system to achieve a goal.

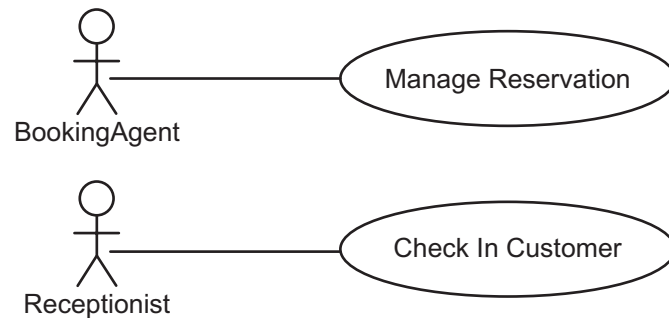


- A use case encapsulates a major piece of system behavior with a definable outcome.
- A use case is represented as an oval with the use case title in the center.
- A good use case title should consist of a brief but unambiguous verb-noun pair.
- A use case can often be UI independent.

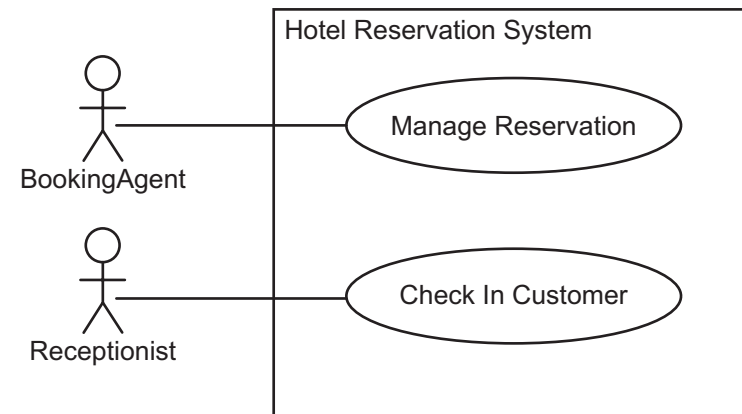


System Boundary

The use cases may optionally be enclosed by a rectangle that represents the system boundary.



The system boundary box is optional.

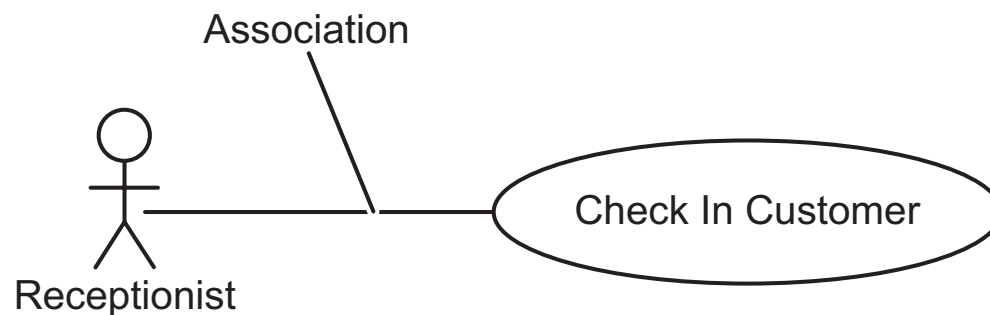


This equivalent Use Case diagram shows the system boundary for clarity.



Use Case Associations

A use case association represents “the participation of an actor in a use case.” (UML v1.4 spec. page 357)



- An actor must be associated with one or more use cases.
- A use case must be associated with one or more actors.
- An association is represented by a solid line with no arrowheads. However, some UML tools use arrows by default.



Creating the Initial Use Case Diagram

One of the primary aims of the initial meeting with the project's business owner is to identify the business-significant use cases.

- A use case diagram may be created during the meeting.
- Alternatively, the diagrams can be created after the meeting from textual notes.

The next two slides present some text showing an abstract of the use-case-specific topics discussed during the meeting.



Creating the Initial Use Case Diagram

The booking agent (internal staff) must be able to manage reservations on behalf of customers who telephone or e-mail with reservation requests. The majority of these requests will make a new reservation, but occasionally they will need to amend or cancel a reservation. A reservation holds one or more rooms of a room type for a single time period, and must be guaranteed by either an electronic card payment or the receipt of a purchase order for corporate customers and travel agents. These payment guarantees must be saved for future reference.

A reservation can also be made electronically from the Travel Agent system and also by customers directly via the internet.



Creating the Initial Use Case Diagram

The receptionist must be able to check in customers arriving at the hotel. This action will allocate one or more rooms of the requested type. In most cases, a further electronic card payment guarantee is required.

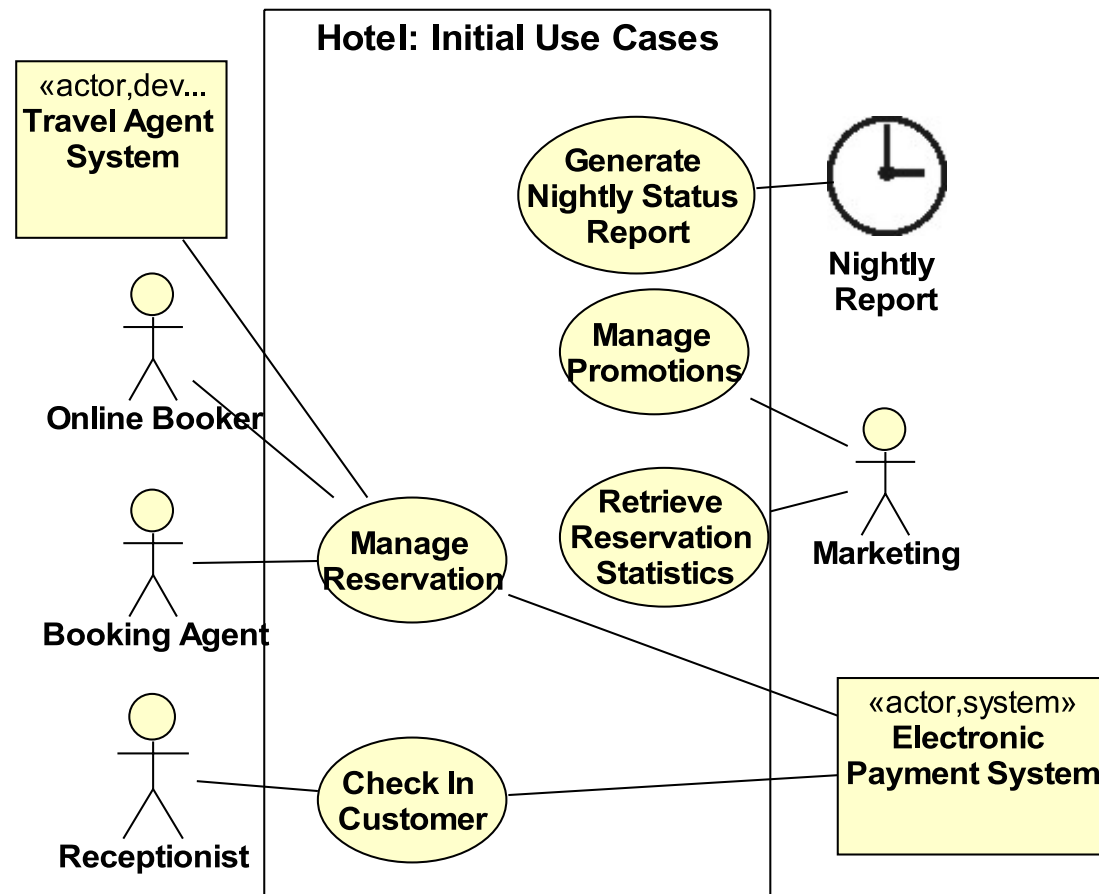
Most receptionists will be trained to perform the booking agent tasks for customers who arrive without a booking or need to change a booking.

The marketing staff will need to manage promotions (special offers) based on a review of past and future reservation statistics. The marketing staff will elaborate on the detailed requirements in a subsequent meeting.

The management needs a daily status report, which needs to be produced when the hotel is quiet. This activity is usually done at 3 a.m.



Creating the Initial Use Case Diagram





Identifying additional Use Cases

During the meeting with the business owner, you will typically discover 10 to 20 percent of the use cases needed for the system.

During the meeting with the other stakeholders, you will discover many more use case titles that you can add to the diagram. For example:

- Maintain Rooms
 - Create, Update, and Delete
- Maintain RoomTypes
 - Create, Update, and Delete



Identifying additional Use Cases

The time of discovery depends upon the development process.

- In a non-iterative process:
 - You ideally need to discover all of the remaining use case titles, bringing the total to 100 percent.
 - However, this is a resource-intensive task and is rarely completely accurate.



Identifying additional Use Cases

- In an iterative/incremental development process, an option is to:
 - Discover a total of 80 percent of the use case titles in the next few iterations for 20 percent of the effort. This is just one of the many uses of the 80/20 rule.
 - Discover the remaining 20 percent of use case titles in the later iterations for minimal effort.

This process works well with software that is built to accommodate change.

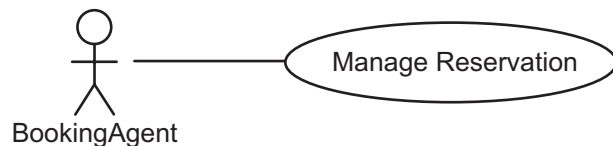


Use Case Elaboration

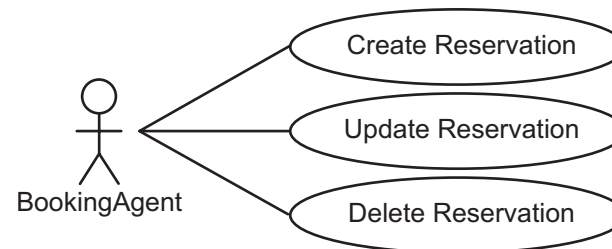
During the meeting with the other stakeholders, you will discover many more use cases that you can add to the diagram.

You might also find that some use cases are too high-level. In this case, you can introduce new use cases that separate the workflows.

Example:



Becomes:





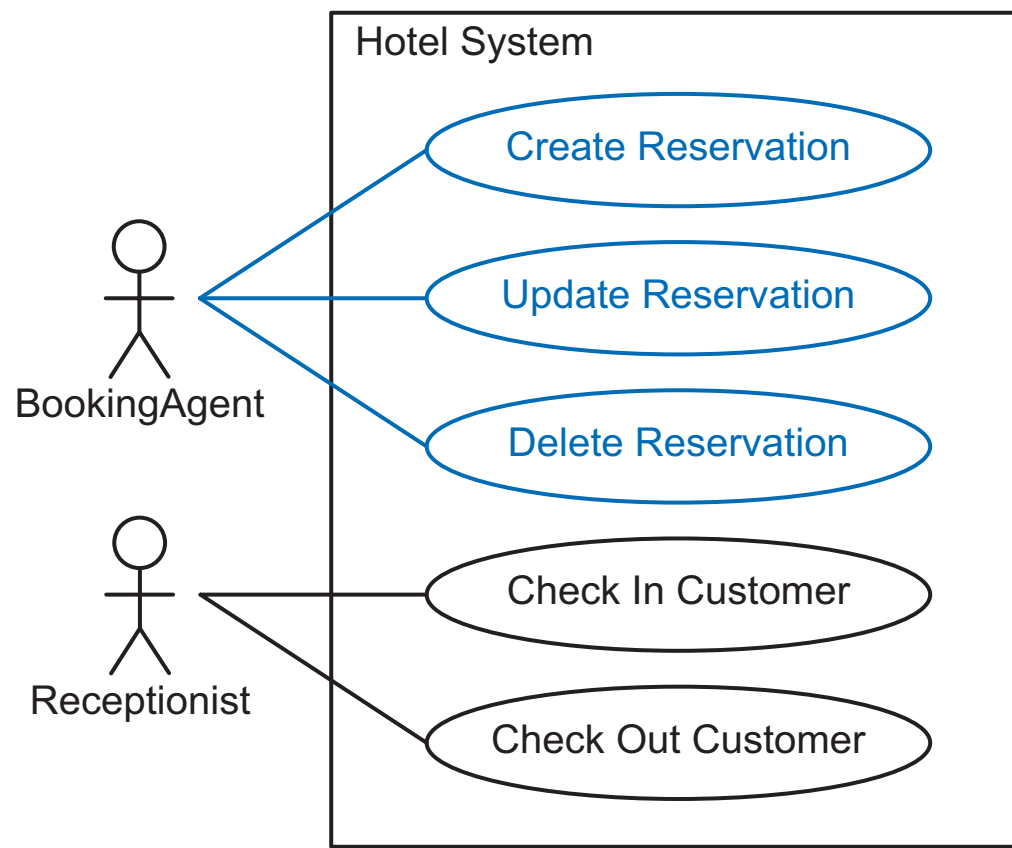
Expanding High-Level Use Cases

- Typically, *managing an entity* implies being able to Create, (Retrieve), Update, and Delete an entity (so called, CRUD operations). Other keywords include:
 - Maintain
 - Process
- Other high-level use cases can occur. Identify these by analyzing the use case scenarios and look for significantly divergent flows.
- If several scenarios have a different starting point, these scenarios might represent different use cases.



Expanding High-Level Use Cases

- The expanded diagram:





Analyzing Inheritance Patterns

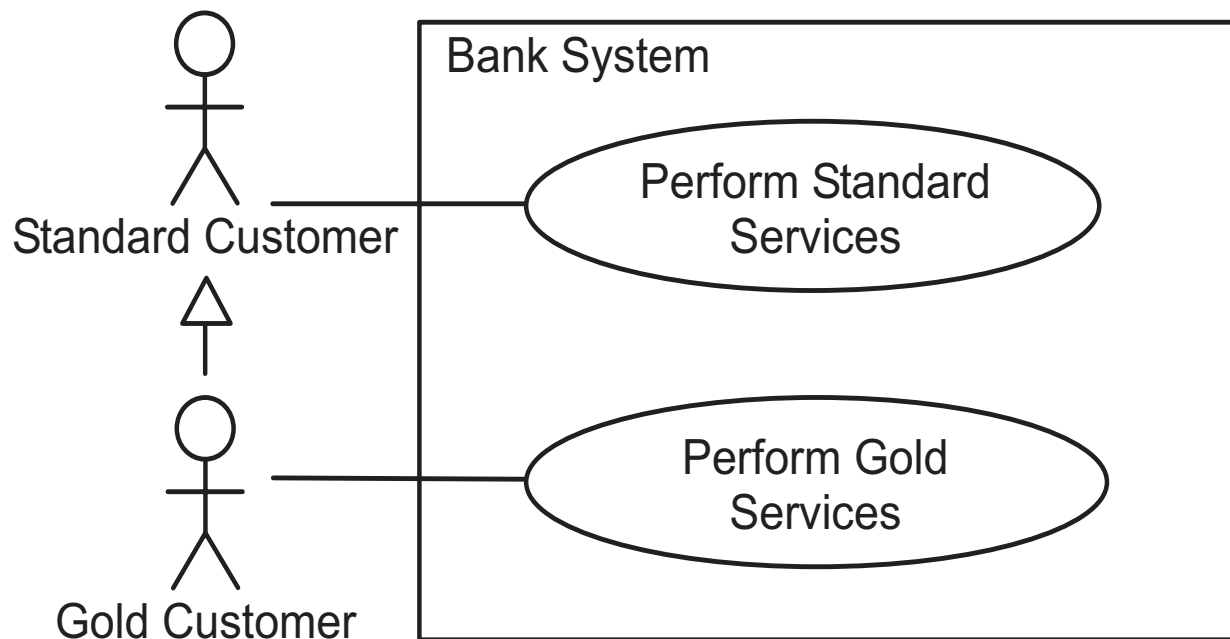
Inheritance can occur in Use Case diagrams for both actors and use cases:

- An actor can inherit all of the use case associations from the parent actor.
- A use case can be *subclassed* into multiple, specialized use cases.



Actor Inheritance

An actor can inherit all of the use case associations from the parent actor.

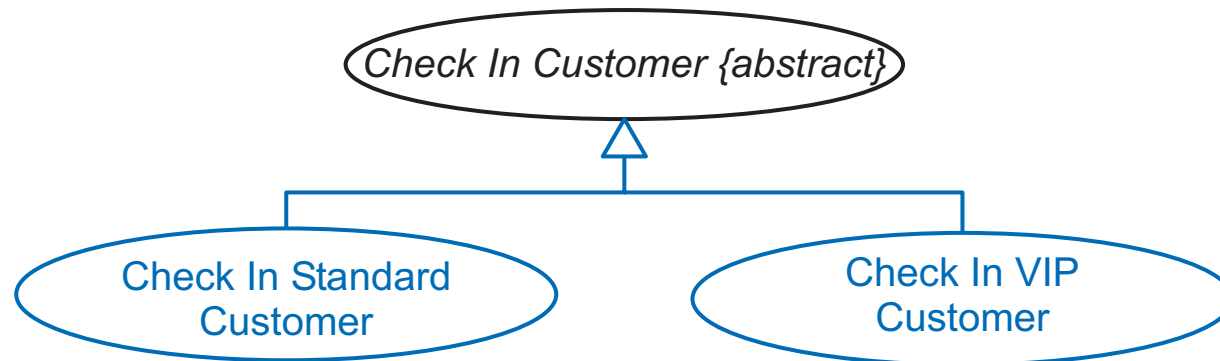


This inheritance should be used only if you can apply the “*is a kind of*” rule between the actors.



Use Case Specialization

A use case can be *subclassed* into multiple, specialized use cases:



- Use case specializations are *usually* identified by significant variations in the use case scenarios.
- If the base use case cannot be instantiated, you must mark it as abstract.



Analyzing Use Case Dependencies

Use cases can depend on other use cases in two ways:

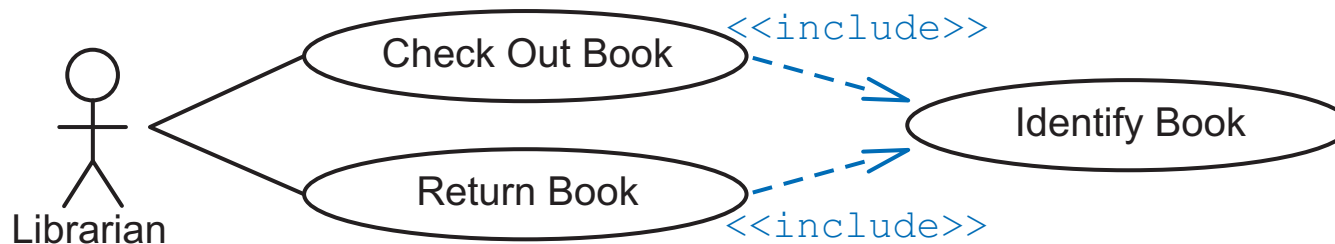
- One use case (a) *includes* another use case (i).
This means that the one use case (a) requires the behavior of the other use case (i) and *always* performs the included use case.
- One use case (e) can *extend* another use case (b).
This means that the one use case (e) can (optionally) extend the behavior of the other use case (b).



The «include» Dependency

The include dependency enables you to identify behaviors of the system that are common to multiple use cases.

This dependency is drawn like this:

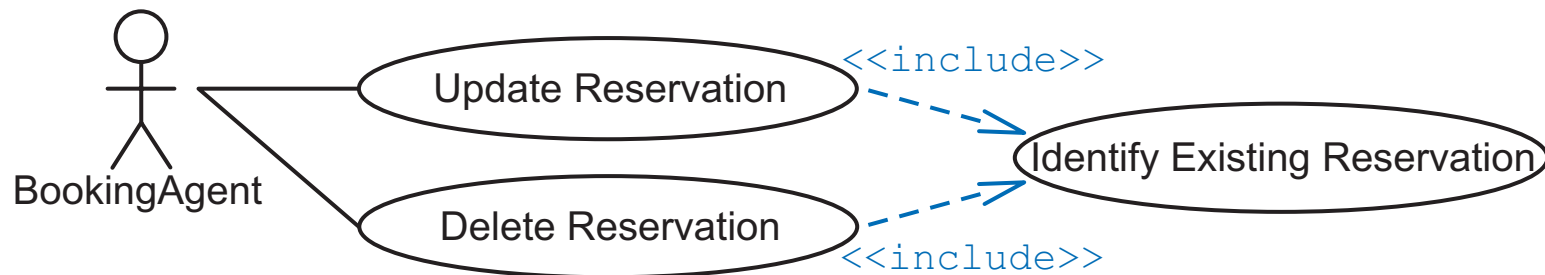




The «include» Dependency

Identifying and recording common behavior:

- Review the use case scenarios for common behaviors.
- Give this behavior a name and place it in the Use Case diagram with an «include» dependency.





The «include» Dependency

Identifying behavior associated with a secondary actor:

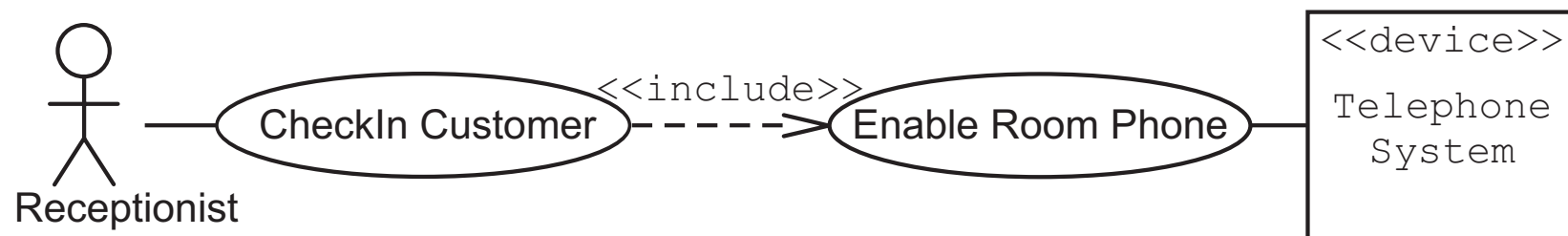
- Review the use case scenarios for significant behavior that involves a secondary actor.





The «include» Dependency

- Split the behavior that interacts with this secondary actor. Give this behavior a Use Case title, and place it in the Use Case diagram with an «include» dependency.

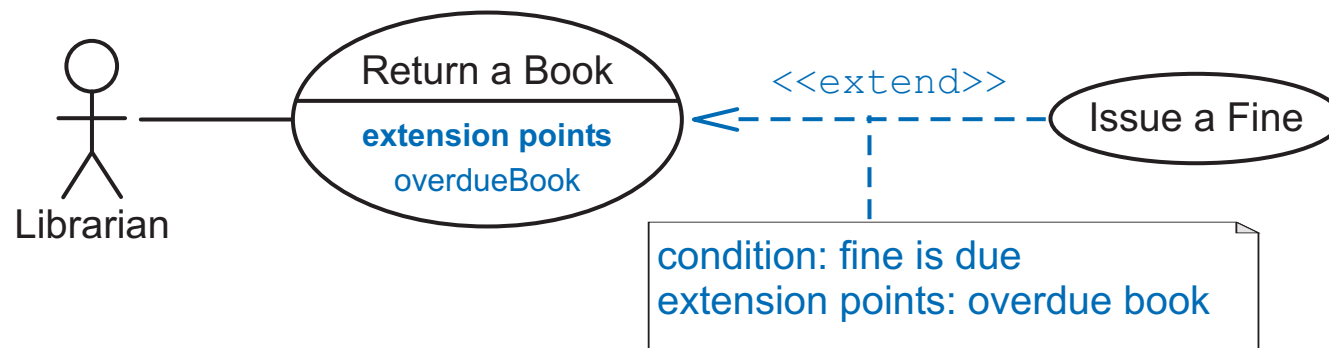




The «extend» Dependency

The extend dependency enables you to identify behaviors of the system that are not part of the primary flow, but exist in alternate scenarios.

This dependency is drawn like this:

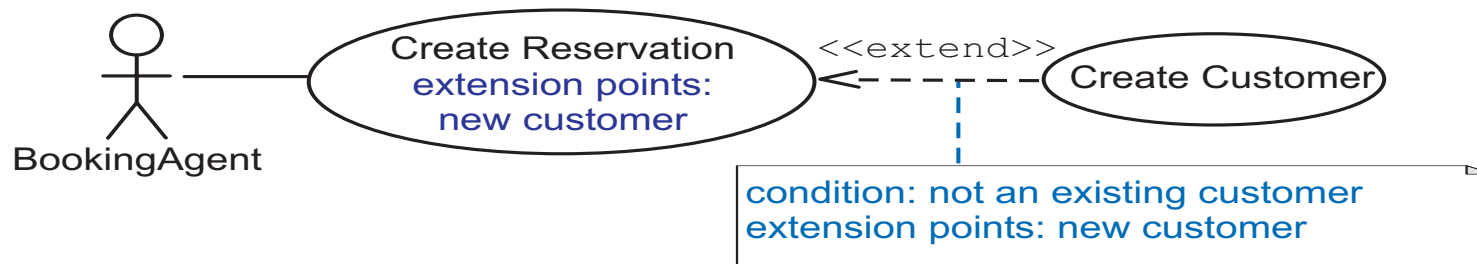


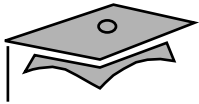


The «extend» Dependency

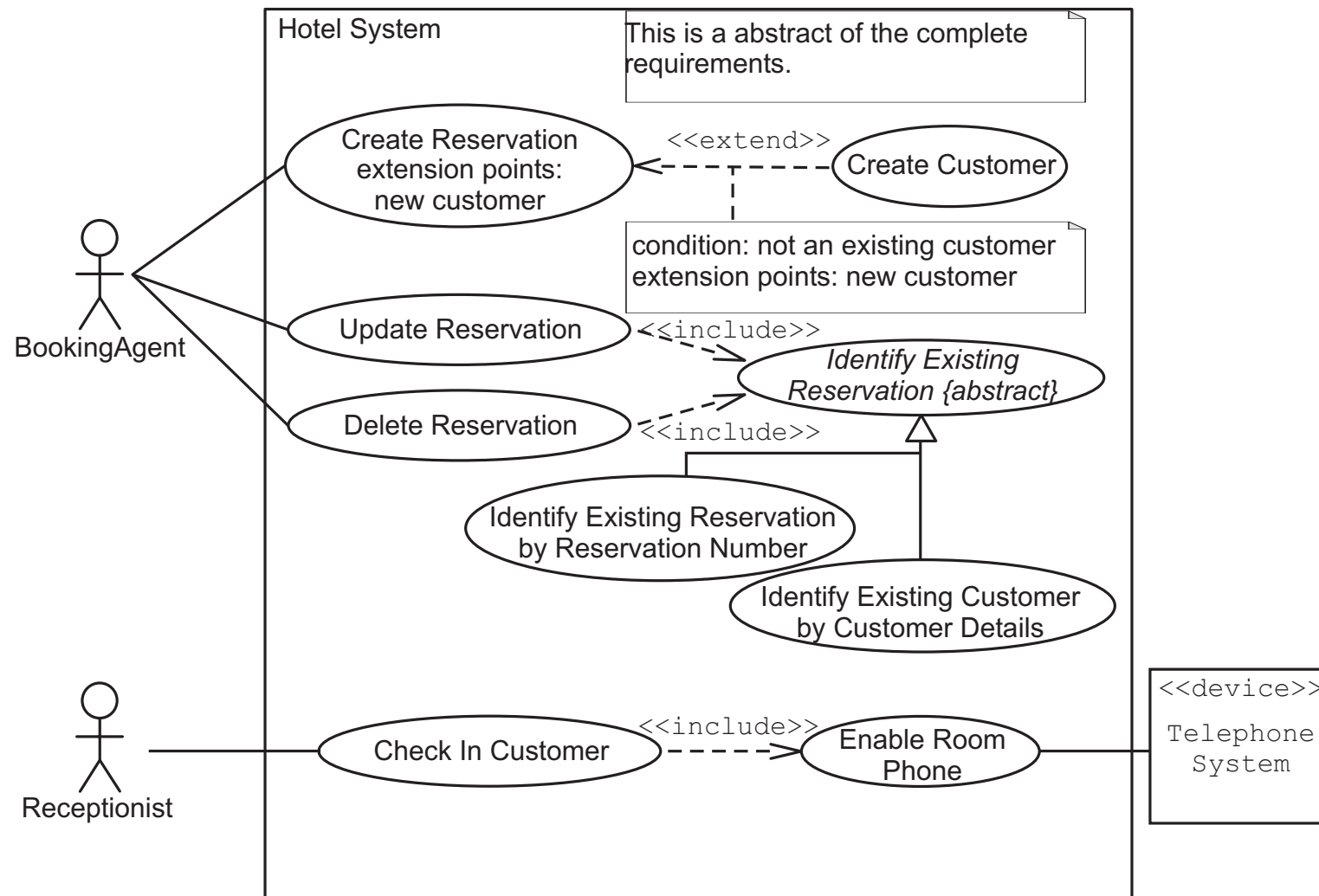
Identifying and recording behaviors associated with an alternate flow of a use case:

- Review the use case scenarios for significant and cohesive sequences of behavior.
- Give this behavior a name and place it in the Use Case diagram with a «extend» dependency.





A Combined Example

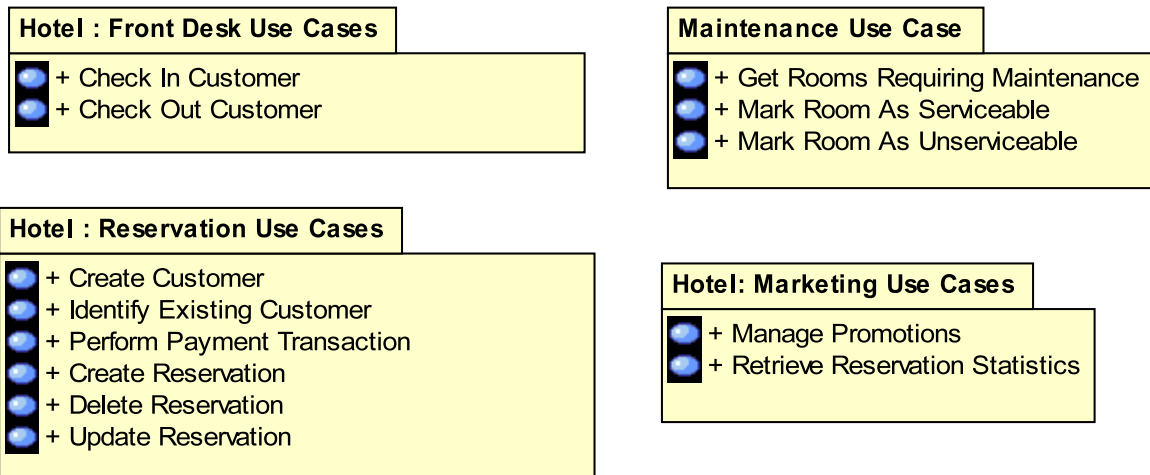




Packaging the Use Case Views

It should be apparent that any non-trivial software development would need more use cases than could be viewed at one time. Therefore, you need to be able to manage this complexity.

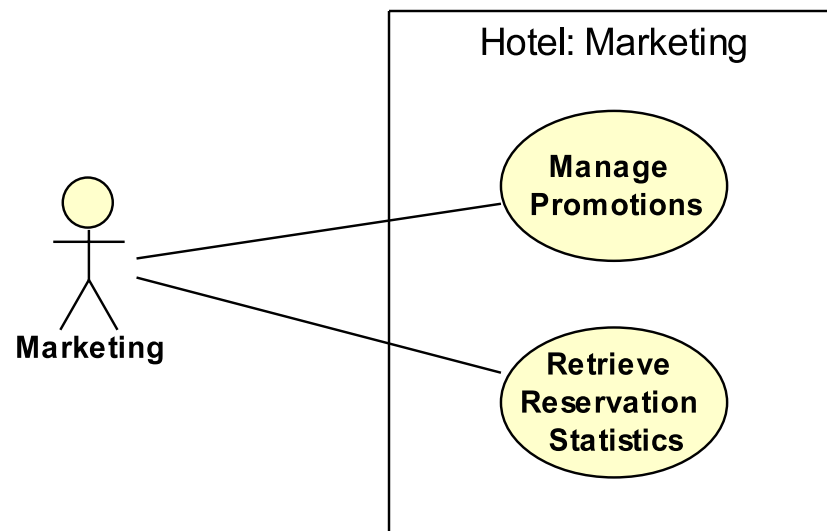
One way of managing this complexity is to break down the use cases into packages.





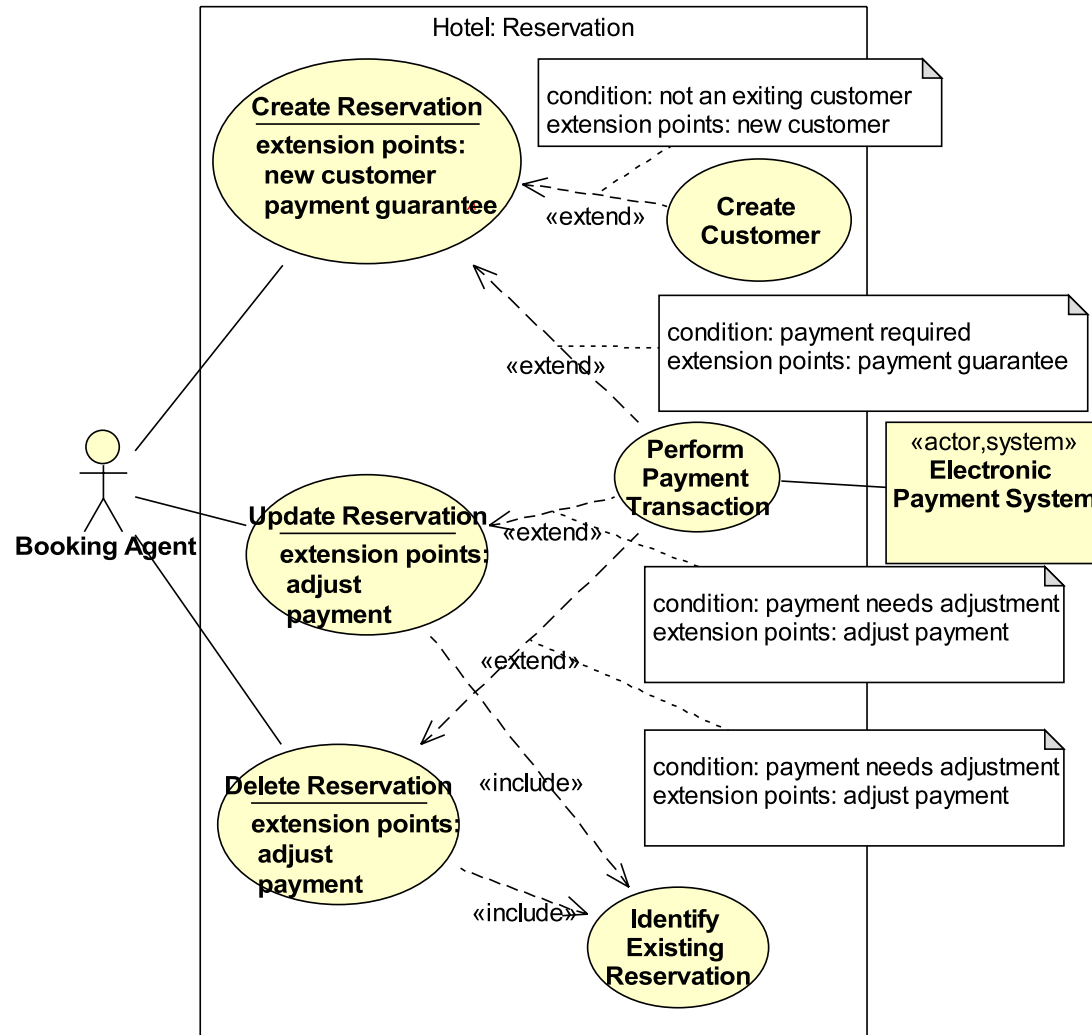
Packaging the Use Case Views

- You can look inside each package to reveal the detailed content.
- A use case element may exist in multiple packages, where it participates in multiple views.





Packaging the Use Case Views





Summary

- A Use Case diagram provides a visual representation of the big-picture view of the system.
- The Use Case diagram represents the actors that use the system, the use cases that provide a behavior with a definable goal for an actor, and the associations between them.
- Use Case diagrams can be elaborated to show a software system based on the goals of the business owner and all the other stakeholders



Summary

- Use Case diagrams can be elaborated to show use case dependencies by using UML notation for extends, includes, and generalization.
- Complex Use Case diagrams can be broken down into views by using UML packages.



Module 4

Creating Use Case Scenarios and Forms



Objectives

Upon completion of this module, you should be able to:

- Identify and document scenarios for a use case
- Create a Use Case form describing a summary of the scenarios in the main and alternate flows
- Describe how to reference included and extending use cases.
- Identify and document non-functional requirements (NFRs), business rules, risks, and priorities for a use case
- Identify the purpose of a Supplementary Specification Document



Recording Use Case Scenarios

A Use Case scenario is a concrete example of a use case.

A Use Case scenario should:

- Be as specific as possible
- Never contain conditional statements
- Begin the same way but have different outcomes
- Not specify too many user interface details
- Show successful as well as unsuccessful outcomes (in different scenarios)

Use Case scenarios drive several other OOAD workflows.



Selecting Use Case Scenarios

While it is ideal to have multiple scenarios for all use cases, doing so would take a lot of time. Therefore, you can select appropriate scenarios by the following criteria:

- The use case involves a complex interaction with the actor.
- The use case has several potential failure points, such as interaction with external systems or a database.

There are two types of scenarios:

- Primary (Happy) scenarios record successful results.
- Secondary (Sad) scenarios record failure events.



Writing a Use Case Scenario

A Use Case scenario is a story that:

- Describes how an actor uses the system and how the system responds to the actions of the actor.
- Has a beginning, a middle, and an end.



Primary Use Case Scenario: Example

The beginning:

The use case begins when the booking agent receives a request to make a reservation for rooms in the hotel.



Primary Use Case Scenario: Example

The middle:

The booking agent enters the arrival date, the departure date, and the quantity of each type of room that is required. The booking agent then submits the entered details. The system finds rooms that will be available during the period of the reservation and allocates the required number and type of rooms from the available rooms. The system responds that the specified rooms are available, returns the provisional reservation number, and marks the reservation as “held”. The booking agent accepts the rooms offered.



Primary Use Case Scenario: Example

More of the middle:

The booking agent selects that the customer has visited one of the hotels in this group before, and enters the zip code and customer name. The system finds and returns a list of matching customers with full address details. The booking agent selects one of the customers as being the valid customer. The system assigns this customer to the reservation. The booking agent performs a payment guarantee check. This check is successful.



Primary Use Case Scenario: Example

The end:

The system assigns the payment guarantee to the reservation and changes the state of the reservation to “confirmed”. The system returns the reservation ID and booking details.



Secondary Use Case Scenario: Example

The beginning:

The use case begins when the booking agent receives a request to make a reservation for rooms in the hotel.

The middle:

The booking agent enters the arrival date, the departure date, and the quantity of each type of room that is required. The booking agent then submits the entered details. The system responds that there are no rooms available of any type for the date range specified in the request.

The end:

The use case ends.



Supplementary Specifications

Some of the project information that you gather cannot be stored with the use cases because this information needs to be shared by several use cases.

This additional information can be documented in a Supplementary Specification Document, which often contains:

- NFRs
- Project Risks
- Project Constraints
- Glossary of Terms



Non-Functional Requirements (NFRs)

- Non-functional requirements (NFRs) define the qualitative characteristics of the system. As in an animal, the NFRs describe strength, speed, and agility of the internal features of the animal. How fast can the animal move? How much weight can the animal carry?
- Any adverbial phrase can be an NFR.



NFRs: Examples

- NFR1: The system must support 200 simultaneous users in the Web application.
- NFR2: The process for completing any reservation activity must take the average user no more than 10 minutes to finish.
- NFR3: The capacity of reservation records could grow to 2,600 per month.
- NFR4: The Web access should use the HTTPS transport layer when critical customer information is being communicated.
- NFR5: The numerical accuracy of all financial calculations (for example, reports and customer receipts) should follow a 2-significant-digit precision with standard rounding of intermediate results.



NFRs: Examples

- NFR6: The System must be available “7 by 24 by 365”. However, the applications can be shut down for maintenance once a week for one hour. This maintenance activity should be scheduled between 3 a.m. and 6 a.m.
- NFR7: Based on historical evidence, there are approximately 600 reservations per month per property.
- NFR8: The search for available rooms must take no longer than 30 seconds.



Glossary of Terms

The Glossary of Terms defines business or IT terms that will be used in the project.

This is a living document, which should be appended with new terms, or amended if a term is found to be incorrect or needs redefinition.



Glossary of Terms: Examples

Term	Definition
Reservation	An allocation of a specific number of rooms, each of a specified <i>room type</i> , for a specified period of days.
Date Range	Specifies a start date and an end date.
Room	A resource that can be allocated to a reservation, and is occupied by that reservation <i>customer</i> and their <i>guests</i> for the <i>date range</i> of the <i>reservation</i> . A room is identified by either a <i>room name</i> or a <i>room number</i> . Each room is assigned a <i>room type</i> .
Payment Guarantee	Debit/Credit card pre-authorization or purchase order from either corporate companies or travel agents.
Basic Rate	The per day price for a room type without any additional <i>in-line charges</i> or <i>promotions</i> .
Room Type	A room type indicates the number of beds, <i>basic rate</i> , and configuration of the room.



Description of a Use Case Form

A Use Case form provides a tool to record the detailed analysis of a single use case and its scenarios.

Form Element	Description
Use Case Name	The name of the use case from the Use Case diagram.
Description	A one-line or two-line description of the purpose of the use case.
Actors	This element should list all relevant actors that are permitted to use this use case.
Priority	This is used to describe the relative priority of this use case. Priority is often in the form of MuSCoW prioritization, which is Must have, Should have, Could have, or Won't have.
Risk	A High, Medium, or Low ranking of this use case's risk factors.



Description of a Use Case Form

Form Element	Description
Pre-conditions and assumptions	The conditions that must be true. If these conditions are not true, the outcome of the use case cannot be predicted.
Extension Points	A list of any extension points used by this use case.
Extends	A list of any use cases that this use case extends.
Trigger	The condition that “informs” the actor that the use case should be invoked.
Flow of Events	The primary trace of user actions and events that constitute this use case.
Alternate Flows	Any and all secondary traces of user actions and events that are possible in this use case.
Post-conditions	The conditions that shall exist after the use case has been completed.



Description of a Use Case Form

Form Element	Description
Business Rules	A list of business rules that must be complied with and that are related to this use case. These rules might be referred to during the execution of the use case in the main flow and the alternate flow, but this is not always necessary. You can describe these rules in this form. Alternatively, you can refer to the list in the Supplementary Specification Document.
Non-Functional Requirements	A list of the NFRs that are related to this use case. You can either summarize the NFRs or list their codes from the Supplementary Specification Document.
Notes	Any other information that can be of value regarding this use case.



Description of a Use Case Form

Some methodologies recommend more or less analysis of the use cases. The Analysis workflow presented in this module tends to be more detailed.

Use Case forms are not standard. There are different styles that can be used to create a Use Case form.



Creating a Use Case Form

Steps to determine the information for the Use Case form:

1. Determine a brief description from the primary scenarios.
2. Determine the actors who initiate and participate in this use case from the Use Case diagrams.
3. Determine the priority of this use case from discussions with the stakeholders.
4. Determine the risk from scenarios and from discussions with the stakeholders.
5. Determine the extension points from the Use Case diagrams.
6. Determine the pre-conditions from the scenarios.
7. Determine the trigger from the scenarios.



Creating a Use Case Form

8. Determine the flow of events from the primary (happy) scenarios.
9. Determine the alternate flows from the secondary (sad) scenarios.
10. Determine the business rules from scenarios and from discussions with stakeholders.
11. Determine the post-conditions.
12. Determine the new NFRs from discussions with stakeholders.
13. Add notes for information—gathered from discussions with stakeholders—that does not fit into the standard sections of the form.



Fill in Values for the Use Case Form

Fill in elements derived from stakeholders and previous artifacts

Form Element	Description
Use Case Name	Create Reservation
Description	The Customer requests a reservation for hotel rooms for a date range. If all the requested rooms are available, the price is calculated and offered to the Customer. If details of the customer and a payment guarantee are provided, the reservation will be confirmed to the Customer.
Actor(s)	Primary: Booking Agent, Online Booker, Travel Agent System Secondary: None Note: Primary actors are proxies for the Customer.
Priority	Must have: Essential to this system



Fill in Values for the Use Case Form

Risk	High: Primarily because of the complexity of identifying if rooms are available and the number of different actor roles that can use this use case.
Trigger	A Customer wishes to make a reservation in the hotel.
Pre-conditions	At least one room exists in the hotel. Primary Actor can be identified.
Post-conditions	One reservation is added. Payment guarantee details are recorded.
Non-Functional Requirements	<i>NFR1 (Simultaneous Users)</i> <i>NFR2 (Duration of Use Case)</i> <i>NFR4 (Web Security)</i> <i>NFR6 (System Availability)</i> <i>NFR8 (Max Time for Room Availability Search)</i>
Notes	A fast method of checking room availability is still under investigation.



Fill in Values for the Main Flow of Events

Flow of Events	<p>1: Use case starts when Customer requests to create a reservation</p> <p>2: Customer enters types of rooms, arrival date, and departure date [A1] [A2]</p> <p>2.1: System creates a reservation and reserves rooms applying BR3 [A3]</p> <p>2.3: System calculates quoted price applying BR4</p> <p>2.3.1 System records quoted price</p> <p>2.4: System notifies Customer of reservation details (including rooms and price)</p> <p>3: Customer accepts rooms offered [A5]</p> <p>3.1: Extension Point (new customer) [A6]</p> <p>3.2: Extension Point (payment guarantee) [A7]</p> <p>3.3: System changes reservation status to “confirmed”</p> <p>3.4: System notifies Customer of confirmed reservation details</p> <p>4: Use case ends</p>
----------------	--



Fill in Values for the Alternate Flow of Events

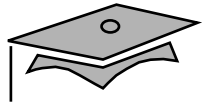
Determine the alternate flows from the secondary scenarios and remaining primary scenarios:

- Perform a *difference analysis* between the scenario used for the main flow and each of the other scenarios (in turn).
- The alternate flows are the steps that are different between the scenario used for the main flow and each of the other scenarios.



Fill in Values for the Alternate Flow of Events

Alternate Flows	<p>A1: Customer can enter duration instead of departure date, go to step 2.1 [A2]</p> <p>A2: Failed date check BR1. Notify error to Customer, go to step 2</p> <p>A3: Complying with BR2, System determines that required rooms are not available, System upgrades one or more room types, go to step 2.1[A4]</p> <p>A4: No further upgrades available. Notify message to Customer, go to step 2</p> <p>A5: Rooms offered are declined, go to step A9</p> <p>A6: Customer already exists, Customer enters customer name and zip code, System searches for matching customers, notifies Customer of matching customers, Customer selects correct customer details, go to step 3.2 [A8]</p>
-----------------	---



Fill in Values for the Alternate Flow of Events

Alternate Flows (continued)	<p>A7: Payment guarantee fails. Notify message to Customer, go to step 3.2</p> <p>A8: Existing customer not found, go to step 3.1</p> <p>A9: Reservation not confirmed, reservation deleted, use case ends</p> <p><i>At any time:</i> Customer may cancel the use case, use case ends [A9]</p> <p><i>After use case inactivity of 10 minutes:</i> use case ends [A9]</p>
--------------------------------	--



Fill in Values for the Business Rules

Business Rules (BR)	<p><i>BR1:</i> The arrival date must not be before today's date, and the departure date must be after the arrival date</p> <p><i>BR2:</i> Overbooking is not allowed</p> <p><i>BR3:</i> Reservations with assigned rooms but no payment guarantee have a status of "held"</p> <p><i>BR4:</i> The quoted price is the sum of the base price of the room types after applying BR5 and BR6</p> <p><i>BR5:</i> Seasonal Adjustment can be applied if reservation dates are applicable</p> <p><i>BR6:</i> Offer adjustments can be applied if reservation qualifies</p> <p><i>BR7:</i> Reservations with "held" status can be deleted</p> <p><i>BR8:</i> Reservations with a status of "confirmed" must be linked to a payment guarantee and a customer</p> <p><i>BR9:</i> Reservation must not exist without being linked to at least one room</p>
---------------------	--



Summary

- A Use Case scenario is written to provide a detailed description of the activities involved in one instance of the use case.
- Use Case scenarios should provide as many different situations as possible so that the whole range of activities for that use case are documented.
- Use Case scenarios provide much detail about a use case. An analysis of this detail is recorded in the Use Case form.
- The activities of a use case are distilled into Flow of Events portion of the Use Case form. Alternate flows are identified from unusual situations in one or more scenarios.



Module 5

Creating Activity Diagrams



Objectives

Upon completion of this module, you should be able to:

- Identify the essential elements of an Activity diagram
- Model a Use Case flow of events using an Activity diagram



Describing a Use Case With an Activity Diagram

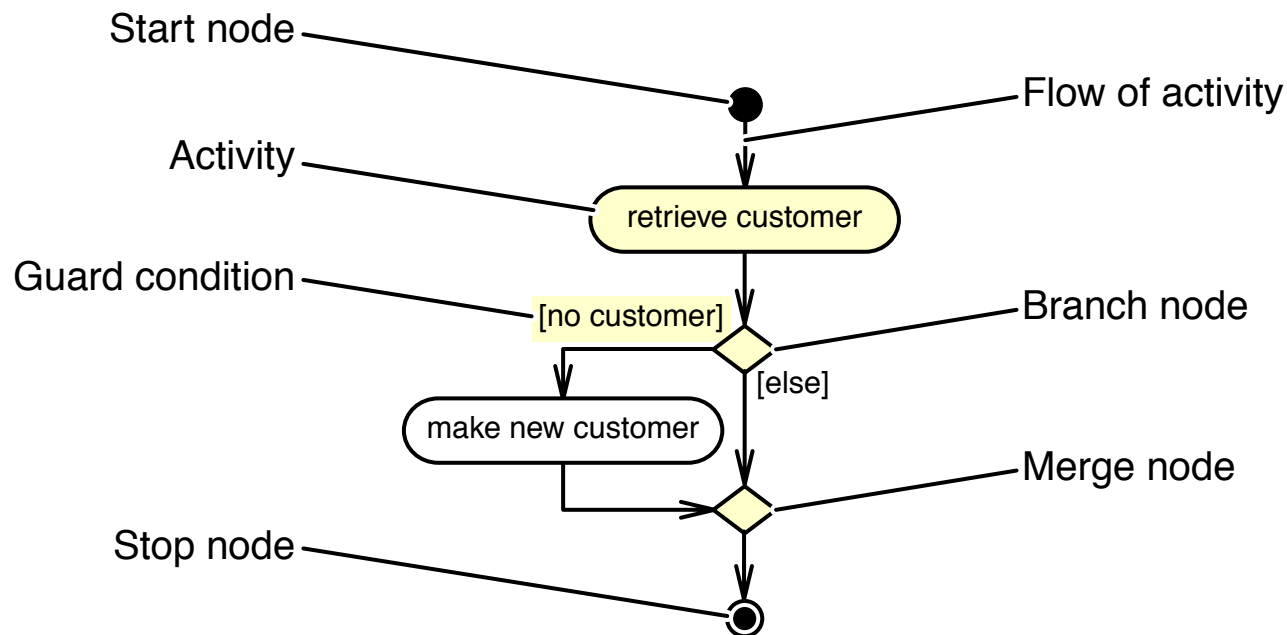
To verify a mental model of a Use Case you can:

- Model the flow of events of an Use Case in an Activity diagram
- Validate the Use Case by reviewing the Activity diagram with the stakeholders



Identifying the Elements of an Activity Diagram

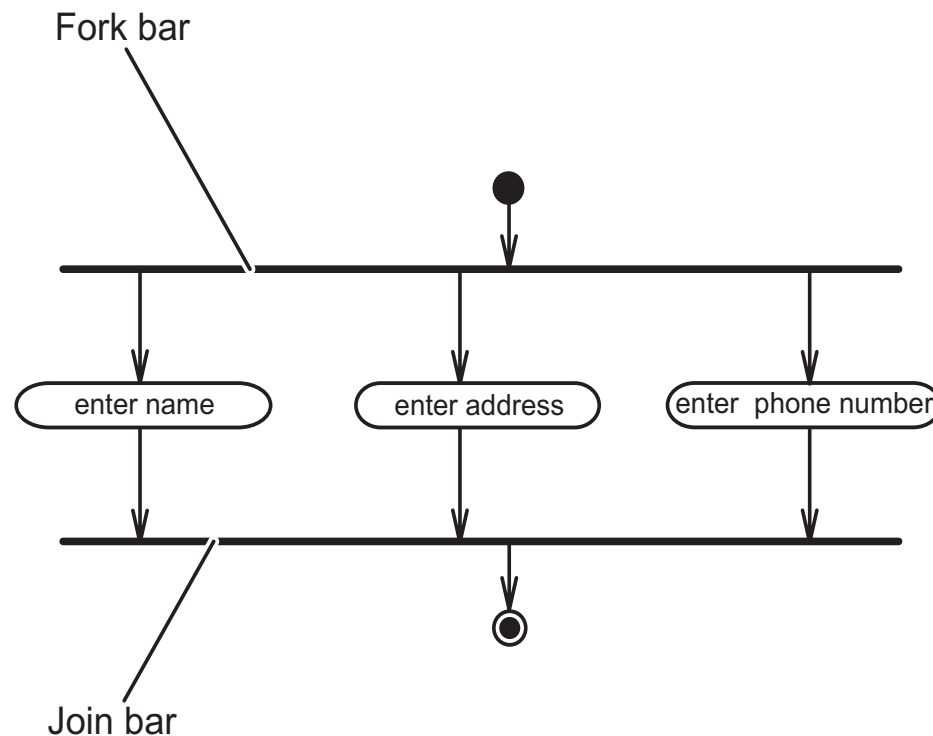
An Activity diagram is composed of the following elements:





Identifying Elements of an Activity Diagram

An example of concurrent activities:





Activities And Actions

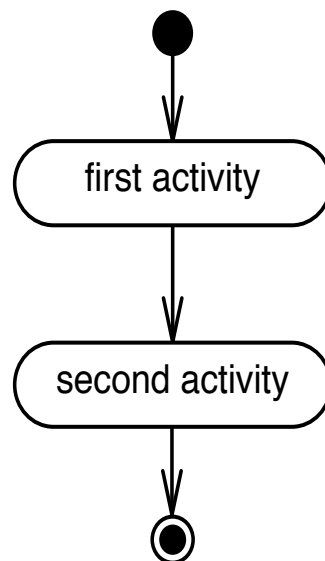
Activities and actions are processes taken by the system or an actor.

- Activity nodes and action nodes use the same notation in UML
- An activity can be divided into other activities or actions
- An action is an activity node which cannot be divided within the context of the current view.
- A primitive form of action results in a change in the state of the system or the return of a value.



Flow of Control

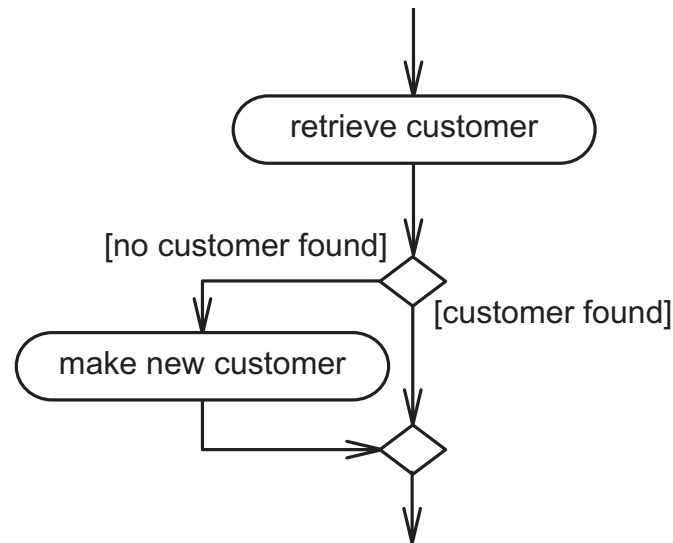
An Activity diagram must start with a Start node and end with a Stop node. Flow of control is indicated by the arrows that link the activities together.





Branching

The branch and merge nodes represent conditional flows of activity.

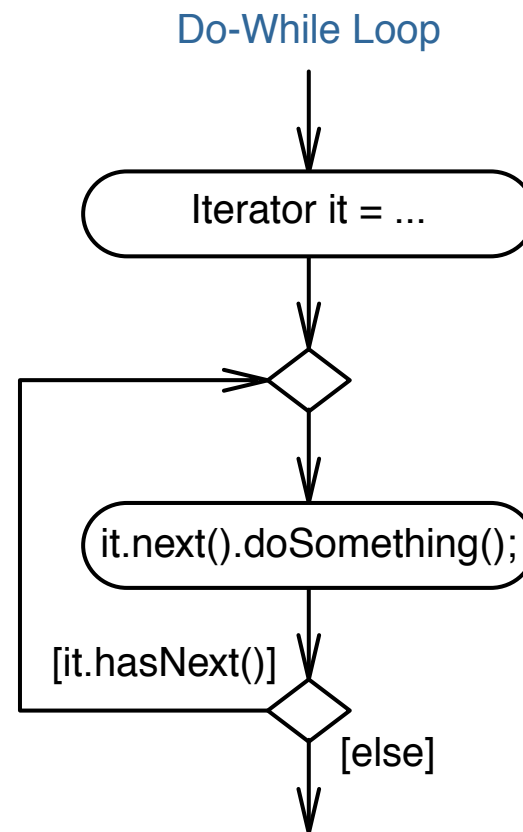
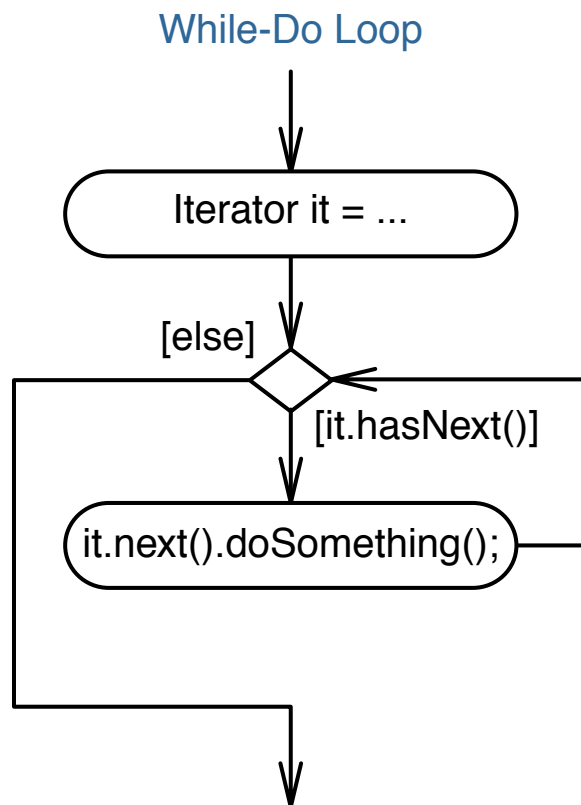


- A branch node has two or more outflows, with Boolean predicates to indicate the selection condition.
- A merge node collapses conditional branches.



Iteration

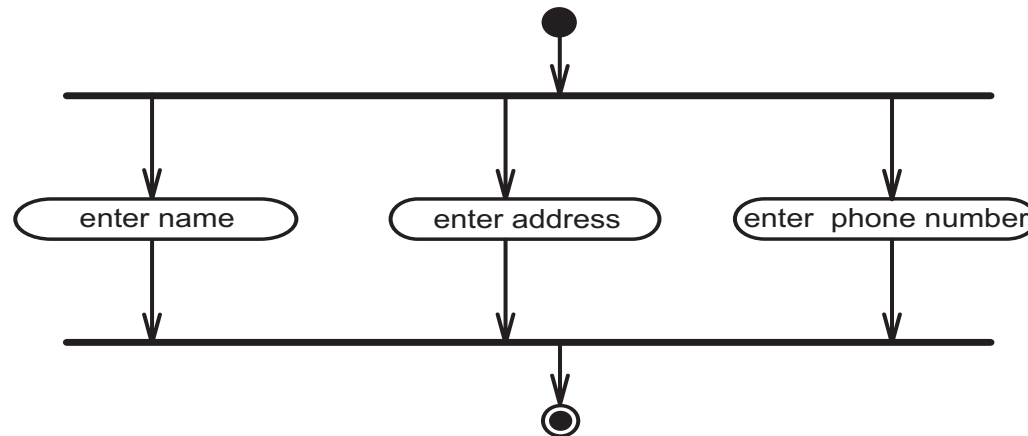
Iteration can be achieved using branch nodes.





Concurrent Flow of Control

The fork and join bars indicate concurrent flow of control.



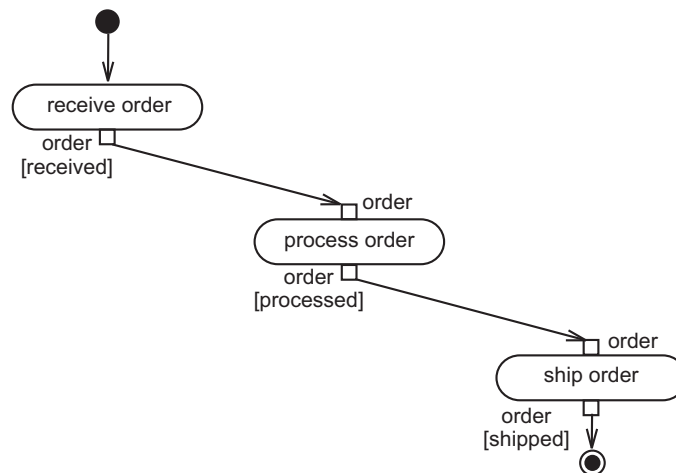
- Fork and join bars can represent either threaded activities or parallel user activities.
- The multiplicity indicator specifies how many of the parallel activities must have been processed.



Passing an Object between Actions

An Activity diagram can show objects being passed between actions

- A pin is a connection point of an action for object input or output



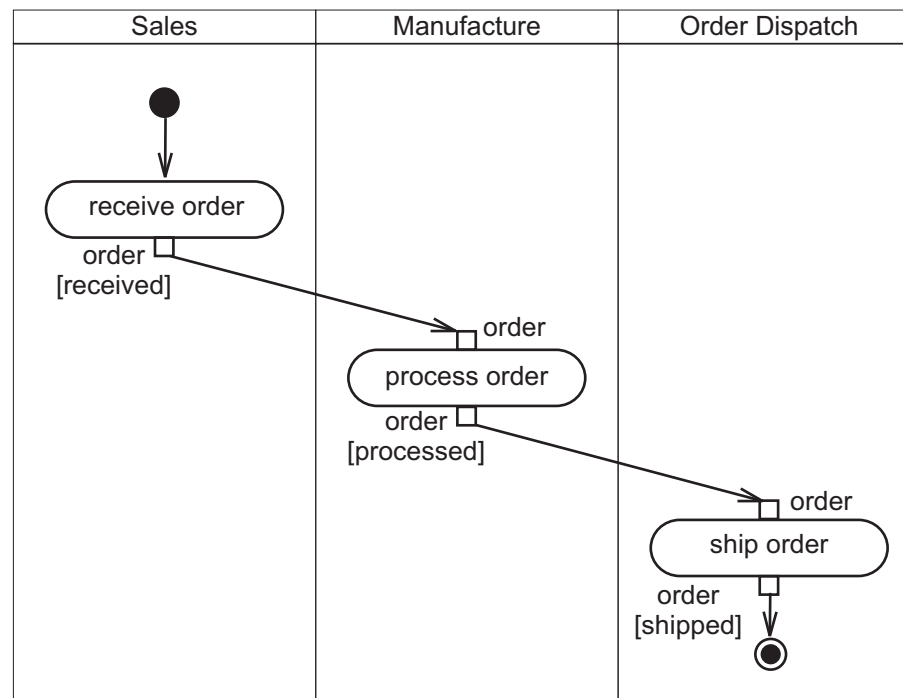
- The name of the pin denotes the object being passed



Partitions in Activity Diagrams

An Activity diagram can show objects grouped into partitions (formerly called swimlanes)

Partitions can be vertical, horizontal or both





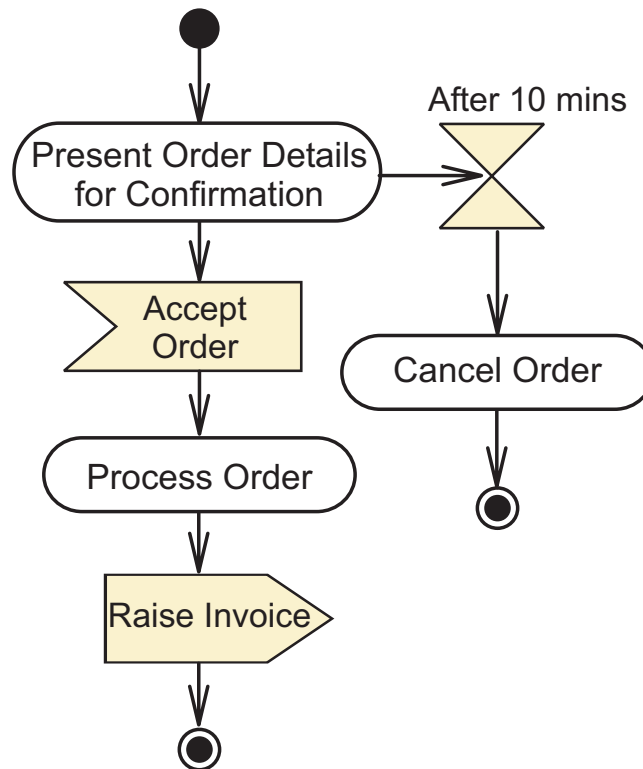
Signals in Activity Diagrams

An Activity diagram can show the receiving and sending of signals.

- An Accept Event Action element or an Accept Time Event element is used to show the receiving of a signal
- A Send Event Action element is used to show the sending of a signal



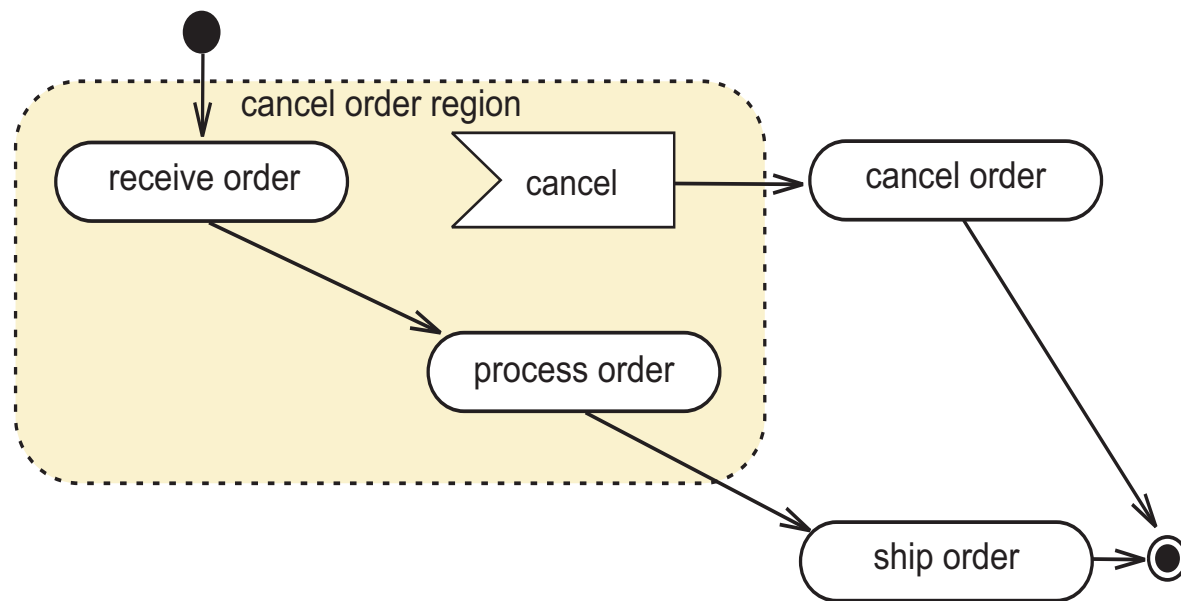
Displaying Signals in Activity Diagrams





Interruptible Activity Regions

An Activity diagram can show a sub set of activities that can be interrupted by an event.





Creating an Activity Diagram for a Use Case

Analyze the flow of events field in the Use Case form:

- Identify activities
- Identify branching and looping
- Identify concurrent activities

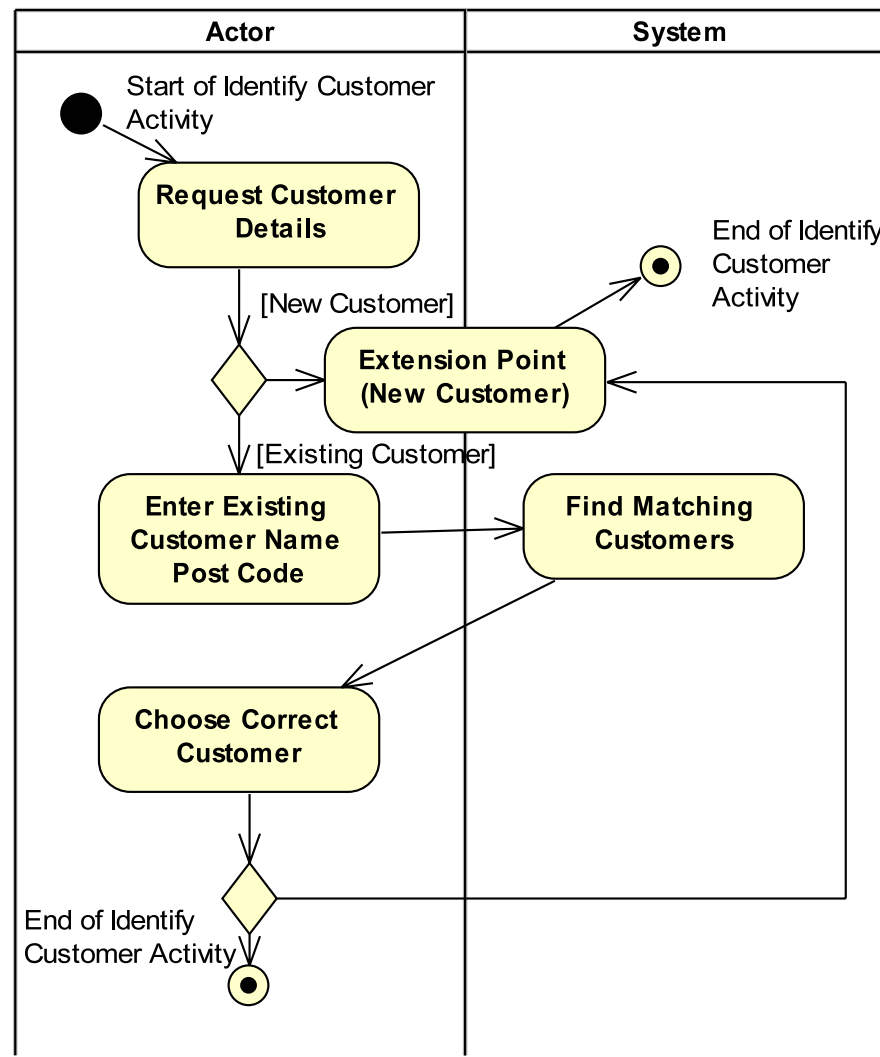


Creating Activity Diagrams – Example 1

- The following slide illustrates a simple sequence of activities for a part of the Create Reservation Use Case.
- The diagram shows the activities involved in identifying the customer:
 - by either delegating the entry of the new customer details to the extension point (New Customer)
 - or by the actor entering a subset of customer information in order to find the existing customer
- If no existing customer is found then the extension point (New Customer) is used.



Creating Activity Diagrams – Example 1



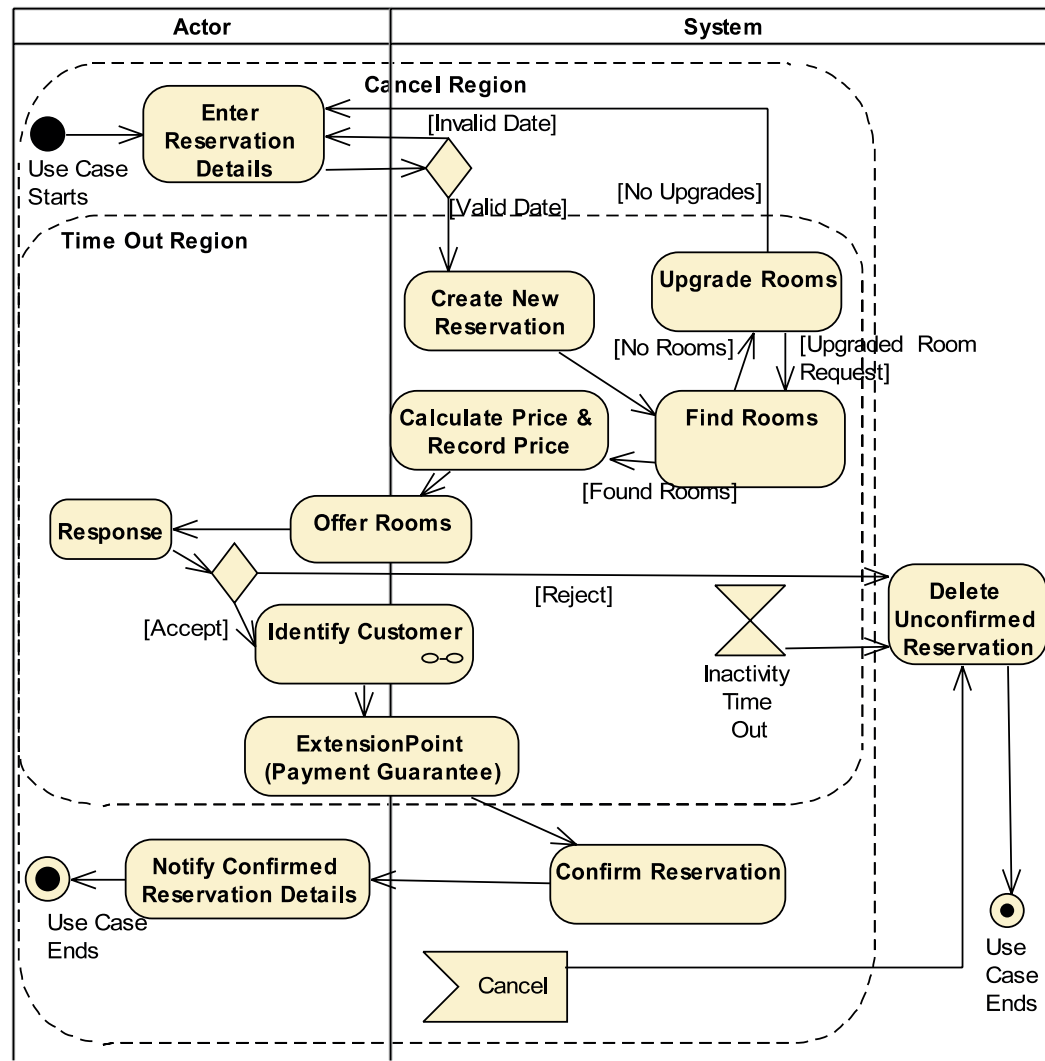


Creating Activity Diagrams – Example 2

The following slide shows an Activity diagram that represents the main flow path and the alternate flow path of the Create Reservation Use Case Form.



Creating Activity Diagrams – Example 2





Summary

In this module you identified:

- The essential elements of an Activity diagram
- How to visually represent the flow of events of a Use Case with an Activity diagram



Module 6

Determining the Key Abstractions



Objectives

Upon completion of this module, you should be able to:

- Identify a set of candidate key abstractions
- Identify the key abstractions using CRC analysis



Introducing Key Abstractions

“A key abstraction is a class or object that forms part of the vocabulary of the problem domain.” (Booch OOAD page 162)

Represents the primary objects within the system. Finding key abstractions is a process of discovery.

1. Identify all candidate key abstractions by listing all nouns from the project artifacts in a “Candidate Key Abstractions Form.”
2. Use CRC analysis to determine the essential set of key abstractions.

Key abstractions are recognized as objects that have responsibilities and are used by other objects (the collaborators).



Identifying Candidate Key Abstractions

Begin the process of identifying all of the unique nouns in the project artifacts by focusing on the following areas in these documents:

- The Main Flow and Alternate Flow sections of the use case forms
- The other sections of the use case forms
- The use case scenarios
- The Glossary of terms
- The Supplementary Specification document.

Tip: With practice you will be able to skip some of the nouns that are obviously not part of the domain.



Identifying the Candidate Abstractions

Here are a few excerpts from the Hotel System artifacts with the nouns marked in bold:

- From the Create Reservation Use Case Form Description Section:

The **Customer** requests a **reservation** for **hotel rooms** for a **date range**. If all the requested **rooms** are available, the **price** is calculated and offered to the **Customer**. If **details of the customer** and a **payment guarantee** are provided, the **reservation** will be confirmed to the **Customer**.



Identifying the Candidate Abstractions

- From the Create Reservation Use Case Form Main Flow Section:

Customer enters types of rooms, arrival date, and departure date

Systems creates a reservation and reserves rooms

System calculates quoted price

System records quoted price

System notifies Customer of reservation details (including rooms and price)

Customer accepts rooms offered

Extension Point (new customer)

Extension Point (payment guarantee)

System changes reservation status to “confirmed”

System notifies Customer of confirmed reservation details



Identifying the Candidate Abstractions

- From the Create Reservation Use Case Form Alternate Flow Section:
Customer can enter **duration** instead of **departure date**
Failed date check BR1. Notify **error** to **Customer**
Complying with BR2, **System** determines that required **rooms** are not available
System upgrades one or more **room types**
No further upgrades available. Notify **message** to **Customer**
Rooms offered are declined
Customer already exists, **Customer** enters **customer name & zip code**
System searches for matching **customers**, notifies **Customer** of matching **customers**, **Customer** selects correct **customer details**
Payment guarantee fails. Notify **message** to **Customer**
Existing **customer** not found
Reservation not confirmed, **reservation** deleted



Identifying the Candidate Abstractions

- From the Create Reservation Use Case Form Business Rules Section:

The **arrival date** must not be before **today's date**, and the **departure date** must be after the **arrival date**

Reservations with assigned **rooms** but no **payment guarantee** have a **status** of “held”

Reservations with a **status** of “confirmed” must be linked to a **payment guarantee** and a **customer Reservation** must not exist without being linked to at least one **room**



Identifying the Candidate Abstractions

- From the Create Reservation Use Case Form
Remaining Sections:
...
- From the Supplementary Specification Documents. For example the Project Glossary:

Reservation: An allocation of a specific **number of rooms**, each of a specified **room type**, for a specified **period of days**.

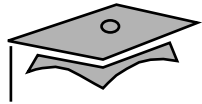
Date Range: Specifies a **start date** and an **end date**



Candidate Key Abstractions Form

The form for recording candidate key abstractions uses three fields:

- Candidate Key Abstraction – This field contains a noun discovered from the project artifacts.
- Reason For Elimination – This field is left blank if the candidate becomes a key abstraction. Otherwise, this field contains the reason why the candidate was rejected.
- Selected Name – This field contains the name of the class if this entry is selected as a key abstraction.



Candidate Key Abstractions Form (Example)

Candidate Key Abstraction	Reason for Elimination	Selected Component Name
Reservation		
Customer actor		
System		
Customer		
Room		
Date Range		
Price		
Customer Details		
...		



Project Glossary

The process of identifying candidate key abstractions is also a good opportunity to verify that your project glossary is up-to-date.

- Verify that all domain-specific terms have been listed and defined.
- Identify synonyms in the project glossary and select a primary term to use throughout the documentation and source code.



Discovering Key Abstractions Using CRC Analysis

After you have a complete list of candidate key abstractions, you need to filter this list. One technique is CRC analysis:

1. Select one candidate key abstraction.
2. Identify a use case in which this candidate is prominent.
3. Scan the use case forms, use case scenarios to determine responsibilities and collaborators.
4. Scan the Glossary for all references to the noun.
5. Document this key abstraction with a CRC card.
6. Update Candidate Key Abstractions Form based on findings.



Selecting a Key Abstraction Candidate

Selecting a good key abstraction candidate is largely intuition, but here are a few tactics:

- Ask a domain expert.
- Choose a candidate key abstraction that is used in a use case name.
- Choose a candidate key abstraction that is used in a use case form.



Selecting a Key Abstraction Candidate

The noun “reservation” appears many times in the following areas:

- In the following use case names:
 - Create *Reservation*
 - Update *Reservation*
 - Delete *Reservation*
- In many places throughout the use case forms. For example, the Check In Customer Use Case Form will describe assigning a bill to a *Reservation*



Identifying a Relevant Use Case

To determine whether the candidate key abstraction is a real key abstraction, you must determine if the candidate has any responsibilities and collaborators.

To identify a use case that might declare a candidate's responsibilities and collaborators:

1. Scan the use case names for the candidate key abstraction.
2. Scan the use case forms for the candidate key abstraction.
3. Scan the use case scenarios for the candidate key abstraction.



Identifying a Relevant Use Case - Contd.

4. Scan the text of the use case scenarios to see if the candidate key abstraction is mentioned.



Identifying a Relevant Use Case

As mentioned previously, there are three use cases that focus on the reservation key abstraction:

- Create *Reservation*
- Update *Reservation*
- Delete *Reservation*



Determining Responsibilities and Collaborators

Scan the scenarios and use case forms of the identified use cases for responsibilities (operations and attributes) of the candidate key abstraction and the objects with which it must collaborate.

If you cannot find any responsibilities, then you can reject this candidate.



Determining Responsibilities and Collaborators

Following are a few relevant artifacts:

- Glossary Term, Reservation: An allocation of a specific number of rooms, each of a specified room type, for a specified period of days
- Business Rule BR9: Reservation must not exist without being linked to at least one room
- Business Rule BR8: Reservations with a status of “confirmed” must be linked to a payment guarantee and a customer
- Main Flow 3.3: System changes reservation status to “confirmed”

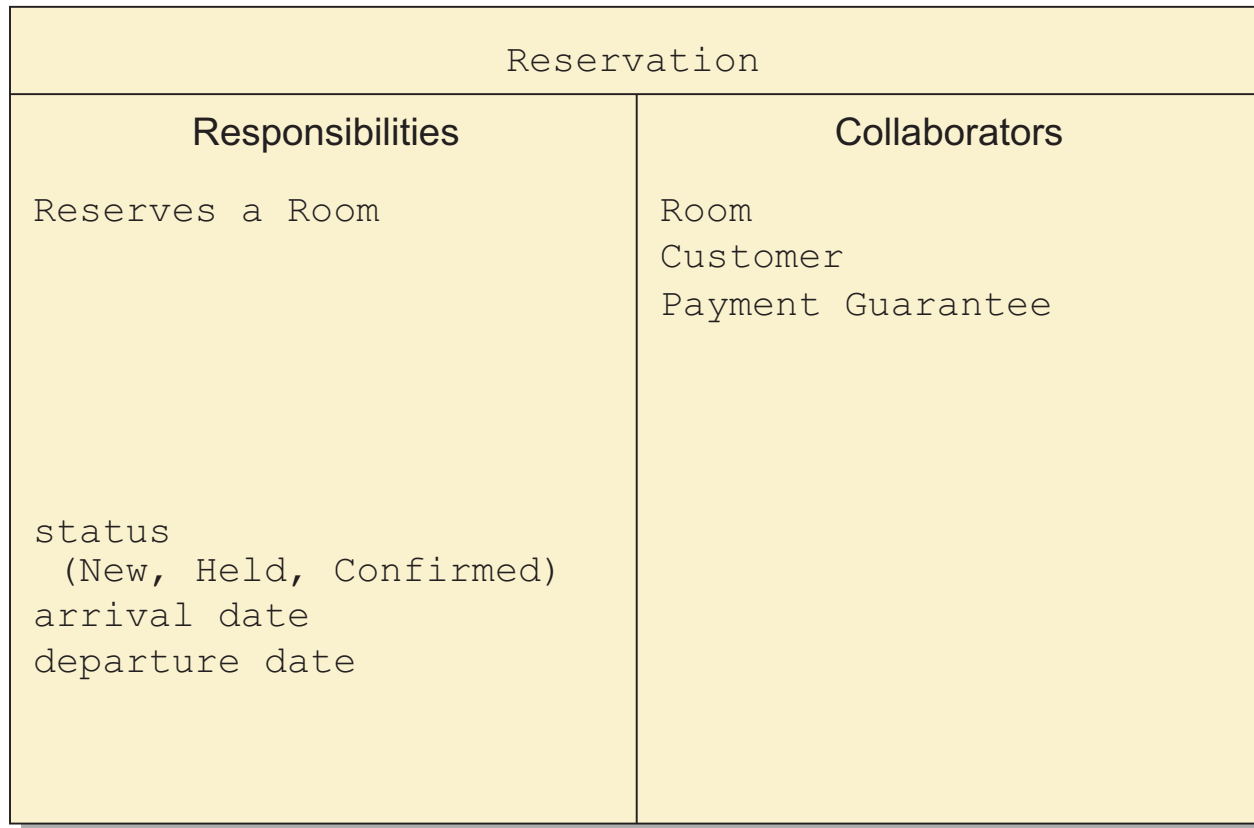


Documenting a Key Abstraction Using a CRC Card

<i>Class Name</i>	
Responsibilities	Collaborators



Documenting a Key Abstraction Using a CRC Card





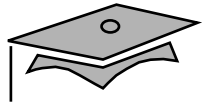
Updating the Candidate Key Abstractions Form

- If the candidate you selected has responsibilities, then enter the name of the key abstraction (from the CRC card) into the “Selected Name” field.
- Otherwise, enter an explanation why the candidate was not selected as a key abstraction.



Updating the Candidate Key Abstractions Form

Candidate Key Abstraction	Eliminated for the Following Reason	Selected Component Name
Reservation		Reservation
Customer actor	External to the system	
System	The whole system	
Customer		Customer
Rooms		Room
Date Range	A synonym for Arr. and Dept. Date	
Price	A synonym for Quoted Price	
Customer Details	Same as Customer	



Updating the Candidate Key Abstractions Form

Candidate Key Abstraction	Eliminated for the Following Reason	Selected Component Name
Payment Guarantee		Payment Guarantee
Room Type		RoomType
Arrival Date	Attribute of Reservation	
Departure Date	Attribute of Reservation	
Quoted Price	Attribute of Reservation	
Reservation Details	Same as Reservation	
Customer Name	Attribute of Customer	
Customer Zip Code	Attribute of Customer	
Today Date	External to the system	
Period of Days	A synonym for Duration	



Summary

- Key abstractions are the essential nouns in the language of the problem domain.
- To identify the key abstractions:
 - a. List all (problem domain) nouns from the project analysis artifacts, in a Candidate Key Abstractions Form.
 - b. Use CRC analysis to identify the key abstractions (a class with responsibilities and collaborators) from the candidate list.



Module 7

Constructing the Problem Domain Model



Objectives

Upon completion of this module, you should be able to:

- Identify the essential elements in a UML Class diagram
- Construct a Domain model using a Class diagram
- Identify the essential elements in a UML Object diagram
- Validate the Domain model with one or more Object diagrams



Introducing the Domain Model

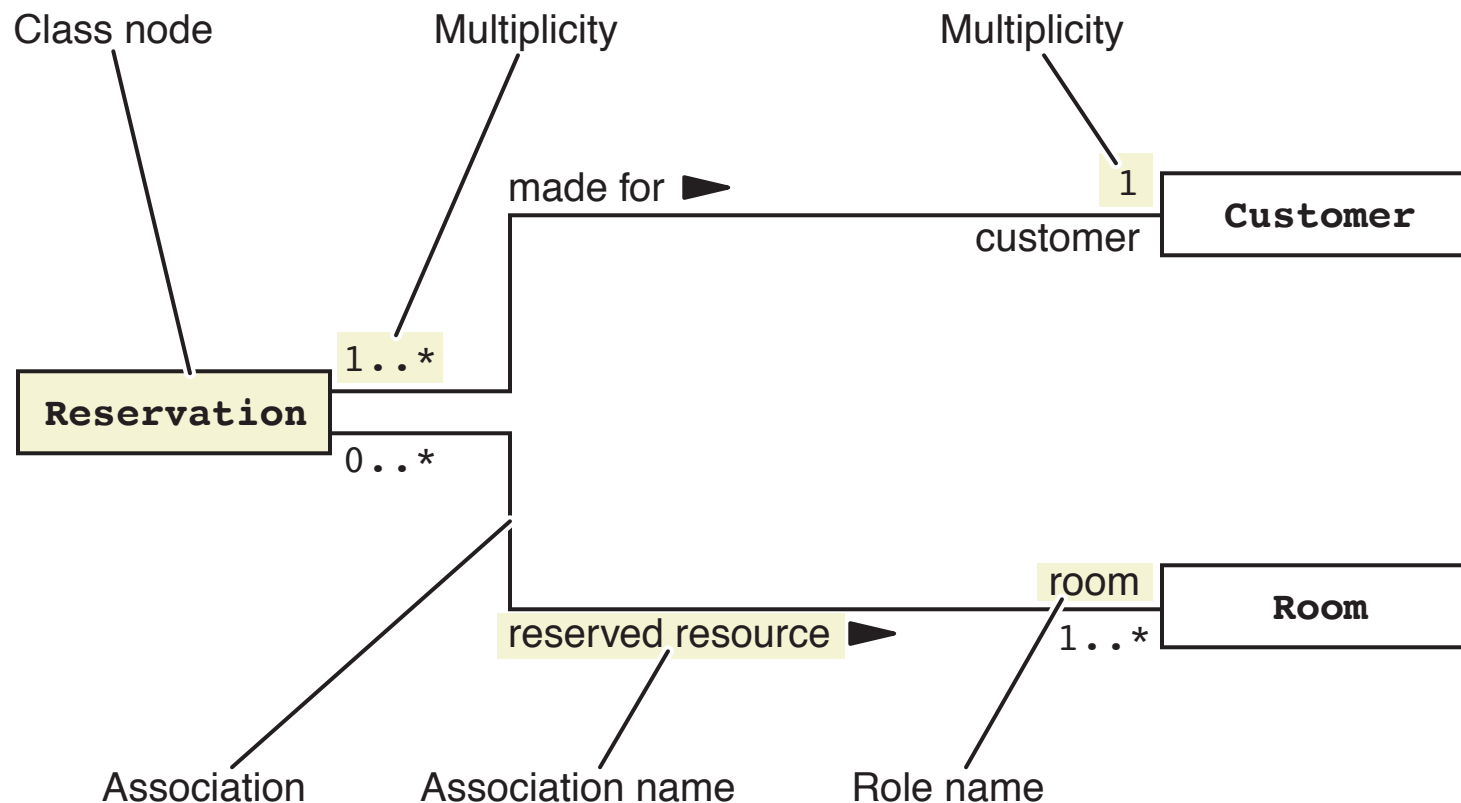
Domain model – “The sea of classes in a system that serves to capture the vocabulary of the problem space; also known as a conceptual model.” (Booch Object Solutions page 304)

- The classes in the Domain model are the system’s key abstractions.
- The Domain model shows relationships (collaborators) between the key abstractions.



Identifying the Elements of a Class Diagram

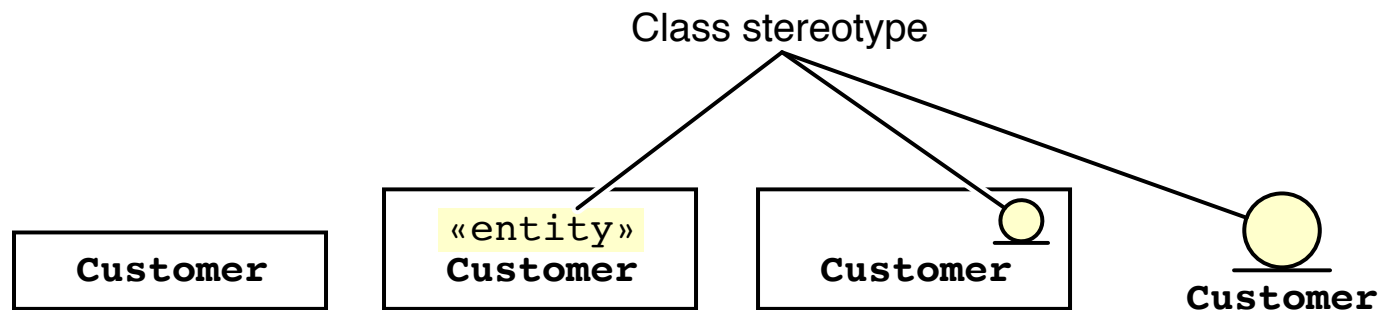
A UML Class diagram is composed of the following elements:





Class Nodes

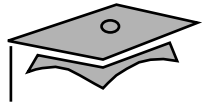
Class nodes represent classes of objects within the model.



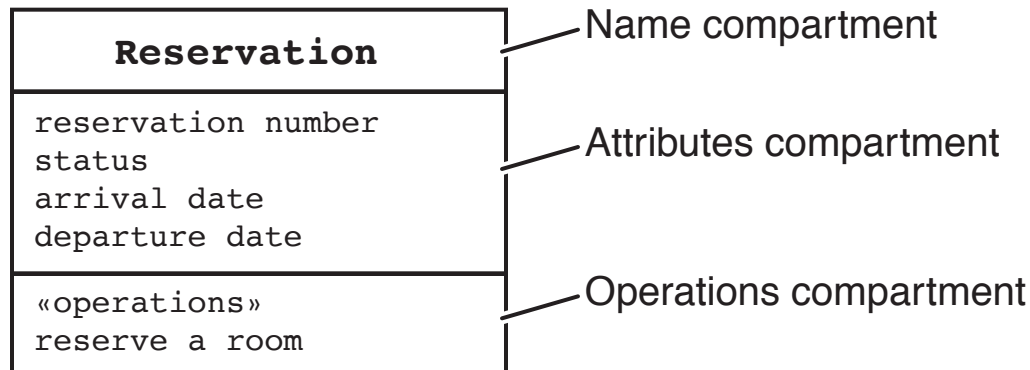
These can represent:

- Conceptual entities, such as key abstractions
- Real software components

A stereotype can help identify the type of the class node.



Class Node Compartments



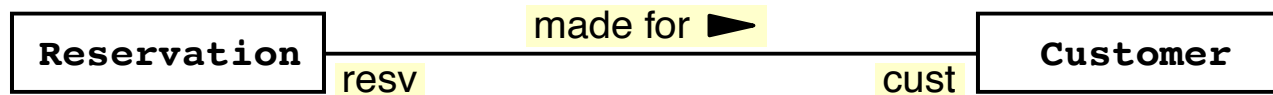
- The name compartment records the name of the class.
- The attributes compartment records attributes (or instance variables) of the class.
- The operations compartment records operations (or methods) of the class.
- Additional compartments may be added.



Associations

Associations represent relationships between classes. Associations are manifested at runtime, but these models represent all possible runtime arrangements between objects.

Relationship and Roles



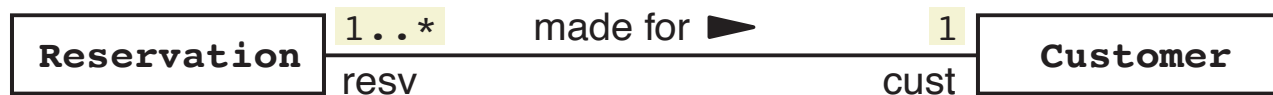
This association would be read as “A reservation is made for a customer.”



Multiplicity

Multiplicity determines how many objects might participate in the relationship.

For example:



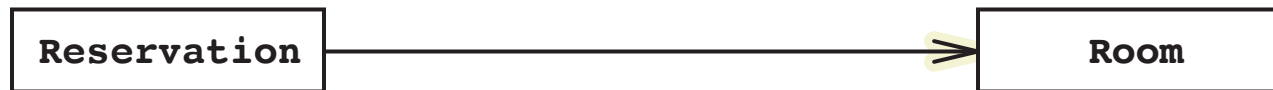
This association would be read as “A reservation is made for one and only one customer.” Reading it in the other direction is “A customer can make one or more reservations.”



Navigation

Navigation arrows on the association determine what direction an association can be traversed at runtime.

For example:

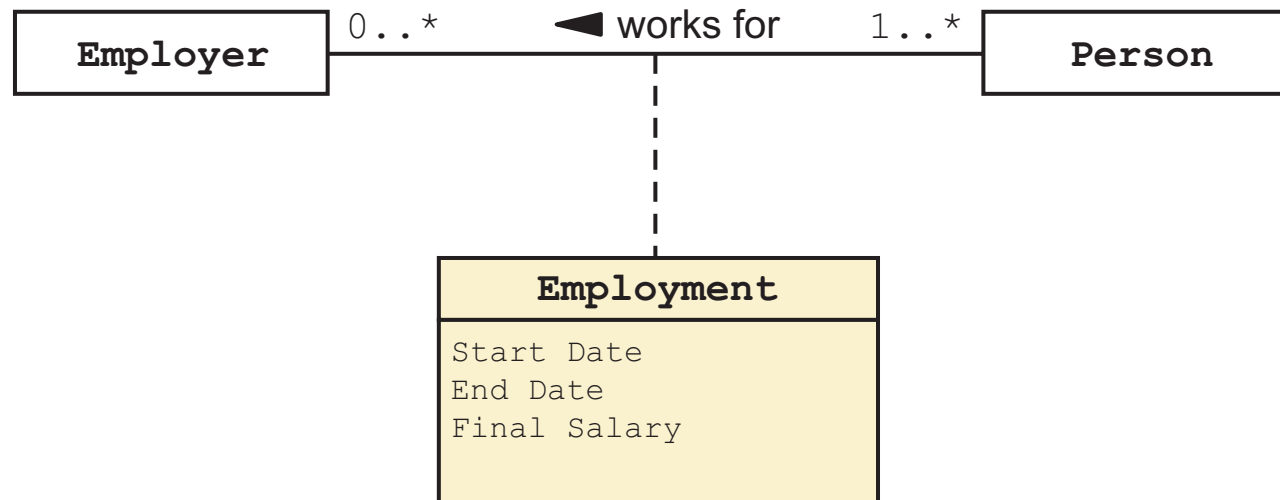


This association would be read as “From a reservation the system *can directly* retrieve the room, and from a room the system *cannot directly* retrieve the reservation for that room.”



Association Classes

Sometimes information is included in the association between two classes. For example:



Employment is an association class that records the employment details for each term of employment between a person and an employer.



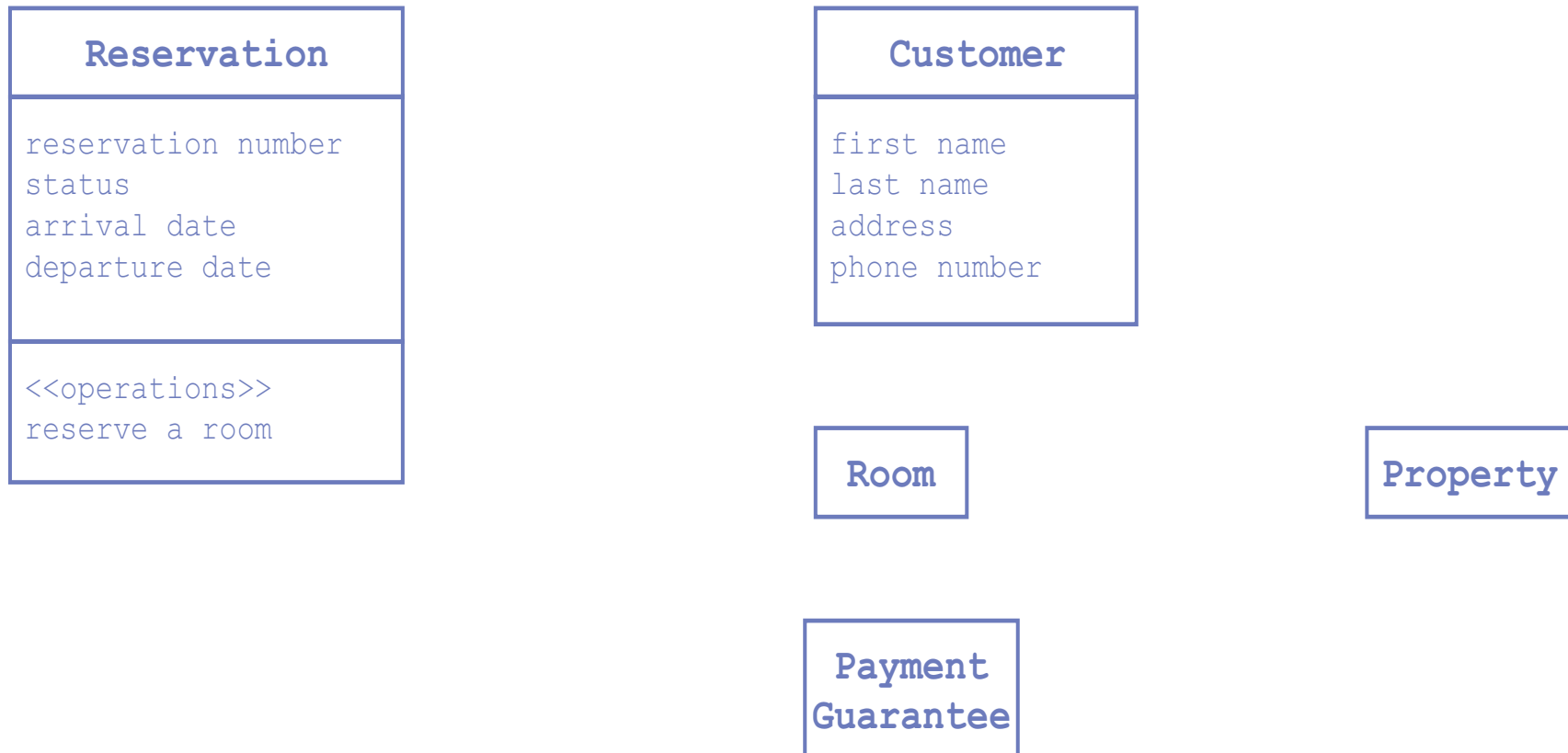
Creating a Domain Model

Starting with the key abstractions, you can create a Domain model using these steps:

1. Draw a class node for each key abstraction, and:
 - a. List known attributes.
 - b. List known operations.
2. Draw associations between collaborating classes.
3. Identify and document relationship and role names.
4. Identify and document association multiplicity.
5. Optionally, identify and document association navigation.

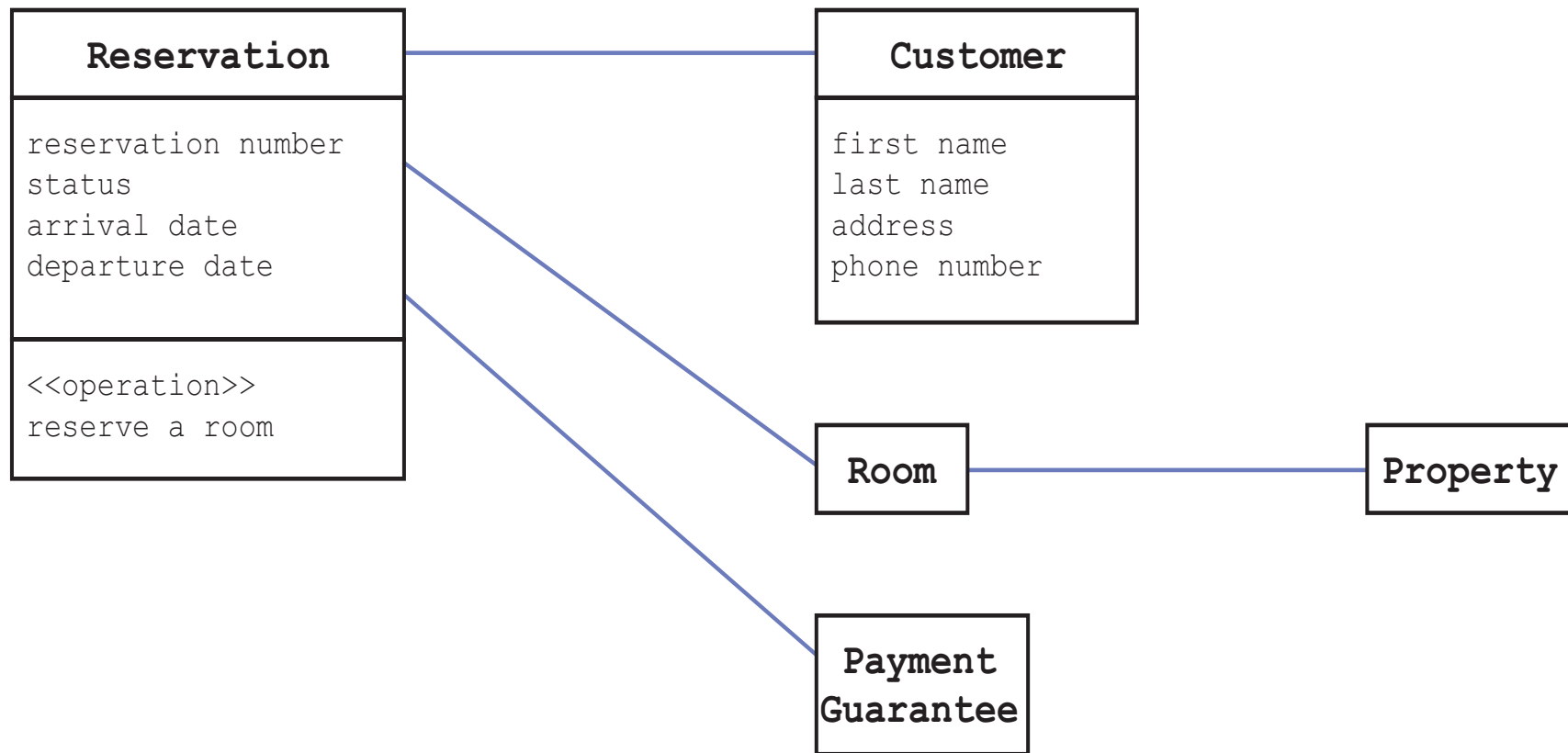


Step 1 – Draw the Class Nodes



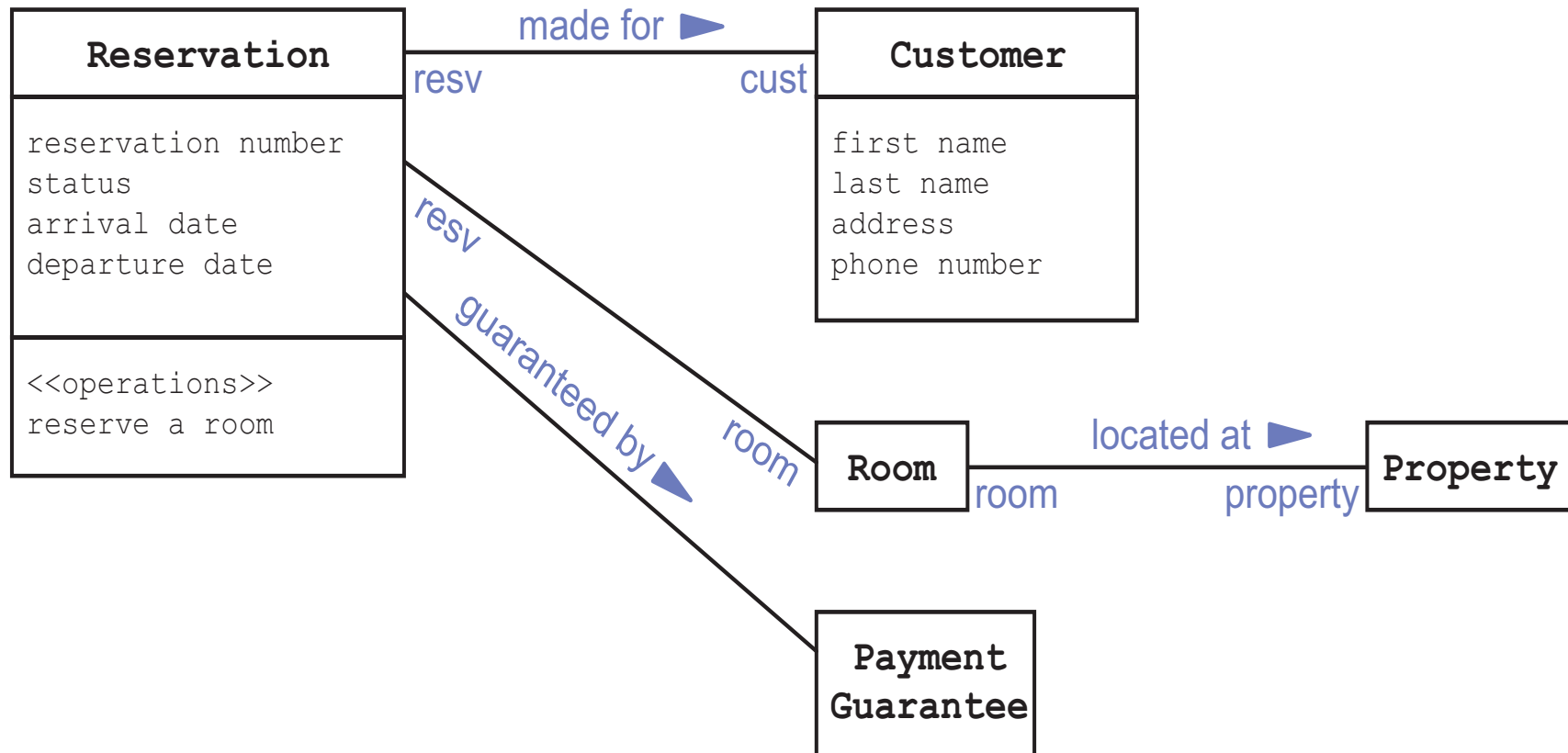


Step 2 – Draw the Associations



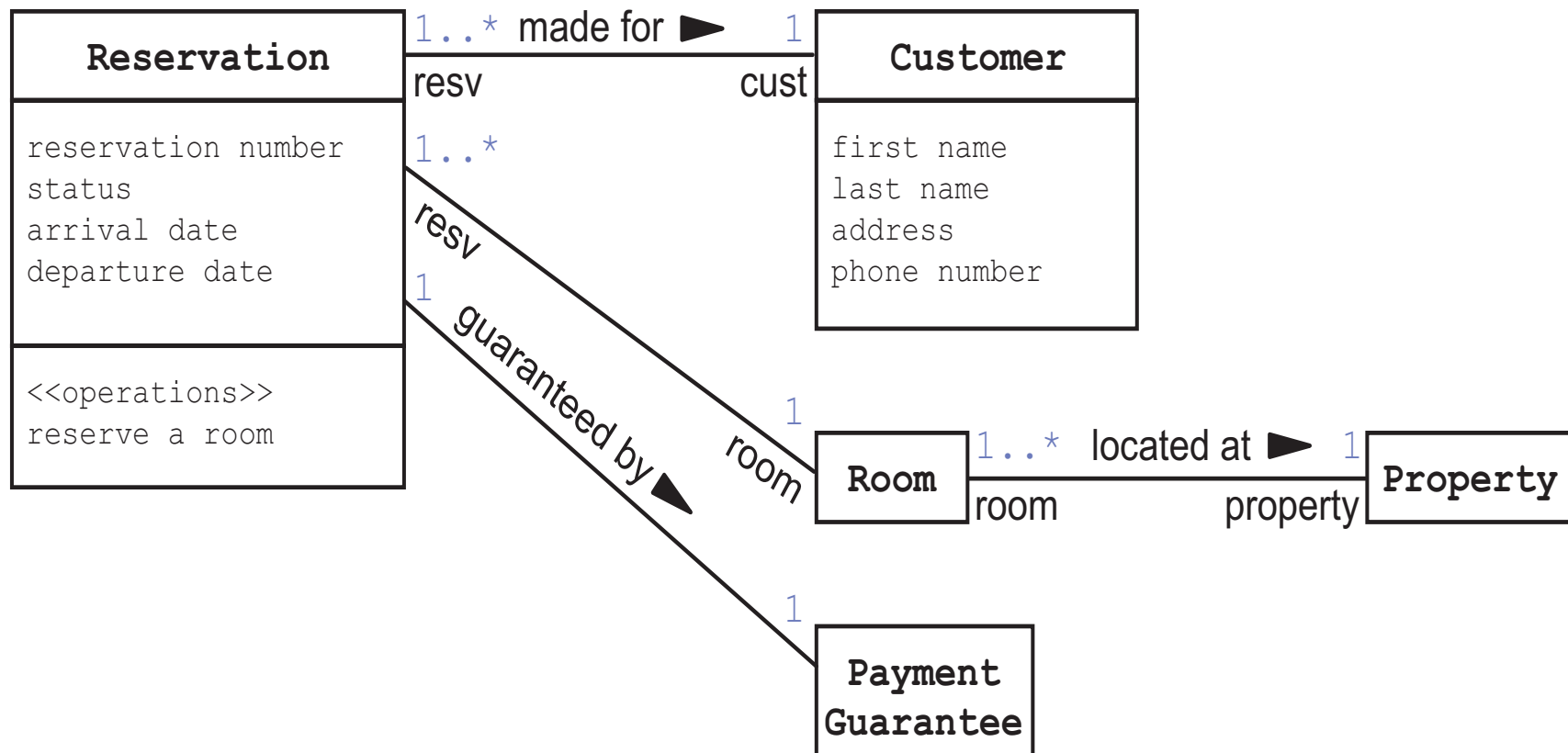


Step 3 – Label the Associations and Role Names



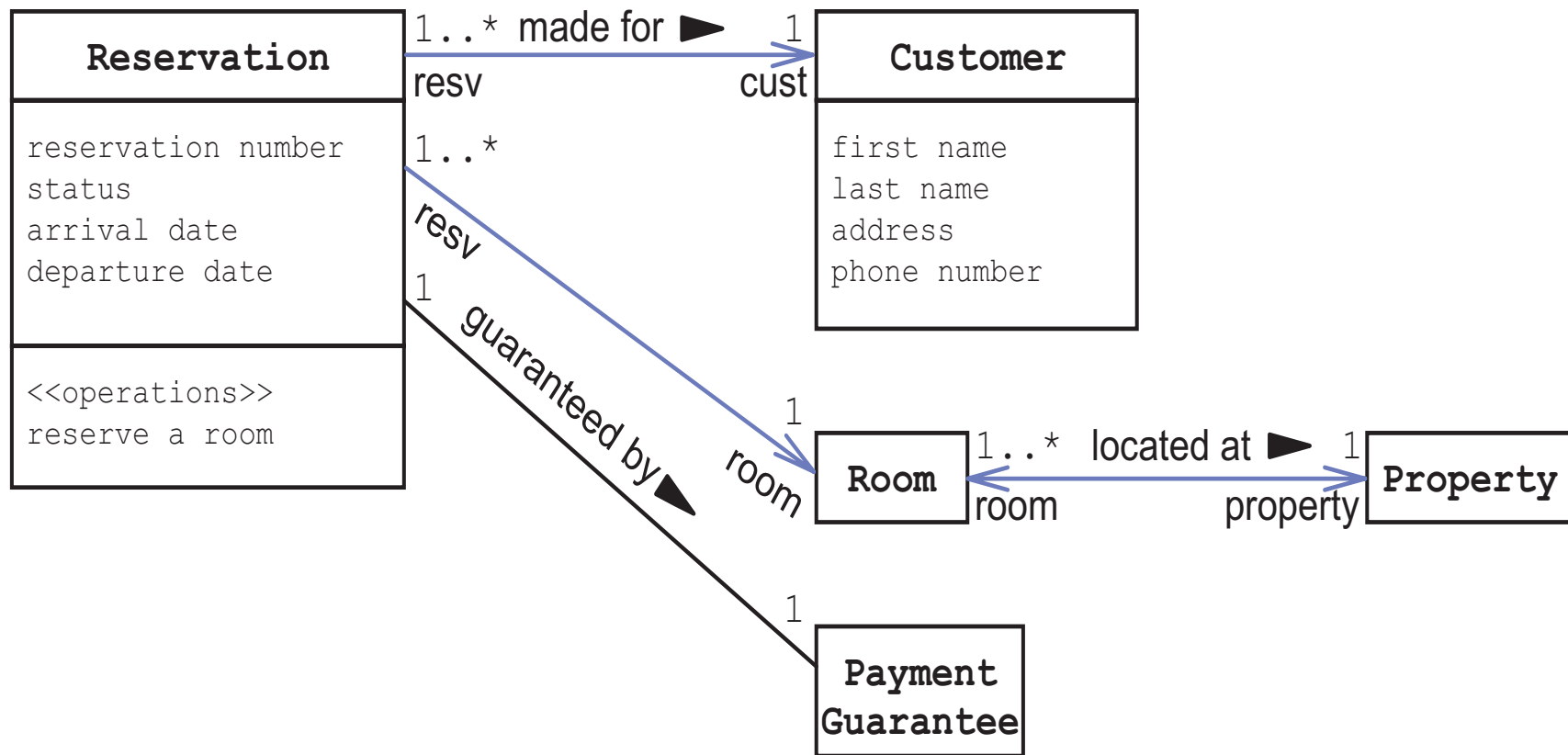


Step 4 – Label the Association Multiplicity





Step 5 – Draw the Navigation Arrows





Validating the Domain Model (Intro)

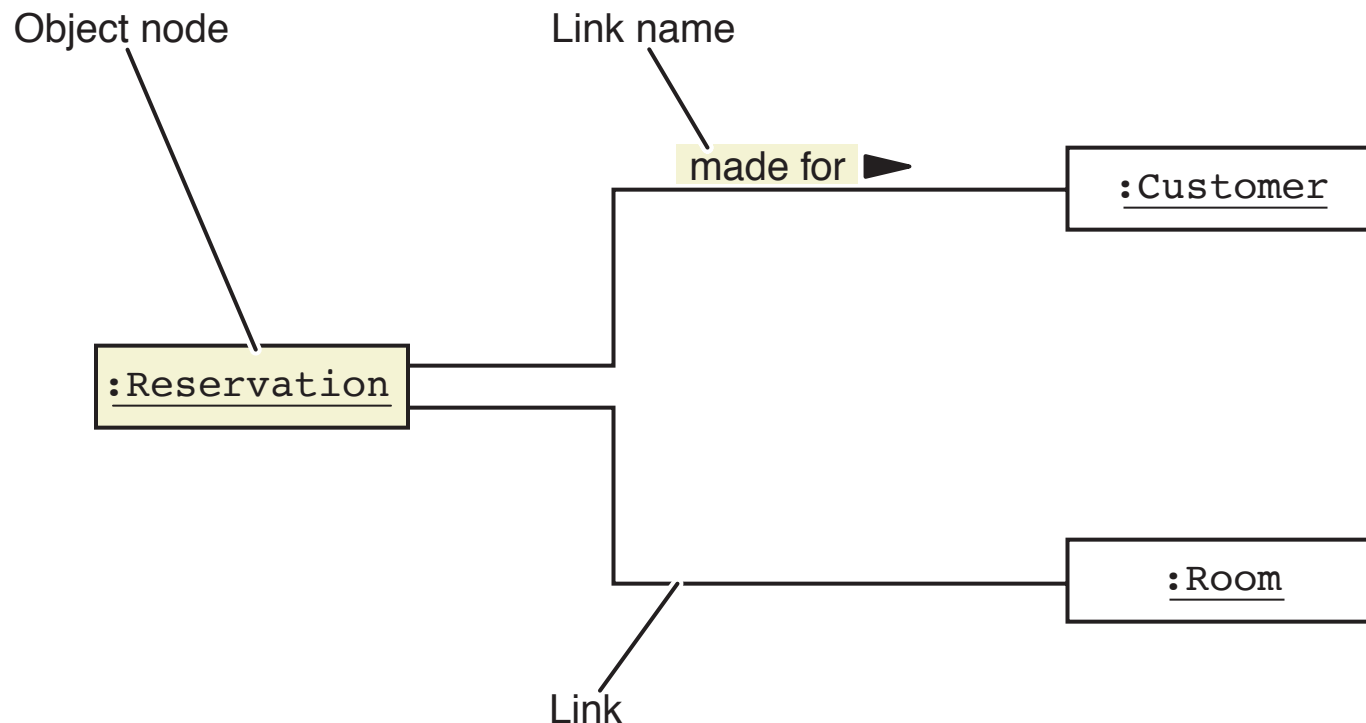
You can validate the Domain model by analyzing multiple Object diagrams based on use case scenarios.

First, the essential elements of Object diagrams are presented.



Identifying the Elements of an Object Diagram

A static object diagram is an instance of a class diagram; it shows a snapshot of the detailed state of a system at a point in time.
[UML specv1.4, page 3-35]





Object Nodes

An object node includes some form of name and data type:

Object name without type

Victoria

Type without a name

:Room

Type with a name

Blue:Room

An object node might also include attributes:

:Customer

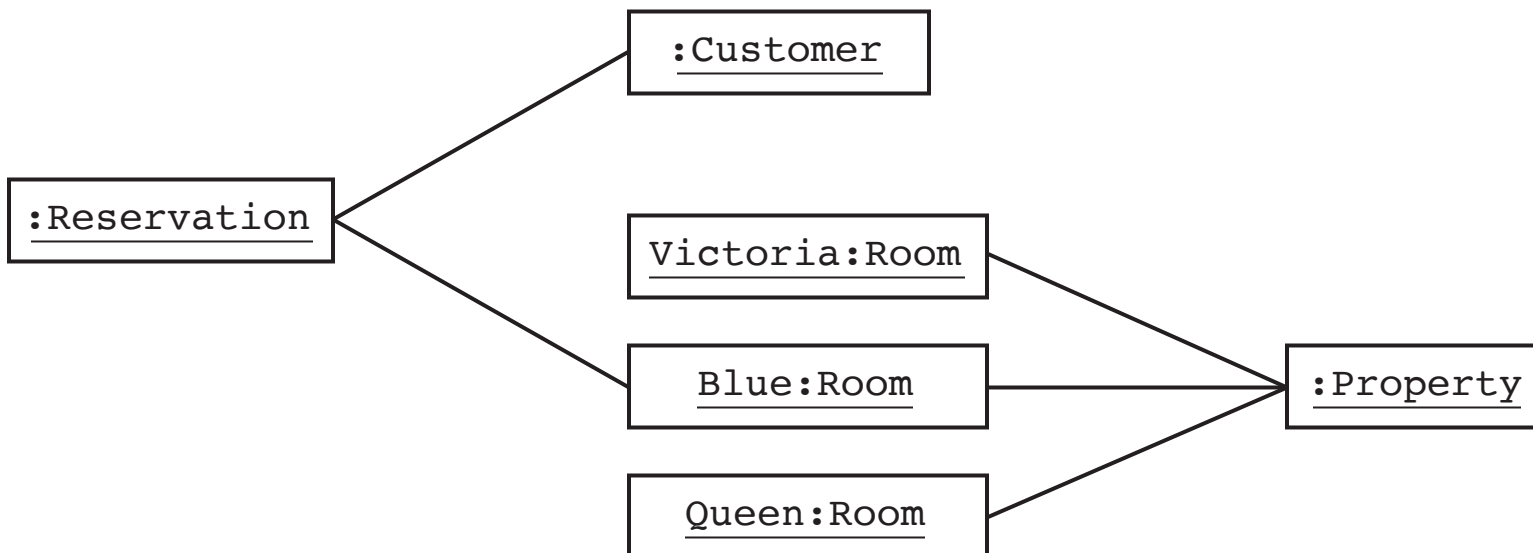
```
first name = "Jane"  
last name = "Googol"  
address = "2 Main St, ..."  
phone number = "999-555-4747"
```



Links

In Object diagrams each link is unique and is one-to-one with respect to the participants.

For example:





Validating the Domain Model Using Object Diagrams

1. Pick one or more use cases that exercise the Domain model.
2. Pick one or more use case scenarios for the selected use cases.
3. Walk through each scenario (separately), and construct the objects (with data) mentioned in the scenario.
4. Compare each Object diagram against the Domain model to see if any association constraints are violated.



Step 1 – Create Reservation Scenario 1

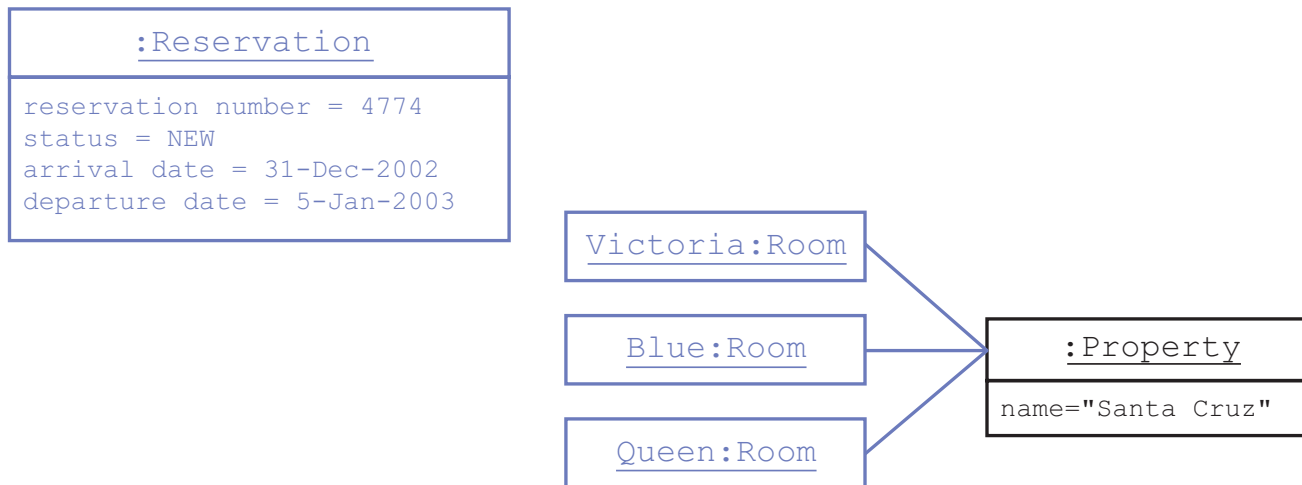
The use case begins when the booking agent receives a request to make a reservation for rooms in the hotel. The booking agent enters the arrival date, the departure date, and the quantity of each type of room that is required.





Step 2 – Create Reservation Scenario 1

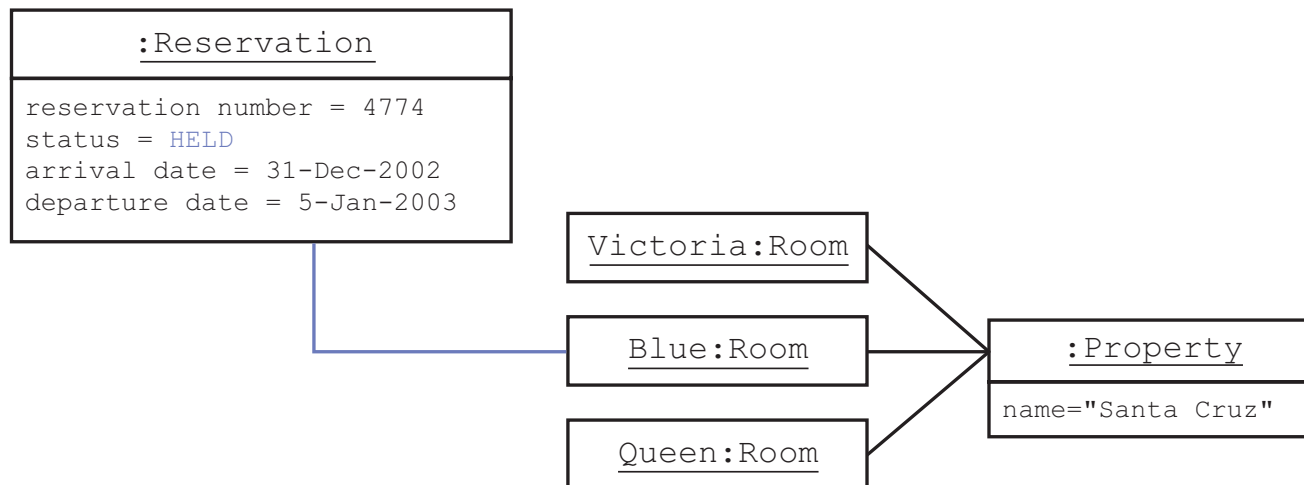
The booking agent then submits the entered details. The system finds rooms that will be available during the period of the reservation.





Step 3 – Create Reservation Scenario 1

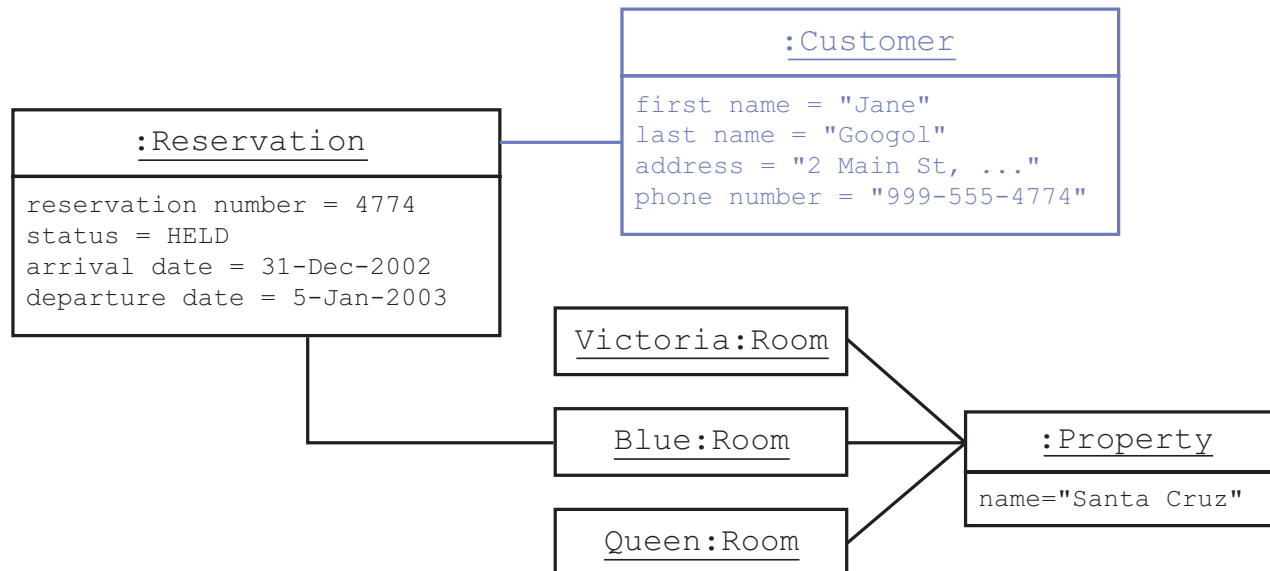
The system allocates the required number and type of rooms from the available rooms. The system responds that the specified rooms are available, returns the provisional reservation number, and marks the reservation as “held”.





Step 4 – Create Reservation Scenario 1

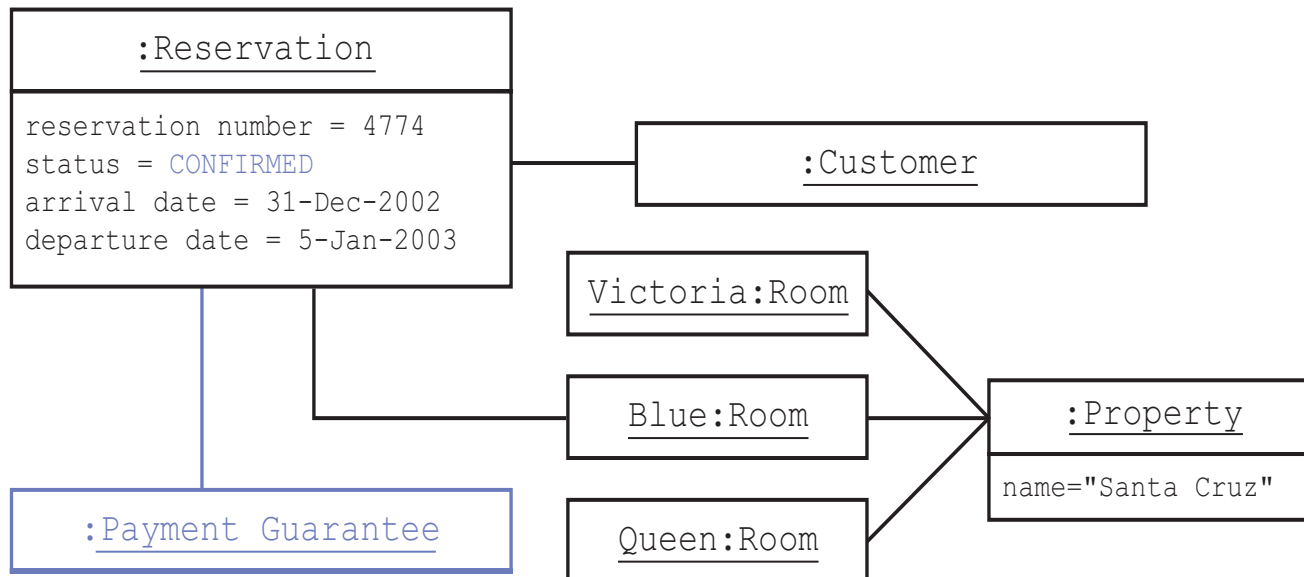
The booking agent accepts the rooms offered. The booking agent selects that the customer has visited one of the hotels in this group before, and enters the zip code and customer name. The system finds and returns a list of matching customers with full address details. The booking agent selects one of the customers as being the valid customer. The system assigns this customer to the reservation.





Step 5 – Create Reservation Scenario 1

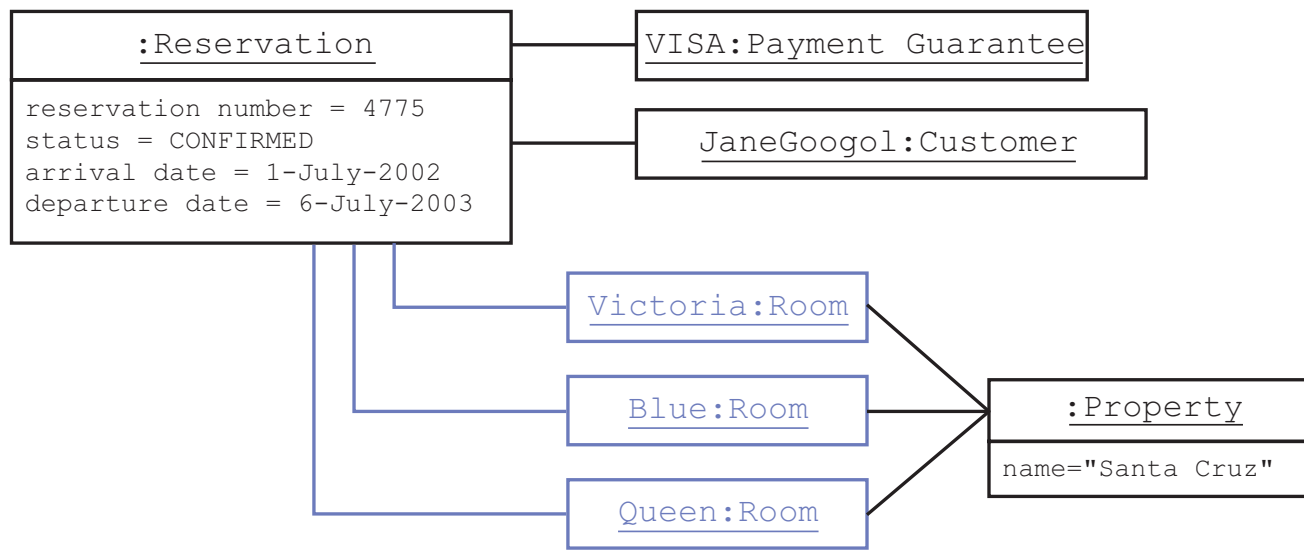
The booking agent performs a payment guarantee check. This check is successful. The system assigns the payment guarantee to the reservation and changes the state of the reservation to “confirmed”. The system returns the reservation ID and booking details.





Create Reservation Scenario No. 2

Another “Create a Reservation” scenario has the Actor making a reservation for a small family reunion in which three rooms are booked:





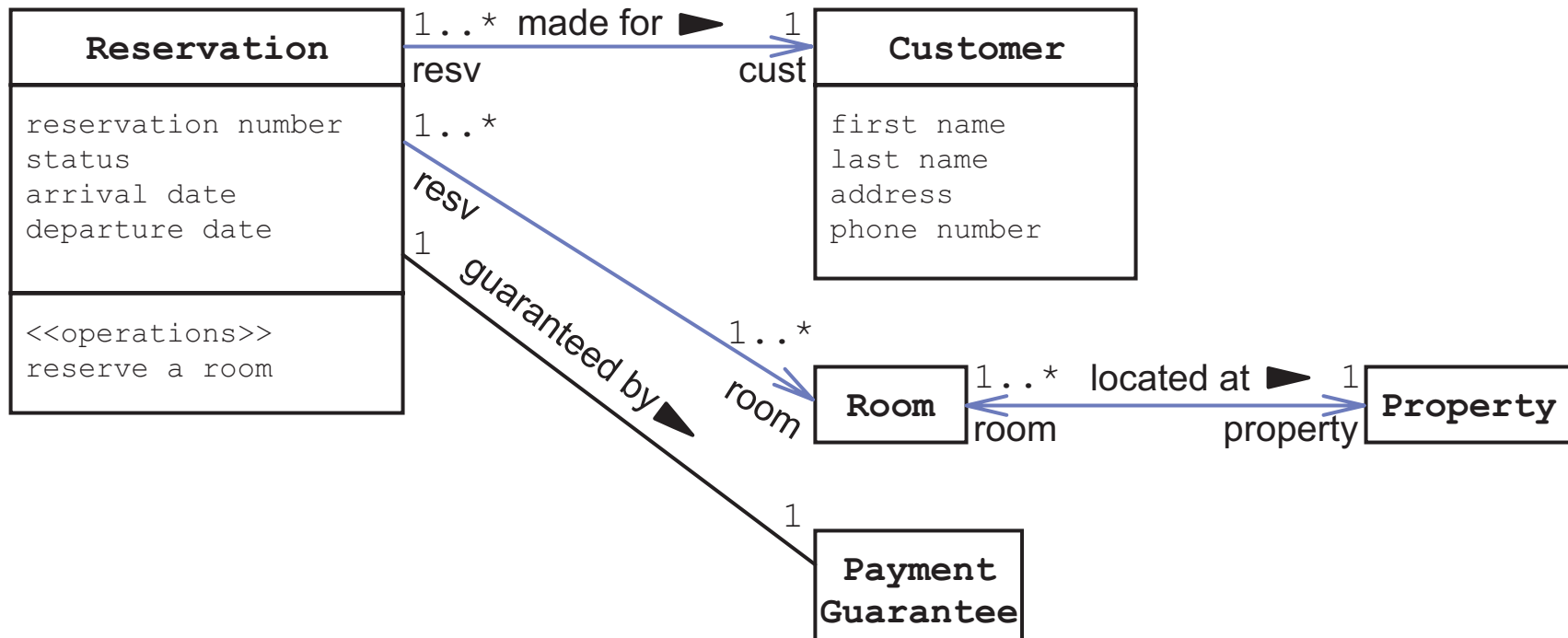
Comparing Object Diagrams to Validating the Domain Model

To validate the Domain model, compare the Class diagram with the scenario Object diagrams.

- Are there attributes or responsibilities mentioned in a scenario that are not listed in the Domain model?
- Are there associations in the Object diagrams that do not exist in the Domain model?
- Are there scenarios in which the multiplicity of a relationship is wrong?



Revised Domain Model for the Hotel Reservation System





Summary

- Use the Domain model to provide a static view of the key abstractions for the problem domain.
- Use the UML Class diagrams to represent the Domain model.
- Validate the Domain model by creating Object diagrams from use case scenarios to see if the network of objects fits the association constraints specified by the Domain model.



Module 8

Transitioning from Analysis to Design Using Interaction Diagrams



Objectives

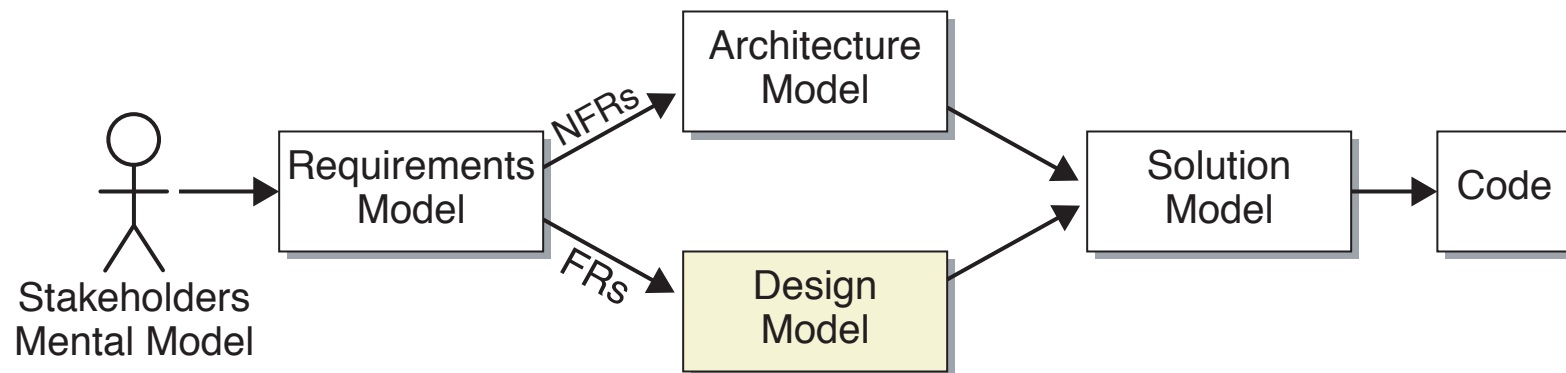
- Explain the purpose and elements of the Design model
- Identify the essential elements of a UML Communication diagram
- Create a Communication diagram view of the Design model
- Identify the essential elements of a UML Sequence diagram
- Create a Sequence diagram view of the Design model



Introducing the Design Model

The Design model is created from the Requirements model (use cases and Domain model).

The Design model is merged with the Architecture model to produce the Solution model.





Interaction Diagrams

UML Interaction diagrams are the collective name for the following diagrams:

- Sequence diagrams
- Communication diagrams
Formerly known as Collaboration diagrams
- Interaction Overview diagrams
A combination of Activity diagram and Sequence diagram fragments



Interaction Diagrams

Each UML Interaction diagram is used to show the sequence of interactions that occur between objects during:

- One or two use case scenarios
- A fragment of one use case scenario

UML Interaction diagrams may also be used to show the sequence of interactions that occur between:

- Systems
- Subsystems



Comparing Analysis and Design

Analysis helps you model *what* is known about a business process that the system must support:

- Use cases
- Domain model

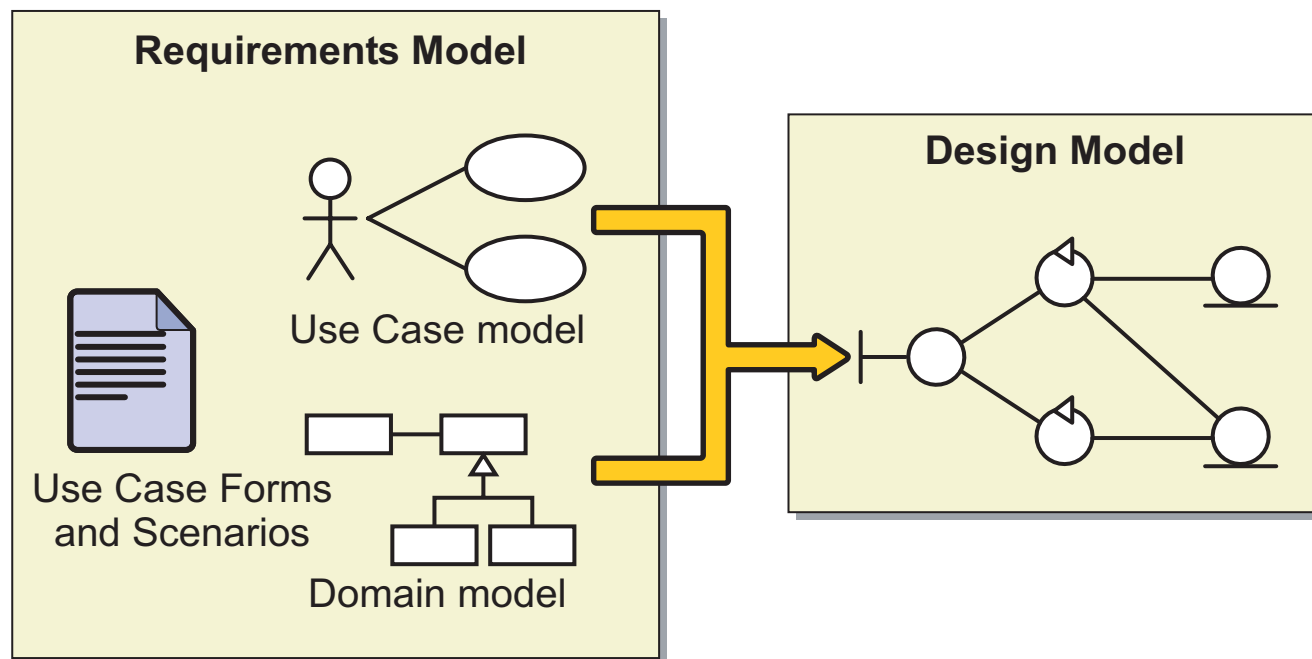
Design helps you model *how* the system will support the business processes. The Design model consists of:

- Boundary (UI) components
- Service components
- Entity components



Robustness Analysis

Robustness analysis is a process that assists in identifying design components that would be required in the Design model:





Robustness Analysis

Inputs to Robustness Analysis:

- A use case
- The use case scenarios for that use case
- The use case Activity diagram (if available) for that use case
- The Domain model

Output from Robustness Analysis:

The Design model is usually captured in UML Interaction diagrams with design components such as Boundary, Service, and Entity components.



Boundary Components

“A boundary class (component) is used to model interaction between the system and its actors (that is, users and external systems).” (Jacobson, Booch, and Rumbaugh page 183)



- Abstractions of UI screens, sensors, communication interfaces, and so on.
- High-level UI components.
- Every boundary component must be associated with at least one actor.



Service Components

“Control (Service) classes (components) represent coordination, sequencing, transactions, and control of other objects and are often used to encapsulate control related to a specific use case.”
(Jacobson, Booch, and Rumbaugh page 185)

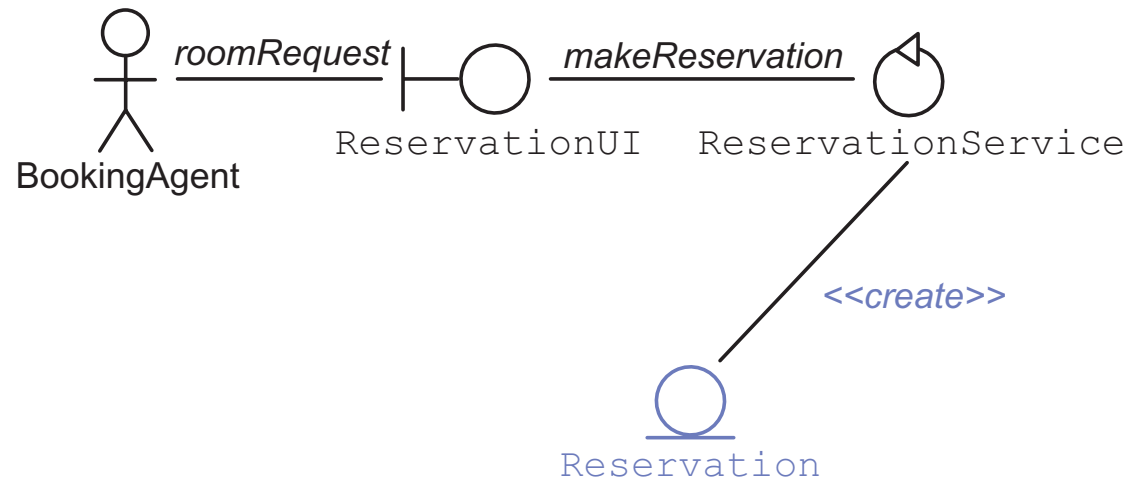


- Coordinate control flow
- Isolate any changes in workflow from the boundary and entity components



Entity Components

“An entity class (component) is used to model information that is long-lived and often persistent.” (Jacobson, Booch, and Rumbaugh page 184)

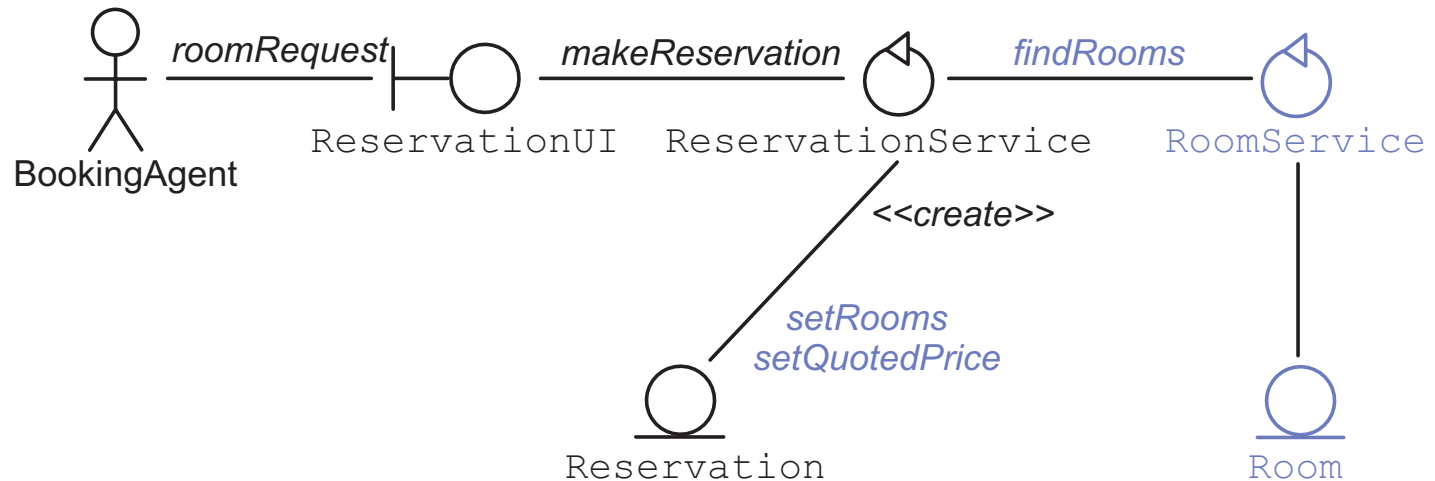


- Entities usually correspond to domain objects.
- Most entities are persistent.
- Entities can have complex behavior.



Service and Entity Components

An Entity component will often have a corresponding Service component

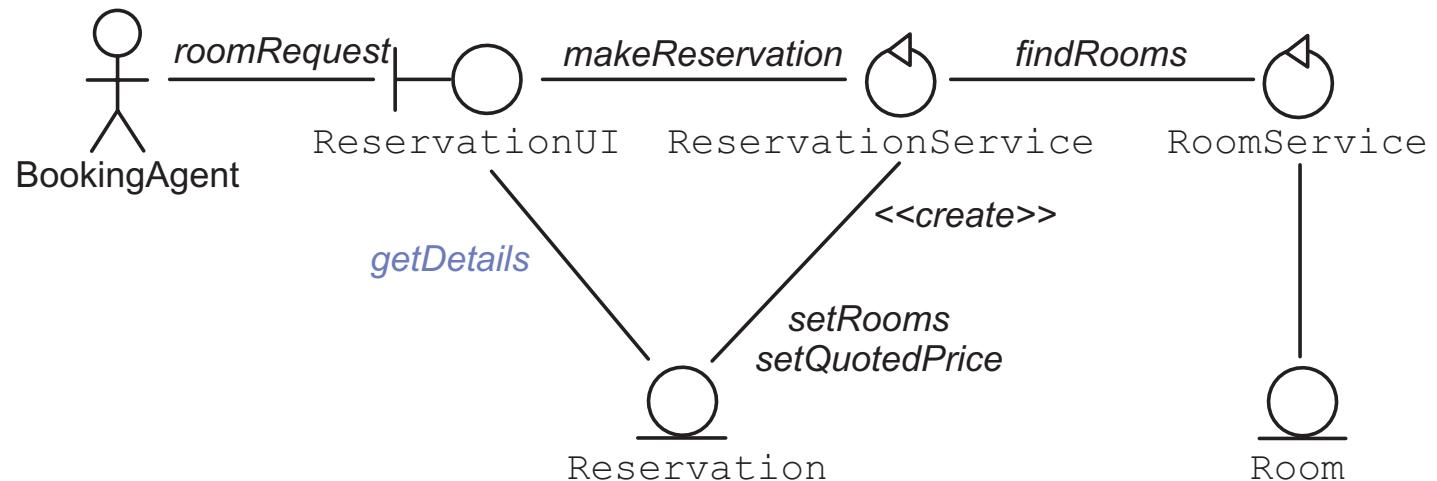


- A service object will often control its corresponding entity object
- A service object can delegate to another service object



Boundary And Entity Components

A Boundary component can often retrieve the attributes of an Entity component





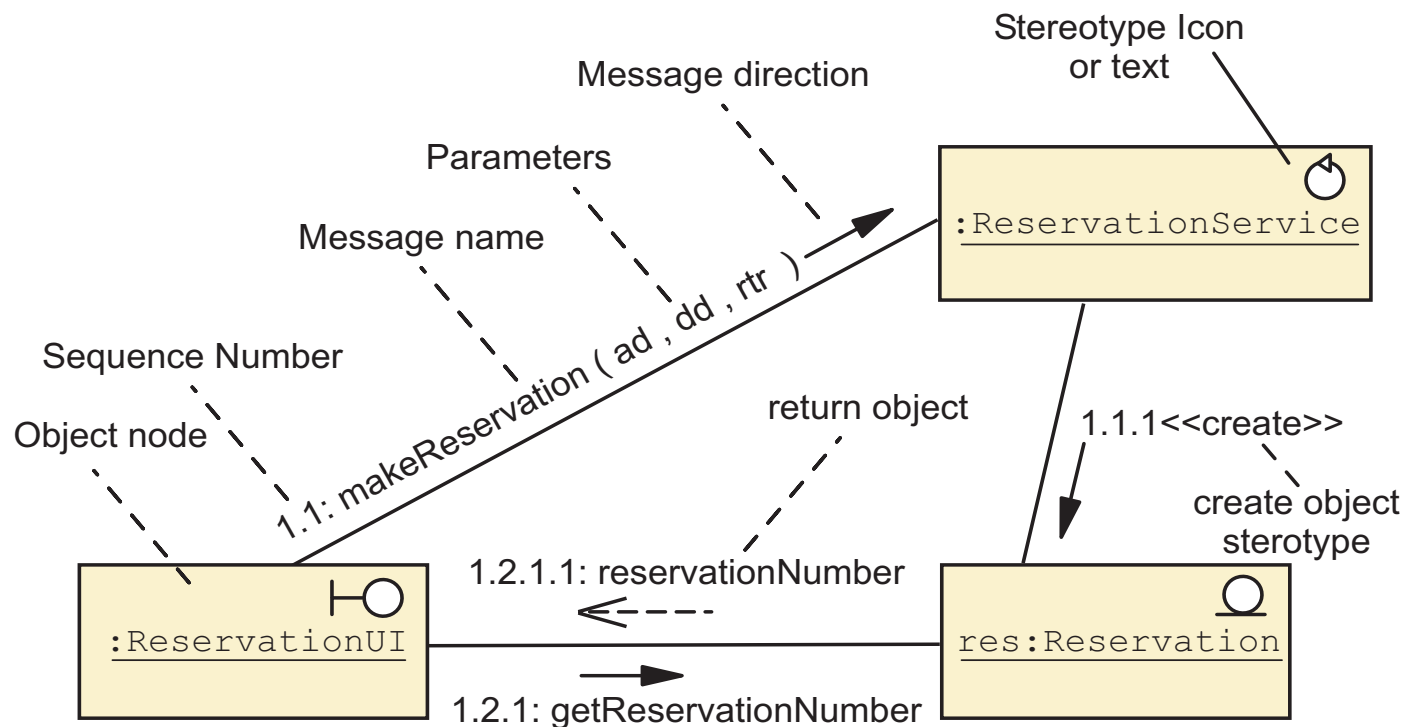
Describing the Robustness Analysis Process

1. Select a use case.
2. Construct a Communication diagram or a Sequence diagram that satisfies the activities of the use case.
 - a. Identify Design components that support the activities of the use case.
 - b. Draw the associations between these components.
 - c. Label the associations with messages.
3. Convert the Communication diagram into a Sequence diagram, or vice versa, for an alternate view (optional).



Identifying the Elements of a Communication Diagram

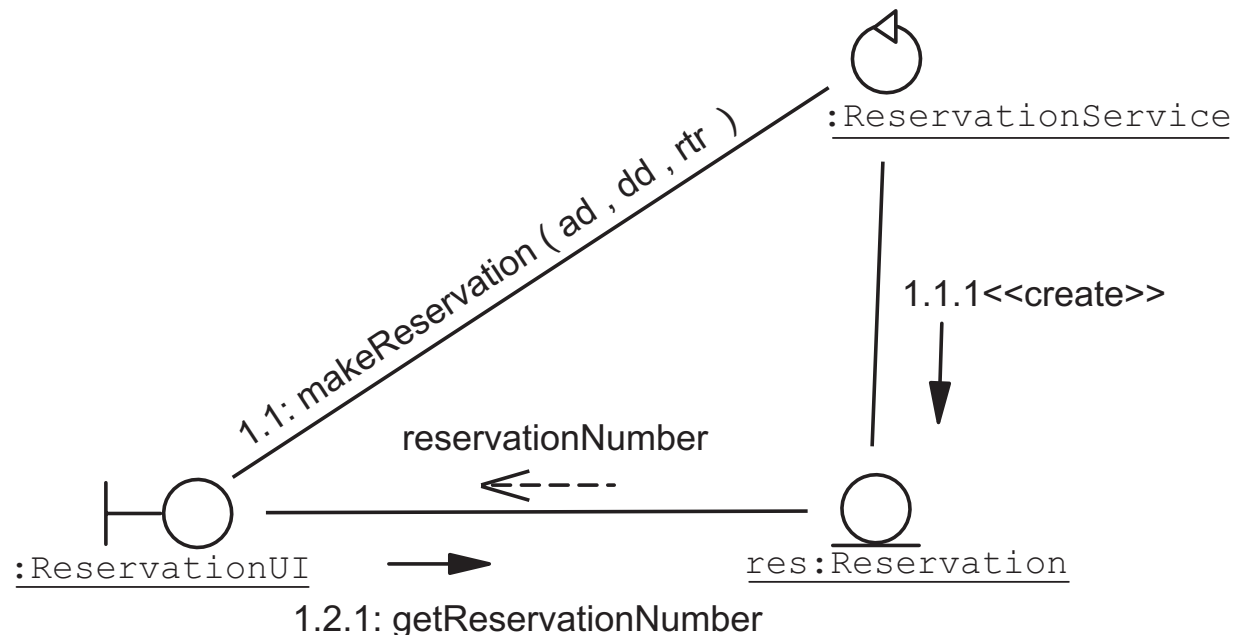
A UML Communication diagram is composed of the following elements:





Identifying the Elements of a Communication Diagram

A variation of the previous Communication diagram:





Identifying the Elements of a Communication Diagram

A message can indicate:

- A message name
- A direction arrow
 - An solid arrowhead is a synchronous message
 - An open arrowhead is an asynchronous message
- A sequence number describing the order of the message
- A list of parameters passed to the receiving object
- A guard condition indicating a conditional message
- A return parameter



Creating a Communication Diagram

Select an appropriate use case.

1. Place the actor in the Communication diagram.
2. Analyze the Use Case form or the Activity diagram for the use case.
For every action in the use case:
 - a. Identify and add a Boundary component.
 - b. Identify and add a Service component.
 - c. Identify and add an Entity component.
 - d. Identify and add further Interactions, discovering new Methods, Boundary, Service and Entity components.



Step 1 — Place the Actor in the Diagram

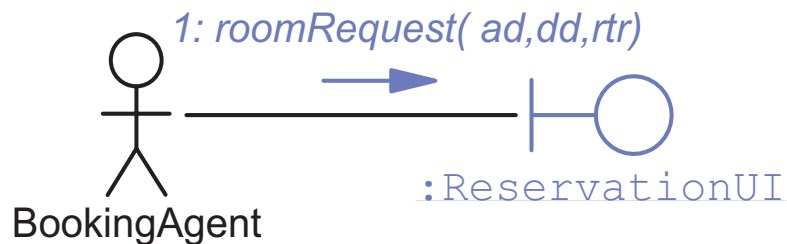
Place the actor in the Communication diagram:





Step 2a — Identify Boundary Components

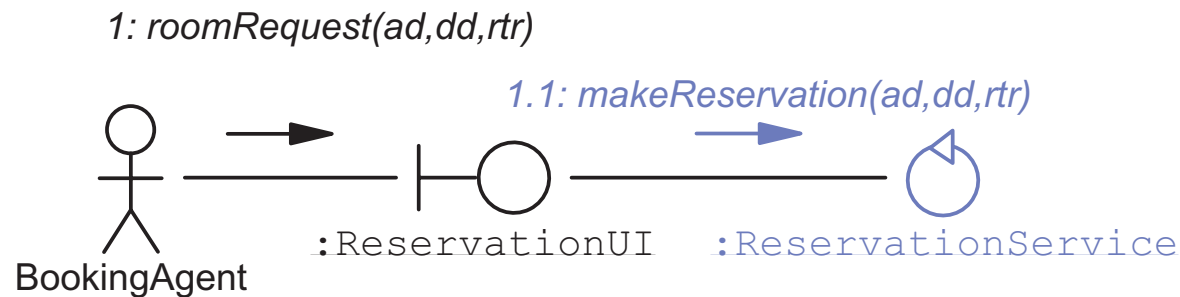
BookingAgent makes a room request passing the arrival date (ad), departure date (dd), requested types of room (rtr):





Step 2b — Identify Service Components

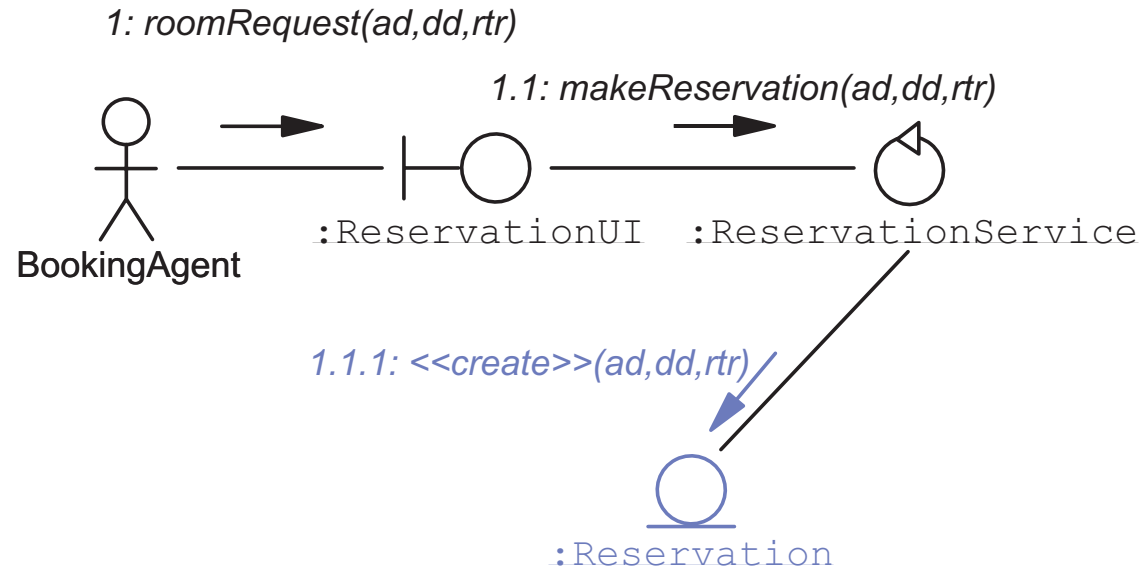
The ReservationUI boundary object uses ReservationService object to make the Reservation:





Step 2c — Identify Entity Components

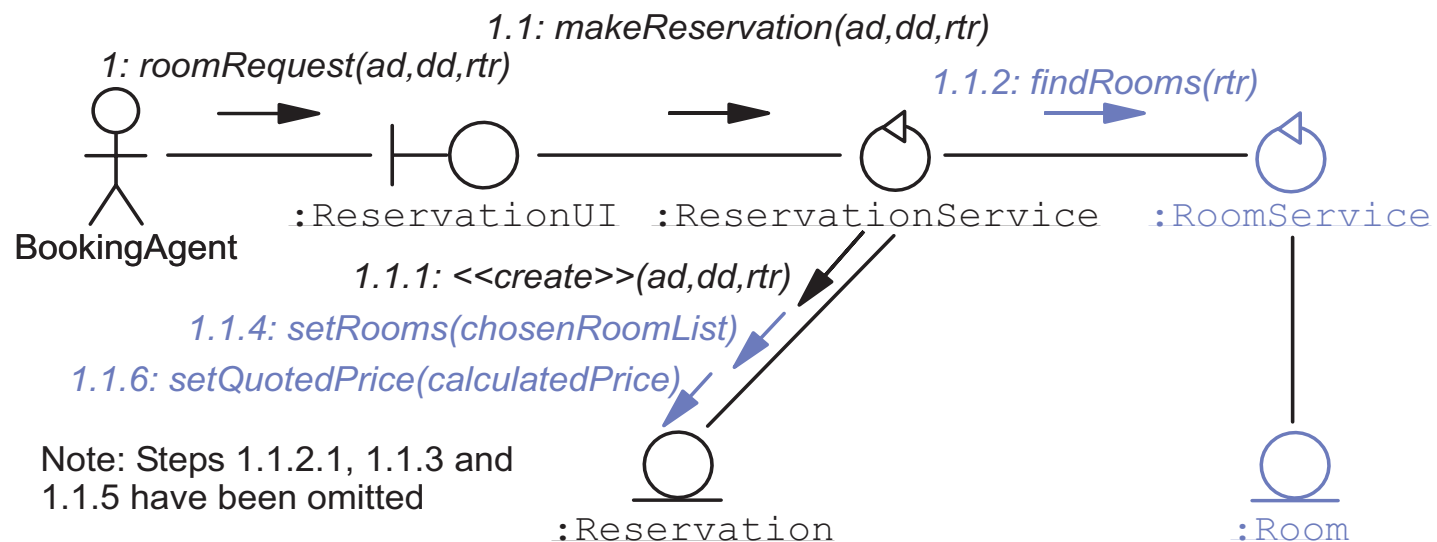
The makeResevation method in the ReservationService object creates the Reservation entity object:





Step 2d — Identify Additional Interactions

The makeReservation method uses the findRoom method to find rooms of the required type. After finding and choosing free rooms (not shown) the setRooms method assigns the rooms to the Reservation. The calculated price (not shown) is assigned to the Reservation:



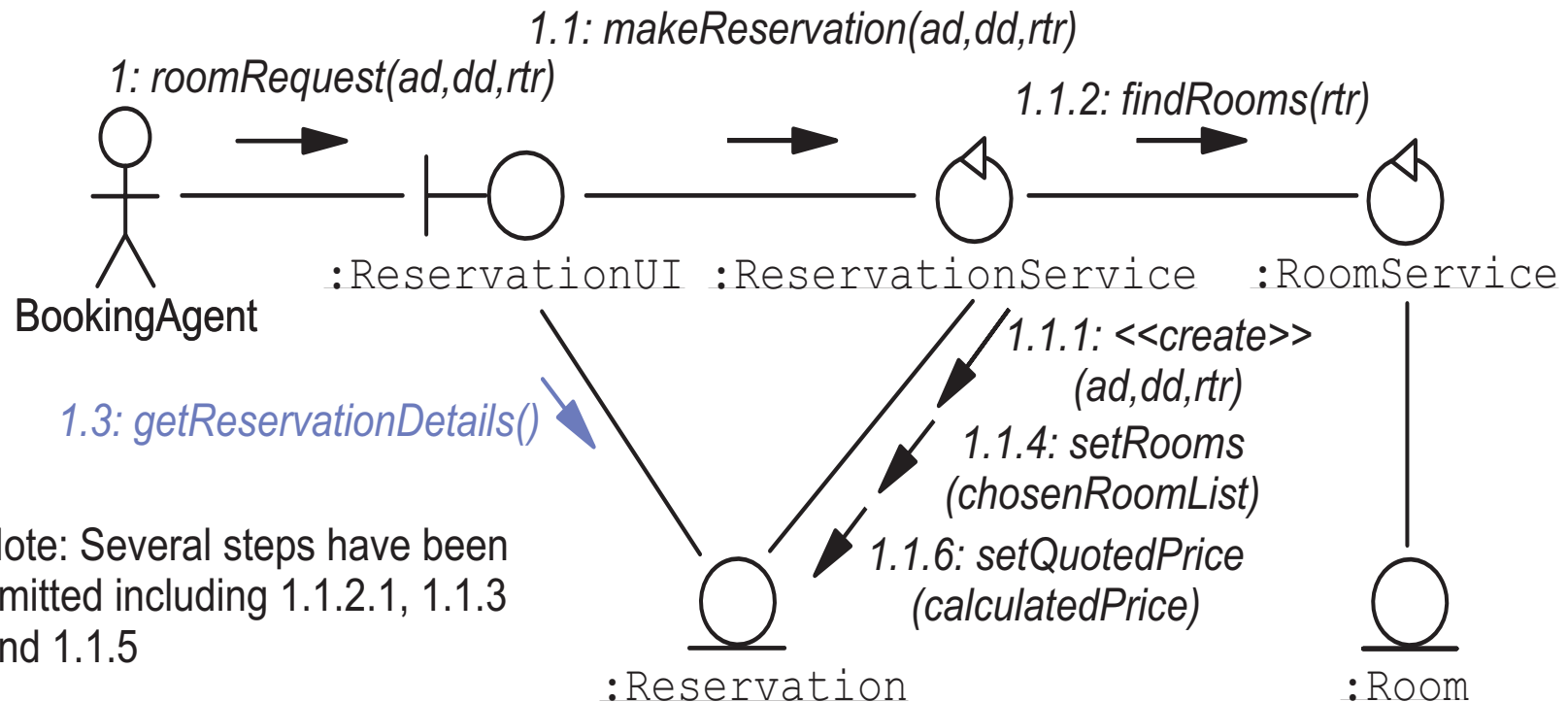


Step 2d — Identify Additional Methods

The `makeReservation` method of the `ReservationService` object returns the `Reservation` object (not shown) to the `ReservationUI`. The `roomRequest` method of `ReservationUI` then calls the `getReservationDetails` method of the `Reservation` object, which returns the details of the reservation. These will then be notified to the booking agent (not shown):



Step 2d — Identify Additional Methods



Note: Several steps have been omitted including 1.1.2.1, 1.1.3 and 1.1.5



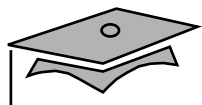
Communication Diagram Examples

The following two Communication diagrams show a more detailed view of the CreateReservation:

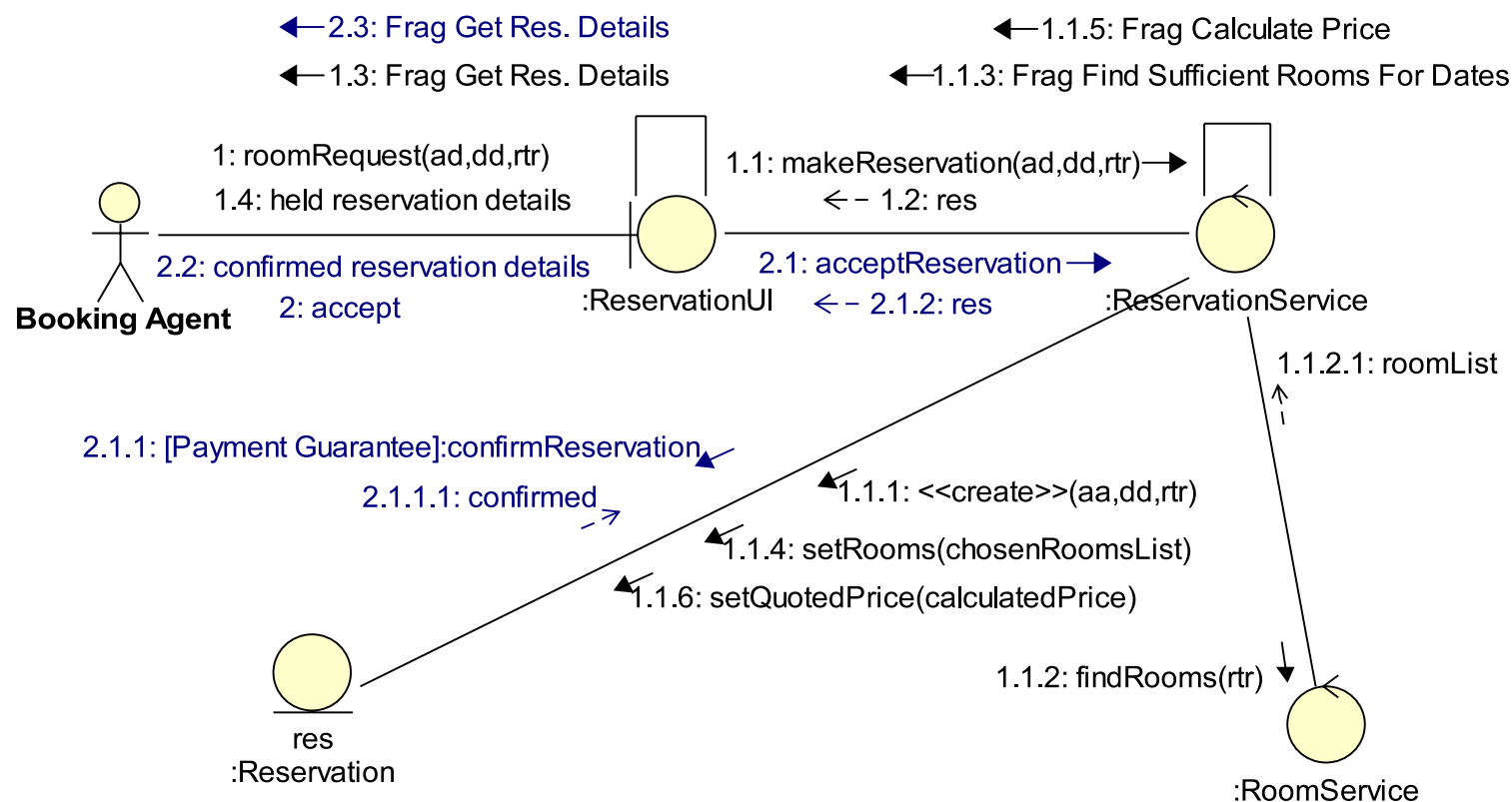
1. Primary (successful) scenario
2. Secondary (unsuccessful) scenario

Rooms offered are rejected by the booking agent

The finer details has been omitted, to reduce complexity.



Communication Diagram Example 1

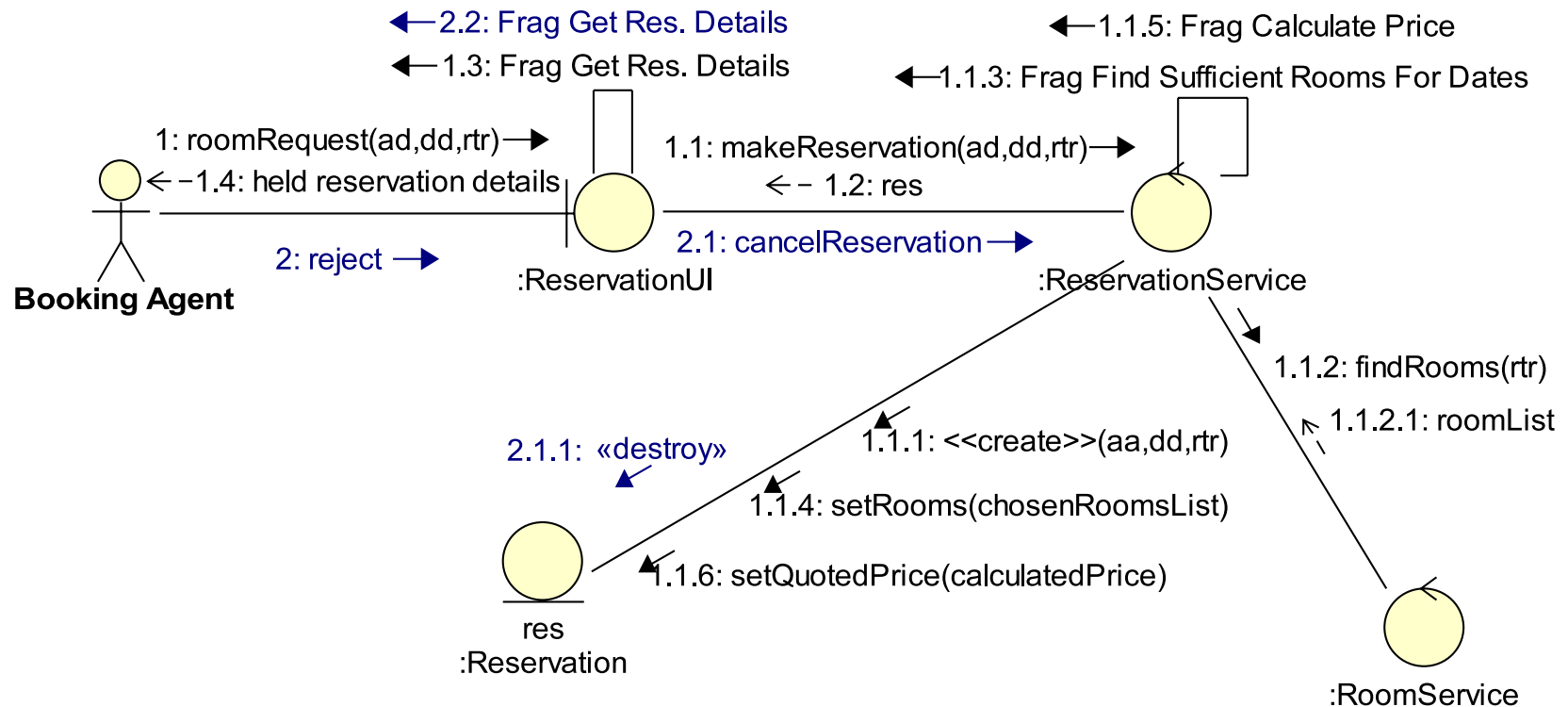


Key:
ad - arrivalDate
dd - departureDate
rtr - requestedTypeOfRoom

Note:
Frag: These are fragments where detail can be shown in another diagram
We have omitted two fragments that Identify Customer and Payment Guarantee



Communication Diagram Example 2



Key:
ad - arrivalDate
dd - departureDate
rtr - requestedTypeOfRoom

Note:
Frag: These are fragments where detail can be shown in another diagram
We have omitted two fragments that Identify Customer and Payment Guarantee



Sequence Diagrams

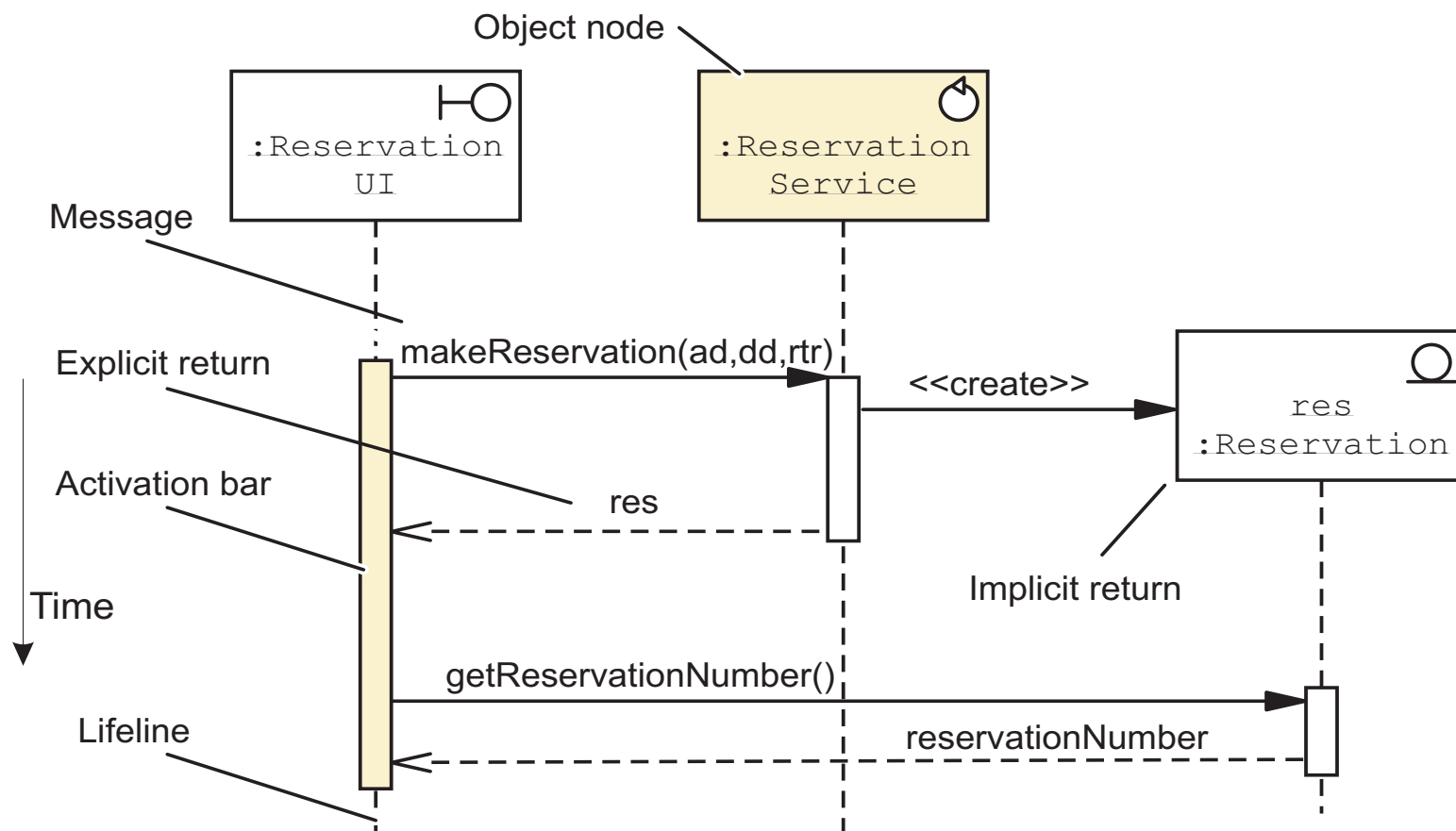
Sequence diagrams:

- Provide a different perspective of the interactions between objects
- Can be used instead of Communication diagrams
- Can be converted to or from Communication diagrams
- Prove to be more useful for developers.
- Highlight the time ordering of the interactions

The next section describes UML Sequence diagrams.



Identifying the Elements of a Sequence Diagram





Fragments

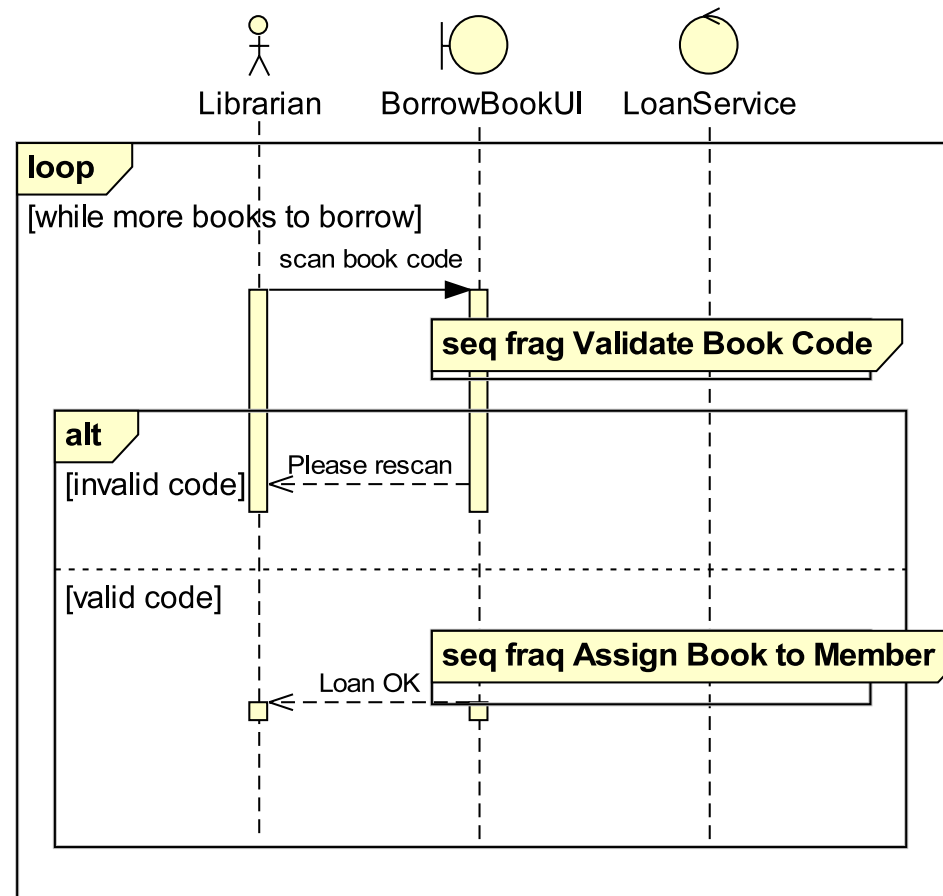
Sequence diagrams support a Fragment notation. The uses of fragments include:

- Showing sequence loops
- Showing alternative paths
- Allowing two or more scenarios to be shown on one diagram
- Showing a reference to another detailed Sequence diagram fragment
- Allowing you to break up a large diagram into several smaller diagrams

The following slide shows examples of the Fragment notation



Fragment Examples



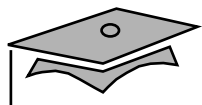


Sequence Diagram Examples

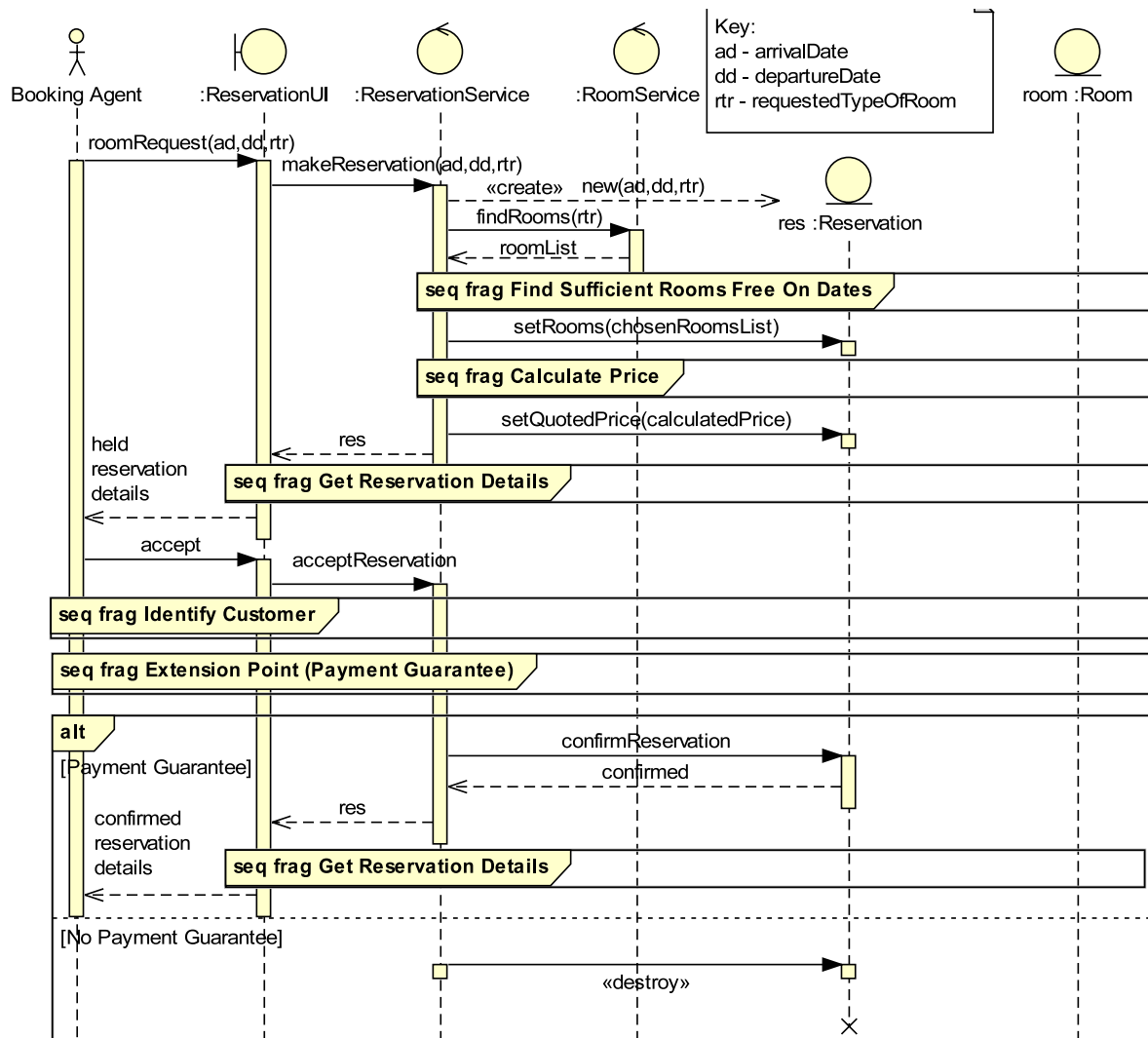
The following three Sequence diagrams show a more detailed view of CreateReservation:

1. Primary (successful) scenario and a secondary (unsuccessful) scenario in one diagram using an *alt* fragment
2. Secondary (unsuccessful) scenario where rooms offered are rejected by the booking agent
3. A Fragment Sequence diagram showing the finer details for the GetReservationDetails fragment, and includes a *loop* fragment

The finer detail has been hidden in fragments.

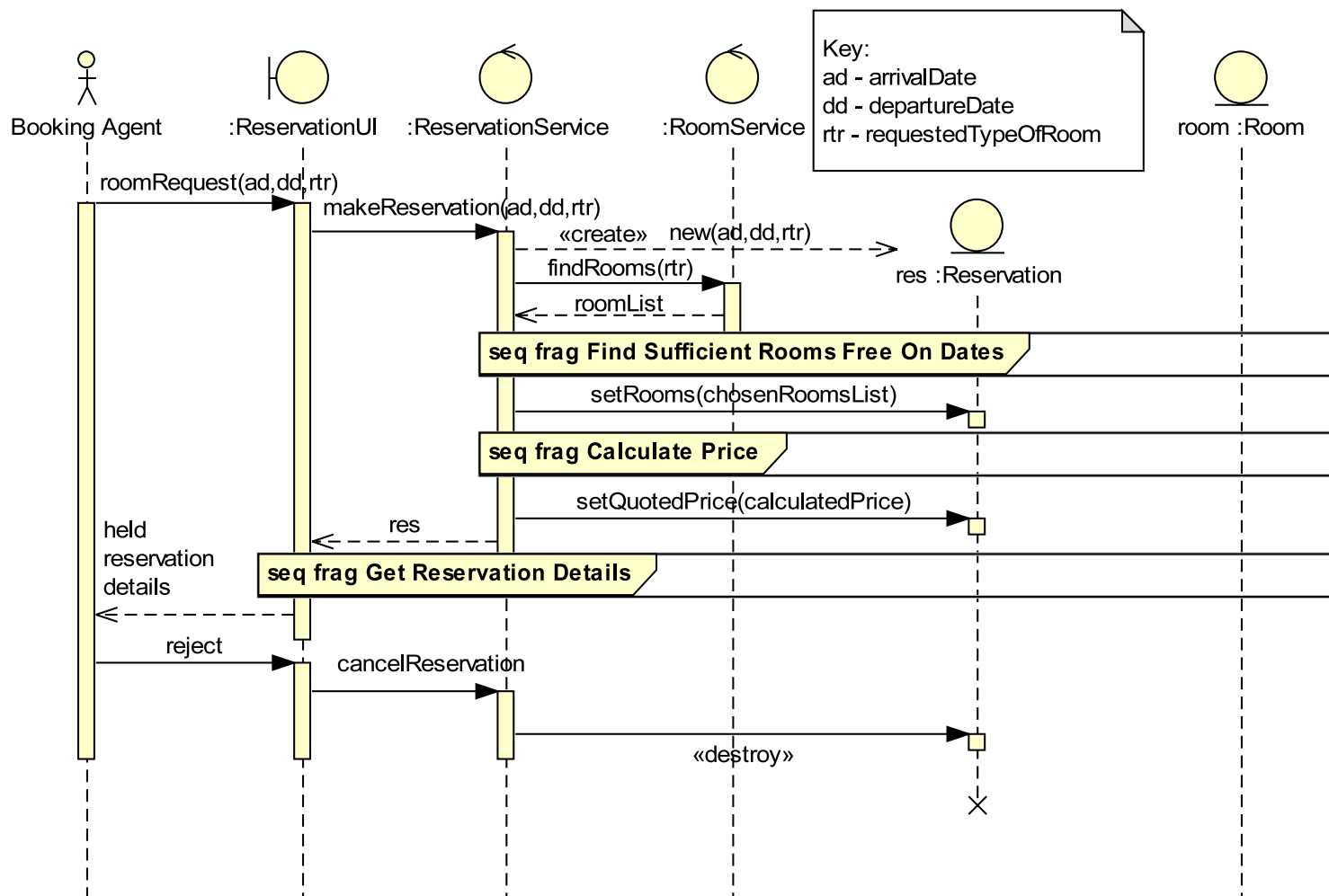


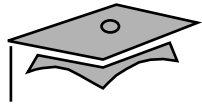
Sequence Diagram Example 1



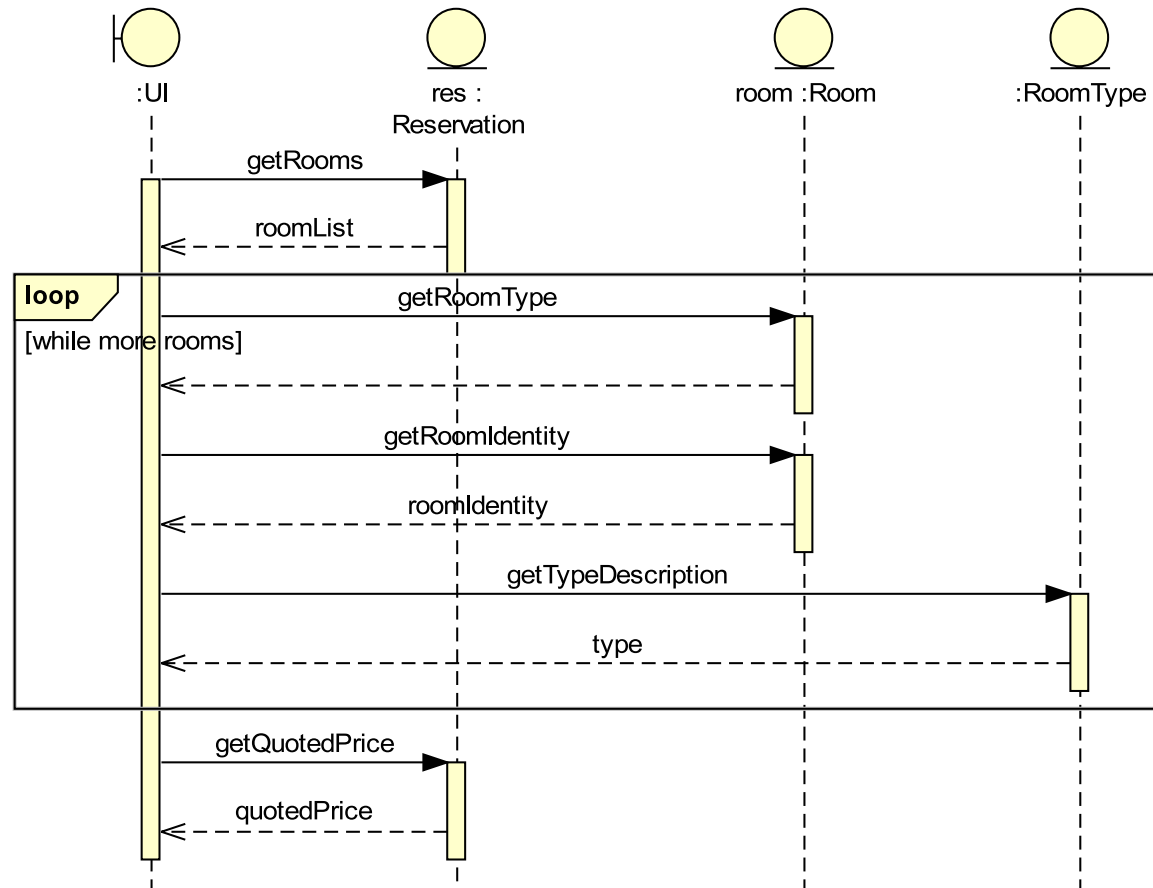


Sequence Diagram Example 2





Sequence Diagram Fragment Example 3





Summary

- Interaction diagrams are used to identify design components that satisfy a use case.
- Object interactions can be visualized with a UML
 - Communication diagram
 - Sequence diagram



Module 9

Modeling Object State Using State Machine Diagrams



Objectives

Upon completion of this module, you should be able to:

- Model object state
- Describe the essential elements of a UML State Machine diagram



Introducing Object State

State is “1a: mode or condition of being” (Webster)

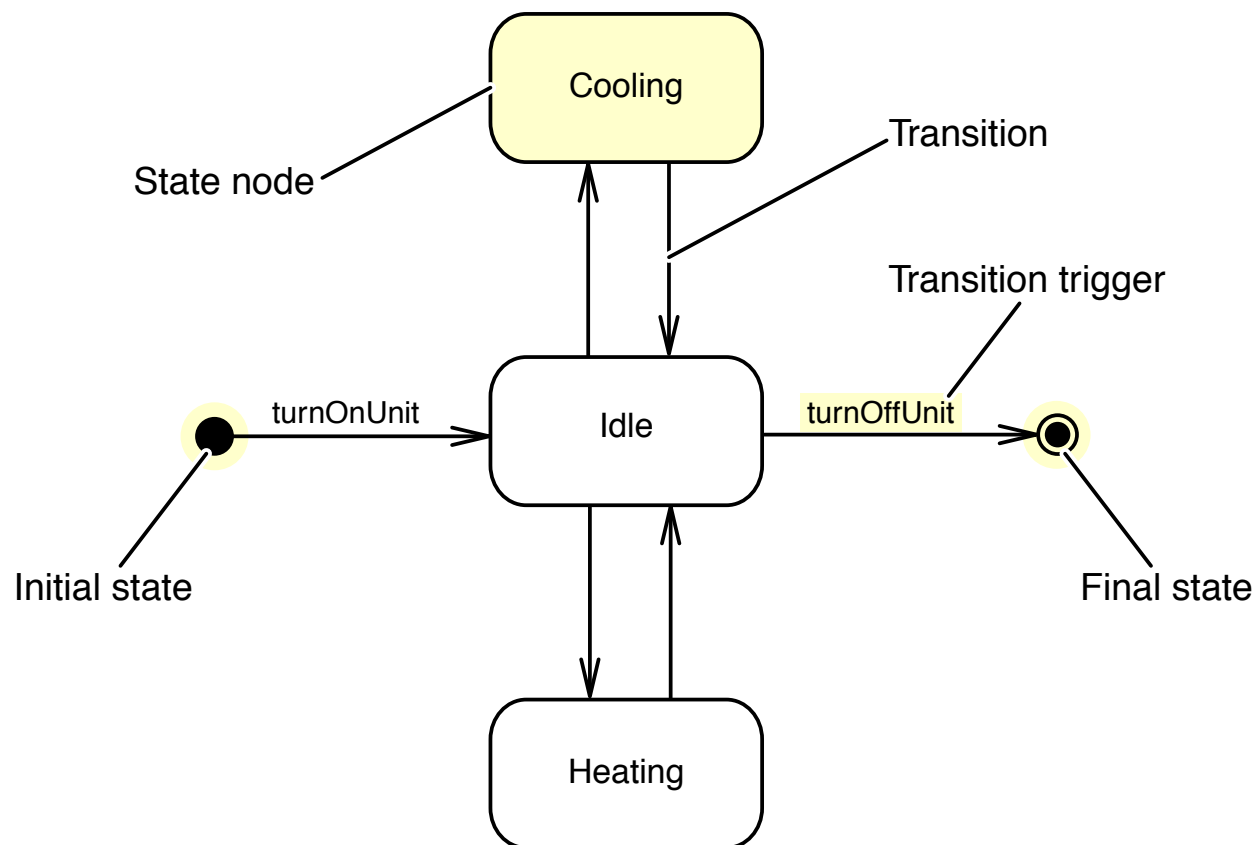
There are two ways to think about object state:

- The state of an object is specific collection of attribute values for the object.
- The state of an object describes the behavior of the object relative to external stimuli.

This module considers the second definition.



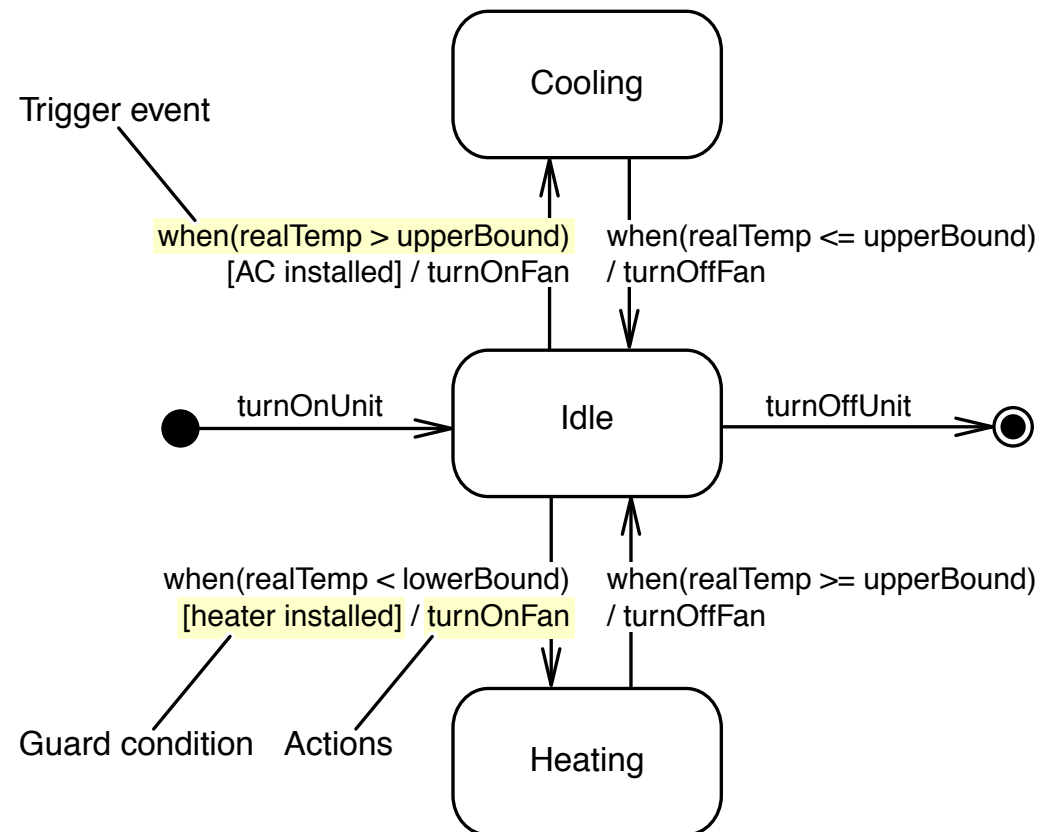
Identifying the Elements of a State Machine Diagram





State Transitions

A state transition represents a change of state at runtime.



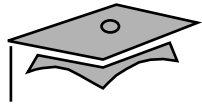


Internal Structure of State Nodes

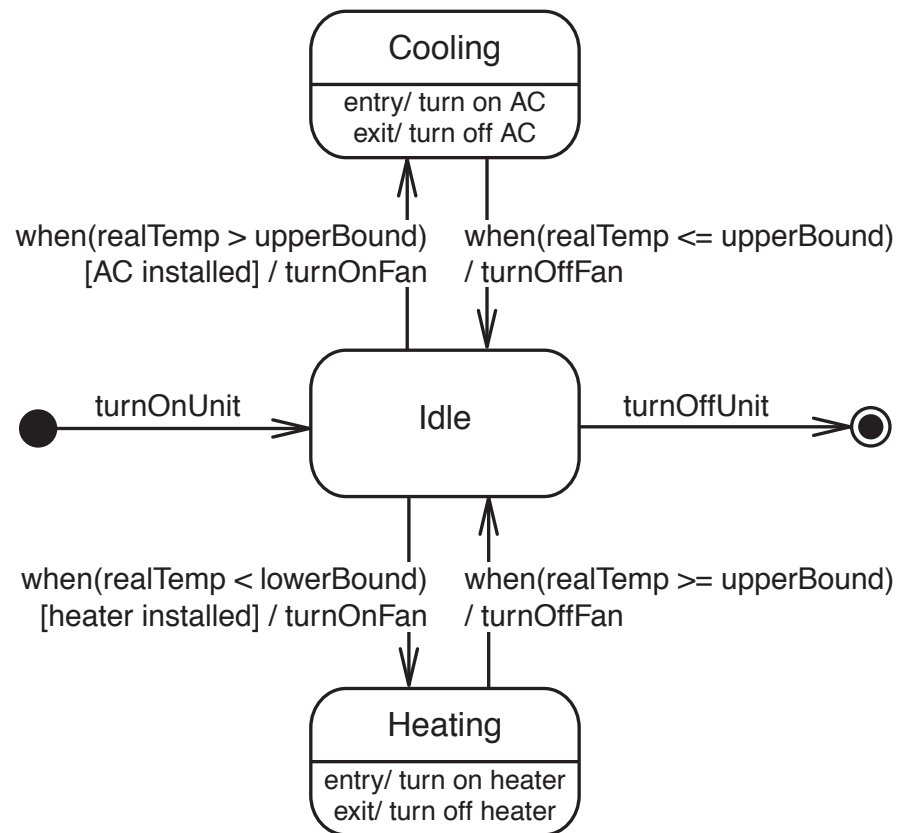
State nodes represents a state of a single object at runtime.



- “Entry” event specifies actions upon entry into the state.
- “Exit” event specifies actions upon exit from the state.
- “Do” event specifies ongoing actions.
- You can also specify specific events with corresponding actions.



Complete HVAC State Machine Diagram





Creating a State Machine Diagram for a Complex Object

1. Draw the initial and final state for the object.
2. Draw the stable states of the object.
3. Specify the partial ordering of stable states over the lifetime of the object.
4. Specify the events that trigger the transitions between the states of the object. Specify transition actions (if any).
5. Specify the actions within a state (if any).

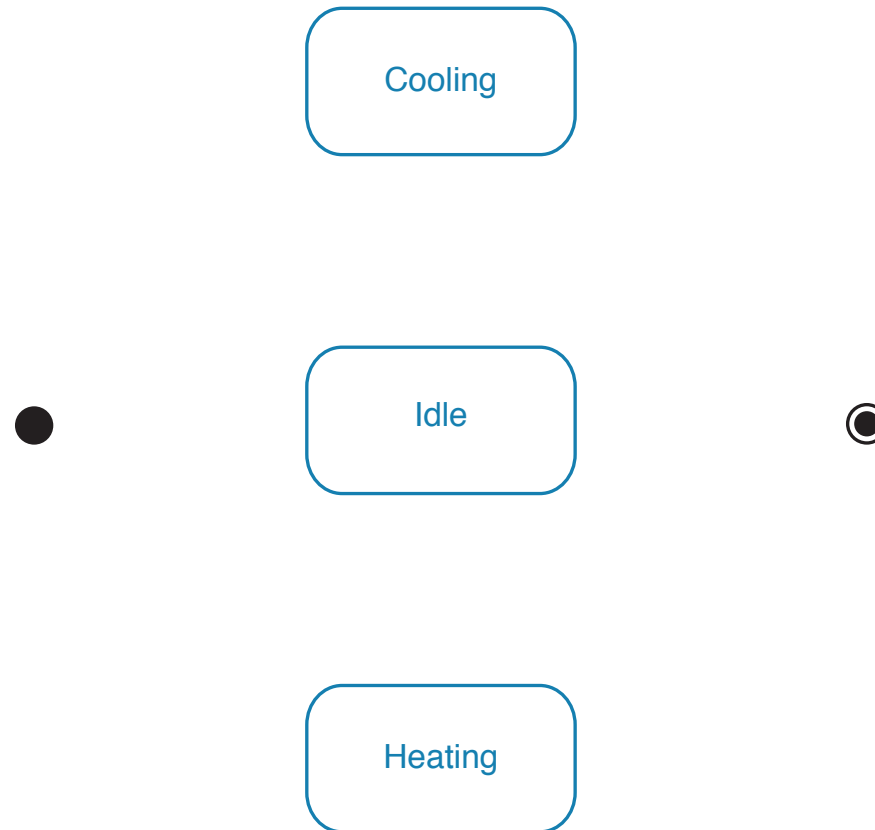


Step 1 – Start With the Initial and Final States



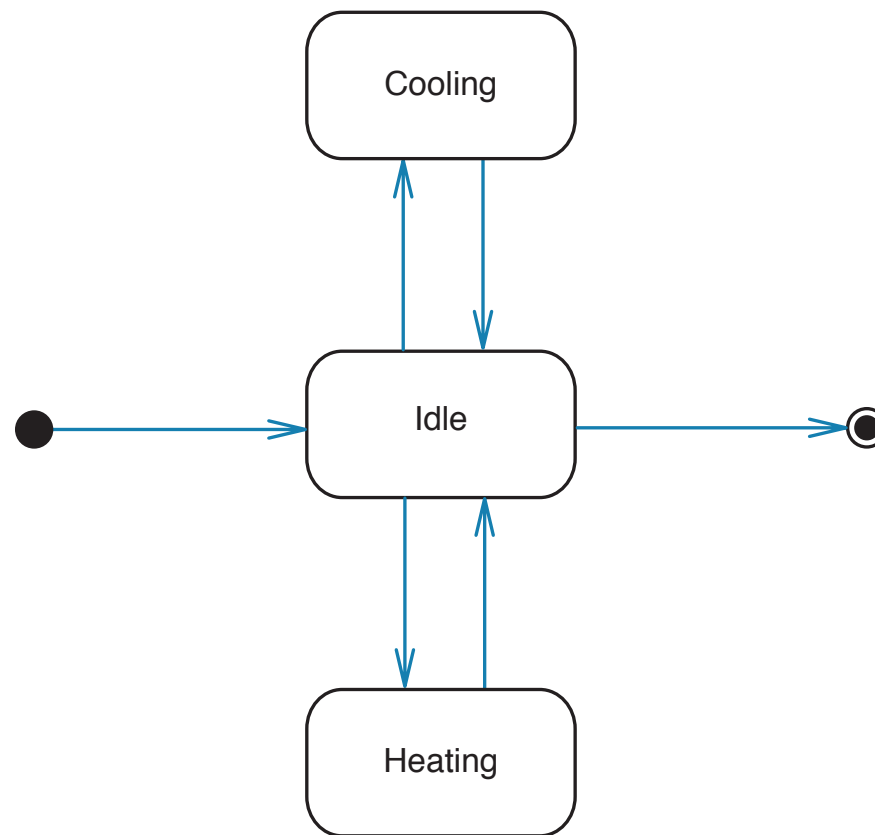


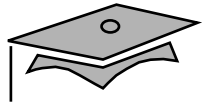
Step 2 – Determine Stable Object States



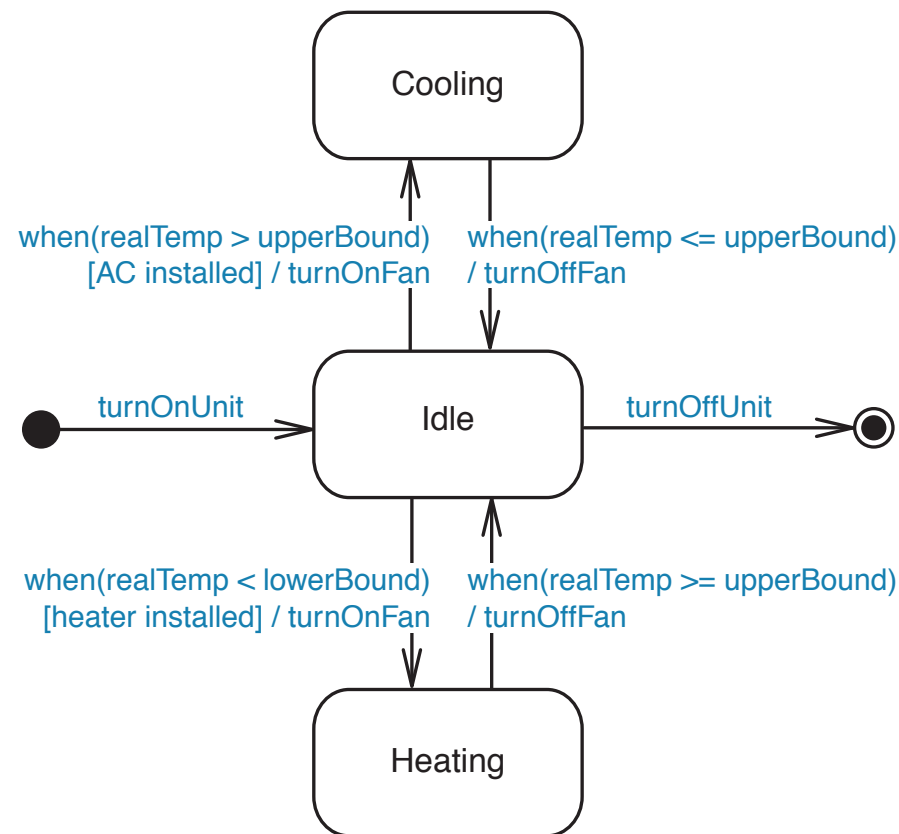


Step 3 – Specify the Partial Ordering of States



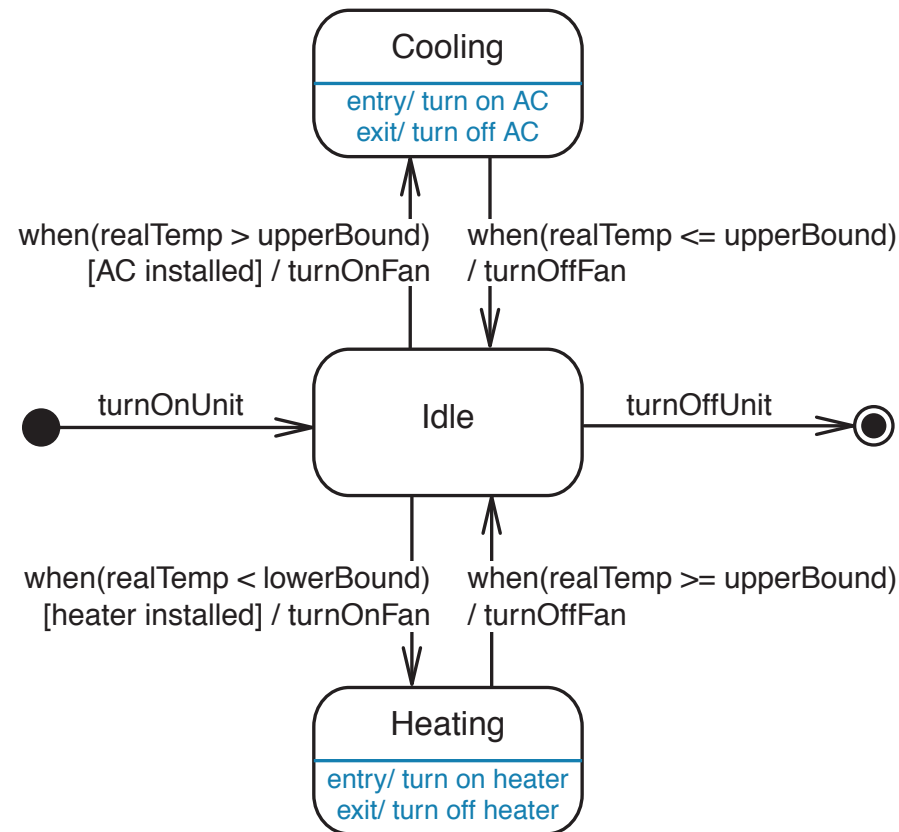


Step 4 – Specify the Transition Events and Actions





Step 5 – Specify the Actions Within a State





After Trigger Event

Events that should occur after a period of time are shown by using the *after* trigger event.

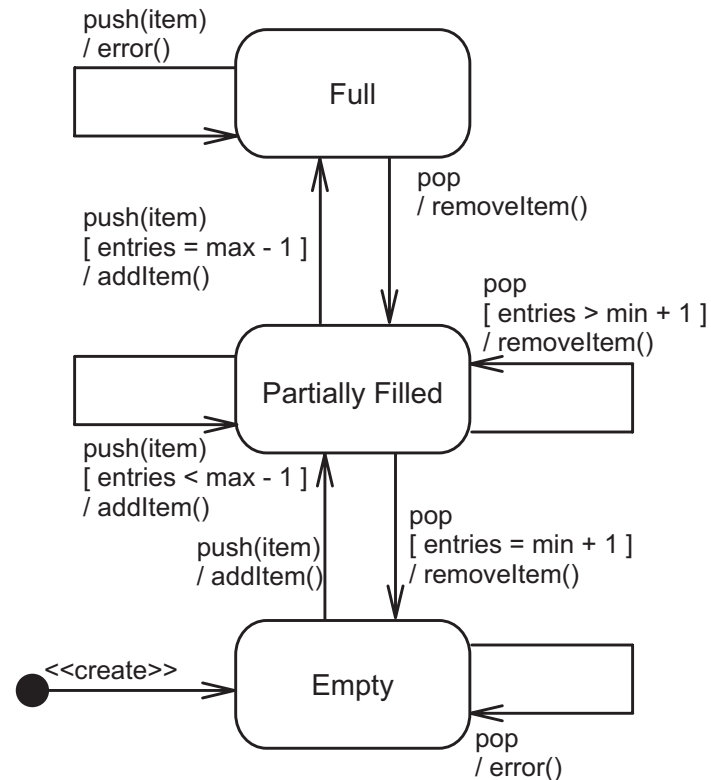
To fire a transition after 10 minutes, specify the event on the transition as `after(10 mins)`.

This event is often used for timeouts.



Self Transition

A self transition is a state transition with the same state for the source and the destination of the transition.

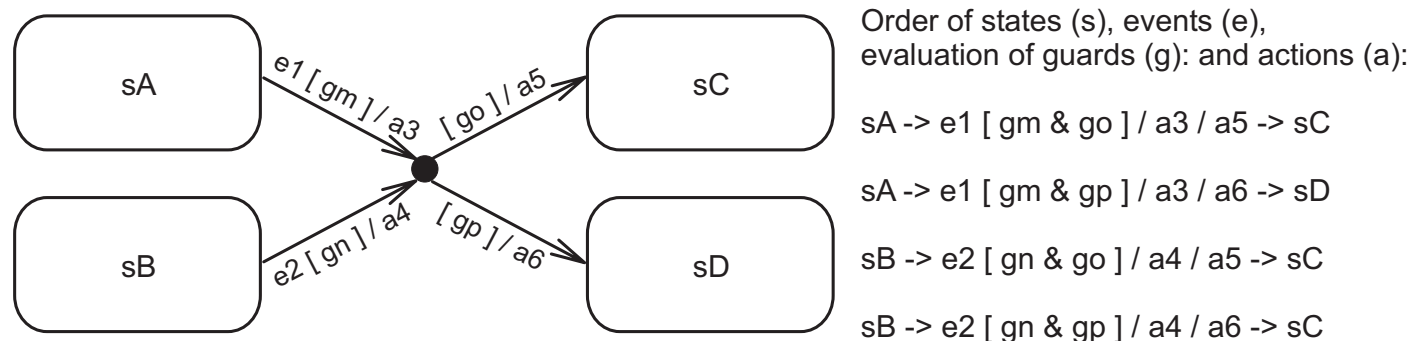


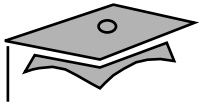


Junction

In UML, a Junction can be used to simplify diagrams by breaking the transition into several fragments, thereby reducing the duplication of trigger events, guards, and actions.

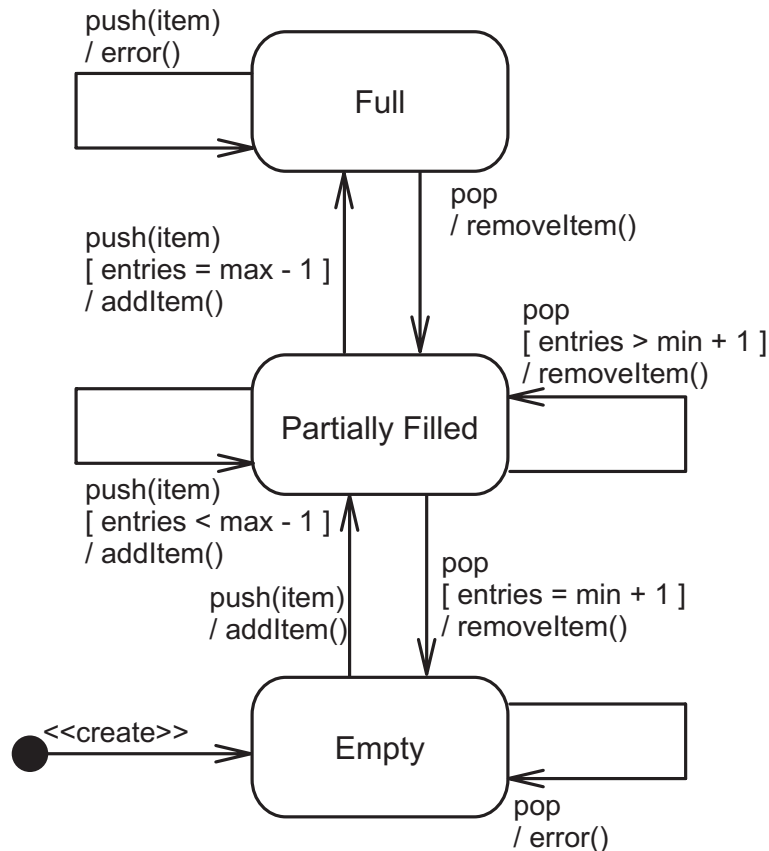
The following example shows the order of state transitions, order of events, evaluation of guards, and actions.



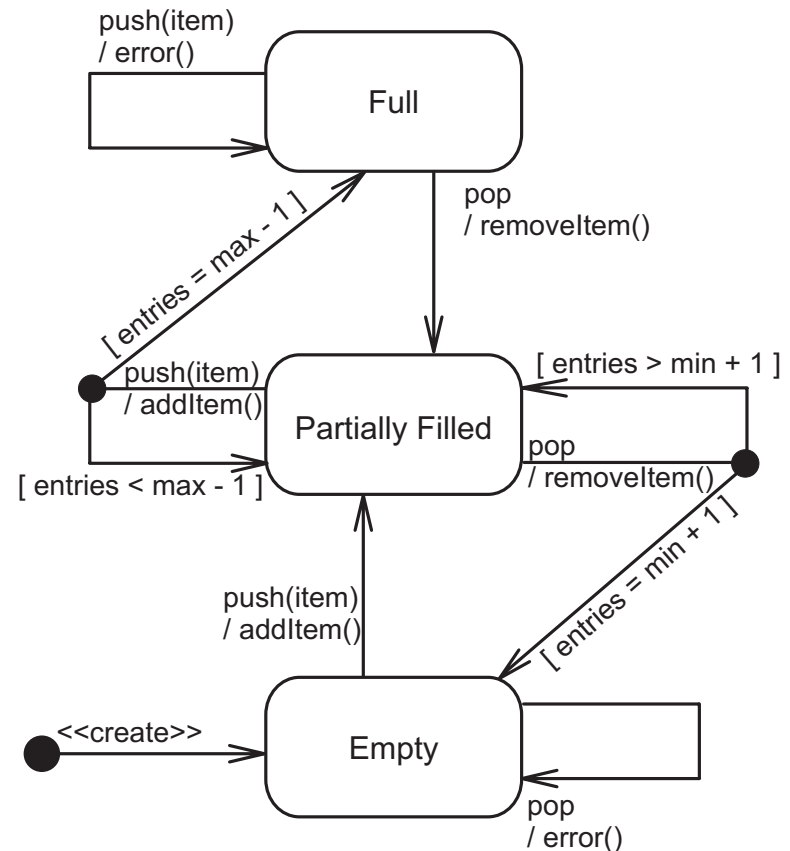


Junction Example

Stack example without using a junction



Equivalent stack example using a junction

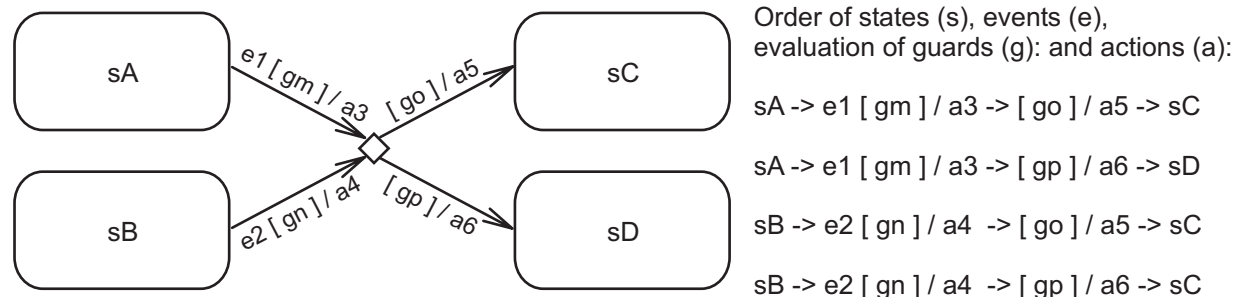


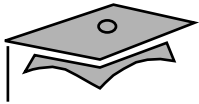


Choice

In UML, a Choice is used to simplify diagrams by breaking the transition into several fragments, thereby reducing the duplication of trigger events, guards, and actions. Choice also enables dynamic evaluation of the guards, after the previous transition fragment actions are executed.

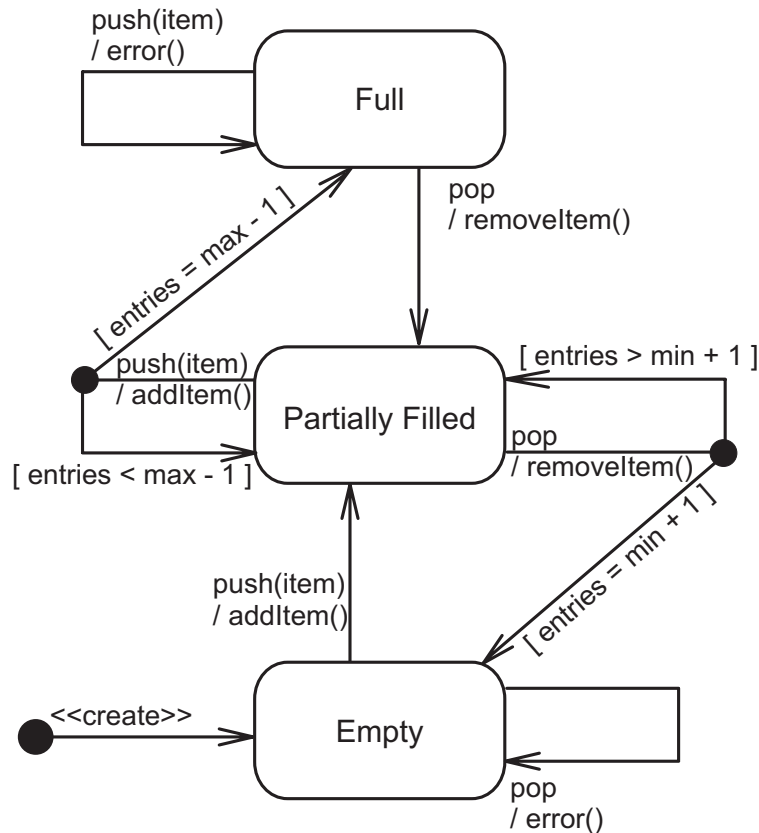
The following example shows the order of state transitions, order of events, evaluation of guards, and actions.



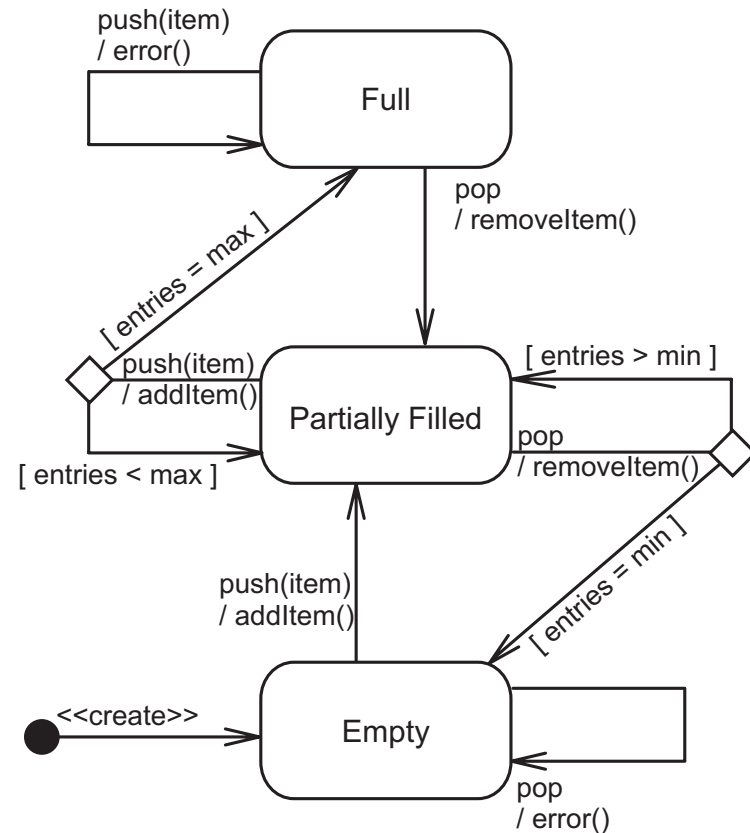


Choice Example

Stack example using a junction



Equivalent stack example using a choice





Summary

- An object might have states that define unique behaviors for the object.
- The State Machine diagram provides a mechanism for modeling the states and transitions of an object.



Module 10

Applying Design Patterns to the Design Model



Objectives

Upon completion of this module, you should be able to:

- Define the essential elements of a software pattern
- Describe the Composite pattern
- Describe the Strategy pattern
- Describe the Observer pattern
- Describe the Abstract Factory pattern
- Describe the State pattern



Explaining Software Patterns

A software pattern is a “description of communicating objects and classes that are customized to solve a general design problem in a particular context.” (Gamma, Helm, Johnson, and Vlissides page 3)

- Inspired by building architecture patterns
- Essential elements of a software pattern:
 - Pattern name
 - Problem
 - Solution
 - Consequences



Levels of Software Patterns

- Architectural patterns:
 - Manifest at the highest software and hardware structures
 - Usually support non-functional requirements
- Design patterns:
 - Manifest at the mid-level software structures
 - Usually support functional requirements
- Idioms:
 - Manifest at the lowest software structures (classes and methods)
 - Usually support language-specific features



Design Principles

There are several design principles that support the solutions of software patterns:

- Open Closed Principle (OCP)
- Composite Reuse Principle (CRP)
- Dependency Inversion Principle (DIP)

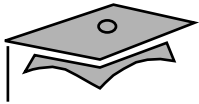


Design Principles

GridContainer
-components:ArrayList
«constructor» +GridContainer(width:int, height:int)
«methods» +add(:Component) +remove(:Component) +doLayout()

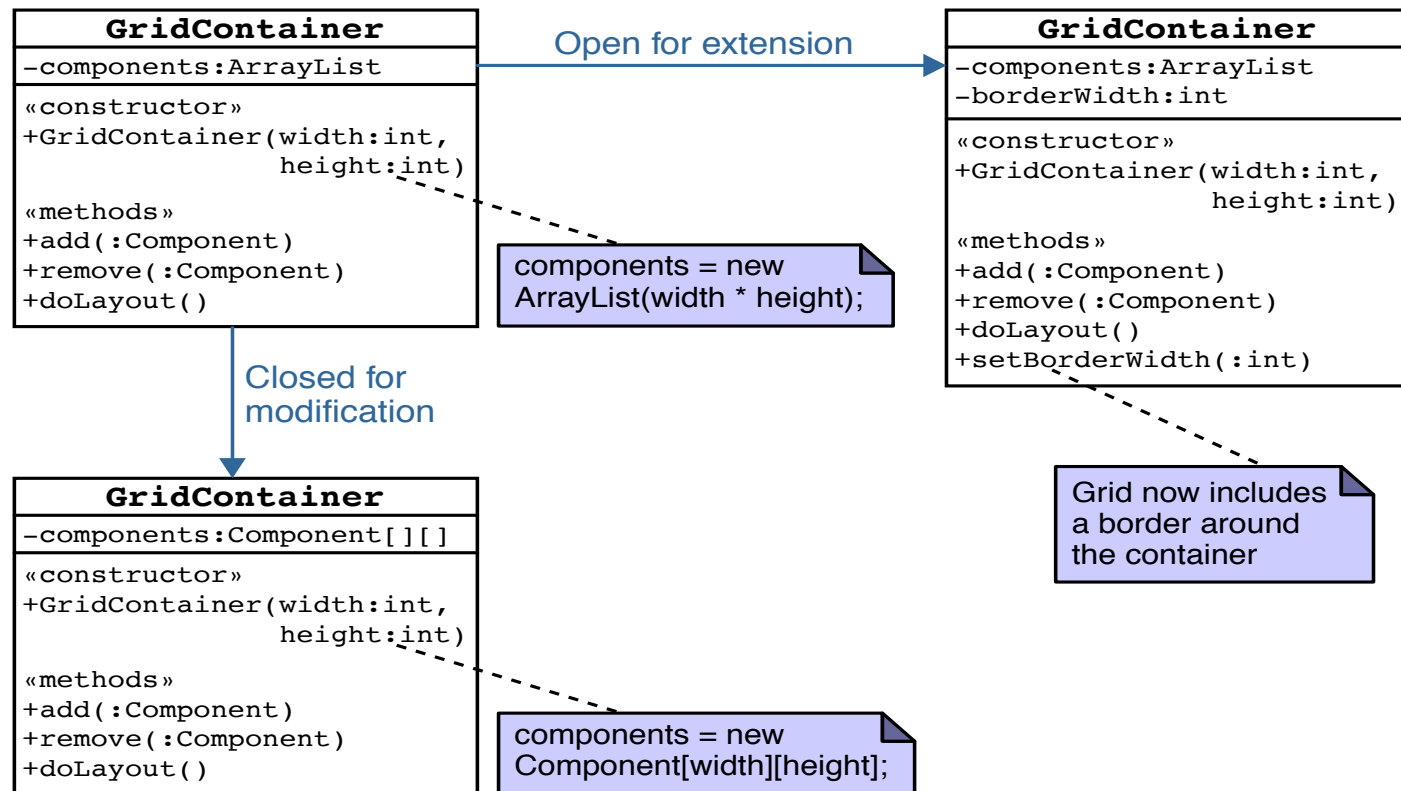
```
GridContainer calcGUI = new GridContainer(4,4);  
calcGUI.add(new Button("1"));  
calcGUI.add(new Button("2"));  
calcGUI.add(new Button("3"));  
calcGUI.add(new Button("+"));  
calcGUI.add(new Button("4"));  
calcGUI.add(new Button("5"));  
calcGUI.add(new Button("6"));  
calcGUI.add(new Button("-"));
```

1	2	3	+
4	5	6	-
7	8	9	*
0	.	=	/



Open Closed Principle

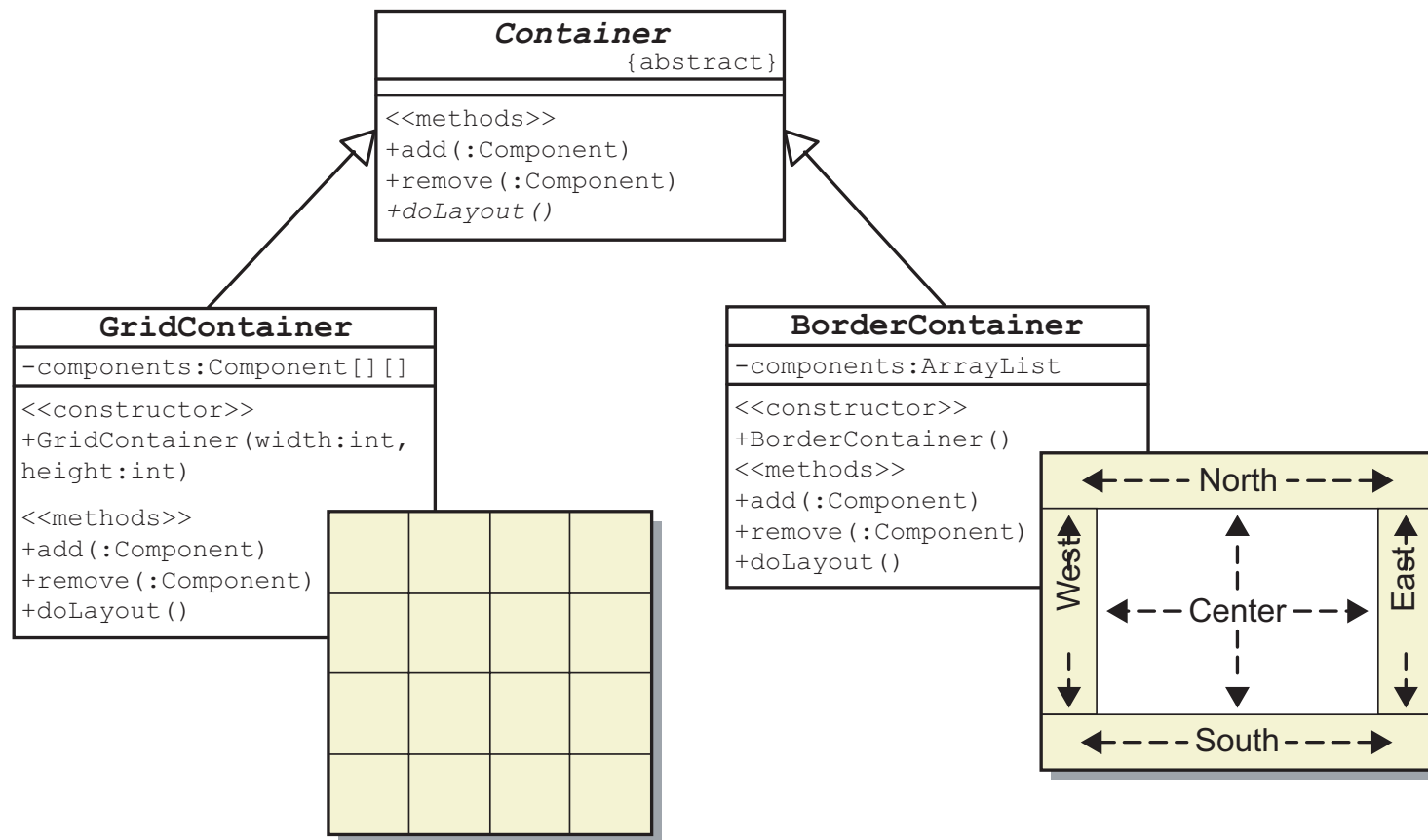
“Classes should be open for extension but closed for modification.” (Knoernschild page 8)





Composite Reuse Principle

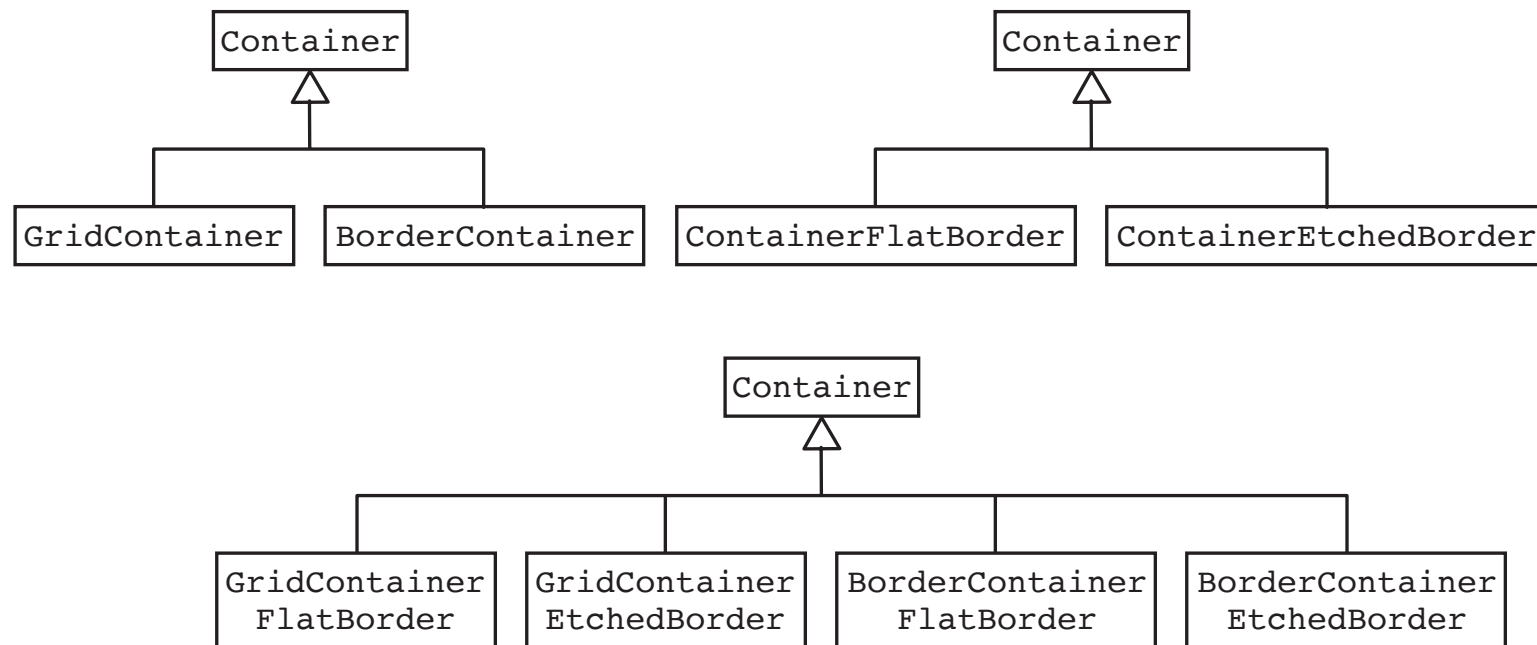
“Favor polymorphic composition of objects over inheritance.”
(Knoernschild page 17)





Composite Reuse Principle

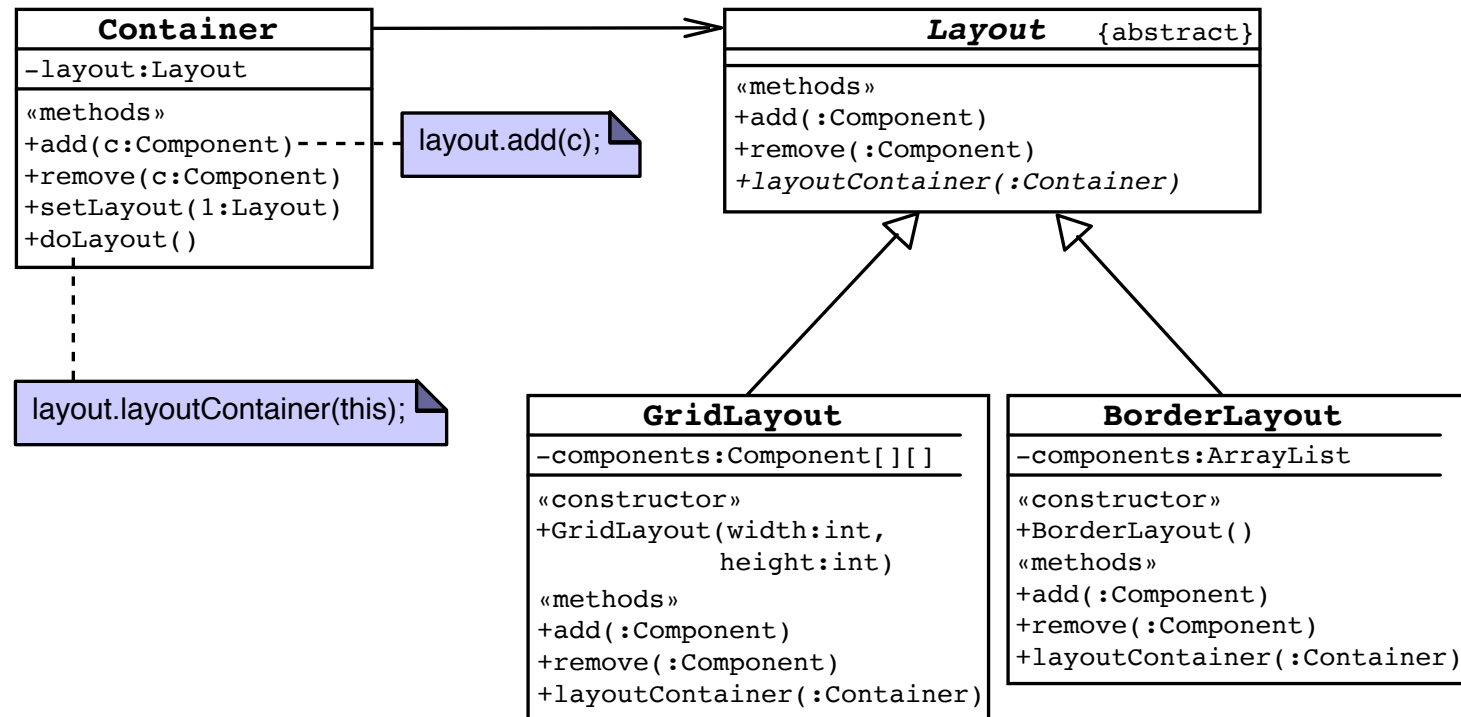
Excessive inheritance leads to brittle and large hierarchies:





Composite Reuse Principle

CRP leads to flexible reuse through delegation:

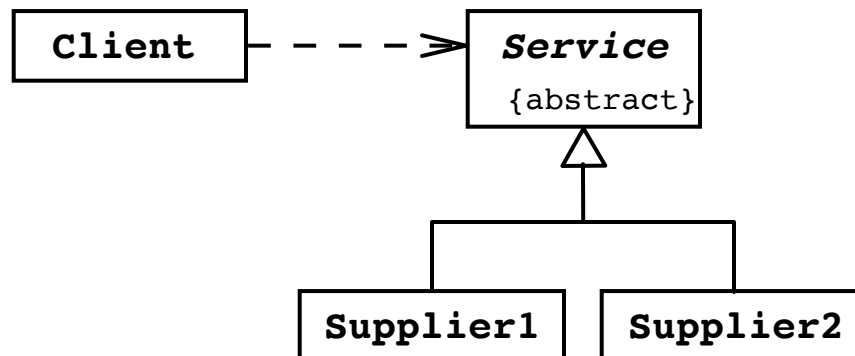




Dependency Inversion Principle

“Depend on abstractions. Do not depend on concretions.”
(Knoernschild page 12)

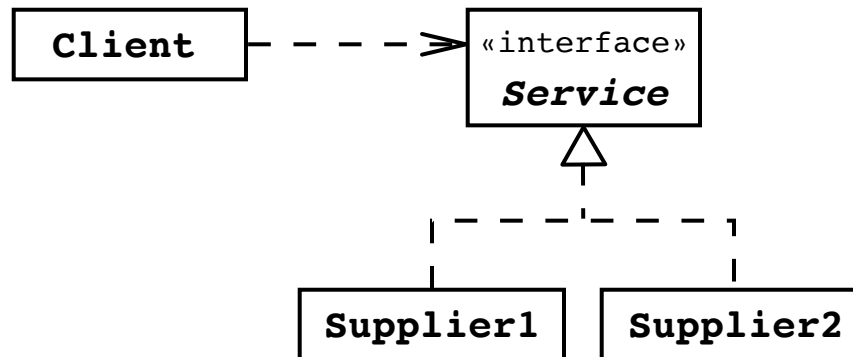
An abstraction can be an abstract class:





Dependency Inversion Principle

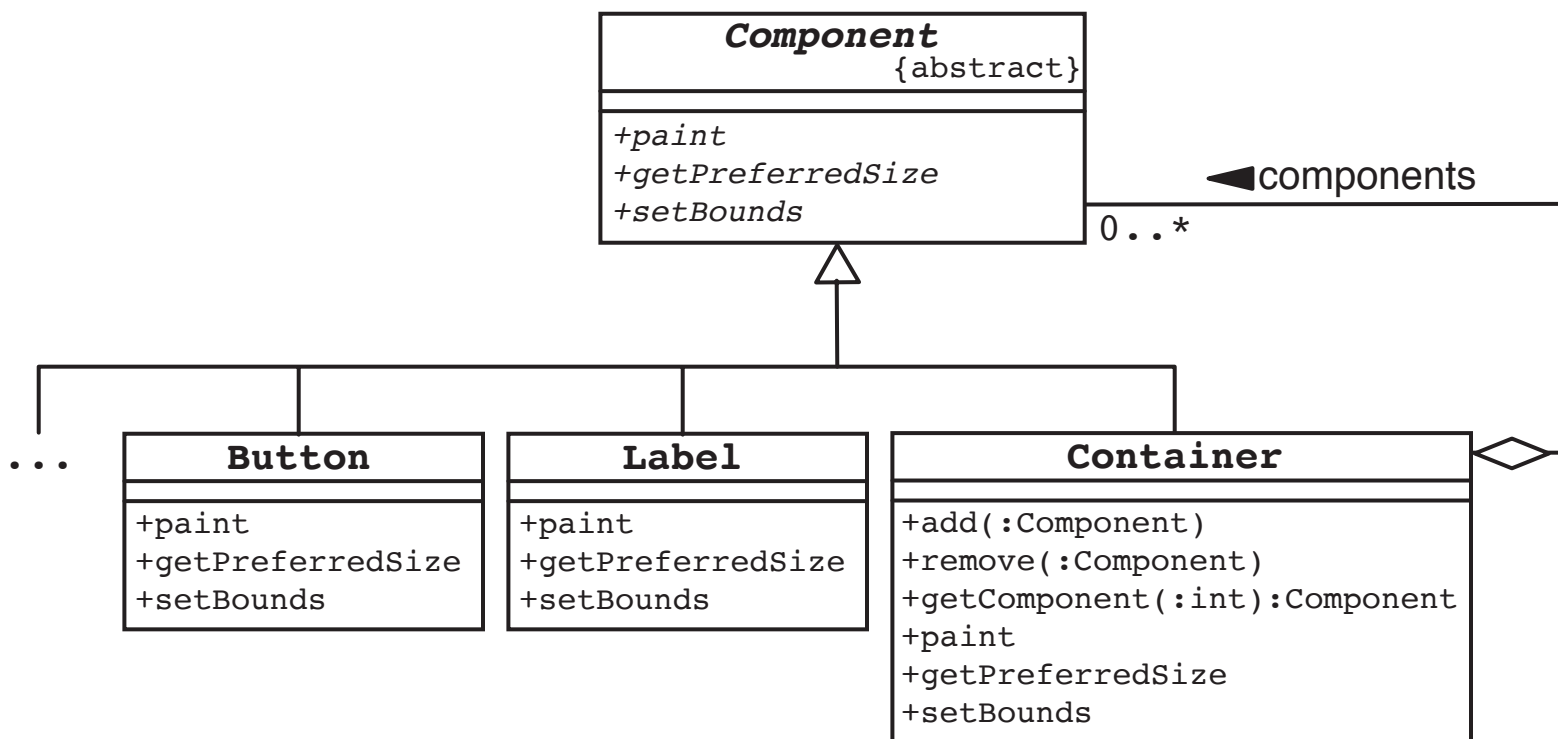
An abstraction can be a Java technology interface:





Describing the Composite Pattern

“Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.” (GoF page 163)





Describing the Composite Pattern

Problem:

- You want to represent whole-part hierarchies of objects
- You want to use the same interface on the assemblies and the components in an assembly

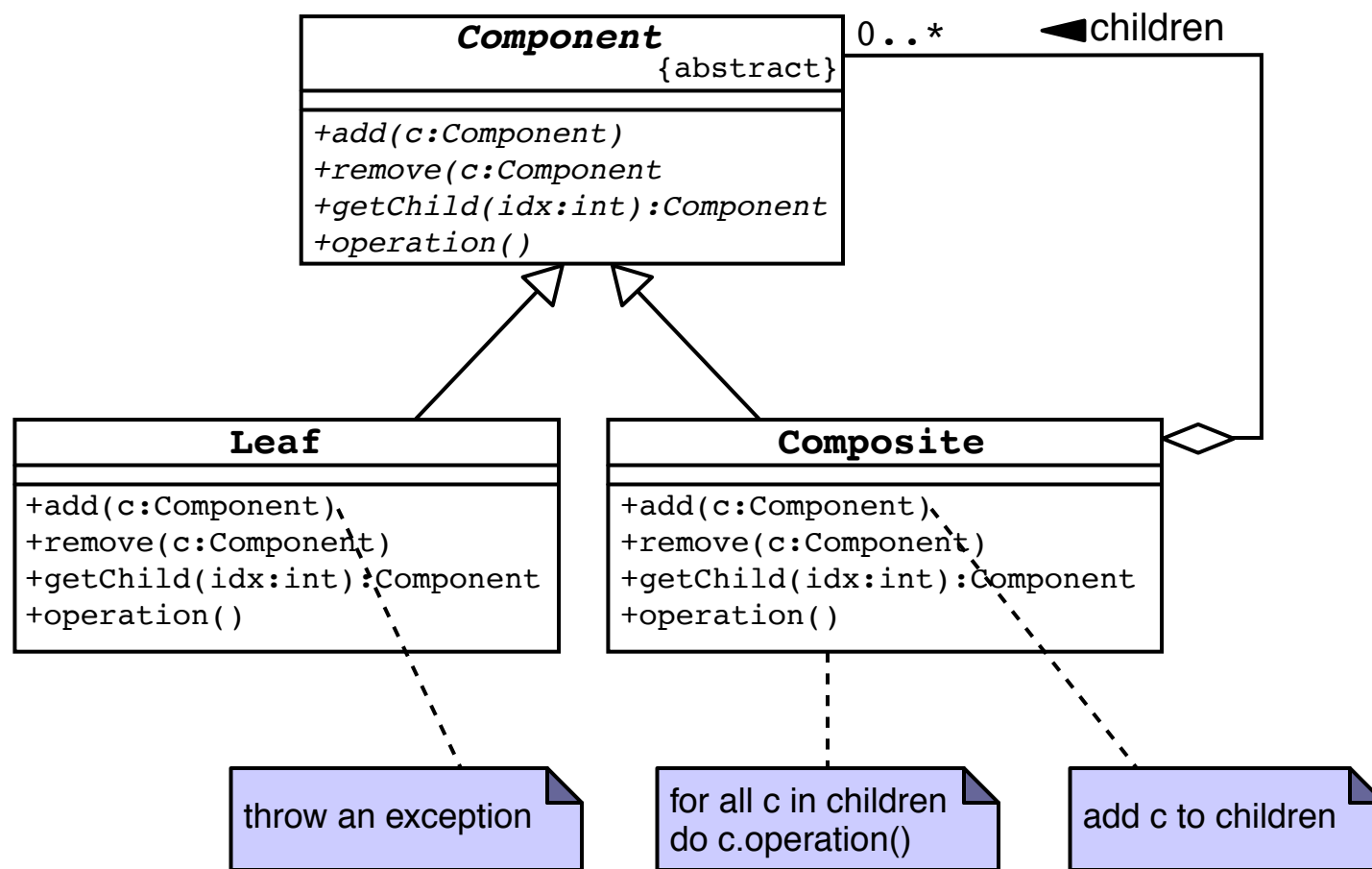
Solution:

- Create an abstract class, `Component`, that acts as the superclass for concrete “leaf” and `Composite` classes.
- The `Composite` class can be treated as a component because it supports the `Component` class interface.



Describing the Composite Pattern

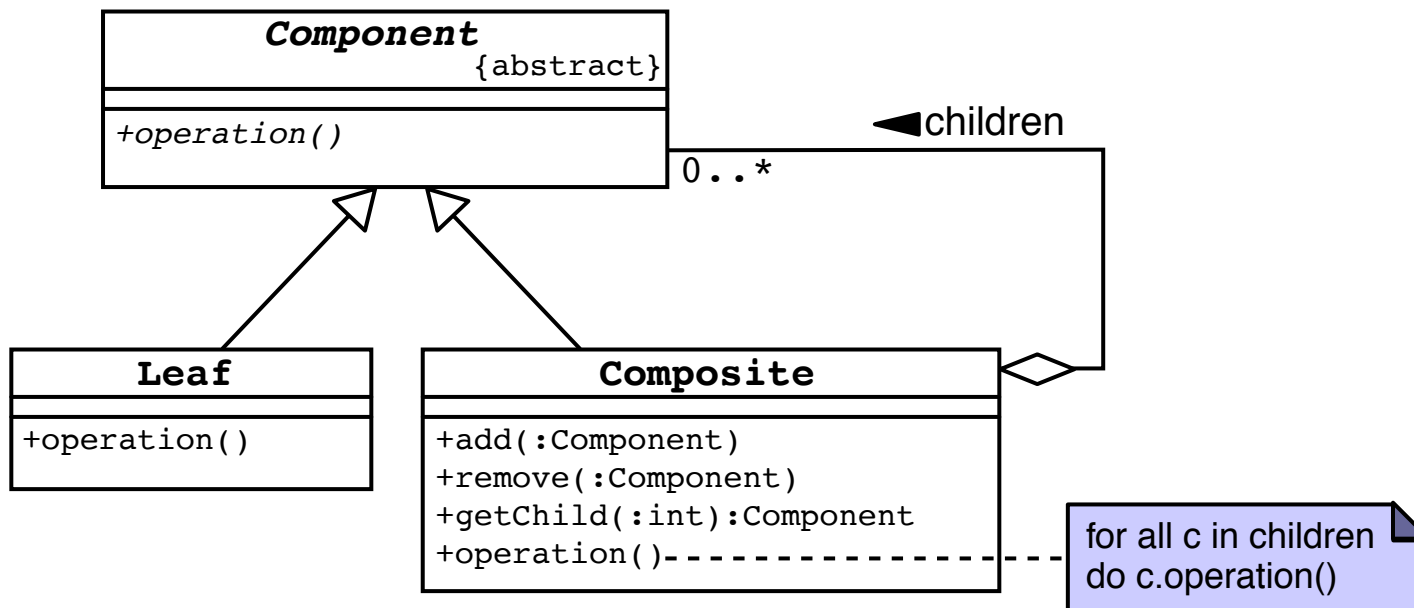
Composite (GoF) Model:





Describing the Composite Pattern

Alternate Model:





Describing the Composite Pattern

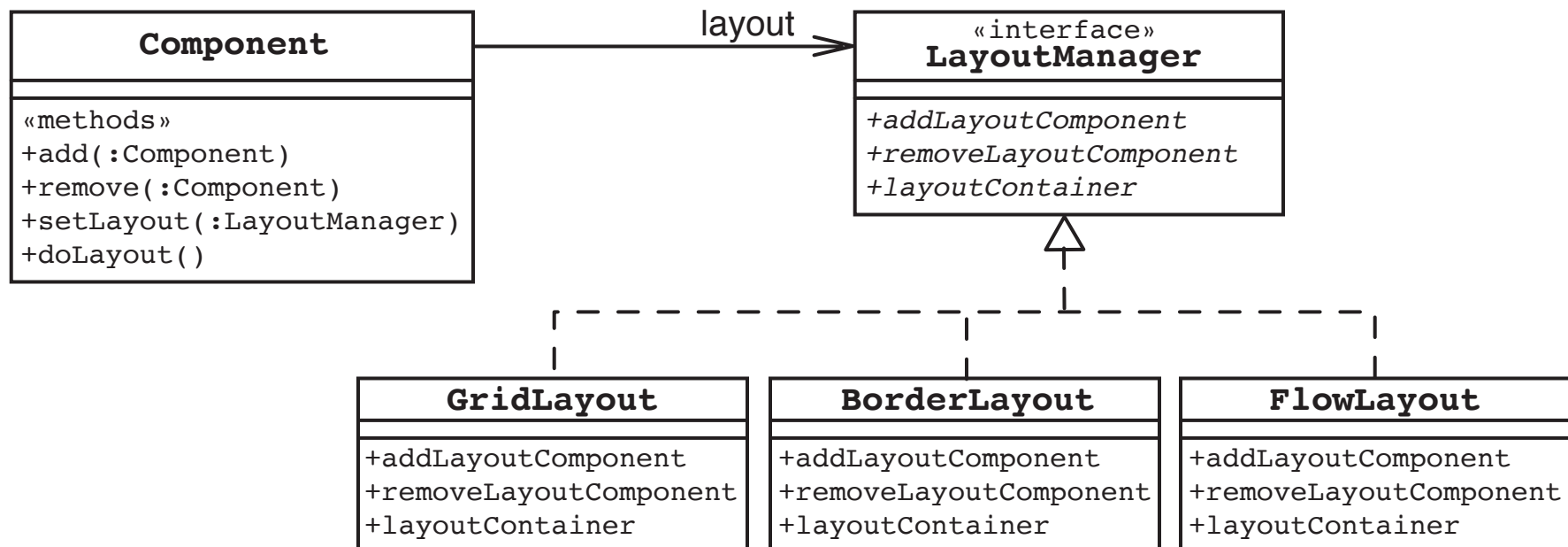
Consequences:

- Makes the client simple
- Makes it easier to add new kinds of components
- Can make the design model too general



Describing the Strategy Pattern

“Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.” (GoF page 315)





Describing the Strategy Pattern

Problem:

- You have a set of classes that are only different in the algorithms that they use
- You want to change algorithms at runtime

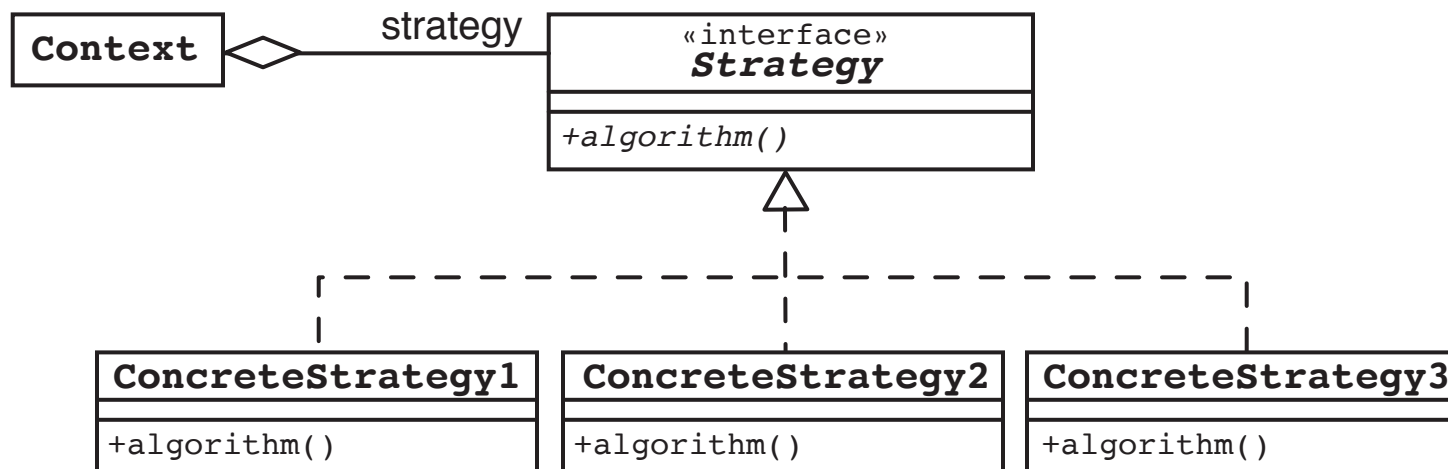
Solution:

- Create an interface, `Strategy`, that is implemented by a set of concrete “algorithm” classes.
- At runtime, select an instance of these concrete classes within the `Context` class.



Describing the Strategy Pattern

Strategy Model:





Describing the Strategy Pattern

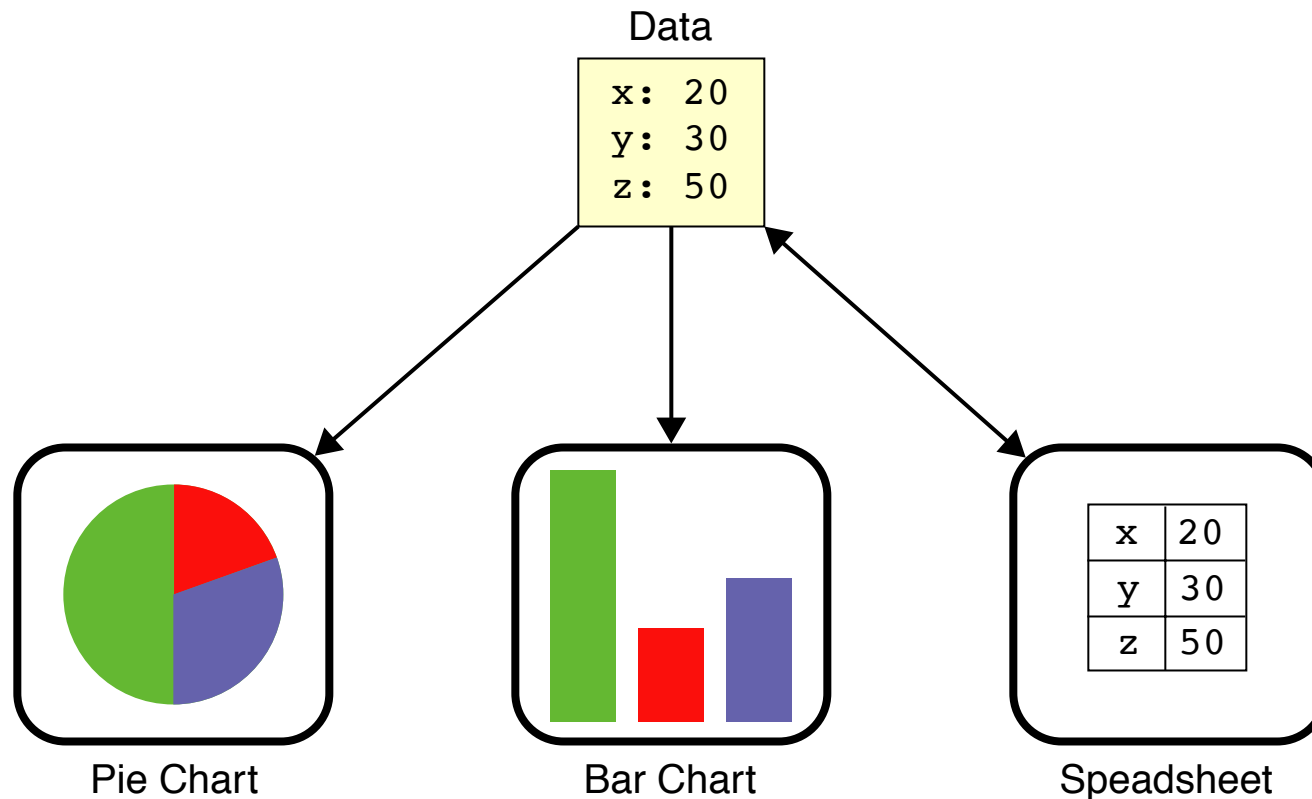
Consequences:

- An alternate to subclassing
- Strategies eliminate conditional statements
- A choice of implementations
- Communication overhead between Strategy and Context patterns
- Increased number of objects



Describing the Observer Pattern

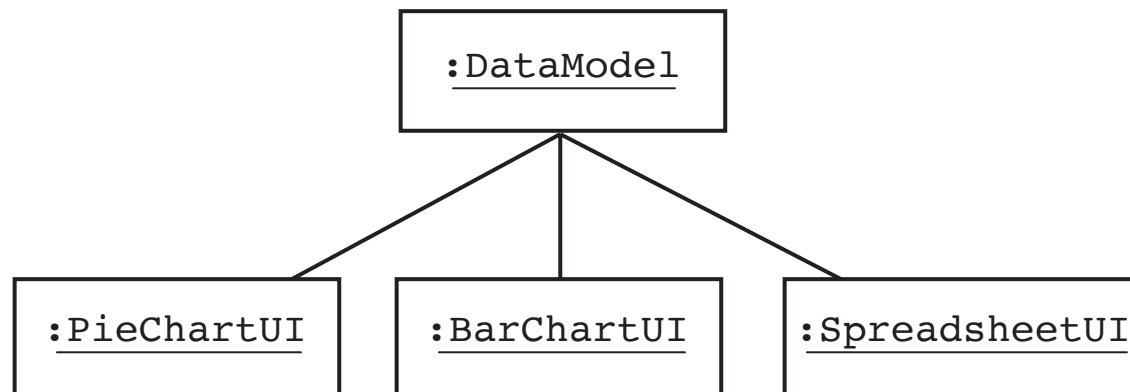
“Define a one-to-many dependency between objects so that when one object changes, all its dependents are notified and updated automatically.” (GoF page 293)





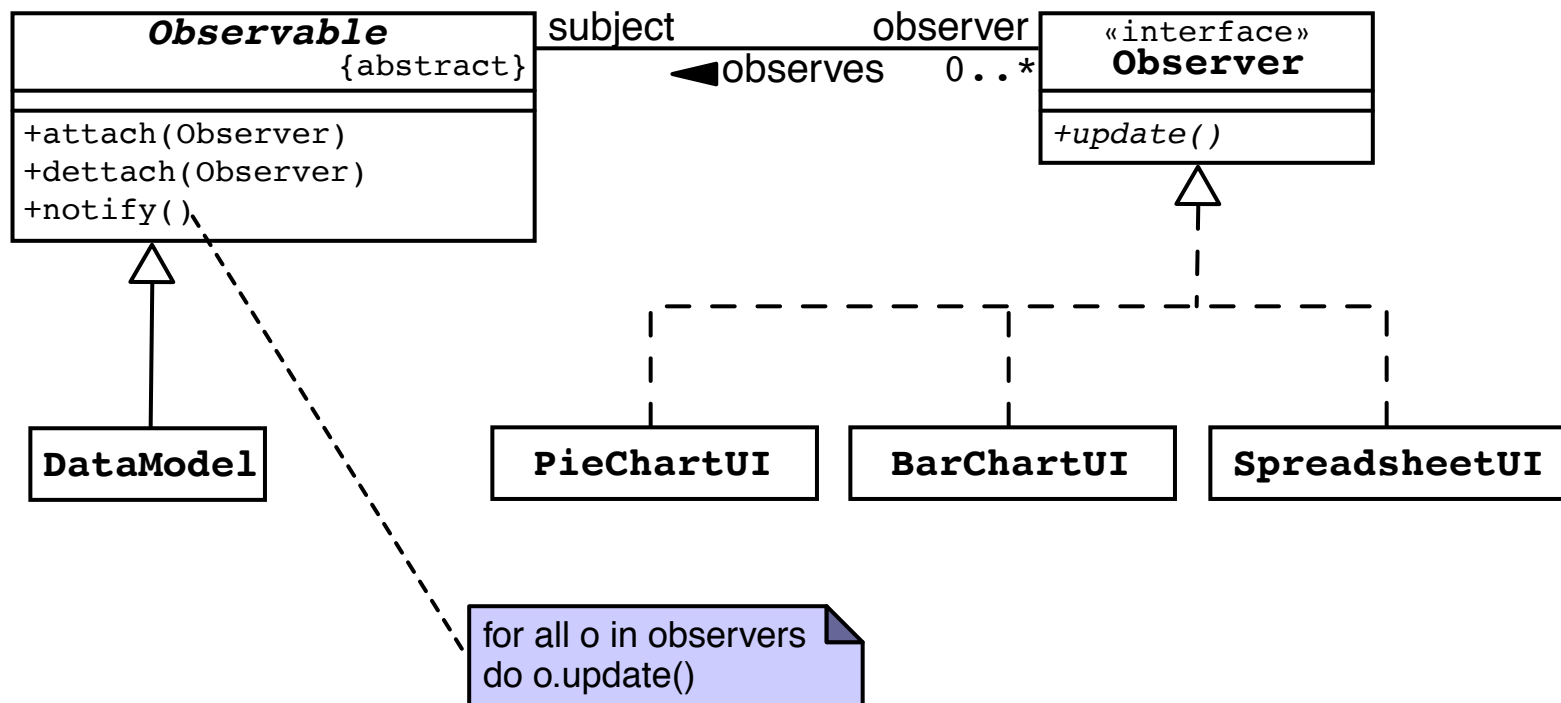
Describing the Observer Pattern

- Separate the data model class from the UI view classes.
- The UI elements are loosely coupled with the data model.





Describing the Observer Pattern





Describing the Observer Pattern

Problem:

- You need to notify a set of objects that an event has occurred.
- The set of observing objects can change at runtime.

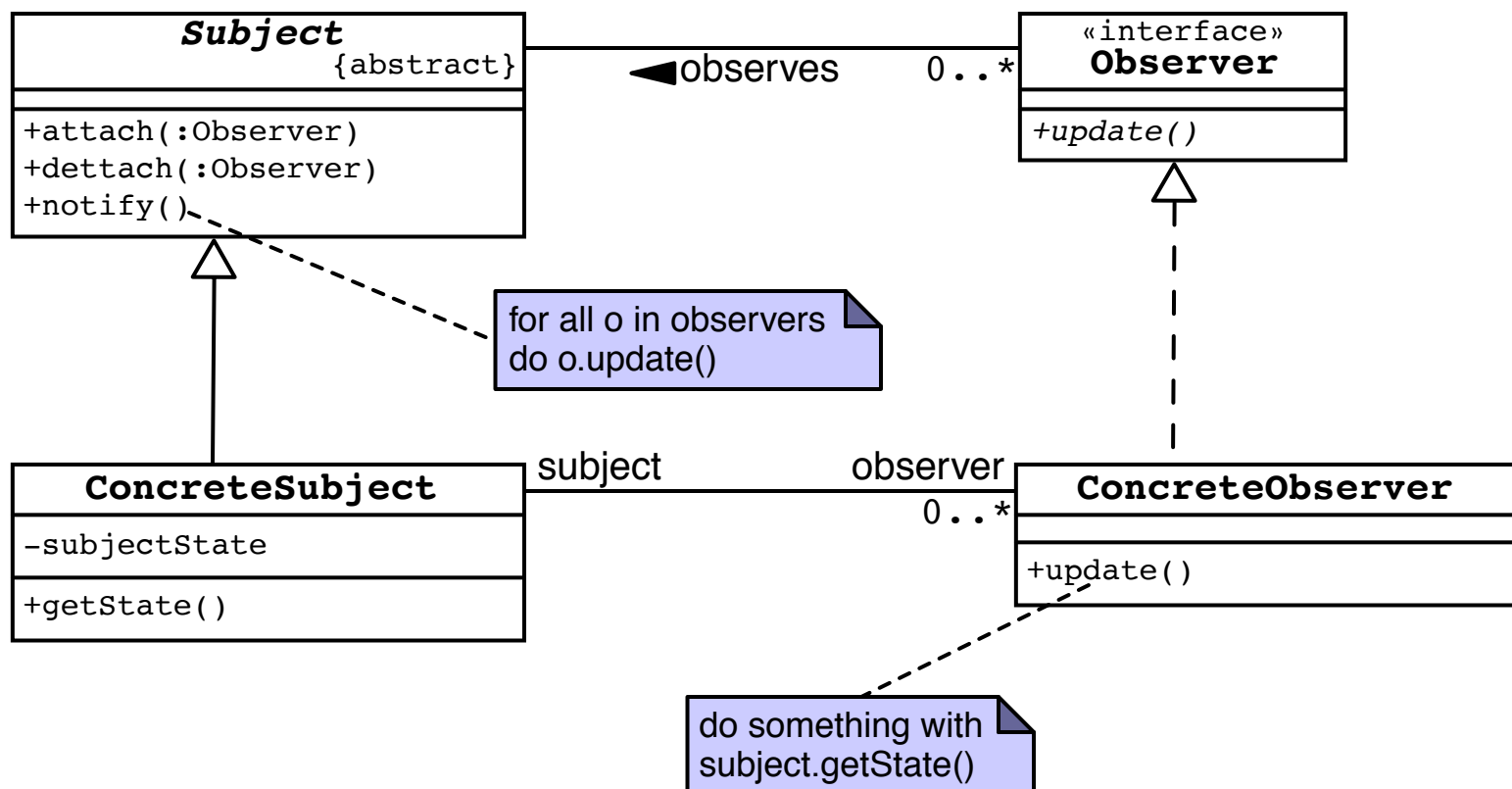
Solution:

- Create an abstract class **Subject** that maintains a collection of **Observer** objects.
- When a change occurs in a subject, it notifies all of the observers in its set.



Describing the Observer Pattern

Solution Model:





Describing the Observer Pattern

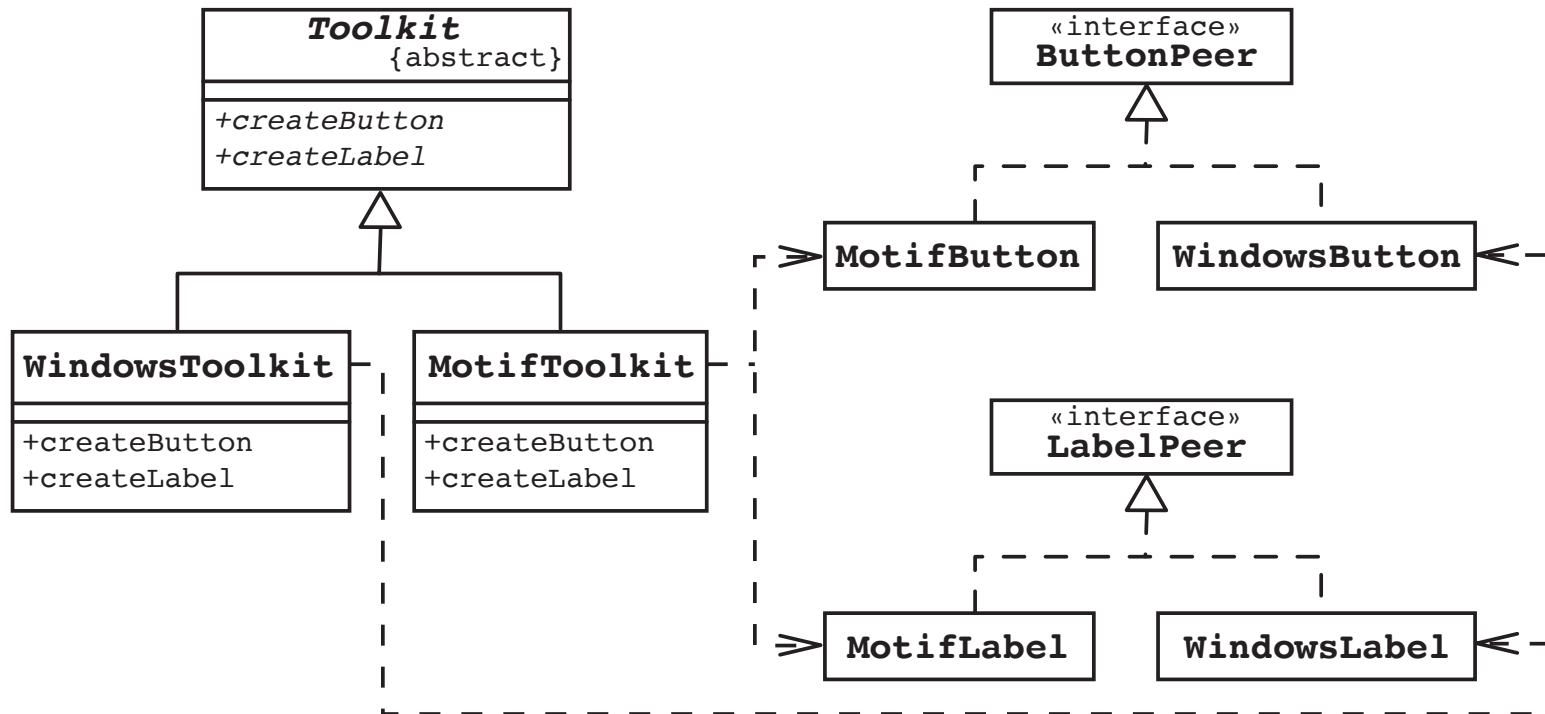
Consequences:

- Abstract coupling between Subject and Observer
- Support for multicast communication
- Unexpected updates



Describing the Abstract Factory Pattern

“Provide an interface for creating families of related or dependent objects without specifying their concrete classes.”
(GoF page 87)





Describing the Abstract Factory Pattern

Problem:

- A system has multiple families of products.
- Each product family is designed to be used together.
- You do not want to reveal the implementation classes of the product families.

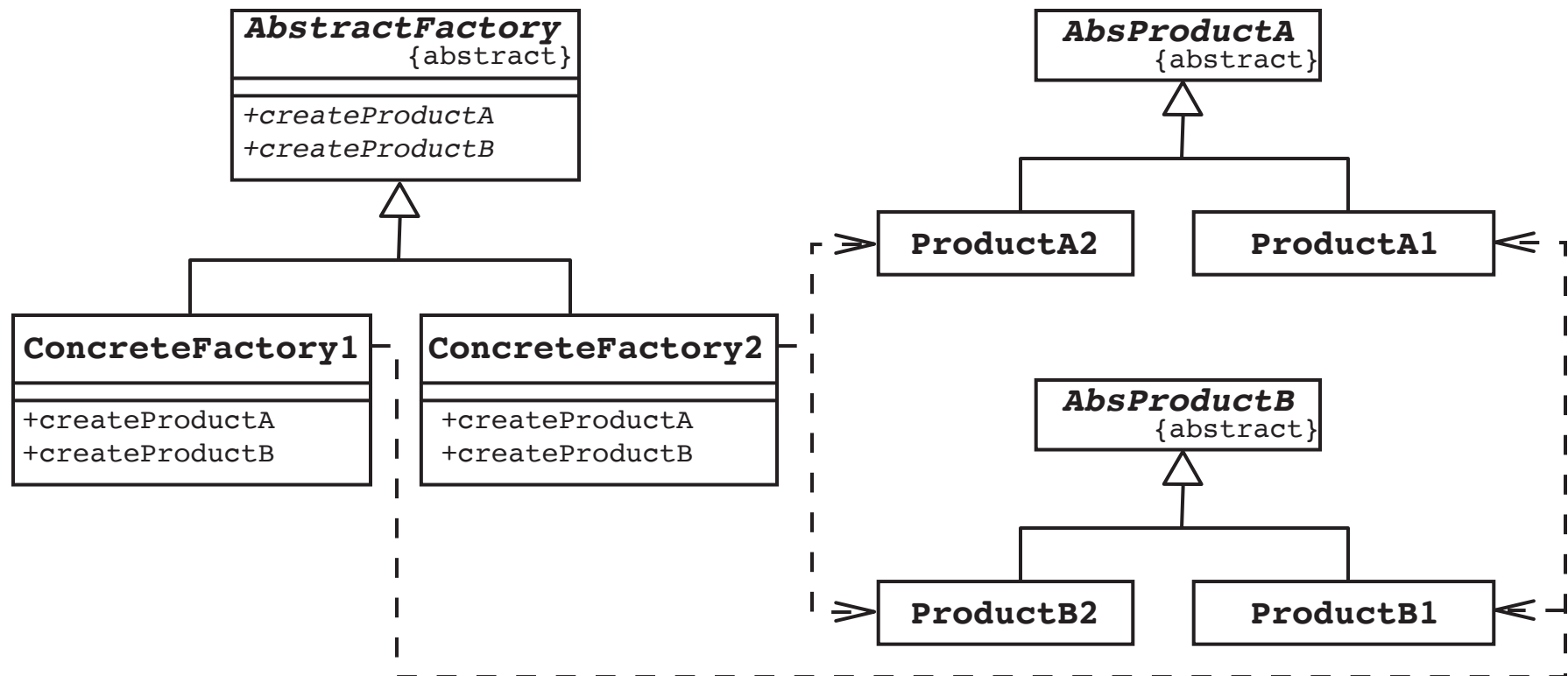
Solution:

- Create an abstract creator class that has a factory method for each type of product.
- Create a concrete creator class that implements each factory method which returns a concrete product.



Describing the Abstract Factory Pattern

Solution Model:





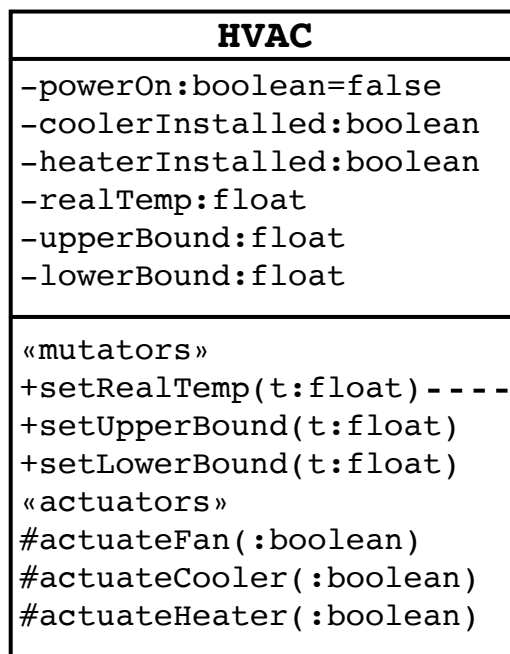
Describing the Abstract Factory Pattern

Consequences:

- Isolates concrete classes
- Makes exchanging product families easy
- Promotes consistency among products
- Supporting new kinds of products is difficult



Coding an Object with Complex State

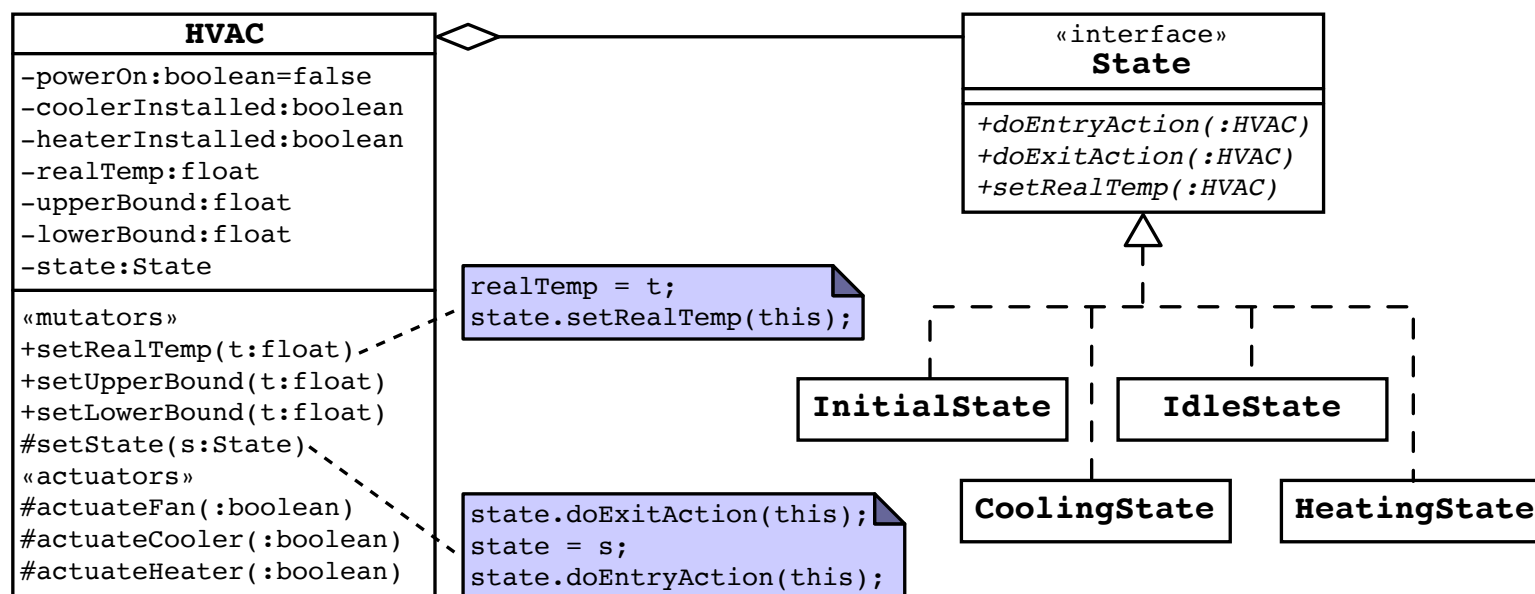


```
if ( powerOn ) {  
    if ( realTemp > upperBound ) {  
        // cooling  
        actuateFan(true);  
        if ( coolerInstalled ) {  
            actuateCooler(true);  
        }  
    } else if ( realTemp < lowerBound ) {  
        // heating  
        actuateFan(true);  
        if ( heaterInstalled ) {  
            actuateHeater(true);  
        }  
    } else {  
        // idle state  
        actuateFan(false);  
        actuateCooler(false);  
        actuateHeater(false);  
    }  
}
```



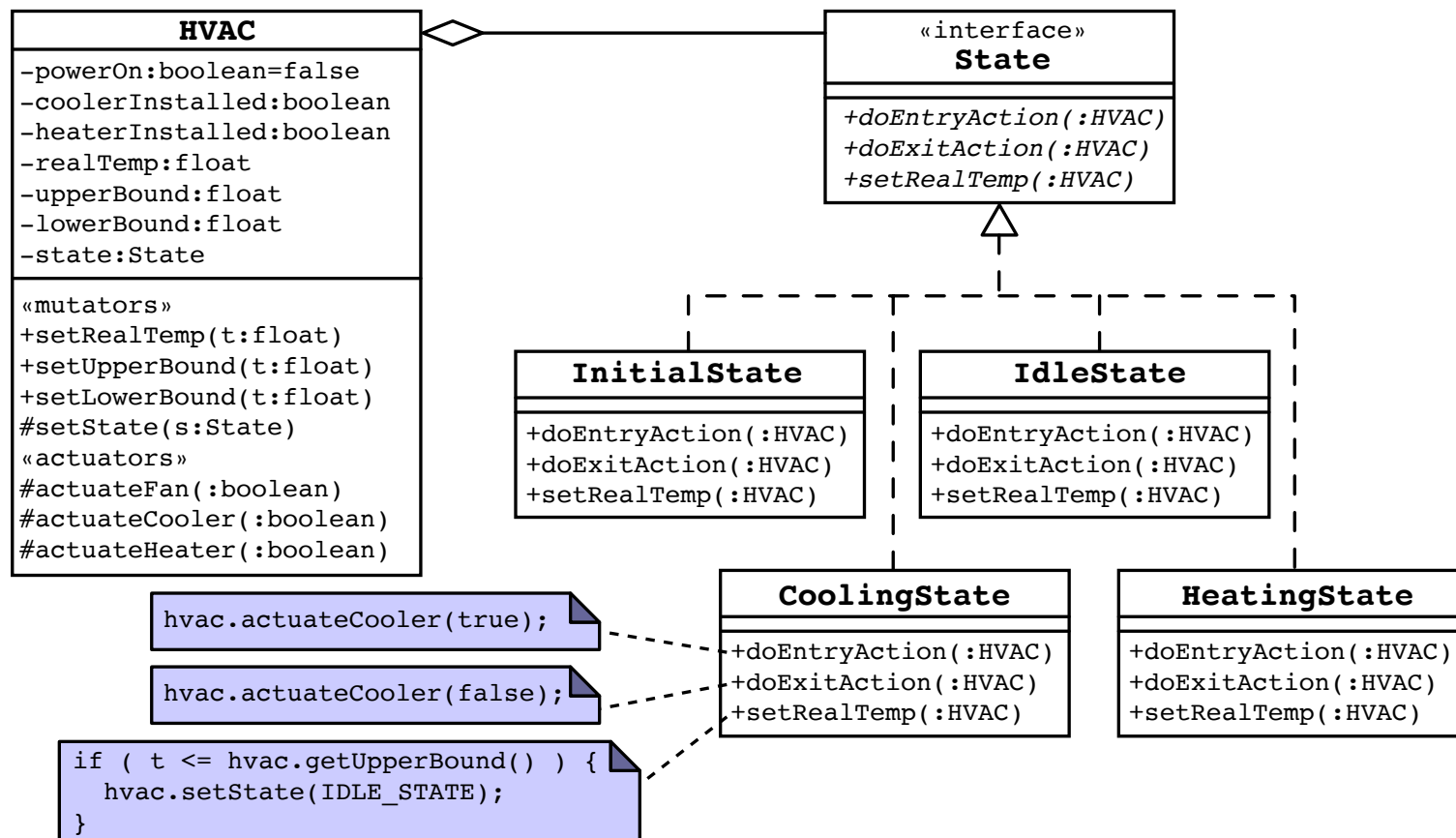

Describing the State Pattern

“Allow an object to alter its behavior when its internal state changes The object will appear to change its class.” (GoF page 305)





Describing the State Pattern





Describing the State Pattern

Problem:

- An object's runtime behavior depends on its state.
- Operations have large, multipart conditional statements that depend on the object's state.

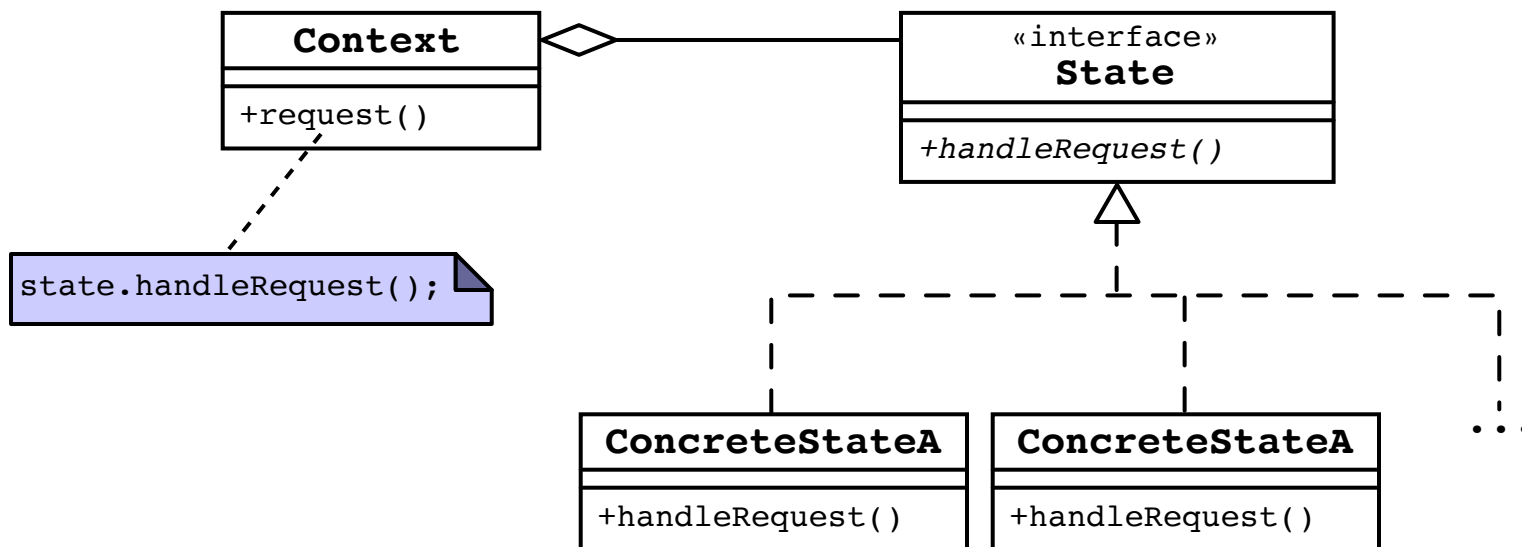
Solution:

- Create an interface that specifies the state-based behaviors of the object.
- Create a concrete implementation of this interface for each state of the object.
- Dispatch the state-based behaviors of the object to the object's current state object.



Describing the State Pattern

Solution Model:





Describing the State Pattern

Consequences:

- Localizes state-specific behavior
- Reduces conditional statements
- Makes state transitions explicit
- Increases the number of objects
- Communication overhead between the State and Context objects



Summary

- Software patterns provide proven solutions to common problems.
- Design principles provide tools to build and recognize software patterns.
- Patterns are often used together to build more flexible and robust systems and frameworks (such as AWT and JDBC).



Module 11

Introducing Architectural Concepts and Diagrams



Objectives

Upon completion of this module, you should be able to:

- Distinguish between architecture and design
- Describe tiers, layers, and systemic qualities
- Describe the Architecture workflow
- Describe the diagrams of the key architecture views
- Select the Architecture type
- Create the Architecture workflow artifacts



Justifying the Need for the Architect Role

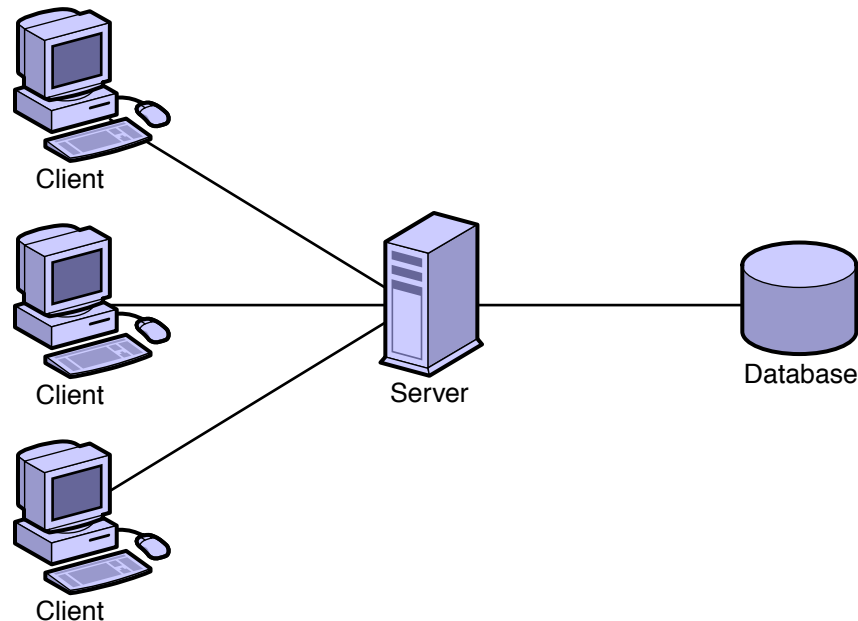
Why is it that software engineering is now employing people in this role? Because of two crucial changes:

- Scale
- Distribution



Risks Associated With Large-Scale, Distributed Enterprise Systems

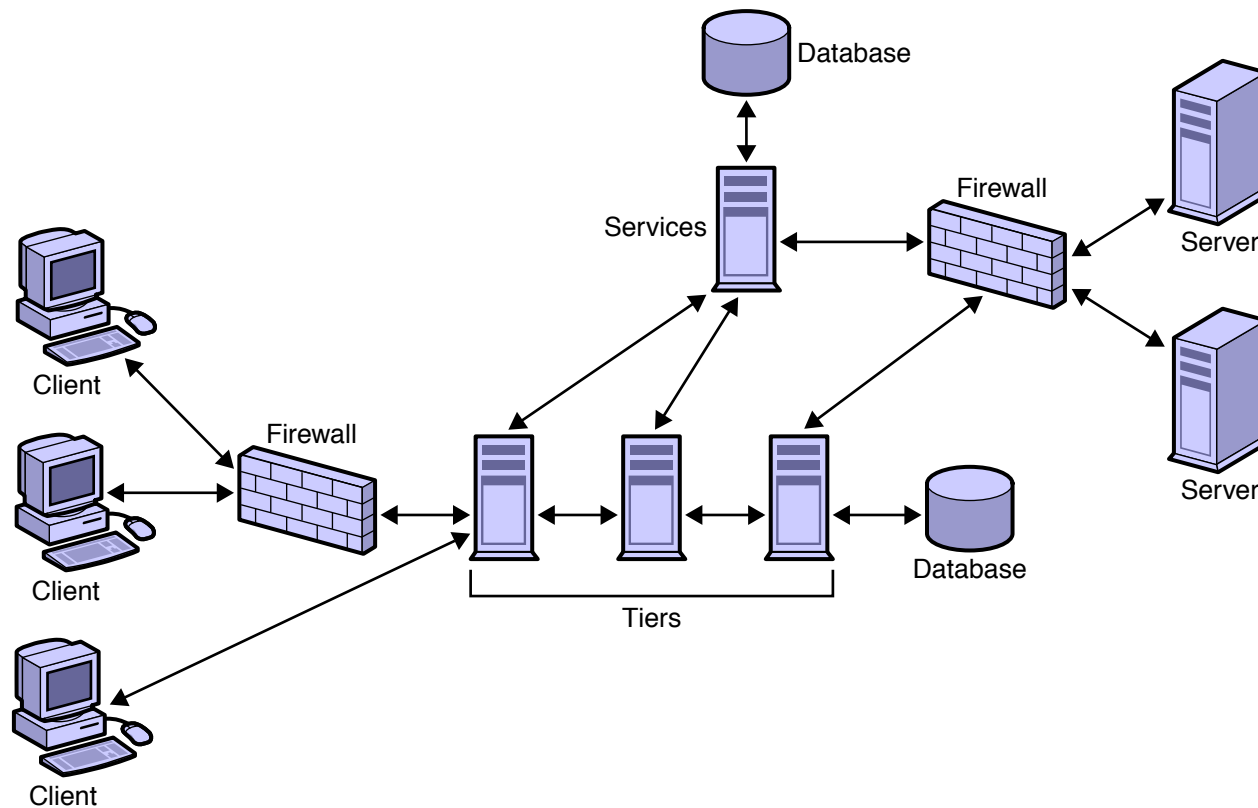
Minimally distributed systems (such as client-server)





Risks Associated With Large-Scale, Distributed Enterprise Systems

Highly distributed systems





Distinguishing Between Architecture and Design

The table shows how architects differ from designers.

	Architect	Designer
Abstraction level	High/broad Focus on few details	Low / specific Focus on many details
Deliverables	System and subsystem plans, architectural prototype	Component designs, code specifications
Area of focus	Nonfunctional requirements, risk management	Functional requirements



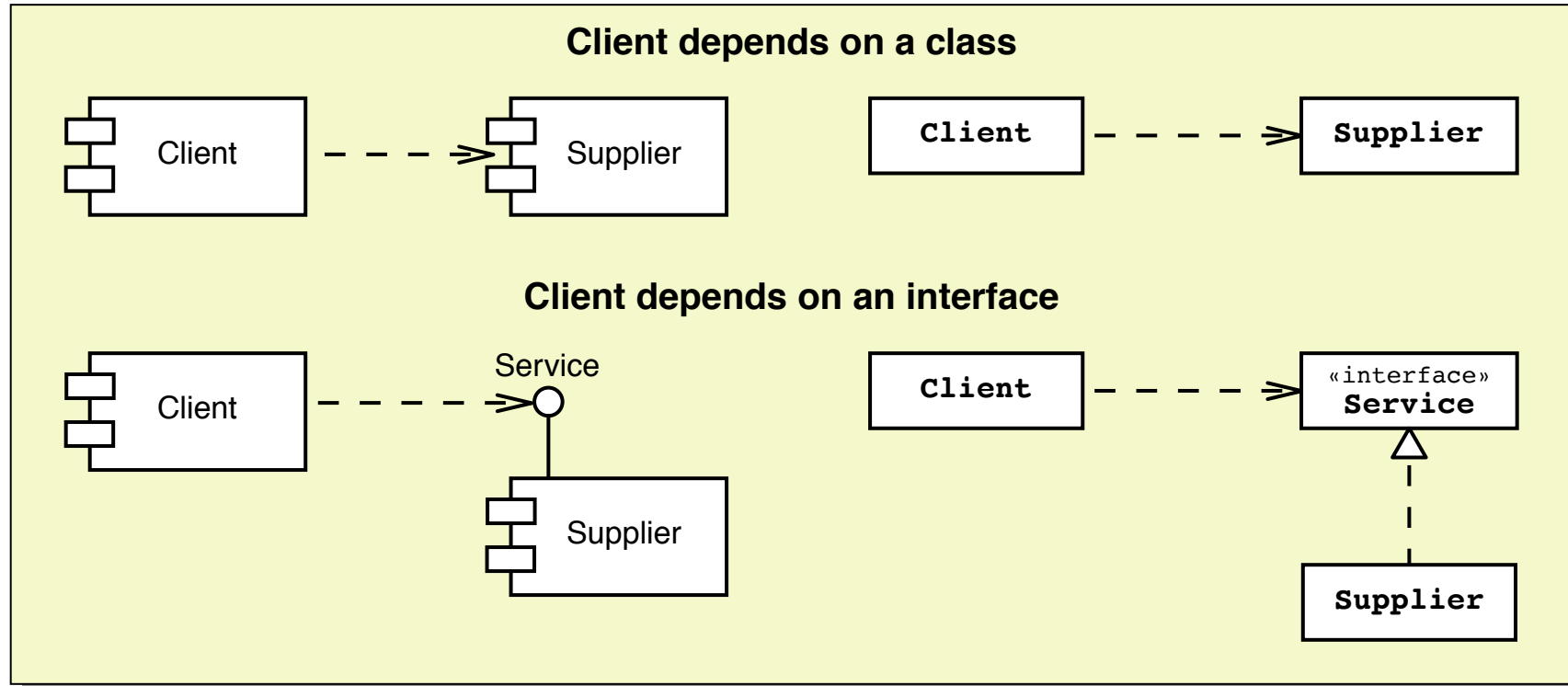
Architectural Principles

- Separation of Concerns
- Dependency Inversion Principle
- Separate volatile from stable components
- Use component and container frameworks
- Keep component interfaces simple and clear
- Keep remote component interfaces coarse-grained



Dependency Inversion Principle

“Depend on abstractions. Do not depend on concretions.”
(Knoernschild page 12)





Architectural Patterns and Design Patterns

- An architect plans systems using a pattern-based reasoning process.
- An architect must be familiar with a variety of pattern catalogs to be effective.
- Types of patterns:
 - Design patterns define structure and behavior to construct effective and reusable OO software components to support functional requirements.
 - Architectural patterns define structure and behavior for systems and subsystems to support nonfunctional requirements.



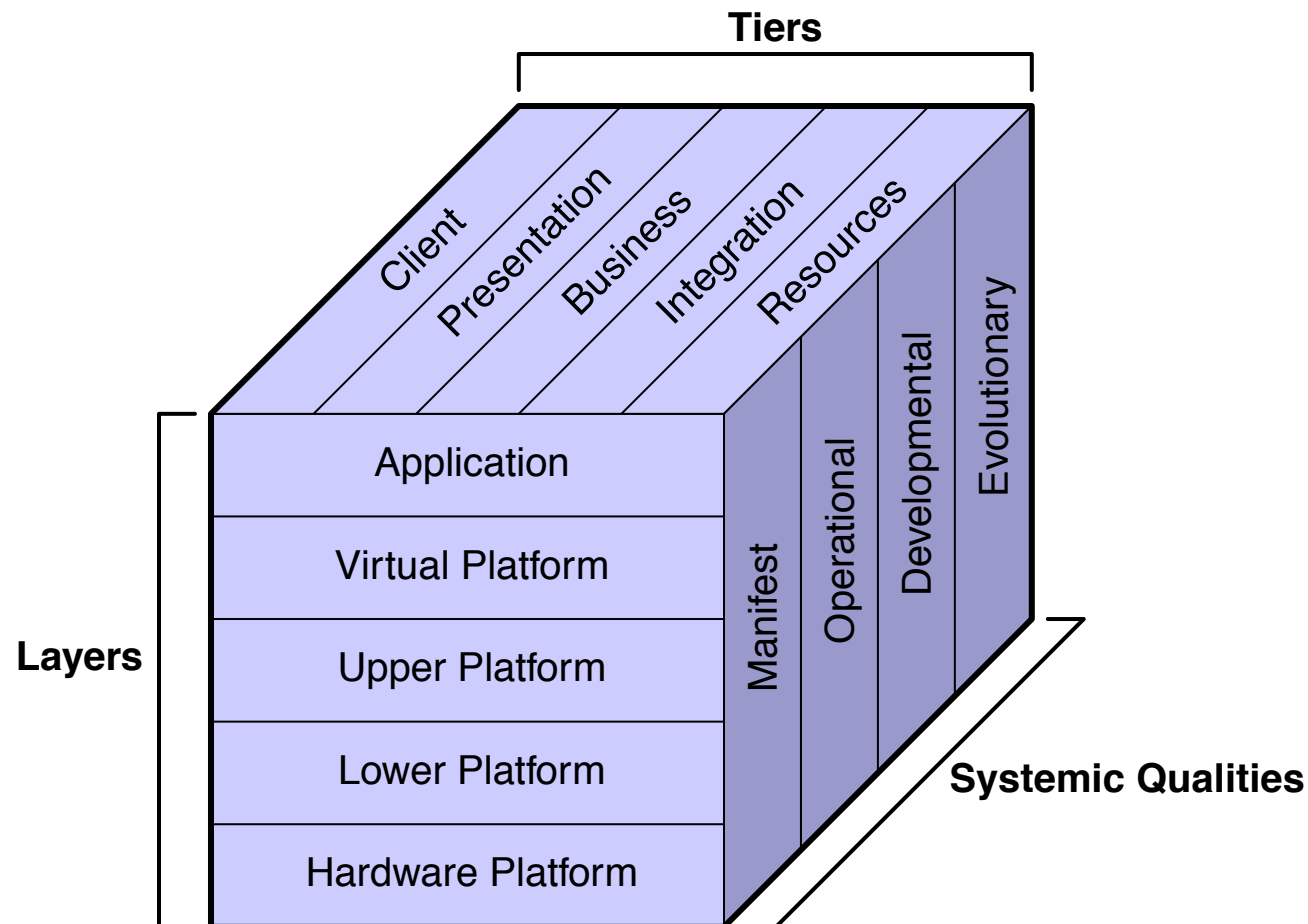
Tiers, Layers, and Systemic Qualities

The SunTone Architecture Methodology recommends the following architectural dimensions:

- The tiers to separate the logical concerns of the application
- The layers to organize the component and container relationships
- The systemic qualities identify strategies and patterns across the tiers and layers



Applying the SunTone Architecture Methodology





Tiers

tiers – “A logical or physical organization of components into an ordered chain of service providers and consumers.”
(SunTone Architecture Methodology page 10)

- Client – Consists of a thin client, such as a web browser.
- Presentation – Provides the HTML pages and forms that are sent to the Web browser and processes the user's requests.
- Business – Provides the business services and entities.
- Integration – Provides components to integrate the Business tier with the Resource tier.
- Resource – Contains all backend resources, such as a DataBase Management System (DBMS) or Enterprise Information System (EIS).



Layers

layers – “The hardware and software stack that hosts services within a given tier. (layers represent component/container relationships)” (SunTone Architecture Methodology page 11)

- Application – Provides a concrete implementation of components to satisfy the functional requirements.
- Virtual Platform – Provides the APIs that application components implement.
- Upper Platform – Consists of products such as web and EJB technology containers and middleware.
- Lower Platform – Consists of the operating system.
- Hardware Platform – Includes computing hardware such as servers, storage, and networking devices.



Systemic Qualities

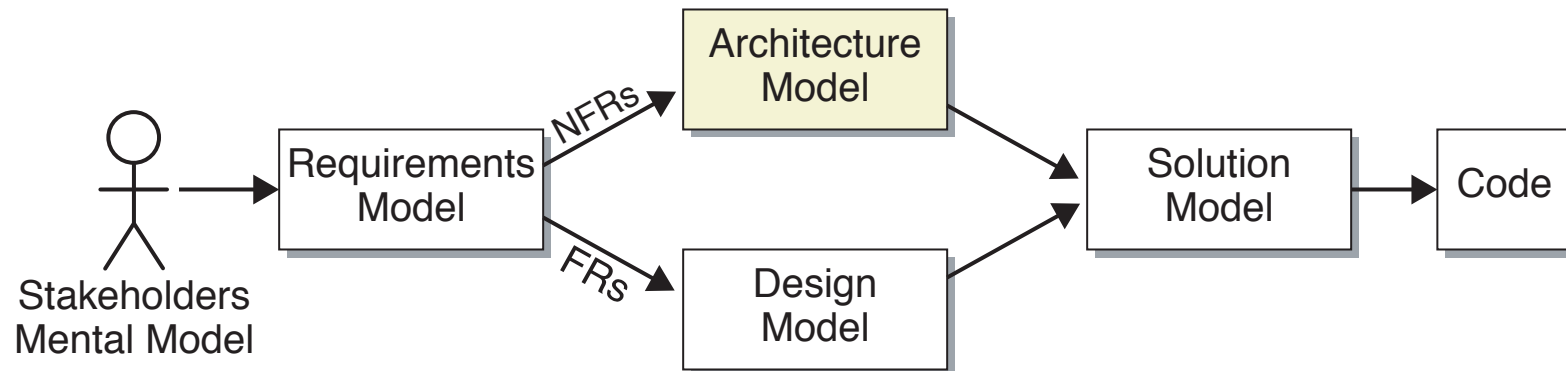
“The strategies, tools, and practices that will deliver the requisite quality of service across the tiers and layers.” (SunTone Architecture Methodology page 11)

- Manifest – Addresses the qualities reflected in the end-user experience.
- Operational – Addresses the qualities reflected in the execution of the system.
- Developmental – Addresses the qualities reflected in the planning, cost, and physical implementation of the system.
- Evolutionary – Addresses the qualities reflected in the long-term ownership of the system.



Exploring the Architecture Workflow

The Architecture model is essential to the creation of the Solution model:



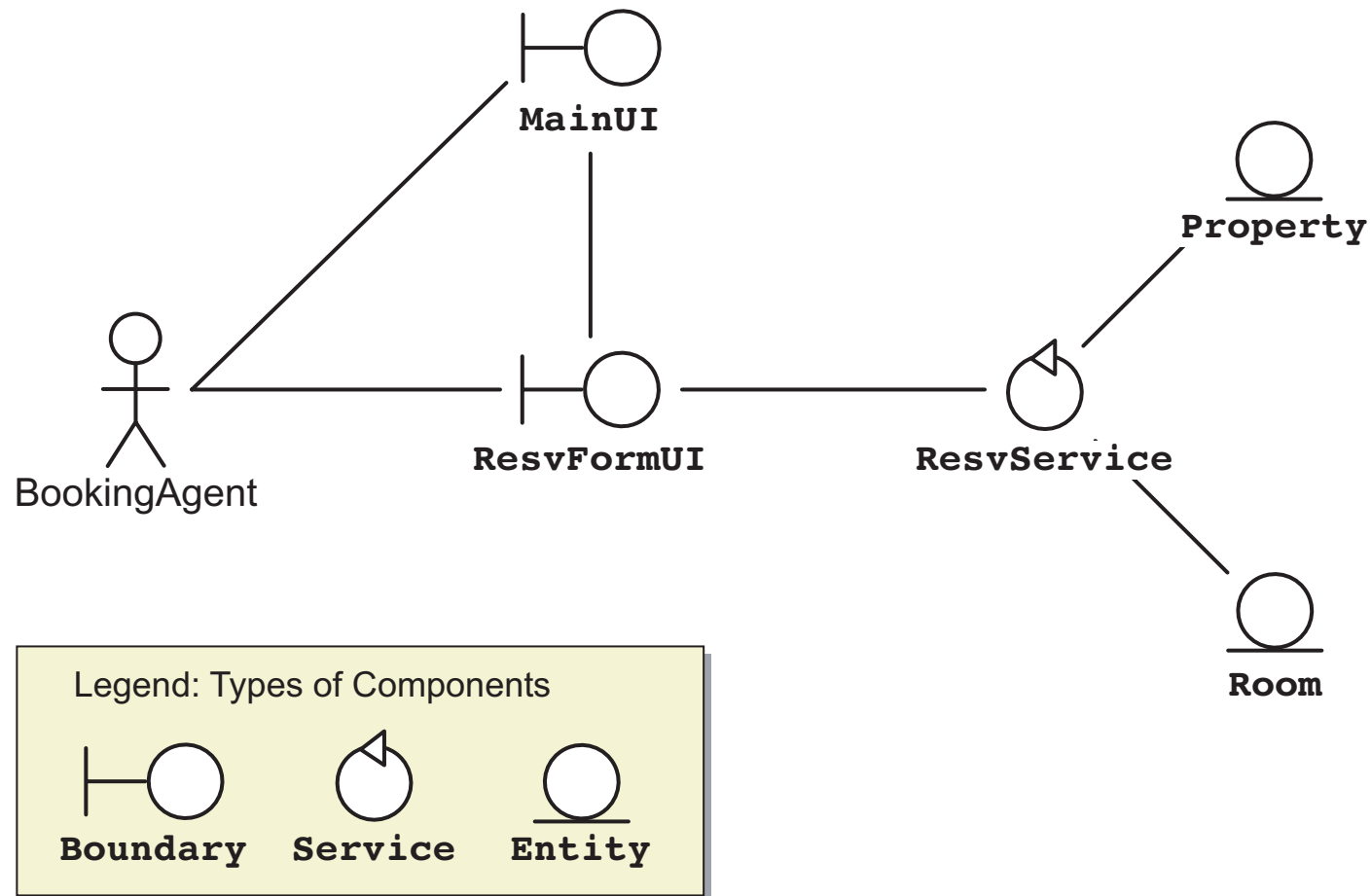


Introducing the Architecture Workflow

1. Select an architecture type for the system.
2. Create a detailed Deployment diagram for the architecturally significant use cases.
3. Refine the Architecture model to satisfy the NFRs.
4. Create and test the Architecture baseline.
5. Document the technology choices in a tiers and layers Package diagram.
6. Create an Architecture template from the final, detailed Deployment diagram.

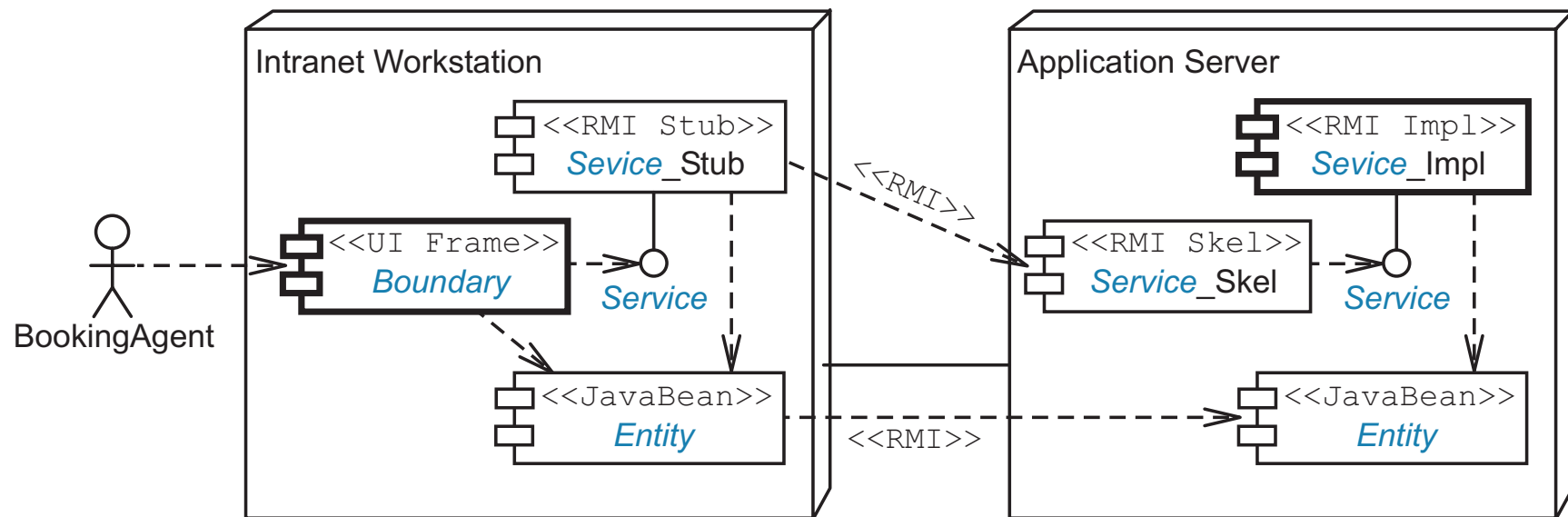


Example Design Model



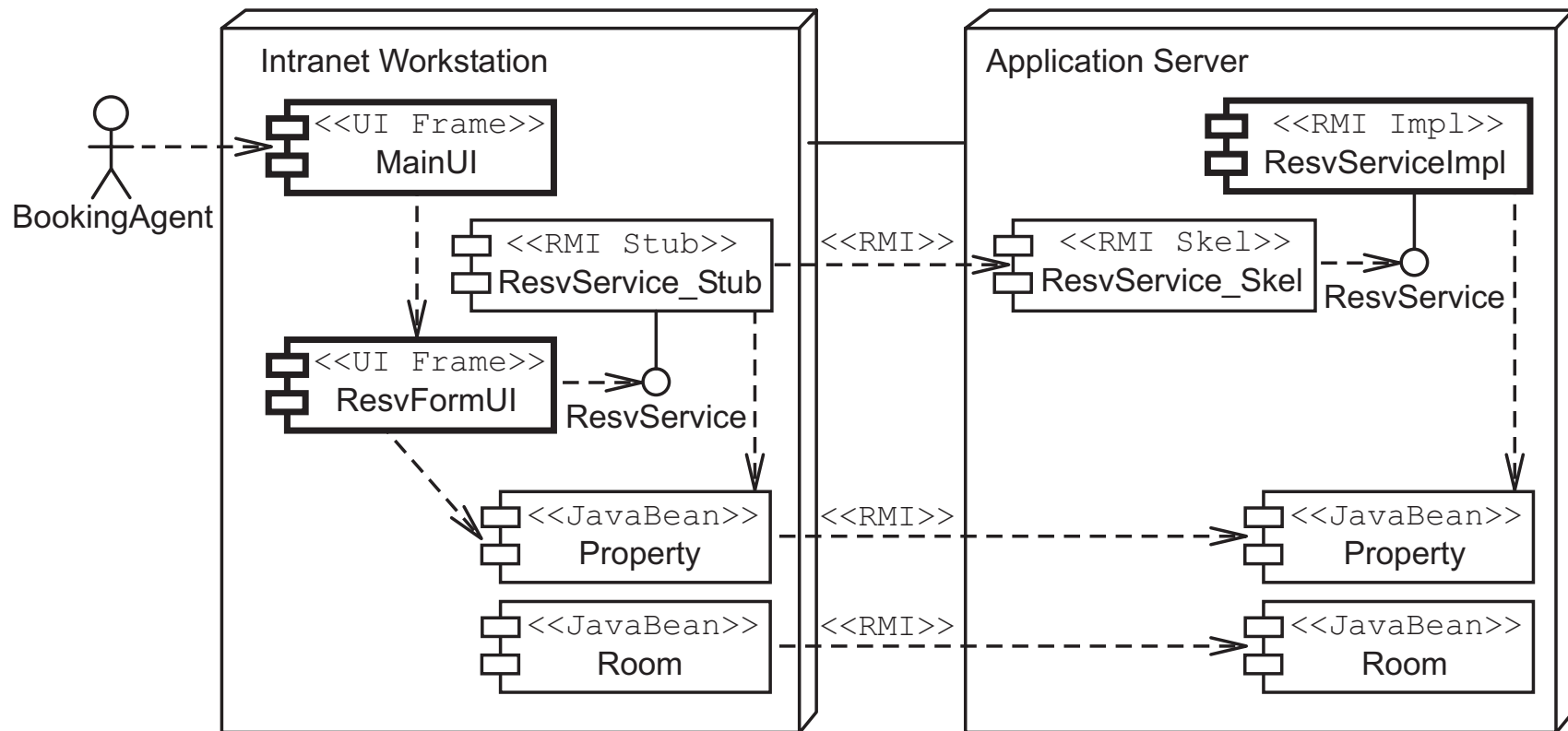


Example Architecture Template





Example Solution Model





Architectural Views

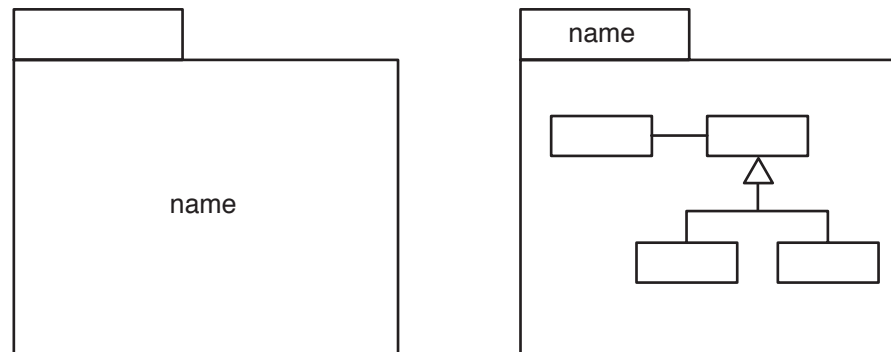
The views of the Architecture model take many forms. Some elements (such as risk mitigation plans) are documented with text. Others can be recorded using UML diagrams:

- Package diagrams
- Component diagrams
- Deployment diagrams



Identifying the Elements of a Package Diagram

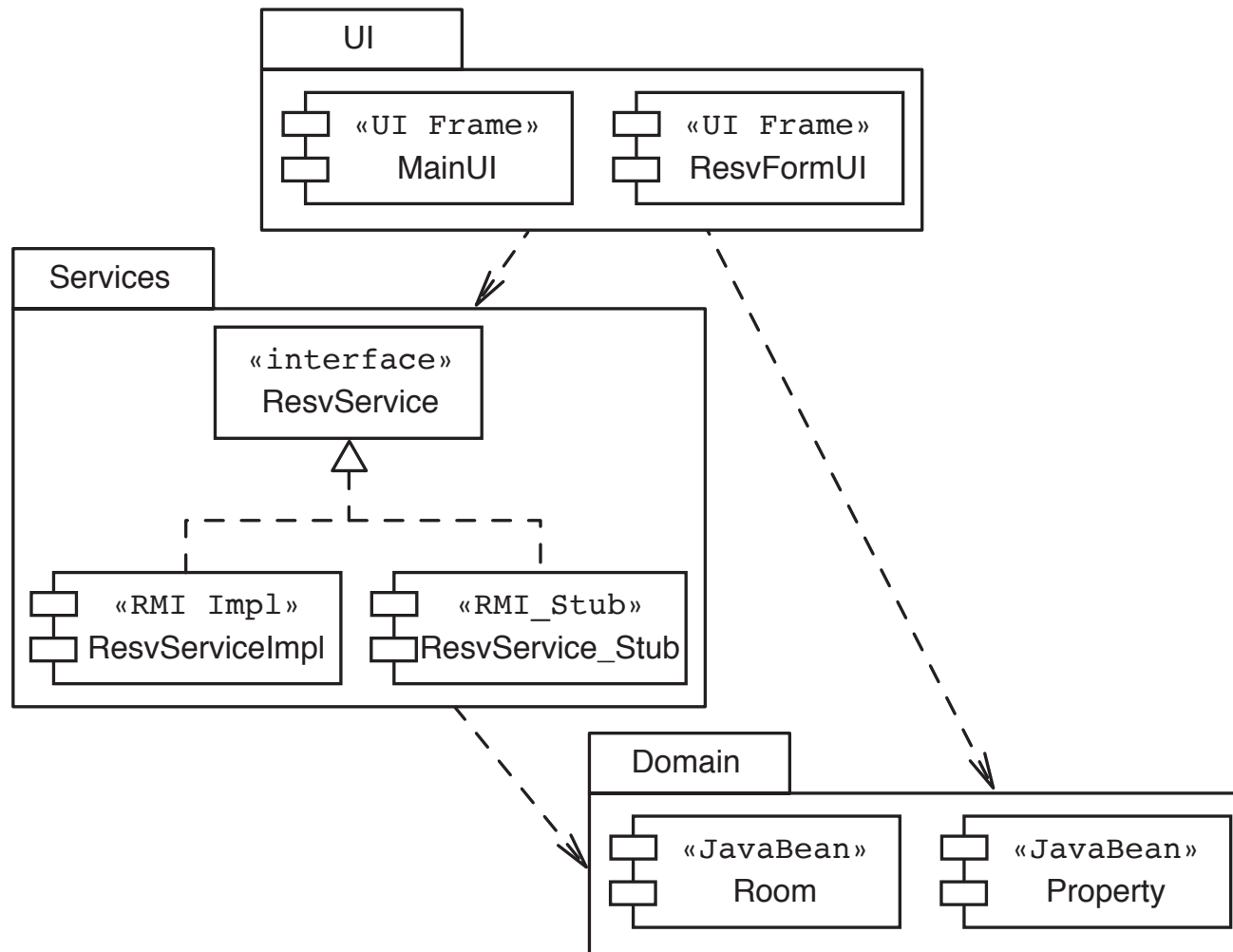
- A UML package diagram shows dependencies between packages, which can hold any UML element.
- The UML package notation:



- The package name can be placed in the body box or in the name box.
- You can place any UML entity in a package, including other packages.

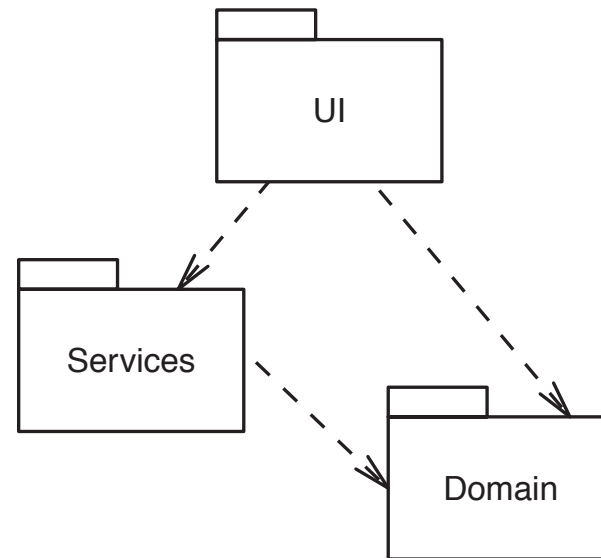


Example Package Diagram





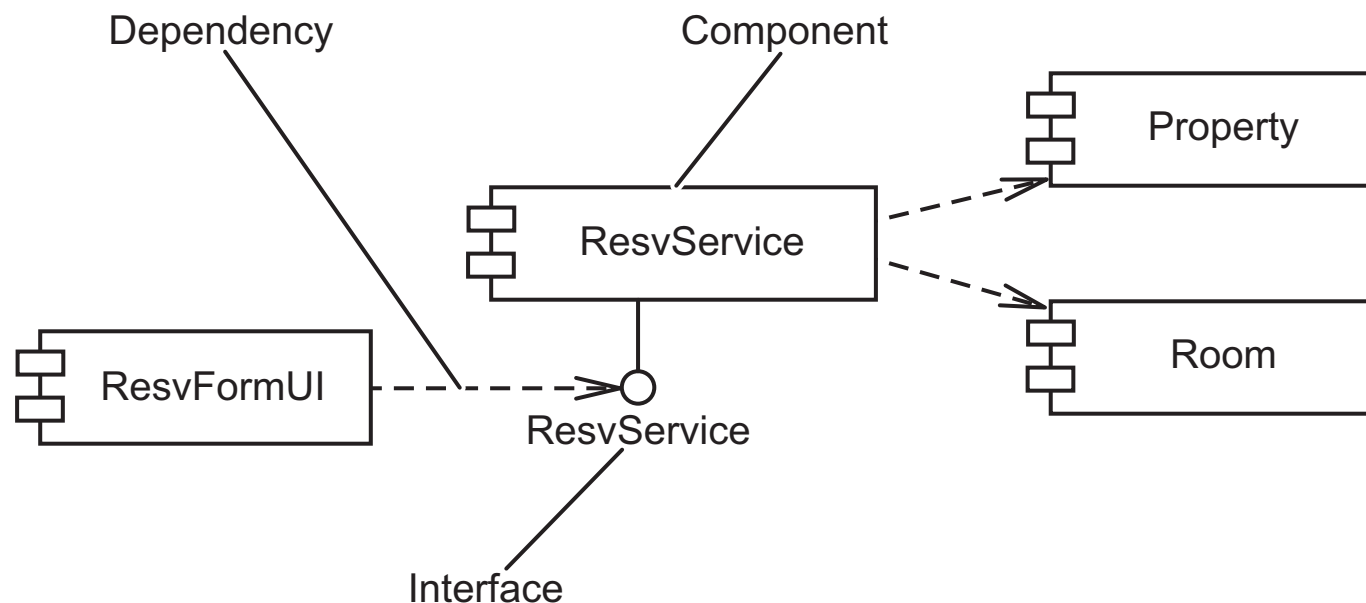
An Abstract Package Diagram





Identifying the Elements of a Component Diagram

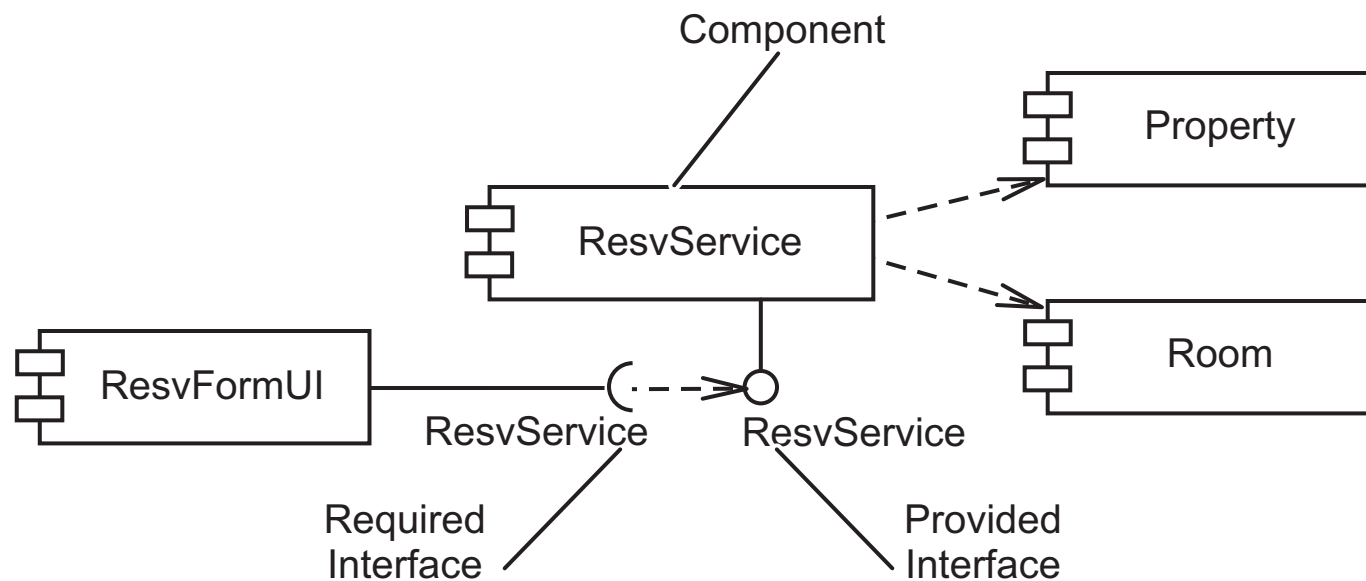
A UML Component diagram is composed of the following elements:





Identifying the Elements of a Component Diagram

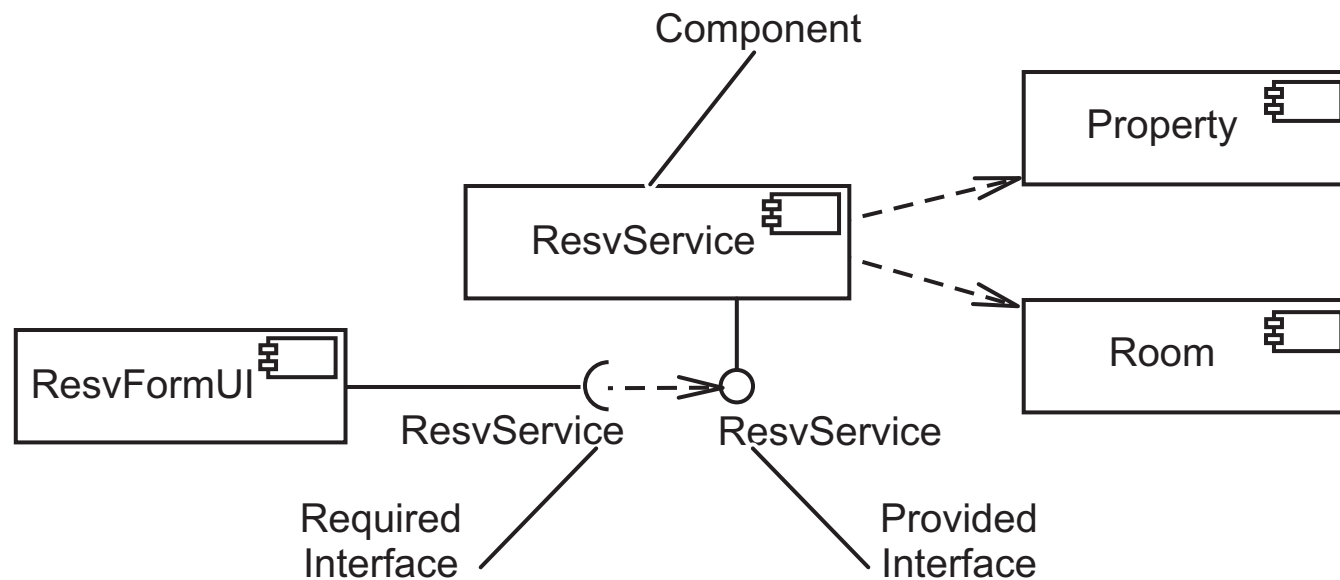
Notation for required interfaces in UML 2:





Identifying the Elements of a Component Diagram

Component notation in UML 2:





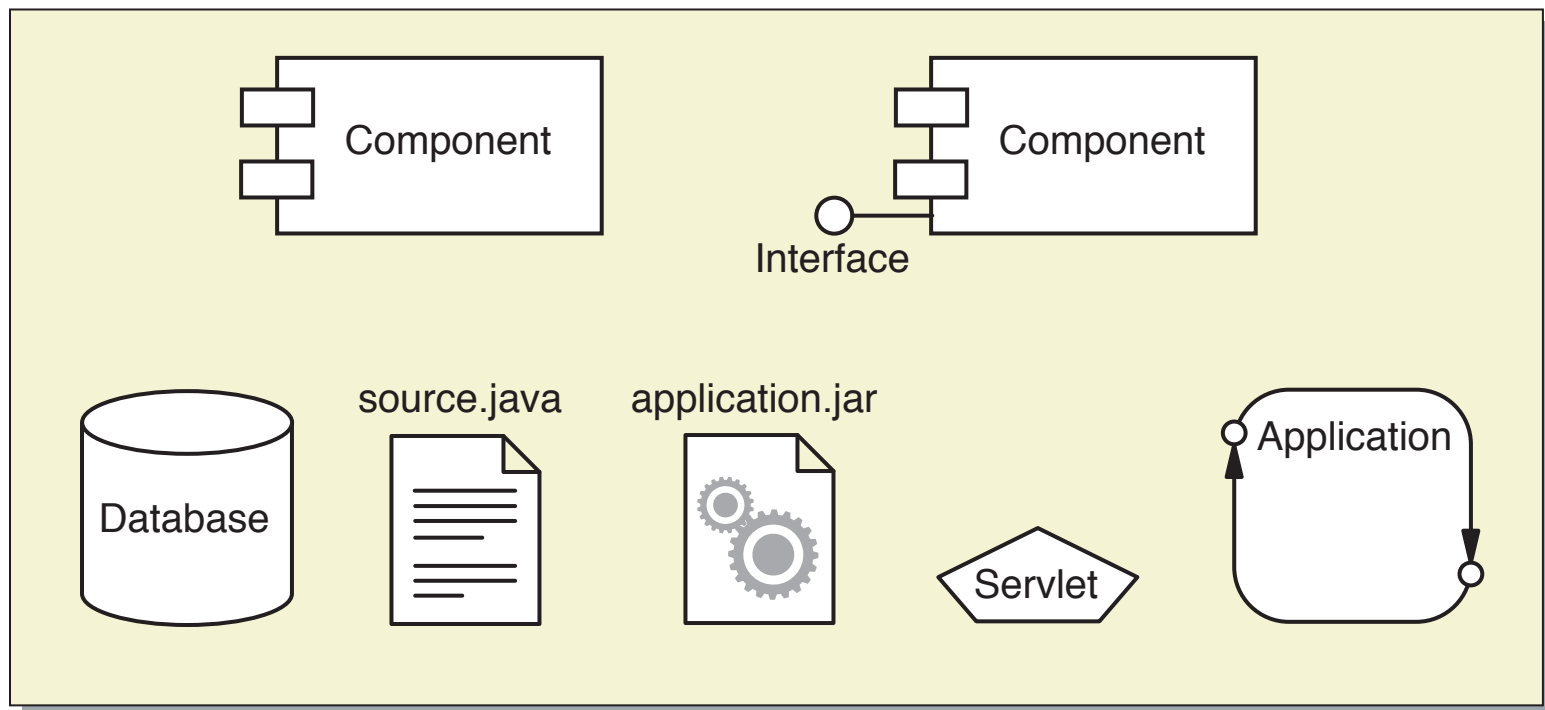
Characteristics of a Component

- A component represents any *software unit*.
- A component can be large and abstract.
- A component can be small.
- A component might have an interface that it exports as a service to other components.
- A component can be a file, such as a source code file, an *object* file, a complete executable, a data file, HTML or media files, and so on.



Types of Components

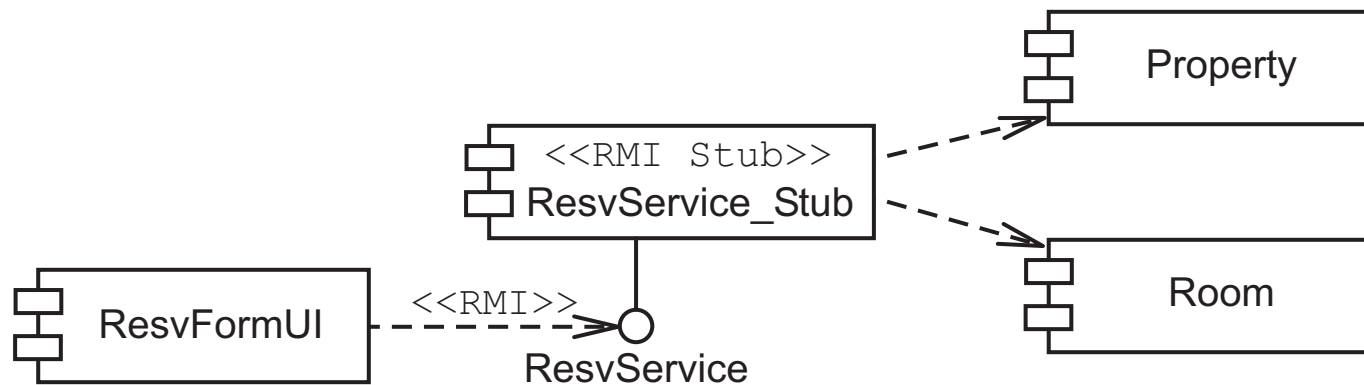
A component is any physical software unit:





Example Component Diagrams

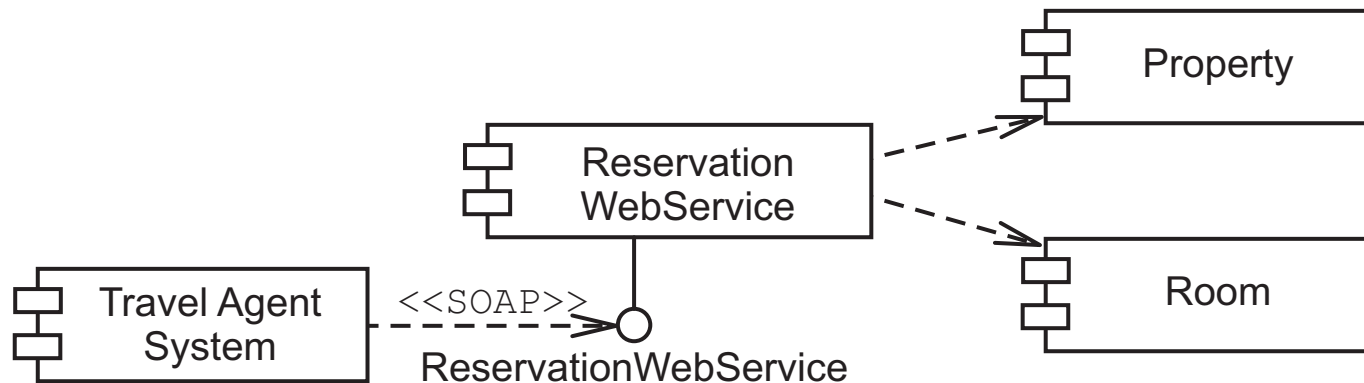
Component diagrams can show software dependencies. This example shows an RMI dependency:





Example Component Diagrams

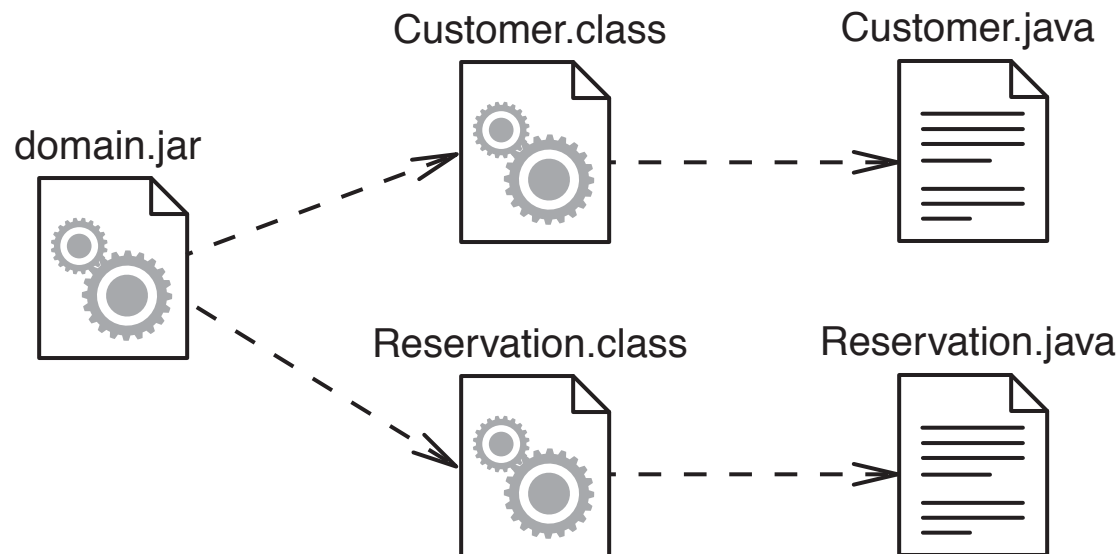
Component diagrams can show software dependencies. This example shows a Web Service dependency:





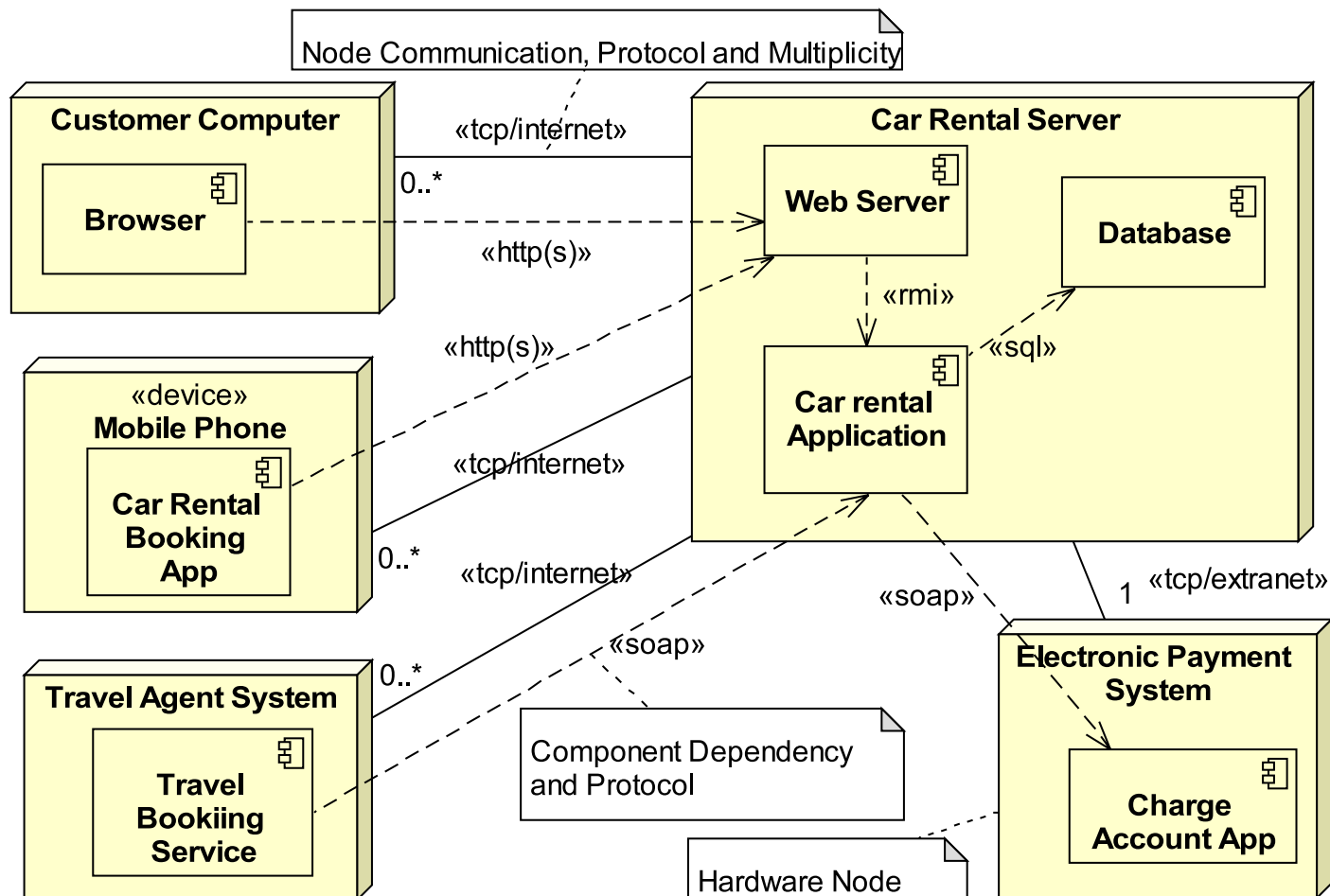
Example Component Diagrams

Component diagrams can represent build structures:





Identifying the Elements of a Deployment Diagram





The Purpose of a Deployment Diagram

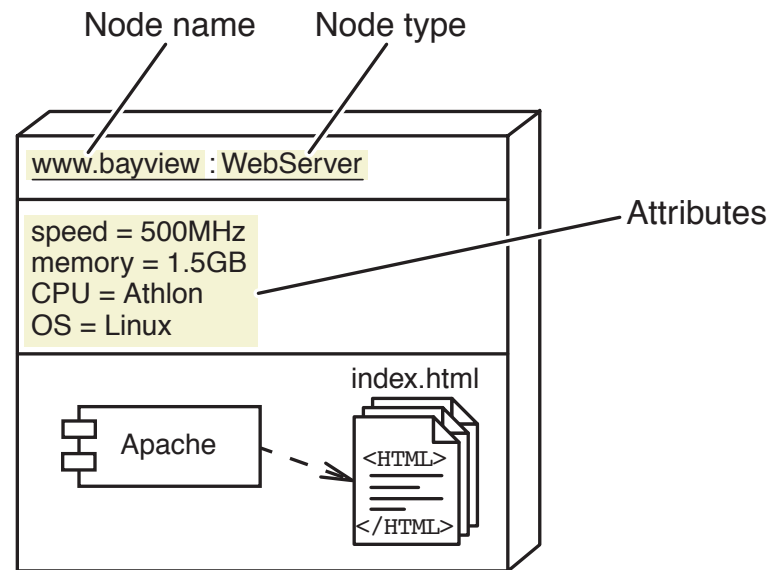
- Hardware nodes can represent any type of physical hardware.
- Links between hardware nodes indicate connectivity and can include the communication protocol used between nodes.
- Software components are placed within hardware nodes to show the distribution of the software across the network.



Types of Deployment Diagrams

There are two forms:

- A *descriptor* Deployment diagram shows the fundamental hardware configuration.
- An *instance* Deployment diagram shows a specific hardware configuration. For example:





Selecting the Architecture Type

The architecture you use depends on many factors, including:

- The platform constraints in the system requirements
- The modes of user interaction
- The persistence mechanism
- Data and transactional integrity



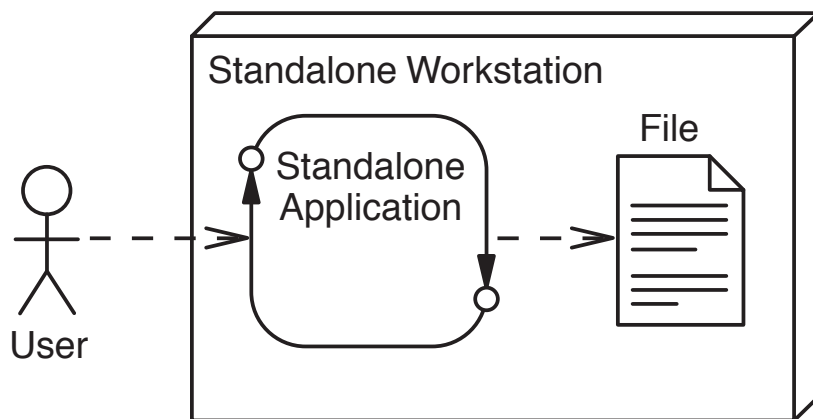
Selecting the Architecture Type

There are hundreds of successful software architectures. Here are a few common types:

- Standalone applications
- Client/Server (2-tier) applications
- N-tier applications
- Web-centric n-tier applications
- Enterprise n-tier applications



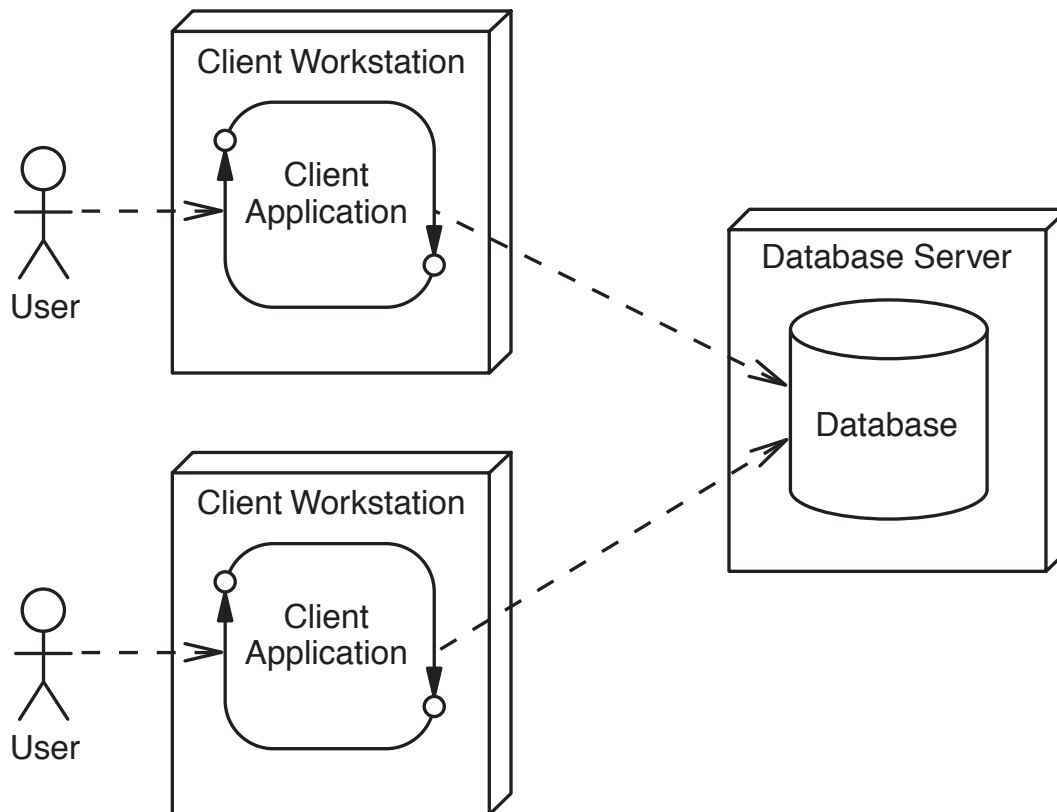
Standalone Applications



- No external data sources (all application data exists on a file server)
- No network communication (all application components exist on one machine)



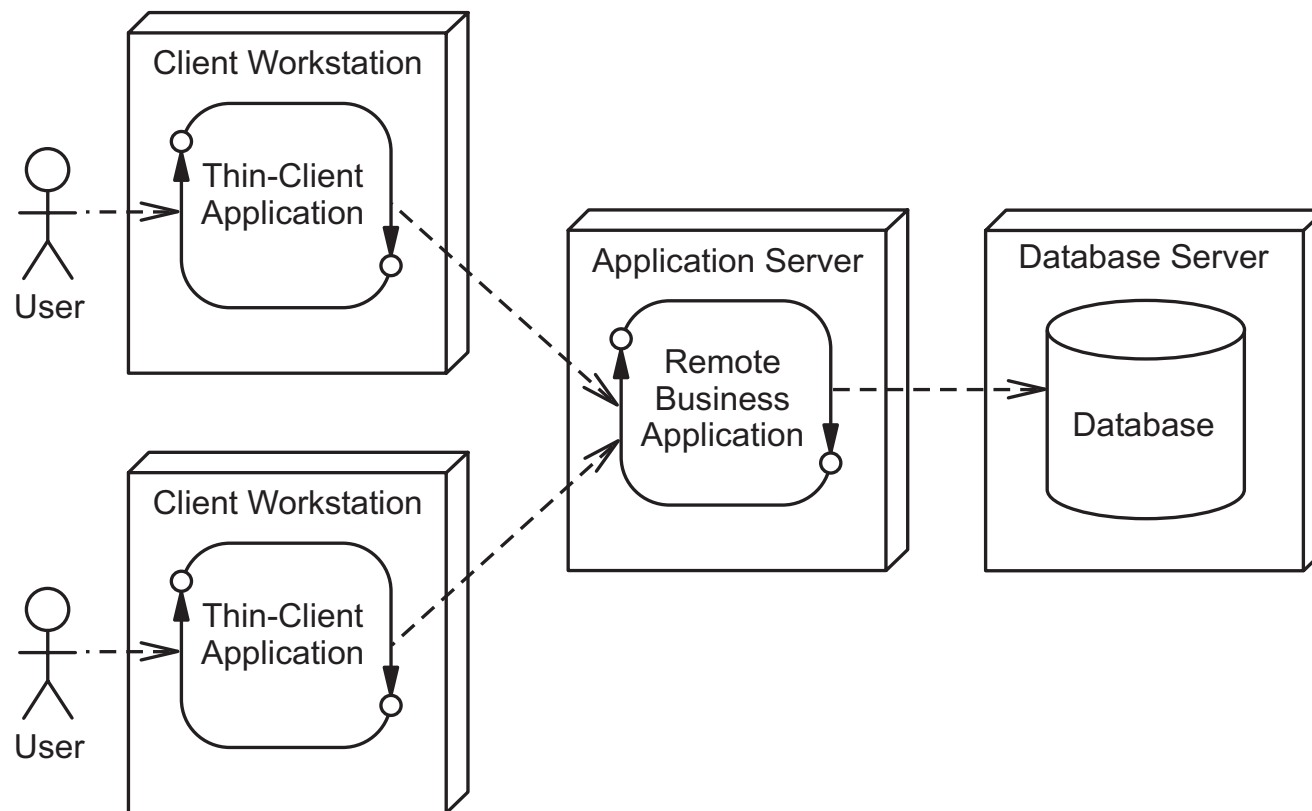
Client/Server (2-Tier) Applications



- Thick client (with business logic in the client tier)
- Data store manages data integrity



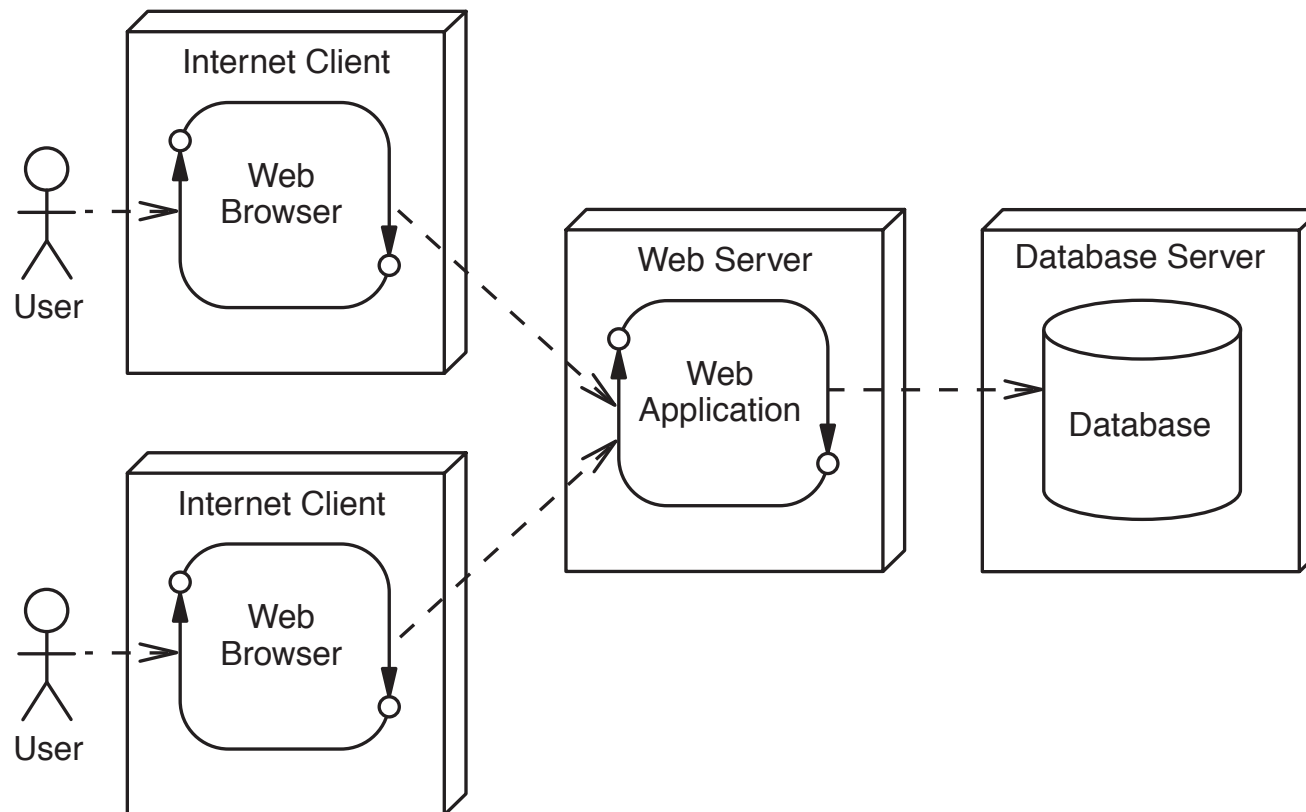
N-Tier Applications



- Thin client (business logic is in the application server)
- Application server manages data integrity



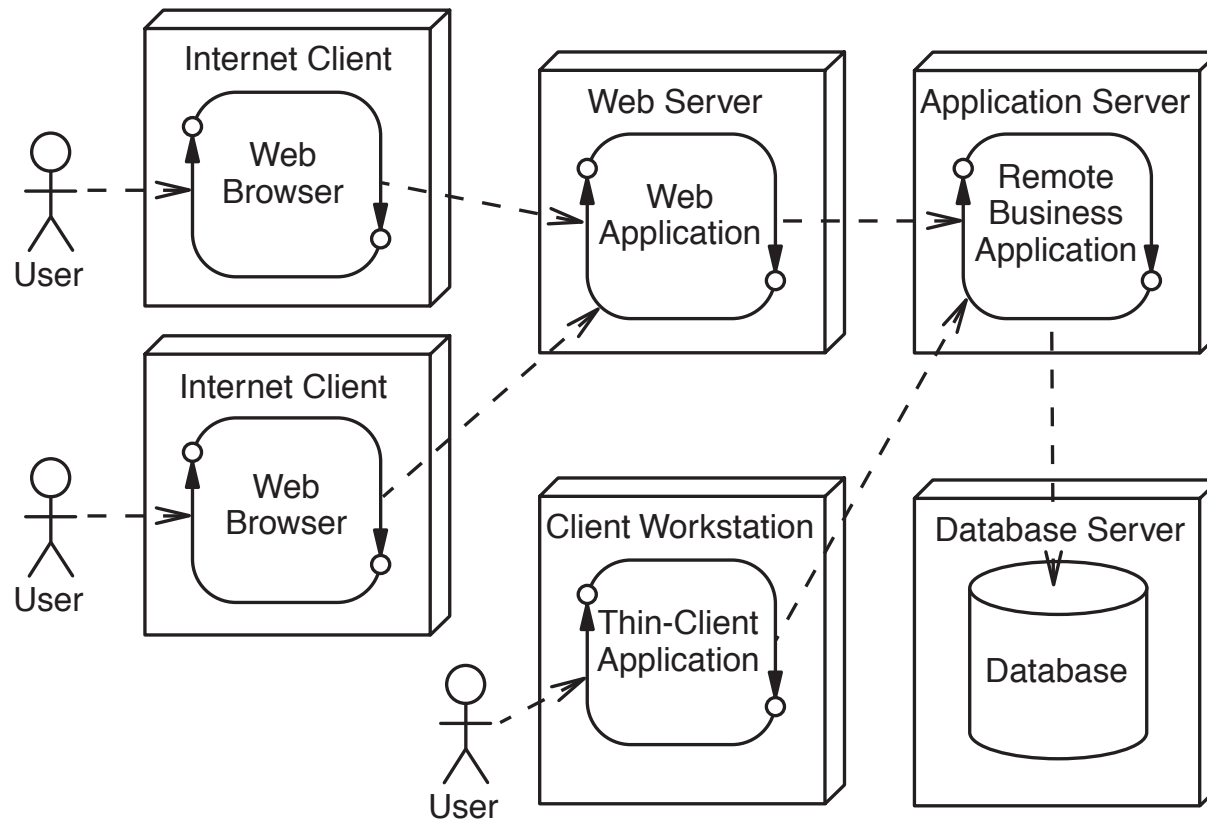
Web-Centric (N-Tier) Applications



- Web browser becomes the thin client
- Web server provides presentation and business logic



Enterprise (N-Tier) Architecture Type



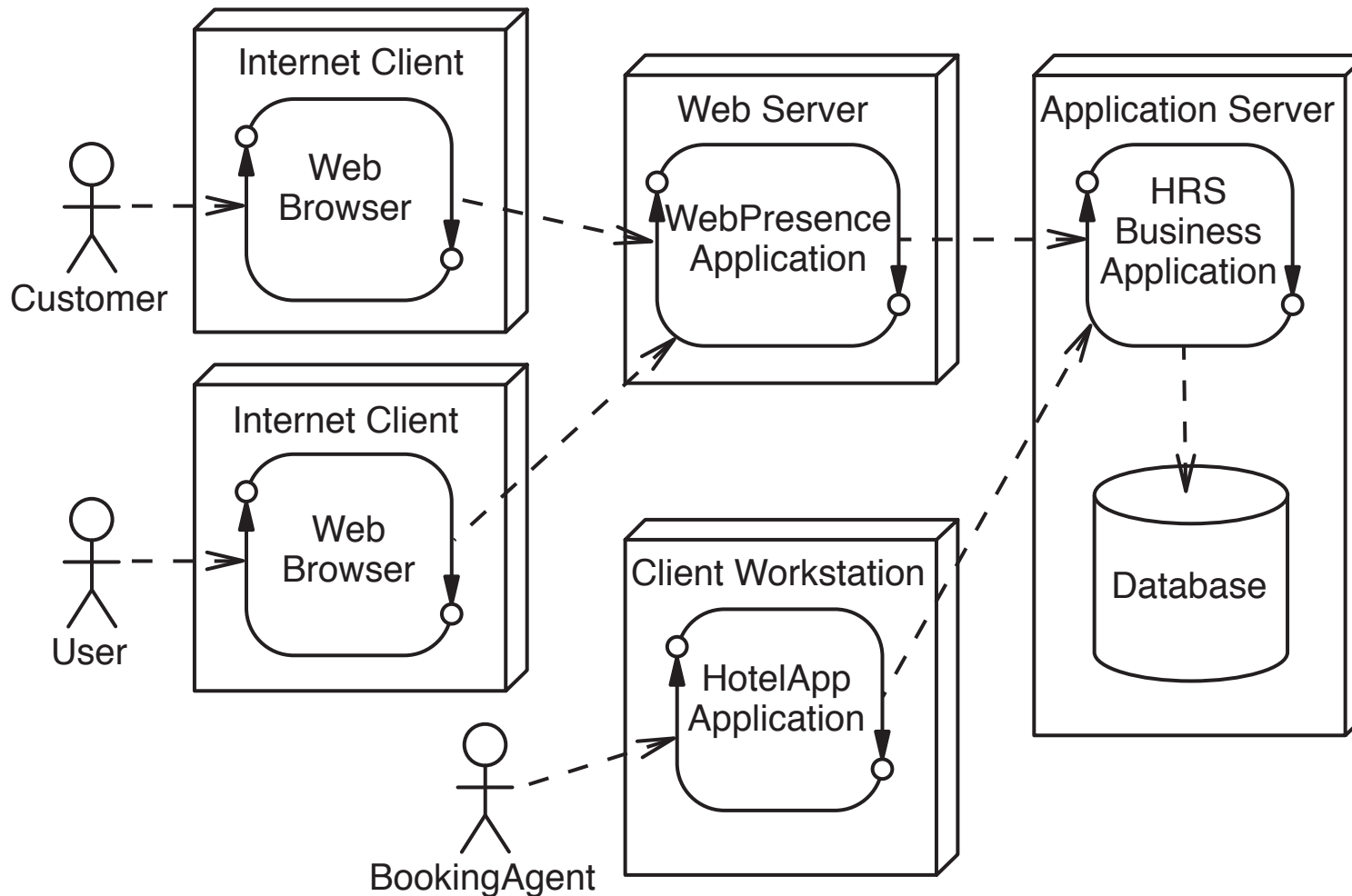


Enterprise (N-Tier) Architecture Type

- Two thin clients:
 - Web browser for Internet users
 - GUI thin client for intranet users
- Web application server provides presentation logic
- Application server provides business logic



Hotel Reservation System Architecture



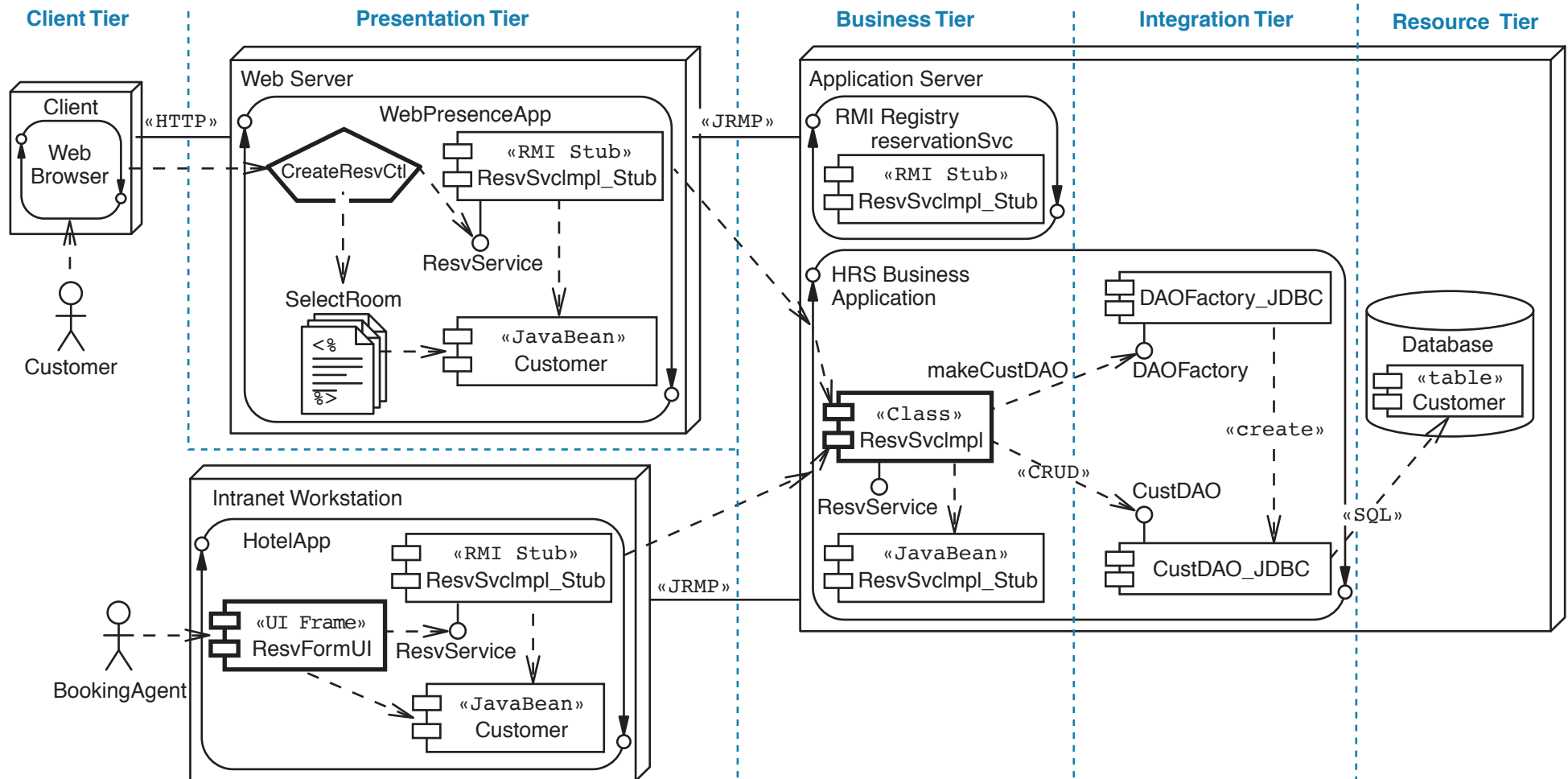


Creating The Detailed Deployment Diagram

1. Design the components for the architecturally significant use cases.
2. Place design components into the Architecture model.
3. Draw the detailed Deployment diagram from the merger of the design and infrastructure components.



Example Detailed Deployment Diagram



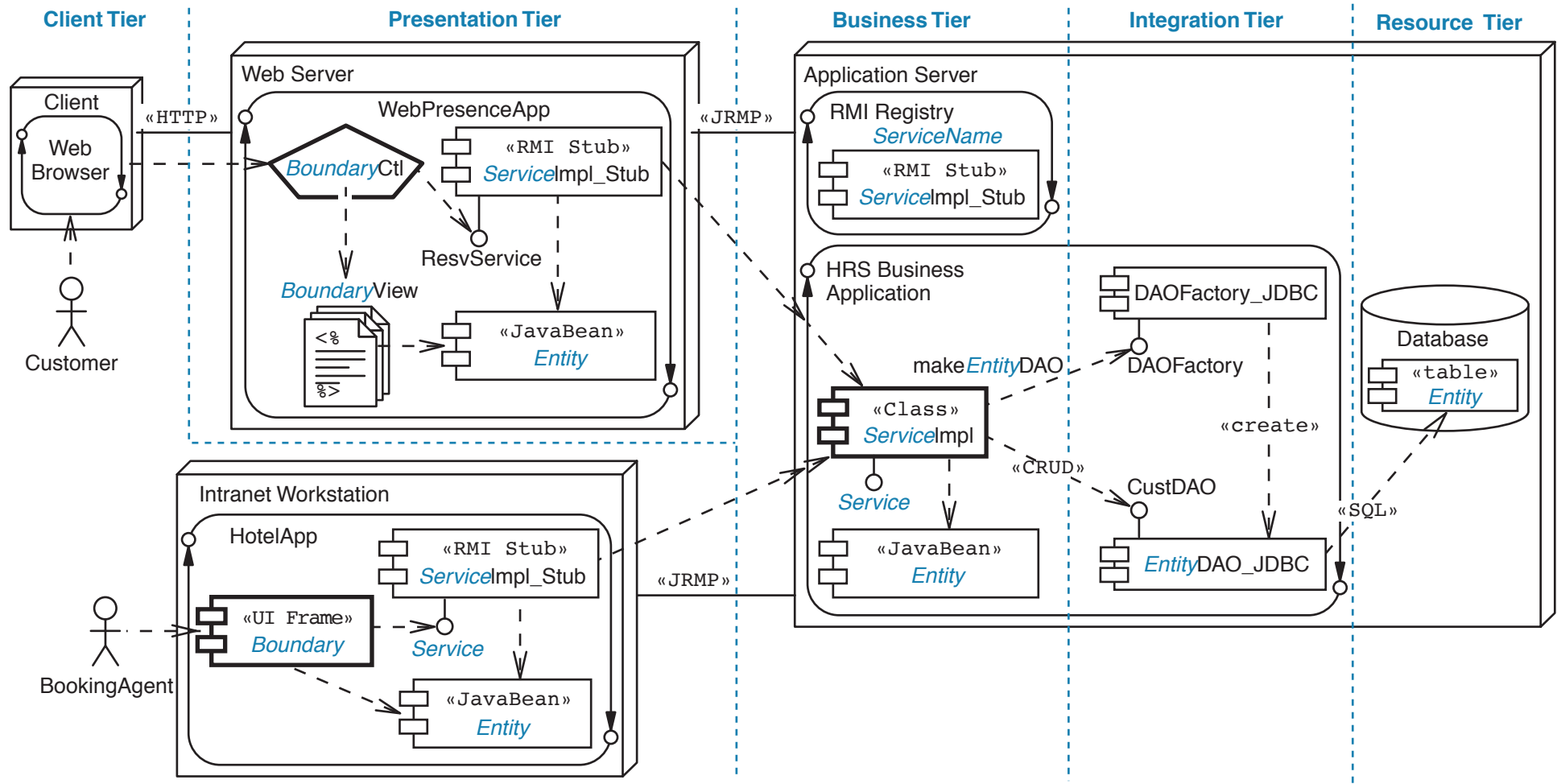


Creating the Architecture Template

1. Strip the detailed Deployment diagram to just one set of Design components: boundary, service, and entity.
2. Replace the name of the Design component with the type (for example, ResvSvcImpl_Stub becomes *Service*Impl_Stub).



Example Architecture Template





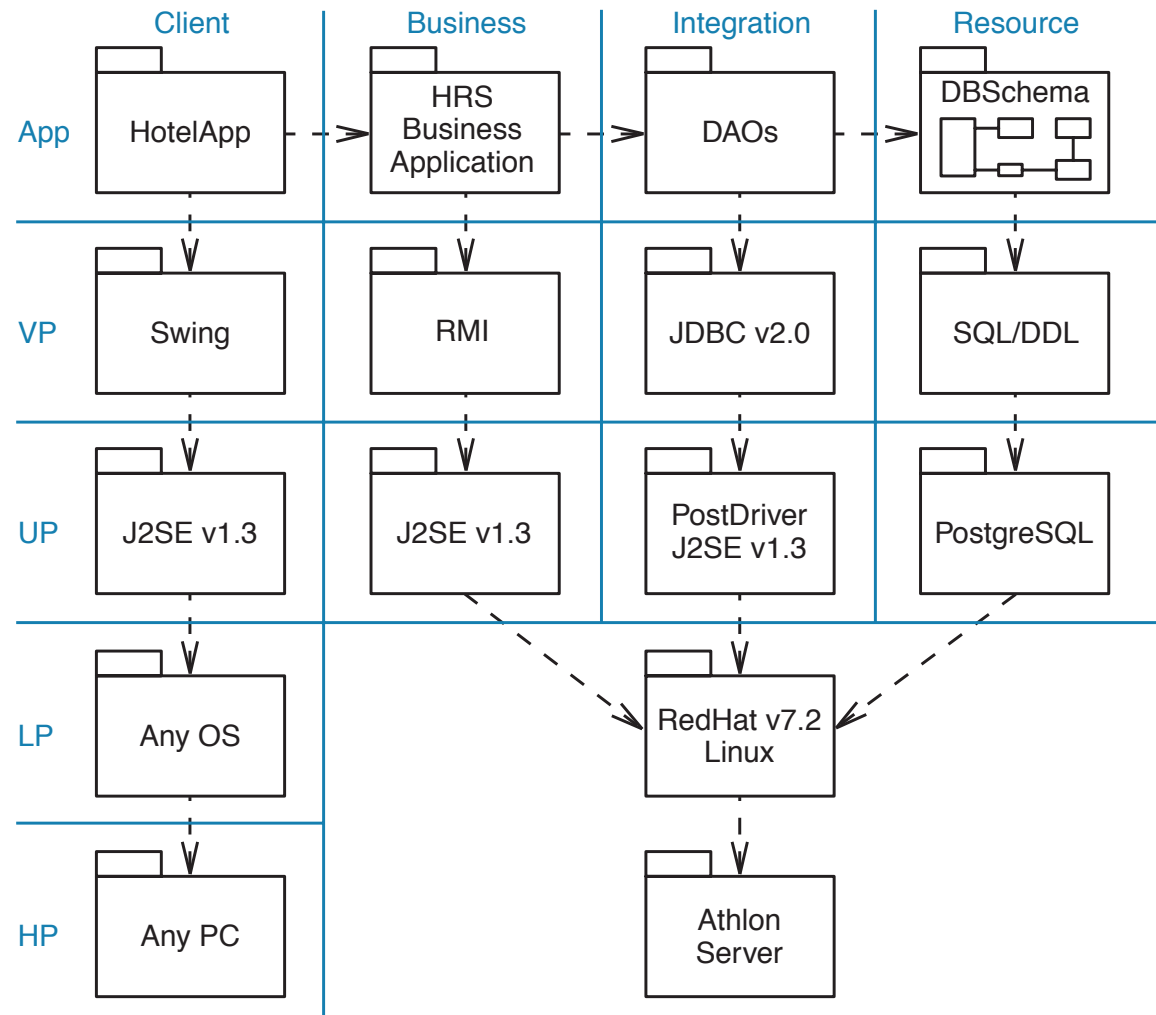
Creating the Tiers and Layers Package Diagram

For each tier:

1. Determine what application components exist.
2. Determine what technology APIs, communication protocols, or specifications are required that the components require.
3. Determine which container products to use.
4. Determine which operating system to use.
5. Determine what hardware to use.

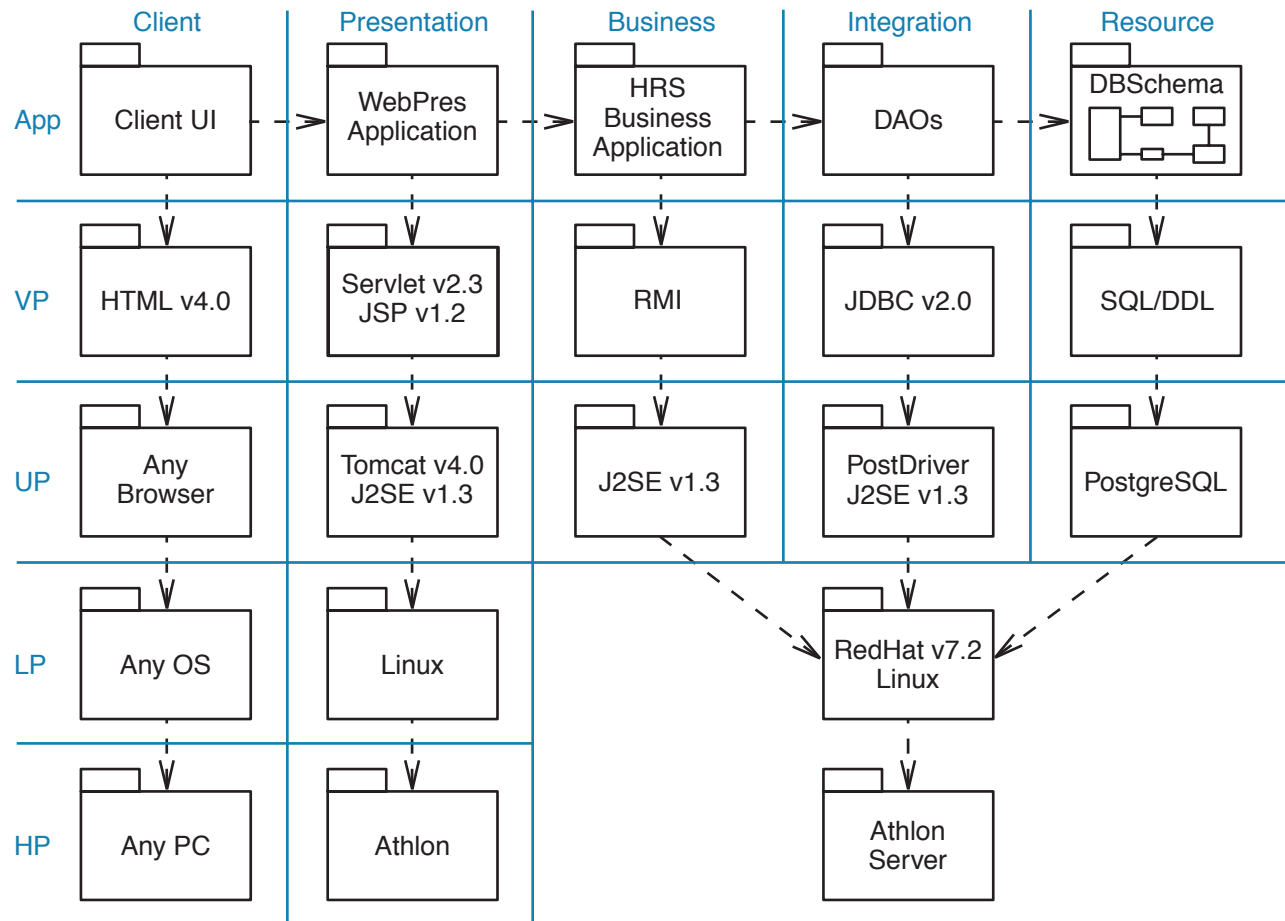


Tiers and Layers Diagram for the HotelApp





Tiers and Layers Diagram for the Hotel System's Web Presence





Summary

- Difference between architecture and design:
 - Design produces components to implement a use case.
 - Architecture provides a template into which the designed components are realized.
- You can model the architecture using:
 - Deployment diagrams
 - Component diagrams
 - Packages
 - Tiers and layers



Module 12

Introducing the Architectural Tiers



Objectives

Upon completion of this module, you should be able to:

- Describe the concepts of the Client and Presentation tiers
- Describe the concepts of the Business tier
- Describe the concepts of the Resource and Integration tiers
- Describe the concepts of the Solution model



Introducing the Client and Presentation Tiers

The Client and Presentation tiers primarily contain:

- Controller components:
 - Control the input from the boundary
 - Perform input sanity checking
 - Call business logic methods
 - Dispatch view components
- View components:
 - Retrieve data required by the view
 - Prepare the view in a format suitable for the recipient
 - Add client-side sanity checking (optional)



Boundary Interface Technologies

The primary Boundary Interface types are:

- Graphical user interface (GUI)
- Web user interface (Web UI)
- Machine or device interface (for example, Web services)

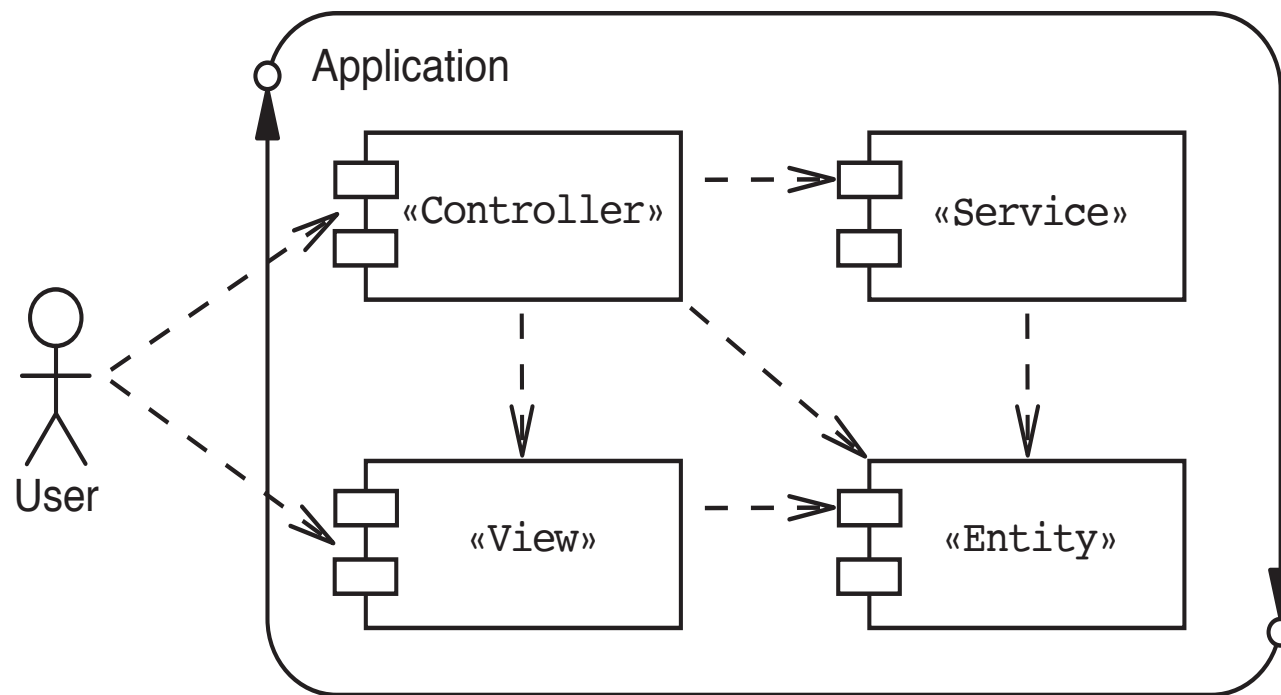
Other types of Boundary Interfaces include:

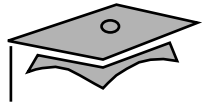
- Touchpads
- Direct manipulation
- Joystick
- Interactive voice recognition
- Keypads
- Command line



Generic Application Components

There are four fundamental types of application components:





GUI Screen Design

An example GUI screen design:

Customer Management Screen

First name:

Last name:

Phone:

Address

Street1:

Street2:

City:

State: ▼

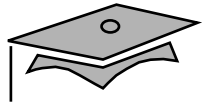
Zip:

Search

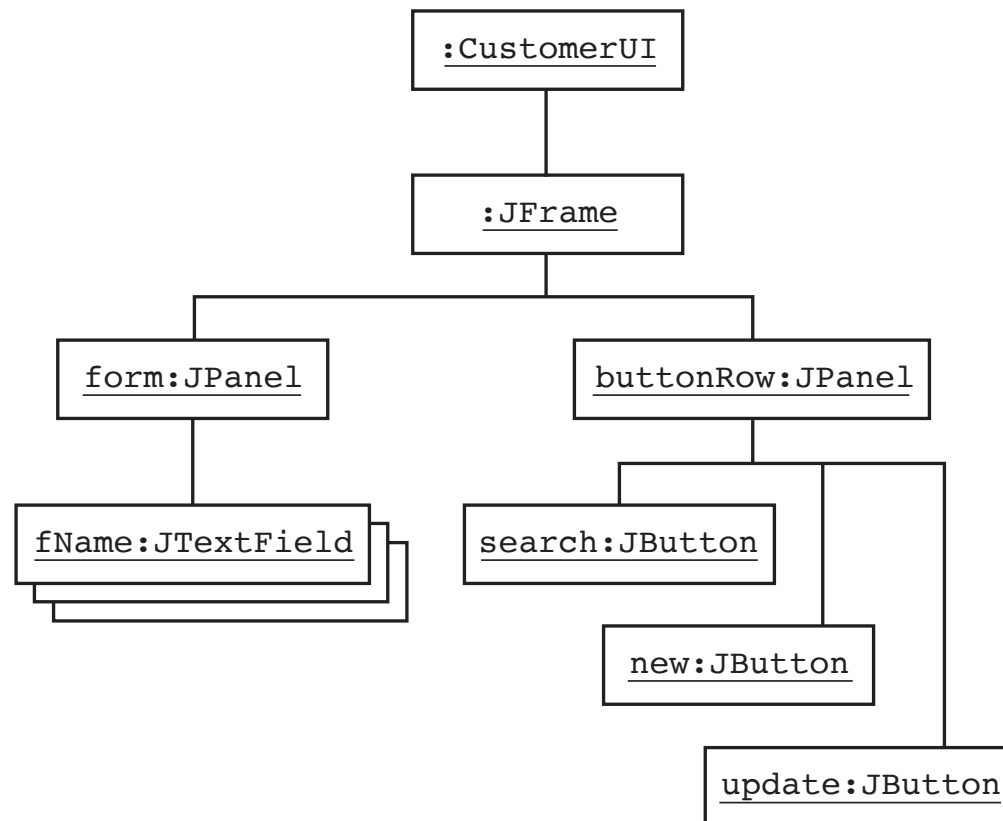
New

Update

Done



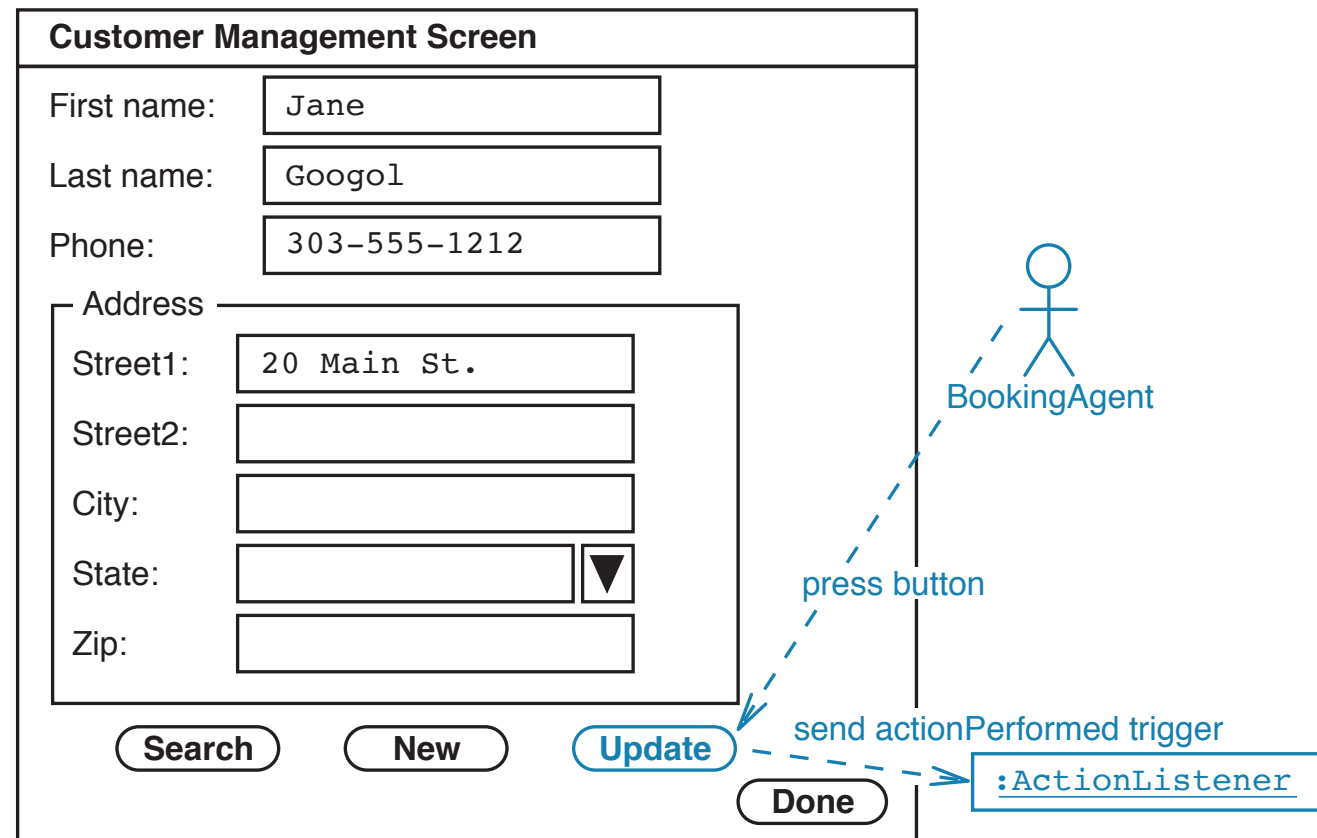
Customer GUI Component Hierarchy





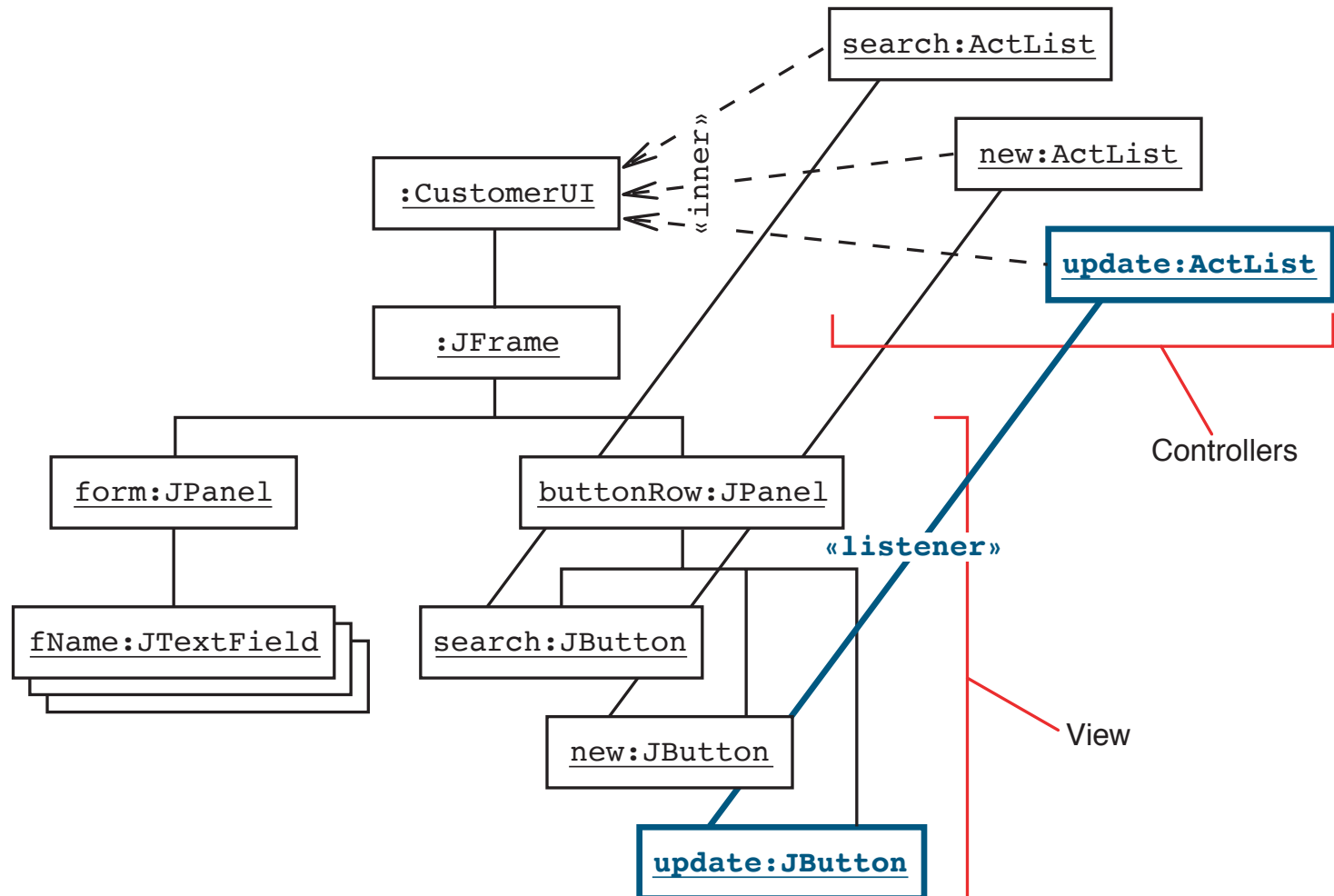
GUI Event Model

Java™ technology uses an event-listener mechanism:





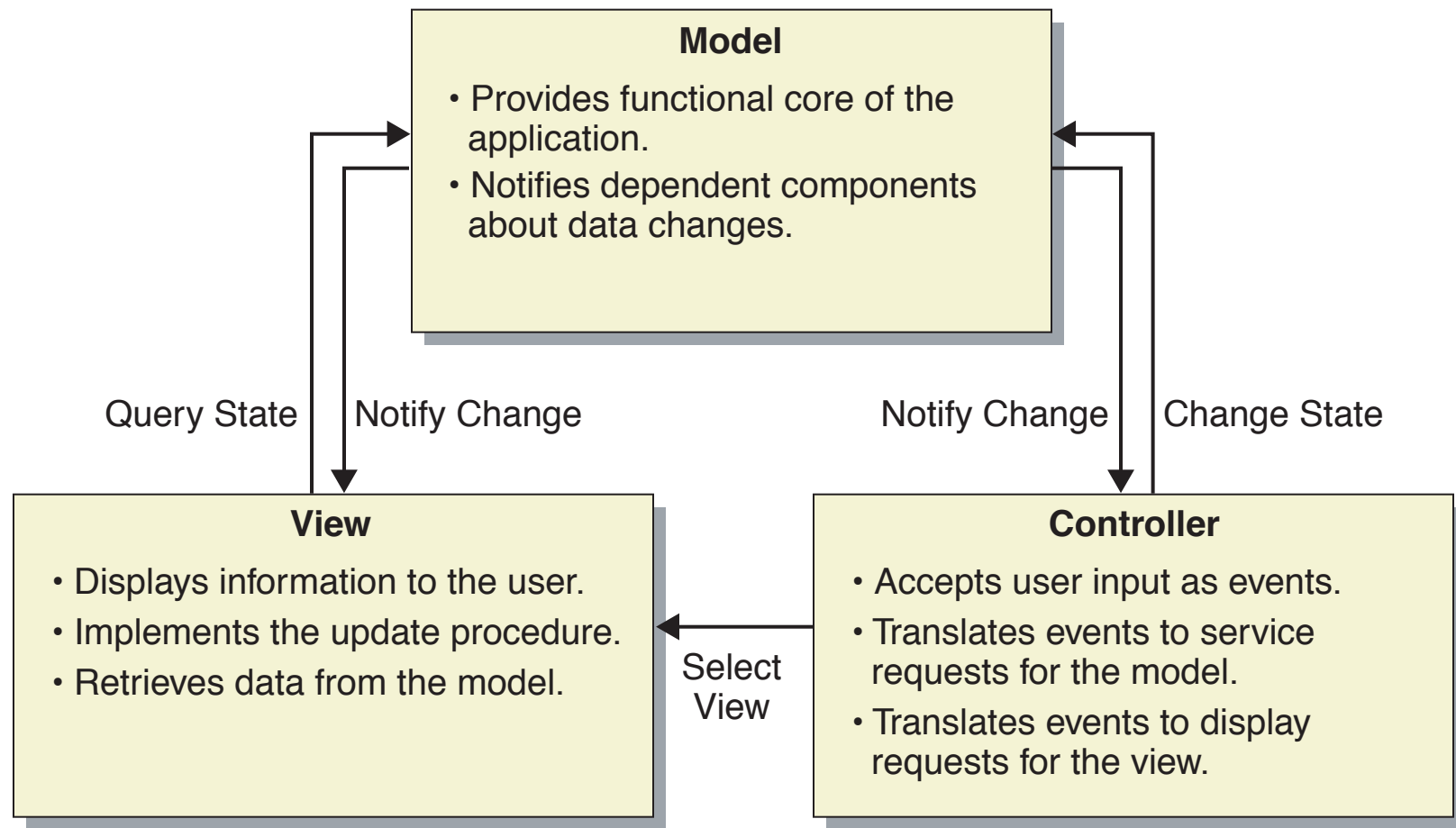
GUI Listeners as Controller Elements





The MVC Pattern

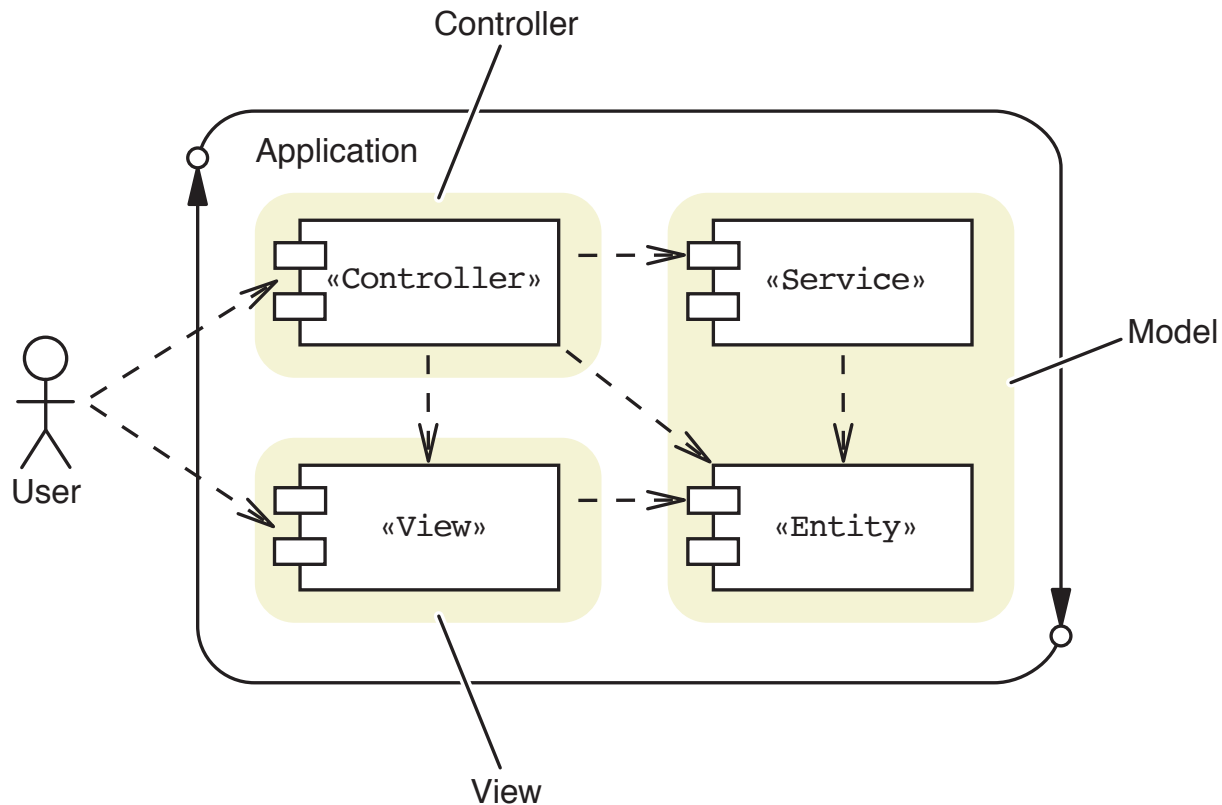
MVC separates Views and Controllers from the Model.





The MVC Component Types

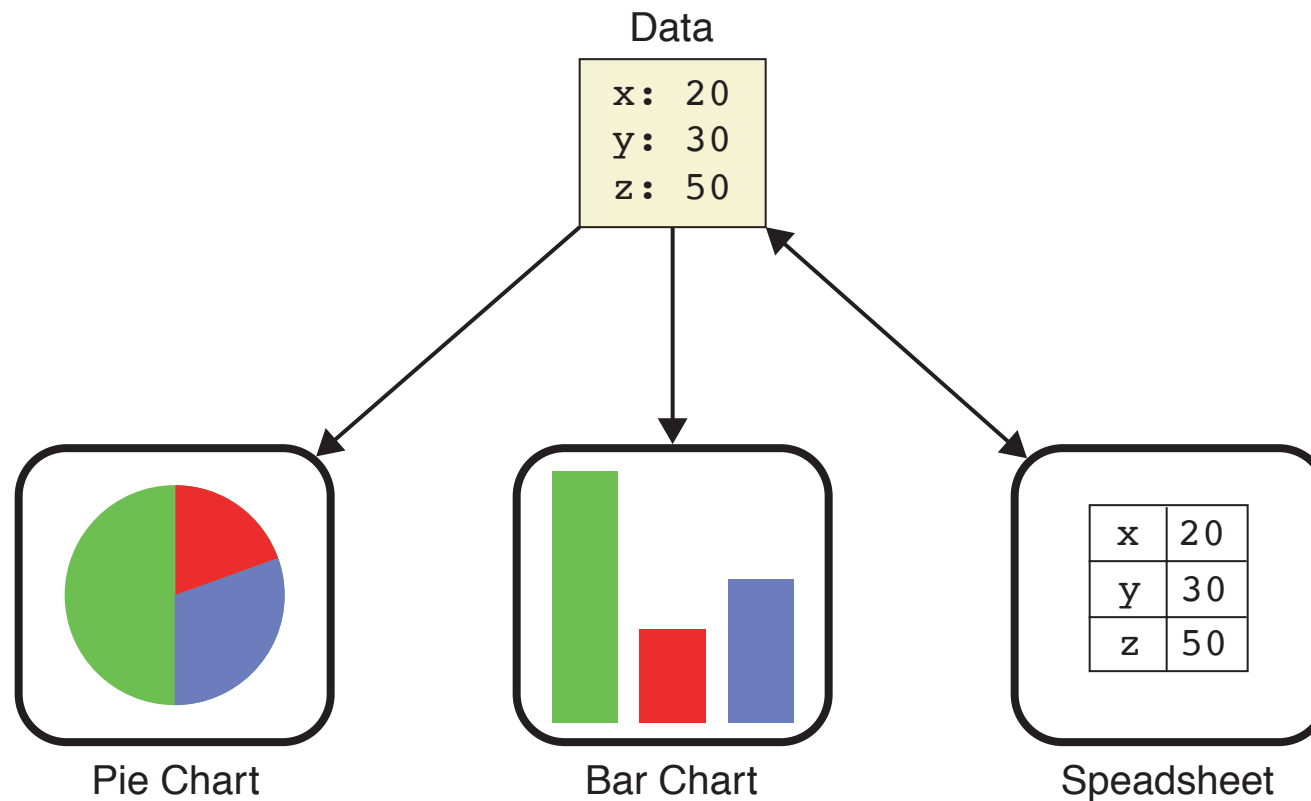
MVC groups Service and Entity components into a single component called Model:





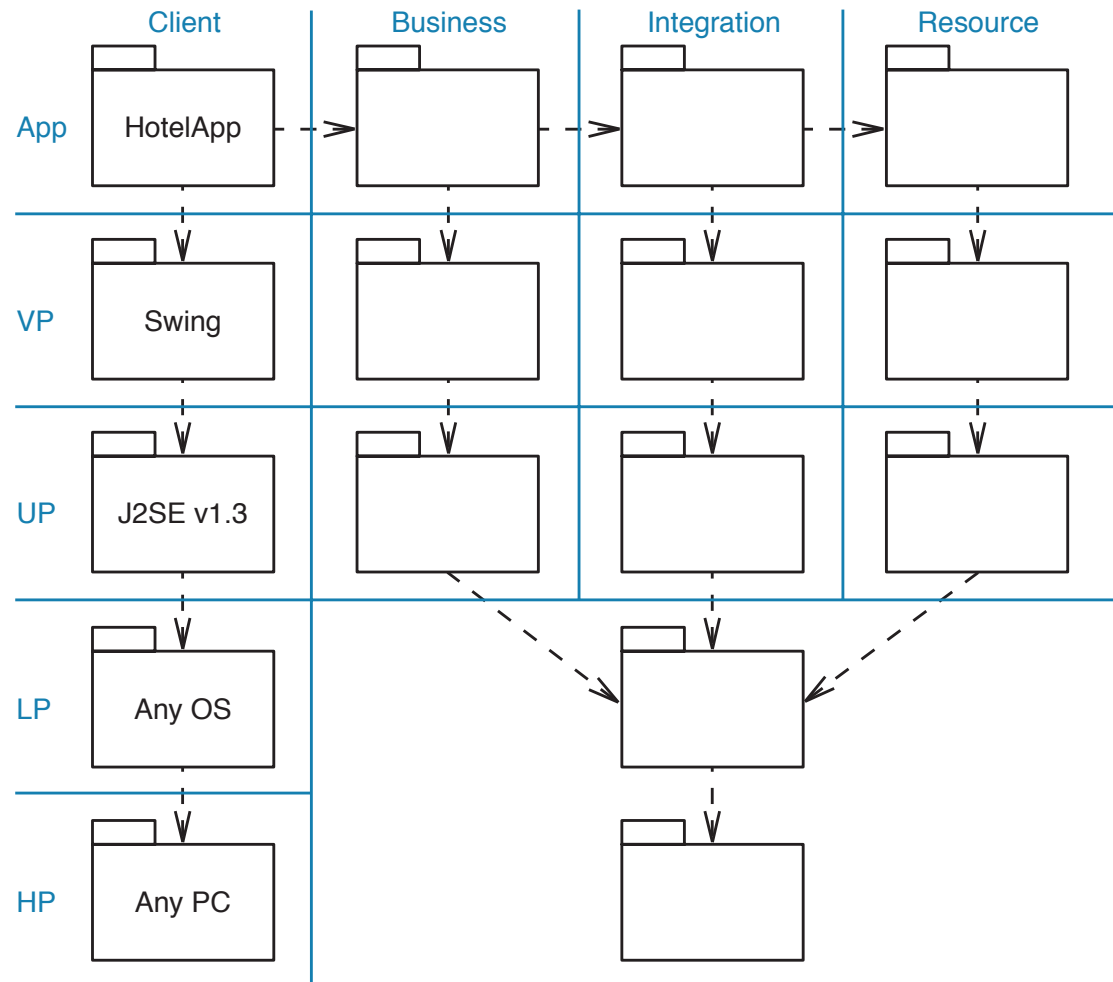
Example Use of the MVC Pattern

Multiple views can be used on the same data:





Overview of the Tiers and Layers Package Diagram





Exploring Web User Interfaces

A Web UI provides a browser-based user interface. Web UIs have the following characteristics:

- Perform a few large user actions (HTTP requests).
- A single use case is usually broken into multiple screens.
- There is often a single path through the screens.
- Only one screen is usually open at a time.



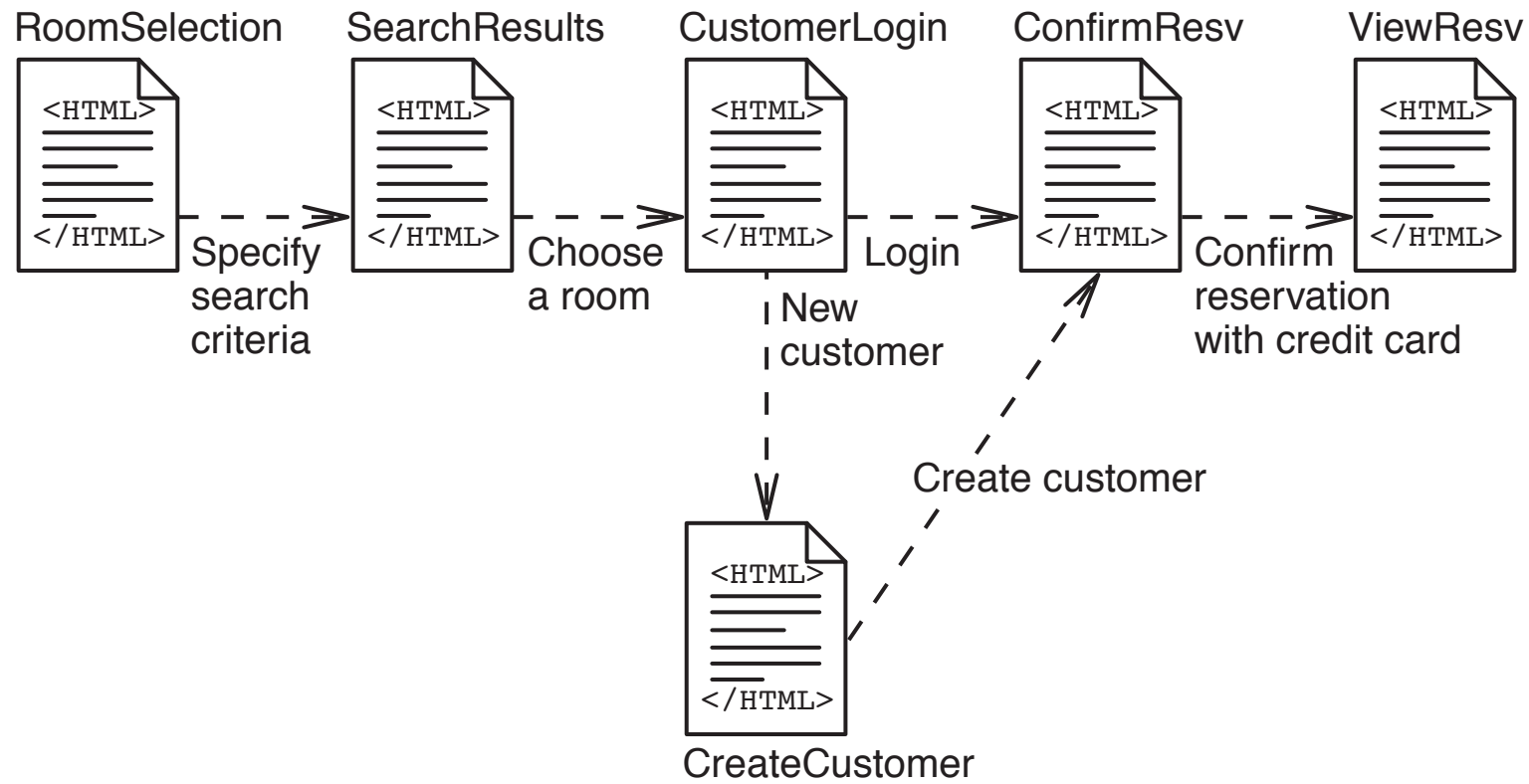
Web UI Screen Design

- A Web UI tends to be constructed as a sequence of related screens.
- Each screen is a hierarchy of UI components.
- A Web UI screen presents the user's view of the domain model as well as presents the user's action controls.
- It is rare that a screen can be reused by multiple use cases.



Example Web Page Flow

The Create a Reservation Online (E5) use case can be viewed as a sequence of Web UI screens:





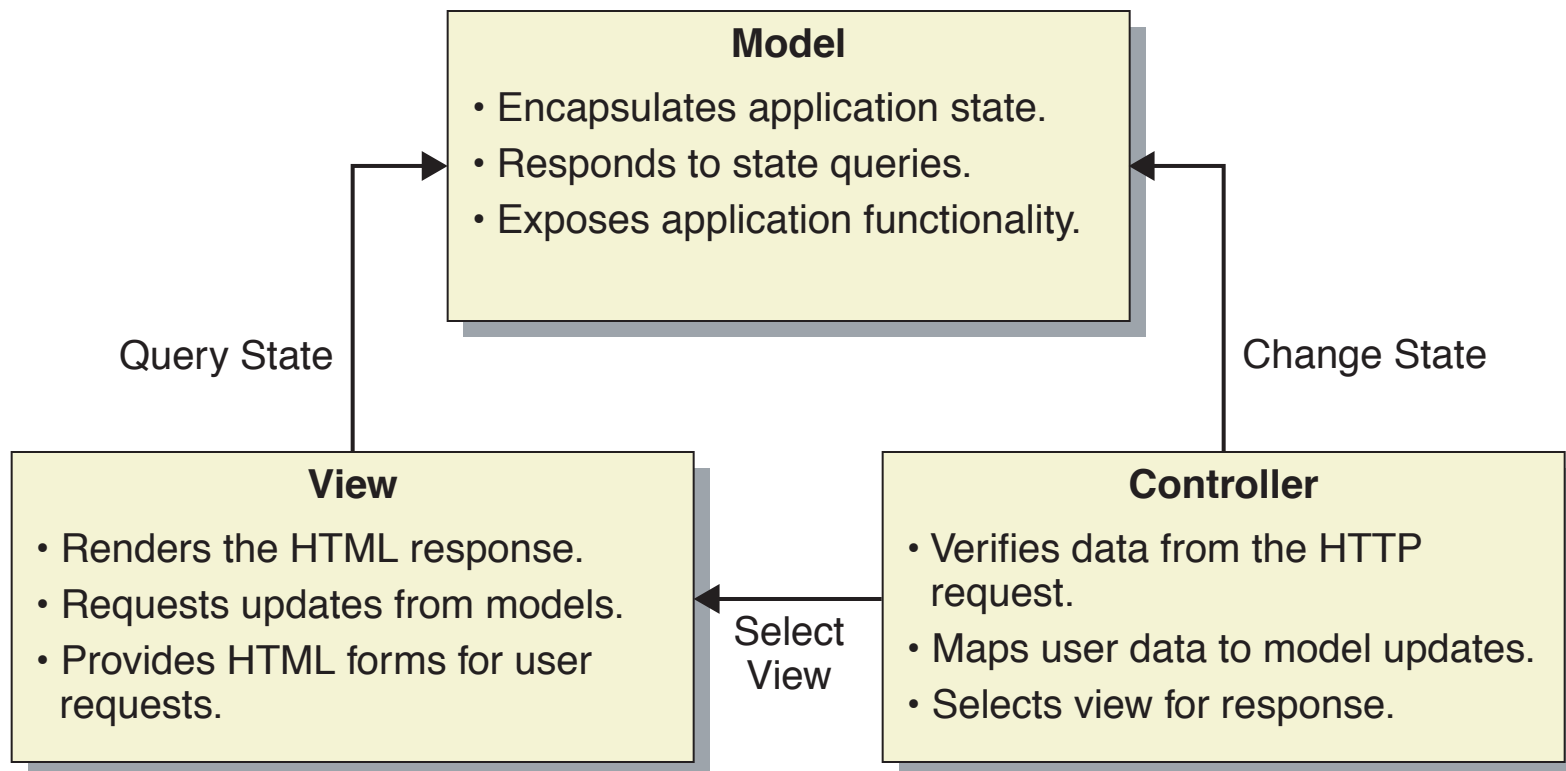
Web UI Event Model

- Micro events can be handled by JavaScript™ technology code.
- Macro events are handled as HTTP requests from the web browser to the Web server.



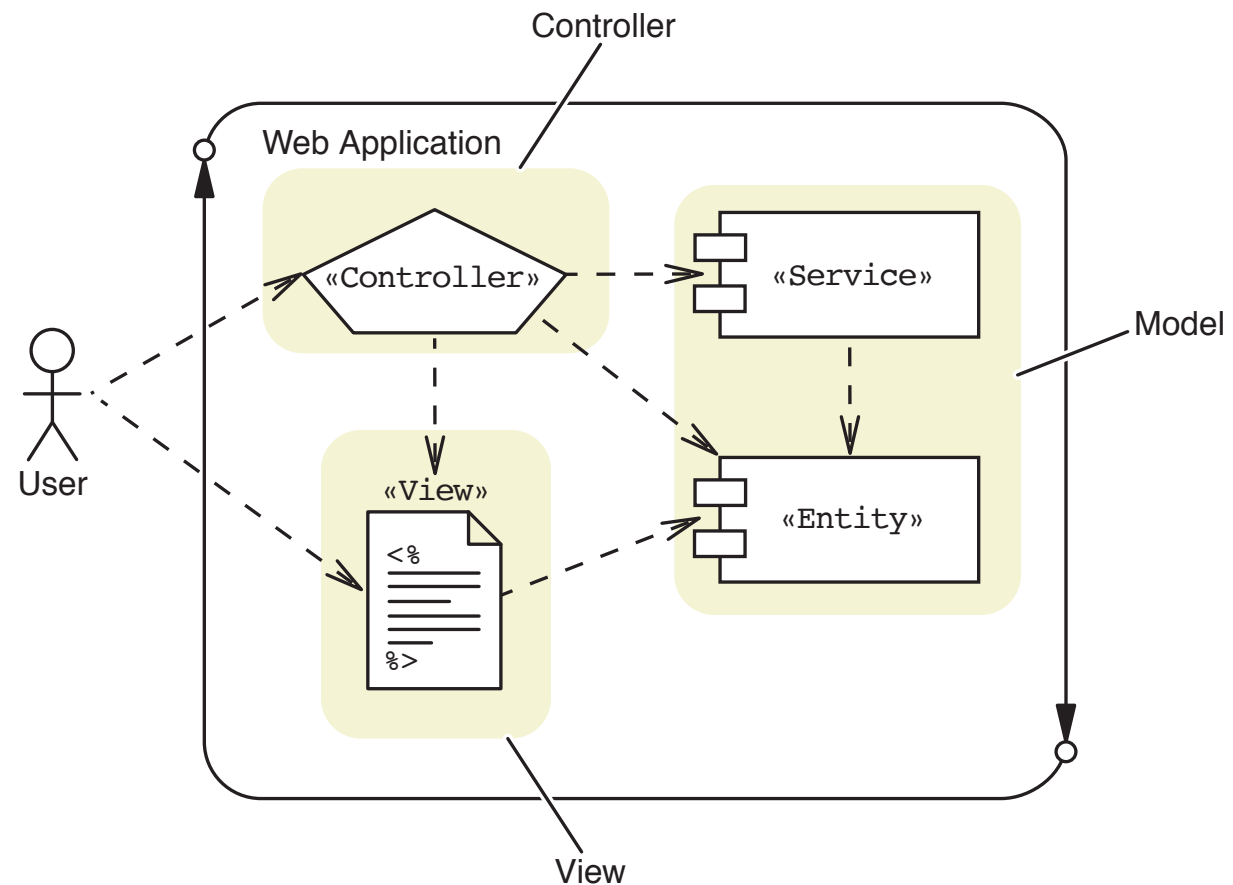
The WebMVC Pattern

WebMVC is based on MVC, but with no Model to View updates:





The WebMVC Pattern Component Types





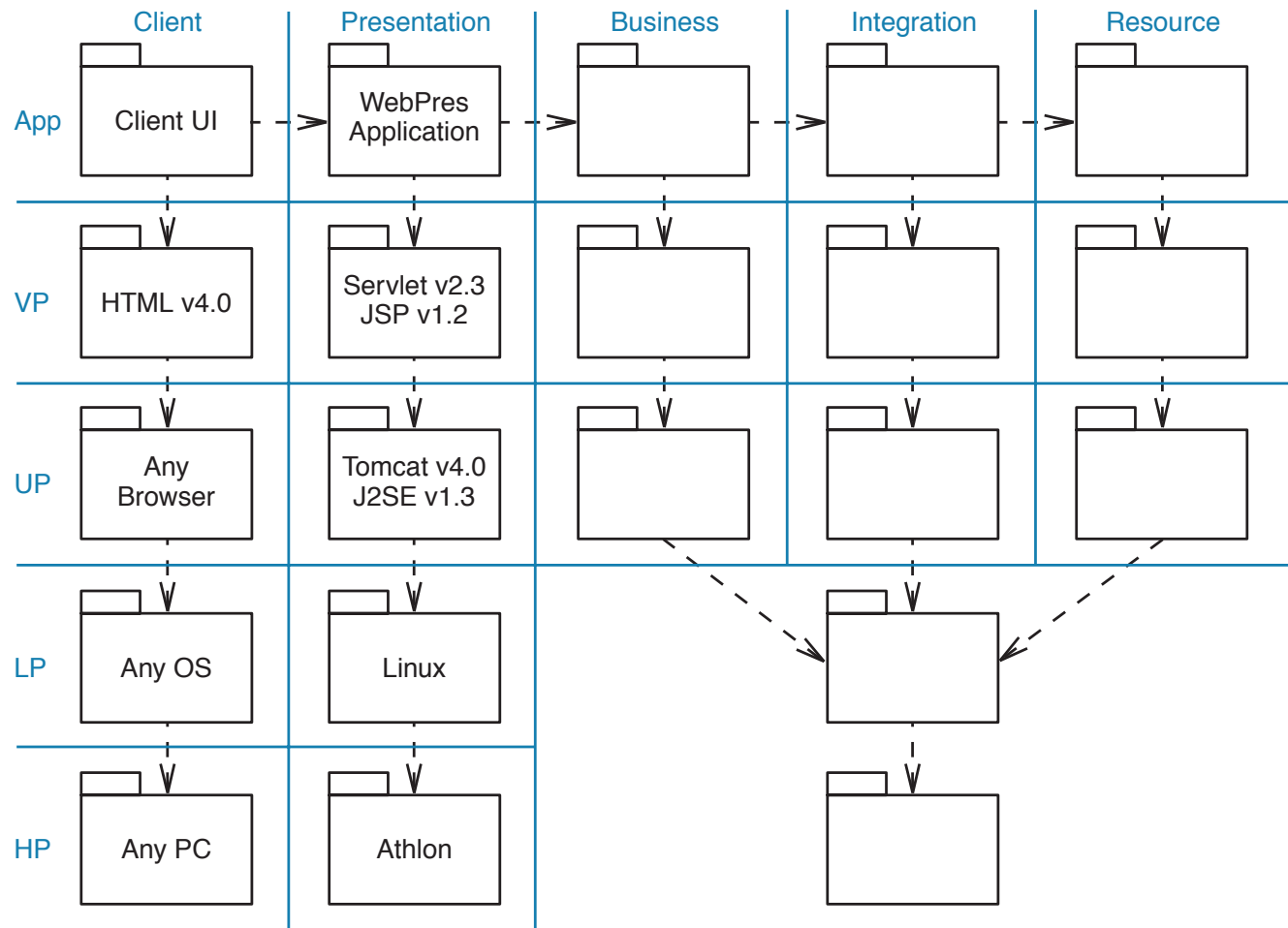
The WebMVC Pattern

The Model 2 architecture use these components:

- Java servlets act as a Controller to process HTTP requests:
 - Verify the HTML form data
 - Update the business Model
 - Select and dispatch to the next View
- JavaServer Pages™ technology acts as the Views that are sent to the user.
- Java technology classes (whether local or distributed) act as the Model for the business services and entities.



Overview of the Tiers and Layers Package Diagram





Introducing the Business Tier

The Business tier primarily contains:

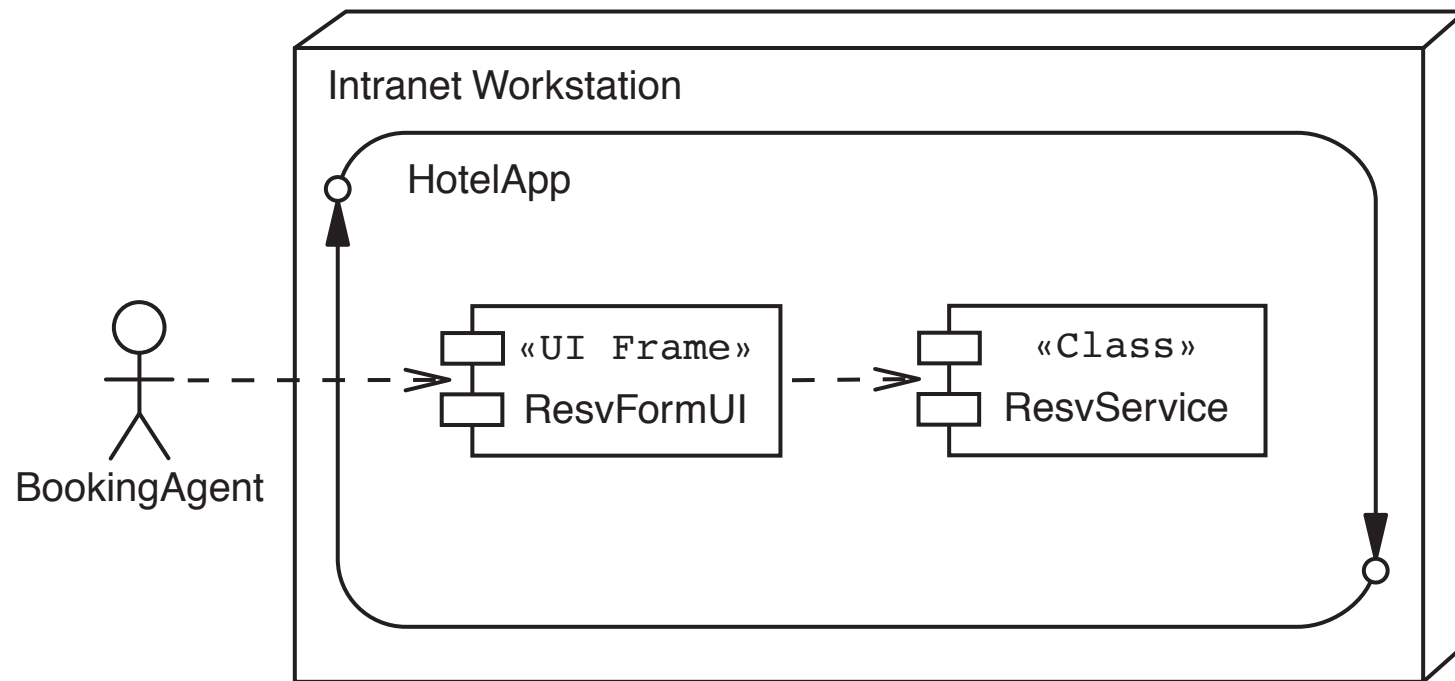
- Entity components
- Service components
 - Perform validation of business rules
 - Perform updates on the entity components

The Business tier may be accessed by:

- Local components
- Remote components, for example:
 - Remote Method Invocation (RMI)
 - Web service protocols

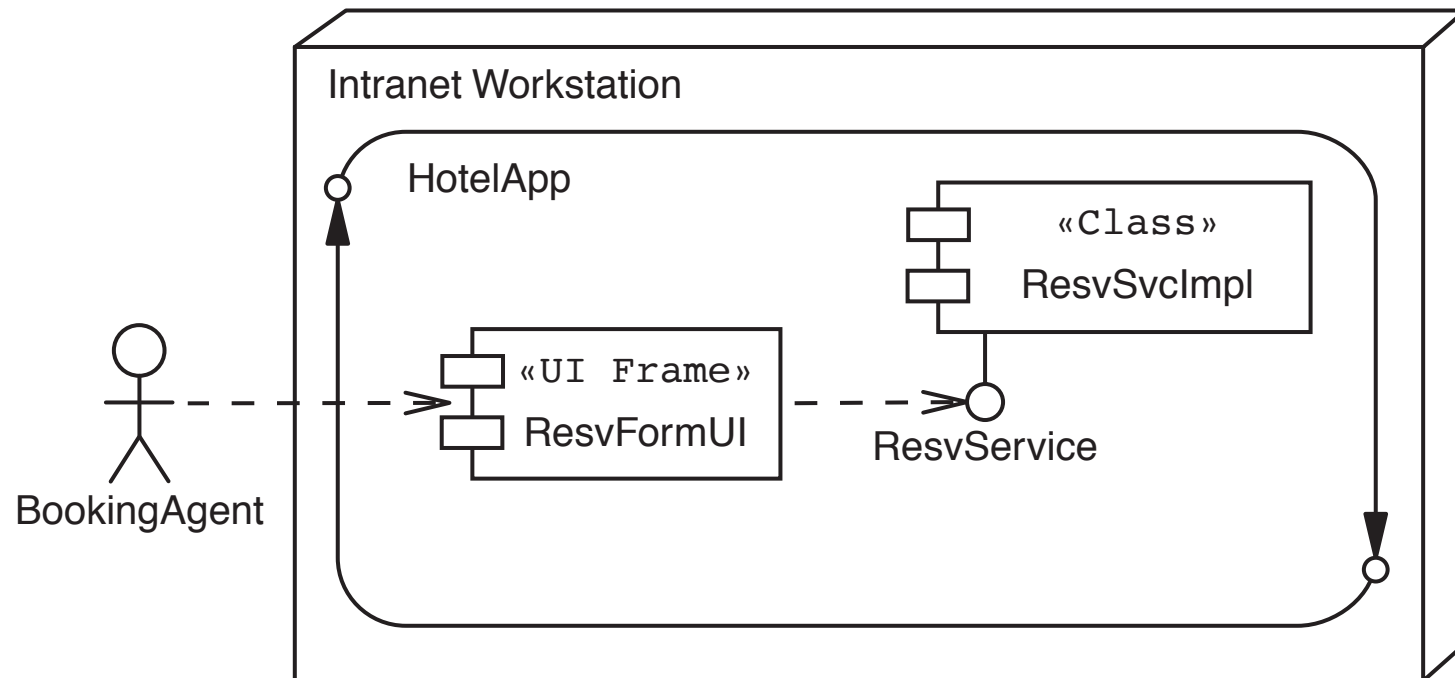


Local Access to a Service Component



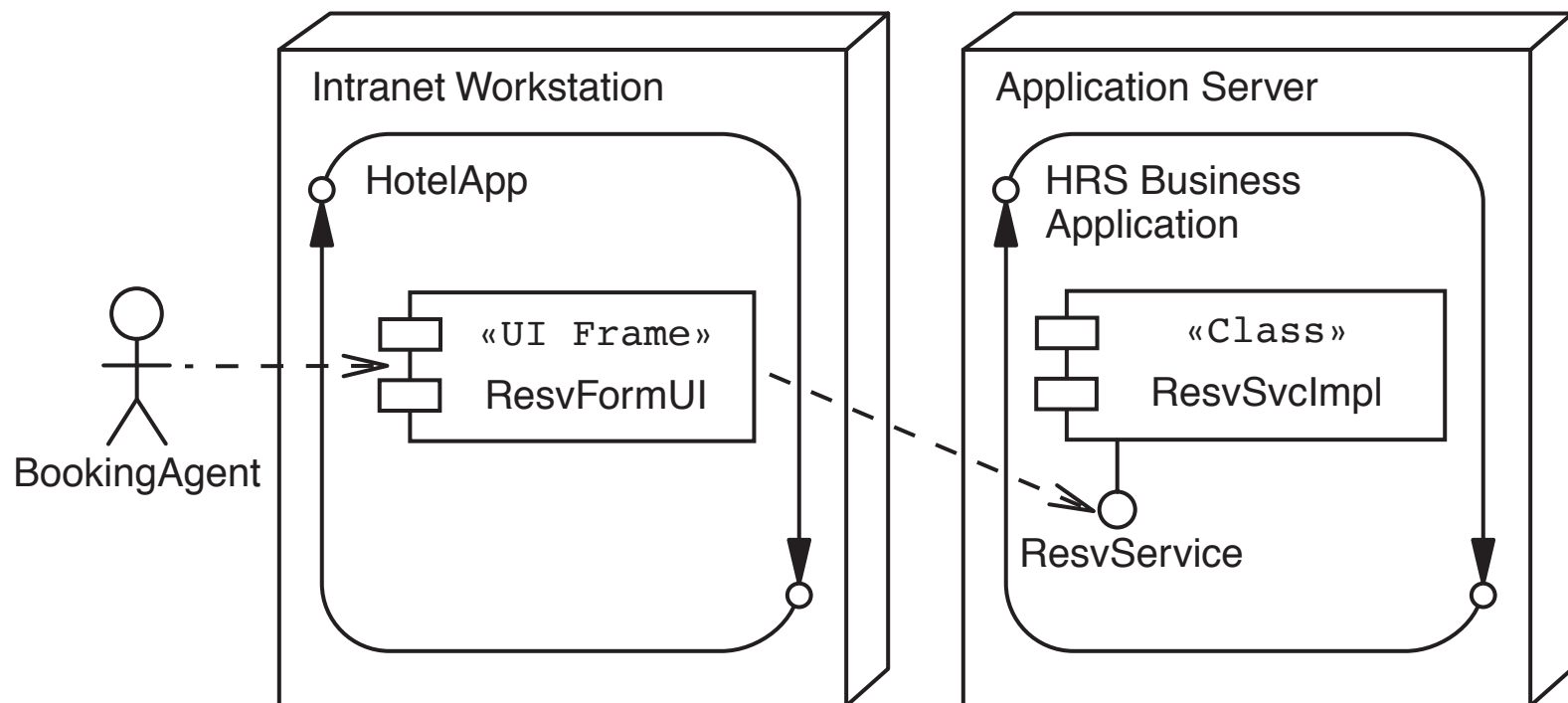


Applying Dependency Inversion Principle



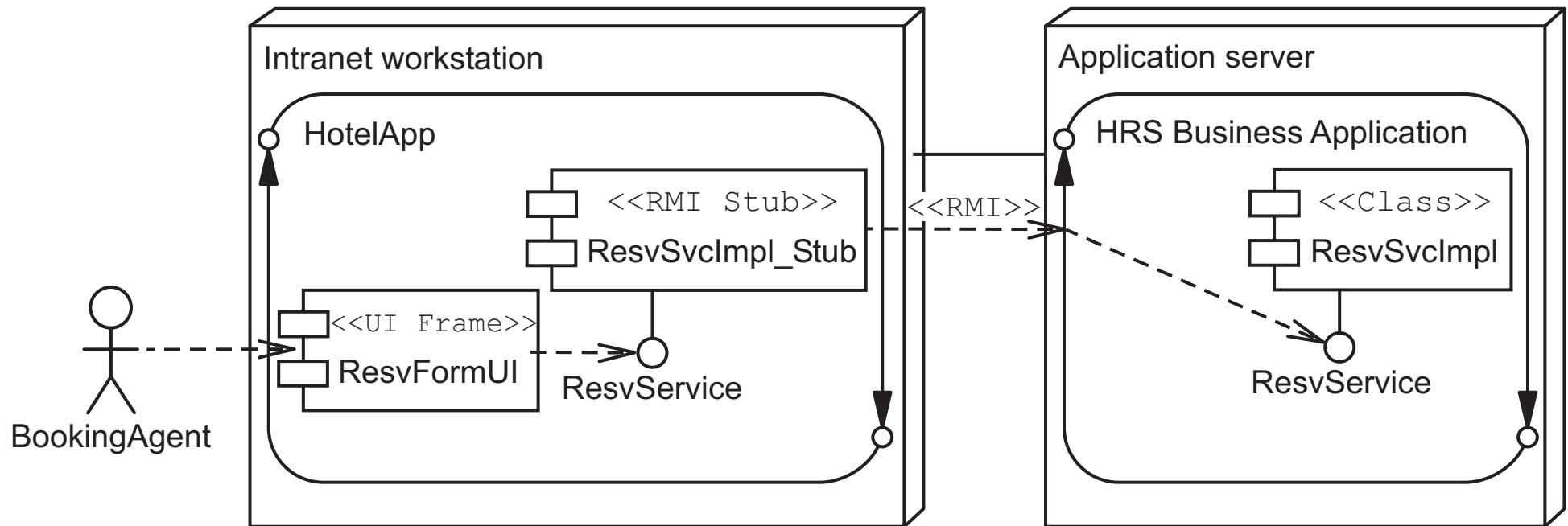


An Abstract Version of Accessing a Remote Service



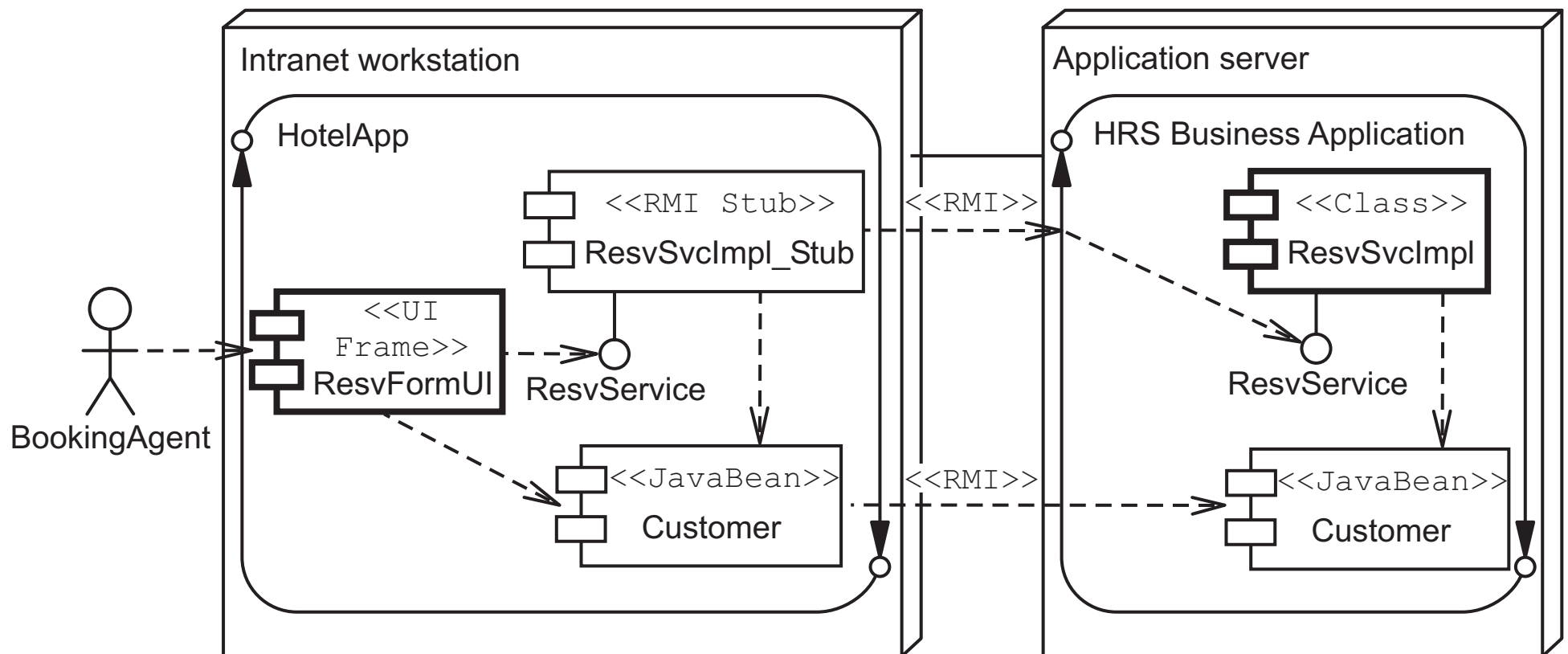


Accessing a Remote Service Using RMI



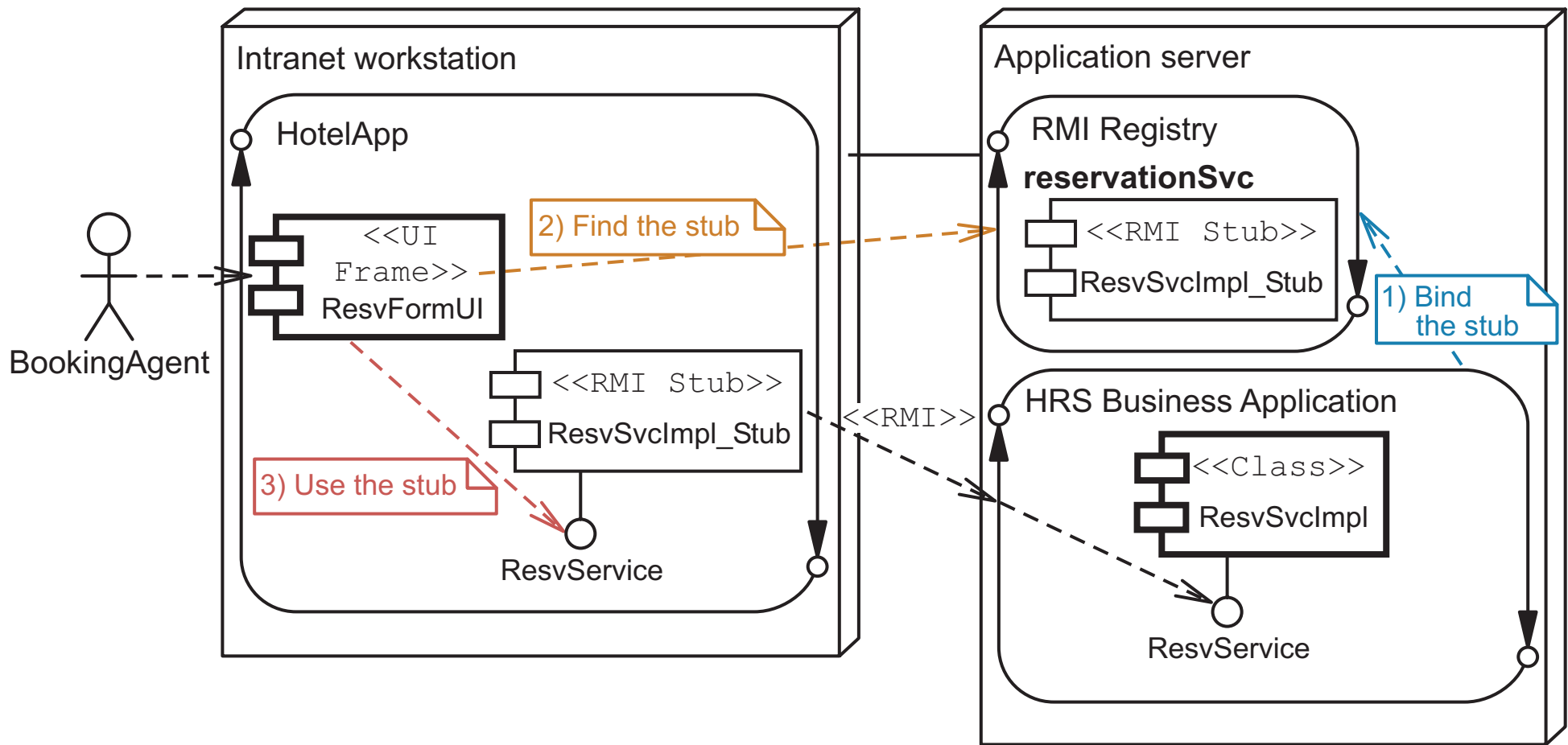


RMI Uses Serialization to Pass Parameters



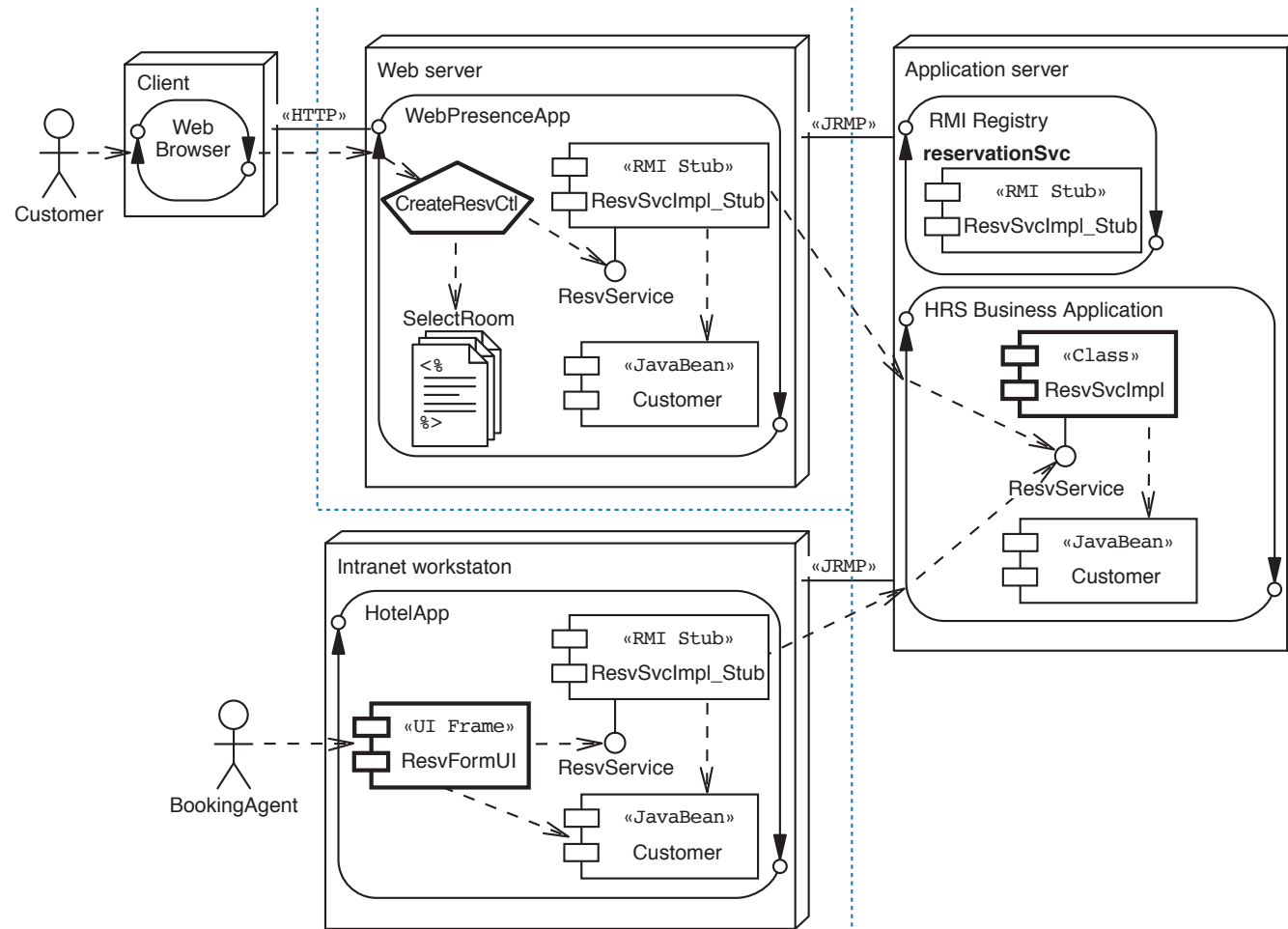


The RMI Registry Stores Stubs for Remote Lookup



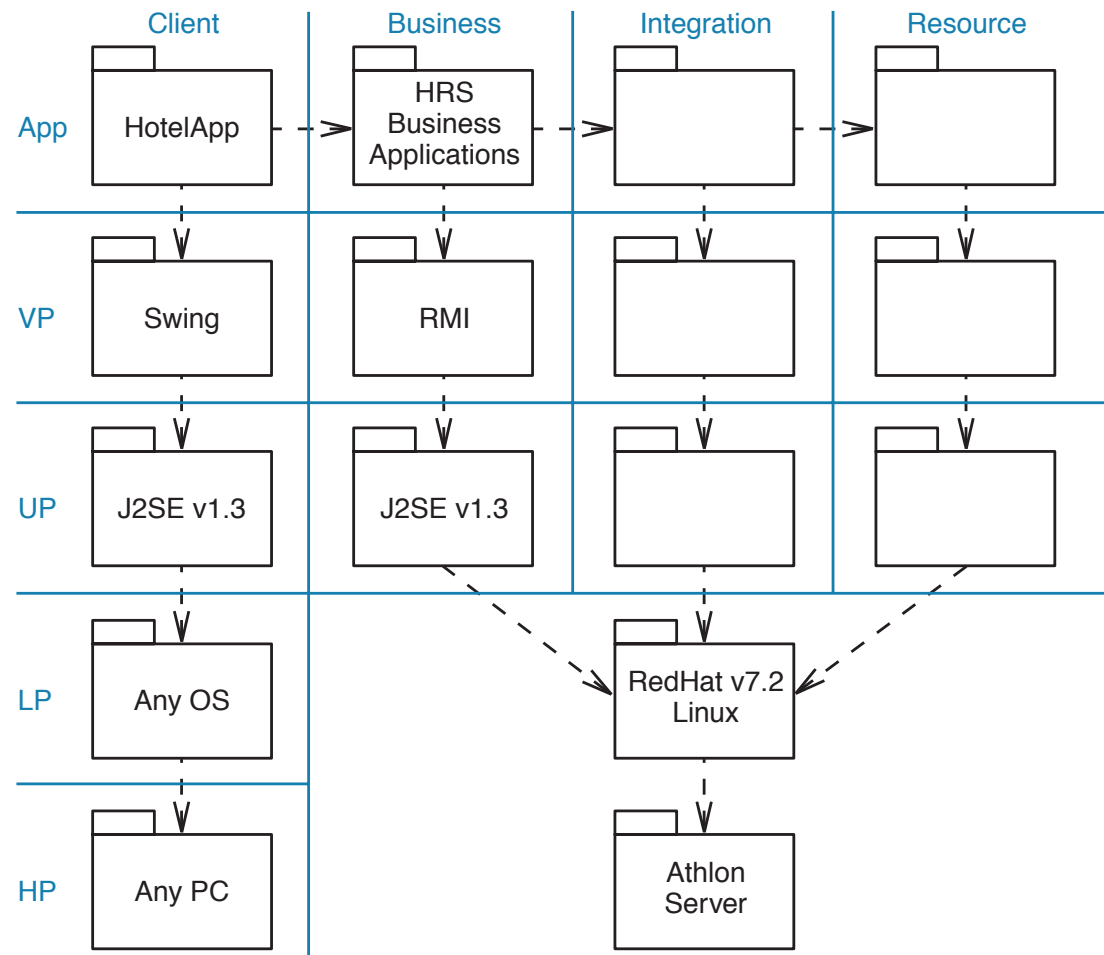


Overview of the Detailed Deployment Diagram





Overview of the Tiers and Layers Package Diagram





Exploring the Resource Tier

The Resource tier includes:

- Database
- File
- Web service
- Enterprise Information System (EIS)



Exploring Object Persistence

Persistence is “The property of an object by which its existence transcends time and space.” (Booch OOAD with Apps page 517)

A persistence object is:

- An object that exists beyond the time span of a single execution of the application
- An object that is stored independently of the address space of the application



Persistence Issues

Here are a few of the persistence issues that must be addressed:

- Type of data storage
- Database schema that maps to the Domain model
- Integration components
- CRUD operations: Create, Retrieve, Update, and Delete



Creating a Database Schema for the Domain Model

Defining the database schema is usually done in two phases:

- The logical entity-relationship (ER) diagram contains:
 - The OO entities as tables
 - The OO entity attributes as fields within the tables
 - The OO associations as relationships between tables
- The physical ER diagram takes the logical diagram and adds:
 - Data types on fields
 - Indexes on tables
 - Data integrity constraints



Creating a Database Schema for the Domain Model

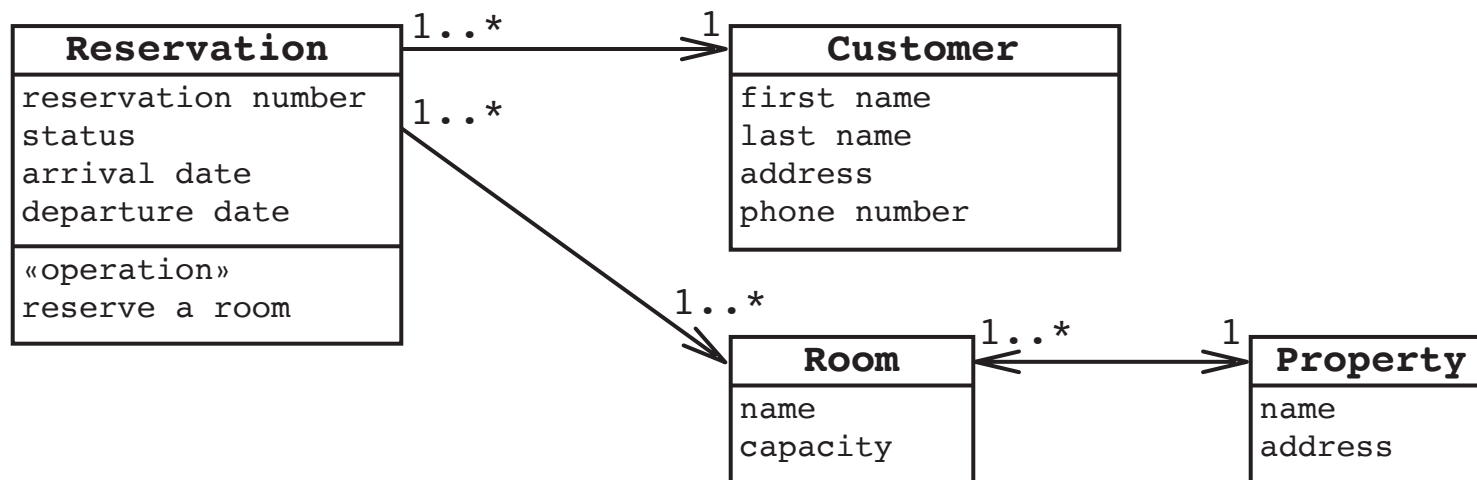
The strategy for converting the Domain model into a logical ER diagram is:

1. Convert each class into a table.
2. Specify the primary key for each table.
3. Use the association multiplicity to determine the type of ER association.



Simplified HRS Domain Model

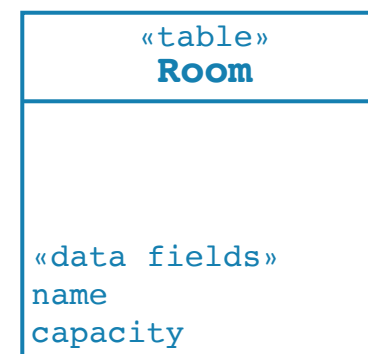
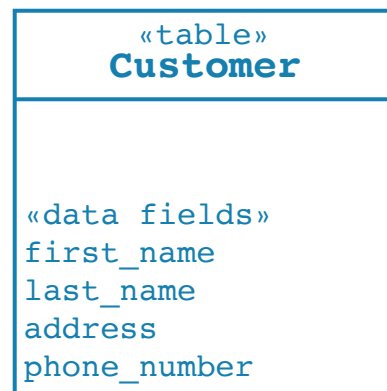
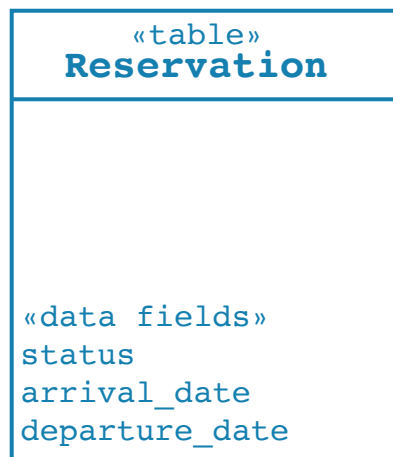
This example uses a simplified Domain model:





Step 1 – Map OO Entities to DB Tables

Create tables for each entity in the Domain model:





Step 2 – Specify the Primary Keys

Add the primary key fields:

«table» Reservation
«primary key» reservation_id
«data fields» status arrival_date departure_date

«table» Customer
«primary key» customer_id
«data fields» first_name last_name address phone_number

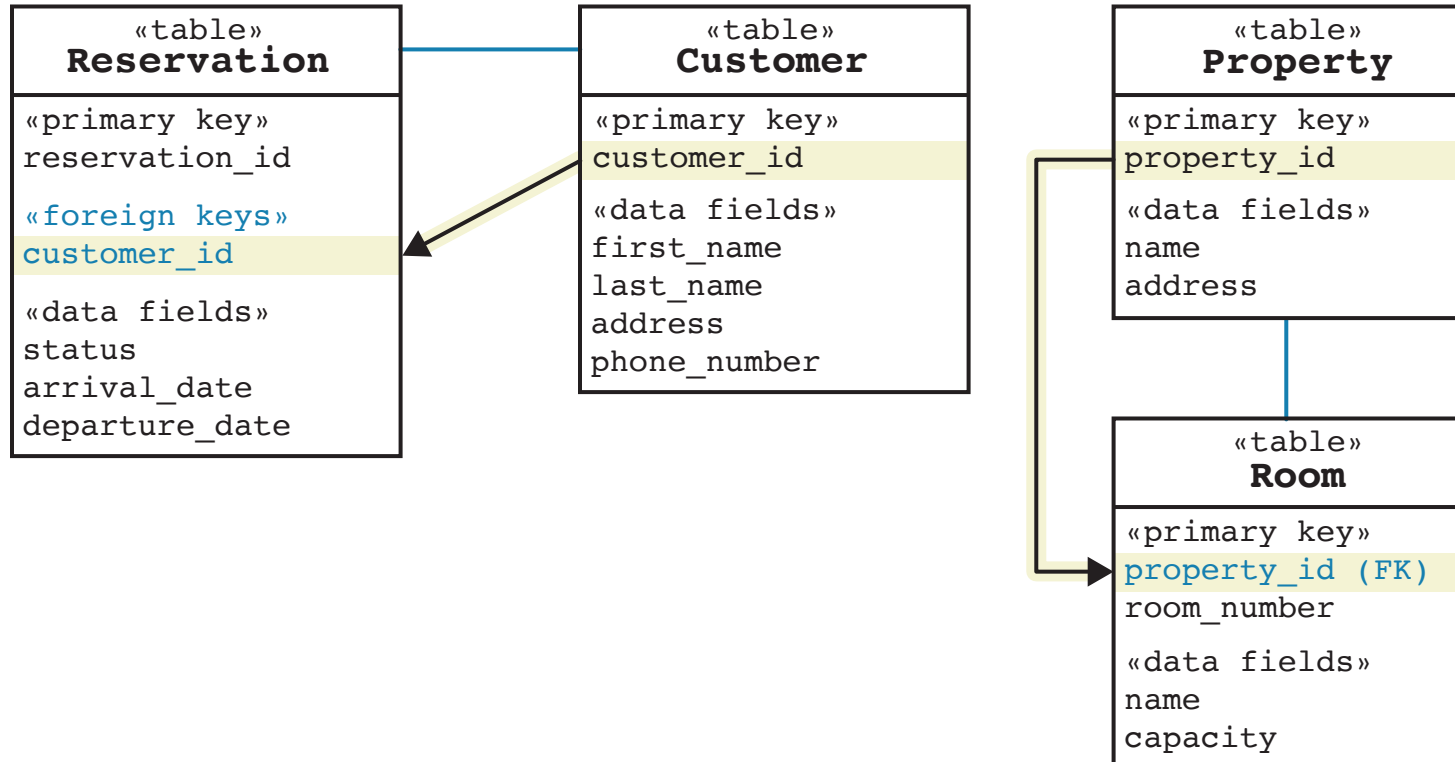
«table» Property
«primary key» property_id
«data fields» name address

«table» Room
«primary key» property_id (FK) room_number
«data fields» name capacity



Step 3 – Specify One-to-Many Relationships

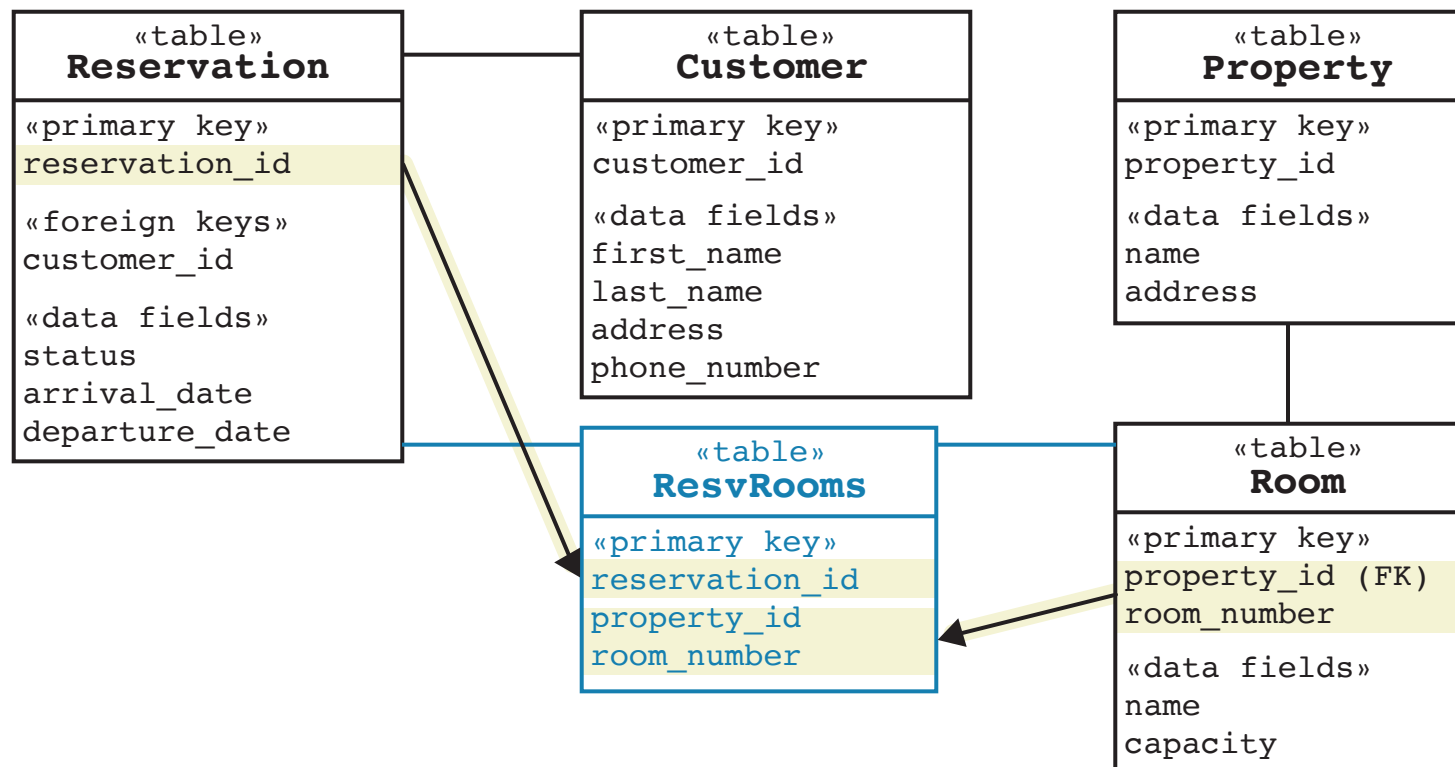
Use foreign keys:





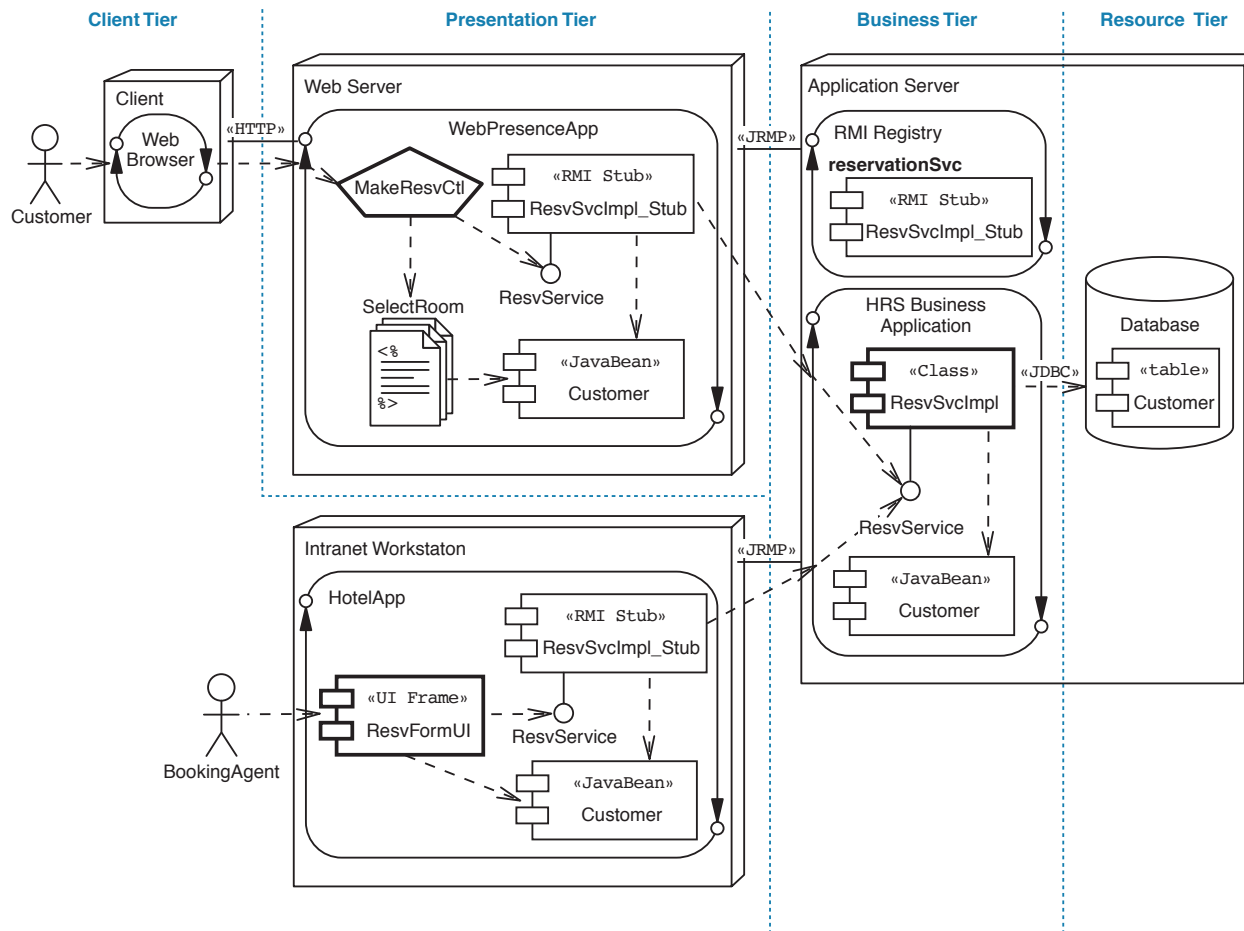
Step 3 – Specify Many-to-Many Relationships

Introduce a resolution table:



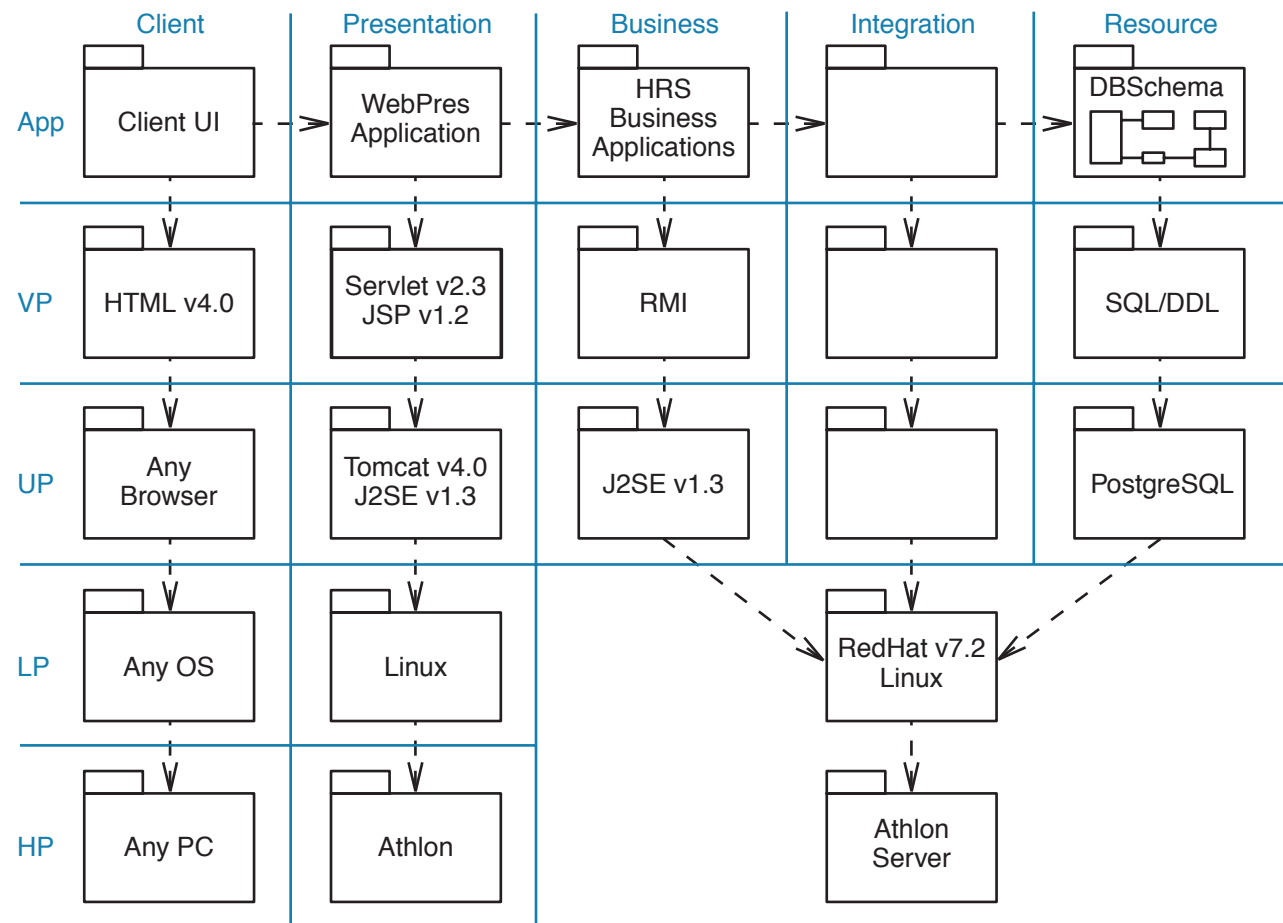


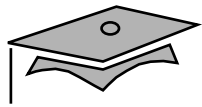
Overview of the Detailed Deployment Diagram





Overview of the Tiers and Layers Package Diagram





Exploring Integration Tier Technologies

The Integration tier separates the entity components from the resources.

Resources that require integration are:

- Data sources
- Enterprise Information Systems (EIS)
- Computation libraries
- Message services
- B2B services



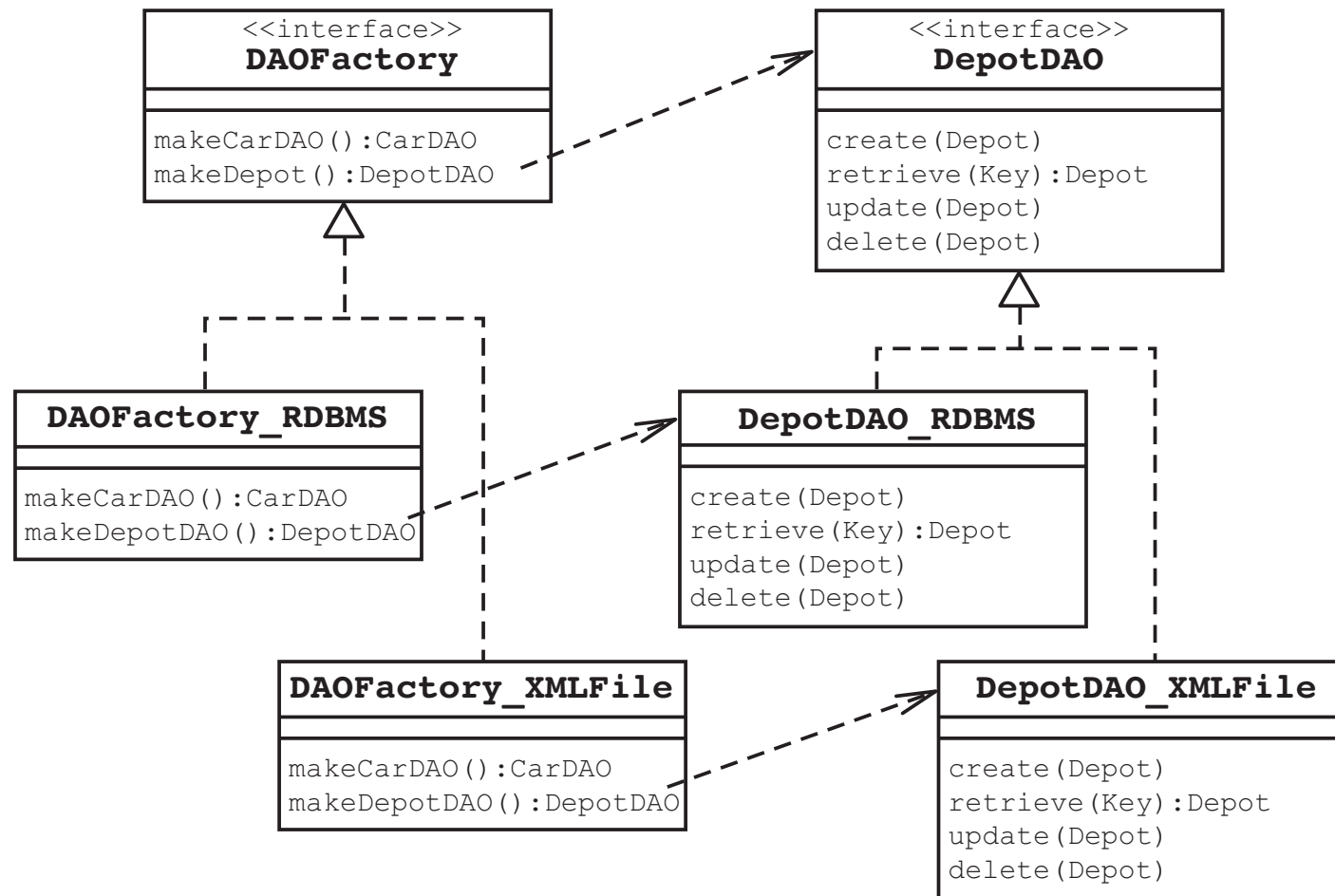
The DAO Pattern

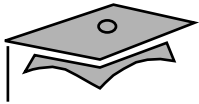
The Data Access Object (DAO) Pattern:

- Separates the implementation of the CRUD operations from the application tier
- Encapsulates the data storage mechanism for the CRUD operations for a single entity with one DAO component for each entity
- Provides an Abstract Factory for DAO components if the storage mechanism is likely to change

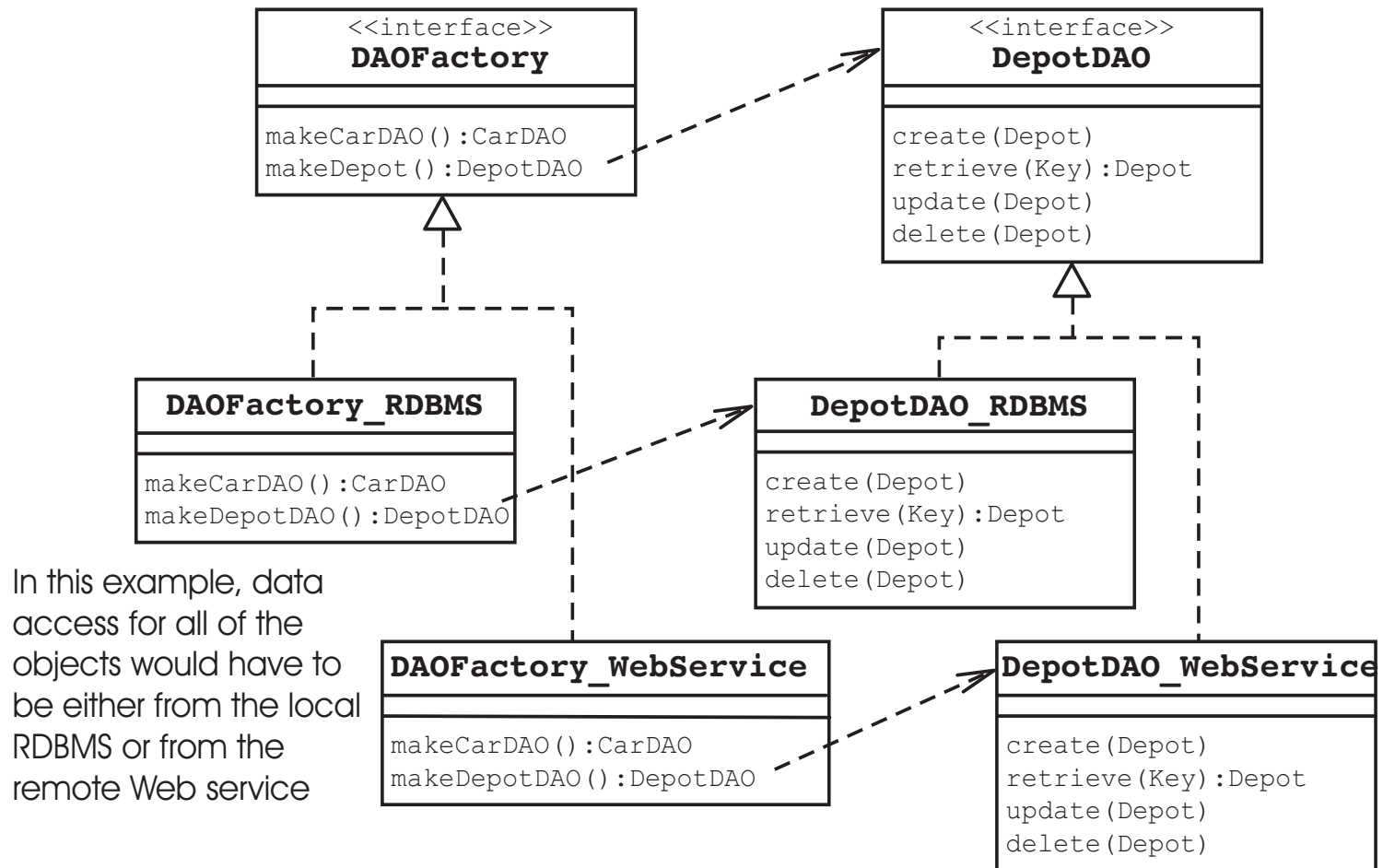


The DAO Pattern: Example 1



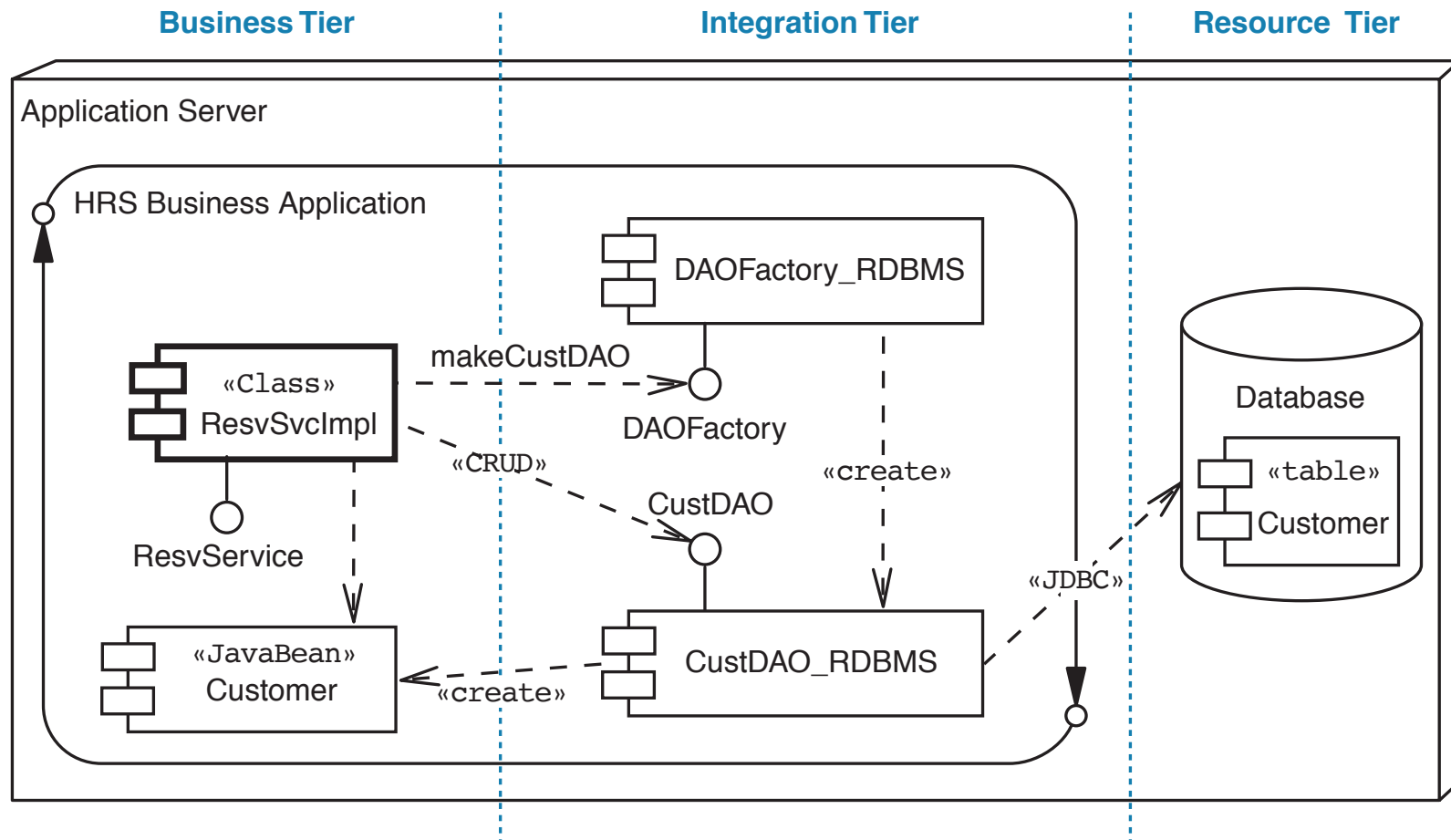


The DAO Pattern: Example 2



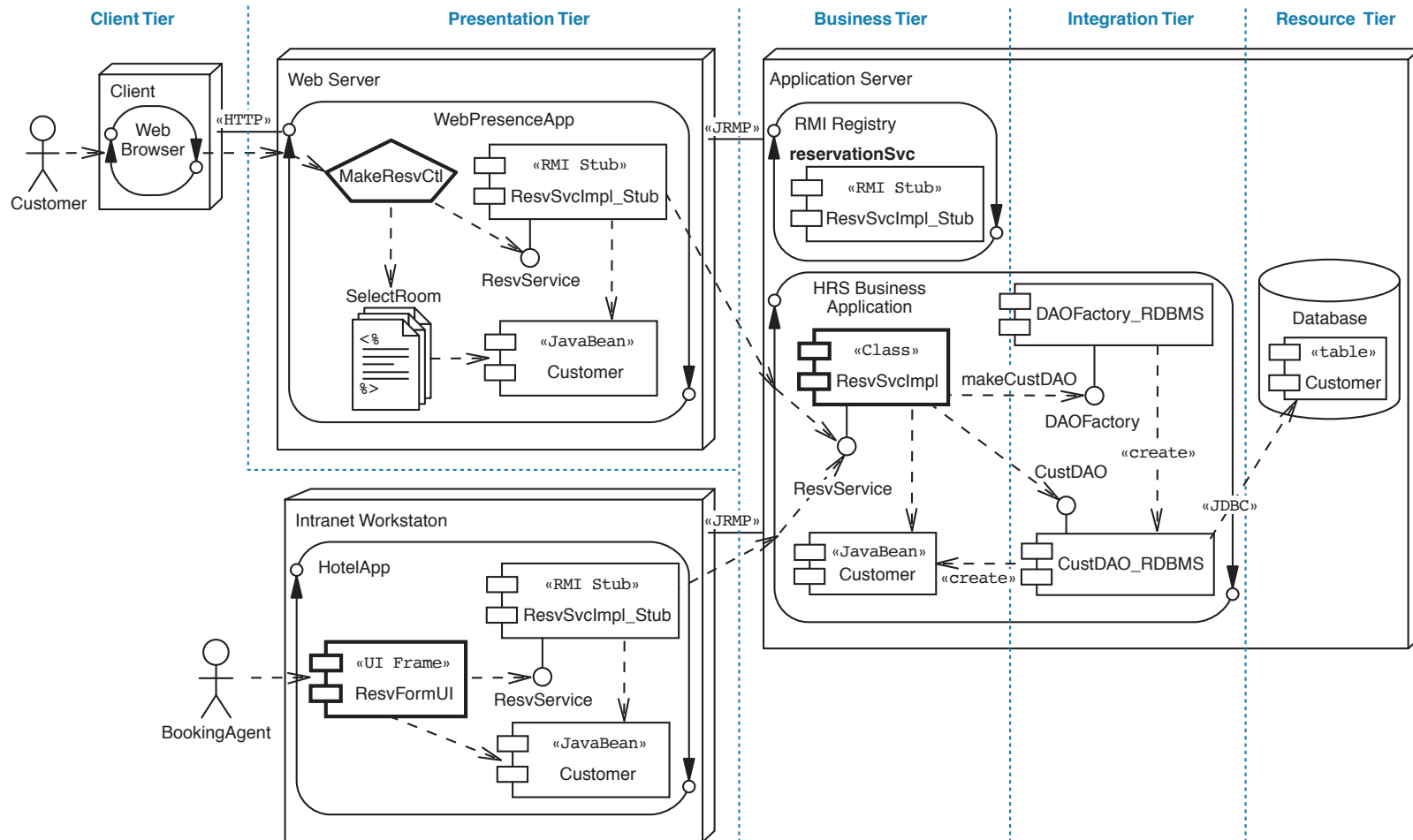


The DAO Pattern



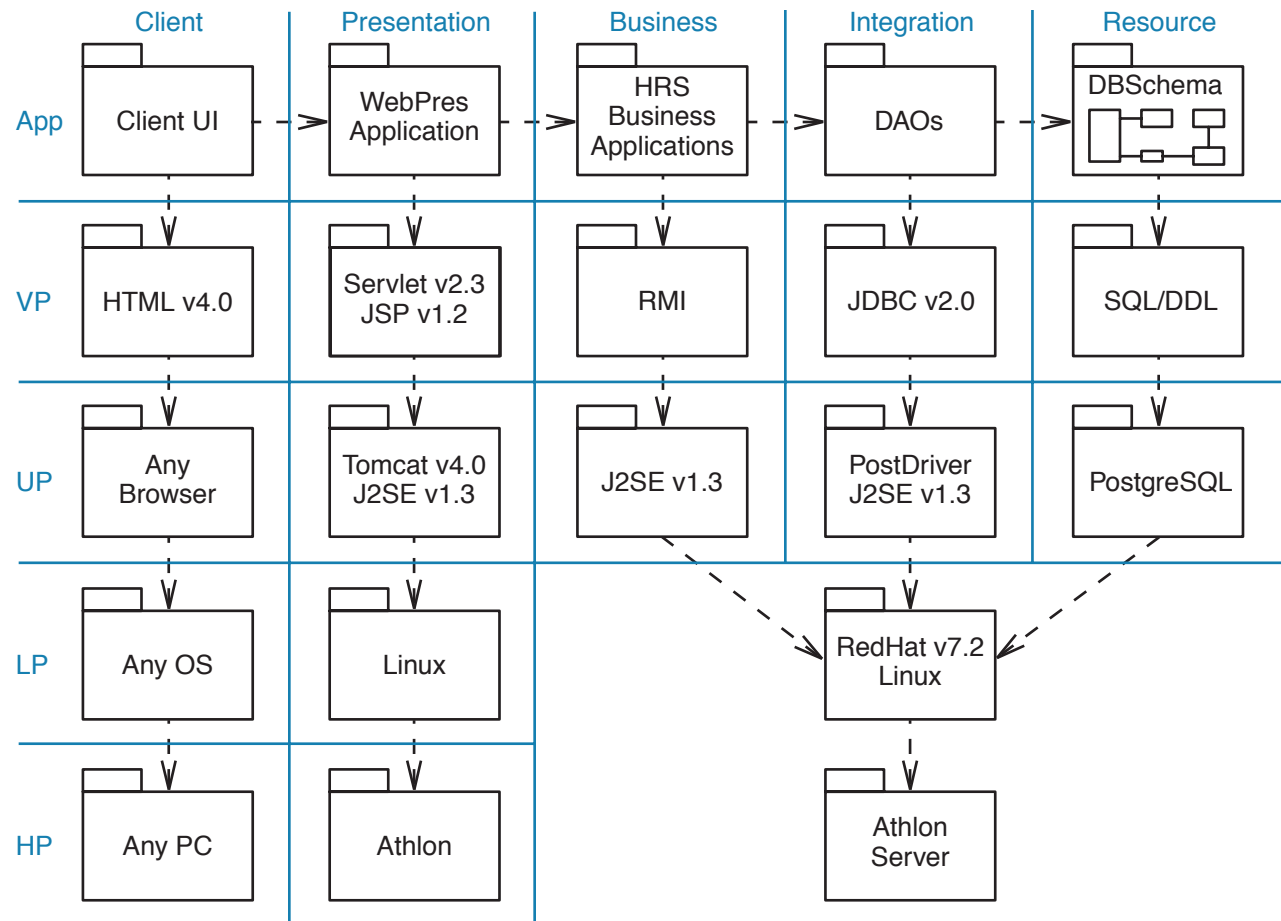


Overview of the Detailed Deployment Diagram





Overview of the Tiers and Layers Package Diagram





Java™ Persistence API

Java Persistence API:

- Is an alternative to DAO
- Draws on the best ideas from alternative persistence technologies such as Hibernate, TopLink, and Java Data Objects (JDO)
- Uses a persistence provider such as Hibernate or TopLink
- Is a Plain Old Java Object (POJO) persistence API for Object/Relational mapping
- Uses an entity manager to manage objects



Java Persistence API Entity Manager

A Java Persistence API entity manager can be requested to:

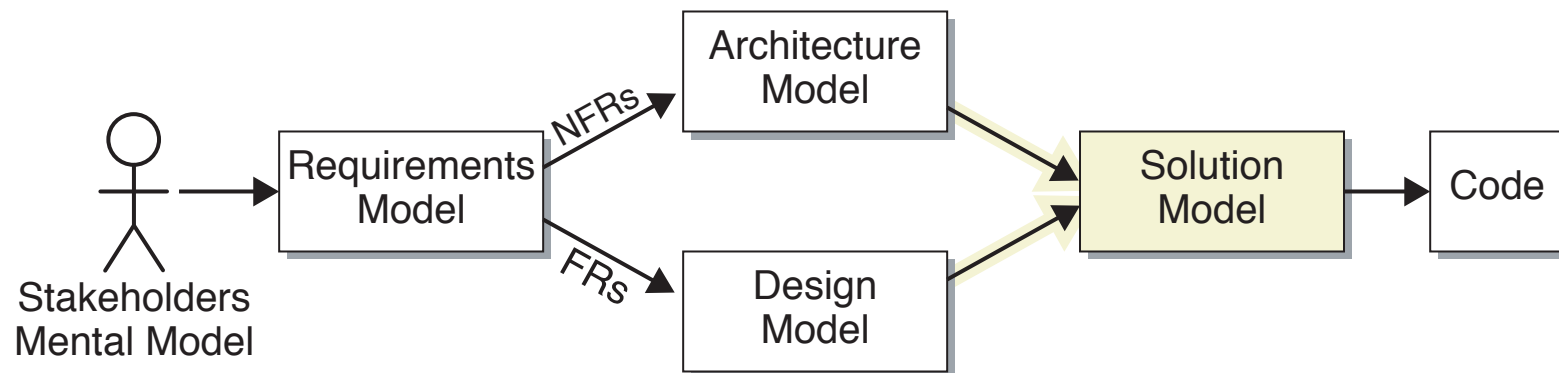
- Manage objects by keeping their data in synchronization with the database record
- Persist or merge unmanaged objects, which makes these objects managed
- Find an object using the primary key, which creates an object from the database record

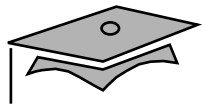


Introducing the Solution Model

The Solution model is the basis upon which the development team will construct the code of the system solution.

The Solution model is constructed by merging the Design model into the Architecture model (template).





Overview of a Solution Model for GUI Applications

GUI applications use the standard set of design components.

Entity

Icon

Boundary



Service

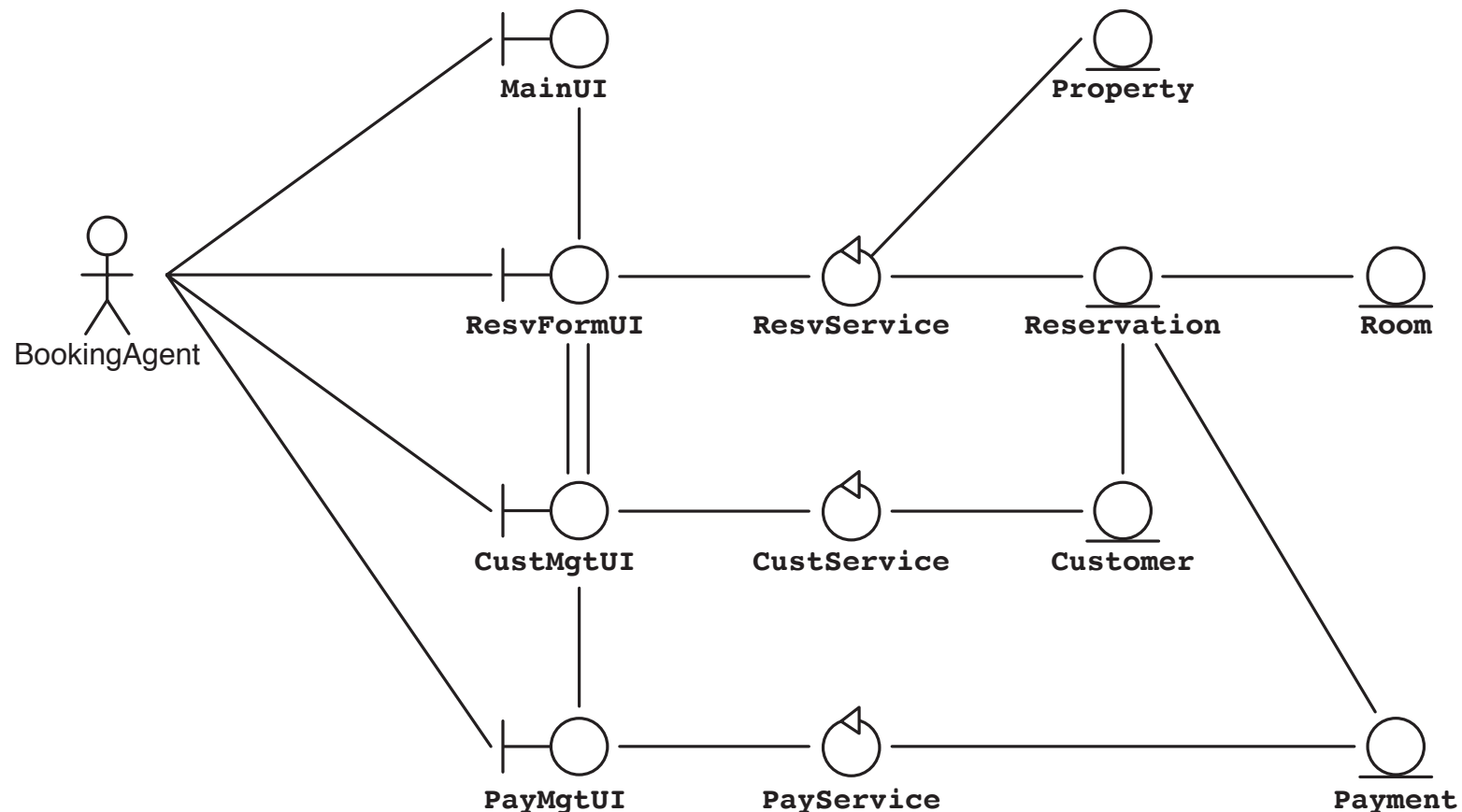


Entity



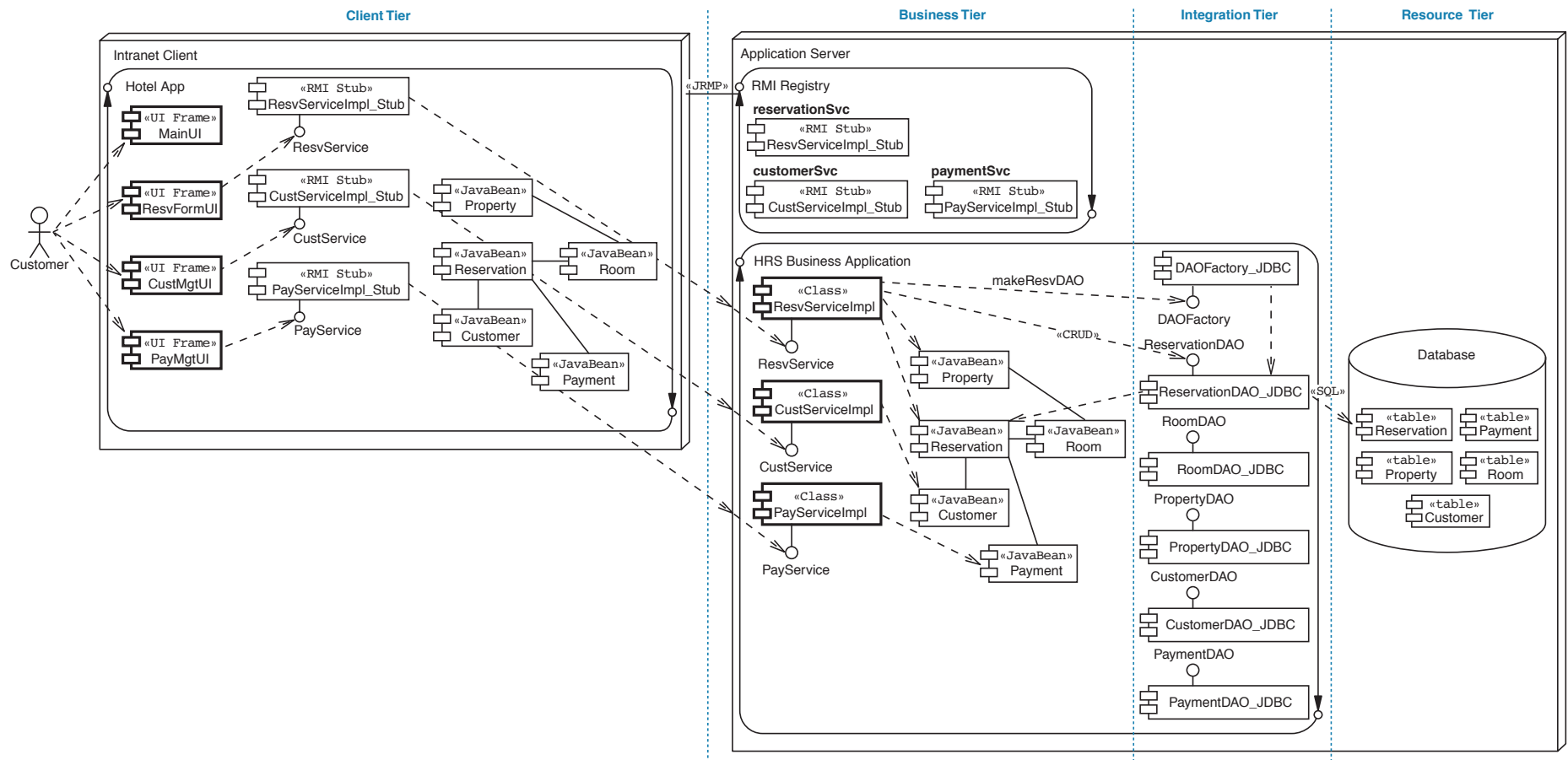


A Complete Design Model for the Create Reservation Use Case





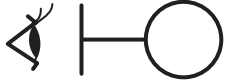



A Complete Solution Model for the Create Reservation Use Case





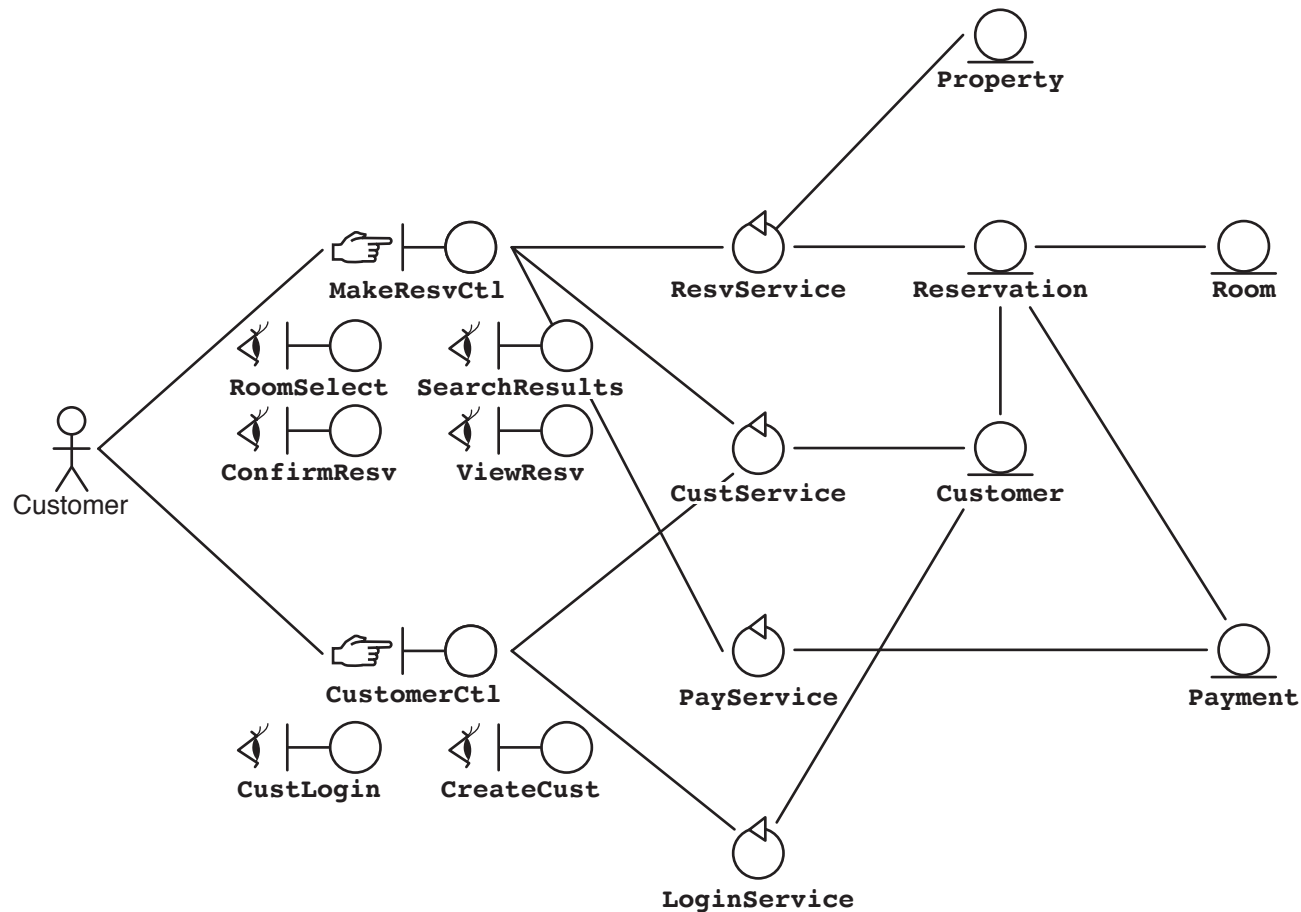
Overview of a Solution Model for WebUI Applications

Web applications split the Boundary component into two separate components: Views and Controllers.

Entity	Icon
View	
Controller	
Service	
Entity	

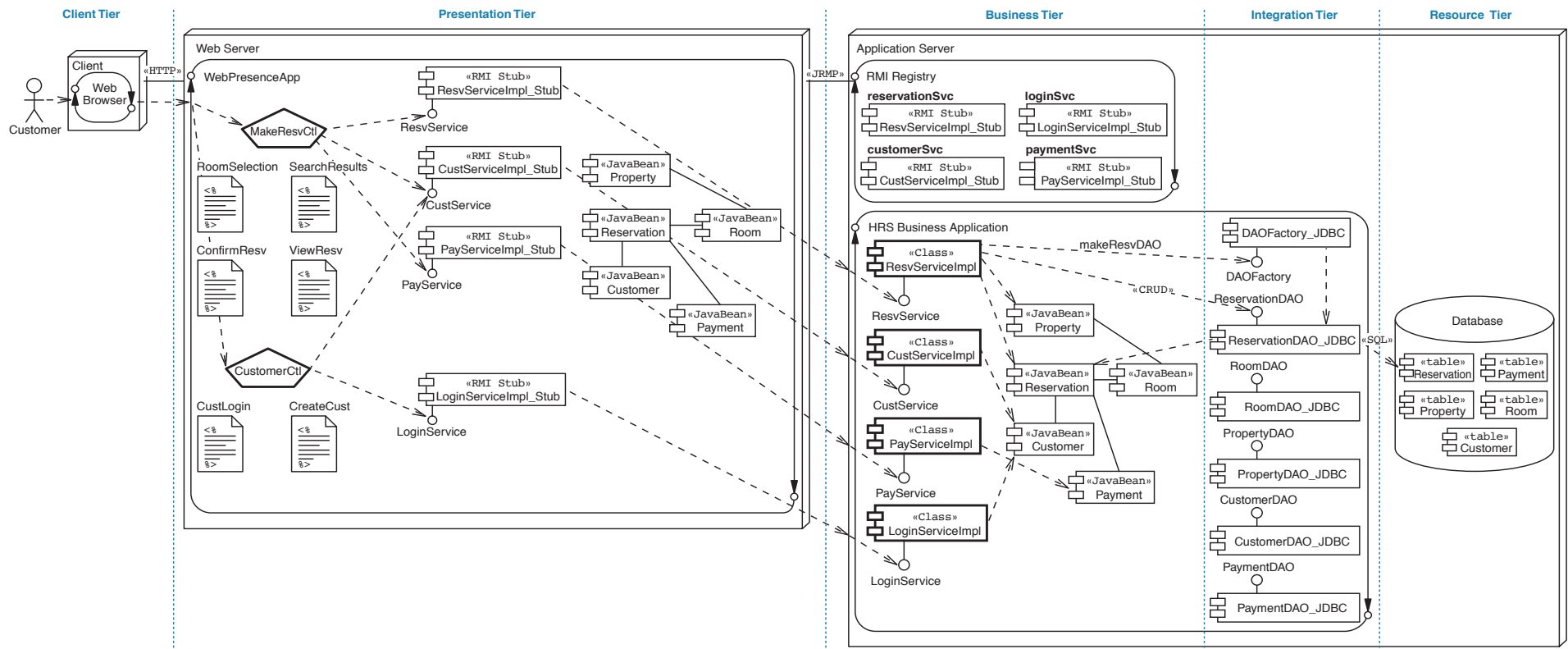


A Complete Design Model for the Create Reservation Use Case





A Complete Solution Model for the Create Reservation Online Use Case





Summary

- The Client and Presentation tiers include Boundary View and Controller components.
- The Business tier includes business services and entity components.
- The Integration tier includes components to separate the business entities from the resources.
- The Resource tier includes one or more data sources such as an RDBMS, OODBMS, EIS, Web services, or files.
- The Solution model provides a view of the software system that can be implemented in code.
- The Solution model is created by merging the Design model into the Architecture model (template).



Module 13

Refining the Class Design Model



Objectives

Upon completion of this module, you should be able to:

- Refine the attributes of the Domain model
- Refine the relationships of the Domain model
- Refine the methods of the Domain model
- Declare the constructors of the Domain model
- Annotate method behavior
- Create components with interfaces



Refining Attributes of the Domain Model

Refining attributes involves the following:

- Refining the metadata of the attributes
- Choosing an appropriate data type
- Creating derived attributes
- Applying encapsulation



Refining the Attribute Metadata

An attribute declaration in UML Class diagrams includes the following:

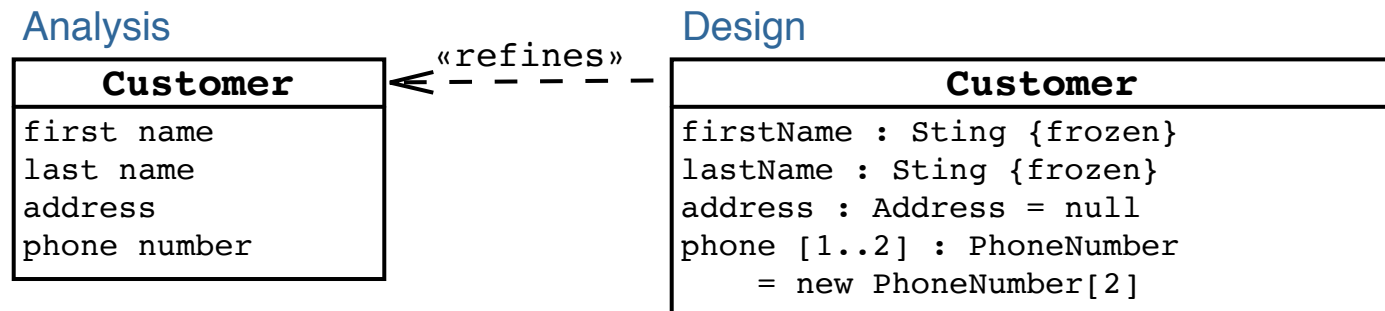
- Name
- Visibility
- Type
- Multiplicity
- Initial value
- Constraint (one of changeable, addOnly, or frozen)



Refining the Attribute Metadata

Syntax:

[visibility] name [multiplicity] [: type] [= init-value] [{constraint}]





Choosing an Appropriate Data Type

Choosing a data type is a trade-off of:

- Representational transparency
- Computational time
- Computational space



Choosing an Appropriate Data Type

For a phone number attribute:

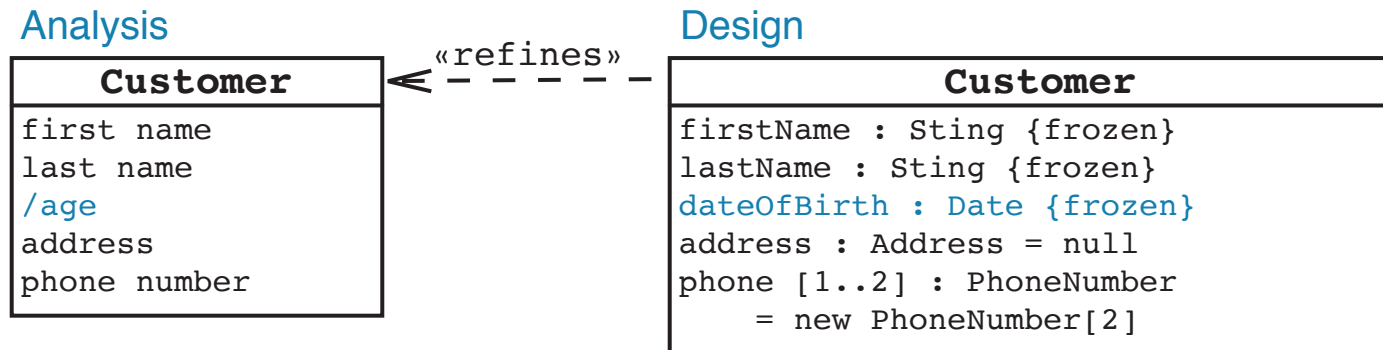
Data Type	Discussion
String	This data type might require mapping between the UI representation and the storage (DB) representation.
long	This data type conserves space, but might not be sufficient to represent large phone number (such as international numbers).
PhoneNumber	A value object is a class that represent the phone data. This data type representationally transparent, but requires additional coding.
char array	This data type is similar to a String and adds no value.
int array	This data type conserves space, but is not representationally transparent.



Creating Derived Attributes

In Analysis, you might have an attribute that you know can be derived from another (more stable) source.

The canonical example is the calculation of a person's age from their date of birth:





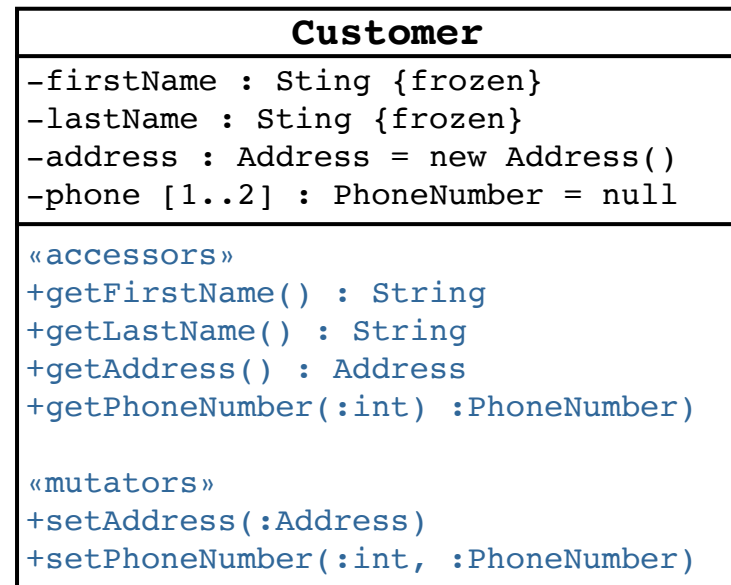
Applying Encapsulation

To apply encapsulation, follow these steps:

1. Make all attributes private (visibility).
2. Add public accessor methods for all readable attributes.
3. Add public mutator methods for all writable (non-frozen) attributes.



An Encapsulation Example





Refining Class Relationships

There is no clear distinction between Analysis and Design, especially in regards to modeling class associations.

Design usually addresses these details:

- Type: association, aggregation, and composition
- Direction of traversal (also called navigation)
- Qualified associations
- Declaring association management methods
- Resolving many-to-many associations
- Resolving association classes



Relationship Types

There are three types of relationships:

- Association
- Aggregation
- Composition

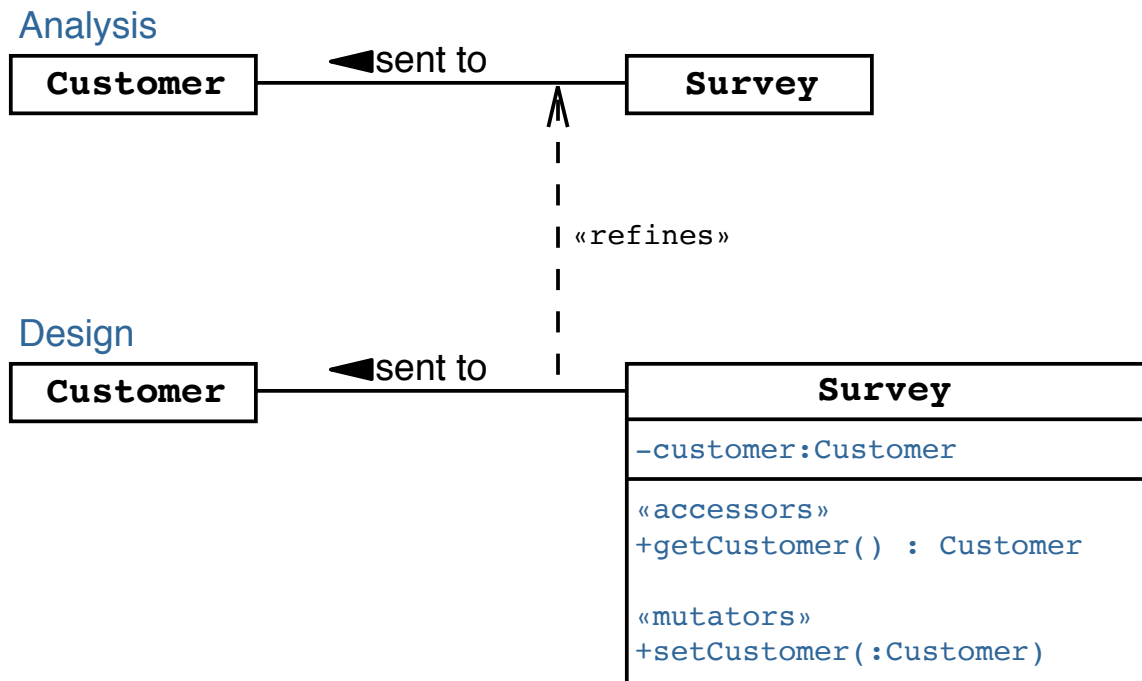
These relationships imply that the related object is somehow tied to the original object (usually as an instance attribute).

There is another type of relationship, Dependency, which states that one object uses another object to do some work, but that there is no instance attribute holding that object.



Association

“The semantic relationship between two or more classifiers that specifies connections among their instances.” (OMG page 537)





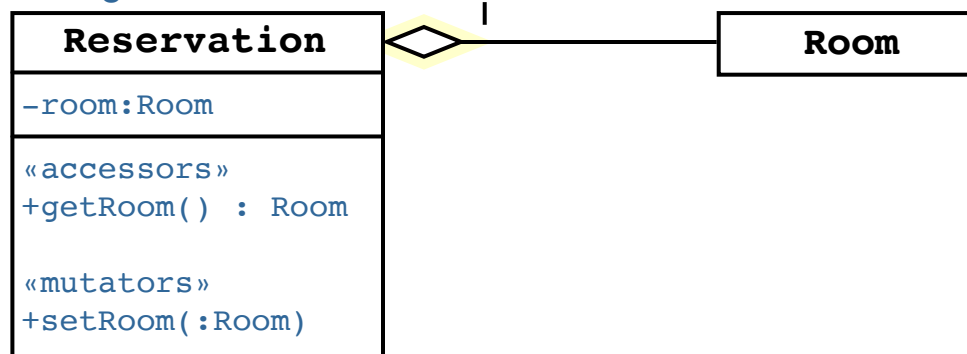
Aggregation

“A special form of association that specifies a whole-part relation between the aggregate (whole) and a component part.”
(OMG page 537)

Analysis



Design

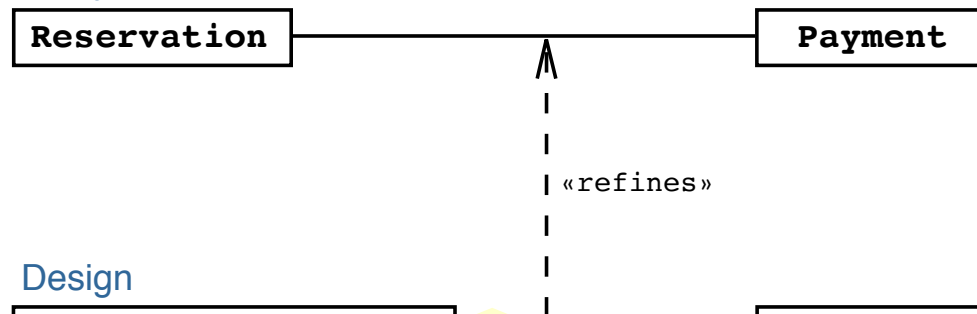




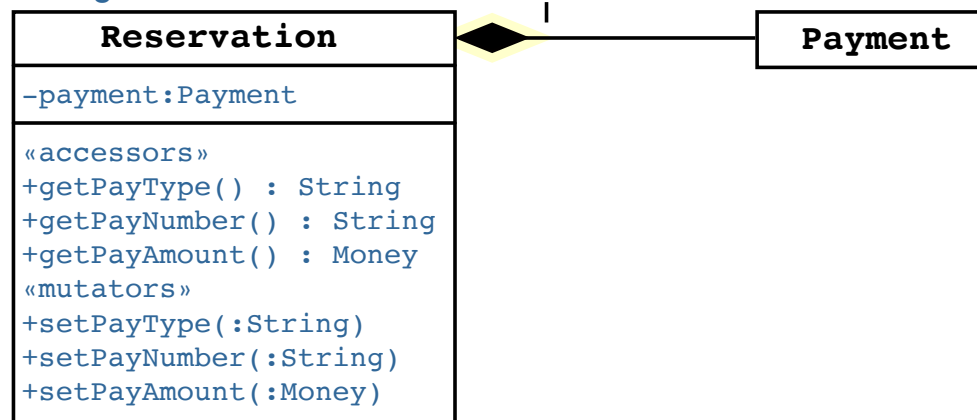
Composition

“A form of aggregation that requires that a part instance be included in at most one composite at a time, and that the composite object is responsible for the creation and destruction of the parts.” (OMG page 540)

Analysis



Design

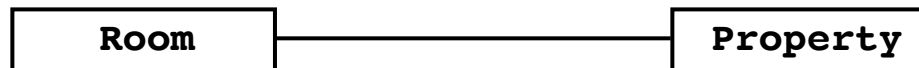




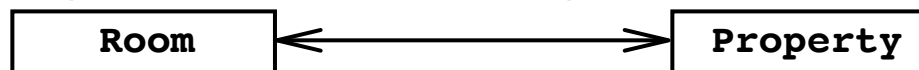
Navigation

A navigation arrow shows the direction of object traversal at runtime.

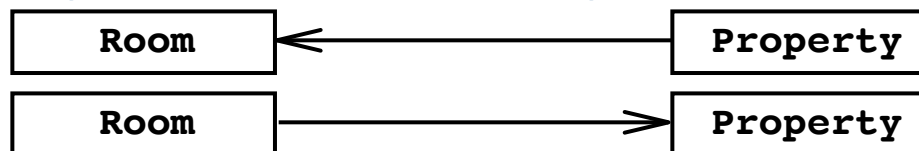
Implicit bidirectional relationship



Explicit bidirectional relationship



Explicit unidirectional relationship

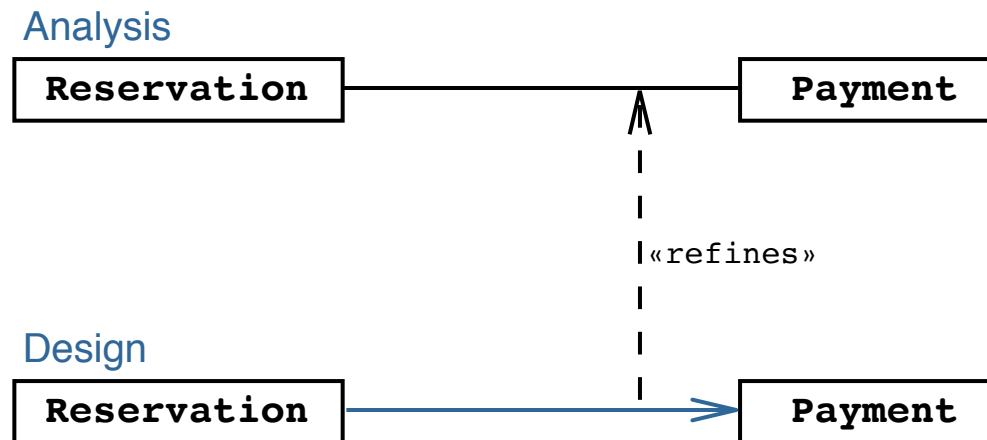




Navigation

Sometimes in analysis, you do not know what direction the software will need to navigate the association. This problem should be resolved in design.

For example:

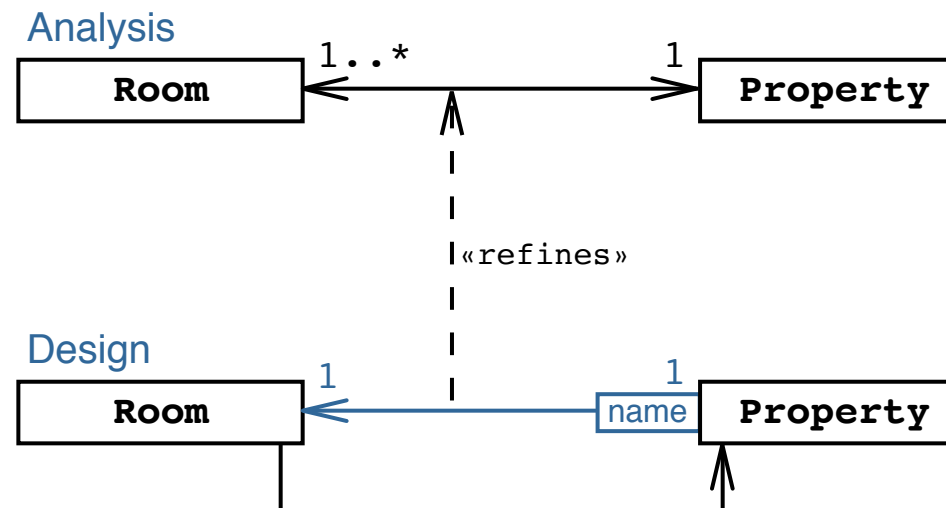


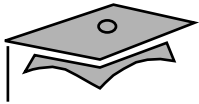


Qualified Associations

In one-to-many or many-to-many associations, it is often useful to model how the system will access a single element in the association.

For example:

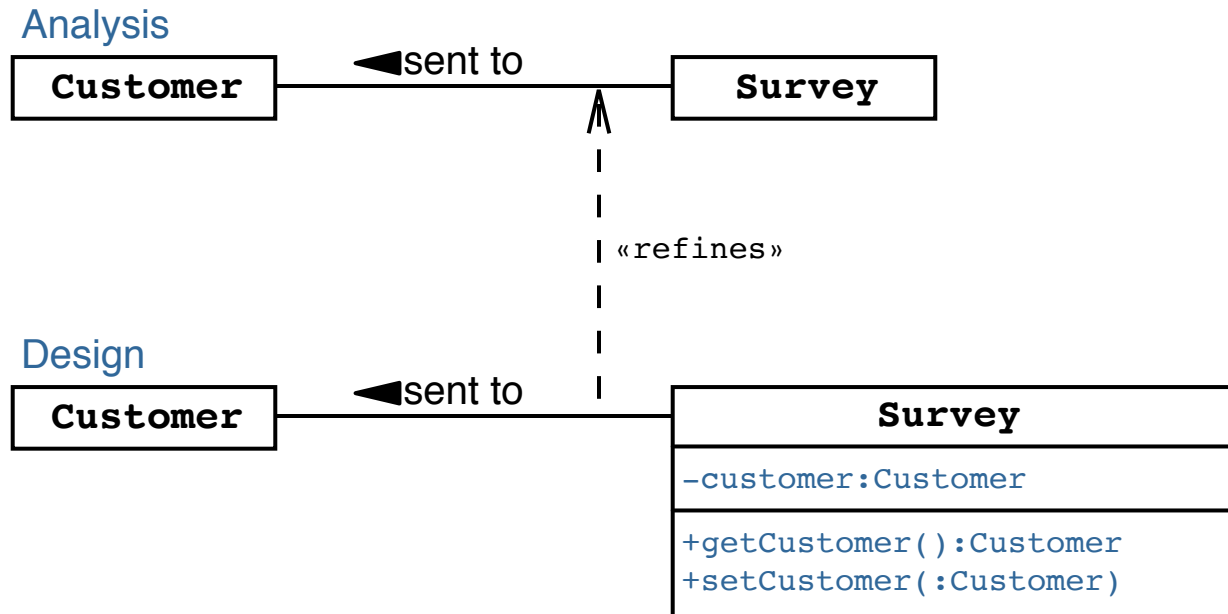




Relationship Methods

Association methods enable the client to access and change associated objects. There are three cases: one-to-one, one-to-many, and many-to-many.

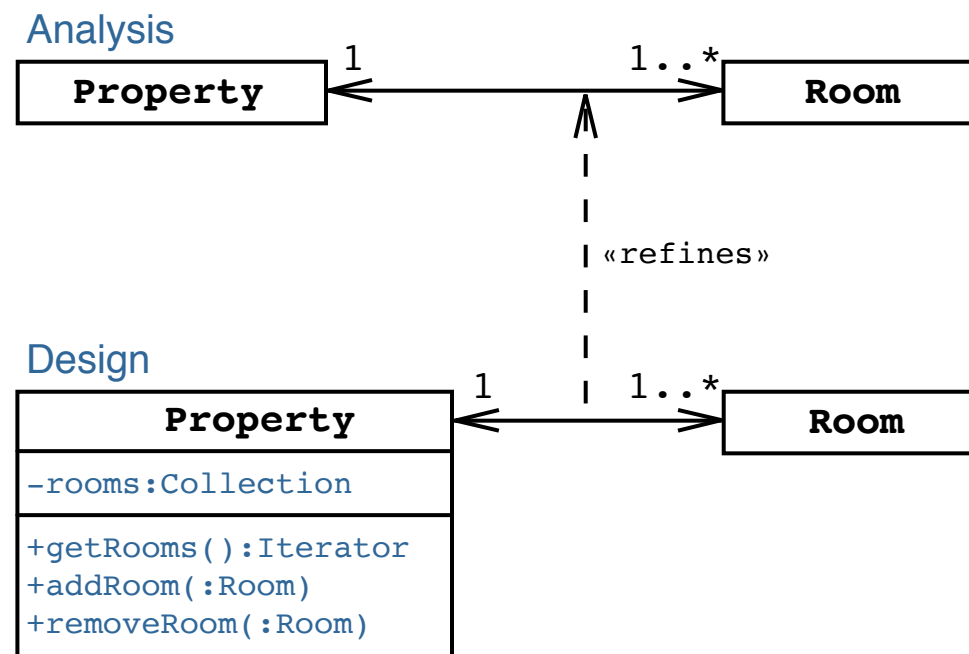
One-to-one relationships require a single instance variable:





Relationship Methods

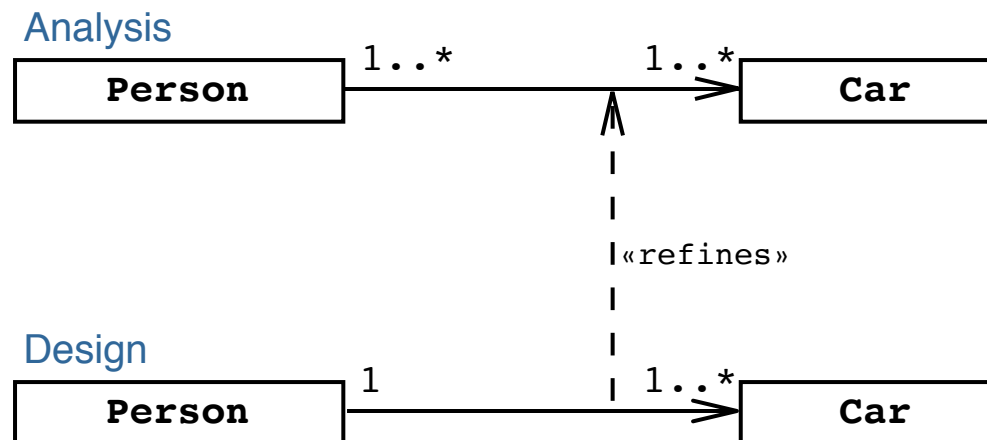
One-to-many relationships require the use of collections:





Resolving Many-to-Many Relationships

Managing many-to-many associations is challenging. Consider dropping this requirement at design-time.

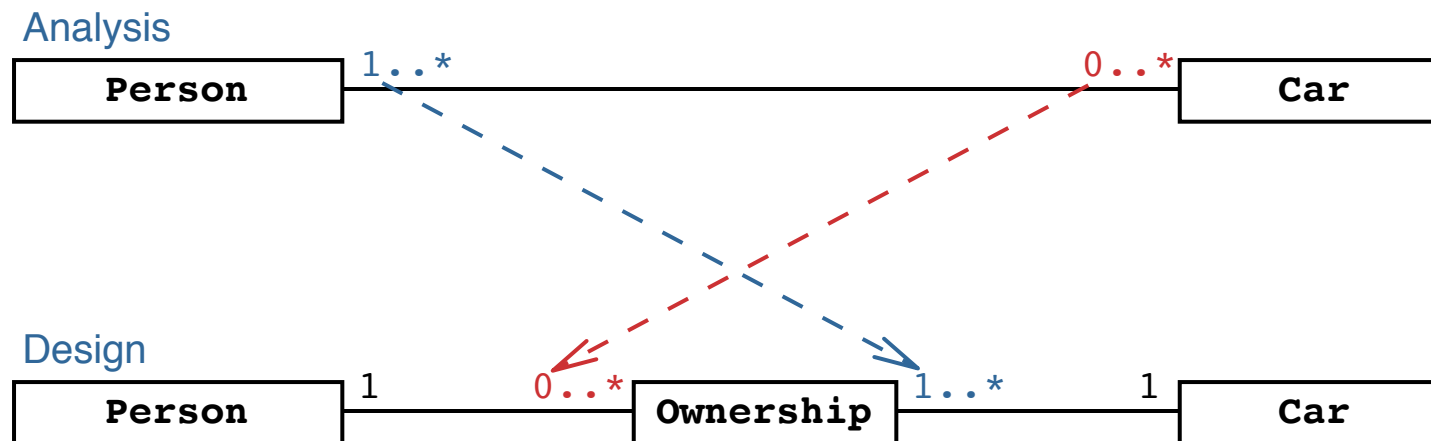




Resolving Many-to-Many Associations

If the many-to-many association must be preserved, you can sometimes add a class in between that reduces the single many-to-many association to two one-to-many associations.

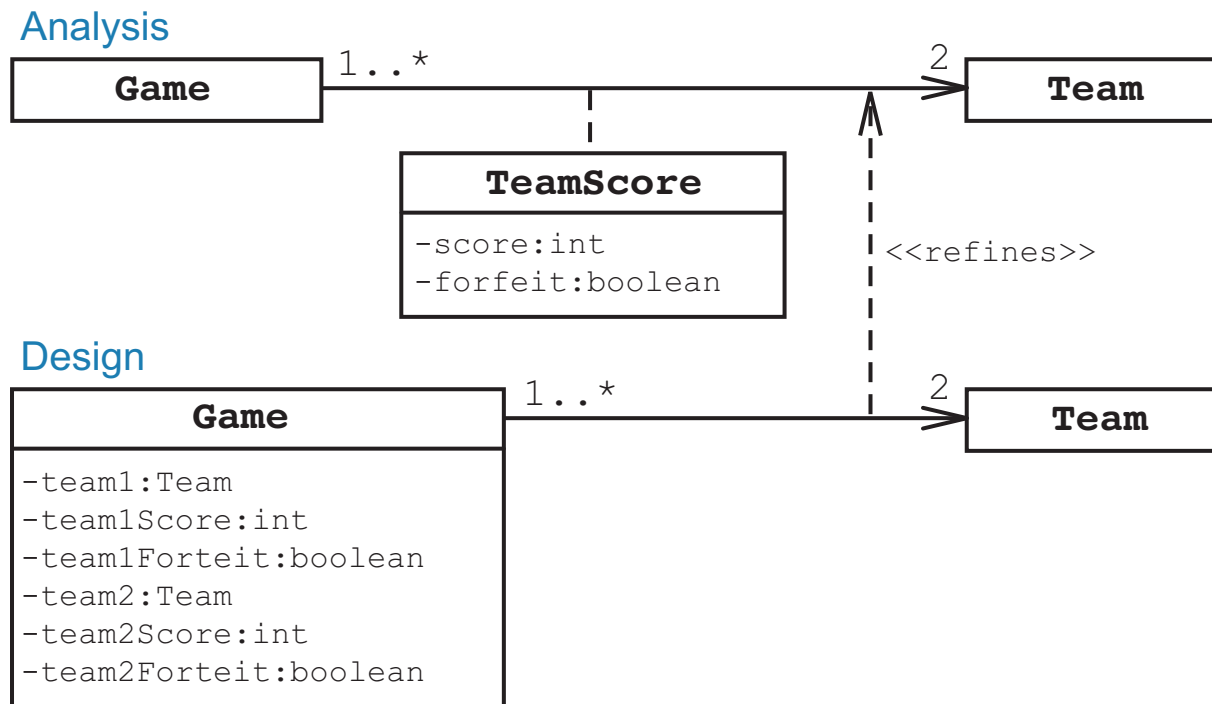
For example:





Resolving Association Classes

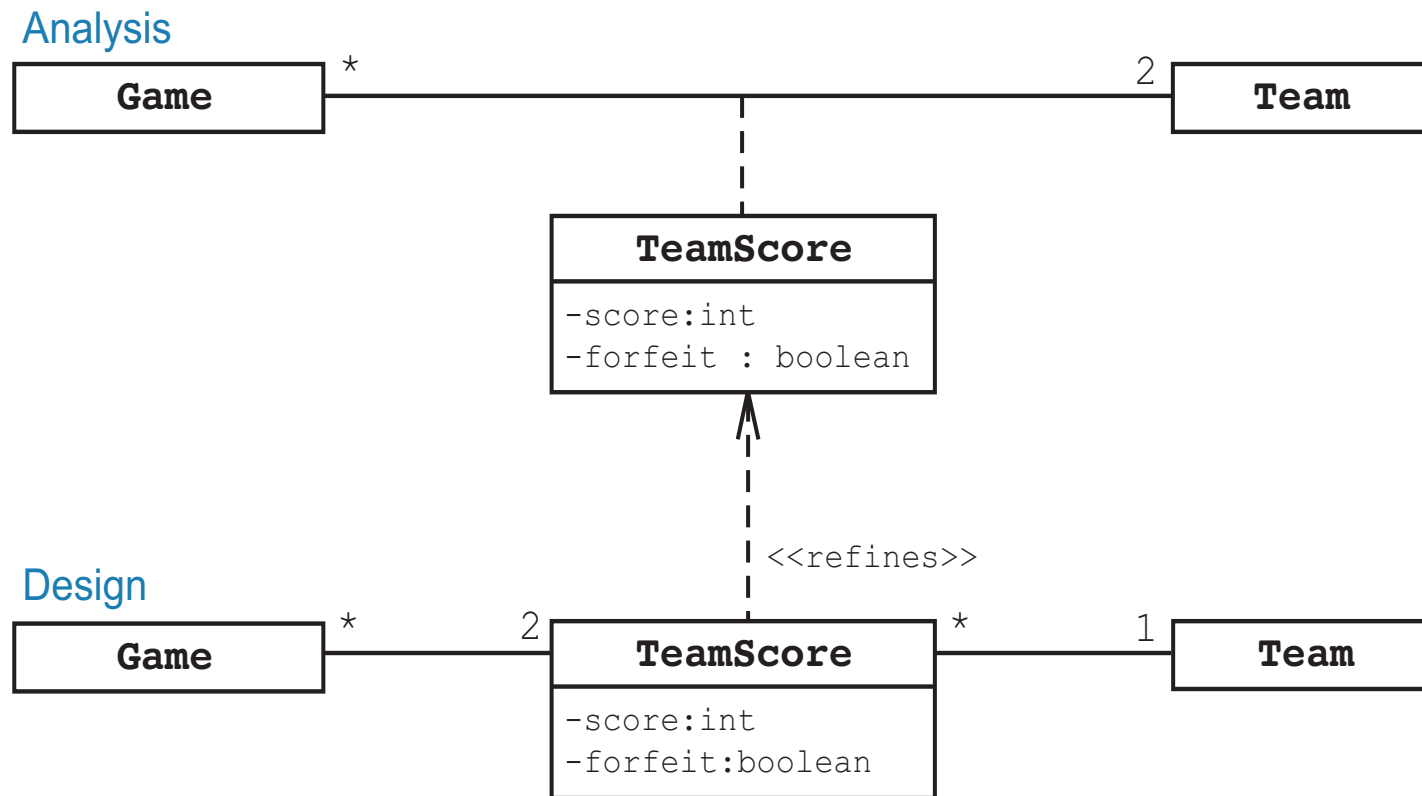
An association class can only exist in the Analysis model. It should be resolved into a programmable class at design.





Resolving Association Classes

Alternatively, the association class can be placed in between the two primary classes:





Refining Methods

Methods are identified during the following workflows:

- CRC analysis, which determines responsibilities
- Robustness analysis, which identifies methods in Service classes
- Design, which identifies accessor and mutator methods for attributes and associations

Other types of methods:

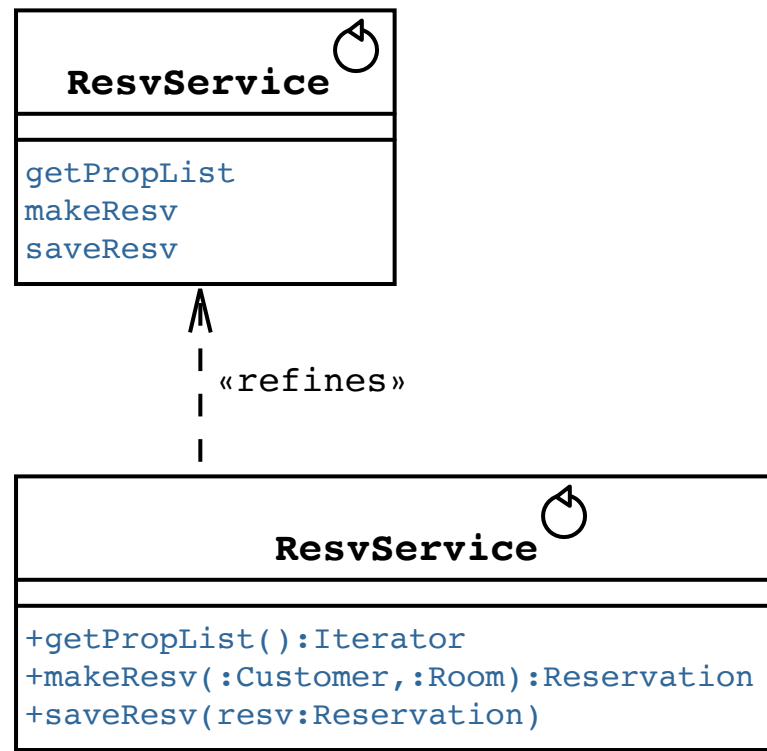
- Object management
- Unit testing
- Recovery, inverse, and complimentary operations

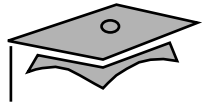


Refining Methods

Syntax:

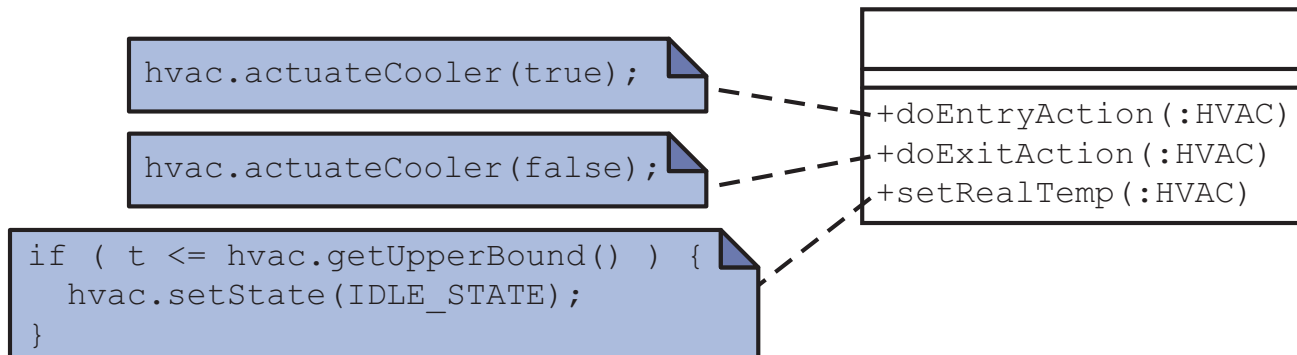
```
[visibility] name [({[param] [:type]}*)]  
[:return-value] [{constraint}]
```





Annotating Method Behavior

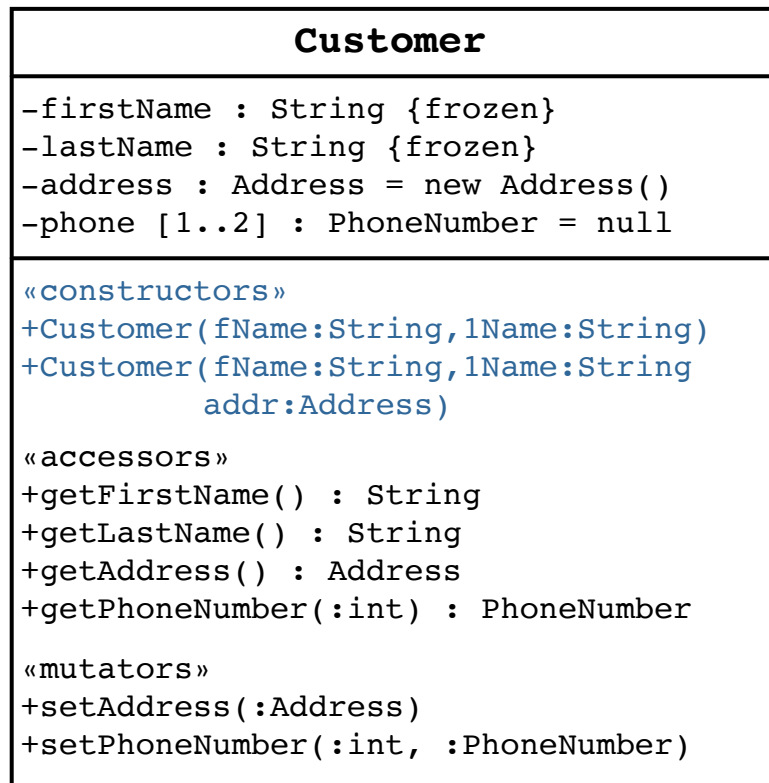
UML annotations can be attached to each method to document the behavior of the method.





Declaring Constructors

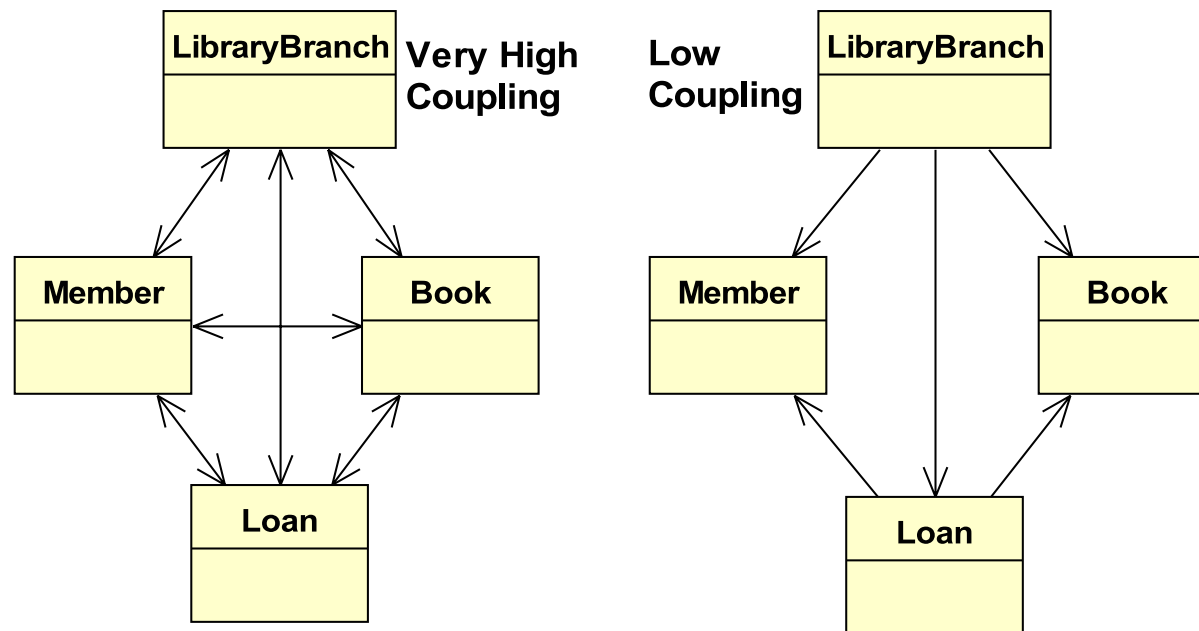
Constructors initialize an object and a syntax similar to methods:





Reviewing Coupling

Ideally, your model should have the lowest coupling while maintaining the FRs and NFRs.

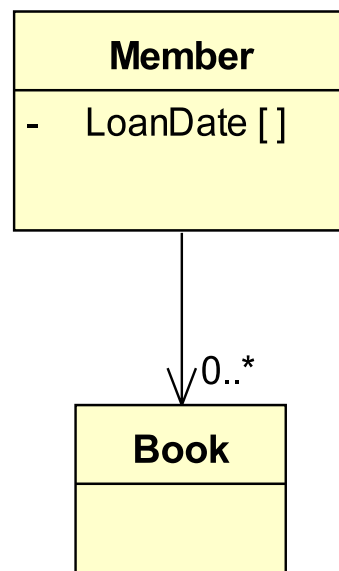




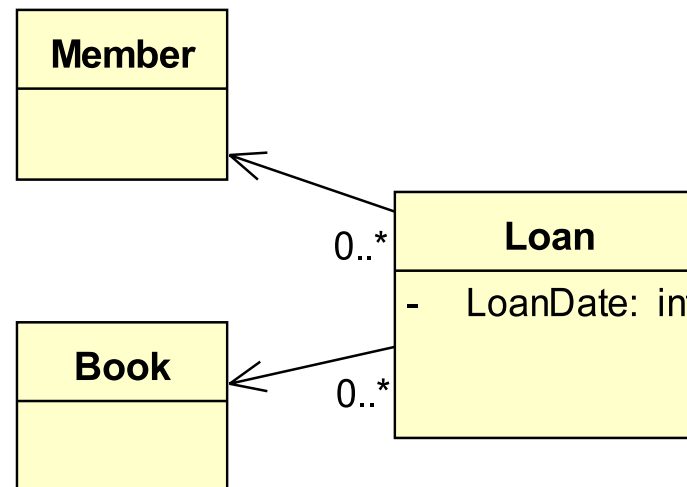
Reviewing Cohesion

Ideally, your model should have the highest cohesion.

Lower Cohesion



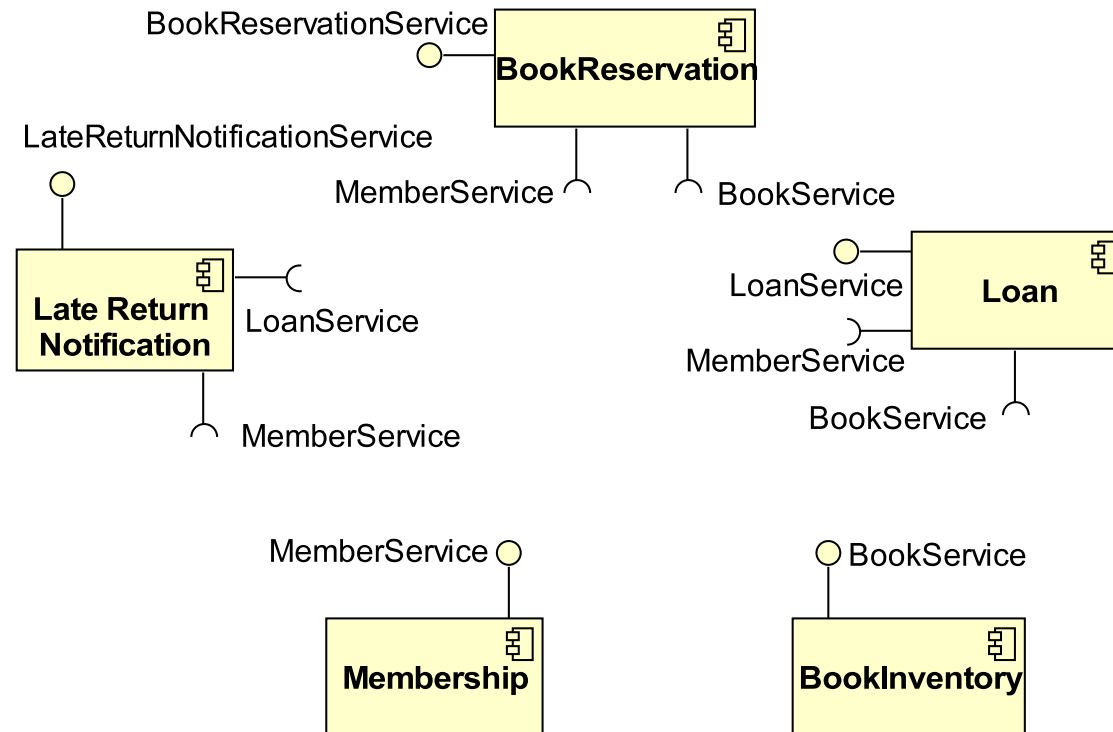
Higher Cohesion





Creating Components with Interfaces

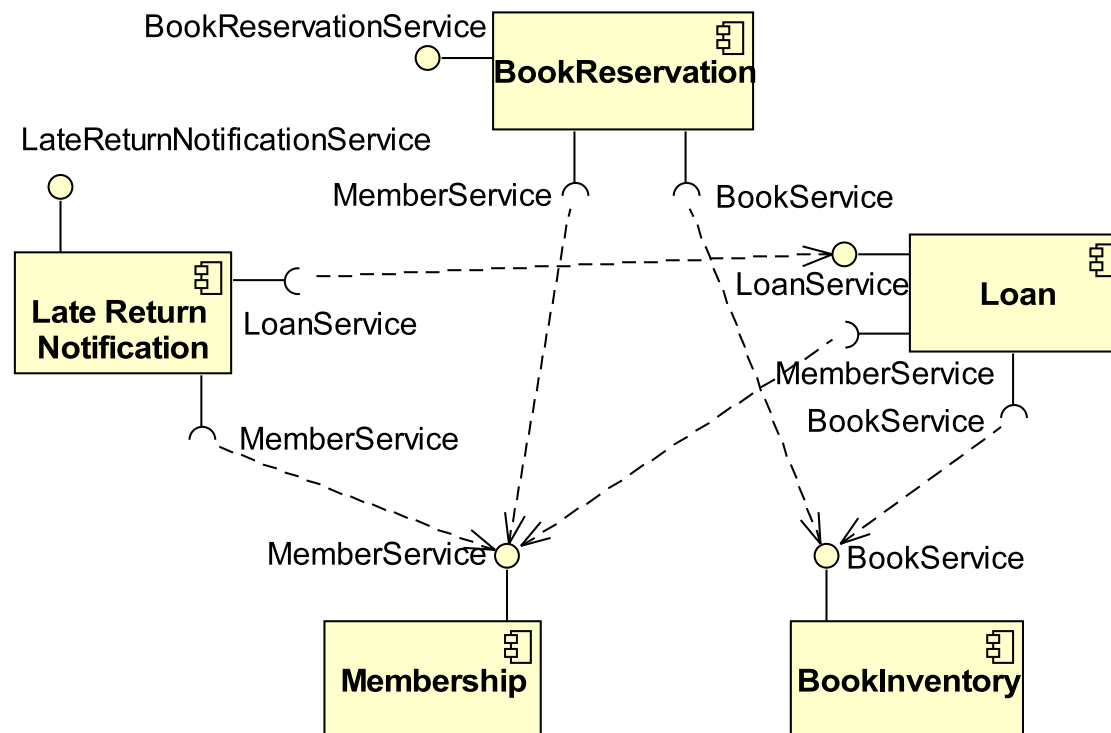
Classes can be grouped into cohesive components with well-defined provided and required interfaces.





Creating Components with Interfaces

Dependency arrows are not required if you are using the Required Interface notation. However, you may show the arrows as illustrated in the following example:





Summary

- During the Design workflow, you must refine the Domain model to reflect the implementation paradigm.
- This module described how to refine the following Domain model features: attributes, relationships, methods, constructors, and method behavior (by using annotations).
- The classes should be reviewed to ensure that they maintain high cohesion and low coupling.
- Classes can be grouped into cohesive components with well-defined interfaces.



Module 14

Overview of Software Development Processes



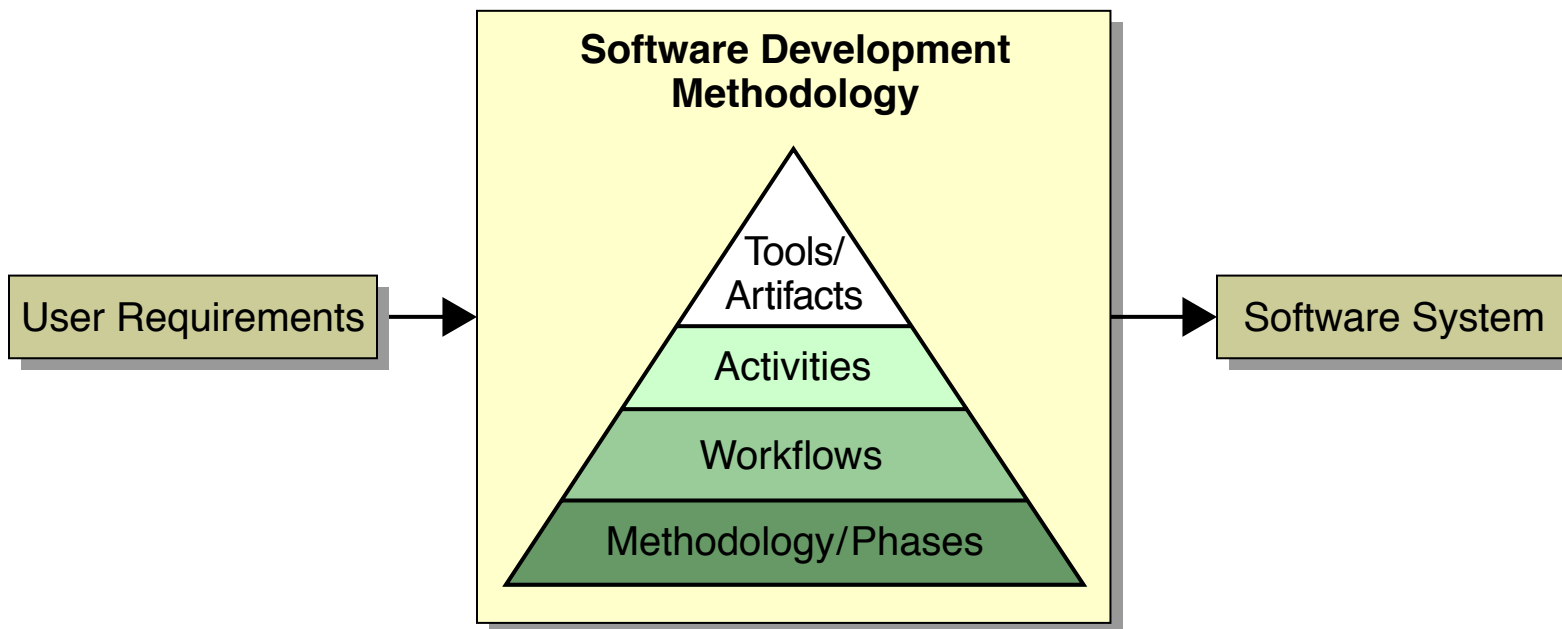
Objectives

Upon completion of this module, you should be able to:

- Explain the best practices for OOSD methodologies
- Describe the features of several common methodologies
- Choose a methodology that best suits your project
- Develop an iteration plan



Reviewing Software Methodology





Exploring Methodology Best Practices

- Use-case-driven
- Systemic-quality-driven
- Architecture-centric
- Iterative and incremental
- Model-based
- Design best practices



Use-Case-Driven

"A software system is brought into existence to serve its users."
(Jacobson USDP page 5)

- All software has users (human or machine).
- Users use software to perform activities or accomplish goals (use cases).
- A software development methodology supports the creation of software that facilitates use cases.
- Use cases drive the design of the system.



Systemic-Quality-Driven

- Systemic qualities are requirements on the system that are non-functional or related to the quality of service.
- Examples include:
 - Performance – Such as responsiveness and latency
 - Reliability – The mitigation of component failure
 - Scalability – The ability to support additional load, such as more users
- Systemic qualities drive the architecture of the software.



Architecture-Centric

“Architecture is all about capturing the strategic aspects of the high-level structure of a system.” (Arlow and Neustadt page 18)

Strategic aspects are:

- Systemic qualities drive the architectural components and patterns.
- Use cases must fit into the architecture.

High-level structure is:

- Tiers, such as client, application, and backend
- Tier components and their communication protocols
- Layers, such as application, platform, hardware



Iterative and Incremental

“Iterative development focuses on growing the system in small, incremental, and planned steps.” (Knoernschild page 77)

- Each iteration includes a complete OOSD life cycle, including analysis, design, implementation, and test.
- Models and software are built incrementally over multiple iterations.
- Maintenance is simply another iteration (or series of iterations).



Model-Based

Models are the primary means of communication between all stakeholders in the software project.

Types:

- Textual documents
- UML diagrams
- Prototypes

Purposes:

- Communication
- Problem solving
- Proof-of-concept



Design Best Practices

Understanding and applying design-level best practices can improve the flexibility and extensibility of a software solution.

These best practices include the following:

- Design principles
- Software patterns
- Refactoring
- Sun Blueprints

<http://www.sun.com/blueprints/>



Surveying Several Methodologies

This module describes the following methodologies:

- Waterfall
- Unified Software Development Process (USDP or just UP)
- Rational Unified Process (RUP)
- Scrum
- eXtreme Programming (XP)

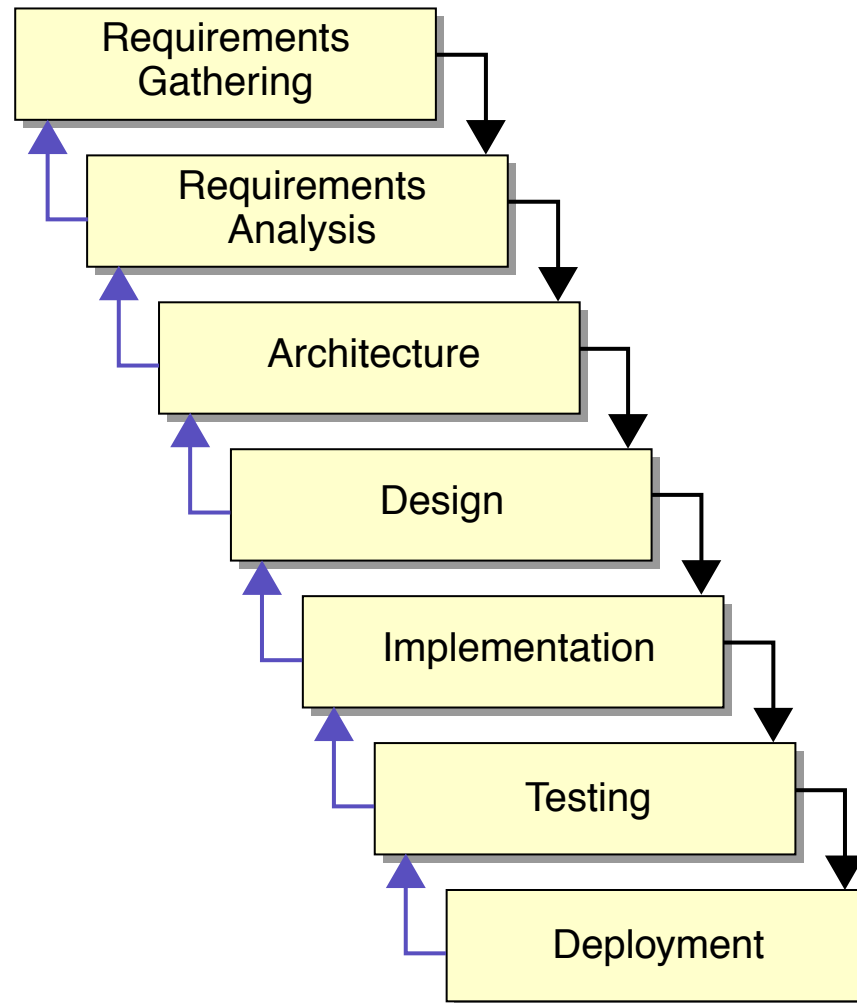


Waterfall

- Waterfall uses a single phase in which all workflows proceed in a linear fashion.
- This methodology does not support iterative development.
- This methodology works best for a project in which all requirements are known at the start of the project and requirements are not likely to change.
- Some government contracts might require this type of methodology.
- Some consulting firms use this methodology in which each workflow is contracted with a fixed-price bid.



Waterfall





Unified Software Development Process

The Unified Software Development Process (USDP) is the “open” version of Rational’s methodology created by Booch, Jacobson, and Rumbaugh. This is also called the Unified Process (UP).

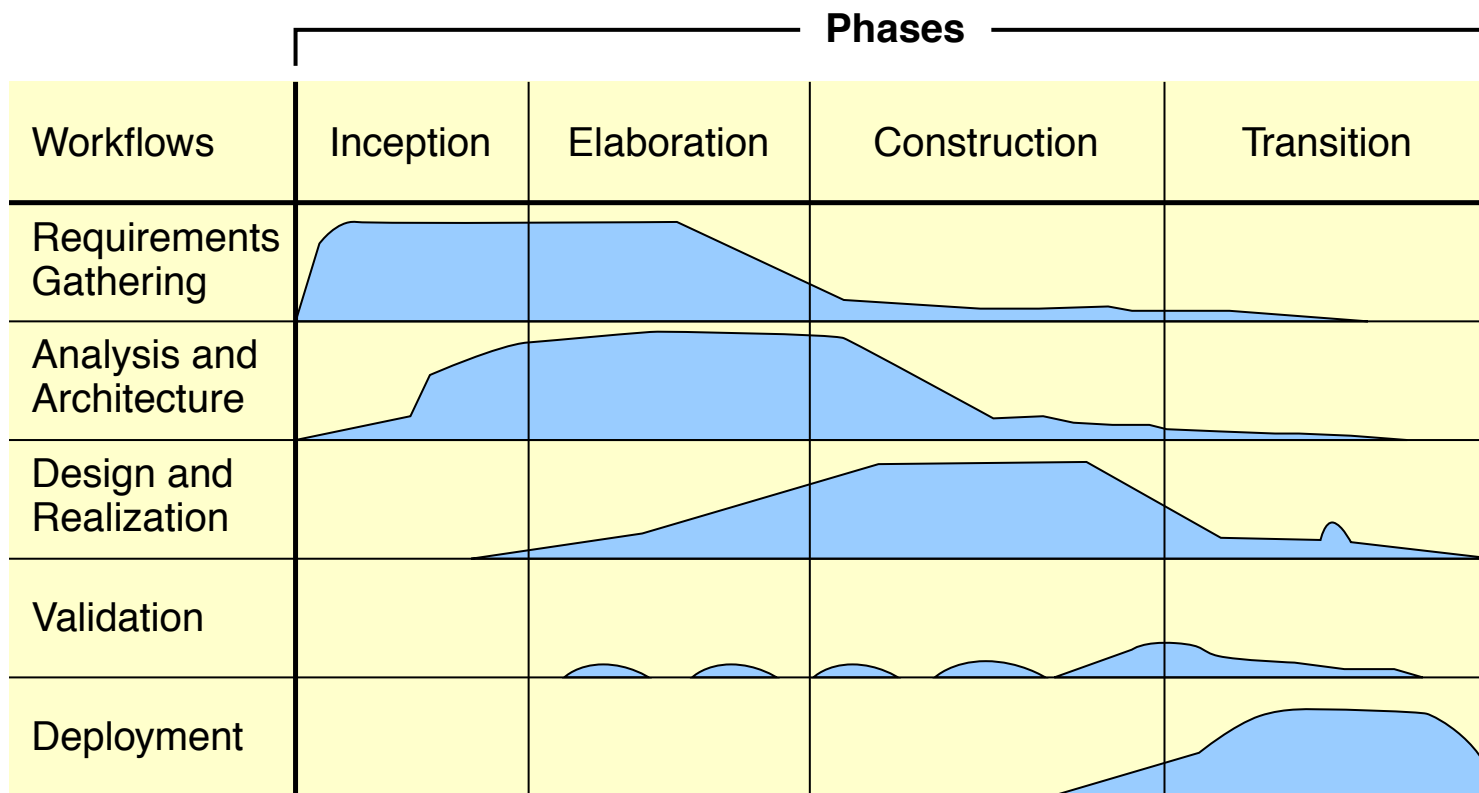
Four phases:

- Inception – Creates a vision of the software
- Elaboration – Most use cases are defined plus the system architecture
- Construction – The software is built
- Transition – Software moves from Beta to production

There can be multiple iterations per phase.



Unified Software Development Process





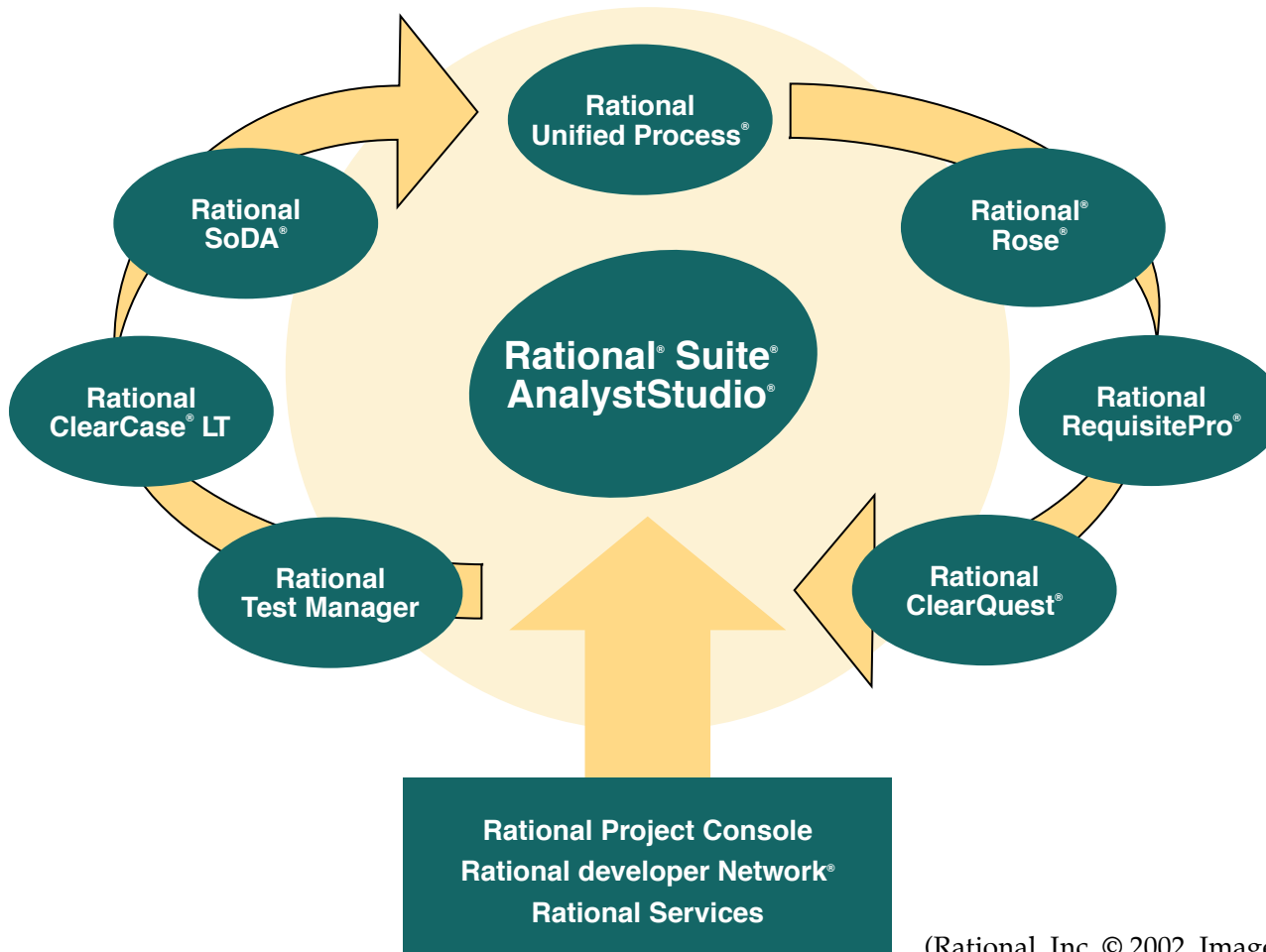
Rational Unified Process

RUP is the commercial version of the UP methodology created by Booch, Jacobson, and Rumbaugh.

- RUP is UP with the support of Rational's tool set.
- These tools manage the phases, workflows, and artifacts throughout the project life cycle.



Rational Unified Process



(Rational, Inc. © 2002. Image used with permission.)



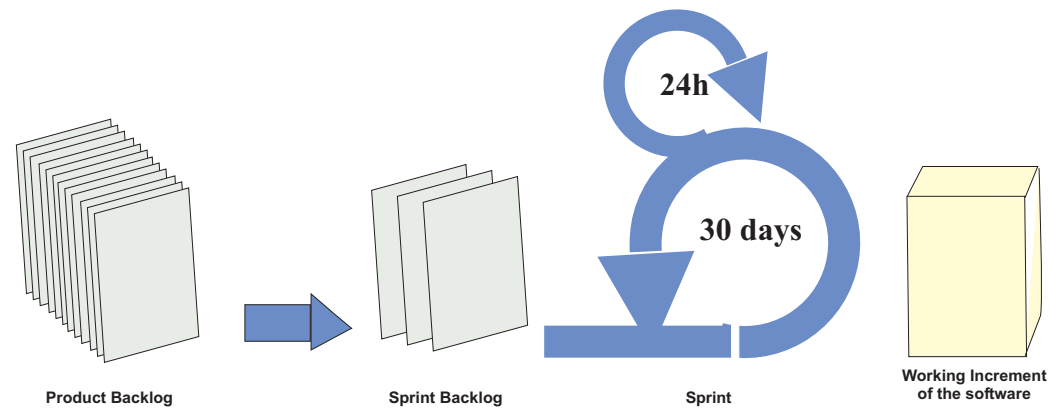
Scrum

Scrum is an iterative and incremental development framework that is often used in conjunction with agile software development. The key features are:

- Each Sprint produces a deliverable increment of the software.
- Each Sprint is typically 15 to 30 days.
- A subset of features are moved from the Product backlog to the Sprint backlog at the beginning of each Sprint.
- The requirements in the Sprint backlog are developed during the Sprint.
- Sprint progress is reviewed every 24 hours in a daily Scrum.
- The Scrum framework requires team-oriented responsibilities.



Scrum





eXtreme Programming

“XP nominates coding as the key activity throughout a software project.” (Erich Gamma, forward to Beck’s XP book, page xiii)

Here are a *few* key ideas:

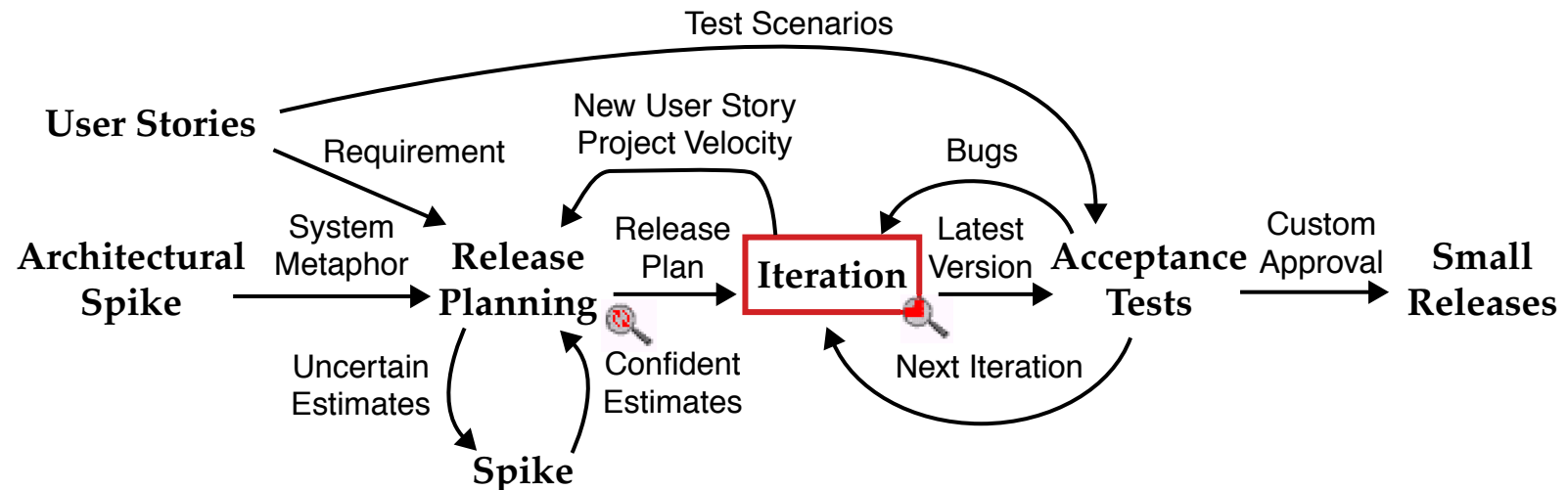
- Pair programming – If code reviews are good, then review code all the time.
- Testing – If testing is good, then test all the time, even the customers.
- Refactoring – If design is good, then make it part of everybody’s daily business.
- Simplicity – If simplicity is good, then always leave the system with the simplest design.



eXtreme Programming



Extreme Programing Project



(J. Donovan Wells © 2002. Image used with permission.)

Copyright 2000 J. Donovan Wells



Choosing a Methodology

There are a number of factors that guide in the choice of a methodology for a given project.

- Company culture – Process-oriented or product-oriented
- Make up of team – Less experienced developers might need more structure and people have distinct job roles
- Size of project – A larger project might need more documentation (communication between stakeholders)
- Stability of requirements – How often requirements change



Choosing Waterfall

When to use:

- Large teams with distinct roles
- Choose waterfall when the project is not risky

Issues:

- Not resilient to requirements changes
- Tends to be documentation heavy



Choosing UP

When to use:

- Company culture is process-oriented
- Teams with members that have flexible job roles
- Medium- to large-scale projects
- Requirements are allowed to change

Issues:

- Tends to be process and documentation heavy
- This is overkill for small projects



Choosing RUP

When to use:

- Same reasons as UP
- Your company owns Rational's tool set

Issues:

- Same issues as UP
- Tool set learning curve
- Tools lock the team into a process



Choosing Scrum

When to use:

- Priorities of the requirements are constantly changing
- When you need to deliver a working increment of the software every 30 days

Issues:

- Requires a committed team
- Primary focus is on the functional requirements for each Sprint. Non-functional requirements might be not be considered adequately.



Choosing XP

When to use:

- Company culture permits experimentation
- Small, close (proximity) teams with flexible work spaces
- Team must have as many experienced developers as inexperienced
- Requirements change frequently

Issues:

- Tends to be documentation light



Project Risks And Constraints

Project constraints and risks are often managed by the Project Architect.

Project constraints and risks should be assessed during the initial stages of the development process



Project Constraints

Project development constraints are often confused with NFRs (non-functional requirements), which are the runtime constraints of the system.

Typical project constraints are:

- Project must be developed on a specific platform
- Project requires specific technologies
- Project has a fixed deadline
- System interacts with specific external systems



Project Risks

Every project involves a level of risk.

Any situation or factor that could lead to an unsuccessful conclusion to a project defines a risk.

Each constraint usually affects several risks factors, some positively and some negatively.



Project Risks

The five main risk areas are:

- Political
- Technological
- Resources
- Skills
- Requirements



Producing an Iteration Plan

- Iteration plans are mainly applicable to iterative and incremental development processes.
- These plans determine the order in which the software components are developed.
- If you are using a use-case-driven methodology or a similar methodology, consider each use case based on priority, risk, and dependencies.
- The iteration plan might be documented in detail early in the project and rigidly followed or might be loosely planned and open for change as the project progresses.



Prioritizing Use Cases

Use cases, user stories, high-level FRs, and features are often categorized based on priority.

A common prioritization method is the MuSCoW prioritization technique, where priorities are defined as:

- **Must** have this
- **Should** have this, if at all possible
- **Could** have this, if it does not affect anything else
- **Won't** have this time, but **would** like to include in the future



Assessing Use Case Risk

Consider each use case based on risk. These risk factor considerations should include:

- Complexity of the use case
- Interfaces to external devices and system



Architectural Significance

Use cases are architecturally significant if they have project-critical NFRs.

These use cases require the architect to choose an architecture that satisfies those NFRs.



Architectural Baseline

An architectural baseline is a subset of the system requirements that are:

- Realized in code early in the project
- Tested to prove that the chosen architecture satisfies major NFRs

The baseline code includes architecturally significant use cases.

This process of testing these architecturally significant use cases is often called “Architectural Proof-of-Concept.”

In the Unified Process (UP), the architectural baseline must be completed at the end of the elaboration phase.



Timeboxing

Each iteration in an iterative and incremental development process has a fixed time duration.

Key features:

- The iteration completes on schedule.
- Unfinished use cases are rescheduled to the next iteration.

You should avoid continuous rescheduling of a specific use case (skidding).



80/20 Rule

In an iterative and incremental development process, you do not have to fully understand every aspect before moving to the next step.

The 80/20 rule can be applied. For example:

- For 20% of the effort, you can understand 80% of the detail.
- For 20% of the effort, your specification can achieve 80% accuracy.

Missing details and accuracy are found in the subsequent iterations for a fraction of the effort.

This rule works only if the software that you build is flexible.



Producing an Iteration Plan

Iteration plan contains the use cases that you intend to develop in each iteration.

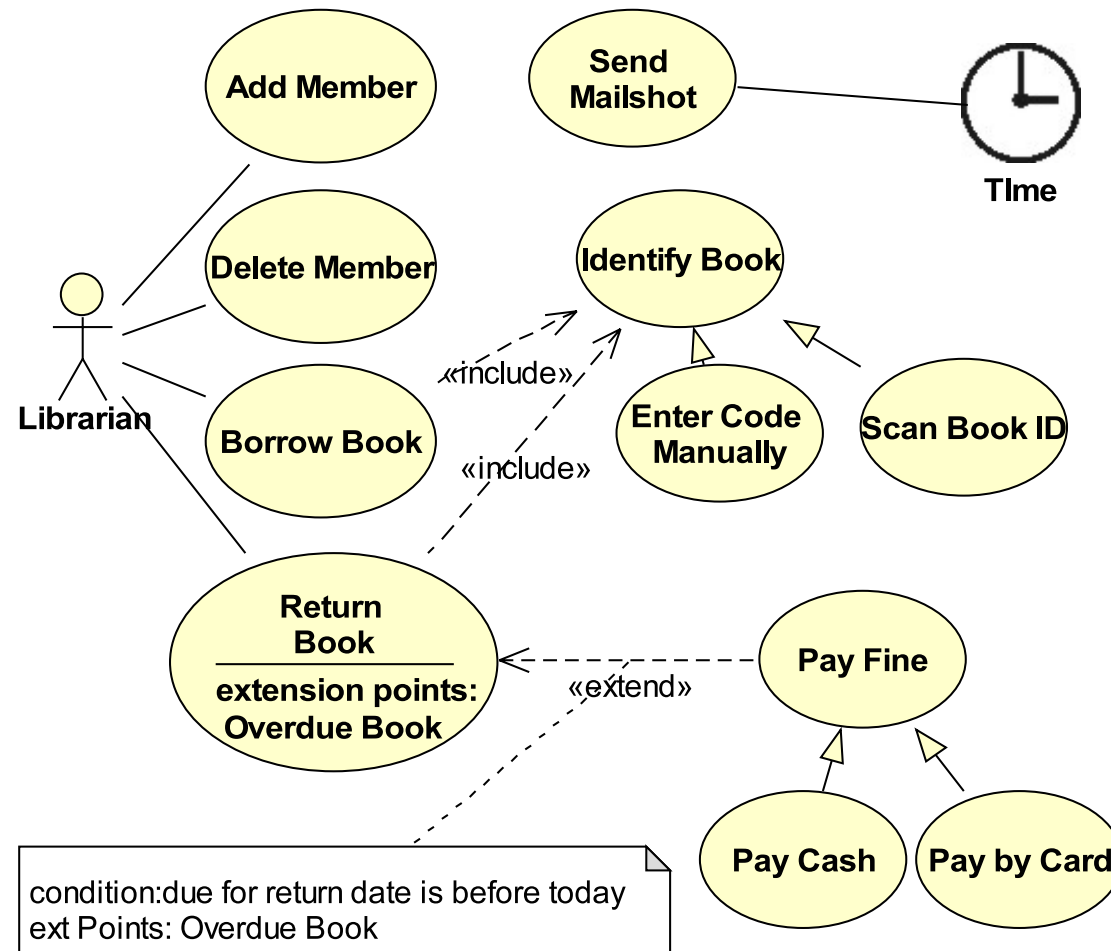
The criteria for assessment includes the following items:

- Use case priority
- Use case risk
- Architectural significance
- Estimated time to develop a use case
- Dependency on other use cases

The Use Case form and the Supplementary Specification Document contains this information.



Producing an Iteration Plan

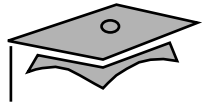




Producing an Iteration Plan

Example Assessment Criteria

Use Case Name	Priority	Risk	Architectural Significance	Dependency
Return Book	Must have	Medium	Must be complete within 60 seconds, and easy to use	Borrow Book
Pay Fine	Should have	Medium	None	Return Book
Pay Fine by Cash	Should have	Low	None	Pay Fine
Pay Fine by Card	Could have	High	Transaction with the external system must be complete in 30 seconds	Pay Fine



Producing an Iteration Plan

Borrow Book	Must have	Medium	Must be complete within 60 seconds, and easy to use	Add Book, Add Member, Identify Member
Identify Book	Must have	Medium	Must be complete within 15 seconds	Add Book
Enter Code Manually	Must have	Low	Must be complete within 15 seconds	Identify Book
Scan Book ID	Should have	Medium	Must be complete within 7 seconds	Identify Book
Delete Member	Should have	Low	None	Add Member
Send Mailshot	Could have	Low	None	Add Member



Example Iteration Plan

Here is a sample interpretation of an iteration plan.

Elaboration Iteration 2	Elaboration Iteration 3	Construction Iteration 4	Construction Iteration 5	Construction Iteration 6
Borrow Book	Return Book	Add Book	Add Member	Send Mailshot
Identify Book	Pay Fine by Card	Pay Fine by Cash	Modify Member	Delete Member
Enter Code Manually	Scan Book ID			
Simulation of Add Member	Pay Fine			
Simulation of Add Book				



Summary

In this module, we reviewed:

- Iterative and incremental development
- Architecture-centric development
- Project risks and constraints
- Developing an iteration plan based on a use case's:
 - Priority
 - Risk
 - Architectural significance
- No one methodology fits every organization or project. You can create your own methodology by adapting an existing methodology to suit your requirements and by following the best practices.



Module 15

Overview of Frameworks



Objectives

Upon completion of this module, you should be able to:

- Define a framework
- Describe the advantages and disadvantages of using frameworks
- Identify several common frameworks
- Understand the concepts of creating your own business domain framework



Description of Frameworks

A software framework is a re-usable software infrastructure that can be extended and configured to provide a specific software solution.

A software framework provides extension points in the framework where the application programmers may make additions or modifications for specific functional requirements.

This infrastructure can include components, APIs, scripts, support applications, configuration files.



Description of Frameworks

A framework can provide the infrastructure for:

- One or more tiers
 - For example, web presentation, business services, entities, and integration tiers.
- A specific business domain
 - For example, insurance, banking, or oil exploration
- A shared business domain requirement
 - For example, resource allocation, event management, or billing



Description of Frameworks

Customization of a framework is done by:

- Extending framework classes or implementing framework interfaces
 - Your classes may be less coherent
 - Your classes may be more difficult to test outside of the framework
- Informing the framework of the plain old Java (POJO) classes it must manage by using configuration files or annotations
 - POJOs tend to be more coherent
 - POJOs can be easier to test outside of the framework



List of common frameworks

- Ruby on Rails
- Spring framework
- Java Server Faces (JSF)
- Hibernate
- Struts
- Microsoft .NET
- Struts 2



Advantages and Disadvantages of Using Frameworks

Advantages include:

- Developers can focus on the new business problem and not the infrastructure problems, or common business aspects
- Frameworks usually include good OO practices and patterns
- Once you have gained experience with a framework, code is easier to write and support



Advantages and Disadvantages of Using Frameworks

Disadvantages include:

- Your code may become bloated due to the one-size-fits-all approach of the framework
- Frameworks may be difficult to learn
- You are restricted by the infrastructure and cannot usually modify the infrastructure files
- Changing to an alternative framework may be difficult



Building Frameworks

A generic framework can be built for a specific business domain

For example an insurance company could create a generic insurance framework that would support any insurance domain product. For example:

- Pet insurance
- Car insurance
- Life insurance
- Property insurance
- Public liability insurance



Building Frameworks

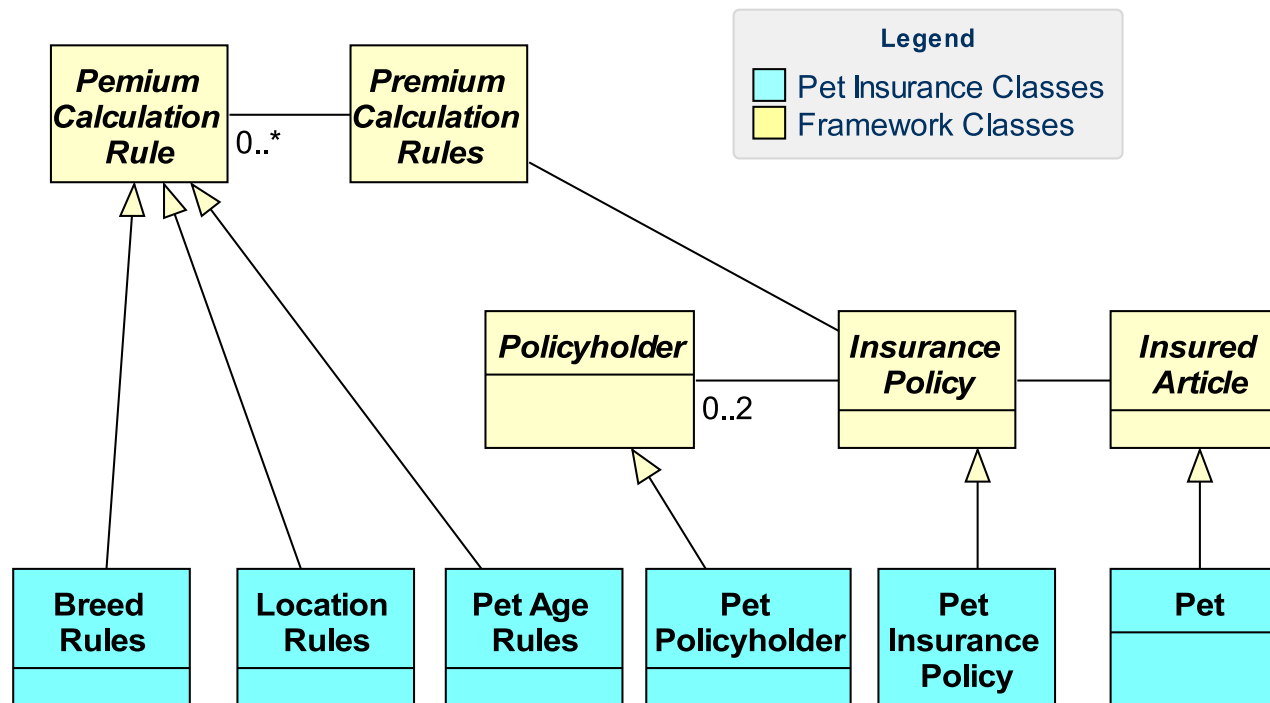
The following examples show two alternative approaches to building a generic framework for an insurance domain with a specialization domain of Pet insurance:

- Example 1: shows a framework based on abstract classes
- Example 2: shows a framework based on interfaces and abstract classes

There are other possible approaches to building frameworks

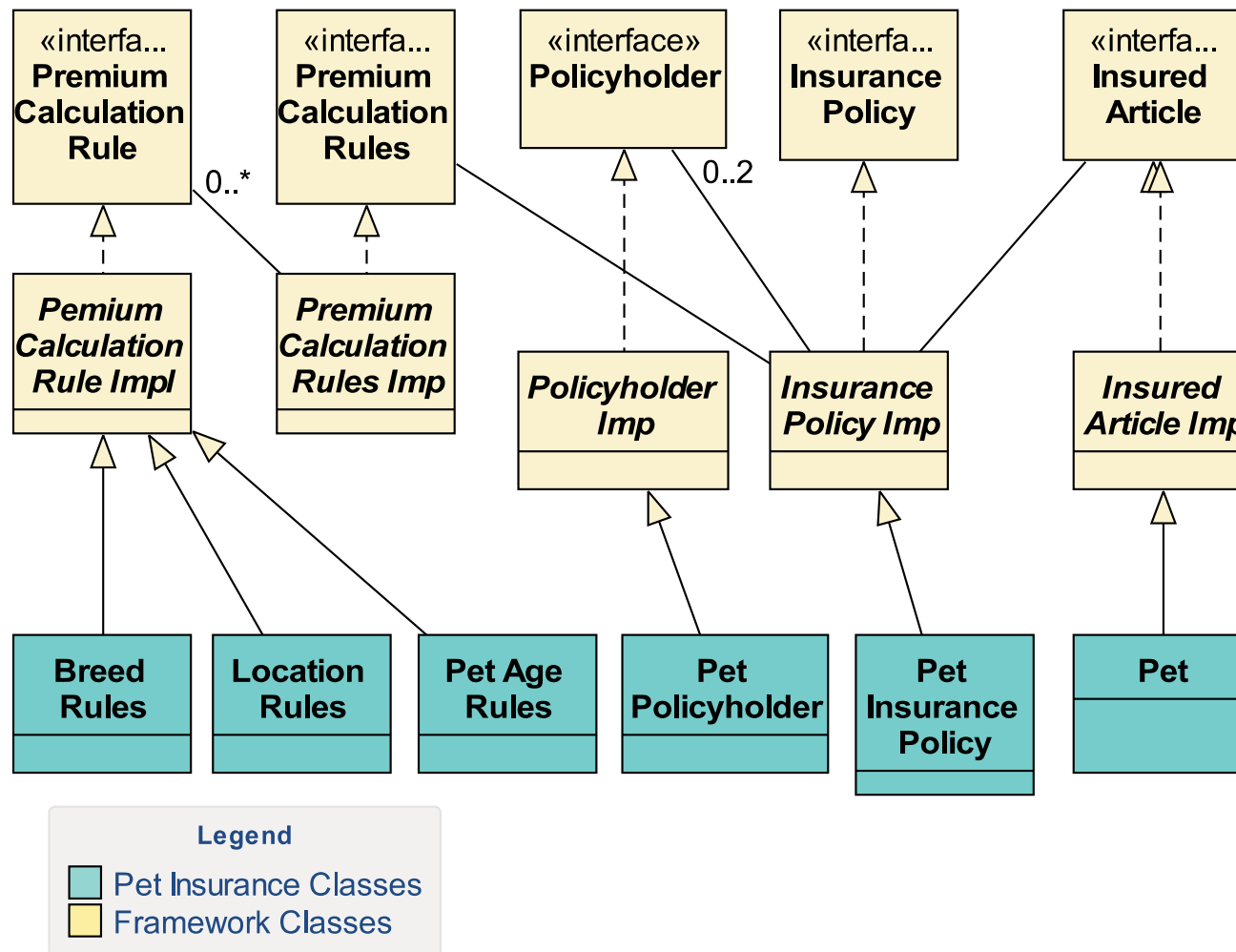


Domain Specific Framework Example 1





Domain Specific Frameworks Example 2





Domain Neutral Frameworks

- Domain neutral frameworks are often used for subsystems.
- These frameworks contain common features required by a variety of different domains. For example:
 - A used car sales system could use a trading framework and a billing framework
 - A human resource system could use a resource allocation framework



Domain Neutral Frameworks

It is possible to find more generic patterns, which are often called analysis patterns.

- For example one such pattern is Party (Person or Company), Place, Thing, Event. This pattern forms the basis of event planning or resource scheduling
- However these may be far too abstract and generic to be of benefit



Advantages and Disadvantages of Building Frameworks

The advantages of building a framework include:

- A reduction in cost and development time for each specific domain version using the framework
- May result in providing a competitive advantage
- Developers can focus on the differences between the specific domain and the framework
- Frameworks usually include good OO practices and patterns



Advantages and Disadvantages of Building Frameworks

The disadvantages of building a framework include:

- Can be expensive to build
- Requires an excellent knowledge of all the domains that the framework will be used for
- The code may become bloated due to the one-size-fits-all approach of a framework
- Frameworks may be difficult to learn
- You are restricted by the infrastructure and cannot usually modify the infrastructure files



Summary

In this module, you were introduced to the basic concepts of:

- Frameworks
- Using existing frameworks
- Creating domain neutral and domain specific frameworks



Module 16

Course Review



Objectives

Upon completion of this module, you should be able to:

- Review the key features of object orientation
- Review the key UML diagrams
- Review the Requirements Analysis (Analysis) and Design workflows



Overview

In this module, your instructor will facilitate an interactive review in which you will recall and review the key topics covered in the course.

This is your opportunity to review the key topics covered in the course.

This course covered three main aspects that ran concurrently throughout the course:

- Object orientation
- UML diagrams
- The development process, focusing on the Analysis and Design workflows



Reviewing Object Orientation

In this instructor-facilitated discussion, you will recall and review the key object-oriented (OO) concepts and terminology covered in the course.



Suggested OO Topics for Discussion

- Object
- Class
- Abstraction
- Encapsulation
- Inheritance
- Abstract class
- Interface
- Polymorphism
- Cohesion
- Coupling
- Class associations and object links
- Delegation
- Design pattern



Reviewing UML Diagrams

In this instructor-facilitated discussion, you will recall and review the key UML diagrams and their purposes.



Suggested UML Topics for Discussion

- Use Case diagram
- Class diagram
- Object diagram
- Communication diagram (formerly Collaboration diagram)
- Sequence diagram
- Activity diagram
- State Machine diagram
- Component diagram
- Deployment diagram
- Package diagram



Suggested UML Topics for Discussion

The following diagrams were not formally covered in the course, but might have been discussed in the review:

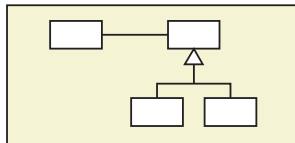
- Interaction Overview diagram
- Timing diagram
- Composite Structure diagram
- Profile diagram



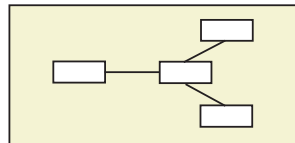
UML Diagrams: A Recap

Structural

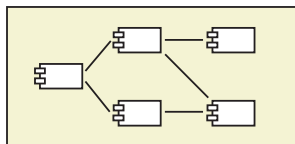
Class



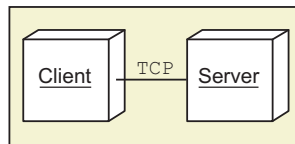
Object



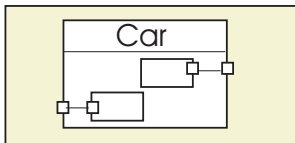
Component



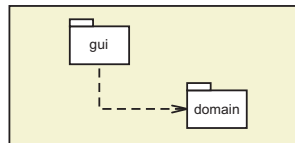
Deployment



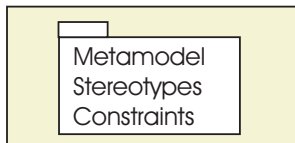
Composite Structure



Package

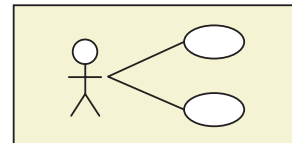


Profile

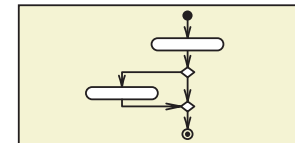


Behavioral

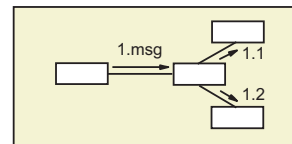
Use Case



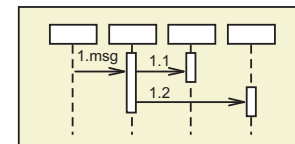
Activity



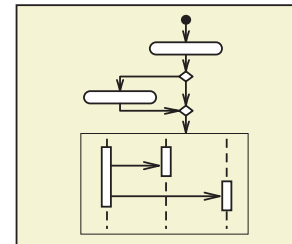
Communication



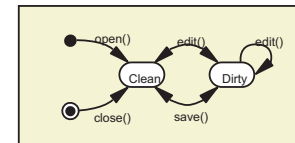
Sequence



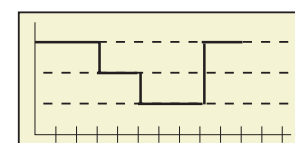
Interaction Overview



State Machine



Timing





Reviewing the Development Process

In this instructor-facilitated discussion, you will first recall and review the key workflows of the entire development process. Then, you will focus on reviewing the key activities and artifacts of the Analysis and Design workflows.

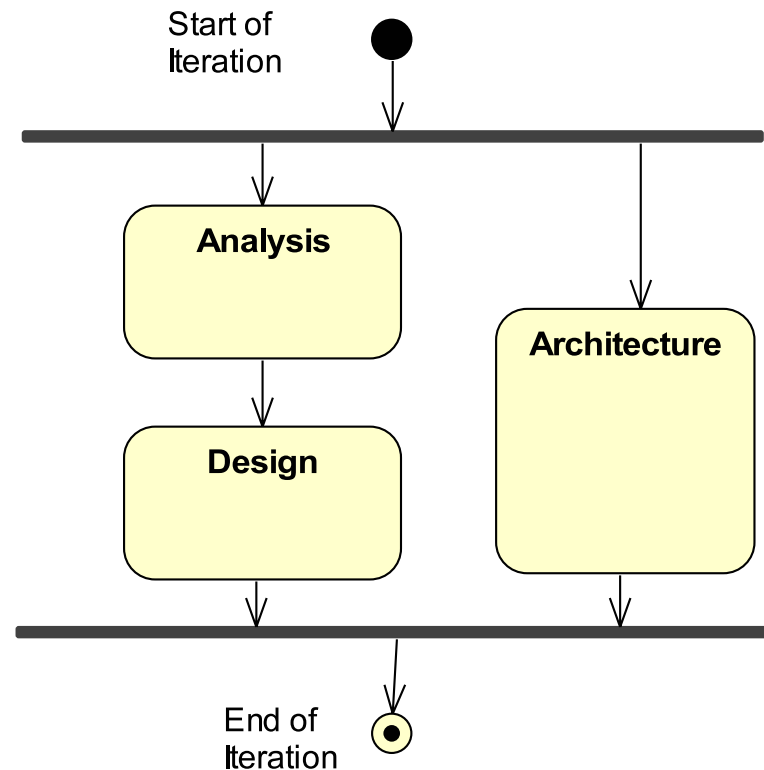


Suggested Development Process Topics for Discussion

- Requirements Gathering
- Requirements Analysis (or just Analysis)
- Architecture
- Design
- Implementation
- Testing
- Deployment

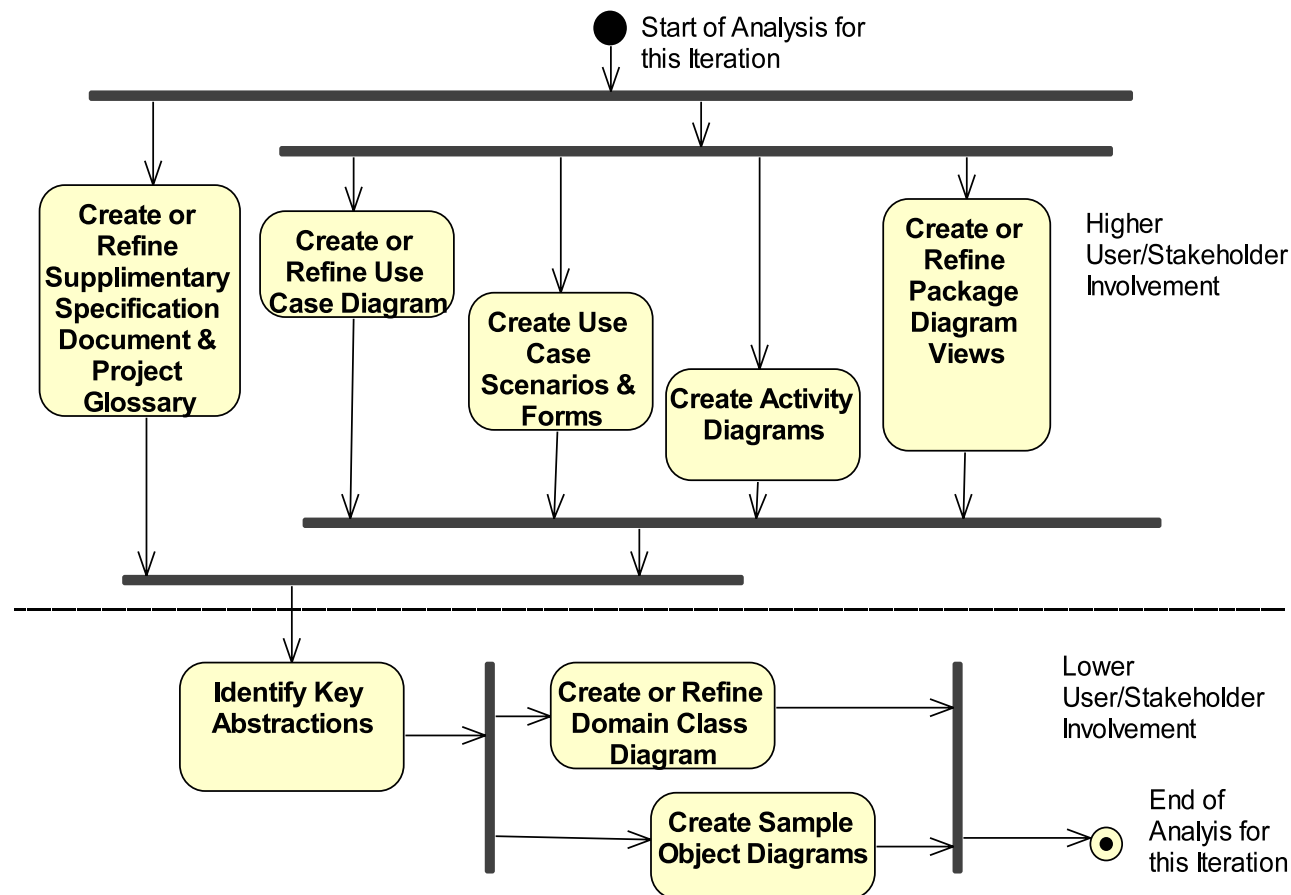


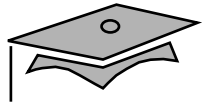
Suggested Discussion on the Analysis and Design Workflows



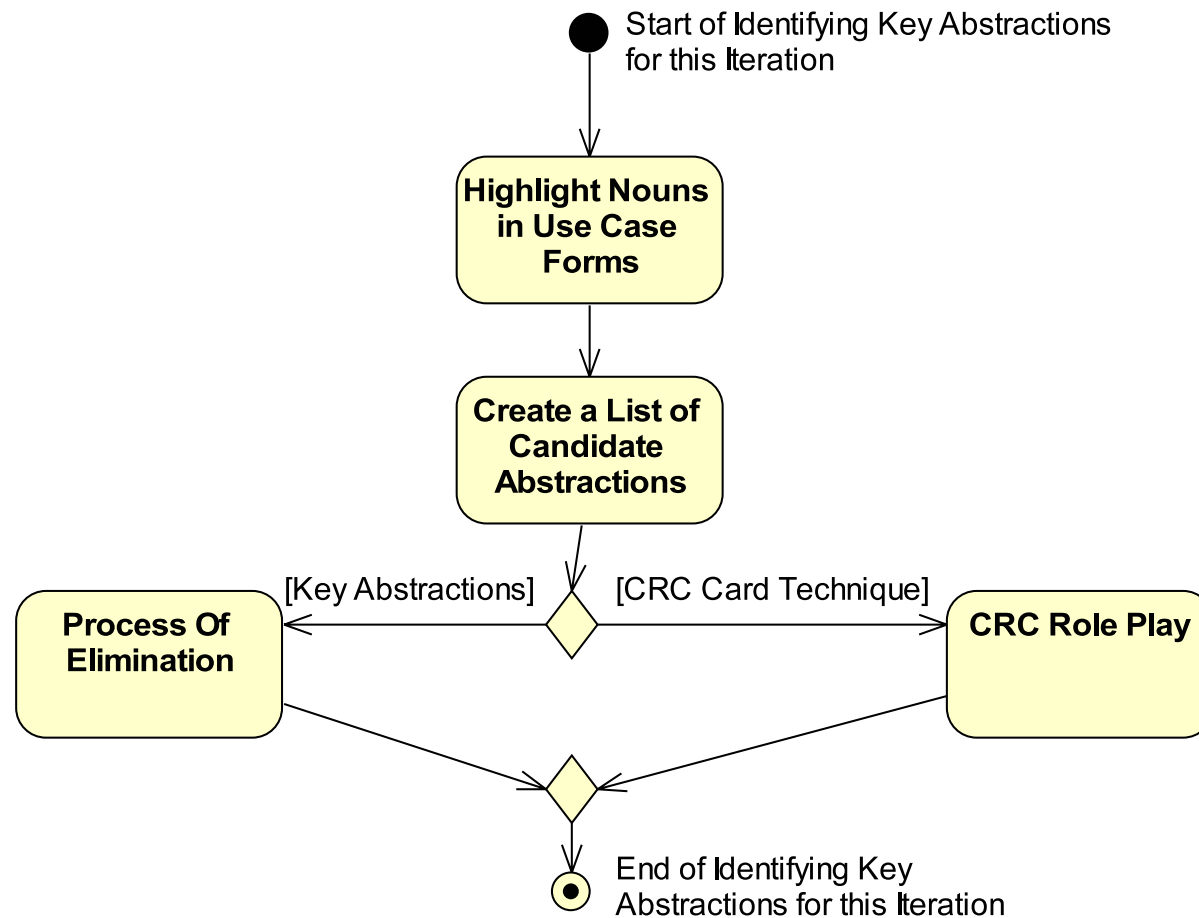


Suggested Discussion on the Analysis Workflow



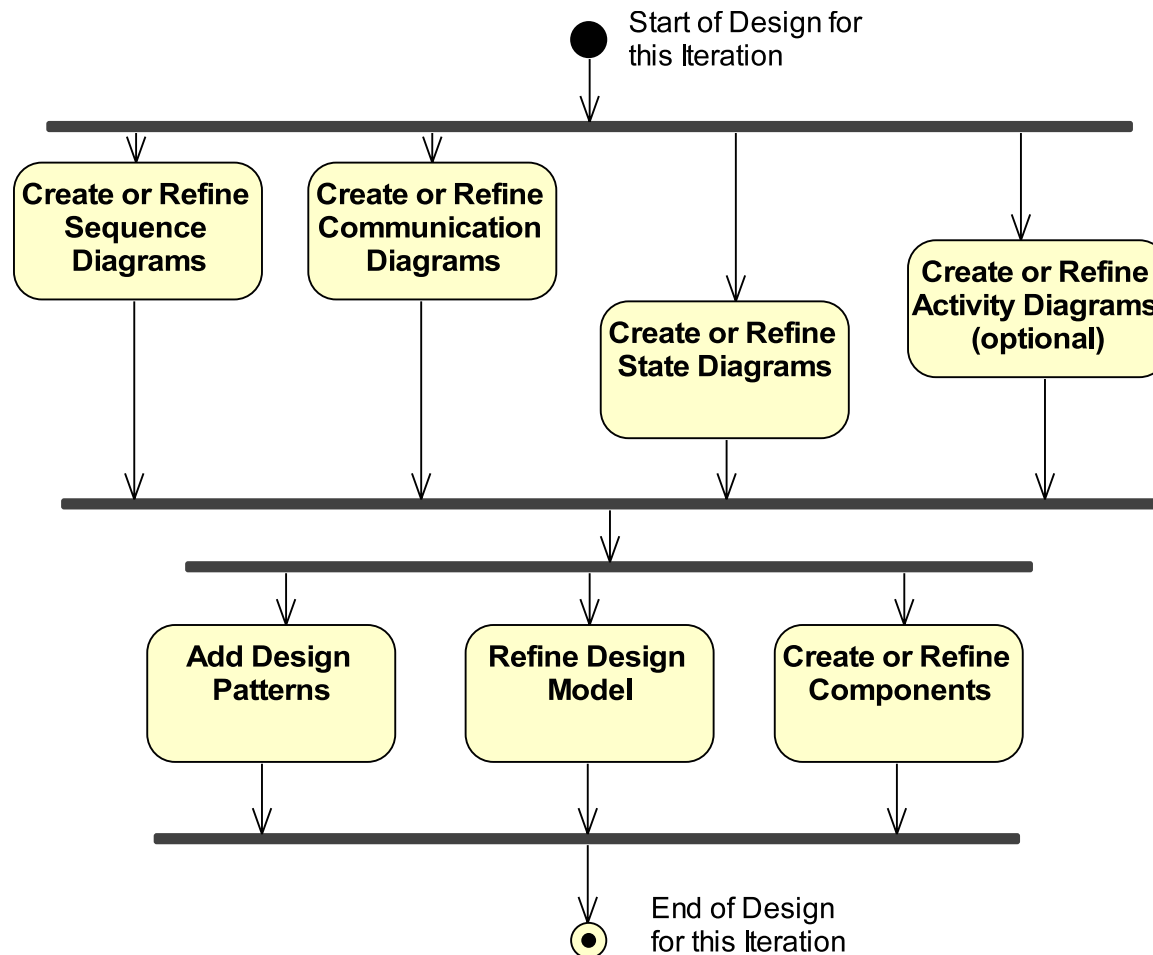


Suggested Discussion on Identifying Key Abstractions





Suggested Discussion on the Design Workflow





Summary

In this module, you reviewed the three main aspects that ran concurrently throughout the course:

- Object orientation
- UML diagrams
- The development process, focusing on the Analysis and Design workflows