Object-Oriented Analysis and Design Using UML

OO-226

Sun microsystems

Preface

About This Course

Course Goals

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

- Describe the Object-Oriented Software Development process (OOSD)
- Determine the system requirements
- Analyze the functional requirements
- Create a system architecture
- Design the system solution

Course Map

Describing the OOSD Process

- Introducing the Software Development Process
- Examining Object-Oriented Technology
- Choosing an Object-Oriented Methodology

Determining System Requirements

- Determining the Project Vision
- Gathering the System Requirements
- Creating the Initial Use Case Diagram

Analyzing the Functional Requirements

- Refining the Use Case Diagram
- Determining the Key Abstractions
- Constructing the Problem Domain Model
- Creating the Design Model Using Robustness Analysis

Creating a System Architecture

- Introducing Fundamental Architectural Concepts
- Exploring the Architecture Workflow
- Creating an Architectural Model for the Client and Presentation Tiers
- Creating an Architectural Model for the Business Tier
- Creating an Architectural Model for the Resource and Integration Tiers

Course Map

Designing the System Solutions

- Creating the Solution Model
- Refining the Domain Model
- Applying the Design Patterns to the Solution Model
- Modeling Complex Object State Using Statechart Diagrams

Construct, Test, and Deploy the System Solution*

- Drafting the Development Plan
- Constructing the Software Solution
- Testing the Software Solution
- Deploying the Software Solution

*These modules are appendices.

Topics Not Covered

- Fundamental Java technology – Covered in SL-275: *Java™ Programming Language*
- Enterprise edition Java technology – Covered in FJ-310: *Developing J2EE™ Compliant Applications*

How Prepared Are You?

To be sure you are prepared to take this course, can you answer yes to the following questions?

- Do you have a general understanding of a programming language?
- Do you have an understanding of the fundamentals of the software system development process?

Introductions

- Name
- Company affiliation
- Title, function, and job responsibility
- Experience related to requirements gathering and analysis
- Experience related to software architecture and design
- Experience related to modeling notations, such as OMT or UML
- Reasons for enrolling in this course
- Expectations for this course

How to Use the Icons

Additional resources

Discussion

Note

Typographical Conventions and Symbols

- `Courier` is used for the names of commands, files, directories, programming code, programming constructs, and on-screen computer output.
- **`Courier bold`** is used for characters and numbers that you type, and for each line of programming code that is referenced in a textual description.
- *`Courier italics`* is used for variables and command-line placeholders that are replaced with a real name or value.
- ***`Courier italics bold`*** is used to represent variables whose values are to be entered by the student as part of an activity.

Typographical Conventions and Symbols

- *Palatino italics* is used for book titles, new words or terms, or words that are emphasized.

Additional Conventions

Java™ programming language examples use the following additional conventions:

- `Courier` is used for the class names, methods, and keywords.
- Methods are not followed by parentheses unless a formal or actual parameter list is shown.
- Line breaks occur where there are separations, conjunctions, or white space in the code.
- If a command on the Solaris™ Operating Environment is different from the Microsoft Windows platform, both commands are shown.

UML Standard Stereotypes

Standard	Non-standard	Non-standard
«actor»	«operations»	«serial port»
«create»	«constructors»	«inner»
«entity»	«accessors»	«Class»
«extend»	«mutators»	«listener»
«import»	«UI Frame»	«methods»
«include»	«RMI Stub»	«Controller»
«instanceOf»	«RMI Skel»	«View»
«interface»	«RMI Impl»	«Service»
«refine»	«JRMP»	«Entity»
«table»	«JavaBean»	«data fields»
	«TCP/IP»	«primary key»
	«HTTP»	«VPN»

Module 1

Introducing the Software Development Process

Objectives

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

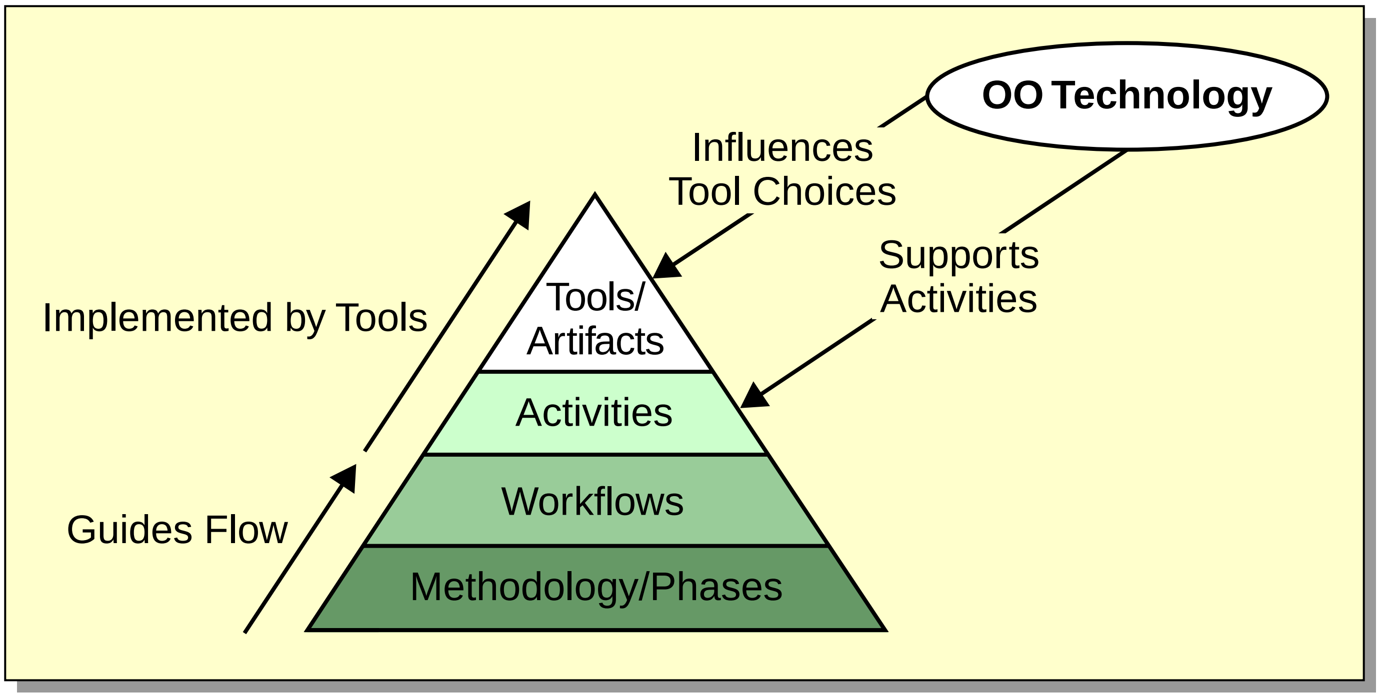
- Describe the Object-Oriented Software Development (OOSD) process
- Describe how modeling supports the OOSD process
- Explain the purpose, activities, and artifacts of the following OOSD workflows: Requirements Gathering, Requirements Analysis, Architecture, Design, Implementation, Test, Deployment

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. 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 artifact: the system requirements.

The OOSD Hierarchy



Listing the Workflows of the OOSD Process

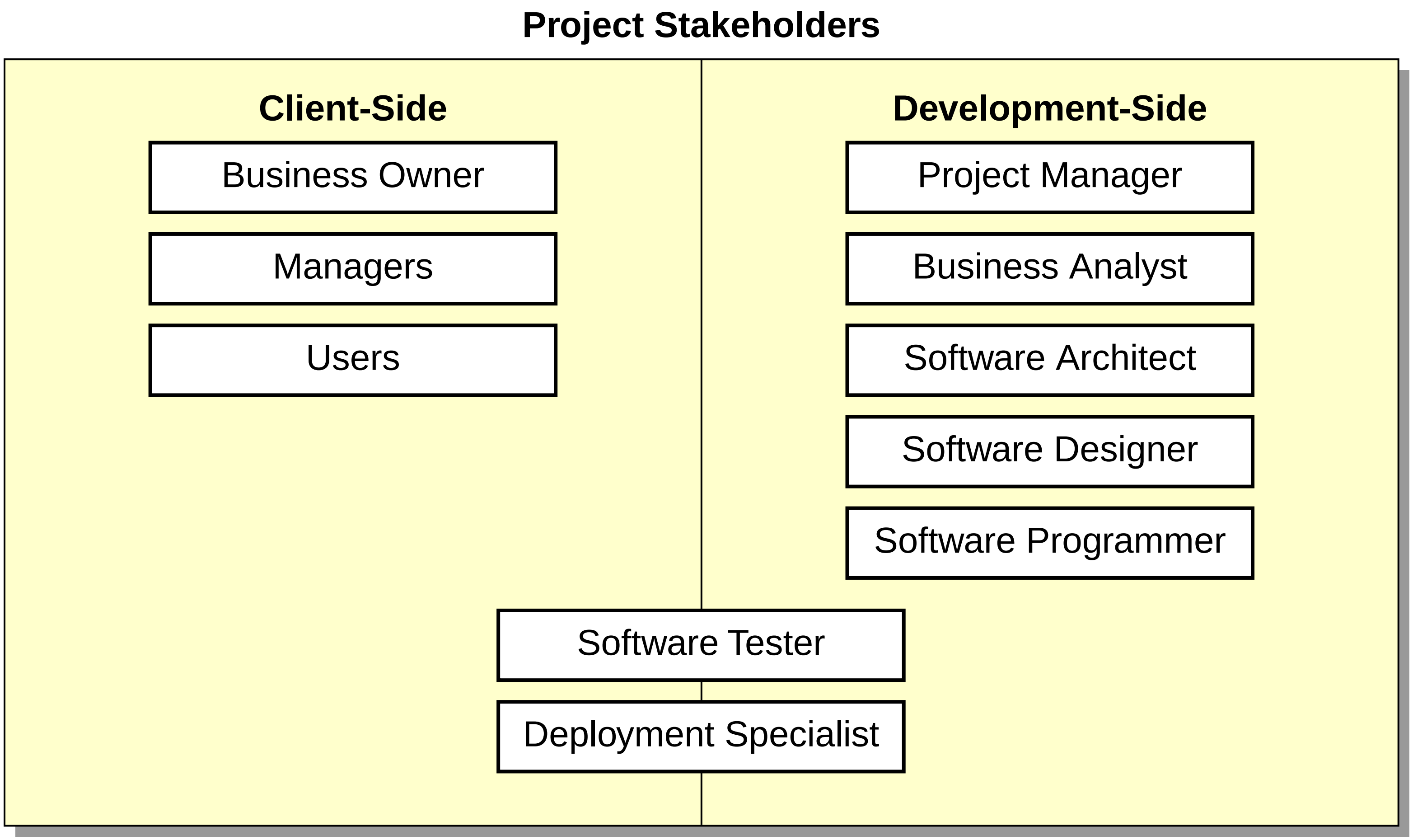
Software development has traditionally encompassed the following workflows:

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

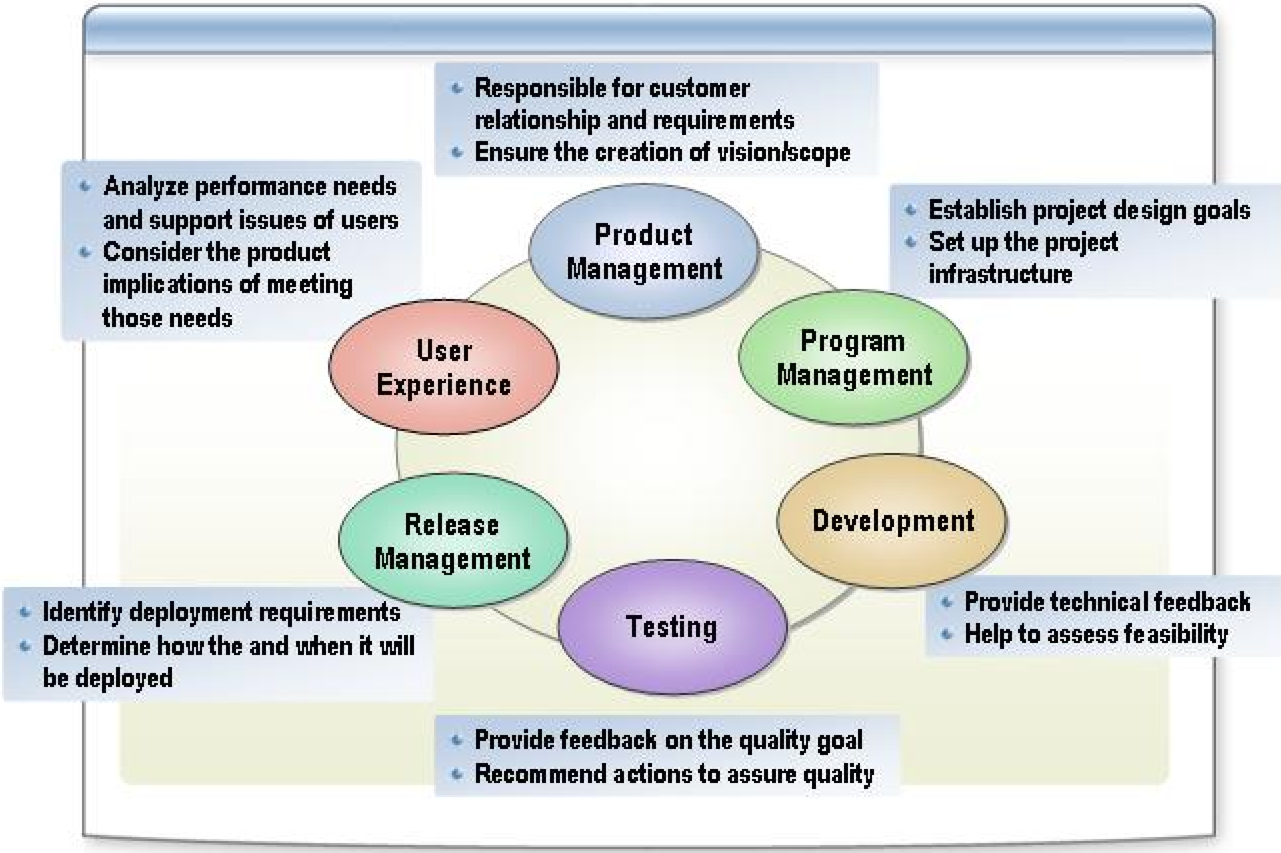
Comparing the Procedural and OO Paradigms

	Procedural Paradigm	***OO Paradigm***
Organizational structure	Hierarchy of tasks and subtasks	Network of collaborating objects
Impact on: • *Ability to modify software* • *Reuse* • *Configuration of special cases* • *Functional separation*	• "Brittle" software that is difficult to change • Reuse of methods leads to copy-and-paste or 1001 parameters • Often leads to nested `if` or `switch` statements • Testing is mostly built into the code, hard to separate functionality	• Robust software that is easy to change • Reuse of code through extension of classes and instantiation of objects • Polymorphic behavior facilitates configuration • Modular code helps separate functional blocks

Describing the Software Team Job Roles



Roles and Responsibilities of the Team Members



Combining Roles Within a Team

Role	Product management	Program management	Development	Testing	User experience	Release management
Product management		N	N	P	P	U
Program management	N		N	U	U	P
Development	N	N		N	N	N
Testing	P	U	N		P	P
User experience	P	U	N	P		U
Release management	U	P	N	P	U	

Legend: P: Possible
U: Unlikely
N: Not recommended

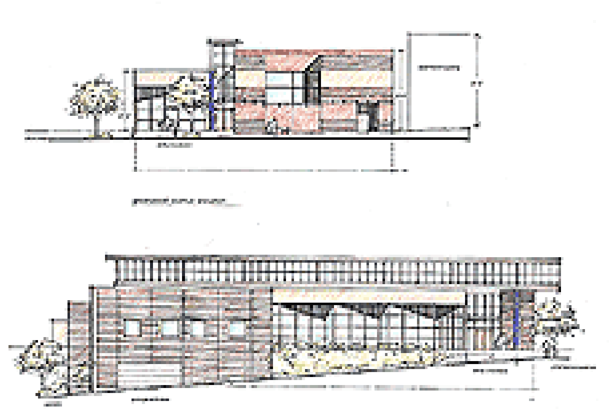
Exploring the Benefits of Modeling Software

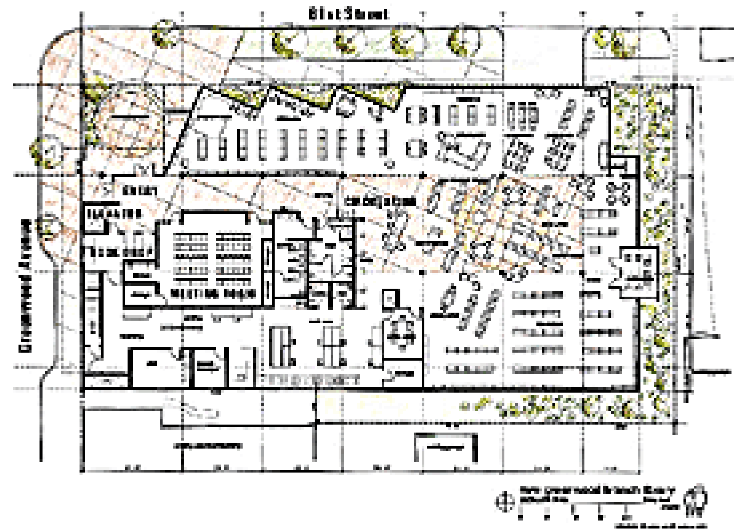
The inception of every software project starts as an idea in someone's mind.

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?

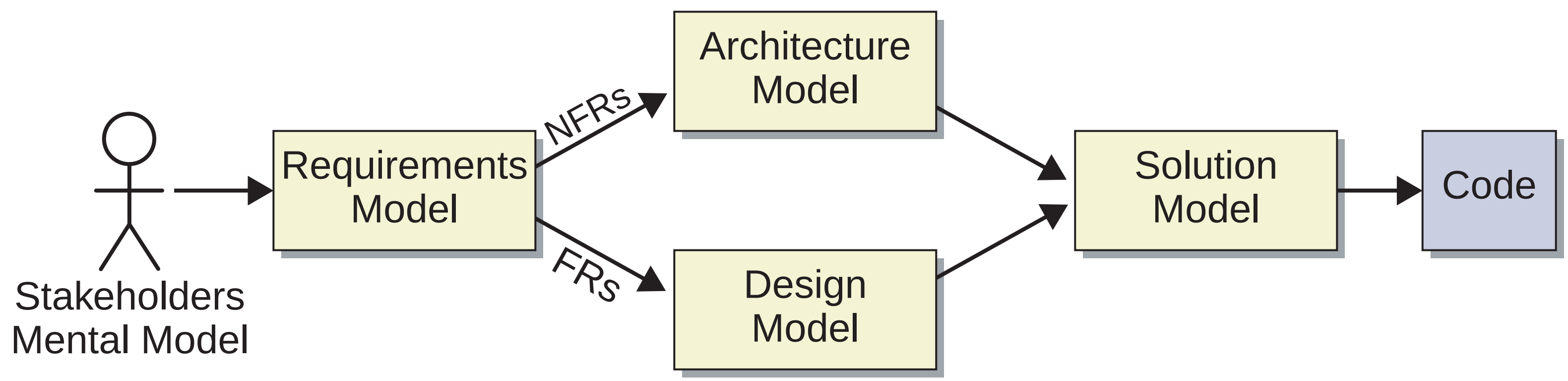
"We build models so that we can better understand the system we are developing." (Booch UML User Guide page 6)

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

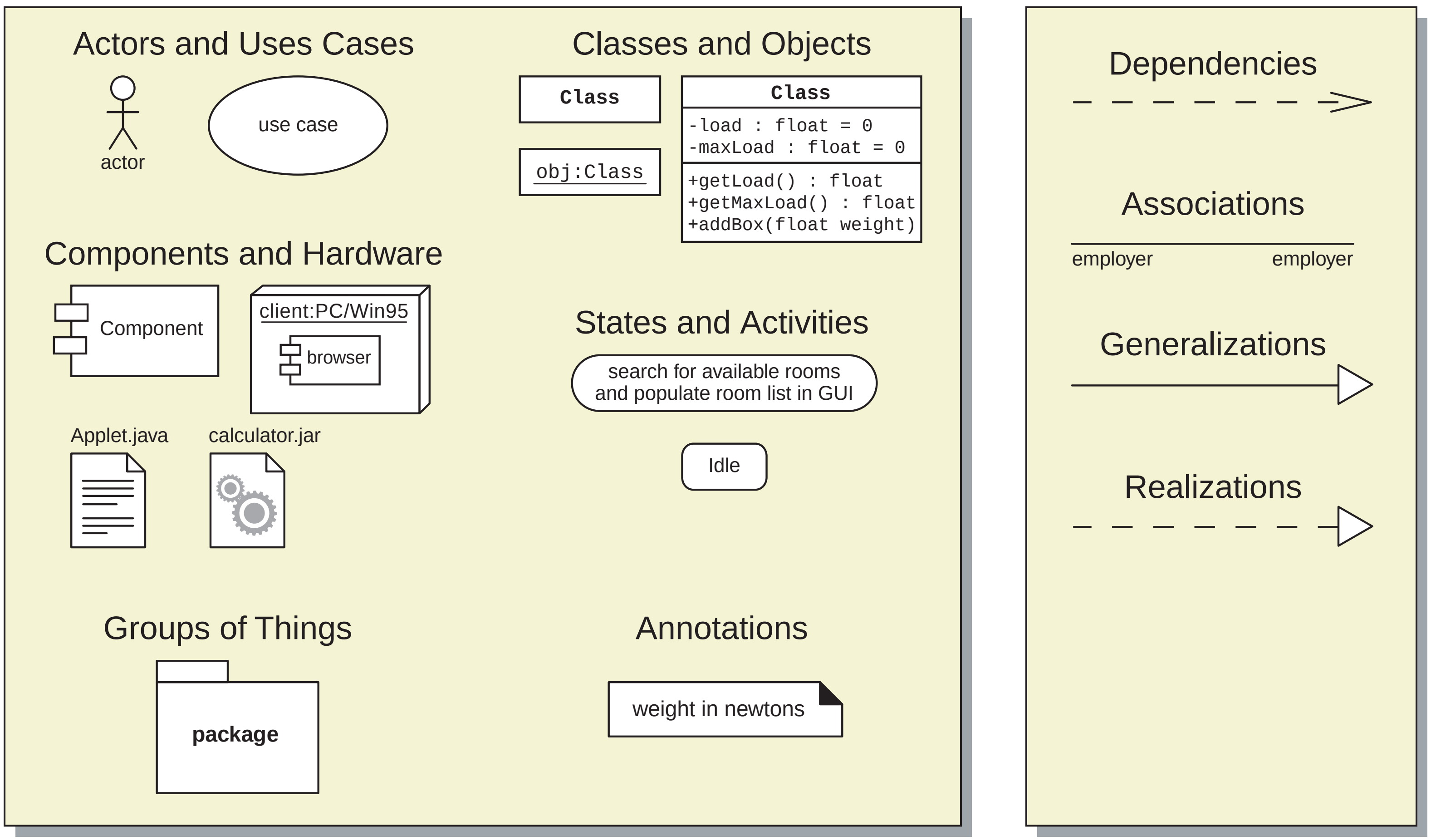
"The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system." (UML v1.4 page xix)

Using the UML, a model is composed of:

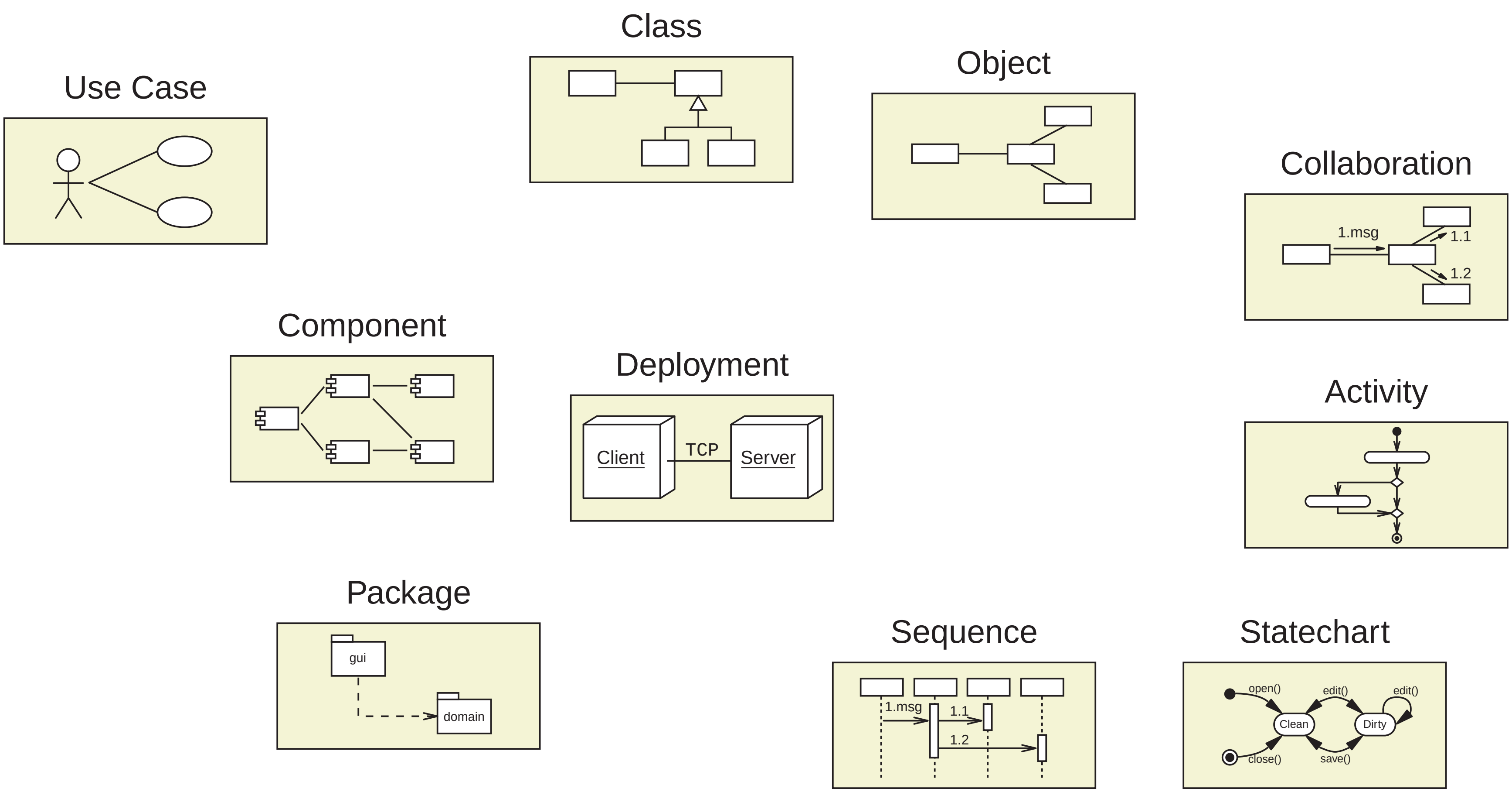
- Elements (things and relationships)
- Diagrams (built from elements)
- Views (diagrams showing different perspectives of a model)

UML Elements

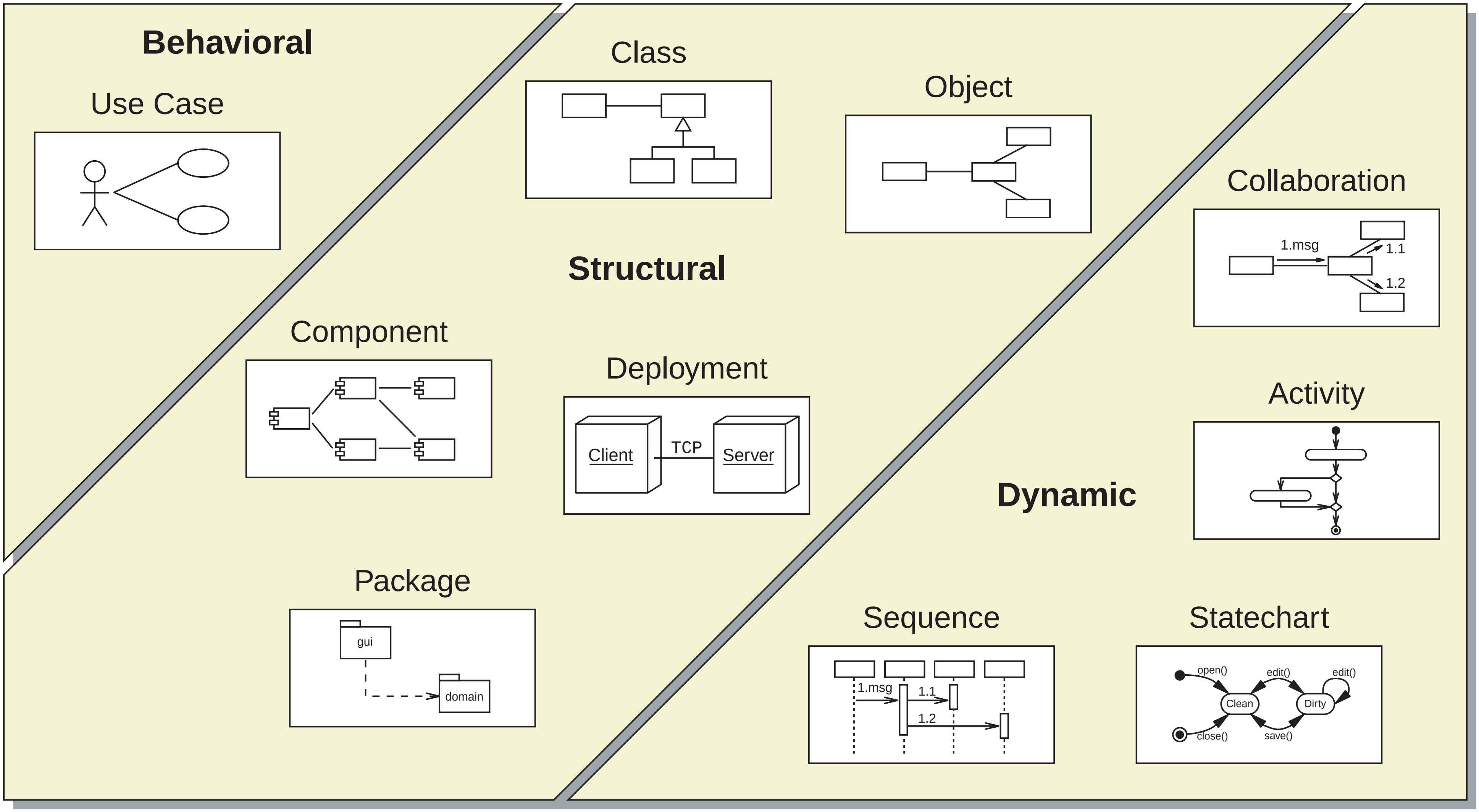
Things **Relationships**



UML Diagrams



Views


Behavioral
Use Case
Class
Object
Structural
Collaboration
1.msg
1.1
1.2
Component
Deployment
Client
TCP
Server
Activity
Dynamic
Package
gui
domain
Sequence
1.msg
1.1
1.2
Statechart
open()
edit()
edit()
Clean
Dirty
close()
save()

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

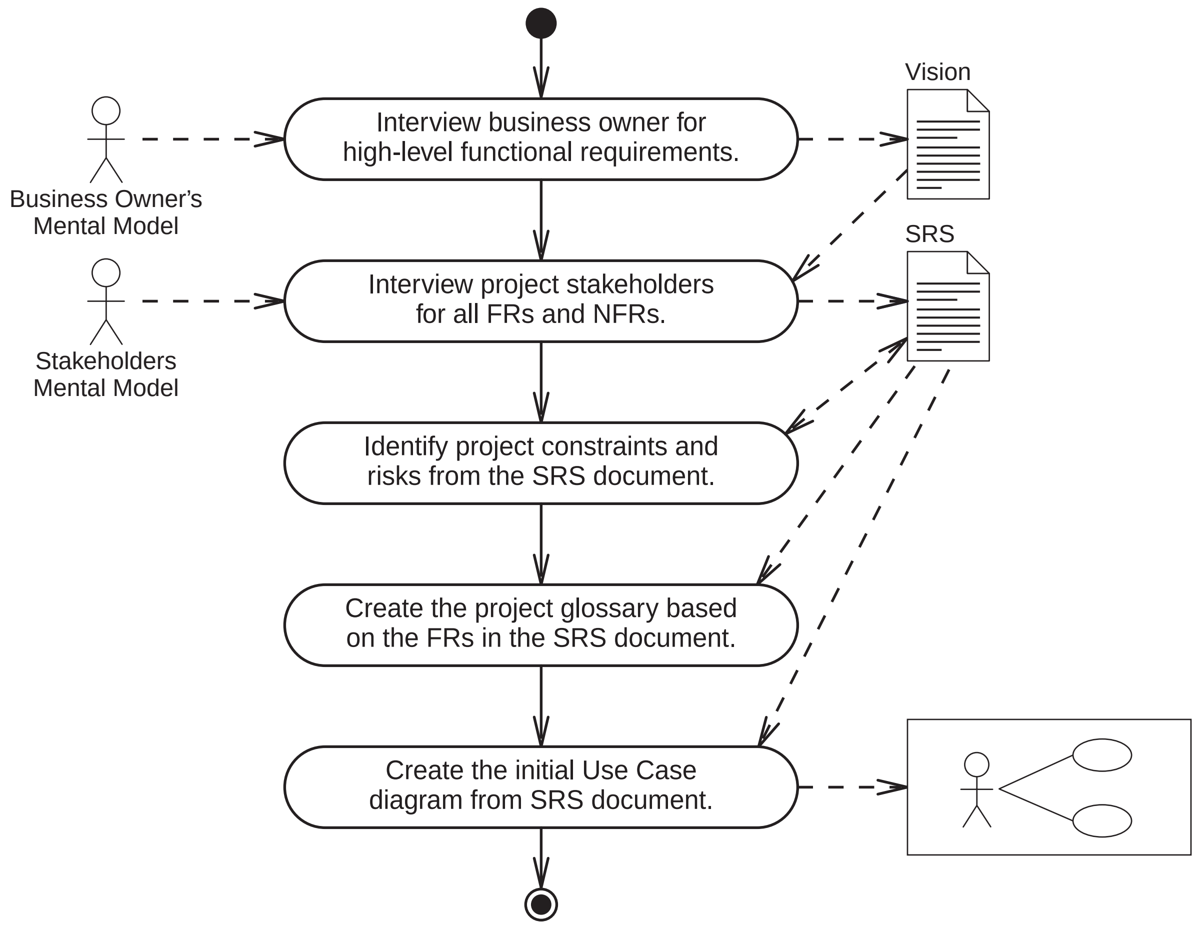
UML itself is a tool. You can create UML diagrams on paper or a white board. However, software tools are available to:

- 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™ 2 Platform, Enterprise Edition (J2EE™ platform) components, and so on

Exploring the Requirements Gathering Workflow

Workflow	*Purpose*	*Description*
Requirements Gathering	Determine what the system must do	Determine: • With whom the system interacts (actor) • What behaviors (called use cases) that the system must support • Detailed functional requirements for each use case • Non-functional requirements

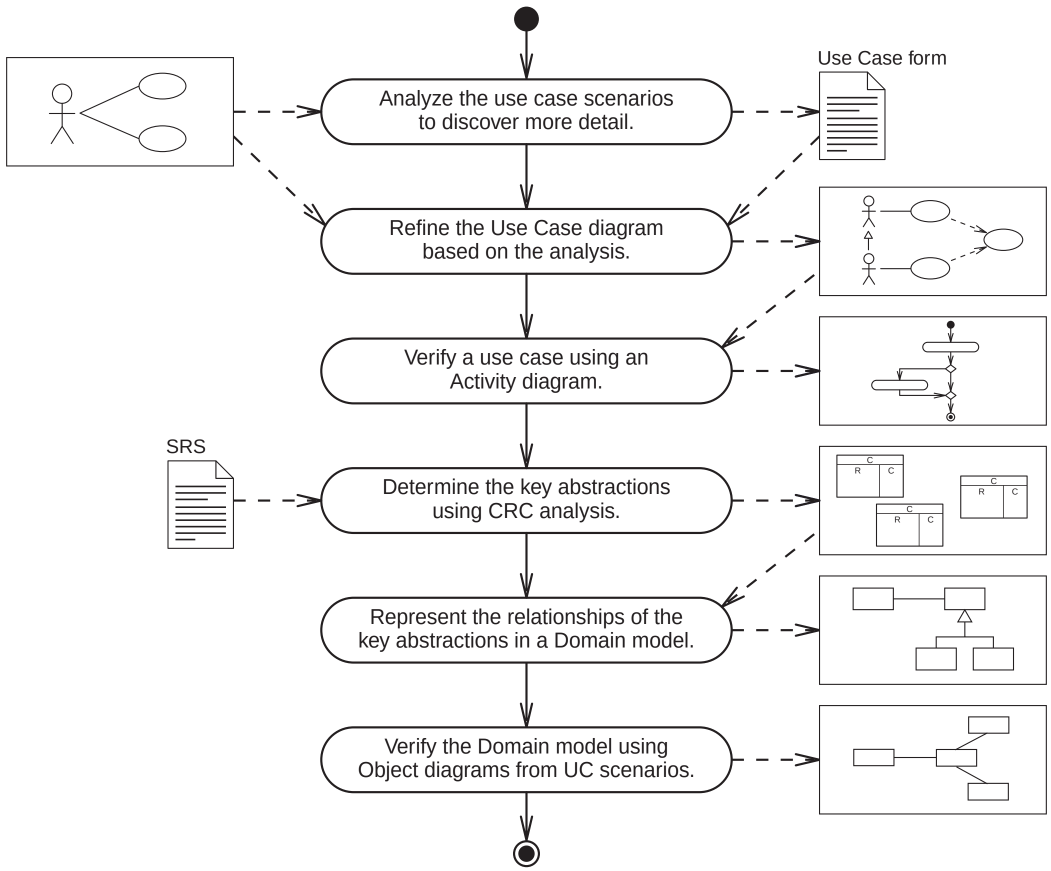
Activities and Artifacts of the Requirements Gathering Workflow



Exploring the Requirements Analysis Workflow

Workflow	*Purpose*	*Description*
Requirements Gathering	Determine what the system must do	
Requirements Analysis	Model the existing business processes	Determine: • A refined Use Case diagram • What key abstractions exist in the system

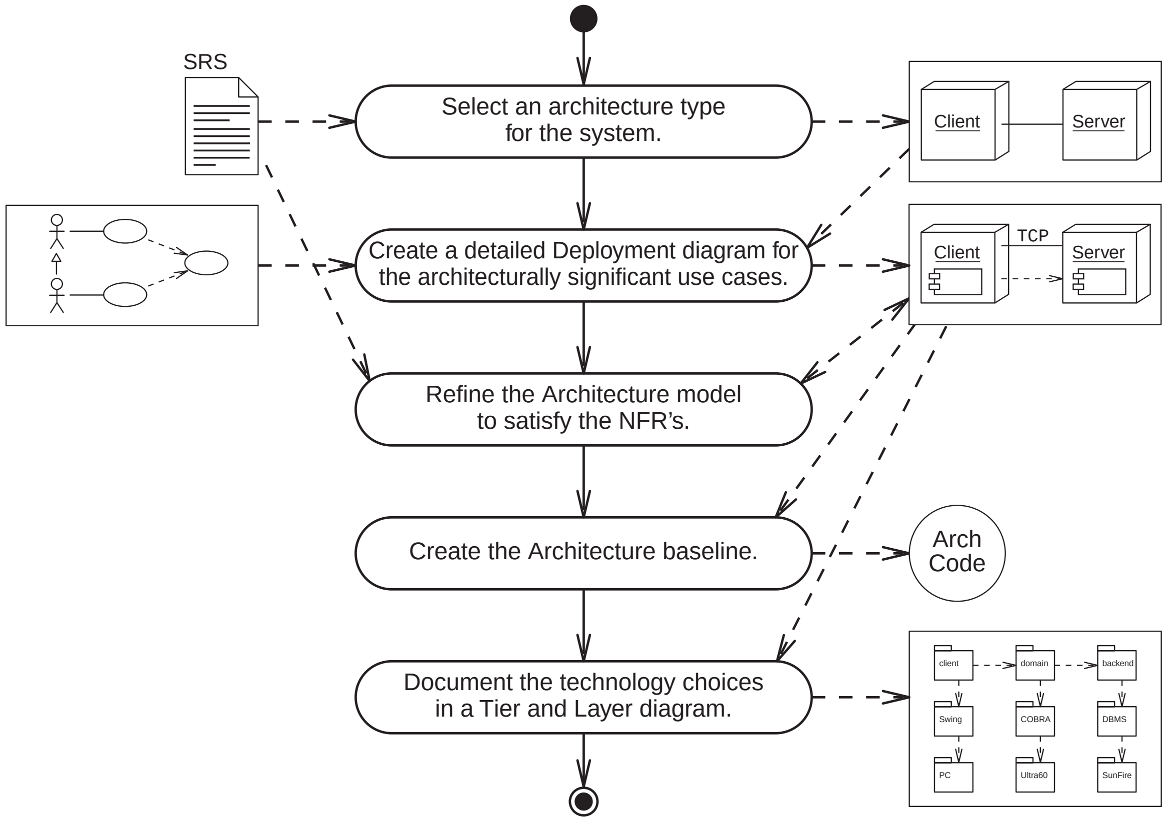
Activities and Artifacts of the Requirements Analysis Workflow



Exploring the Architecture Workflow

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

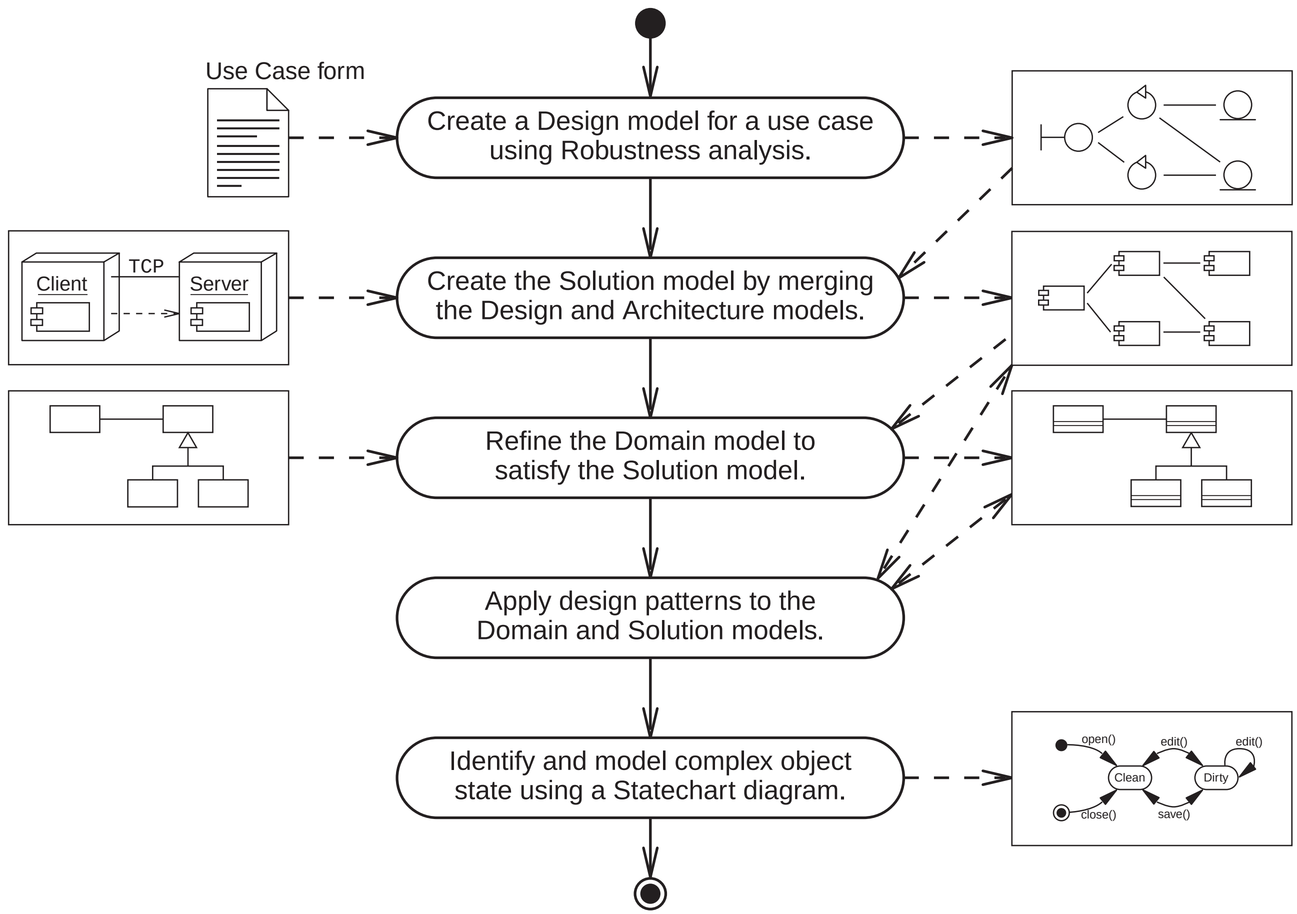
Activities and Artifacts of the Architecture Workflow



Exploring the Design Workflow

Workflow	***Purpose***	***Description***
Requirements Gathering	Determine what the system must do	
Requirements Analysis	Model the existing business processes	
Architecture	Model high-level system structure to satisfy NFRs	
Design	Model how the system will support the use cases	• Create a Design model for a use case • Create a Solution model • Refine the Domain model • Apply design patterns to the Domain and Solution models • Model a complex object state

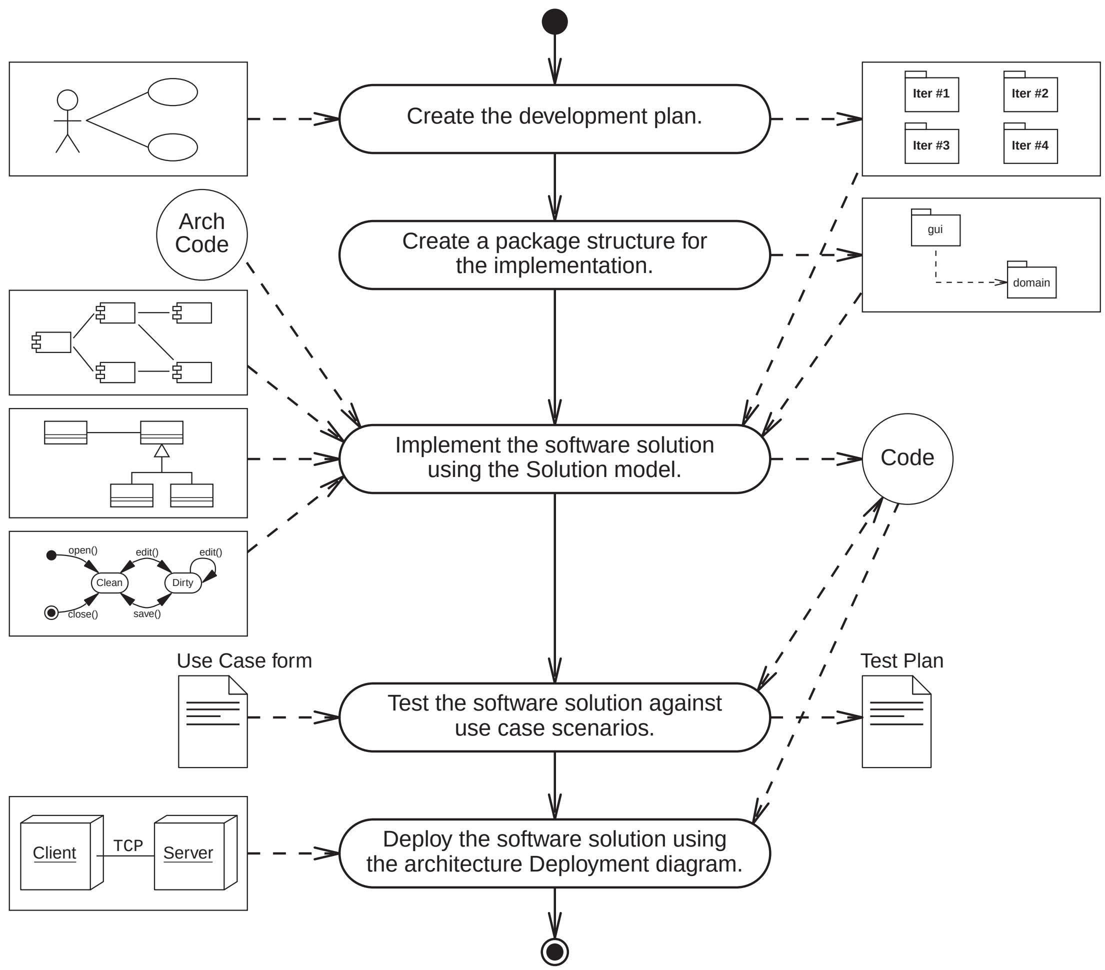
Activities and Artifacts of the Design Workflow



Exploring the Construction Workflow

Workflow	*Purpose*	*Description*
Requirements Gathering	Determine what the system must do	
Requirements Analysis	Model the existing business processes	
Architecture	Model high-level system structure to satisfy NFRs	
Design	Model how the system will support the use cases	
Construction	Implement, test, and deploy the system	• Implement the software • Perform testing • Deploy the software to the production environment

Activities and Artifacts of the Construction Workflow



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 2

Examining Object-Oriented Technology

Objectives

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

- Describe how Object-Oriented (OO) principles affect the software development process
- Describe fundamental OO principles

Examining Object-Oriented Principles

OO principles 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 UML is good at representing mental models.
- 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-Oriented Principles

> "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 principles affect the following issues:

- Software complexity
- Software decomposition
- Software costs

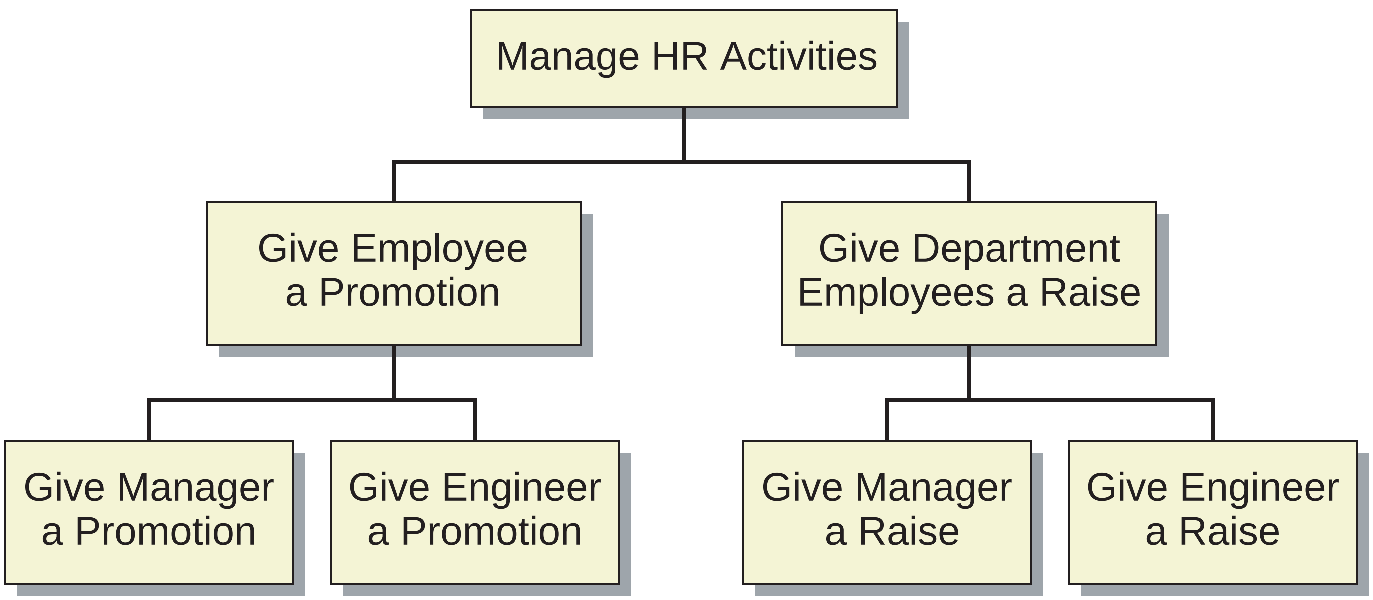
Software Complexity

Complex systems have the following characteristics:

- They have a *hierarchical structure*.
- The choice of *which components are primitive* in the system are 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*.

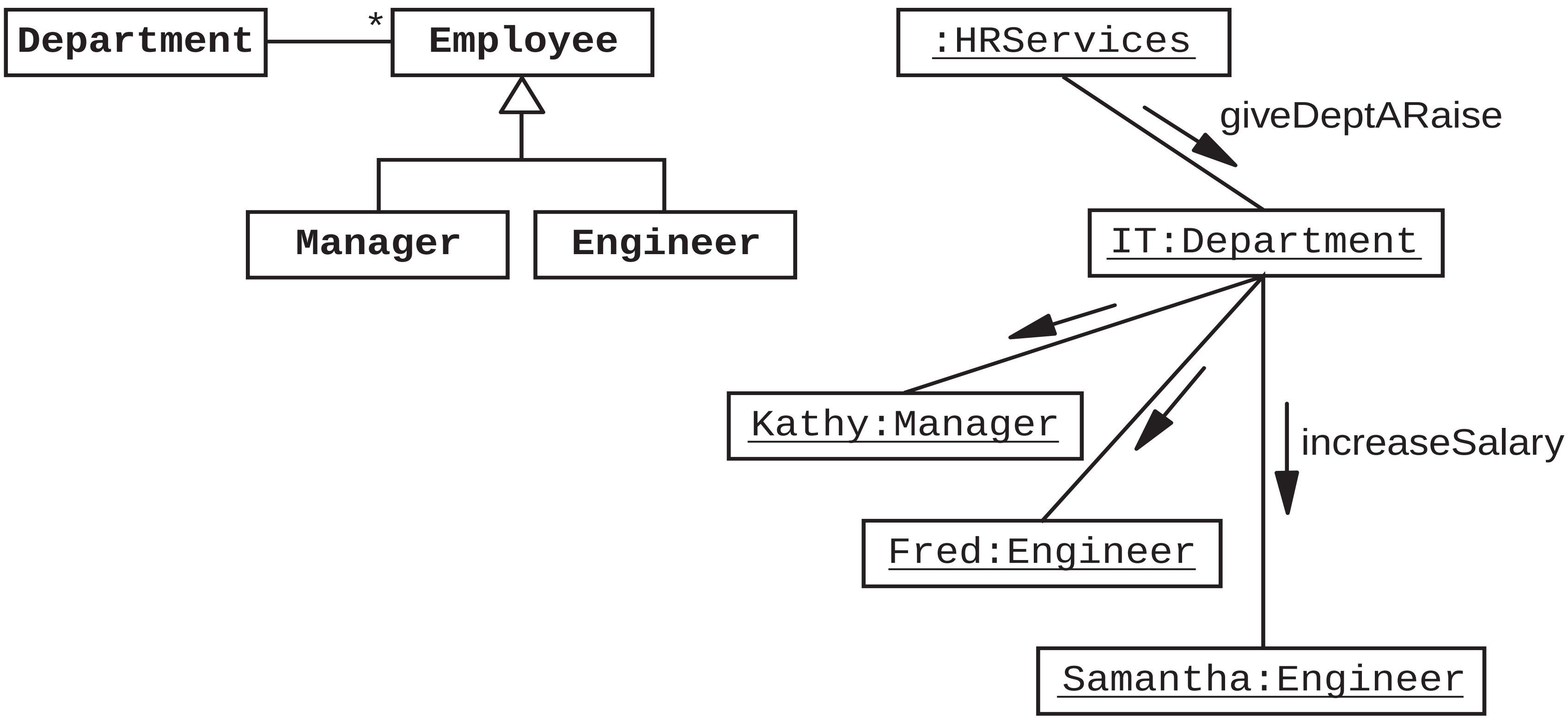
Software Decomposition

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.

Surveying the Fundamental OO Principles

- Objects
- Classes
- Abstraction
- Cohesion
- Encapsulation
- Inheritance
- Polymorphism
- Coupling
- Object associations

Objects

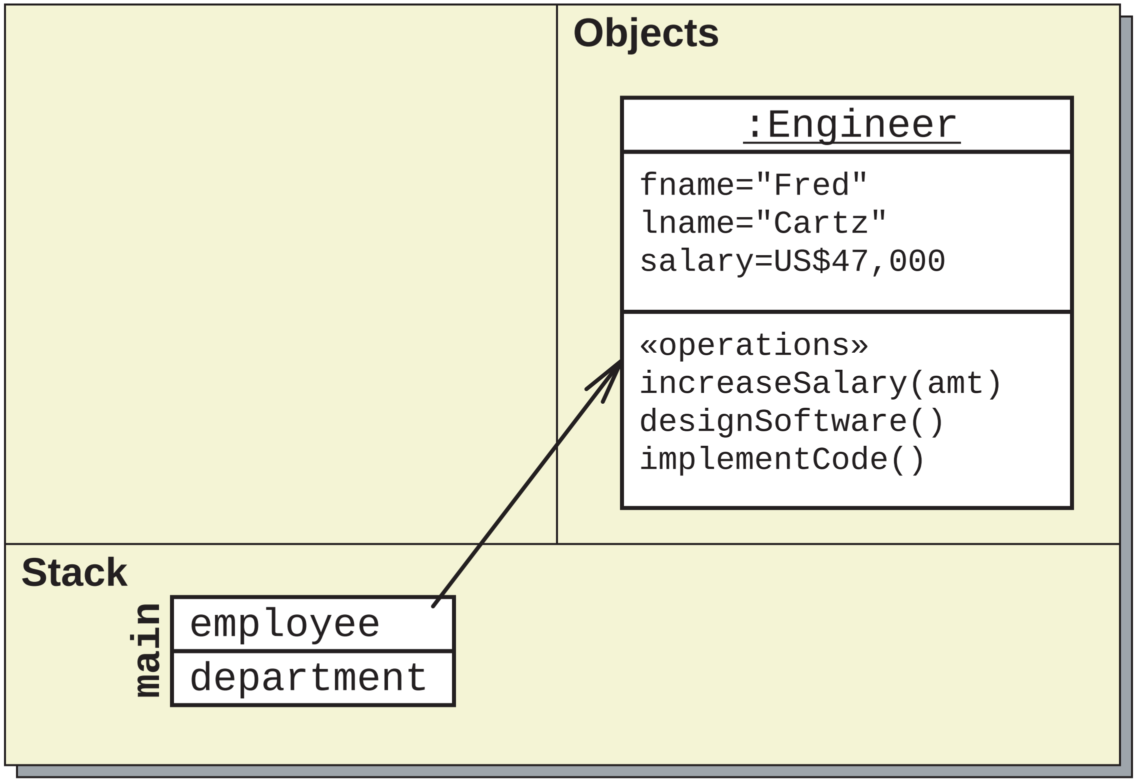
object = state + behavior

“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



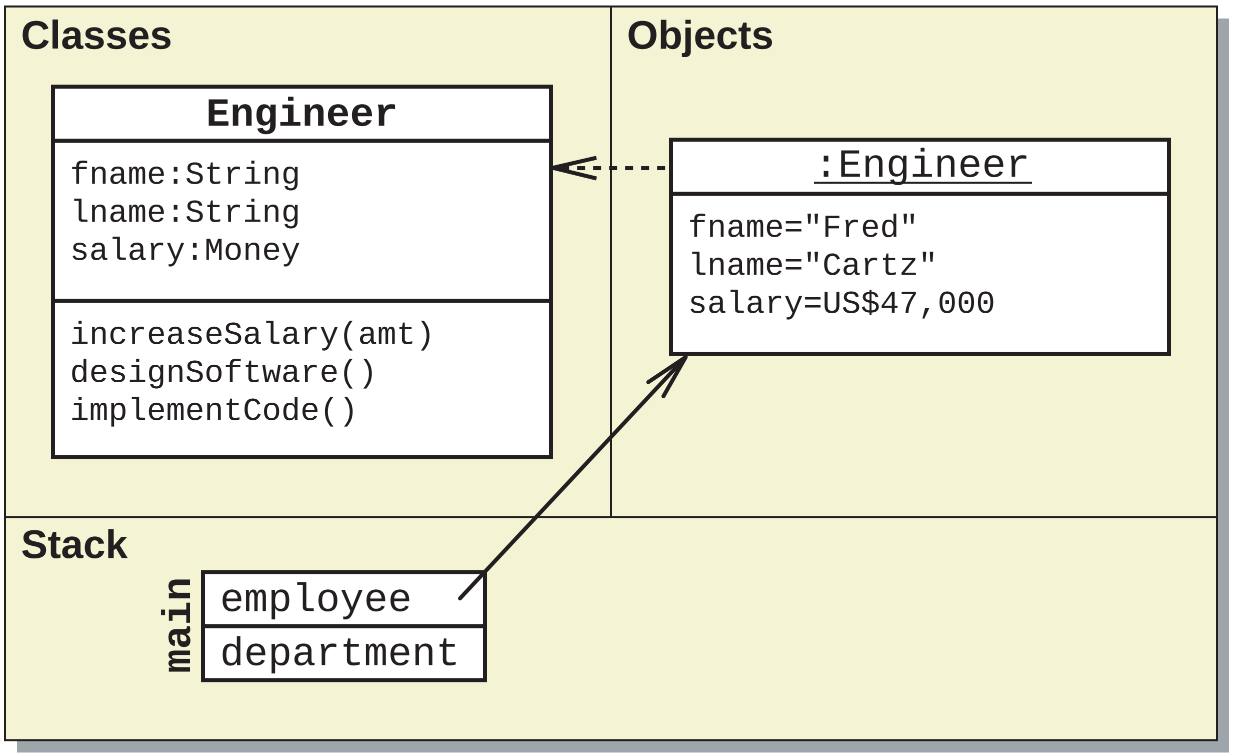
Classes

A class is "A set of objects that share a common structure and a common behavior." (Booch Object Solutions page 303)

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

Abstraction is "something that summarizes or concentrates the essentials of a larger thing" (Webster New Collegiate Dictionary)

In software, the concept of abstraction enables you to create a simplified interface to some service that hides the details of the implementation from the client of that service.

Abstraction: Example

Engineer
fname:String lname:String salary:Money
increaseSalary(amt) designSoftware() implementCode()

Engineer
fname:String lname:String salary:Money fingers:int toes:int hairColor:String politicalParty:String
increaseSalary(amt) designSoftware() implementCode() eatBreakfast() brushHair() vote()

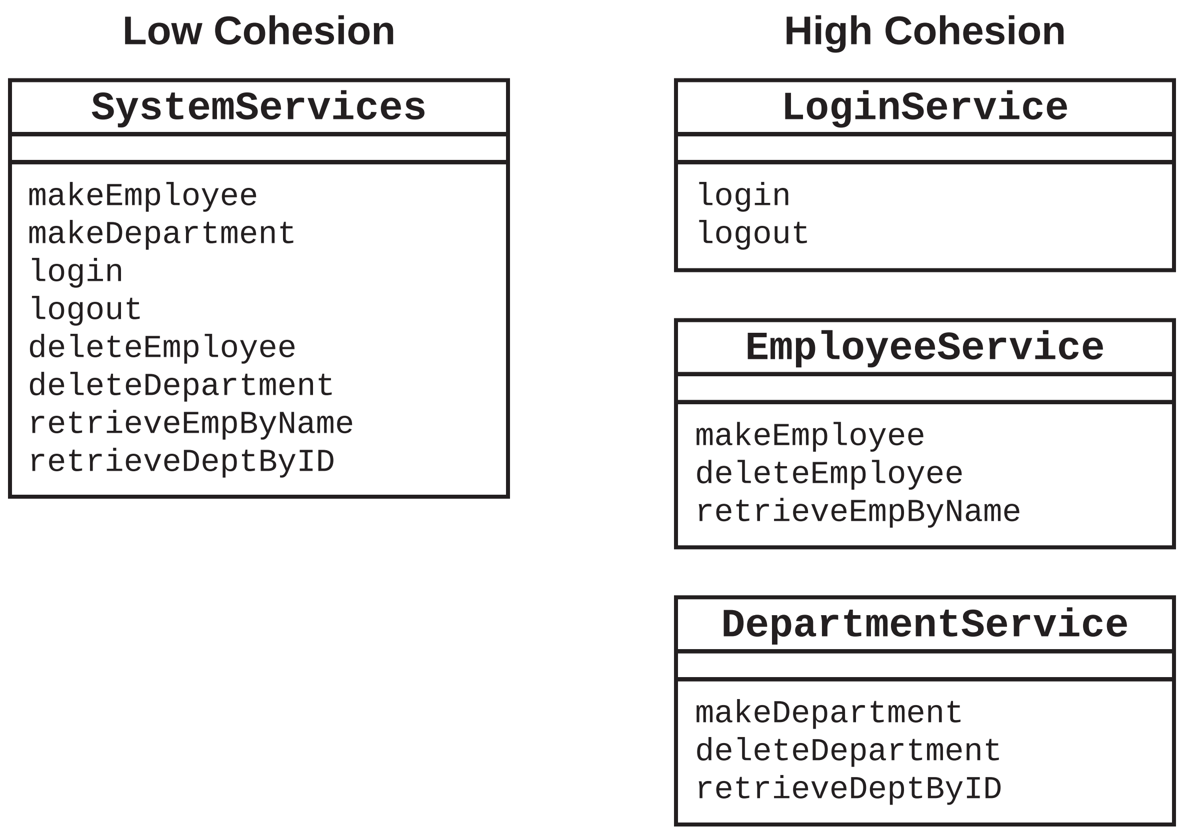
Cohesion

Cohesion is "the measure of how much an entity (component or class) supports a singular purpose within a system." (Knoernschild page 174)

Low cohesion occurs when a component tries to do too many unrelated functions.

High cohesion occurs when a component performs only a set of related functions.

Cohesion: Example



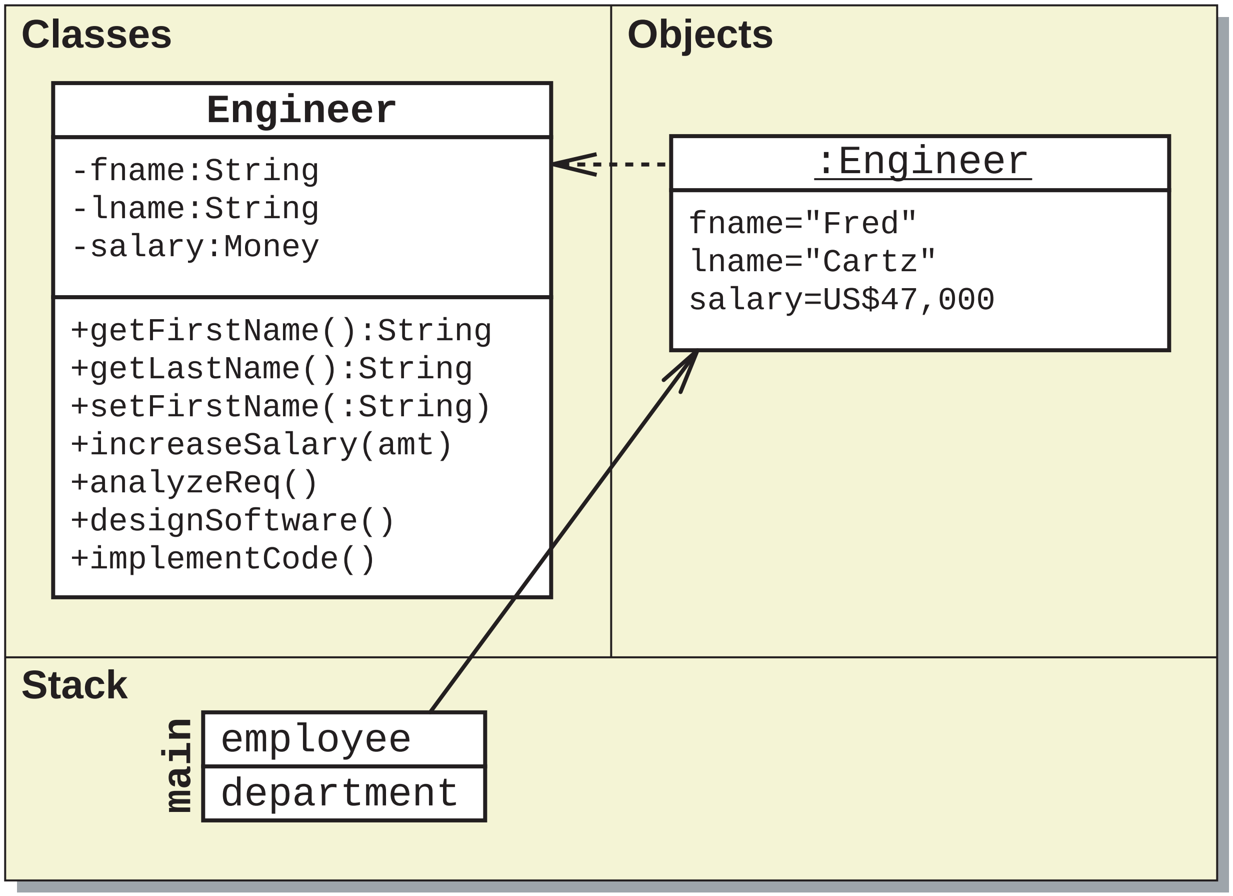
Encapsulation

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 include *information hiding*, which can be defined as: "hide implementation details behind a set of public methods."

Encapsulation: Example




```
✗ name = employee.fname;
✗ employee.fname = "Samantha";
```

```
✓ name = employee.getFirstName();
✓ employee.setFirstName("Samantha");
```

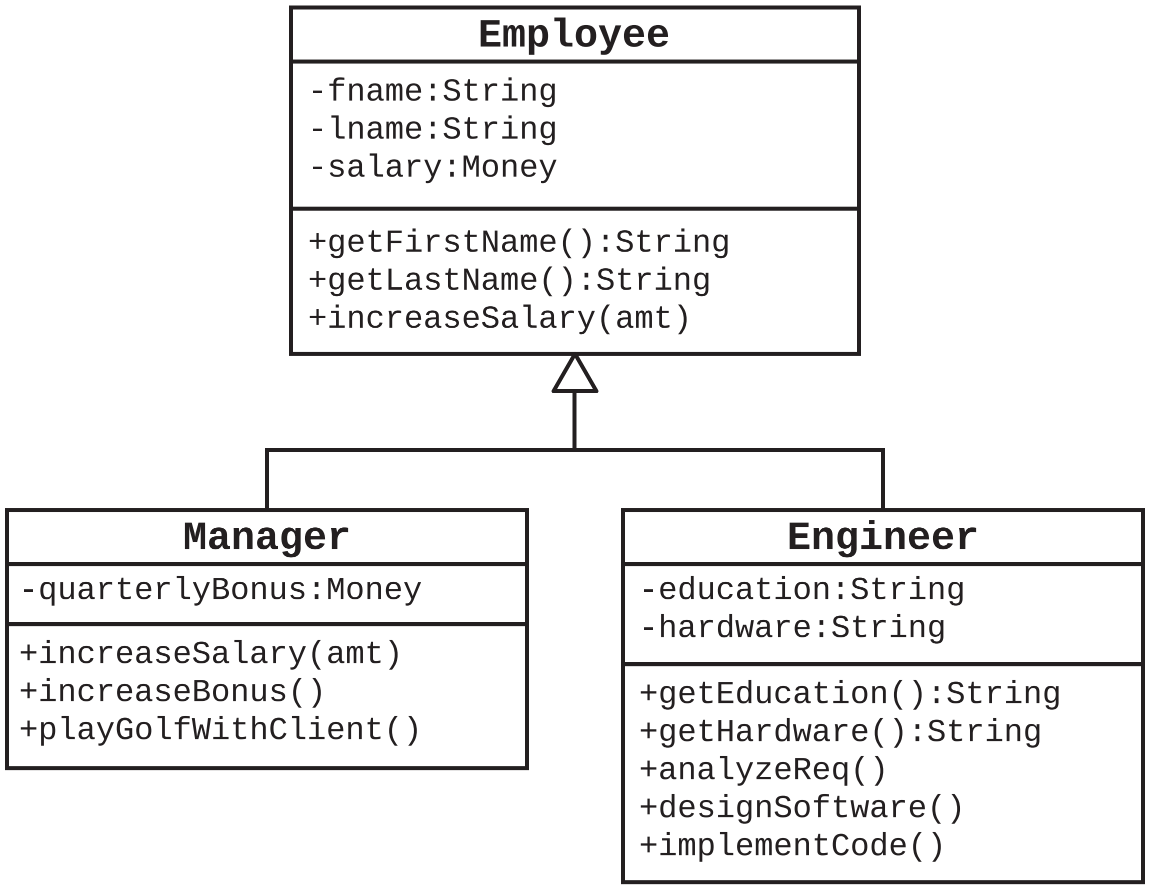
Inheritance

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
- A subclass can inherit from multiple superclasses (called multiple inheritance) *or* a subclass can *only* inherit from a single superclass (single inheritance)

Inheritance: Example



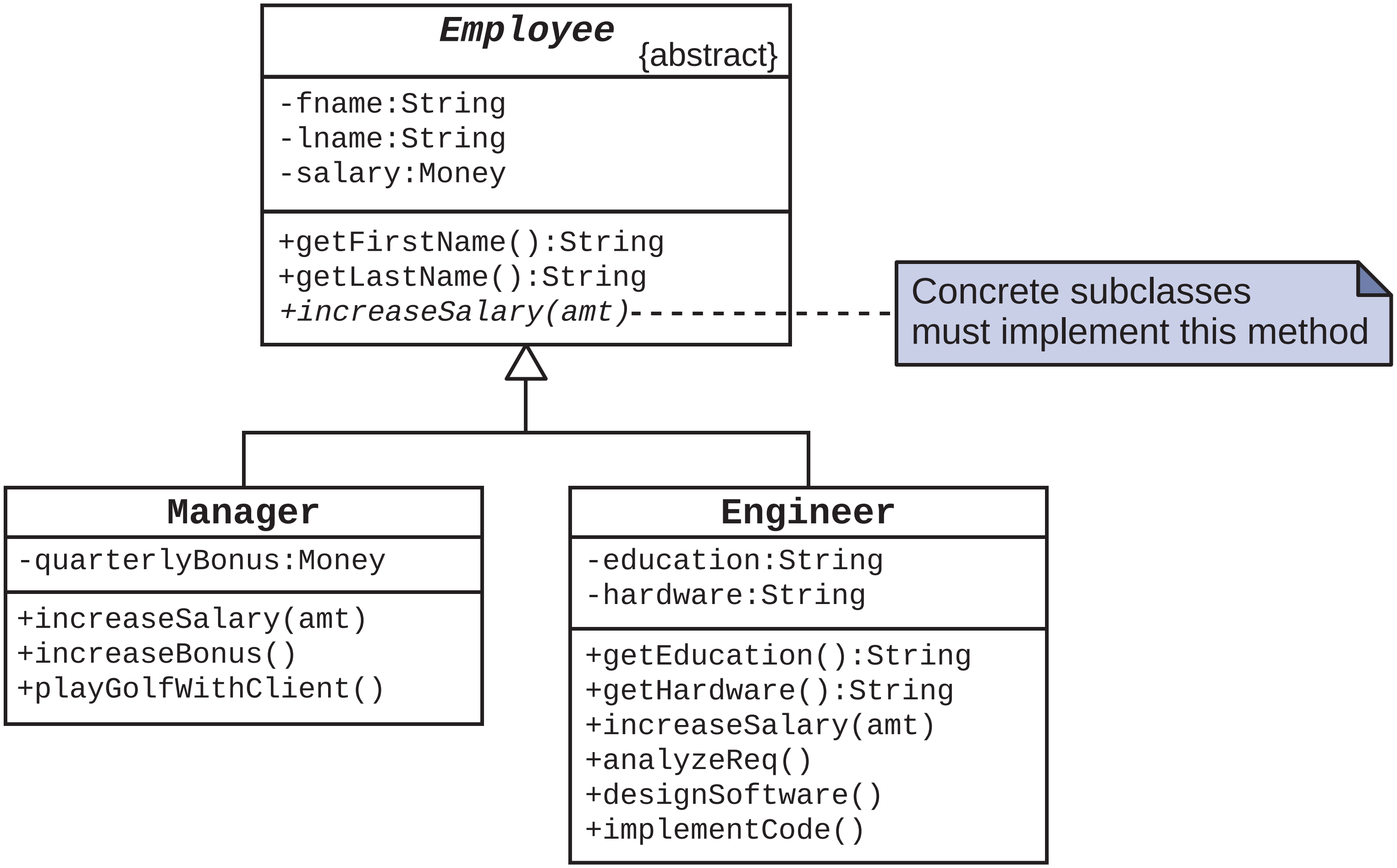
Abstract Classes

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.

Abstract Classes: Example



Interfaces

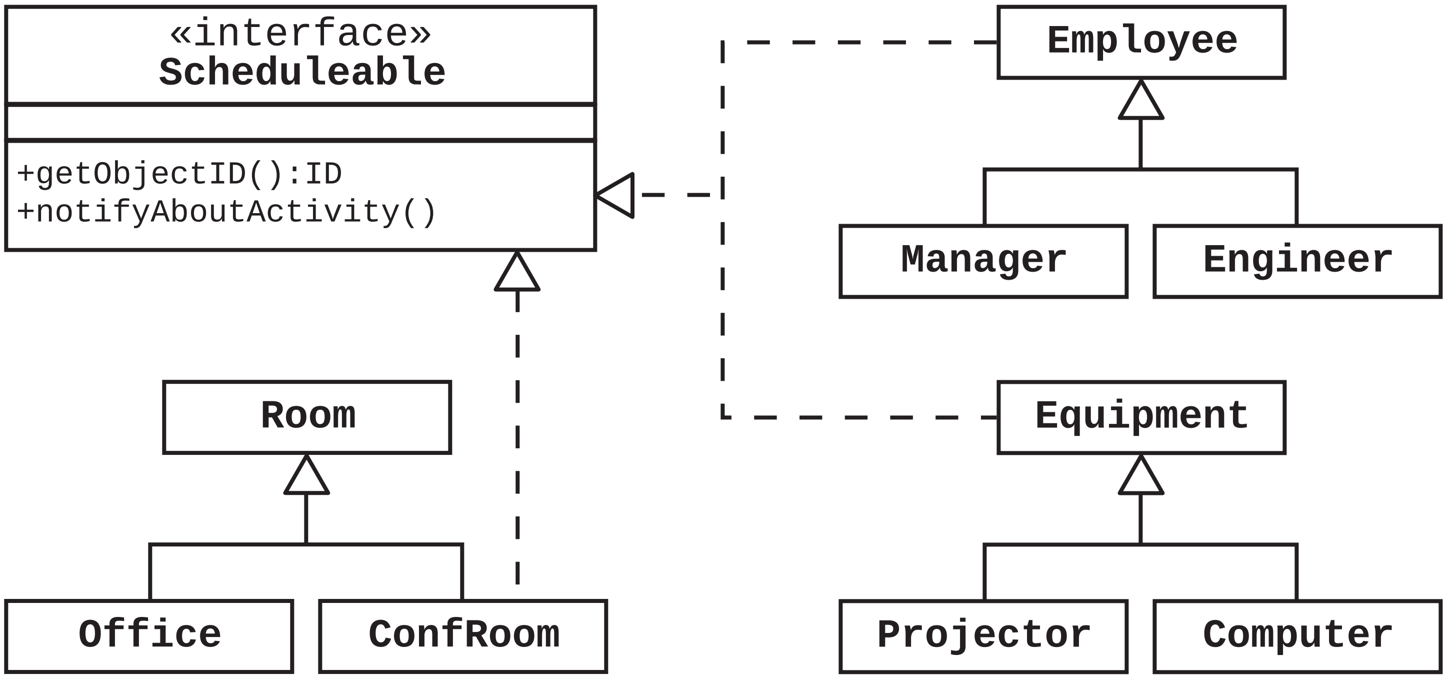
A named set of operations that characterize the behavior of an element. (UML v1.4 page B-11)

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

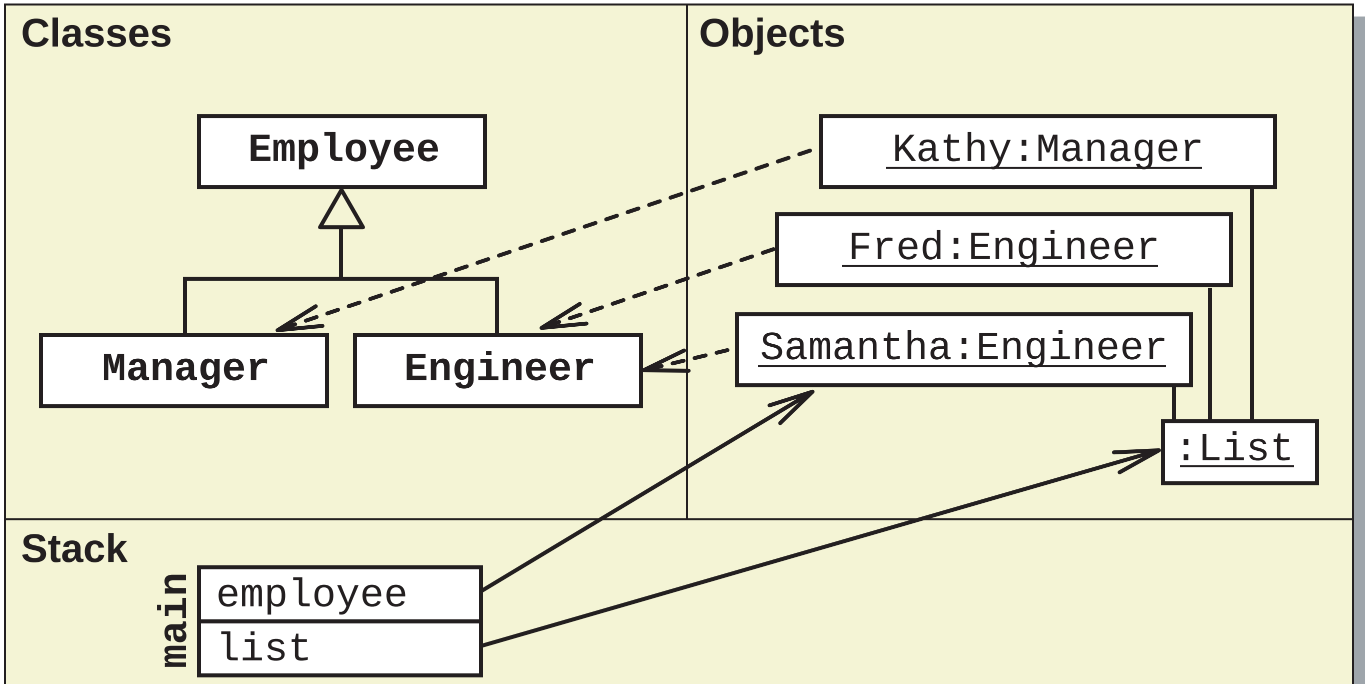
Polymorphic is "having, assuming, or occurring in various forms, characters, or styles" (Webster New Collegiate Dictionary)

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:

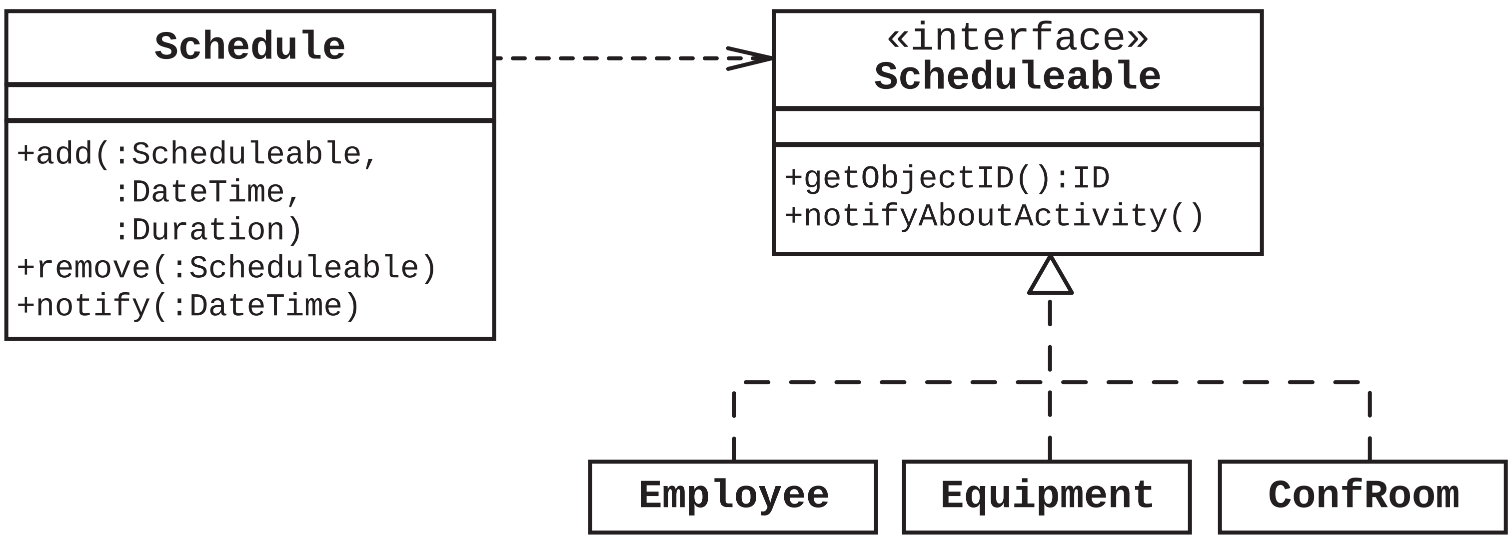
- A variable can be assigned different types of objects at runtime.
- Method implementation is determined by the type of object, not the type of the declaration (dynamic binding).

Polymorphism: Example



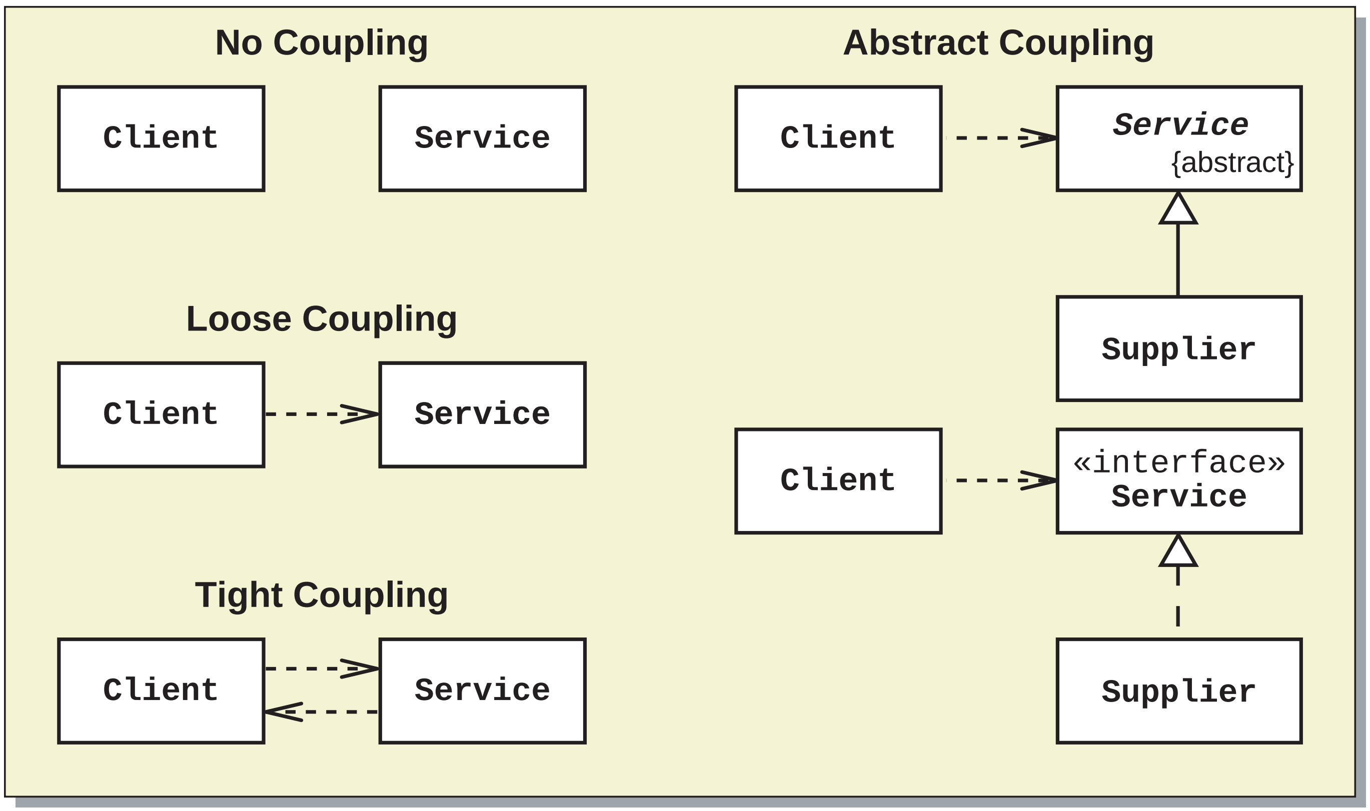

```
List list = getProjectTeam();
for ( int i=0; i<list.size(); i++ ) {
  Employee employee = (Employee) list.get(i);
  employee.increaseSalary(0.10); //10%
}
```

Polymorphism: Example



Coupling

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



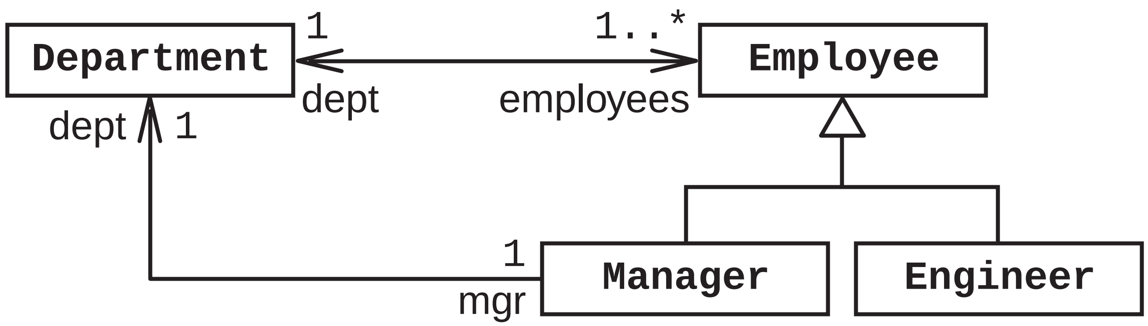
Object Associations

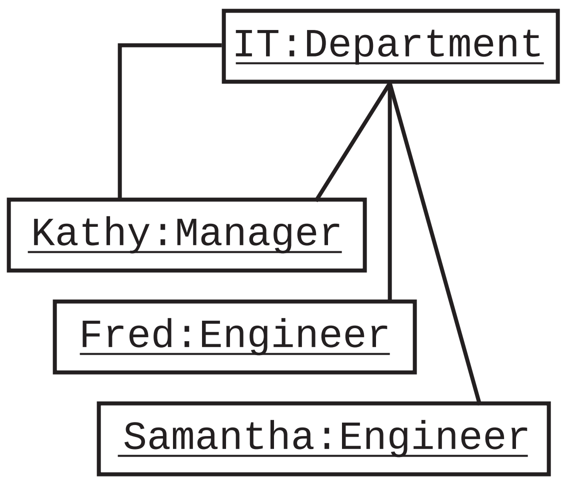
An object association is "a relationship denoting a semantic connection between two classes." (Booch OOAD page 512)

Dimensions of associations are:

- The roles that each class plays
- The multiplicity of each role
- The direction (or navigability) of the association

Object Associations: Example





Summary

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

Module 3

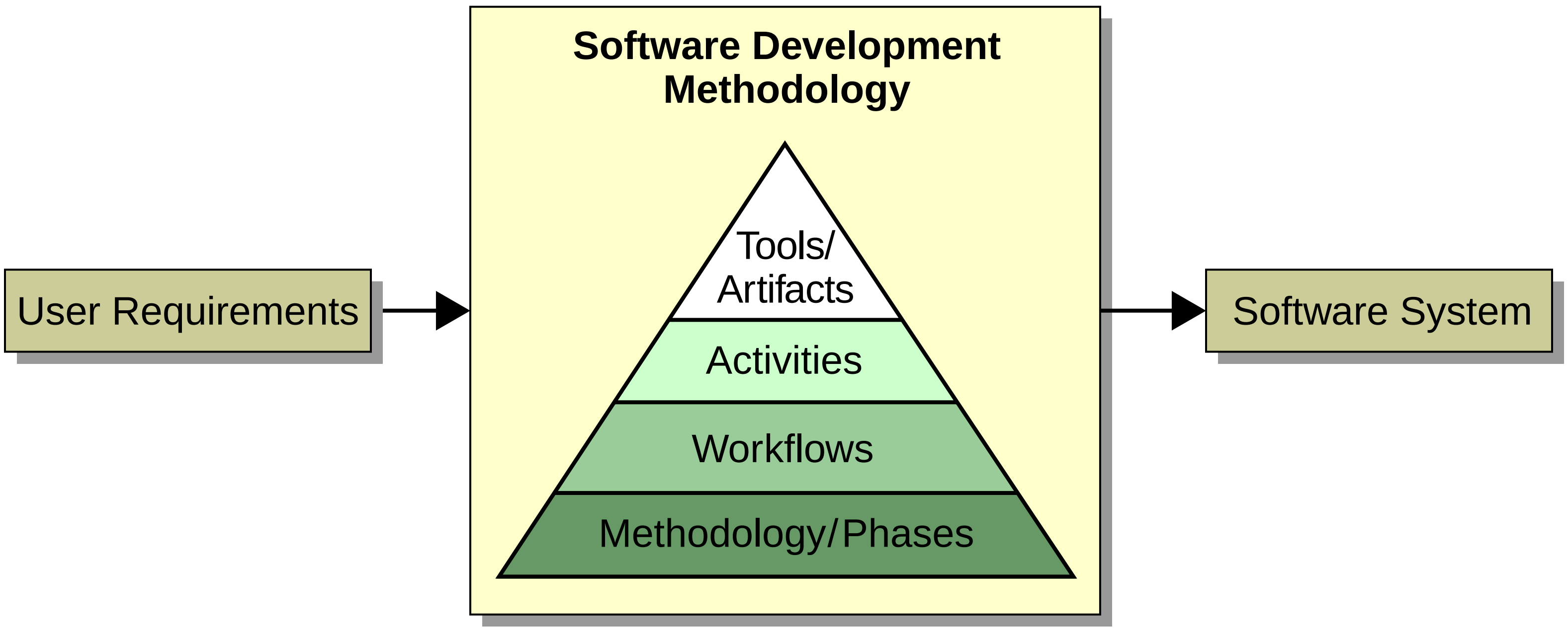
Choosing an Object-Oriented Methodology

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

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

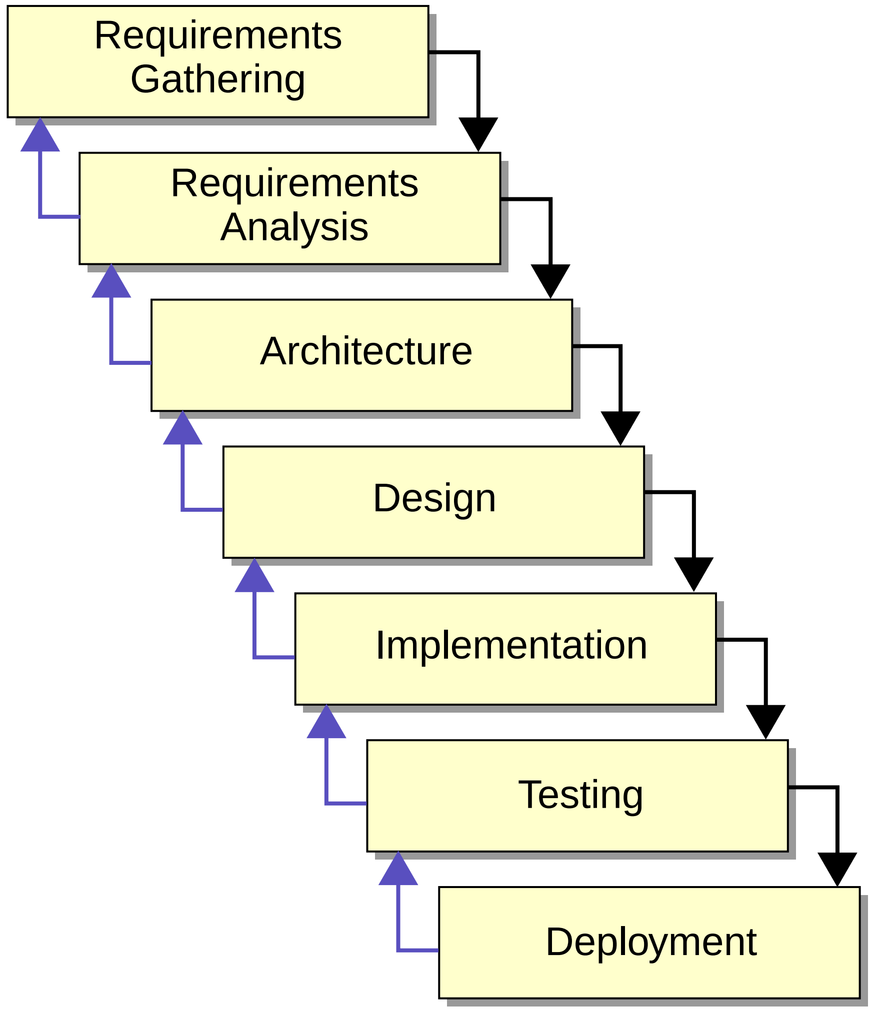
This module describes the following methodologies:

- Waterfall
- Unified Software Development Process (USDP or just UP)
- Rational Unified Process (RUP)
- SunTone^{SM} Architecture Methodology
- 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


Requirements Gathering
Requirements Analysis
Architecture
Design
Implementation
Testing
Deployment

Unified Software Development Process

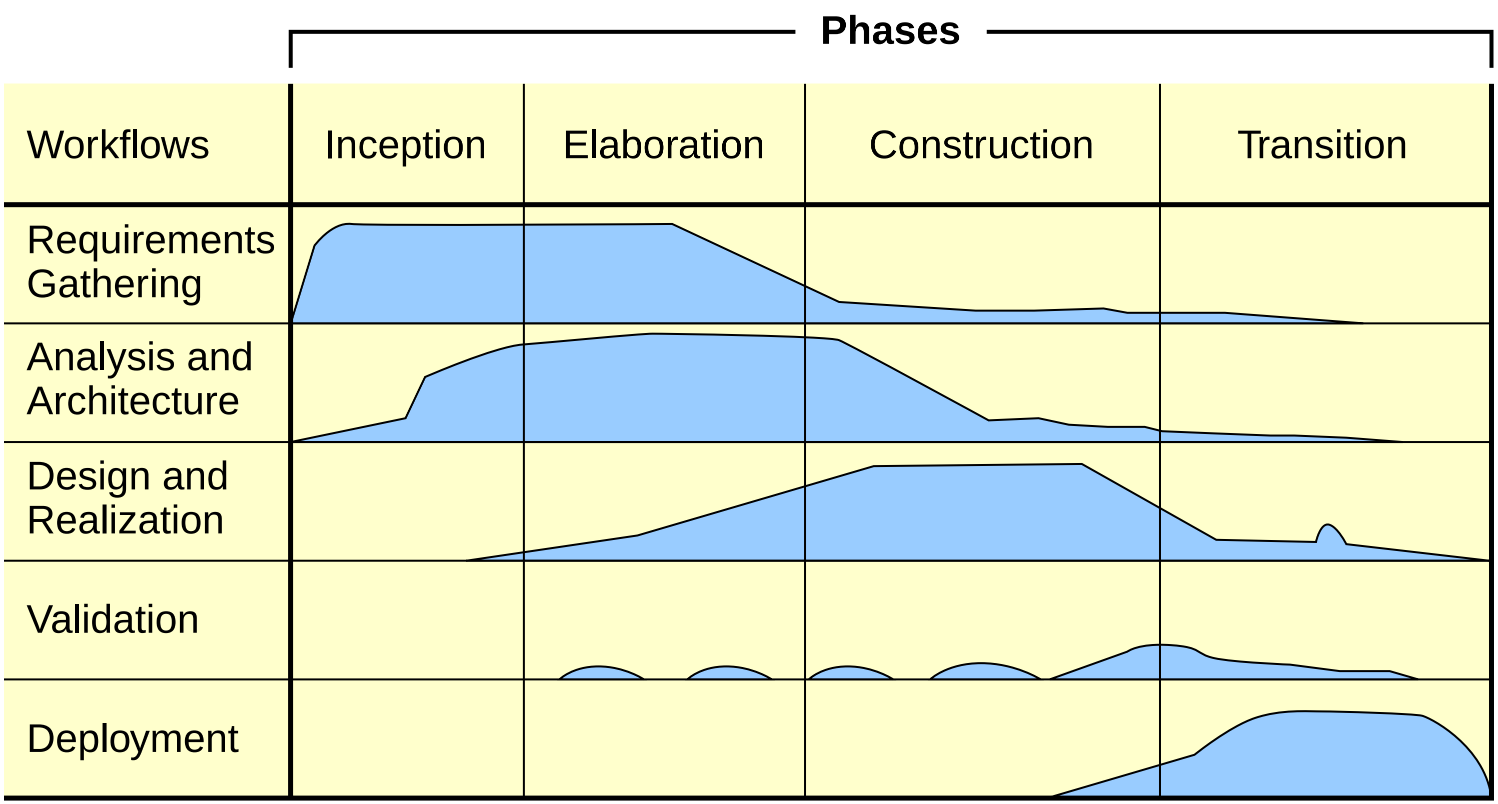
The Unified Software Development Process (USDP) is the "open" version of Rational's methodology created by Booch, Jacobson, and Rumbaugh. This is also called the Unified Process (UP).

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

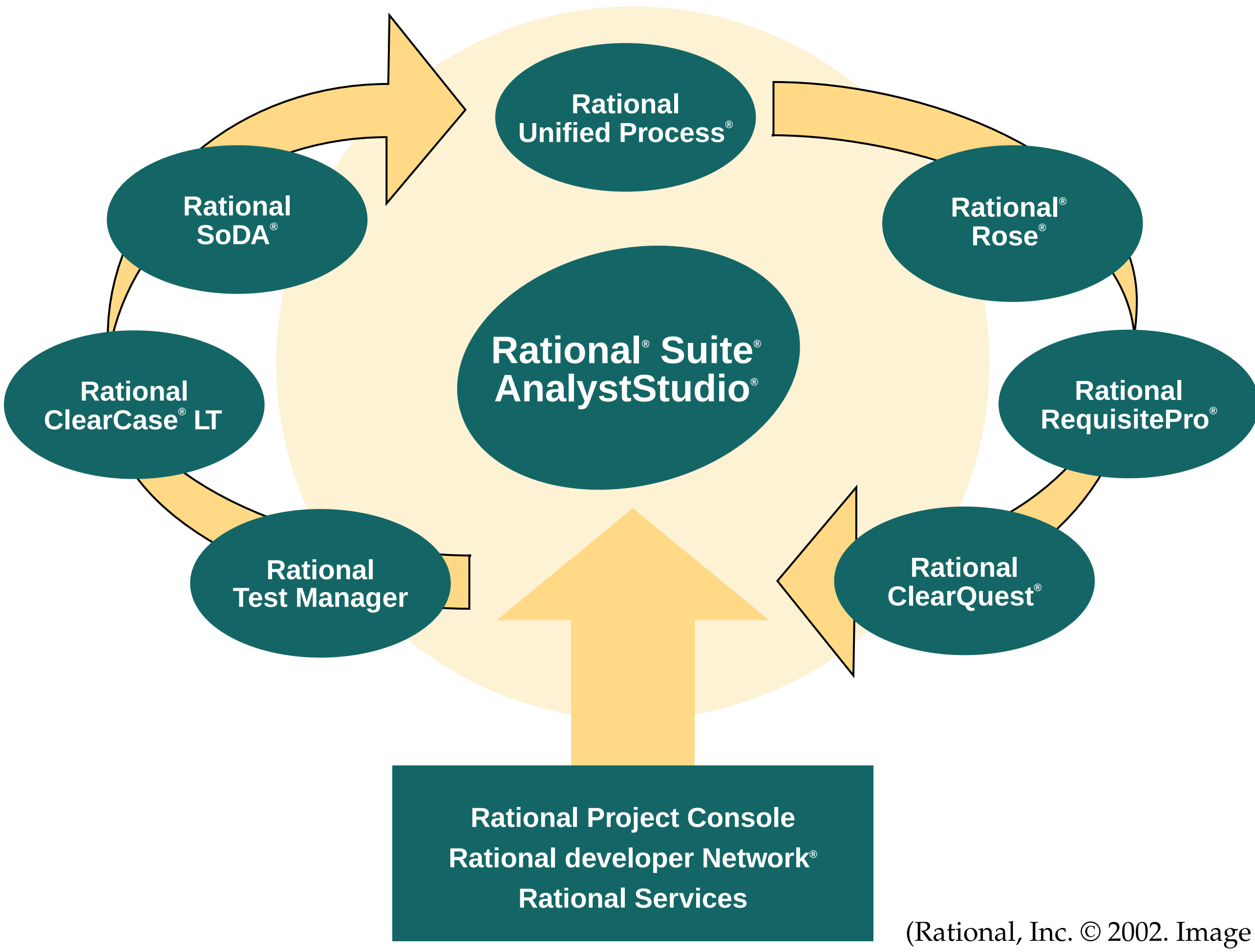


Rational Unified Process

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



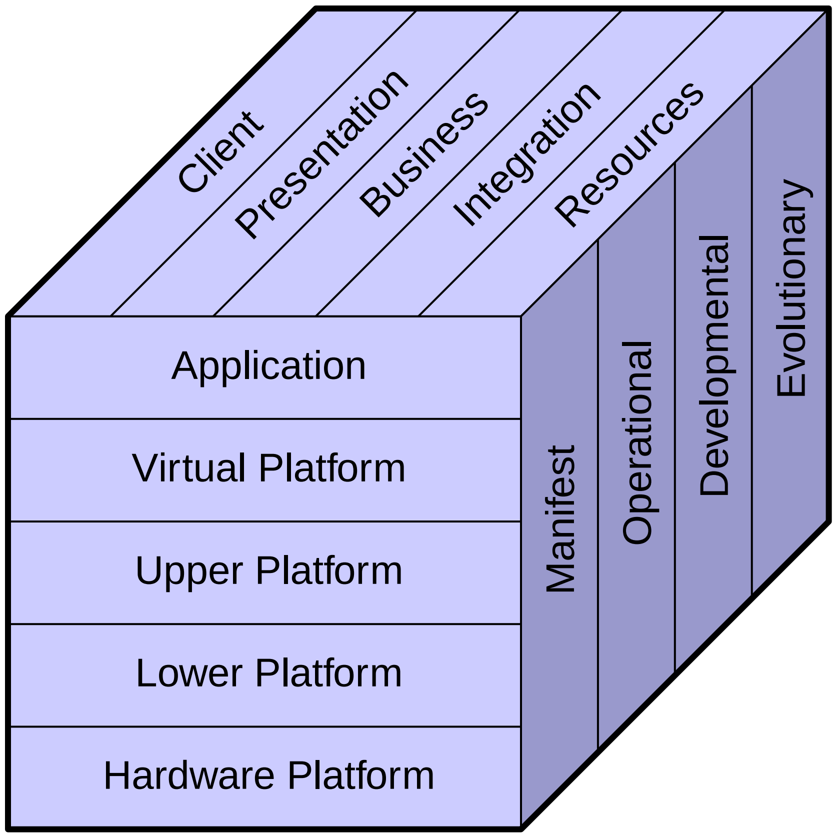
(Rational, Inc. © 2002. Image used with permission.)

SunTone Architecture Methodology

SunTone Architecture Methodology is compatible with UP:

- Uses UP's phases and workflows
- Adds an emphasis on architecture, especially for enterprise applications
- Includes a "3D Cube" visualization of an architecture

SunTone Architecture Methodology



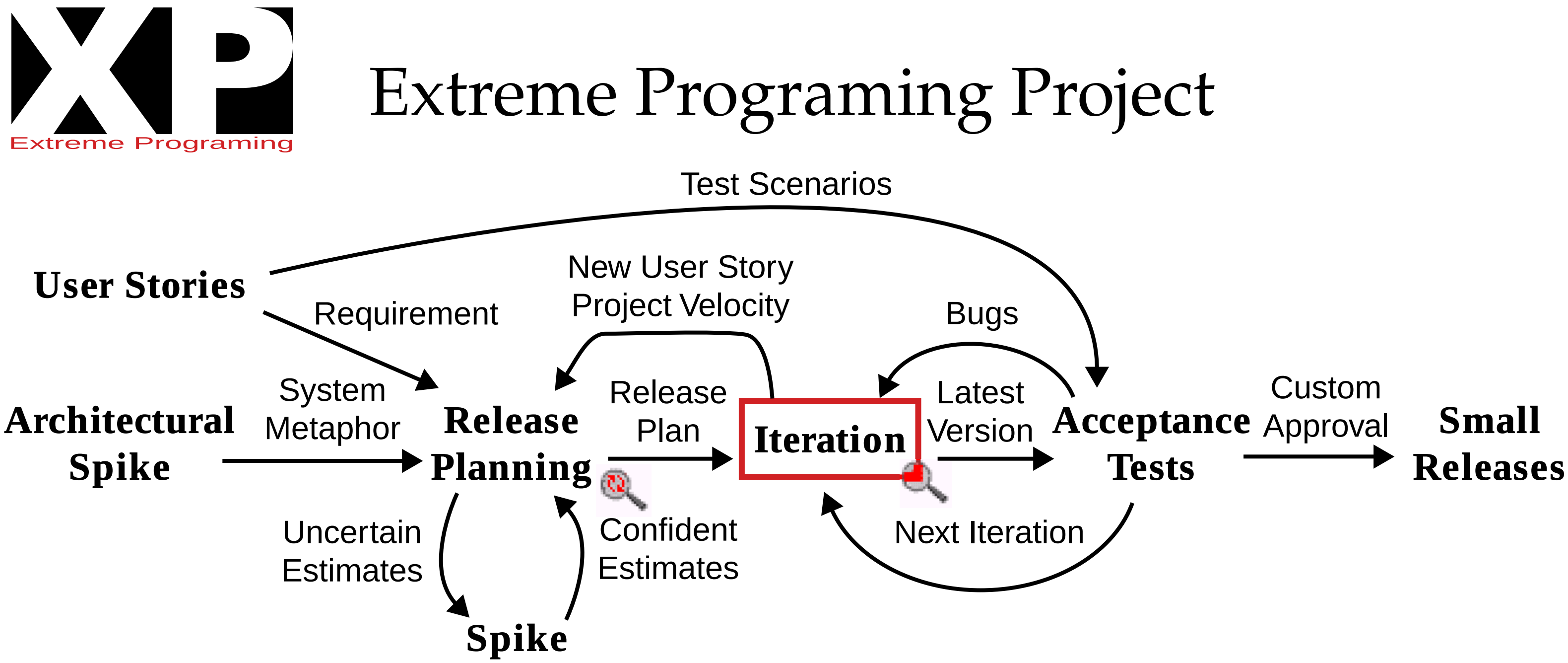
eXtreme Programming

"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



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

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 SunTone Architecture Methodology

When to use:

- Same reasons as UP
- Enterprise applications, or any architecturally *heavy* systems

Issues:

- Same issues as UP
- Until recently, little has been published about SunTone Architecture Methodology Service

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

Summary

- The following are methodological *best practices*:
  - Use-case-driven and systemic-quality-driven
  - Architecture-centric
  - Iterative and incremental
  - Model-based
  - Design best practices
- This course, while mostly methodology-independent, provides examples and context from the SunTone Architecture Methodology.
- No one methodology fits every organization or project. You can create your own methodology by specializing existing methodology and following best practices.

Module 4

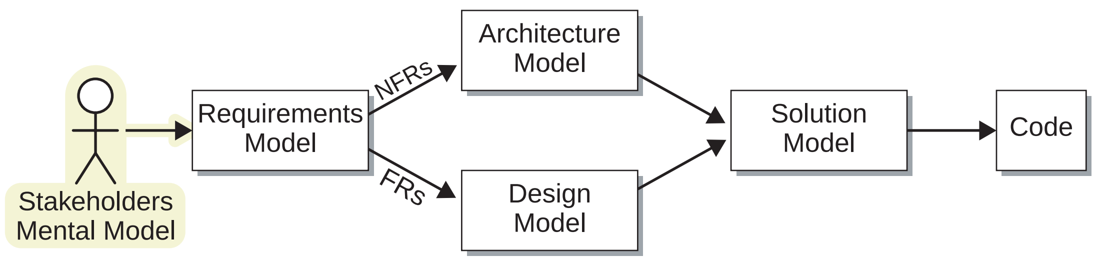
Determining the Project Vision

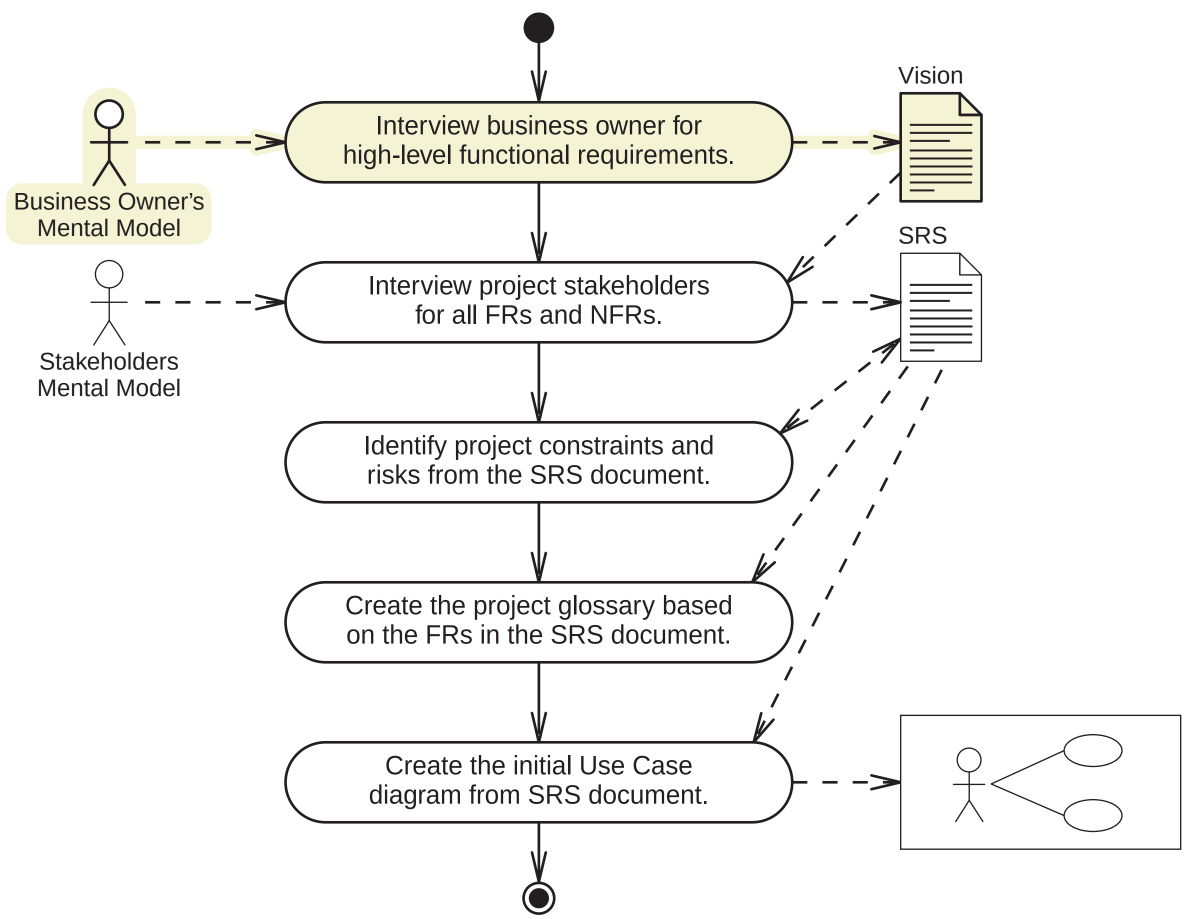
Objectives

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

- Interview business owners to determine functional requirements of the software system
- Identify the project risks and constraints from these interviews
- Create a project Vision document from the results of the interviews and risk analysis

Process Map


Architecture Model
NFRs
Requirements Model
FRs
Design Model
Solution Model
Code
Stakeholders Mental Model


Vision
Interview business owner for high-level functional requirements.
Business Owner's Mental Model
SRS
Interview project stakeholders for all FRs and NFRs.
Stakeholders Mental Model
Identify project constraints and risks from the SRS document.
Create the project glossary based on the FRs in the SRS document.
Create the initial Use Case diagram from SRS document.

Interviewing Business Owners

To determine the vision of a project, you interview the business owner of the project to define the high-level functional requirements.

- A *business owner* is anyone who *owns* the project.
- A functional requirement is a description of *what* activity the system must perform for an actor.

Types of Requirements

Functional requirements (FRs):

- FRs describe features of a system that supports an actor performing a *business operation* using the system.
- For example: The system must collect the following customer information - name and address.

Non-functional requirements (NFRs):

- NFRs describe features of a system that support *how* an operation is performed.
- For example: The system must support 10 simultaneous users in the Web application.

Interview Skills

Interviewing is not an easy skill to learn. Here are a few tips:

- Build rapport with business owner; engage in light conversation, but avoid telling jokes
- Listen carefully
- Gently steer the direction of the interview; carefully interrupt the business owner if she digresses *too long*
- Listen carefully
- Rephrase unclear statements and ask for validation
- Listen carefully
- Take detailed notes
- Listen carefully

Vision Interview Focus

The vision interview will focus on these areas:

- Business case for the project
- Functional requirements for the project
- Risks
- Constraints
- Stakeholders

Business Case Questions

Have the business owner explain why the software is needed.

- How does your current business operate?
  - What does your company do, make, or sell?
  - How is the company structured?
  - May I have an organizational chart of the company (or relevant business unit)?
- How is the new software intended to support the business?
- How is your business changing?
  - Do you plan to expand your business?
  - How might the company be reorganized?

Questions Used to Discover Functional Requirements

Have the business owner explain what the software must do for the business.

- Ask the business owner to list (and describe) the top 10 use cases of the system.
- Reiterate to the business owner each use case:
  - Verify your understanding of the use case.
  - Ask the business owner to prioritize the use cases into these categories: essential, high-value, follow-on.
  - Reiterate the list of use cases and ask if any important use cases have been missed.

Questions Used to Discover Risks

There are five main risk areas. These are some questions that help identify project risks:

- Are there other groups in your business doing similar functions?
- Do you plan to use new technologies (for instance, J2EE™ platform or Simple Object Access Protocol) on the project?
- Do you have the development resources or do you plan to outsource the project?
- Do your team members have the necessary skills?
- What part of the business is likely to change, affecting the software system?

Questions Used to Discover Constraints

These questions identify hidden constraints on the project:

- Will this project be developed on a specific platform?
- Will this project require specific technologies?
- Does the project have a fixed deadline?
- Will the system interact with any external systems?
- What are the constraints on the operational side of the software?

Questions Used to Discover Stakeholders

Ask the business owner for the names of the primary stakeholders of the system.

- Who has the authority to make decisions about the functional requirements of the system?
- Who will be using the system?
- Who will be managing the users of the system?
- Who will be managing the operations of the system?
- Who will be managing the development of the project?

Analyzing the Vision Interview

From the interview notes, the business analyst identifies:

- Functional requirements
- Non-functional requirements
- Risks
- Constraints

Identifying NFRs

Occasionally the business owner might suggest requirements regarding the quality of service that the system must uphold. These are NFRs and should be recorded:

- Any adverbial phrase could be an NFR
  - “Response time must be *really* fast”
  - “We have *many* customer records in the database”
  - “The system must support *up to 100* users”
- Any specific technology mentioned is either a constraint or NFR
  - “We want our customer to use the system online”
  - “We used ORACLE® on a sister project”

Identifying Risks

Risk is a major reason why some projects fail. A successful project identifies risks early and creates a mitigation strategy.

There are five main risk areas:

- Political
- Technological
- Resources
- Skills
- Requirements

Political Risks

A political risk exists when:

- There is a competing project or competitor.
- The project manager’s boss has revoked funding, equipment, or human resources without warning in the past.
- There are interpersonal problems or infighting either within the development team or within the management hierarchy.
- The project conflicts with governmental laws and regulations.

Technology Risks

A technology risk exists when the project might use a technology that is unproven, cutting edge, or difficult to use.

Here are a few warning signs:

- The business owner uses a lot of technology buzz words, and does not really understand their meaning.
- The business owner insists that the project take a specific technological direction.
- The business owner wants to use cutting-edge technologies to solve the business problem.
- The new system must integrate with a legacy system.

Resource Risks

A resource risk exists when the project does not have all of the resources (human, equipment, or money) required for successful completion.

Here are a few warning signs:

- The business owner mentions that the project is under a tight budget.
- The IT staff is currently over committed.
- The project is calendar-driven.

Skills Risks

A skills risk exists when the development team does not have the necessary knowledge or experience to perform the job.

Here are a few warning signs:

- When the project is constrained to use a specific technology but the development team has not been trained in that area.
- When the project will be developed in a language that is not known by the development team.

Requirement Risks

A requirement risk exists when a given use case or any FR is not completely known.

Here are a few warning signs:

- When business owner says something like “I will know it when I see it.”
- When the business owner cannot provide a scenario for the use case.
- When the business owner does not know that a use case exists.

Creating the Vision document

The Vision document records the business owner's *vision* of the software system and also documents the risks and constraints.

There are five sections in the Vision document:

- Introduction (including the problem statement)
- Business opportunity
- Proposed solution (includes FRs and NFRs)
- Risks
- Constraints

Keep the Vision document as short and clear as possible.

Writing a Problem Statement

The Problem Statement is a succinct summary of the business problem. This does not have to include *all* of the FRs or use cases, but *at least* the significant ones.

Example:

> The Hotel Reservation System will be responsible for *managing the reservations for multiple lodging properties*, which include (but are not limited to) bed and breakfast (B&B) and business retreat properties. The system will also include a web application that permits customers to view the properties and rooms, to view current and past reservations, and to make new reservations. The system must also *coordinate small business conferences*.

Documenting the Business Opportunity

This section of the Vision document records the business owner's vision of the company (present and future) and how the software system will support the business.

Example:

> *Bay View B&B is a family-owned company* started in Santa Cruz, CA, by co-owners Peter and Mary Jane Parker in 1987. In 1995, they purchased another B&B in Sonoma, CA. Business has been good, so when Mary Jane and Peter were vacationing at a Sierra Madre resort in 2001, they talked to the owner and discovered that he was ready to retire. *This was the opportunity they were looking for to expand their business.* They are currently working on closing this deal. *The Hotel Reservation System is being proposed to provide an integration of these facilities.*

Documenting the Proposed Solution

This section of the Vision document records all of the requirements (both FR and NFR) identified by the business owner.

- List FRs as short sentences that describe how an actor uses the system. For example:

  The system shall include a unified web presence, in multiple languages, including pictures of rooms and properties.

  The system shall integrate reservations across all properties.

  The receptionist shall be able to check in and check out a customer.

- Group FRs into their priority categories.

Documenting the Proposed Solution

- Briefly describe any NFRs that were identified during the business owner interviews.

Example:

Creating a reservation on-line (by the customer) must take no more than *10 minutes from start to finish*. The expected throughput is no more than *10 transactions per minute*. The expected throughput is expected to scale to only *20 transactions per minute within 5 years*, and possibly 50 transactions per minute within 10 years. The system must allow additional properties to be integrated into the system in the future.

List the Identified Risks

This section of the Vision document records all risks that were identified in the business owner interviews.

Example:

> The main risk in this project is to *determine a strategy for data conversion from the existing data stores* (spreadsheets for B&Bs and flat-file data store for the resort) to the new data store. Another potential risk is the *cost/benefit trade-off of outsourcing the development of the system as opposed to building an in-house development team*. Mitigation of these risks will be addressed in the SRS document.

List the Identified Constraints

This section of the Vision document records all constraints that were identified in the business owner interviews.

Example:

> Due to the cost of buying the resort property, the Parkers cannot afford expensive tools such as a database server or the web application server. *You should use proven, Open Source tools where applicable*.

Summary

- Determine the vision of the project by interviewing the business owner.
- This interview should focus on the high-level use cases of the system. These use cases are the basis for the functional requirements.
- Identify risks and NFRs (both technology constraints and quality of service requirements) from the interview responses.
- Record the results of this interview in a Vision document.

Module 5

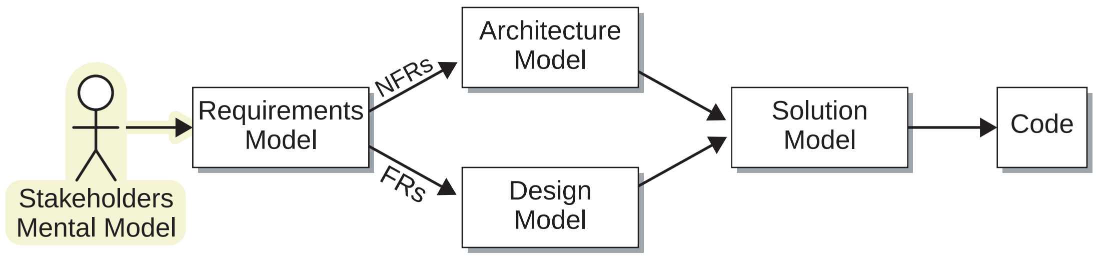
Gathering the System Requirements

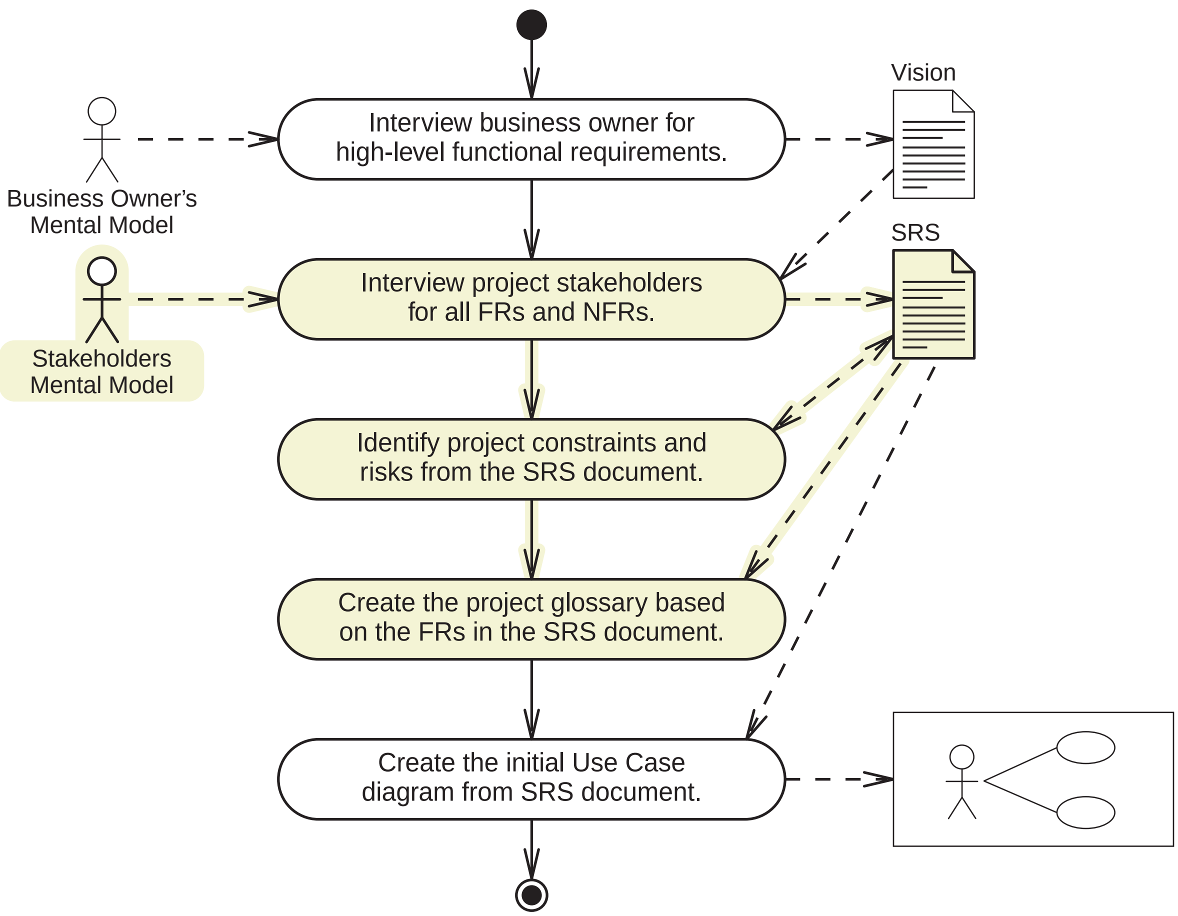
Objectives

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

- Identify sources of requirements
- Interview stakeholders to validate and refine the FRs and NFRs from the Vision document
- Document the system in the System Requirements Specification (SRS) from all requirements sources

Process Map


Stakeholders
Mental Model
Requirements
Model
NFRs
FRs
Architecture
Model
Design
Model
Solution
Model
Code


Business Owner's
Mental Model
Interview business owner for
high-level functional requirements.
Vision
Stakeholders
Mental Model
Interview project stakeholders
for all FRs and NFRs.
SRS
Identify project constraints and
risks from the SRS document.
Create the project glossary based
on the FRs in the SRS document.
Create the initial Use Case
diagram from SRS document.

Planning for Requirements Gathering

You need to:

- Identify all necessary sources of requirements
- Identify all stakeholders to interview

Identifying Sources of Requirements

Requirements for a system can come from many sources. Here are a few important sources:

- Interviewing stakeholders
- Observing users on the job
- Analyzing and documenting policies and procedures
- Analyzing the existing system
- Analyzing and documenting inputs and outputs

Identifying Stakeholders

The initial list of stakeholders comes from the business owner. However, this list should be expanded as you begin to interview the first round of stakeholders.

Stakeholders include:

- Primary users of the system
- Operational users of the system, system administrators, and network administrators
- Managers of primary and operational users
- Domain experts
- Marketing product manager

Stakeholder List

Create a stakeholder list for each actor role:

Job Role	Primary Stakeholder	Secondary Stakeholders
Owner	Mary Jane Parker	Mary Jane Parker Peter Parker
Manager	Frodric Bagend	Samuel Gamgee (Santa Cruz) Luz Hammarstrom (Sonoma) Frodric Bagend (Sierra Madre)
Receptionist (includes Booking Agent)	Medoca Sansumi	Medoca Sansumi (Santa Cruz) David Hammarstrom (Sonoma) Judith Brown (Sierra Madre)
Event Coordinator	Harold Harkening	Harold Harkening
B&B Customers	Peter Parker (as representative)	Mary Jane Parker

Preparing for the Stakeholder Interviews

- Make an initial list of FR questions that:
  - Verifies the business owner's vision for that use case
  - Discovers the next level of detail
  - Provides scenarios of the use case
- For each stakeholder, determine the set of NFR questions to ask.

Detailed FR Questions

For each actor role ask these types of questions:

- Is the "xyz" use case necessary for your job?
- Explain the steps that you do to perform "xyz."
  - What data do you collect at each step?
  - What data is mandatory and what is optional?
  - If you currently use software to support "xyz," can you provide screen images.
- Can you give a concrete scenario of "xyz?"

Detailed FR Questions

- Does your job for “xyz” include generating any reports?
- Does your job require you to interact with any external systems for the “xyz” use case?
- Does your job use external data to perform the “xyz” use case?

Keep additional notes on:

- Differences in terminology between actors
- Differences in purpose and flow of the use case between actors

Requirements Elicitation Issues

The mental models of stakeholders are often inaccurate. There are three fundamental issues:

- Deletion – Information is filtered out
- Distortion – Information is modified by creation and hallucination
- Generalization – The creation of rules

Elicitation Issues: Deletion

Information is filtered out.

> Booking Agent: When I book a reservation, I record the room that is being booked.

Problem: The booking agent leaves out the fact that multiple rooms can be reserved.

Solution: Query the stakeholder for clarification.

> ACME: Is it possible to book more than one room?
>
> BookingAgent: Oh yes, I forgot about that. It does not happen very often, but it does happen. A few months ago, a couple got married at the Sonoma B&B. They booked four rooms for close family members.

Elicitation Issues: Distortion

Information is modified by creation and hallucination.

> Booking Agent: A reservation can be held without confirming the room with a credit card. But it is cancelled if the customer has not confirmed the reservation within two weeks.

Problem: The booking agent misleads the interviewer with incorrect information about the cancellation policy.

Solution: Query other stakeholders for validation and clarification.

Elicitation Issues: Generalization

General rules are created inappropriately.

> Receptionist: I always scan the customer’s credit card when they check in.

Problem: The receptionist is implying that scanning a credit card is *always* part of the check in procedure.

Solution: Query the stakeholder for clarification.

> ACME: Is there any situation in which the customer does not need a credit card for a reservation?

> Receptionist: Yes, I forgot. We occasionally get corporate customers that have confirmed the reservation with a company purchase order (PO).

Detailed NFR Questions

NFR questions are often much more ambiguous. It can be difficult to find the right person to ask. Here are a few starting points:

- Know what systemic qualities are important to the system
- Know whom to ask about these qualities
- Know what key phrases to listen for and what questions to ask to determine the NFRs for a given systemic quality

Systemic Qualities

NFRs determine the systemic qualities of the software system.
NFRs determine *how well* the system behaves at the boundary between the actor and the system.

Manifest	**Operational**	**Developmental**	**Evolutionary**
Performance	Throughput	Realizability	*Scalability*
Availability	Manageability	Planability	Maintainability
Usability	*Security*		Extensibility
	Serviceability		Flexibility
	Testability		Reusability
	Reliability		Portability

Performance

Issues of performance come up often in requirements interviews. Here is what to look for:

- Sometimes a user will mention how fast certain operations must occur.
- Managers of end users usually have predefined expectations about the speed of individual operations as well as the total time to complete a single use case.
- System administrators often know the expected throughput of the system and the capacity of various system resources (number of users, size of files, number of database records, number of transactions, and so on).

Performance

Users make performance-related statements, such as:

- "When I click this button the current system takes too long to respond; I need a response within 5 seconds."
- "This report takes a long time to generate, but that is OK because we batch it at night."

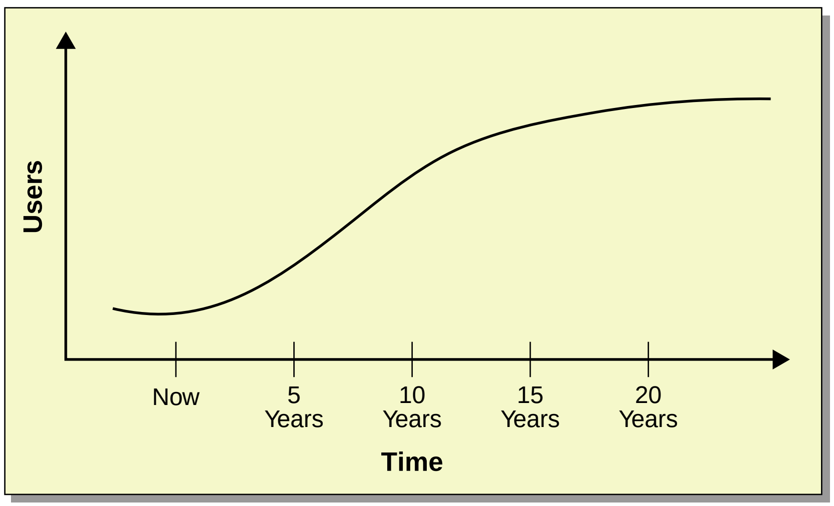
You can ask performance-related questions, such as:

- How fast does this operation need to be?
- At normal and peak times, how many transactions occur at the database?
- How many simultaneous users will be using the system?

Scalability

Scalability is related to performance, but it is the ability to increase throughput over time.

For example, a web application might need to support 20 simultaneous users this year, but will increase to 50 users within five years.



Scalability

Determining scalability is often guess work and depends largely on the projected growth of the business. Therefore, the business owners and managers often have the best perspective on growth issues.

However, system, network, and database administrators often keep track of trends in their areas:

- System load at average and peak times
- Network usage at average and peak times
- Average growth of database tables
- Planned hardware upgrades

Usability

This systemic quality is hard to quantify, but the goal is to determine the *ease of use* considerations of the system.

- The first thing to do is analyze the capabilities of the actors.
  - Some of this analysis can be gained by *observing* the actor during the requirement interviews and by observing them in the workplace.
  - You can also ask the actor's manager about their capabilities.
- The second thing to do is determine specific usability considerations based on the actor's capabilities.

Actor Information

Determine the following for each actor in the system:

- Role within the company
- Job description
- Primary use of the system
- Average length of use of the system at one time
- Average frequency of use of the system
- Education
- Domain expertise
- Experience with computers, target hardware, and OS
- Expected training
- Expected compliance

Usability Considerations

Following are a few usability considerations:

- Consistency of the proposed system with similar currently used systems
- Look and feel: button placement, spacing, multiple windows, tabbed windows, and so on
- Localization and internationalization
- Ease of navigation and workflow
- Accessibility

Security

Most applications, especially enterprise applications, require tight control over who has access to what features of the system.

- Actor roles usually define the initial security roles of the system.
- However, consider these other security issues:
  - How will user and password information be stored?
  - What external (and internal) threats might exist?
  - What granularity of control is necessary?
  - If there is a Web application, what critical data (passwords, credit card, and so on) is exchanged across the Internet?

Creating the SRS Document

The SRS document records the combined set of requirements of the software system.

The SRS document contains six sections:

- Introduction
- Constraints and Assumptions
- Risks
- Functional Requirements
- Non-Functional Requirements
- Project Glossary

The SRS should be detailed without losing clarity or focus.

Writing the Introduction

The introduction section includes:

- Purpose
- Scope
- System Context
- Primary Stakeholders
- Acronyms and Abbreviations
- How This Document is Organized
- Engineering Change Orders
- References

Writing the Functional Requirements

The Functional Requirements section includes the following subsections:

- Primary Features
- Actors
- Use Cases
- Applications
- Use Case Detailed Requirements

Actors Section

Provide a succinct summary of the actor's capabilities. For example:

> This person manages reservations over the phone for customers; as such, they are usually the customer's first point of contact with Bay View Property Management. This person is directly employed by Bay View Property Management.
>
> This person is not required to have an advanced degree (but high-school equivalency is assumed), but is required to be *familiar with operating within the Microsoft Window OS environment* and will have some *touch-typing skills*. This person will be trained on the properties owned by Bay View Property Management. This person will be trained on the system. *Training will be handled by Bay View Property Management*.

Actors Section

This job role works in two eight-hour shifts (6 a.m. to 2 p.m. and 2 p.m.to 10 p.m. PST) and will be using the System the whole time. *There is significant turnover in this job role* (on average, one person quits every six months). It is not suspected that this person is prone to circumvent the system, but this might occur due to inexperience. Therefore, *extra attention to the flow of the UI for this actor is necessary*.

Use Cases Section

The Use Case section provides a table of all use cases with their priority, unique number, and description. For example:

Use Case Name	Priority	Number	Description
Manage Reservation	E	1	This use case enables the Booking Agent to create, retrieve, update, and delete Reservations and Customers.
Check-in Customer	E	2	This use case enables the Receptionist to "check in" a Customer.
Create Promotions	H	7	This use case enables the Manager to create promotions and discounts.
Manage Events	H	8	This use case enables the Event Coordinator to create, retrieve, update, and delete Conference Events.
Send Survey	F	12	This use case enables a programmer to set the automatic time the System must send out Customer surveys.

Applications Section

The Applications section provides a table of all independent applications that comprise the system. For example:

Application Name	Description / Use Cases
HotelApp	This standalone application automates the main functions of managing a hotel and small event center. Supports UCs: E1, E2, E3, E6, H7, H8, F9, F10, F11, and F12
WebPresenceApp	This web site/application enables a Customer to view hotel properties and to book a reservation. Supports UCs: E4 and E5
KioskApp	This standalone application resides in a web-like kiosk in the lobbies of each property. Supports UC: F13

Detailed Requirements Section

The detailed requirements section is a complete list of detailed FRs that comprise each use case.

- Each FR is given an identifier based on the use case code. The use case code is the priority code prepended to the unique UC number.
- Each FR is given a description.
- For example:

FR	Requirement Description
E1-1	The System shall permit a Booking Agent to create, retrieve, update, and delete a reservation.
E1-2	A reservation shall hold one or more rooms for a single time period.

Writing a Detailed Requirement

An important element of a FR description is to use the word "shall." For example, "The Hotel Reservation System shall permit a Booking Agent to create, retrieve, update, but not delete a Customer."

Be as detailed as possible. Here are some areas to think about:

- Delineate data collected or needed for a use case.
- Delineate the relationships between objects or activities in the system.
- Delineate all strategies for accomplishing an activity.
- Delineate all constraints on an object or activity.

The Importance of Traceability

The detailed FRs define what activities the system must perform. Throughout the other workflows (analysis, design, implementation, and testing) it is important to verify that the system satisfies its requirements.

- Use the FR codes in UML annotations to label when a component satisfies the requirement.
- Use the FR codes in the source code comments to indicate the piece of code that satisfies the requirement.
- Use the FR codes in the test plans to show which tests verify that the requirement is satisfied.

Writing the Non-Functional Requirements Section

The non-functional requirements section is a complete list of detailed NFRs, grouped by systemic quality.

- Each NFR is given an identifier based on the use case code.
- Each NFR is given a description. For example:

NFR	**Requirement Description**
E1-101	Based on historical evidence, there are approximately 150 reservations per month per B&B property and about 1,000 reservations per month in the resort property. Therefore, the system shall support a minimum of 1,300 reservation records per month.
E1-102	The GUI interface to the HotelApp shall have a response-time of no more than 5 seconds for any user input. This measurement will be performed at the application server in order to eliminate any network variability.
E1-103	The HotelApp shall support at least five users, simultaneously, at each property.

Writing the Project Glossary

The glossary in the SRS should include:

- Domain terms
- Synonyms
- Technology terms
- Software development terms

Summary

- Gathering requirements is essential to a successful software project because the requirements specify the behavior of the system to be implemented.
- Requirements come from many sources.
- Interviews with stakeholders are the primary means of requirements gathering.
- The SRS document expands upon the Vision document and provides detailed requirements.
- The SRS includes: user profiles, use cases, detailed FRs, constraints, risks, detailed NFRs, and the project glossary.

Module 6

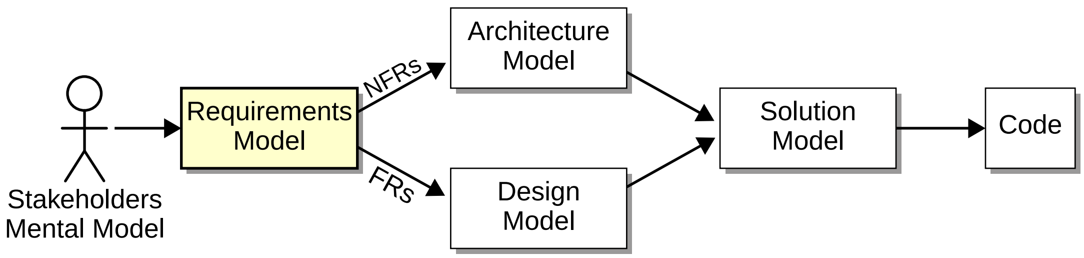
Creating the Initial Use Case Diagram

Objectives

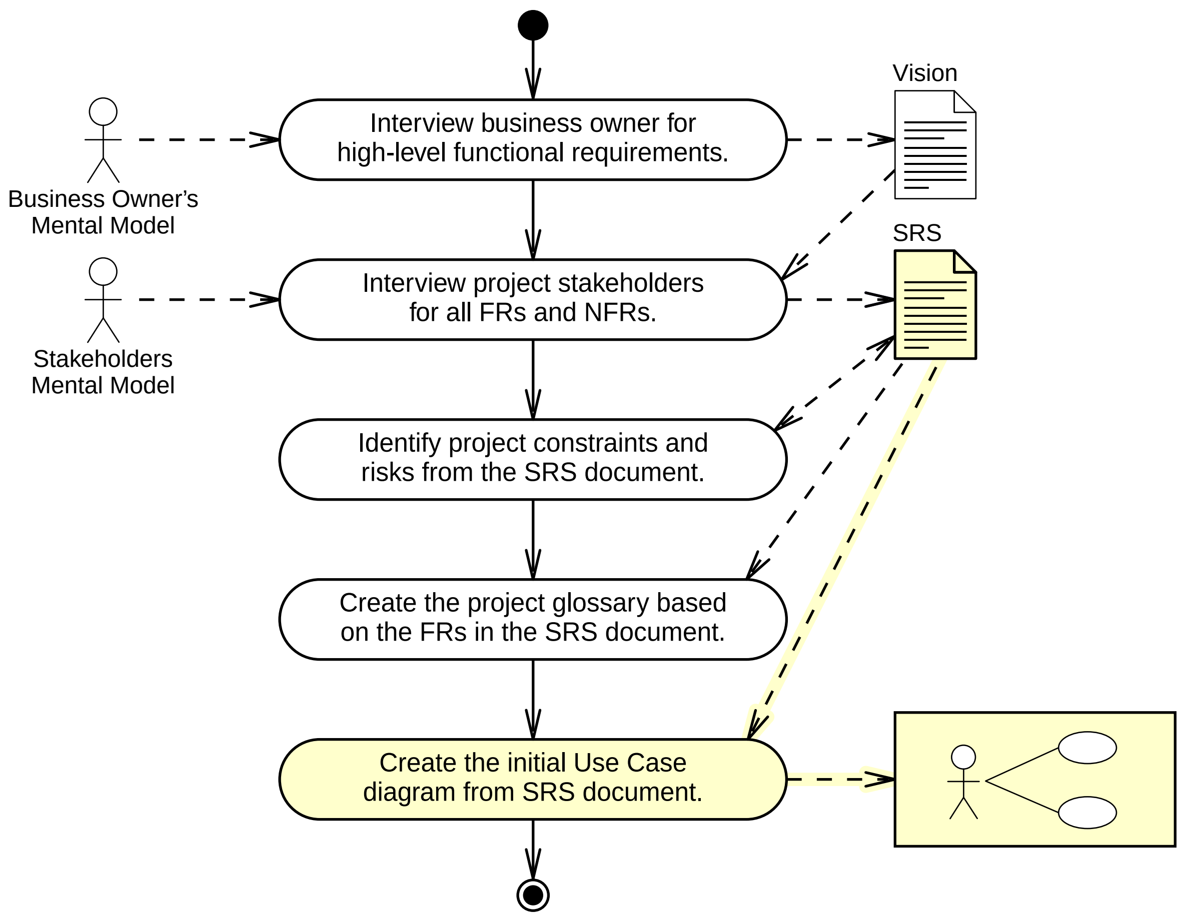
Upon completion of this module, you should be able to:

- Identify and describe the essential elements in a UML Use Case diagram
- Develop a Use Case diagram for a software system based on the SRS
- Record use case scenarios for architecturally significant use cases

Process Map



Stakeholders Mental Model
Requirements Model
NFRs
FRs
Architecture Model
Design Model
Solution Model
Code



Business Owner's Mental Model
Stakeholders Mental Model
Interview business owner for high-level functional requirements.
Interview project stakeholders for all FRs and NFRs.
Identify project constraints and risks from the SRS document.
Create the project glossary based on the FRs in the SRS document.
Create the initial Use Case diagram from SRS document.
Vision
SRS

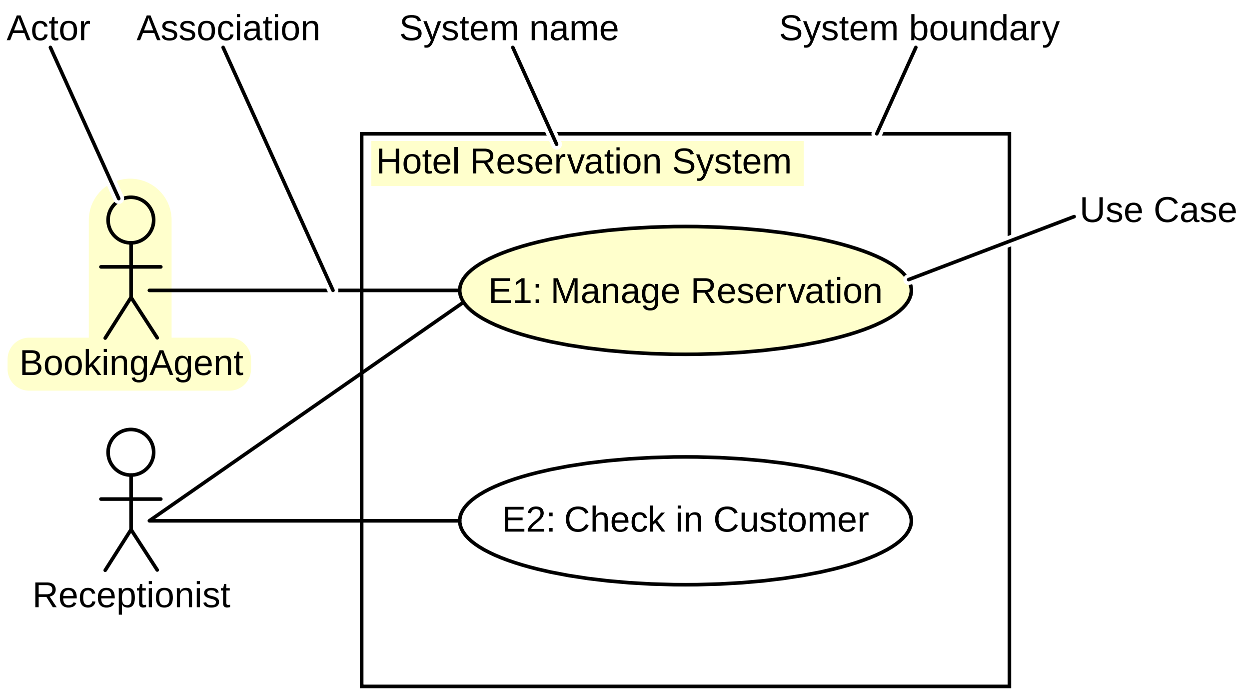
Justifying the Need for a Use Case Diagram

Following are reasons a Use Case diagram is necessary:

- The SRS is filled with detailed requirements. Also, the SRS is predominately text-based.
- The client-side stakeholders need a big picture view of the system.
- The system's use cases form the basis for which all development is focused.

Identifying the Elements of a Use Case Diagram

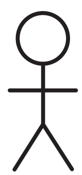
A Use Case diagram is “A diagram that shows the relationships among actors and use cases within the system.” (UML v1.4 spec page B-21)



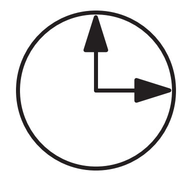
Actors

“An actor is a role that a user plays with respect to the system.” (Fowler UML Distilled page 42)

An actor is also “A coherent set of roles that users of use cases play when interacting with these use cases.” (UML v1.4 spec. page B-3)



BookingAgent



Time

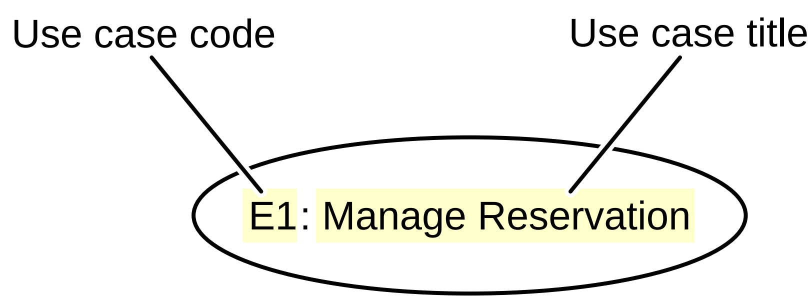
This icon represents a human actor (user) of the system.

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

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

Use Cases

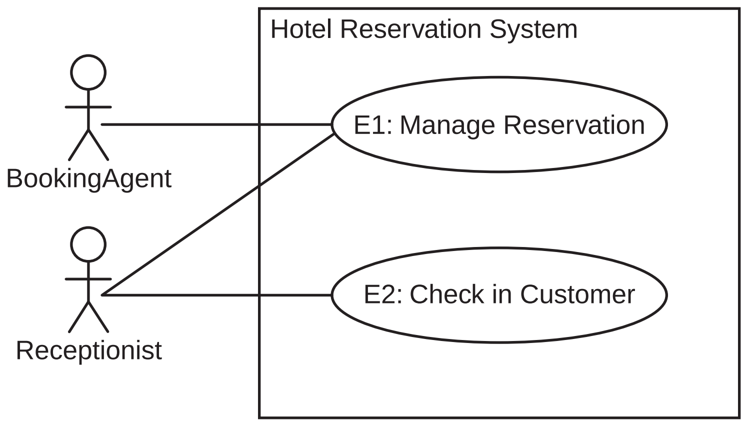
A use-case describes an interaction between an actor and the system to produce a result of value.

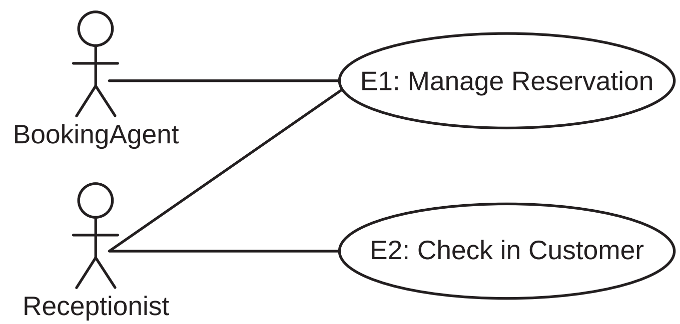


- A use case encapsulates a major piece of system behavior with a definable outcome.
- A use case is represented as an oval with the use case title in the center.
- The use case code can be used in front of the title for quick reference back to the SRS.

System Boundary

"The use cases may optionally be enclosed by a rectangle that represents the boundary of the containing system." (UML v1.4 spec. page 354)



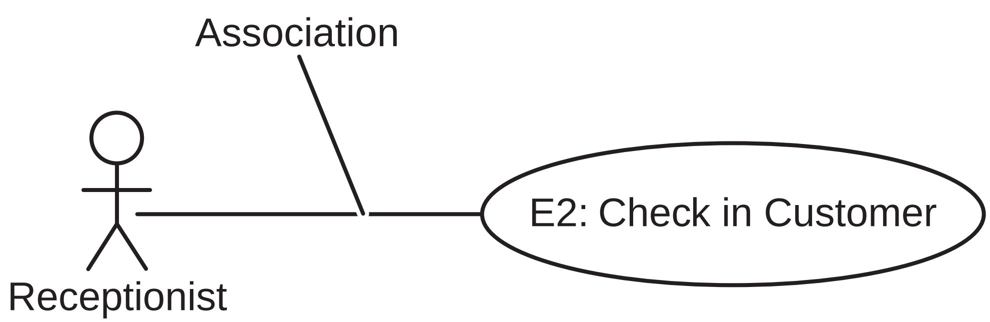


The system boundary box is optional.

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.

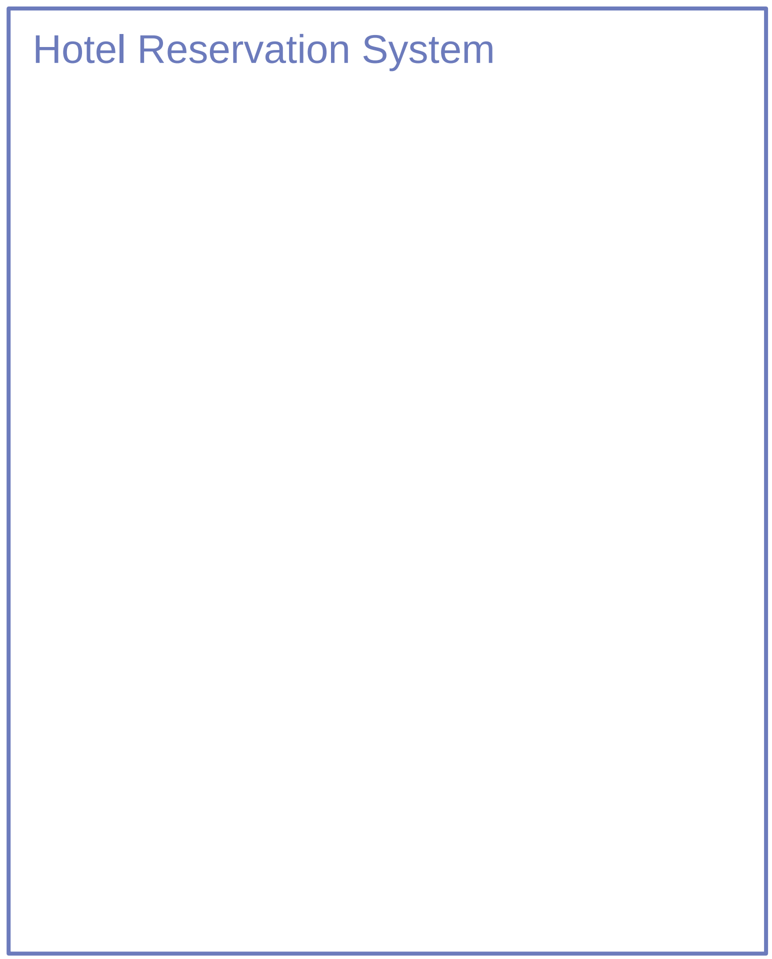
Developing a Use Case Diagram

The use case for a system represents all high-level behaviors (use cases) of the system and which actors that can participate in these behaviors.

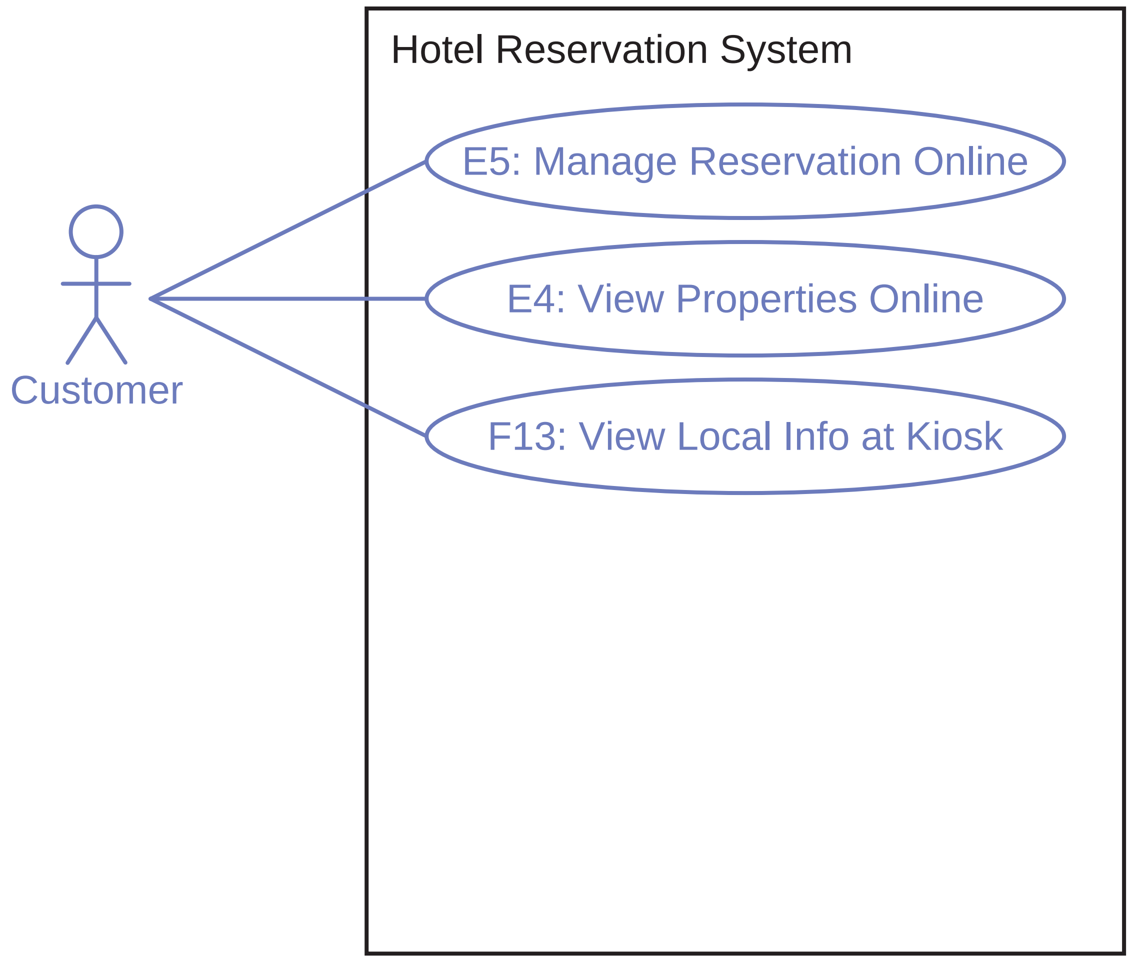
To create the Use Case diagram, follow these steps:

1. Create and name the system boundary rectangle.
2. Identify all actors of the system from the SRS.
3. For each actor:
   a. Add the actor icon to the diagram.
   b. Add use cases to the diagram in which the actor participates.
   c. Draw the use case associations to that actor.

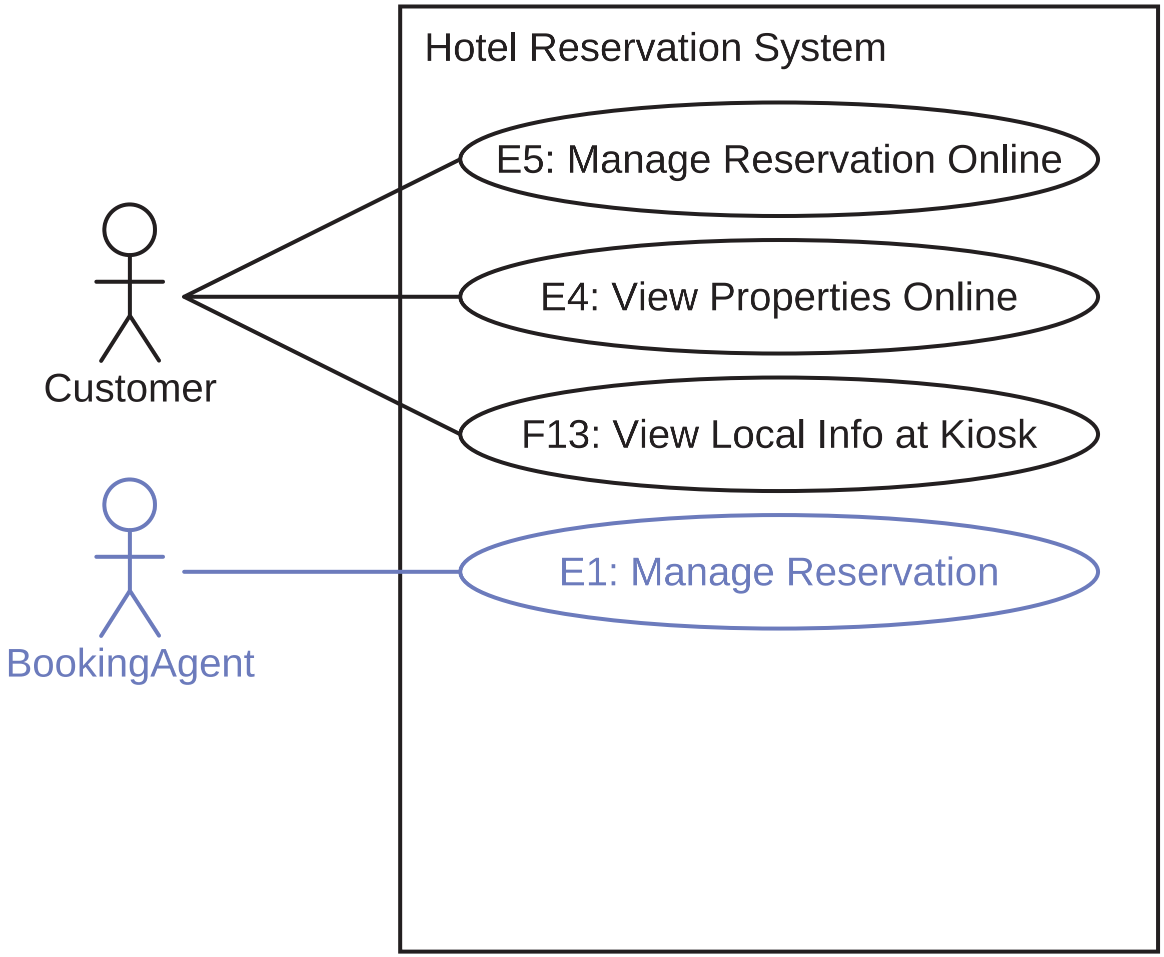
Create the System Boundary



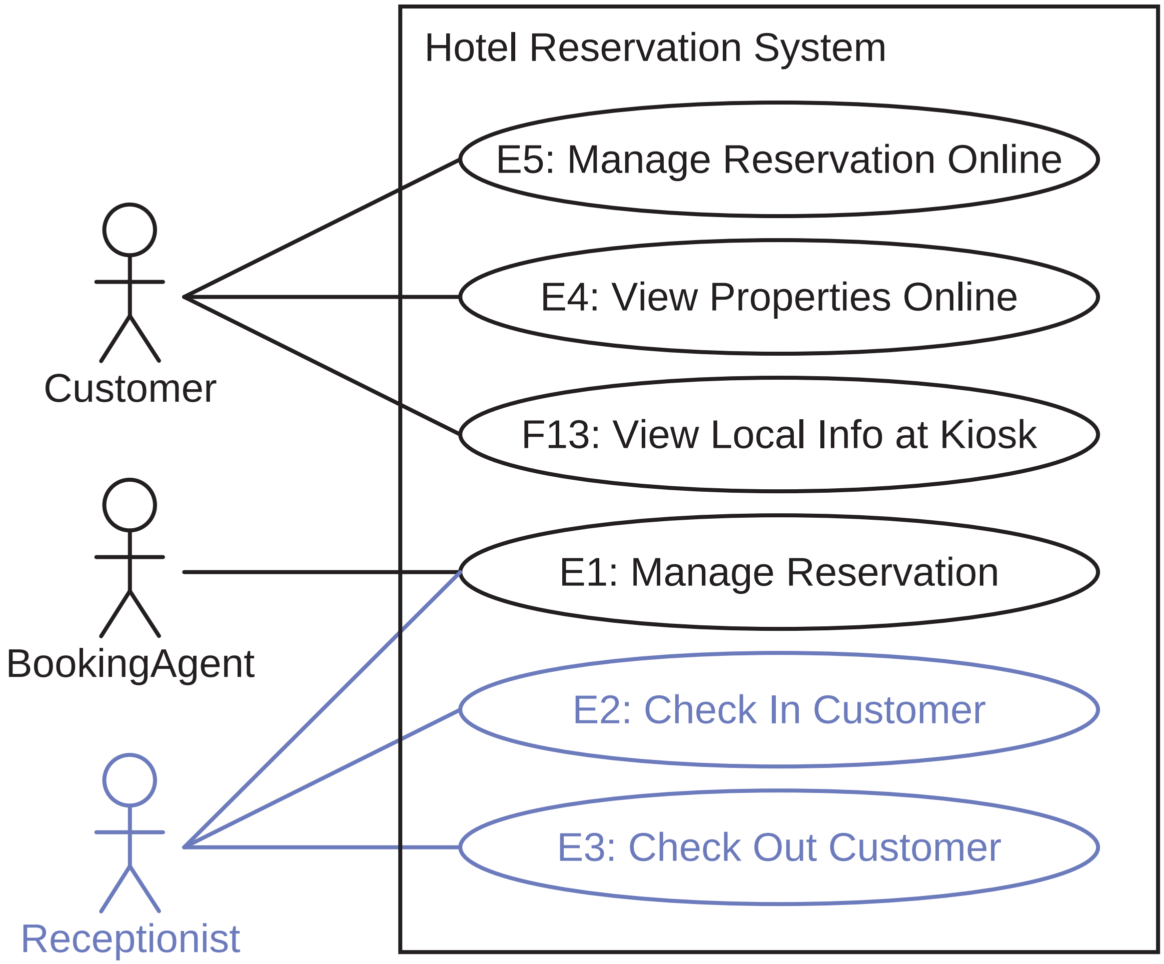
Add the Customer Actor and Use Cases



Add the Booking Agent Actor



Add the Receptionist Actor



Storing the Use Case Diagram

The Use Case diagram can be placed in the SRS.

- The Use Case diagram provides a visual representation of the Functional Requirements section of the SRS.
- Keeping the Use Case diagram within the SRS facilitates keeping these two artifacts synchronized.
- The main purpose of the Use Case diagram is to provide a succinct view of behavior of the system for the client.

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 scenarios record successful results.
- Secondary 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.

Use Case Scenario Example

The beginning:

Medoca Sansumi, booking agent for the Santa Cruz B&B, is waiting for a call and has the HotelApp Main screen displayed. A telephone call comes in from Ms. Jane Googol, a customer from New York city. “Hello, this is Jane Googol. I would like to make a reservation for New Year’s Eve,” says Ms. Googol. Medoca selects the “Create Reservation” function on the main screen of the HotelApp. An empty reservation form appears.

Use Case Scenario Example

The middle:

"When will you arrive?" asks Medoca. "December 31st," says Jane, "and I would like to stay through January 5th." Medoca enters the dates in the form. "What type of room would you like?" Medoca asks. "I will be with my husband so a single room will be sufficient. Is the Blue room available?" Jane asks. Medoca selects "single" in the reservation form and performs the search. The system responds with three available rooms: Victoria, Blue, and Queen. "Yes, it is," replies Medoca. Medoca selects the Blue room and the system populates the reservation form, and marks the reservation as "held."

Use Case Scenario Example

More of the middle:

Medoca enters Jane's full name into the system. Ms. Googol is an existing customer, so the system responds by populating the customer fields in the reservation form. "Would you like to confirm this reservation today?" asks Medoca. "Yes," says Jane, "use my VISA card number 1111-2222-3333-4444." Jane pauses as Medoca types this in. "The expiration date is July, 2004." Medoca enters this information, and selects "Verify Payment" on the system. After about five seconds, the system responds that the credit is verified. The system changes the state of the reservation to "confirmed."

Use Case Scenario Example

The end:

Medoca tells Jane the reservation ID (supplied by the system) and asks, “Is there anything else I can do for you today?” Jane replies no and Medoca thanks her and says goodbye. Medoca closes the reservation form window, which returns her to the Main HotelApp screen.

Storing the Use Case Scenarios

The use case scenarios, which can be quite lengthy, are usually stored in a document separate from the SRS.

The SRS should be updated to reference these Use Case scenario documents as they are written.

Summary

- A Use Case diagram provides a visual representation of the big-picture view of the system.
- The Use Case diagram represents the system, the actors that use the system, the use cases that provide a behavior with definable result for an actor, and the associations between actors and use cases.
- 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.

Module 7

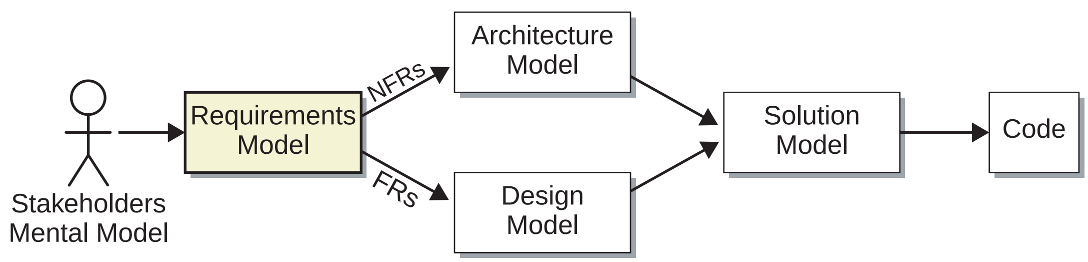
Refining the Use Case Diagram

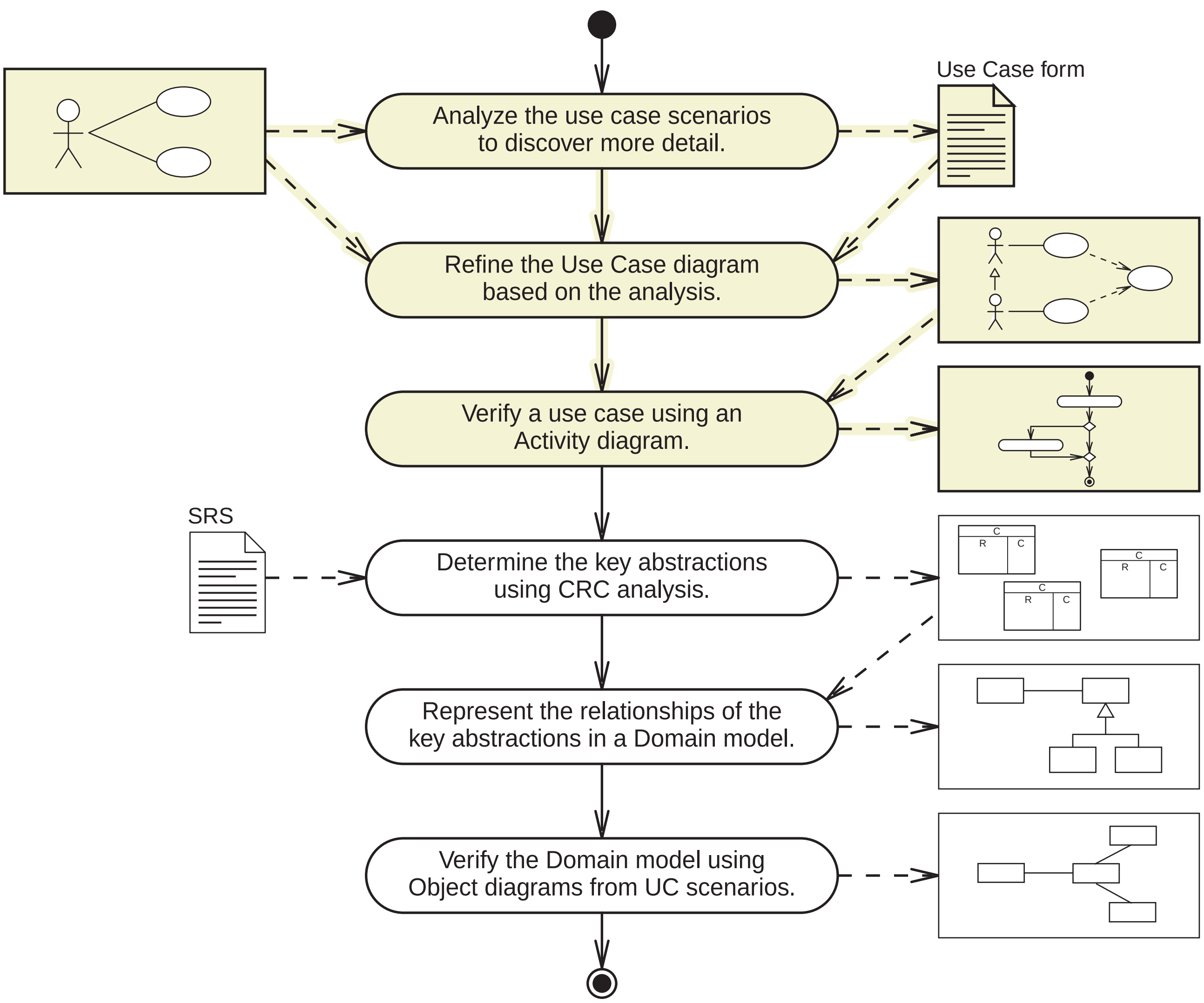
Objectives

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

- Document a use case and its scenarios in a Use Case form
- Recognize and document use case and actor inheritance
- Recognize and document use case dependencies
- Identify the essential elements in an Activity diagram
- Validate a use case with an Activity diagram

Process Map


Stakeholders
Mental Model
Requirements
Model
NFRs
FRs
Architecture
Model
Design
Model
Solution
Model
Code


Use Case form
Analyze the use case scenarios
to discover more detail.
Refine the Use Case diagram
based on the analysis.
Verify a use case using an
Activity diagram.
SRS
Determine the key abstractions
using CRC analysis.
Represent the relationships of the
key abstractions in a Domain model.
Verify the Domain model using
Object diagrams from UC scenarios.

Analyzing a Use Case

- Creating the Use Case form
- Recognizing inheritance patterns
- Recognizing use case dependencies

Use Case Forms

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 ID and Name	The code and name of the use case from the SRS.
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	The Essential, High-value, or Follow-on ranking of the use case from the SRS.
Risk	A High, Medium, or Low ranking of this use case's risk factors.

Use Case Forms

Form Element	Description
Pre-conditions and assumptions	The conditions under which the use case can be invoked.
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.
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 SRS.

Creating a Use Case Form

Perform these steps to determine the information for the Use Case form:

1. Fill in values specified in the SRS document.
2. Determine the pre-conditions from the scenarios.
3. Determine triggers from the scenarios.
4. Determine the flow of events from the primary scenario.
5. Determine the alternate flows from the secondary scenarios.
6. Determine the post-conditions.

Step 1– Fill in Values Specified in the SRS Document

Fill in elements specified in the SRS document:

Use Case ID and Name	E1: Manage Reservation
Description	Use case for creating a new reservation at a lodging property.
Actors	Primary: Booking Agent Secondary: Receptionist, Manager, and Owner
Priority	Essential
Non-Functional Requirements	E1-102 (performance) E1-105 (scalability) E1-108 (reliability)

Step 2 – Determine the Pre-Conditions From Scenarios

The first paragraph in a good Use Case scenario should describe the state of the system before the use case begins. These are the pre-conditions.

> Medoca Sansumi, booking agent for the Santa Cruz bed and breakfast, is waiting for a call and has the HotelApp Main screen displayed.

Pre-condition	BookingAgent is waiting for a call. The Main screen of the HotelApp is displayed.

Step 3 – Determine the Trigger From Scenarios

This opening paragraph of the scenario should also state how the actor knows to initiate the particular use case.

> A telephone call comes in from Ms. Jane Googol, a customer from New York city. "Hello, this is Jane Googol. I would like to make a reservation for New Year's Eve," says Ms. Googol.

Trigger	A call comes in from a customer who asks to make a new reservation.

Step 4 – Determine the Flow of Events From the Primary Scenario

Determine the flow of events from the primary scenario.

> "When will you arrive?" asks Medoca. "December 31st," says Jane, "and I would like to stay through January 5th." Medoca enters the dates in the form. "What type of room would you like?" Medoca asks. "I will be with my husband so a single room will be sufficient. Is the Blue room available?" Jane asks. Medoca selects "single" in the reservation form and performs the search.

Flow of Events	... 3. The Receptionist enters search criteria. 4. The Receptionist searches the Room Schedule for available rooms. 5. The system displays the rooms that are available during the date range. 6. The Receptionist selects a room. ...

Step 5 – Determine the Alternate Flows From the Secondary Scenarios

Determine the alternate flows from the secondary scenarios:

- Perform a *difference analysis* between the primary scenario and each secondary scenario (in turn).
- The alternate flows are the steps that are different between the primary and secondary scenario.

Alternate Flows	In Step 5, if no rooms are available, then the BookingAgent prompts the customer for either a different type of room or a different date range and returns to Step 3.

Step 6 – Determine the Post-Conditions

The last paragraph in a good use case scenario should describe the state of the system at the end of the use case. These are the post-conditions.

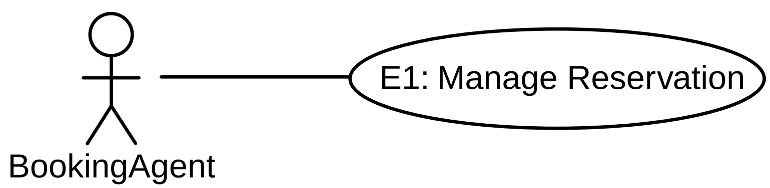
> Medoca tells Jane the reservation ID (supplied by the system) and asks, "Is there anything else I can do for you today?" Jane replies no and Medoca thanks her and says goodbye. Medoca closes the reservation form window, which returns her to the Main HotelApp screen.

Post-Conditions	The reservation has been saved to a database. The reservation form is closed and the Main screen is visible.

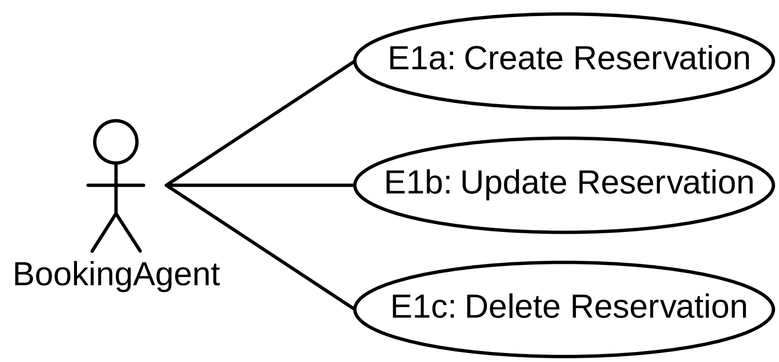
Expanding High-Level Use Cases

Some use cases captured during Requirements Gathering are too high-level. In these situations it is useful to introduce new use cases that separate the workflows.

Example:



Becomes:

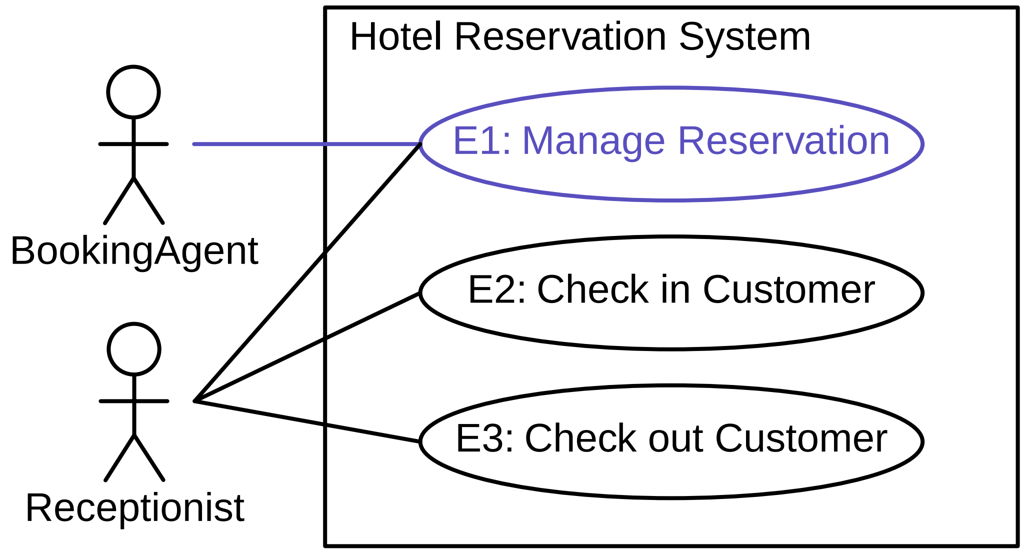


Expanding High-Level Use Cases

- 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.
- Also, if several scenarios have a different starting point, these scenarios might represent different use cases.

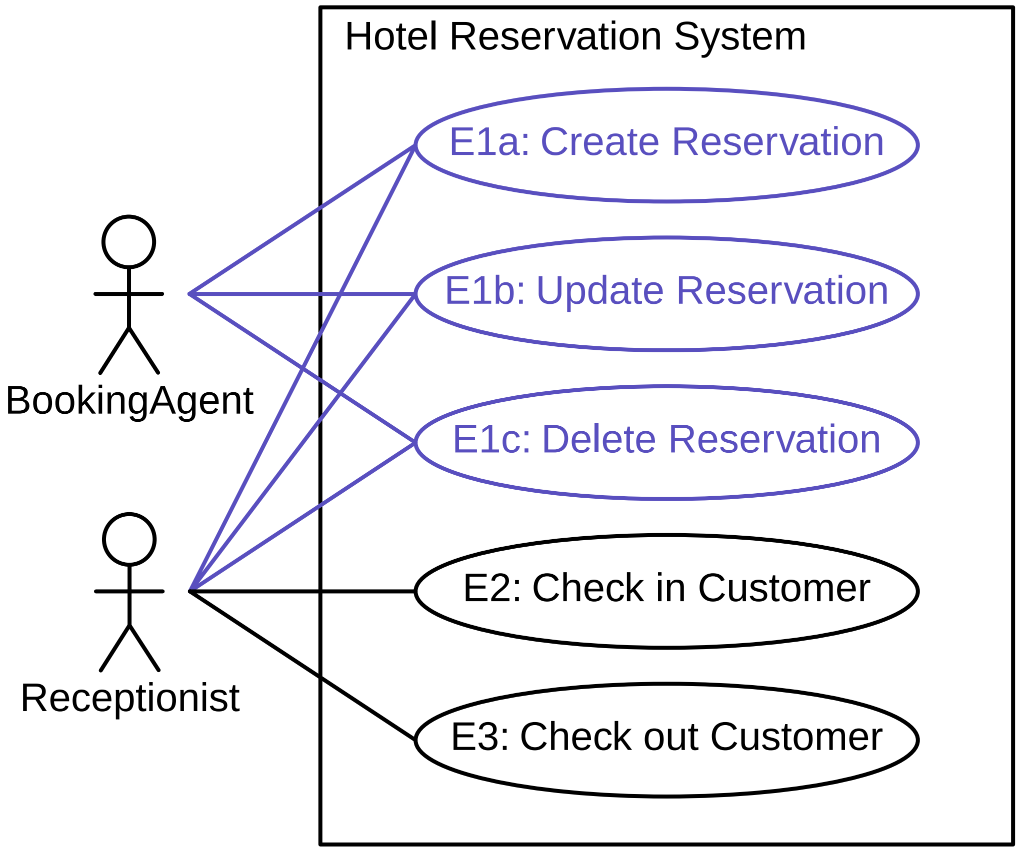
Expanding High-Level Use Cases

Use Case diagram before refinement:



Expanding High-Level Use Cases

The expanded version creates many more associations:



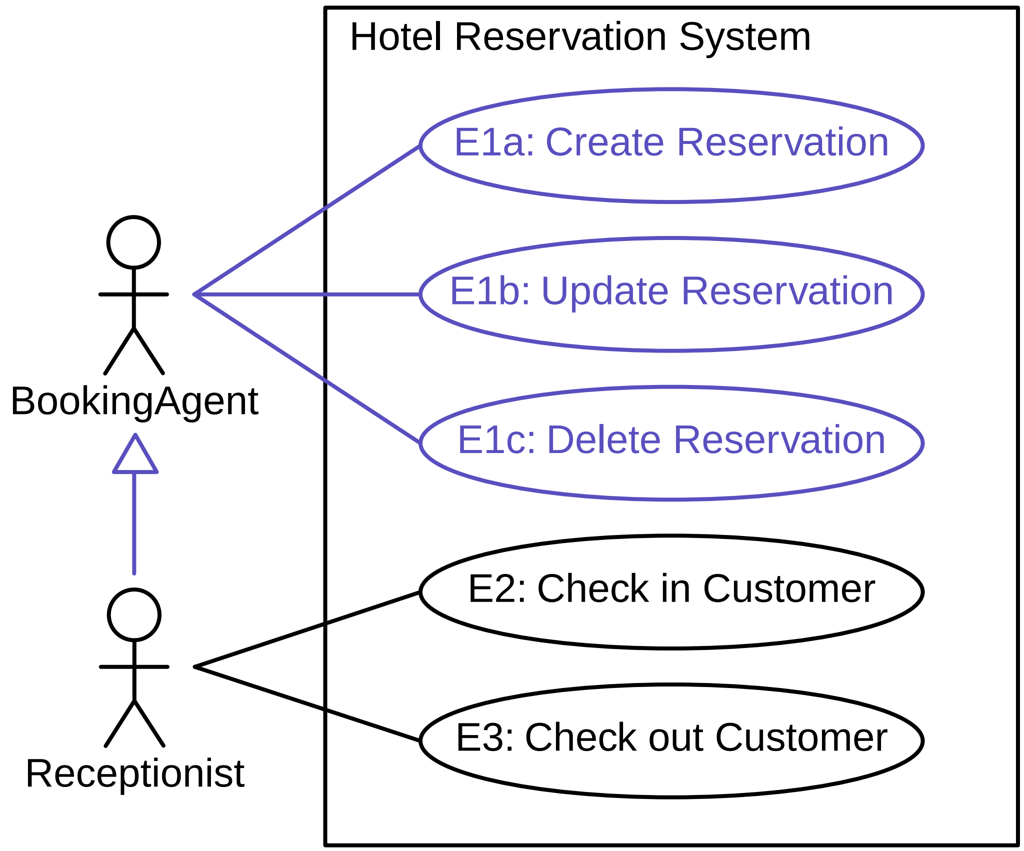
Analyzing Inheritance Patterns

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.

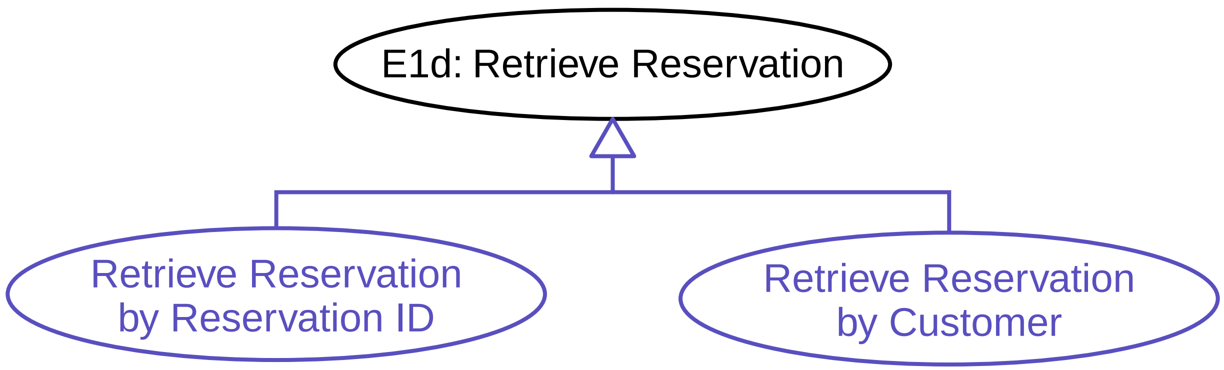
Actor Inheritance

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



Use Case Specialization

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



Use case specializations are *usually* identified by variations in the use case scenarios.

Analyzing Use Case Dependencies

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

- One use case (a) *includes* another use case (b).

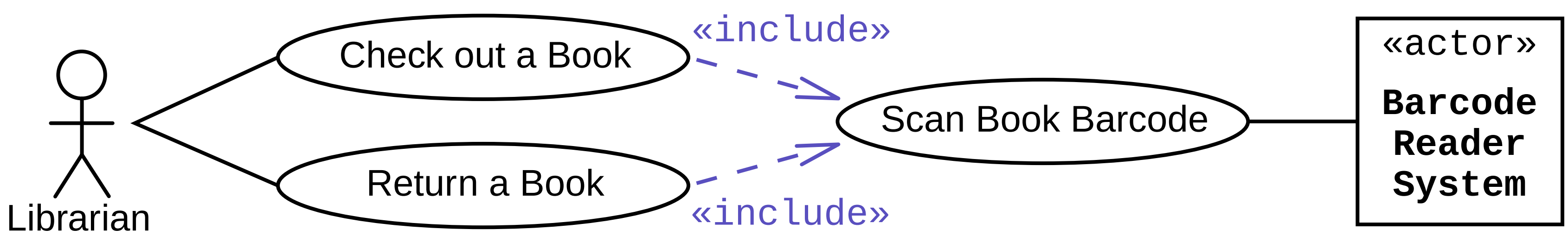
  This means that the one use case (a) requires the functionality of the other use case (b) and *always* performs the included use case.
- One use case (a) can *extend* another use case (b).

  This means that the one use case (a) can (optionally) use the functionality of another use case (b) so extend 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.

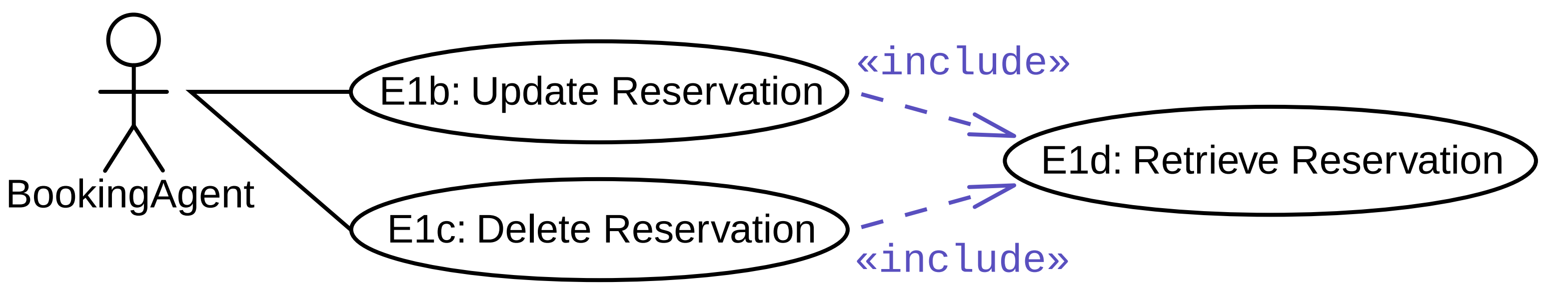
This dependency is drawn like this:



The «include» Dependency

Identifying and recording common behavior:

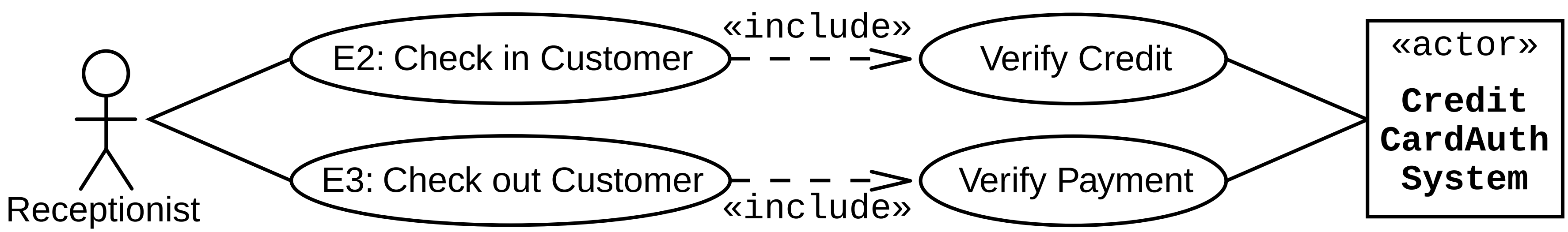
- Review the use case scenarios for common behaviors.
- Give this behavior a name and place it in the Use Case diagram with an «include» dependency.



The «include» Dependency

Identifying and recording behavior associated with an external system.

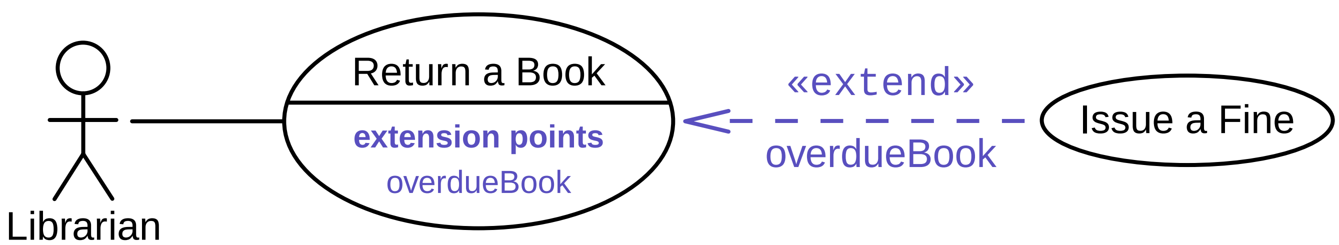
- Review the use case scenarios for sequences of behavior that involves an external system.
- Give this behavior a name and place it in the Use Case diagram with an «include» dependency.



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.

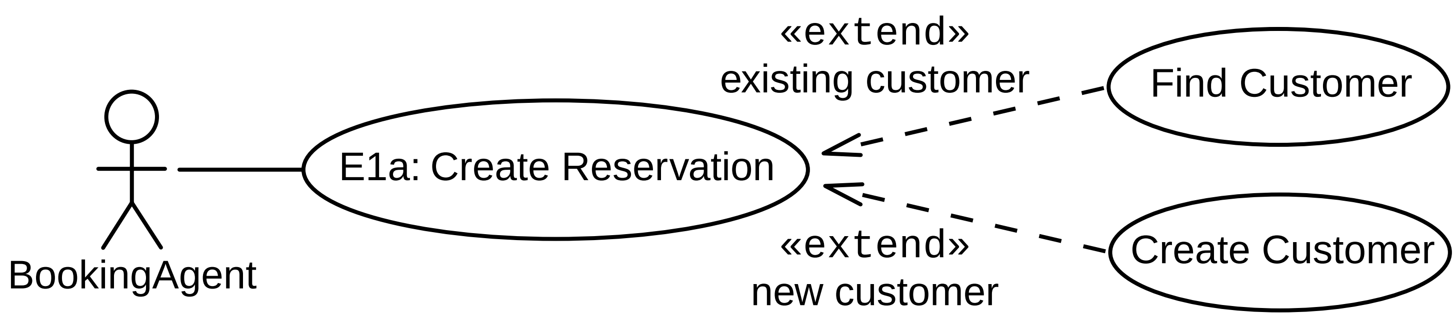
This dependency is drawn like this:



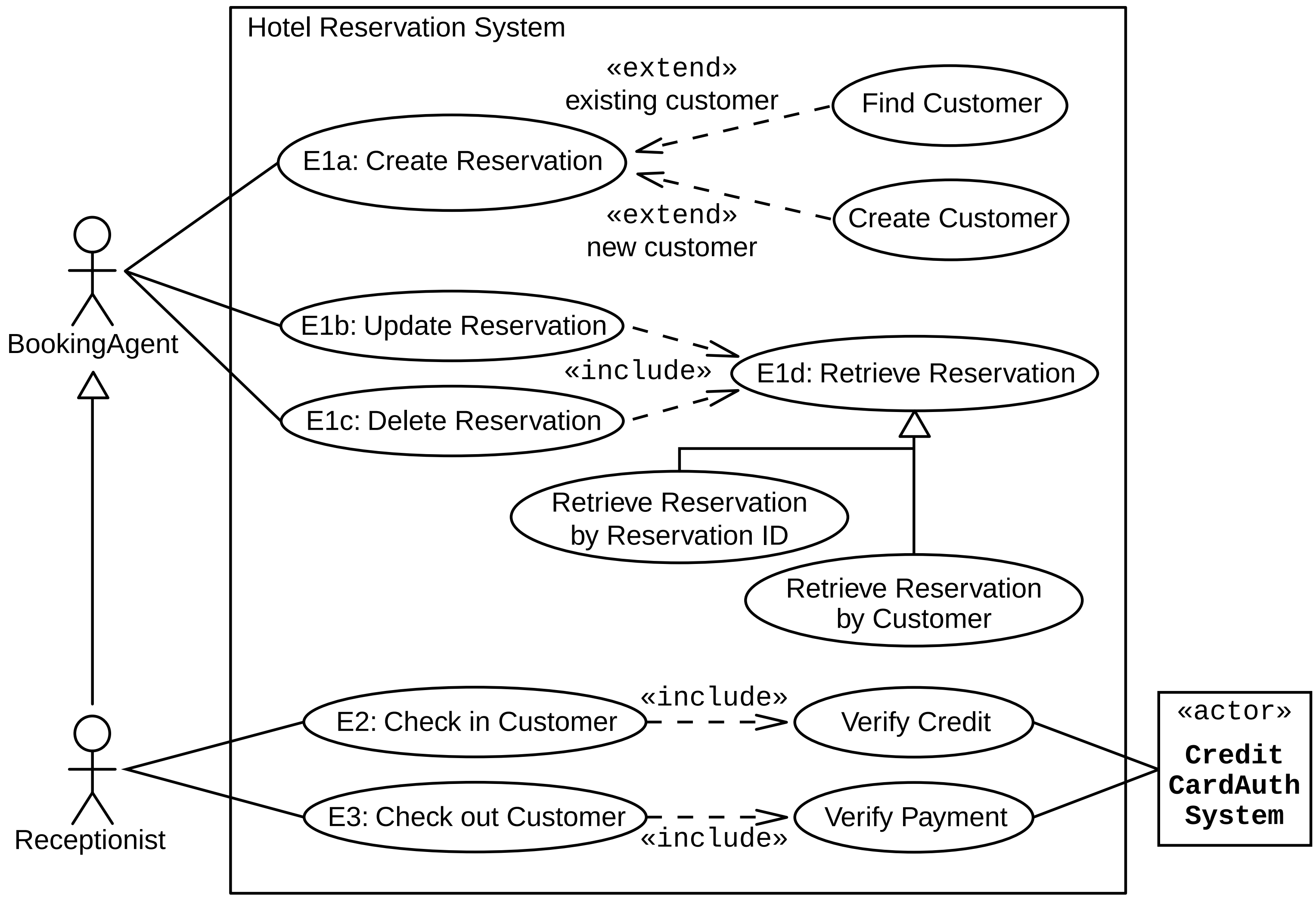
The «extend» Dependency

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 From the Hotel Reservation System

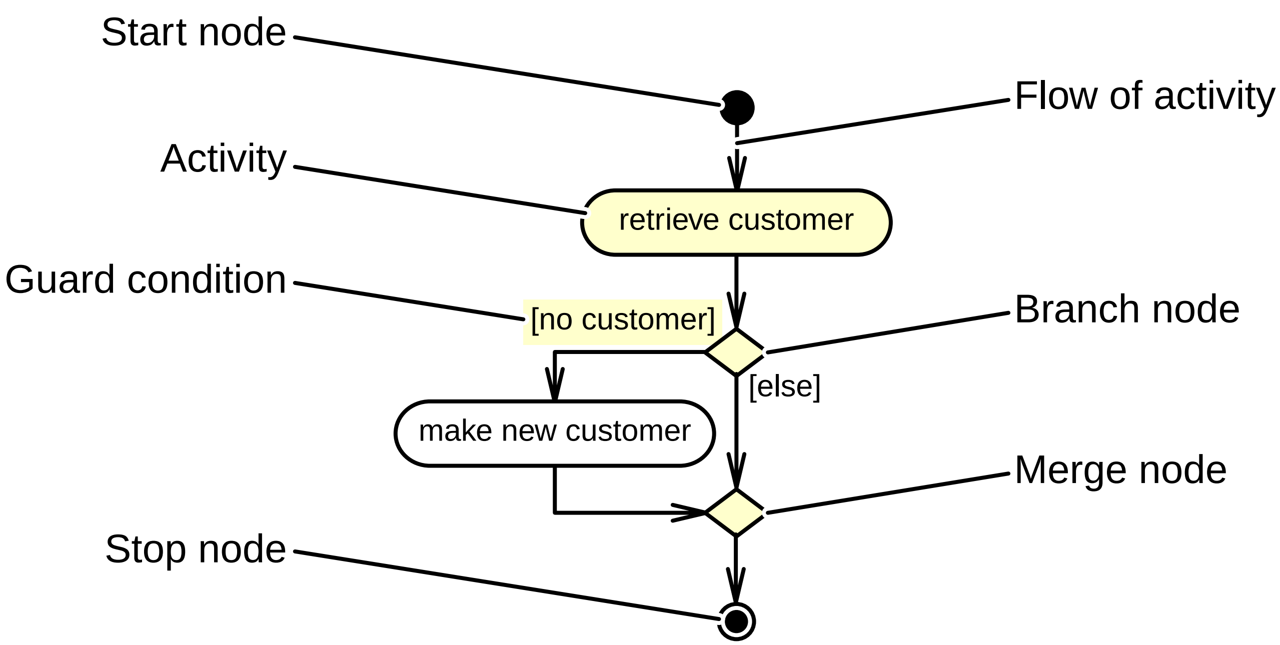


Validating a Use Case With an Activity Diagram

- Represent the Flow of Events of the use case in an Activity diagram
- Validate the use case by reviewing the Activity diagram with the stakeholders

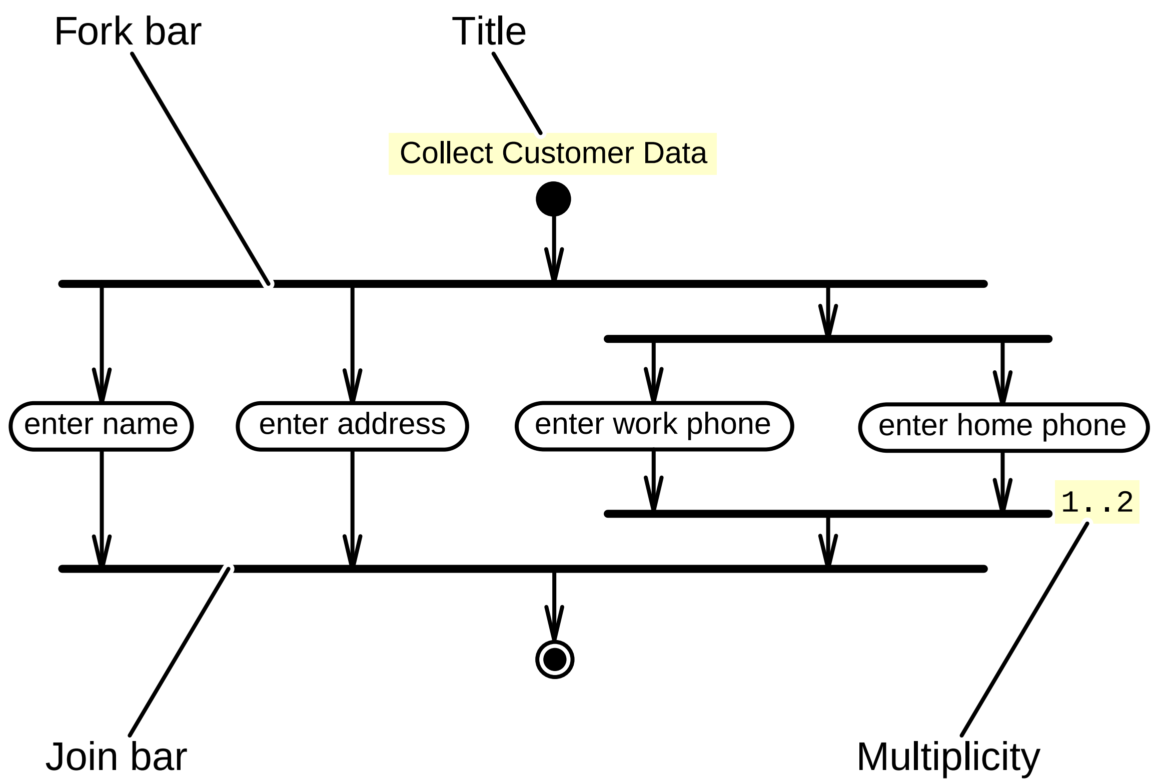
Identifying the Elements of an Activity Diagram

An Activity diagram is composed of the following elements:



Identifying the Elements of an Activity Diagram

An example of concurrent activities:



Activities

An activity is any process or action taken by the system or an actor.

- An activity can be written in a natural language.

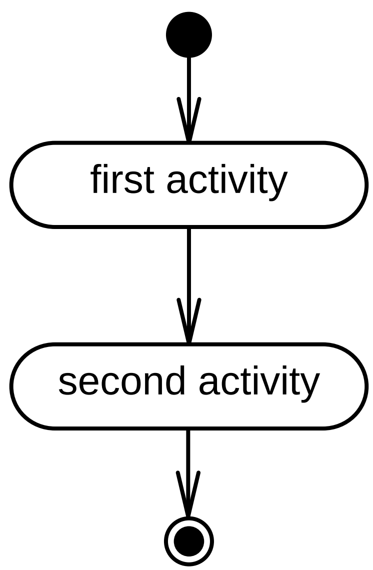
  For example:
  `retrieve customer`
- An activity can be written in pseudo-code.

  For example:
  `cust = retrieve customer`
- An activity can be written in a specific programming language.

  For example:
  `cust = customerSvc.getCustomer(custID);`

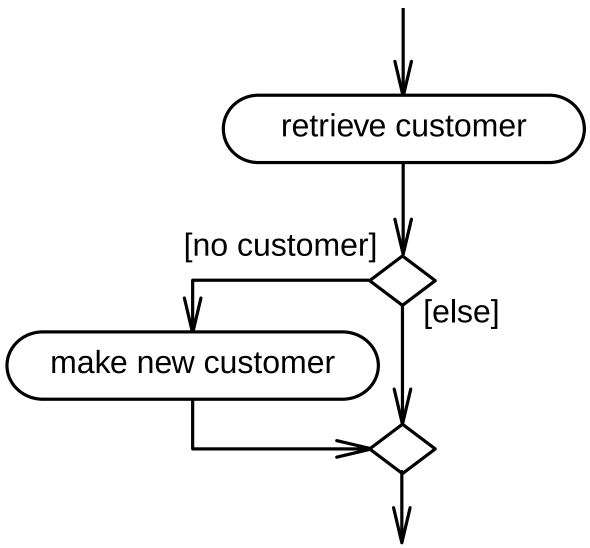
Flow of Control

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

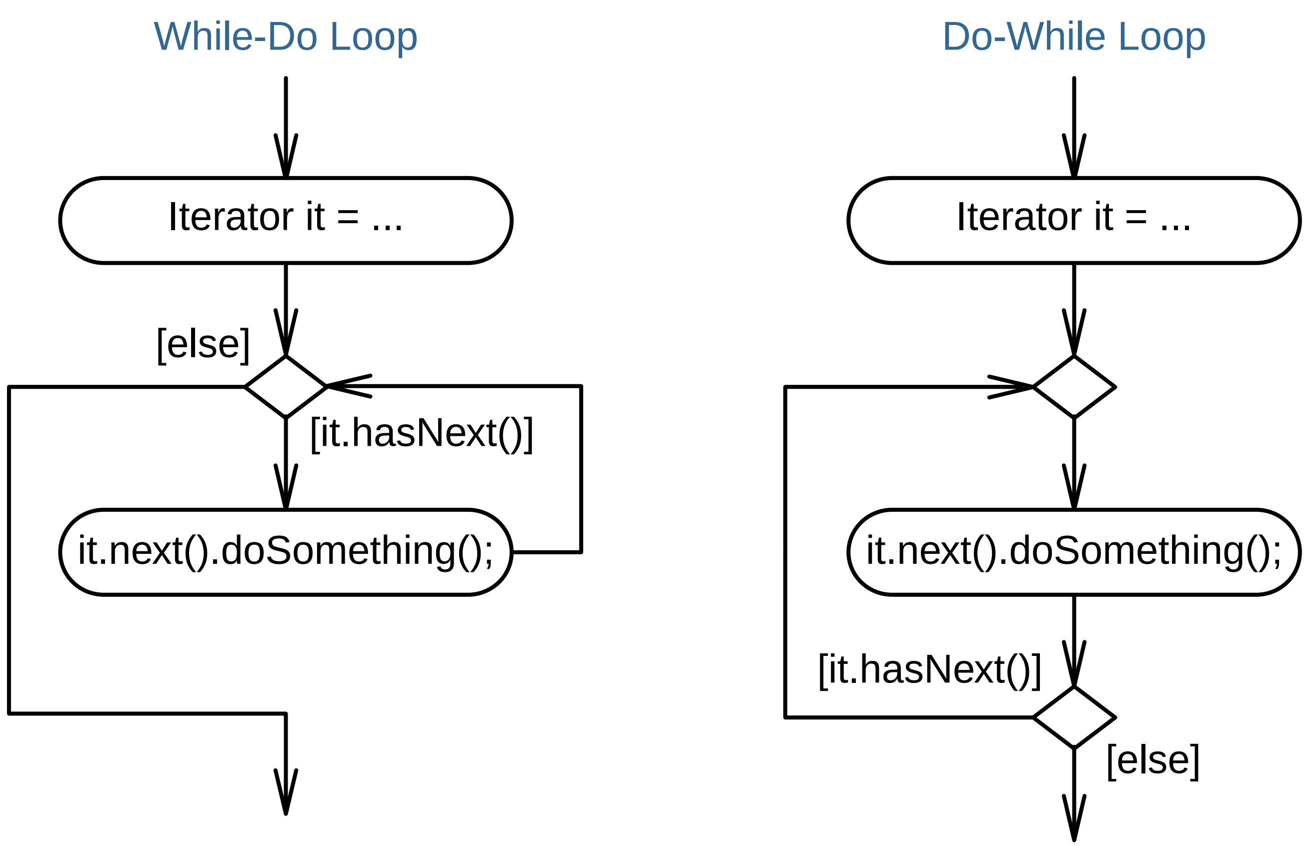
The branch and merge nodes represent conditional flows of activity.



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

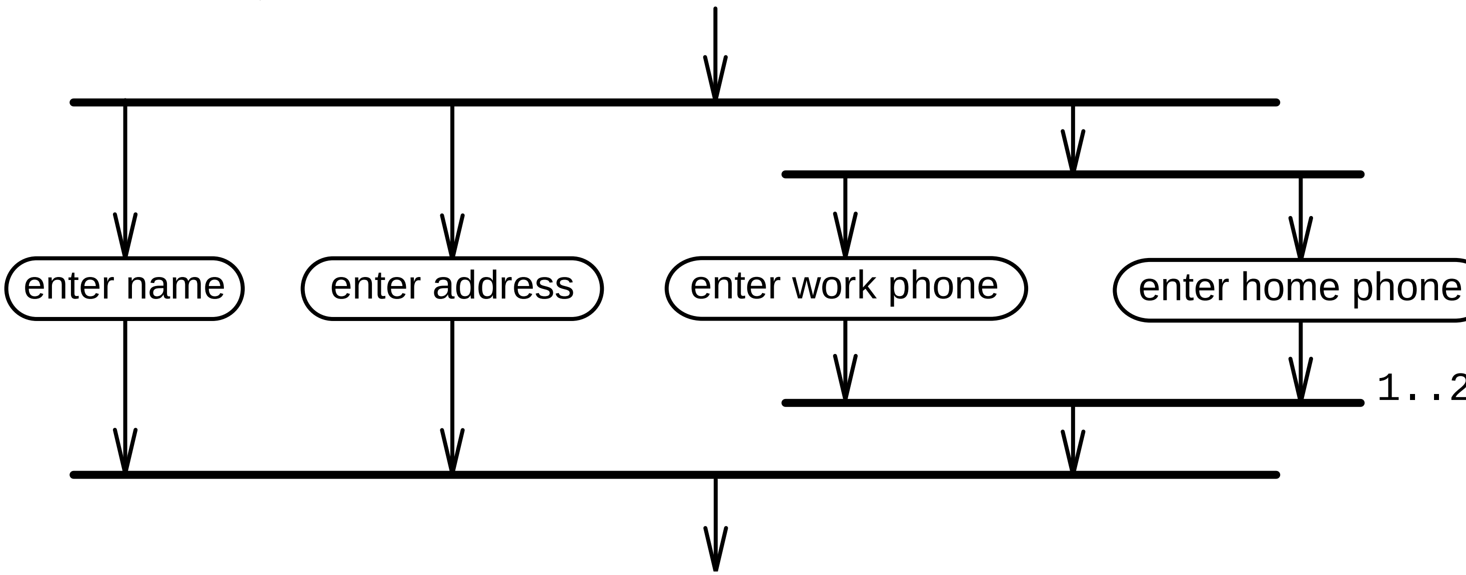
Iteration

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.

Creating an Activity Diagram for a Use Case

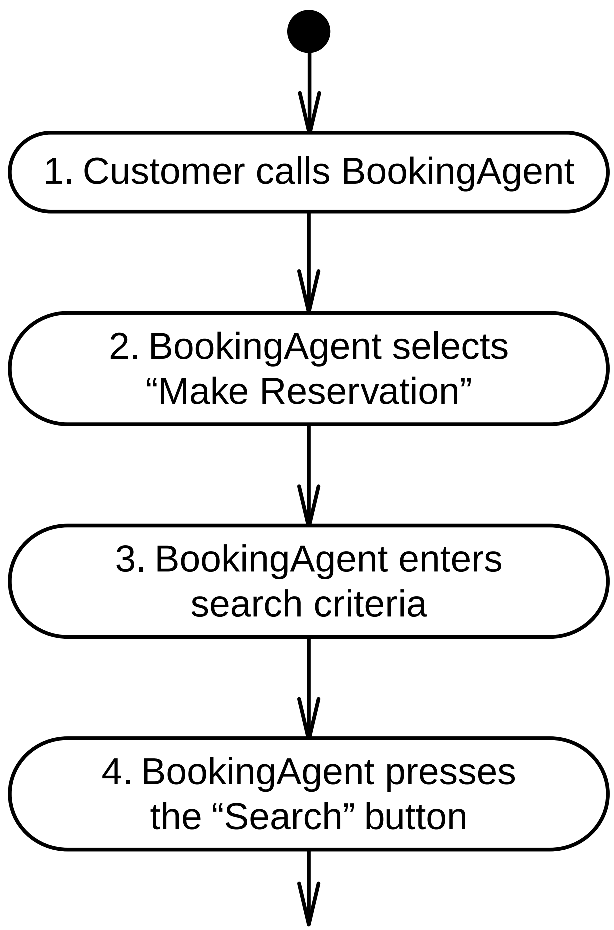
Analyze the Flow of Events field in the Use Case form:

- Identify activities
- Identify branching and looping
- Identify concurrent activities
- Create the Activity diagram

Use Case Activities

Each element in the "Flow of Events" becomes an activity:

1. Customer calls BookingAgent
2. BookingAgent select "Make Reservation" icon
3. BookingAgent enters search criteria
  3.1 BookingAgent enters arrival and departure dates
  3.2 BookingAgent enters type of room
4. BookingAgent presses the "Search" button
...
11. BookingAgent enters customer name
12. BookingAgent presses the "Search" button
13. If a customer match is not found
  13.1 BookingAgent enters address info
  13.2 BookingAgent enters phone info
  13.3 BookingAgent presses "Add New Customer"
14. Else
  14.1 The system display match list
  14.2 BookingAgent selects the correct customer
  14.3 The System populates the GUI with customer info
...
21. The System saves the reservation and displays ResvID



Branching

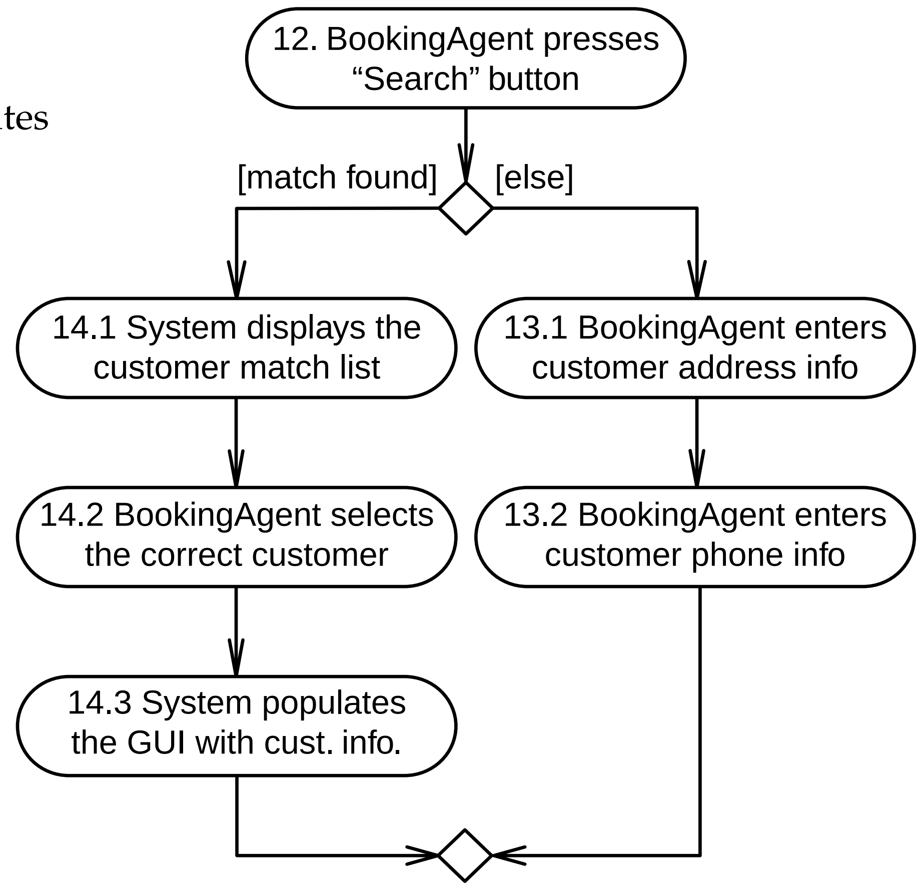
Identify conditional branches in the Flow of Events:

1. Customer calls BookingAgent
2. BookingAgent select "Make Reservation" icon
3. BookingAgent enters search criteria
   3.1 BookingAgent enters arrival and departure dates
   3.2 BookingAgent enters type of room
4. BookingAgent presses the "Search" button

...

11. BookingAgent enters customer name
12. BookingAgent presses the "Search" button
13. If a customer match is not found
   13.1 BookingAgent enters address info
   13.2 BookingAgent enters phone info
   13.3 BookingAgent presses "Add New Customer"
14. Else
   14.1 The system display match list
   14.2 BookingAgent selects the correct customer
   14.3 The System populates the GUI with customer info

...



Concurrent Flow

Identify concurrent activities in the Flow of Events:

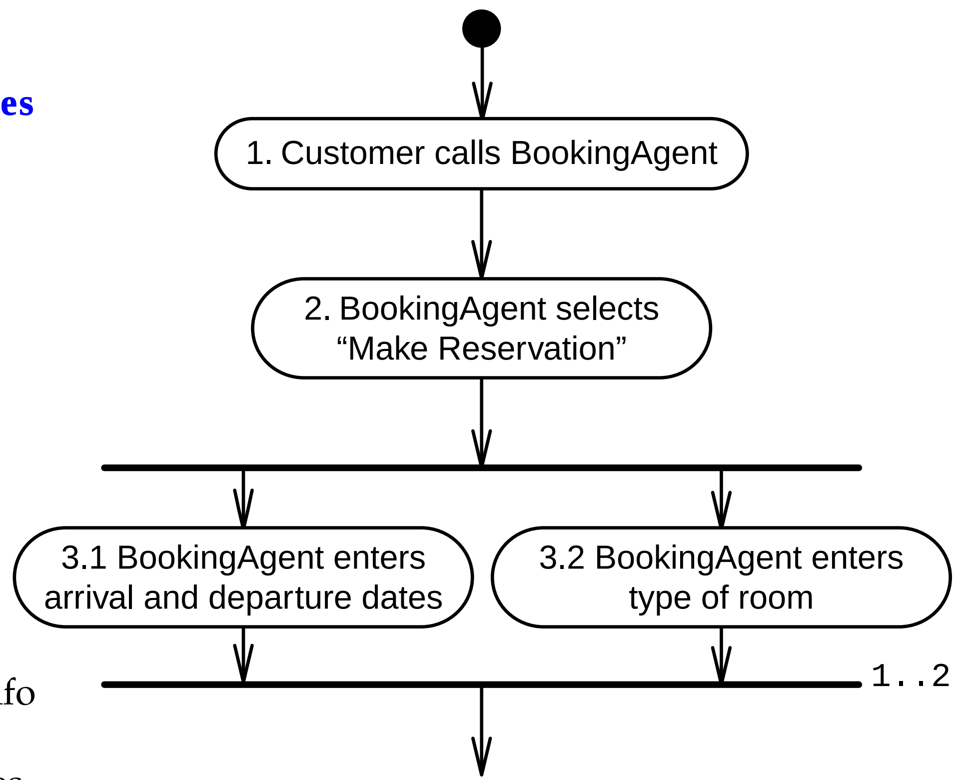
1. Customer calls BookingAgent
2. BookingAgent select “Make Reservation” icon
3. BookingAgent enters search criteria
   3.1 BookingAgent enters arrival and departure dates
   3.2 BookingAgent enters type of room
4. BookingAgent presses the “Search” button

...

11. BookingAgent enters customer name
12. BookingAgent presses the “Search” button
13. If a customer match is not found
    13.1 BookingAgent enters address info
    13.2 BookingAgent enters phone info
    13.3 BookingAgent presses “Add New Customer”
14. Else
    14.1 The system display match list
    14.2 BookingAgent selects the correct customer
    14.3 The System populates the GUI with customer info

...

21. The System saves the reservation and displays ResvID



Summary

- Use Case scenarios provide much detail about a use case. An analysis of this detail is recorded in the Use Case form.
- This analysis also helps you refine the Use Case model.
- You can visually represented the flow of events of a use case with an Activity diagram.

Module 8

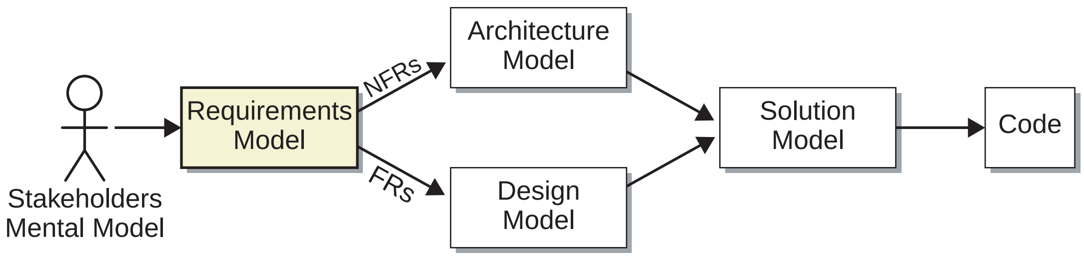
Determining the Key Abstractions

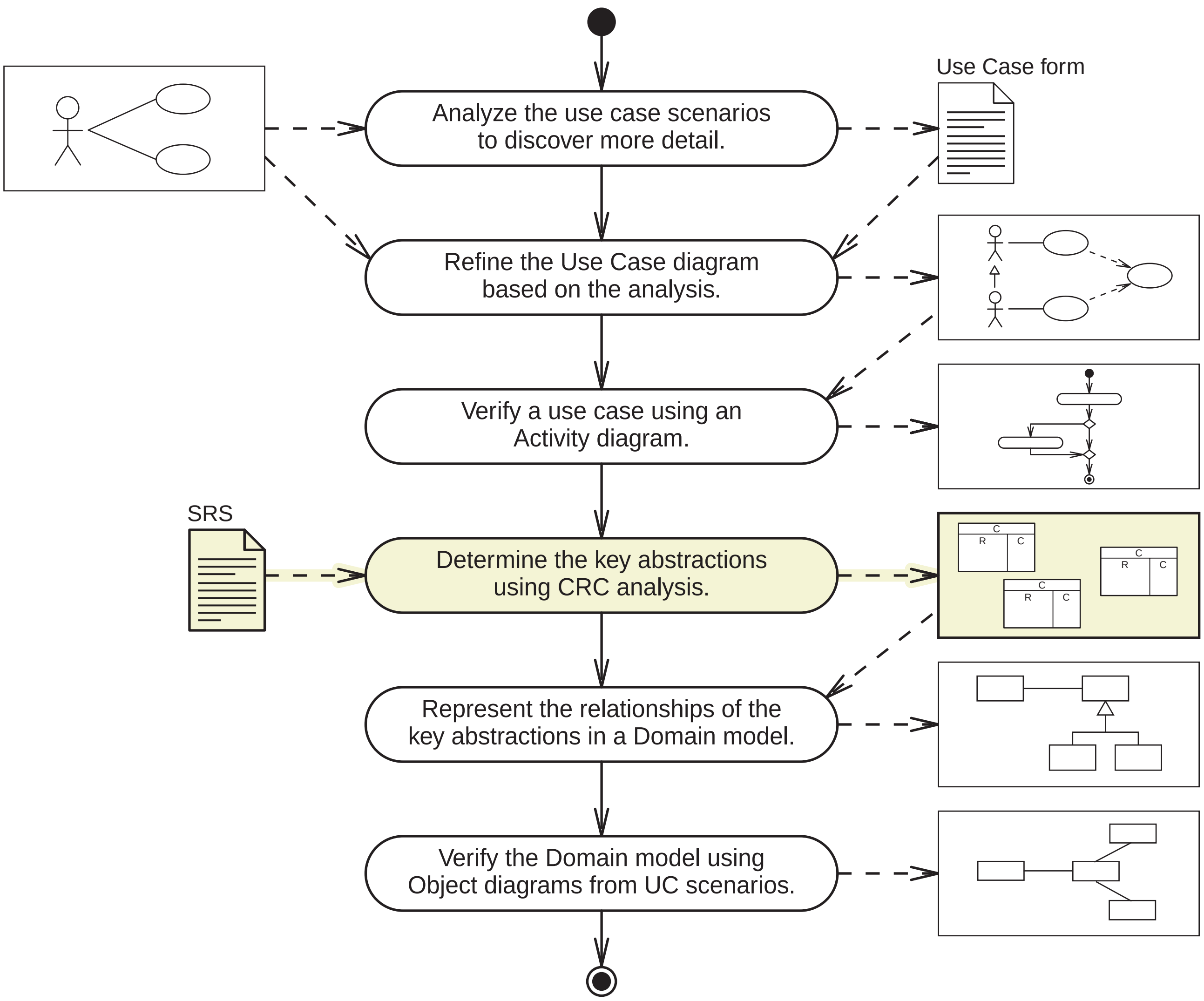
Objectives

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

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

Process Map


Stakeholders
Mental Model
Requirements
Model
NFRs
FRs
Architecture
Model
Design
Model
Solution
Model
Code


Analyze the use case scenarios
to discover more detail.
Use Case form
Refine the Use Case diagram
based on the analysis.
Verify a use case using an
Activity diagram.
SRS
Determine the key abstractions
using CRC analysis.
Represent the relationships of the
key abstractions in a Domain model.
Verify the Domain model using
Object diagrams from UC scenarios.

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 SRS 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 SRS by focusing on the following areas in the SRS document:

- The Scope and Context sections (SRS 1.2 and 1.3)
- The Functional Requirements section (SRS 4.0)
  - The use cases and scenarios
  - The specific functional requirements
- The Project Glossary section (SRS 6.0)

Tip: With practice you will be able to skip some of the nouns that are obviously not part of the domain, but for now you will perform an exhaustive search of the SRS.

SRS Nouns

Here are a few excerpts from the Hotel Reservation System SRS with the nouns marked in blue:

"The Hotel Reservation System will be responsible for managing the reservations for multiple lodging properties, which include (but are not limited to) bed and breakfast (B&B) and business retreat properties. The system will also include a web application that permits customers to view the properties and rooms, to view current and past reservations, and to make new reservations. The system must also coordinate small events (such as retreats and small conferences)." *(the Scope section)*

"There are three main 'touch points' of the Hotel Reservation System: the central DBMS for data storage, the external, credit card authorization system (Authorize.net), and the local movies-on-demand system that controls the movie feed to each room's television set." *(the Context section)*

"The System must collect the following information about a customer: first and last name (as separate fields), address, home phone, a major credit card (type, number, and expiration date)" *(from FR E1-3)*

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 SRS.
- 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		
Lodging properties		
Retreat properties		
Customers		
Rooms		
Small business conference		
Credit card authorization system		
First and last name		
Address		

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 Abstraction 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 scenario and FRs to determine responsibilities and collaborators.
4. Document this key abstraction with a CRC card.
5. 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 mentioned in the Scope section of the SRS.

Selecting a Key Abstraction Candidate

The noun "reservation" appears many times in the following areas:

- In the Scope section

  "The Hotel Reservation System will be responsible for managing the *reservations* for multiple lodging properties"

- In these use case names:
  - E1: Manage *Reservation*
  - E5: Manage *Reservation* Online
- In many places throughout the FRs

  "E1-1: The System shall permit a Booking Agent to create, retrieve, update, and delete 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 descriptions for the candidate key abstraction.
3. Scan the use case scenarios for the candidate key abstraction.
4. Scan the text of the use case scenarios to see if the candidate key abstraction is mentioned. If it is, the scenario will be relevant.

Identifying a Relevant Use Case

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

- E1: Manage *Reservation*
- E5: Manage *Reservation* Online

Determining Responsibilities and Collaborators

Scan the scenarios and FRs 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

Here are a few relevant FRs:

E1-1 – The system shall permit a Booking Agent to create, retrieve, update, and delete a reservation. A reservation has an arrival date, a departure date, and a reservation ID.

E1-2 – reservation holds one or more rooms for a specified time period (between the arrival and departure dates).

E1-3 – A reservation is associated with only one customer (and so on).

E1-5 – A reservation begins in the "held" state (and so on).

Documenting a Key Abstraction Using a CRC Card

Class Name	
Responsibilities	Collaborators

Documenting a Key Abstraction Using a CRC Card

Reservation	
Responsibilities	Collaborators
`Reserves a Room`	`Room` `Customer`

`status`
` (New, Held, Confirmed)`
`arrival date`
`departure date`
`form of payment`
`reservation number`	

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
Lodging properties		Property
Retreat properties	Subtype of Property	
Customers		Customer
Rooms		Room
Small business conference	Subtype of Reservation	
Credit card authorization system	An external system	
First and last name	Attribute of Customer	

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 in the SRS 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 9

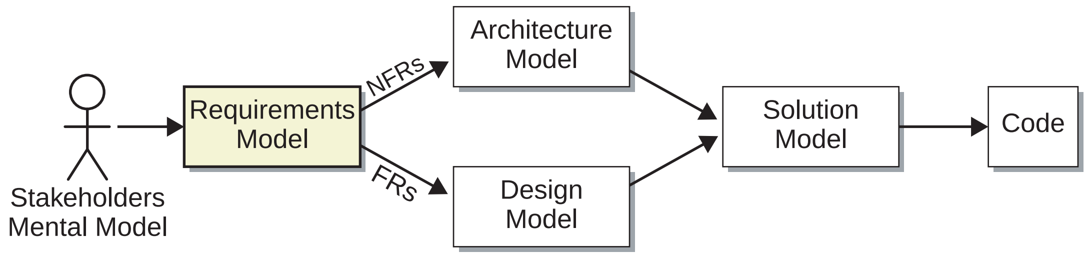
Constructing the Problem Domain Model

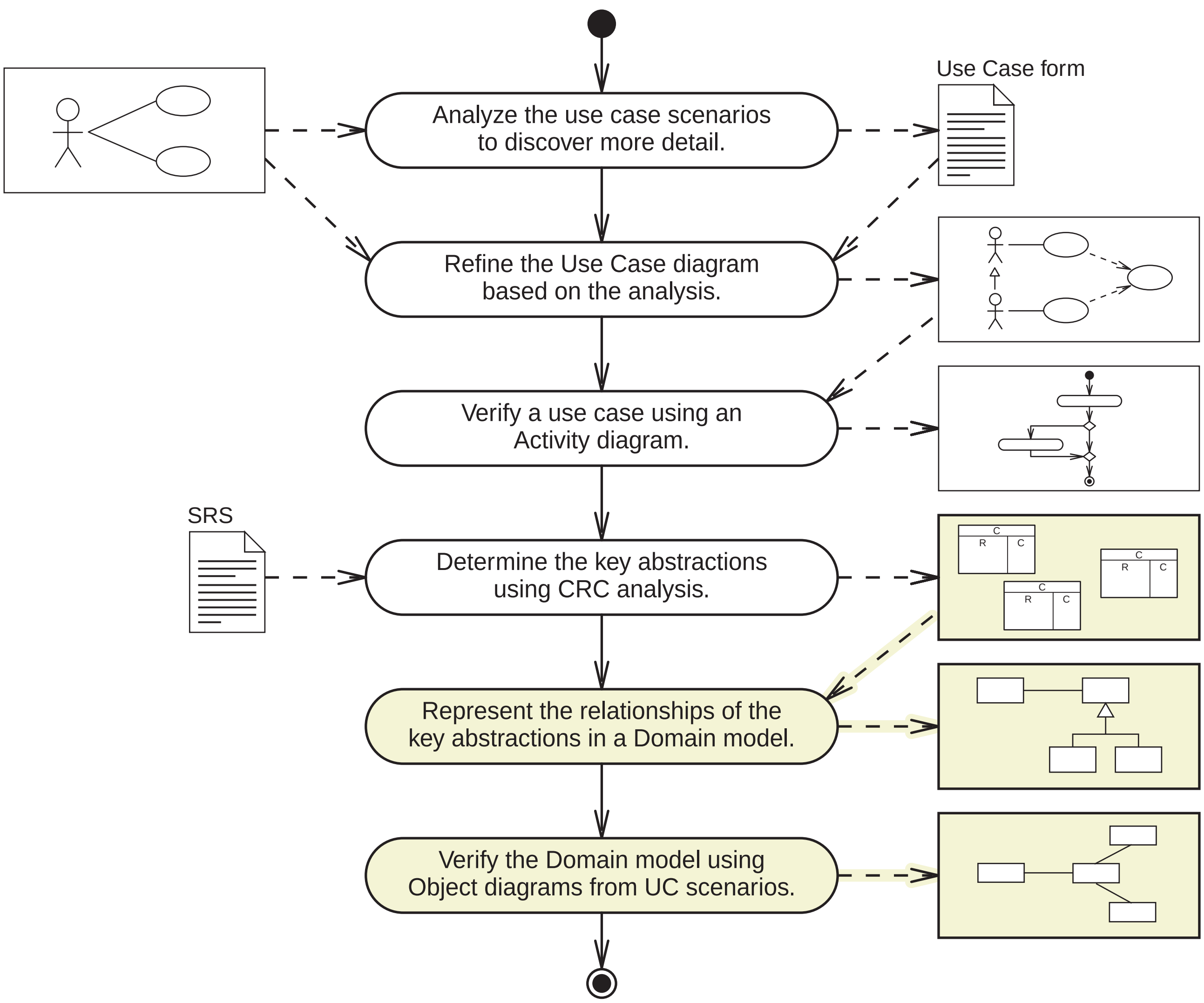
Objectives

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

Process Map


Stakeholders
Mental Model
Requirements
Model
NFRs
FRs
Architecture
Model
Design
Model
Solution
Model
Code


Use Case form
Analyze the use case scenarios
to discover more detail.
Refine the Use Case diagram
based on the analysis.
Verify a use case using an
Activity diagram.
SRS
Determine the key abstractions
using CRC analysis.
Represent the relationships of the
key abstractions in a Domain model.
Verify the Domain model using
Object diagrams from UC scenarios.

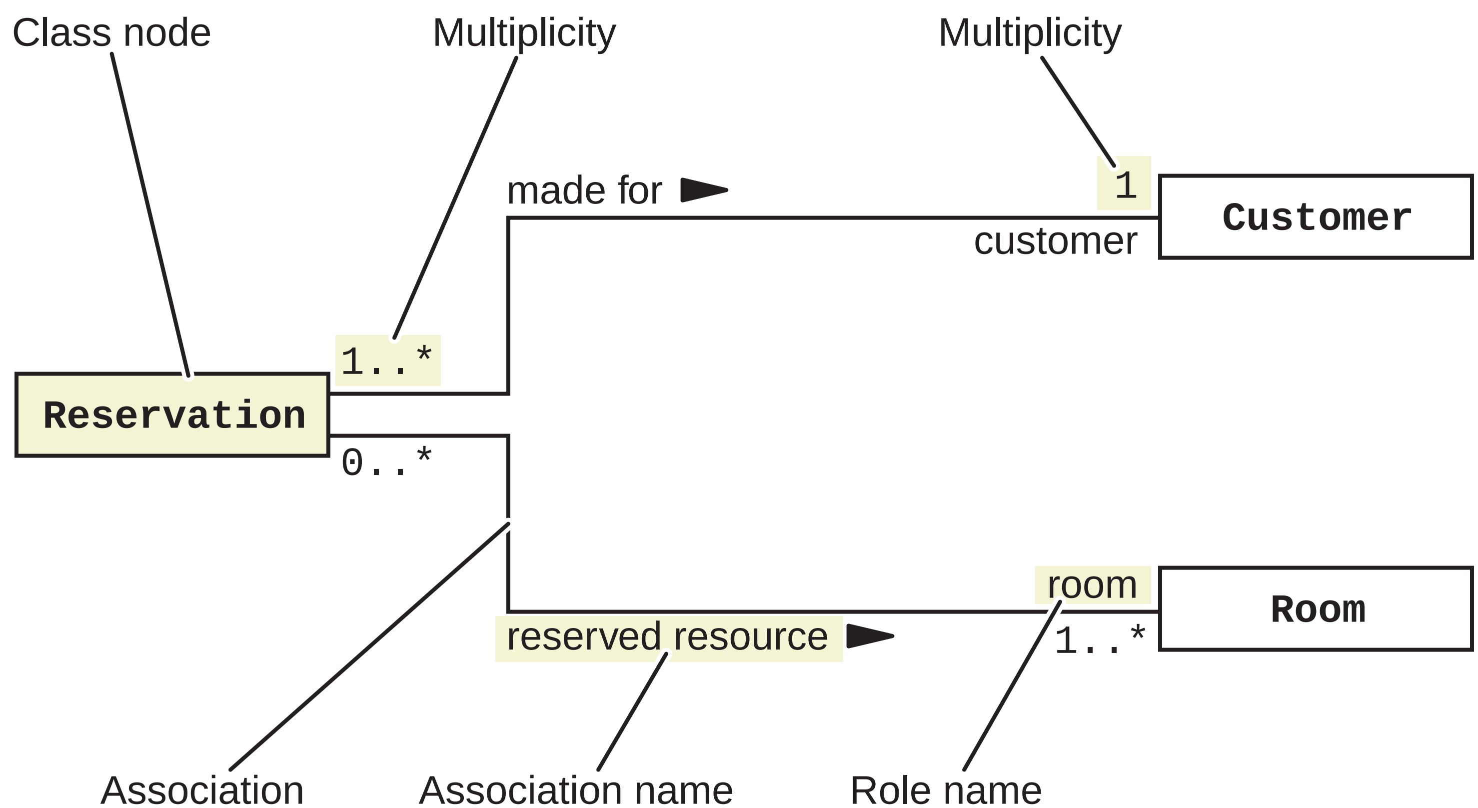
Introducing the Domain Model

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.

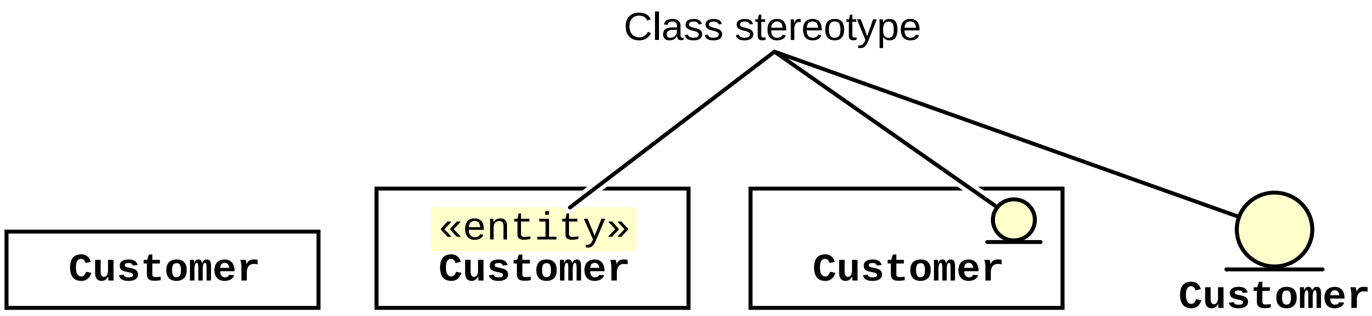
Identifying the Elements of a Class Diagram

A UML Class diagram is composed of the following elements:



Class Nodes

Class nodes represent classes of objects within the model.

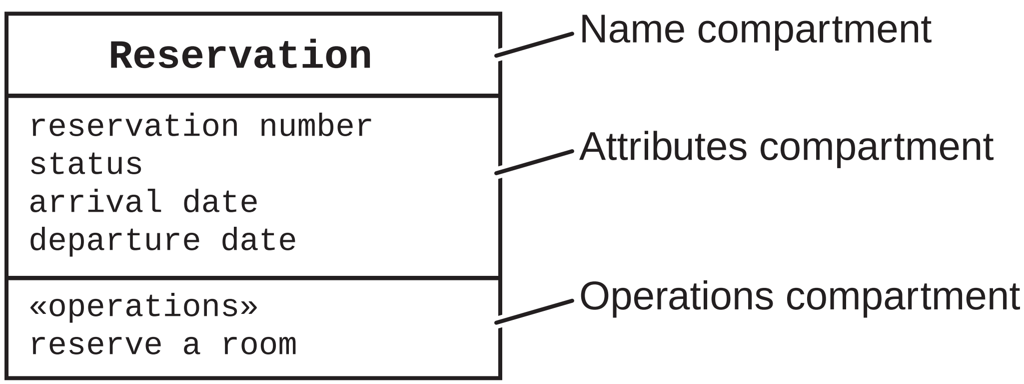


These can represent:

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

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

Class Node Compartments

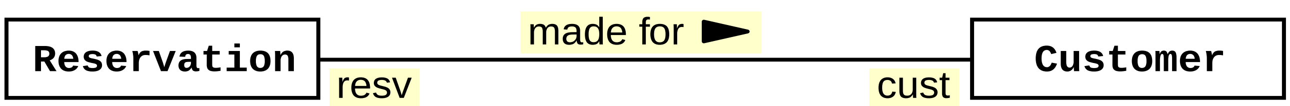


- The name compartment records the name of the class.
- The attributes compartment records attributes (or instance variables) of the class.
- The operations compartment records operations (or methods) of the class.

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

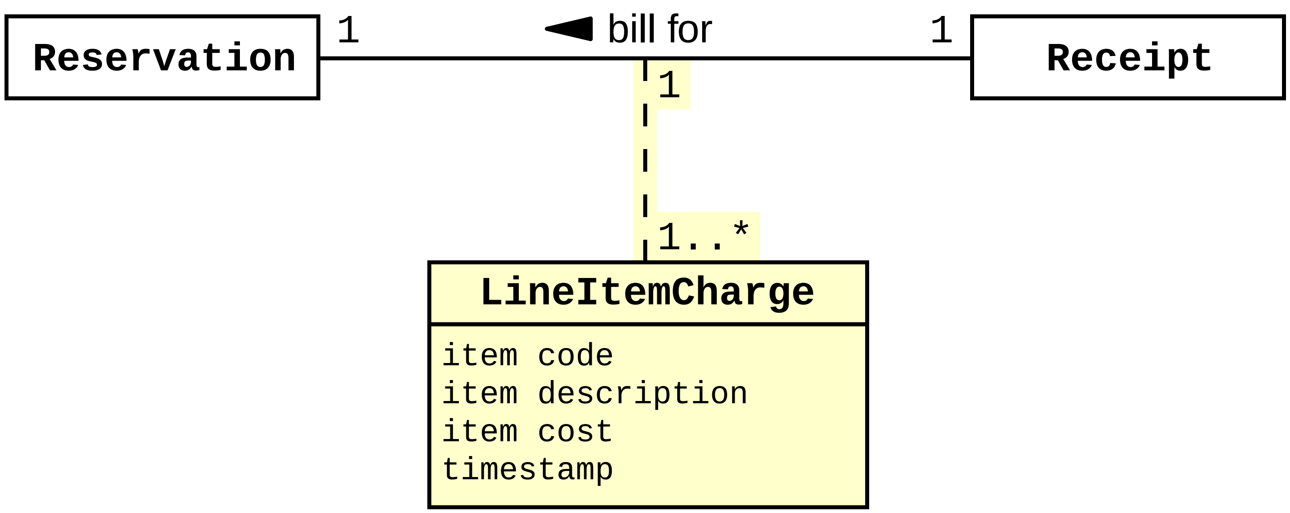
Navigation arrows on the association determine what direction an association can be traversed at runtime.

For example:

This association would be read as “From a reservation the system can retrieve the room.” However, this model excludes this possibility: “From a room the system can retrieve the reservations for that room.”

Association Classes

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



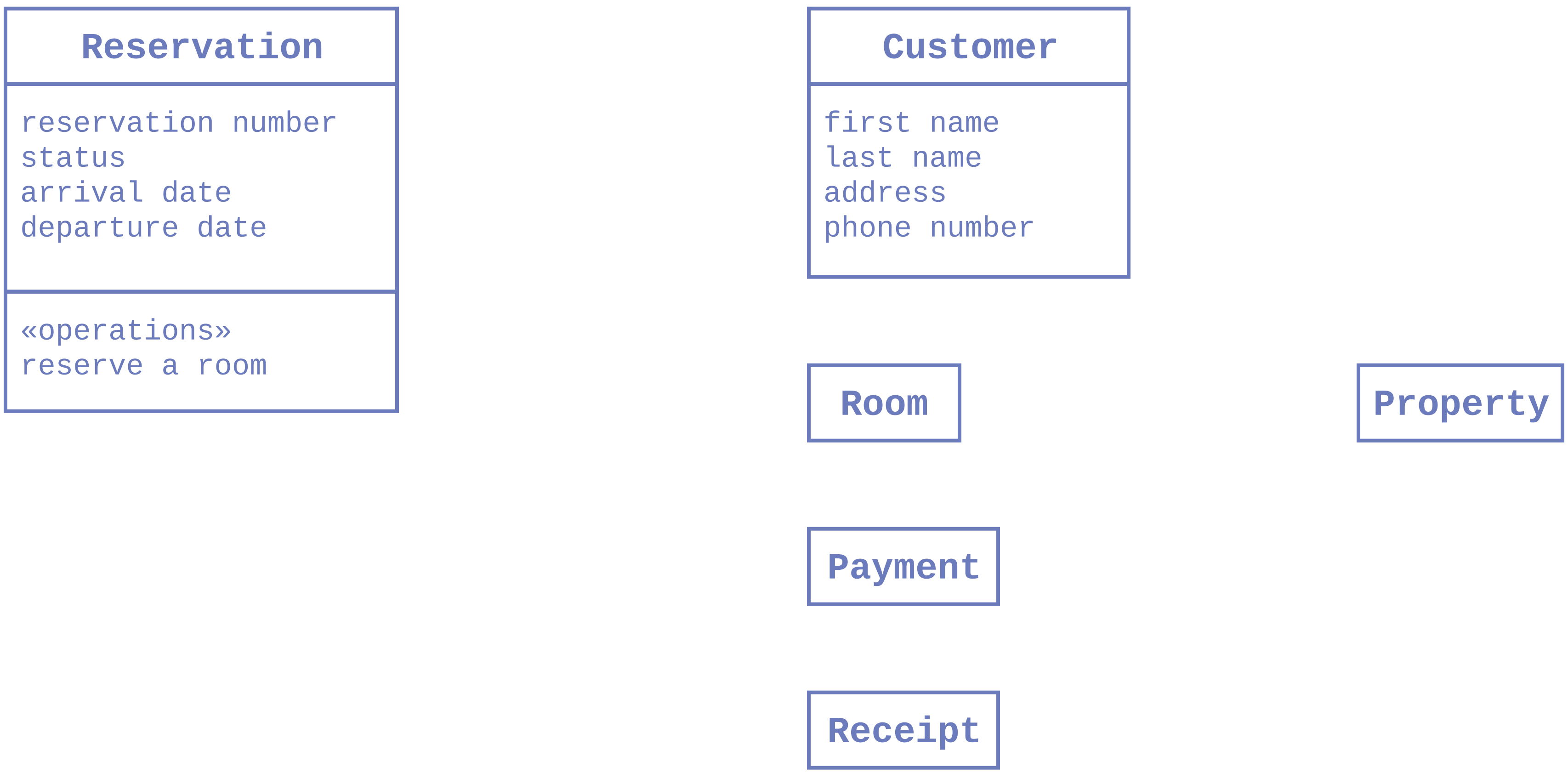
`LineItemCharge` is an association class that records the item and cost of the charge on the reservation receipt.

Creating a Domain Model

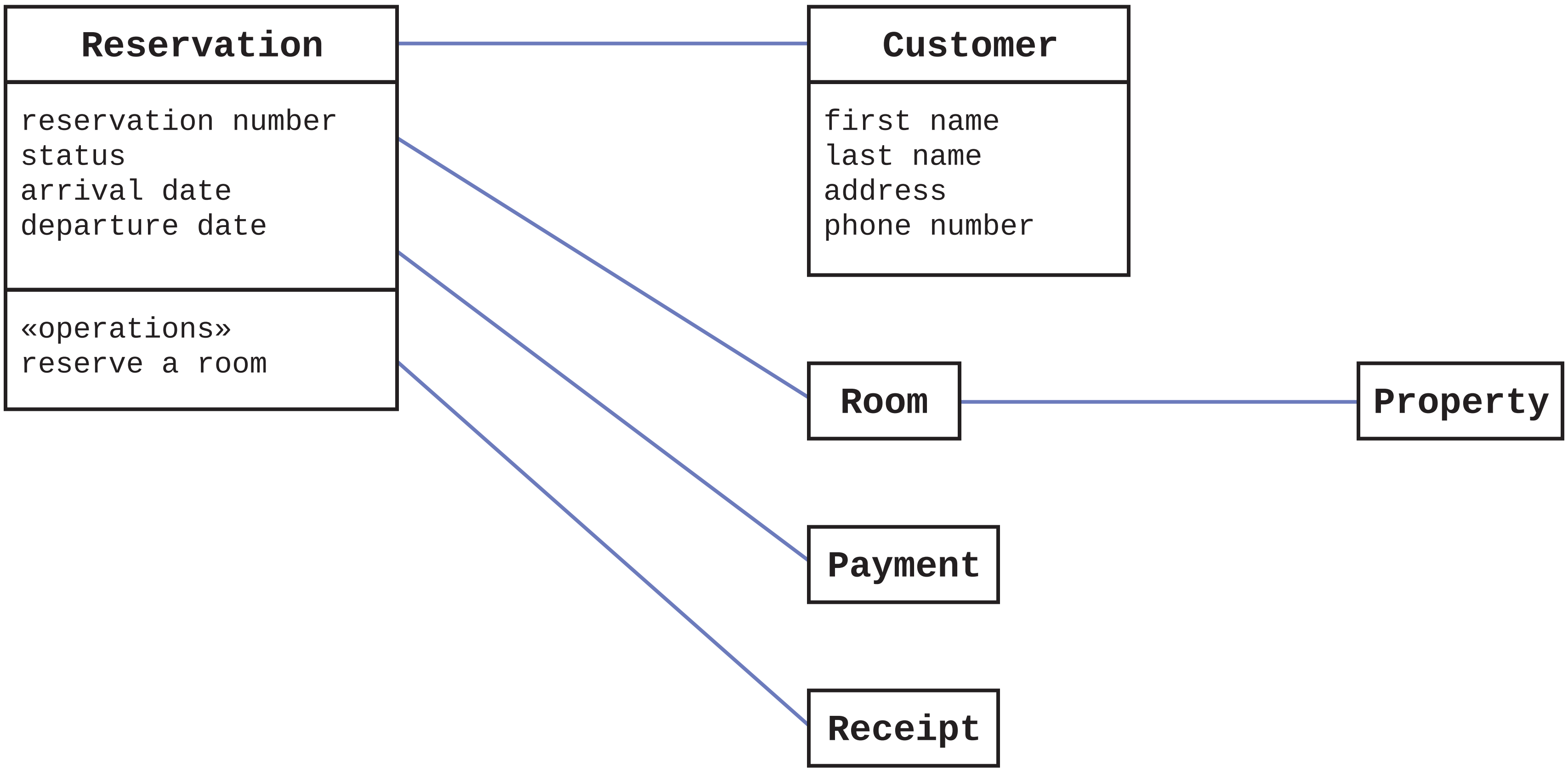
Starting with the key abstractions, you can create a Domain model using these steps:

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. Identify and document association navigation.
6. Identify and document association classes.

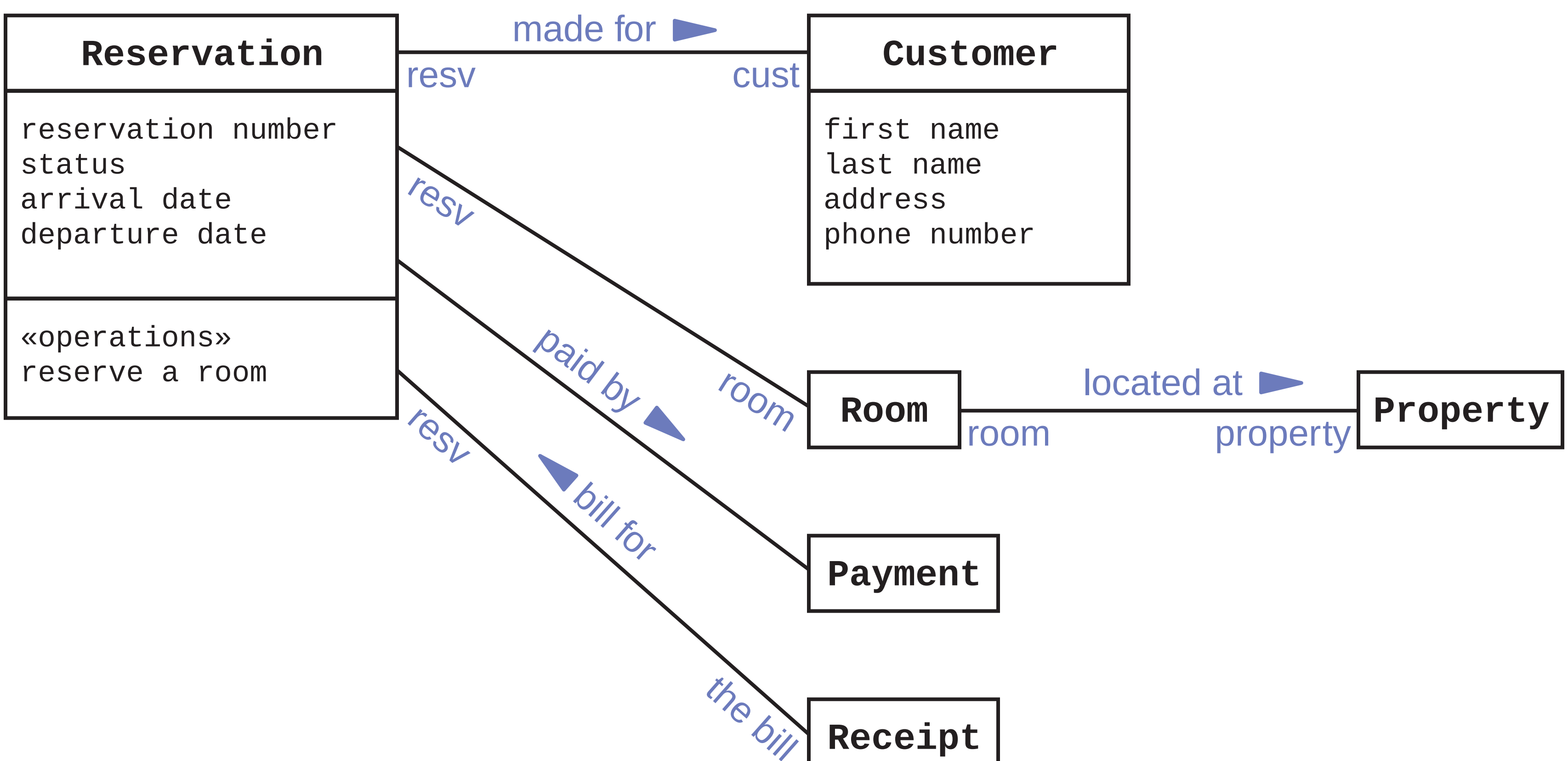
Step 1 – Draw the Class Nodes



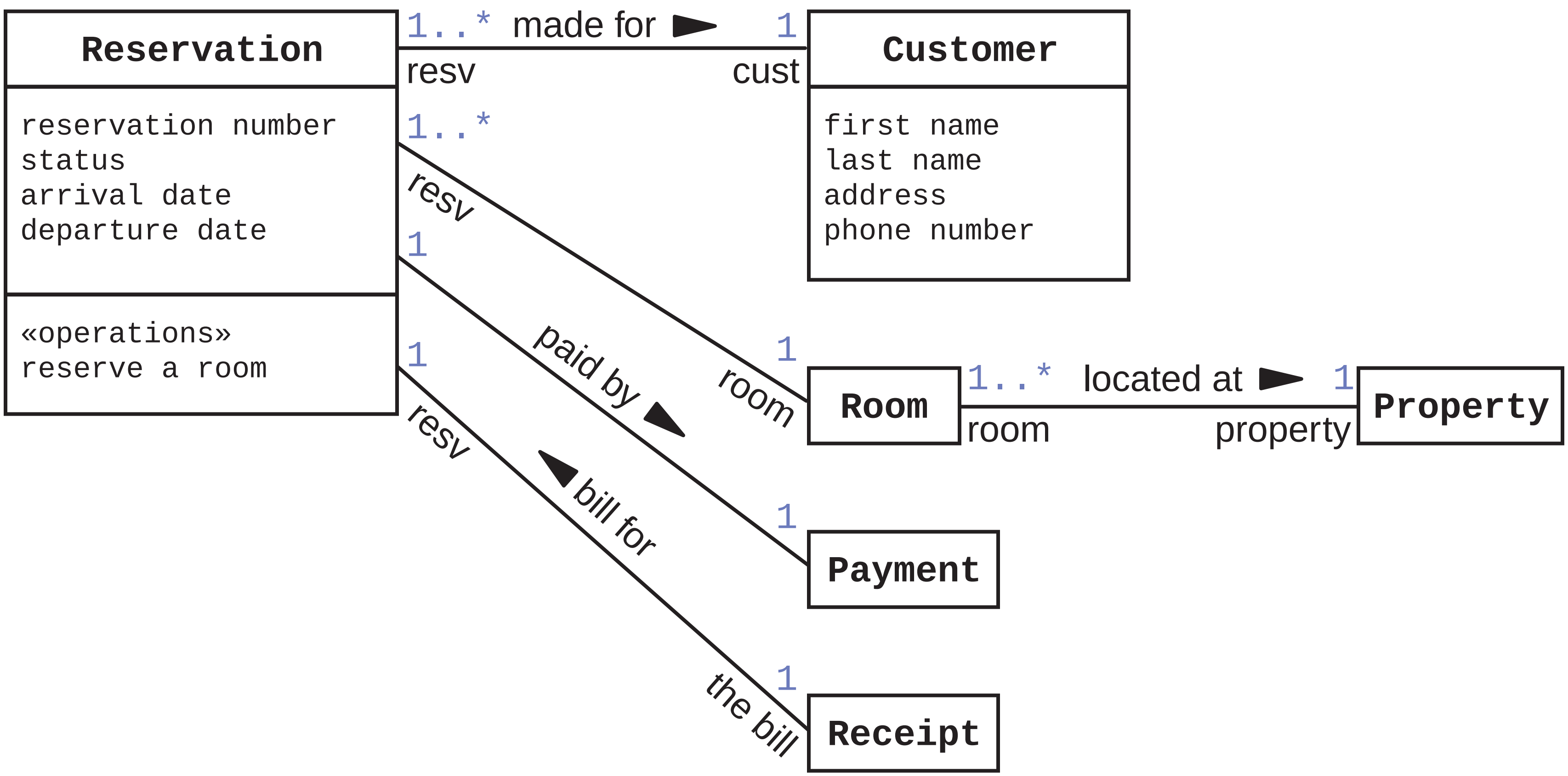
Step 2 – Draw the Associations



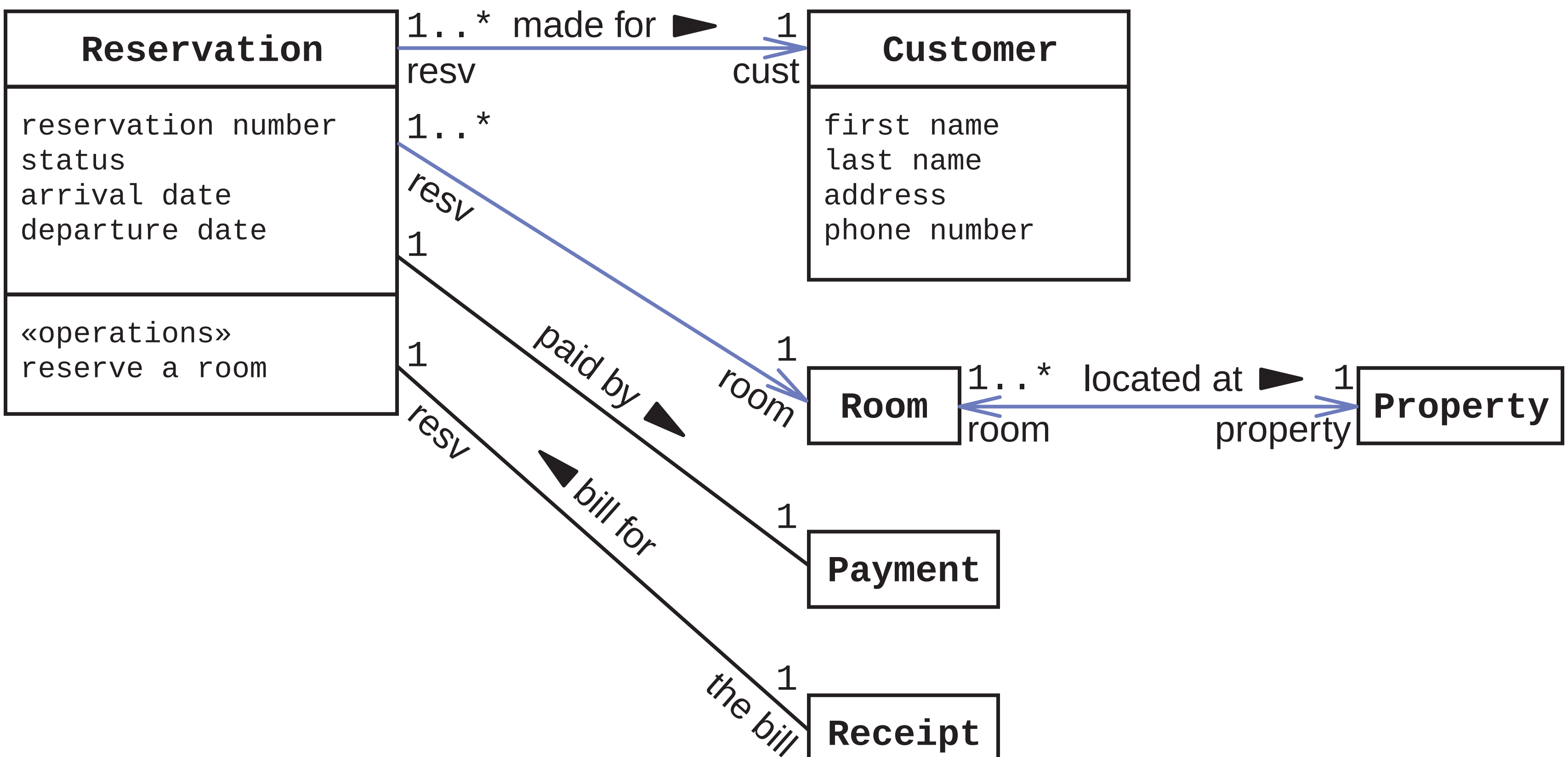
Step 3 – Label the Associations and Role Names



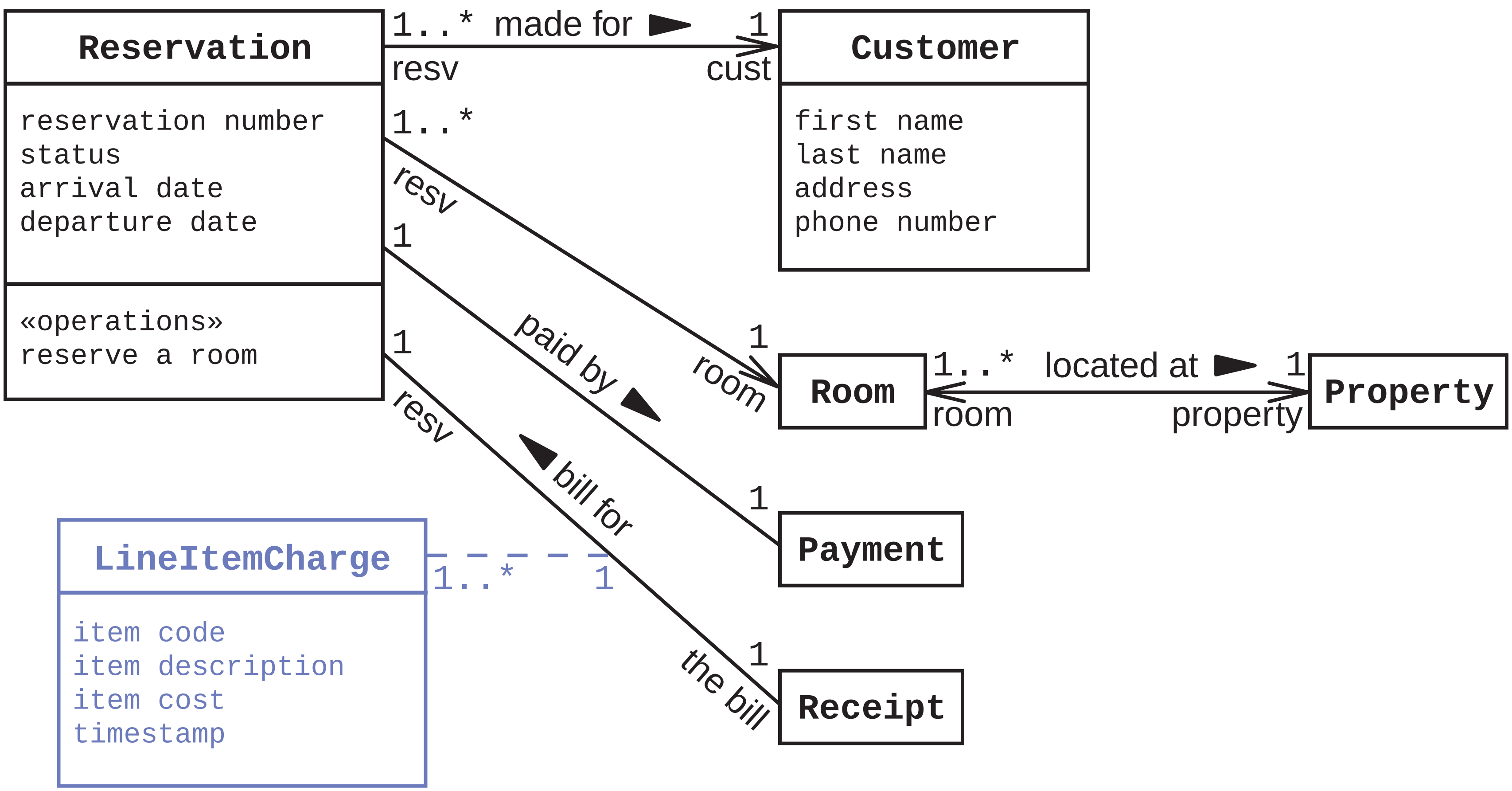
Step 4 – Label the Association Multiplicity



Step 5 – Draw the Navigation Arrows



Step 6 – Draw the Association Classes



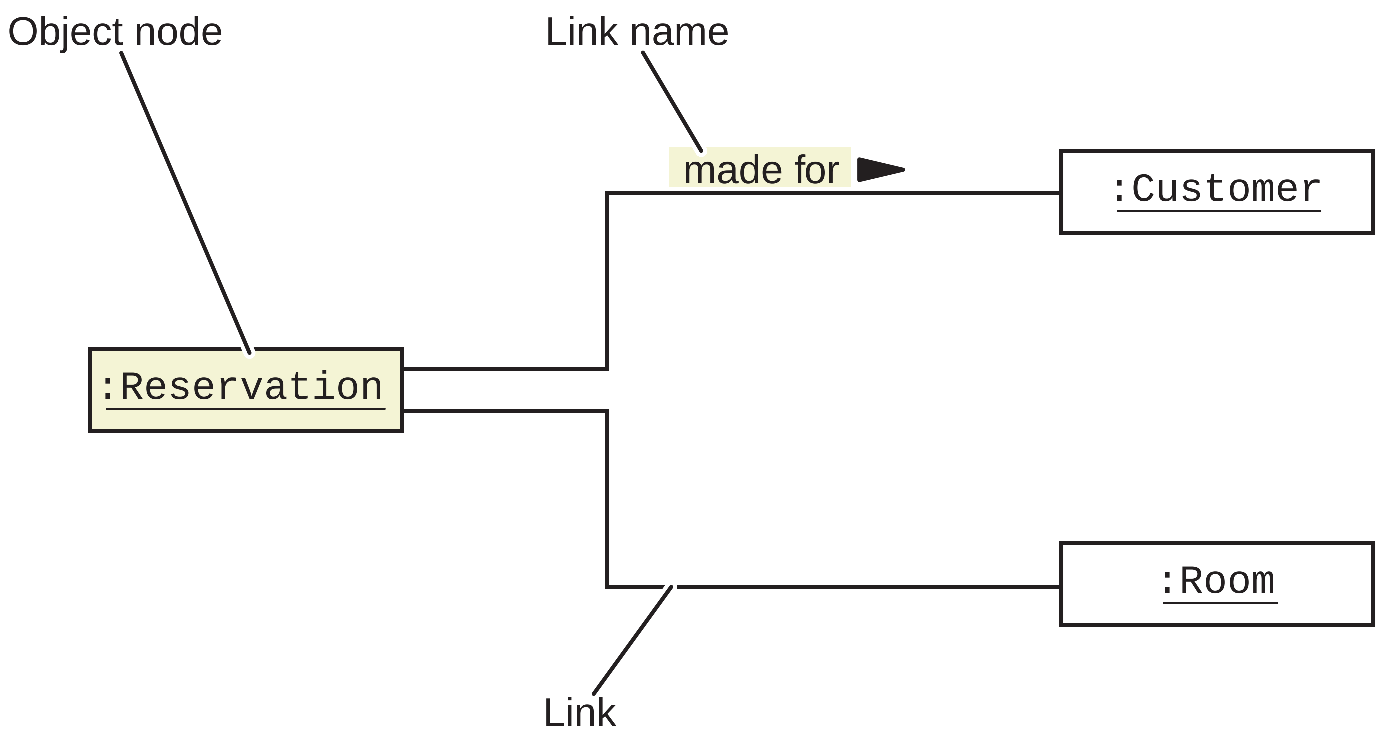
Validating the Domain Model (Intro)

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 spec v1.4, page 3-35]



Object Nodes

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

Object name without type	Type without a name	Type with a name
Victoria	:Room	Blue:Room

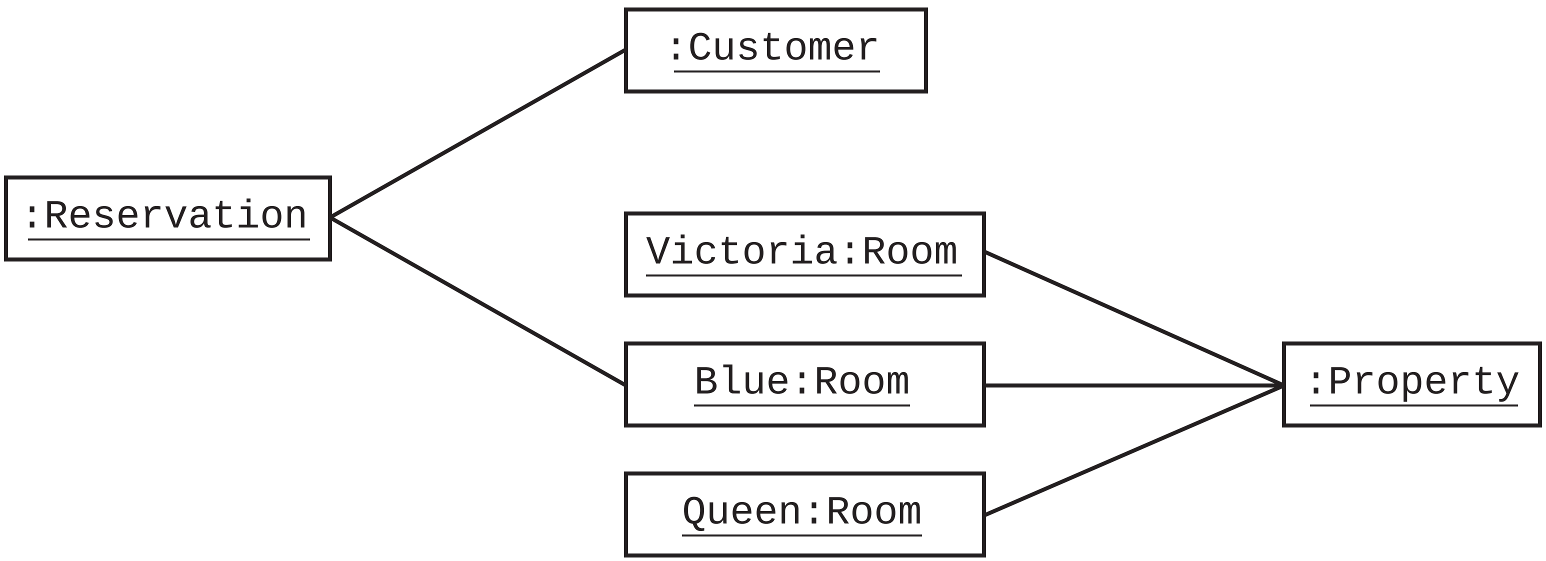
An object node might also include attributes:

```
: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

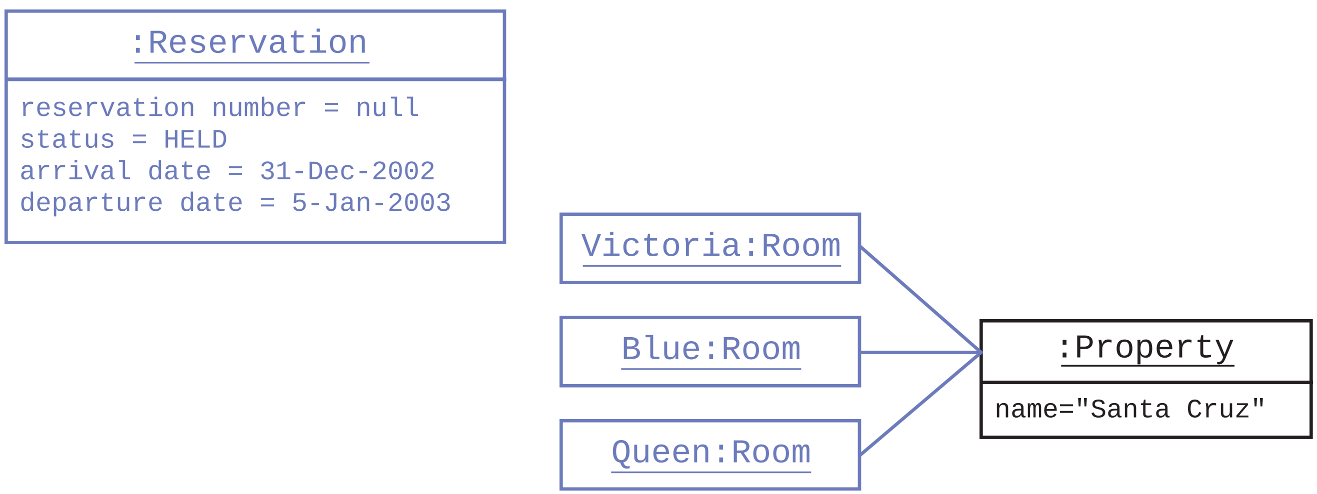
1. Pick one or more use cases that exercise the Domain model.
2. Pick one or more use case scenarios for the selected use cases.
3. Walk through each scenario (separately), and construct the objects (with data) mentioned in the scenario.
4. Compare each Object diagram against the Domain model to see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

Step 1 – Create Reservation Scenario 1

Medoca Sansumi, booking agent for the Santa Cruz B&B, is waiting for a call and has the HotelApp Main screen displayed. A telephone call comes in from Ms. Jane Googol, a customer from New York city. “Hello, this is Jane Googol. I would like to make a reservation for New Year’s Eve,” says Ms. Googol. Medoca selects the “Create Reservation” function on the main screen of the HotelApp. An empty reservation form appears.

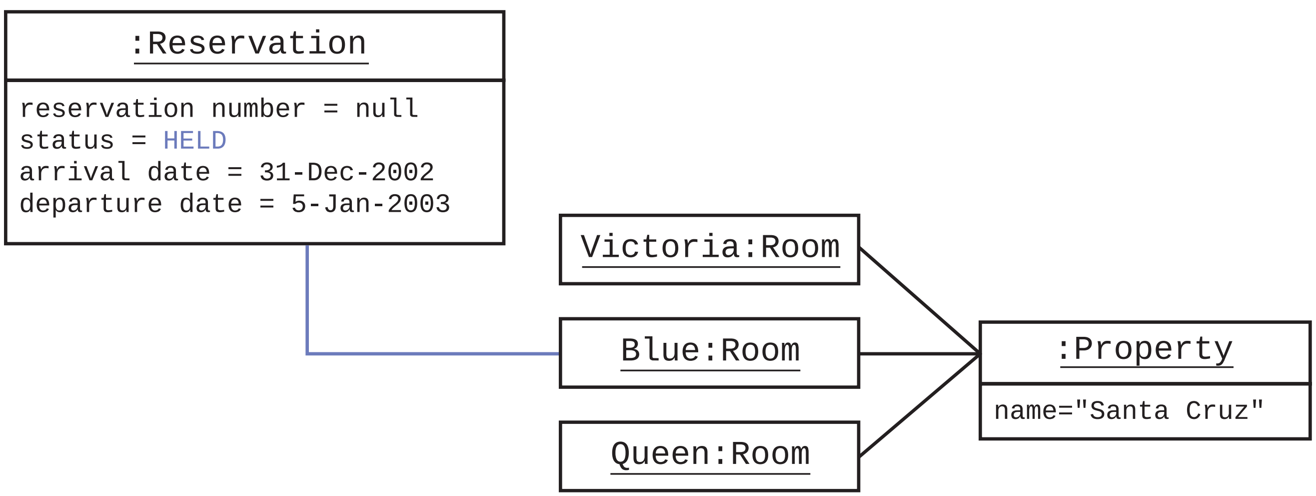
Step 2 – Create Reservation Scenario 1

"When will you arrive?" asks Medoca. "December 31st," says Jane, "and I would like to stay through January 5th." Medoca enters the dates in the form. "What type of room would you like?" Medoca asks. "I will be with my husband so a single room will be sufficient. Is the Blue room available?" Jane asks. Medoca selects "single" in the reservation form and performs the search. The system responds with three available rooms: Victoria, Blue, and Queen.



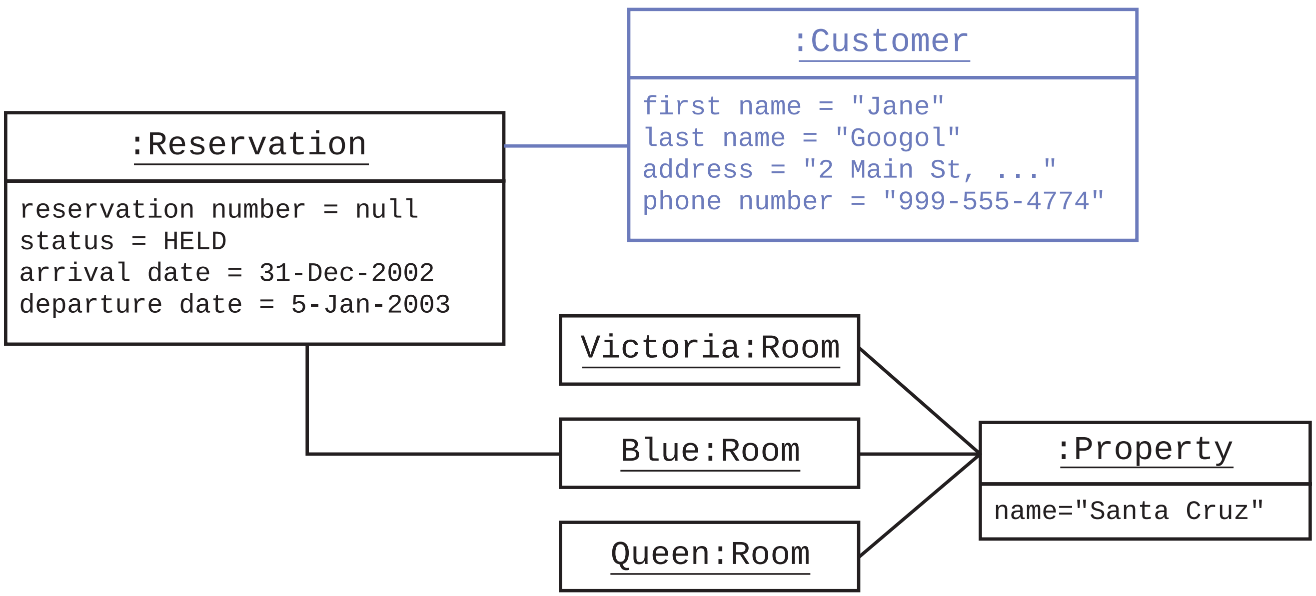
Step 3 – Create Reservation Scenario 1

"Yes, it is," replies Medoca. Medoca selects the Blue room and the system populates the reservation form and marks the reservation as "held."



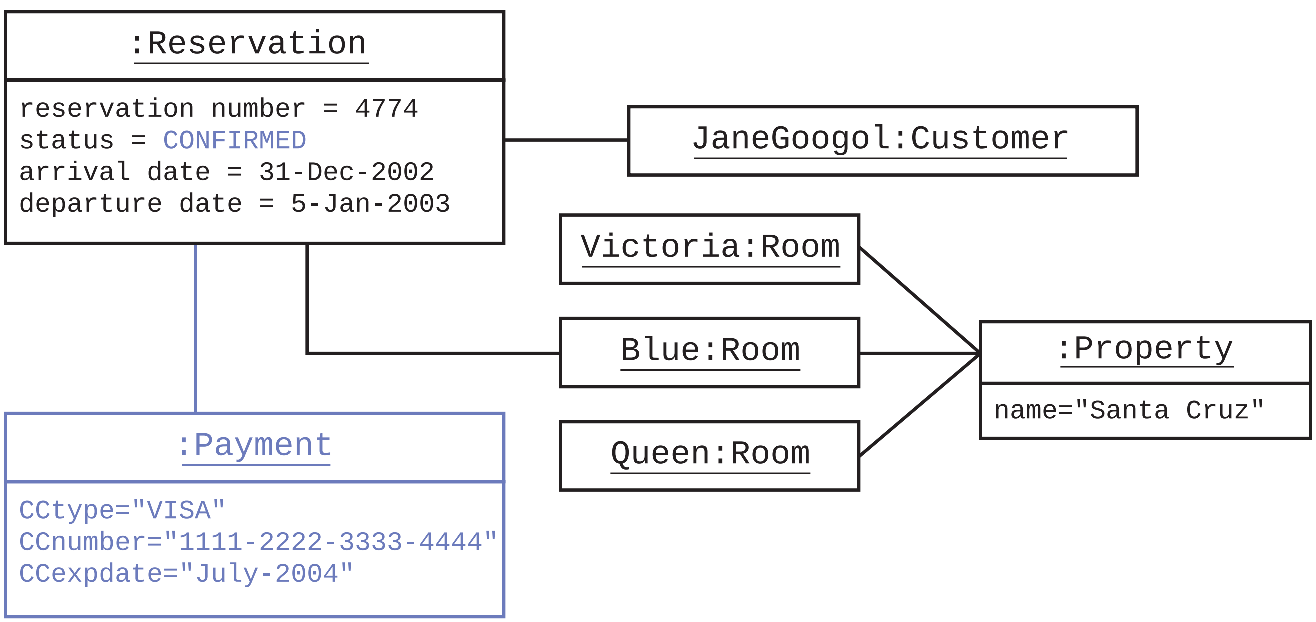
Step 4 – Create Reservation Scenario 1

Medoca enters Jane's full name into the system. Ms. Googol is an existing customer, so the system responds by populating the customer fields in the reservation form.



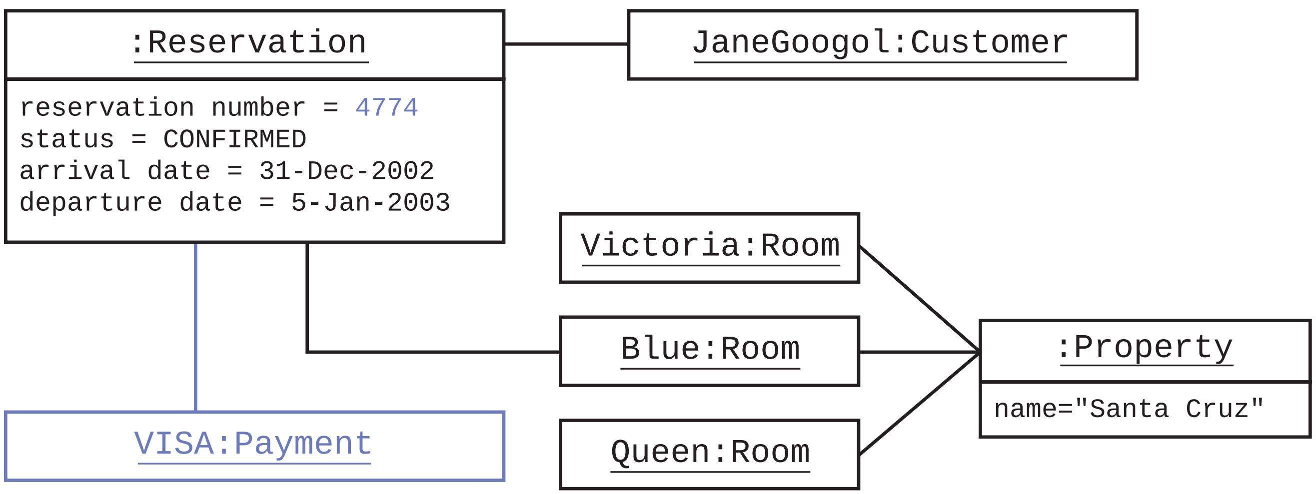
Step 5 – Create Reservation Scenario 1

"Would you like to confirm this reservation today?" asks Medoca. "Yes," says Jane, "use my VISA card number 1111-2222-3333-4444." Jane pauses as Medoca types this in. "The expiration date is July, 2004." Medoca enters this information and selects "Verify Payment" on the system. After about five seconds, the system responds that the credit is verified. The system changes the state of the reservation to "`CONFIRMED`."



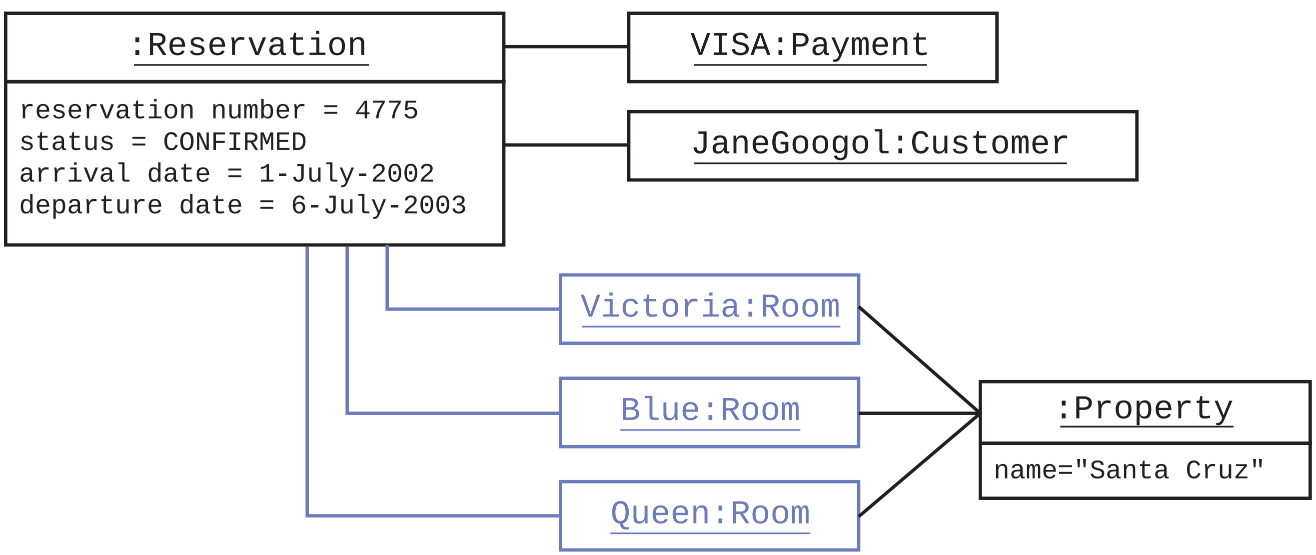
Step 6 – Create Reservation Scenario 1

Medoca tells Jane the reservation ID (supplied by the system) and asks, “Is there anything else I can do for you today?” Jane replies no and Medoca thanks her and says goodbye. Medoca closes the Reservation form window, which returns her to the Main HotelApp screen.



Create Reservation Scenario No. 2

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

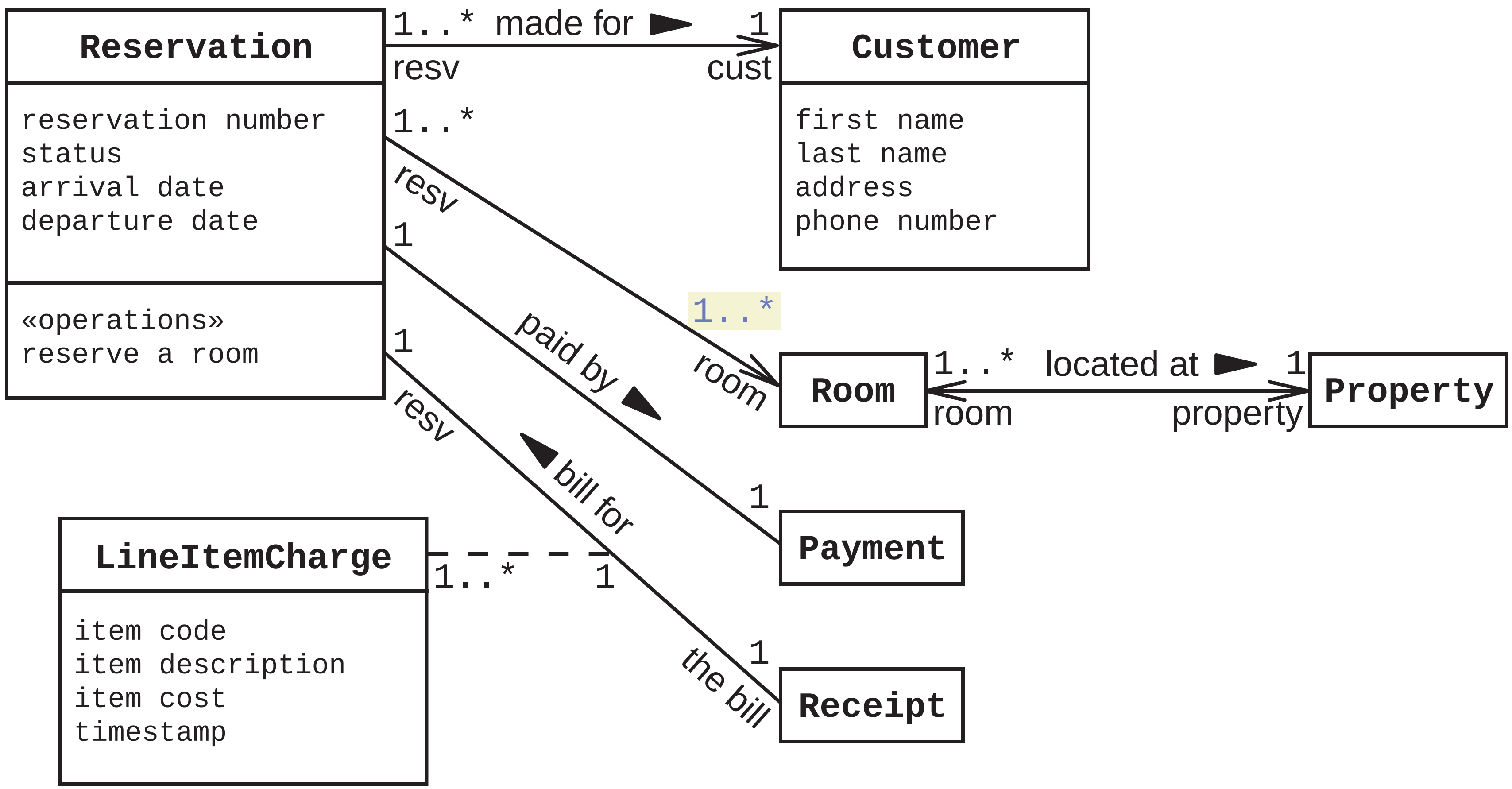


Comparing Object Diagrams to Validating the Domain Model

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.

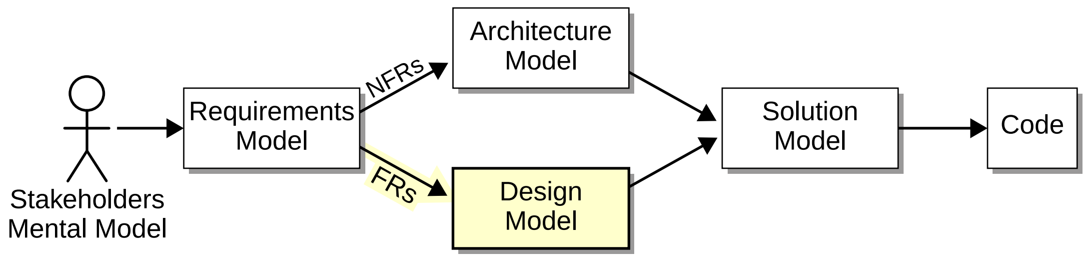
Module 10

Creating the Design Model Using Robustness Analysis

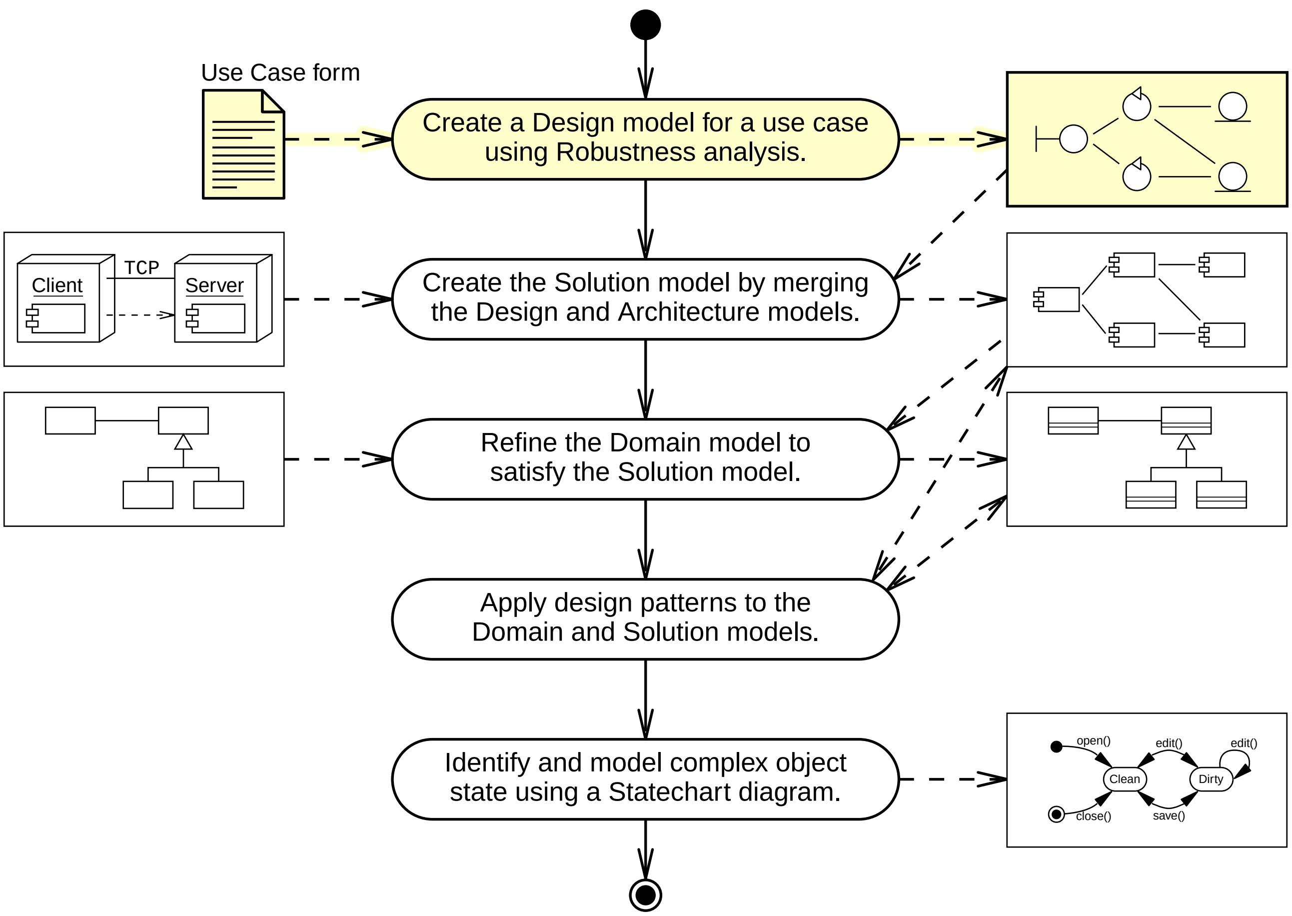
Objectives

- Explain the purpose and elements of a Robustness Analysis and the Design model
- Identify the essential elements of a UML Collaboration diagram
- Create a Design model for a use case using Robustness analysis
- Identify the essential elements of a UML Sequence diagram
- Generate a Sequence diagram view of the Design model

Process Map



Stakeholders Mental Model
Requirements Model
NFRs
FRs
Architecture Model
Design Model
Solution Model
Code

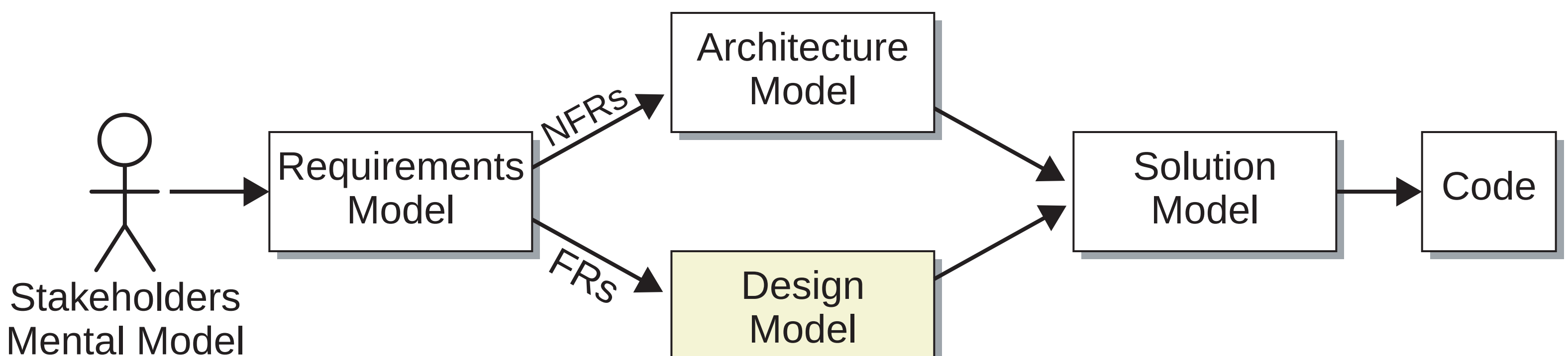


Use Case form
Create a Design model for a use case using Robustness analysis.
Create the Solution model by merging the Design and Architecture models.
Refine the Domain model to satisfy the Solution model.
Apply design patterns to the Domain and Solution models.
Identify and model complex object state using a Statechart diagram.
Client
TCP
Server
open()
edit()
edit()
Clean
Dirty
close()
save()

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.



Comparing Analysis and Design

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

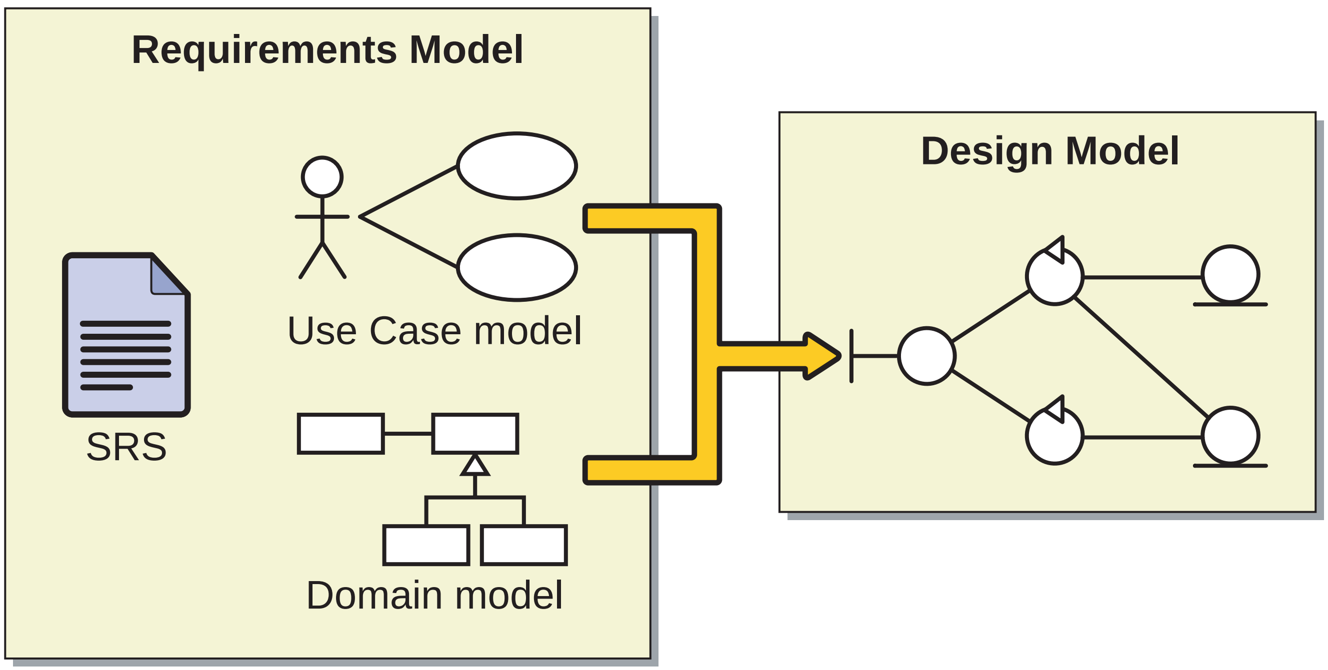
- Use cases
- Domain model

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 leads from a use case to a Design model that supports that use case:



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 as a UML Collaboration diagram with design components: Boundary, Service, and Entity components.

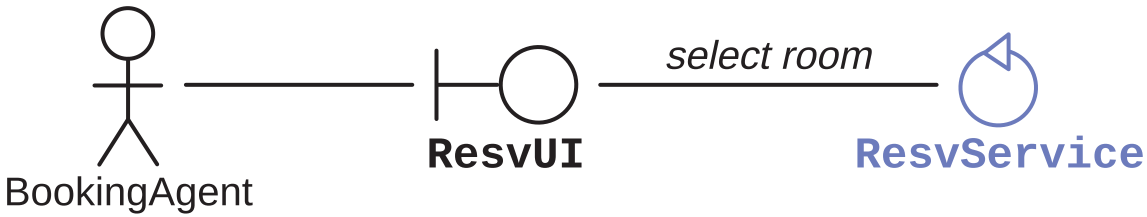
Boundary Components

"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

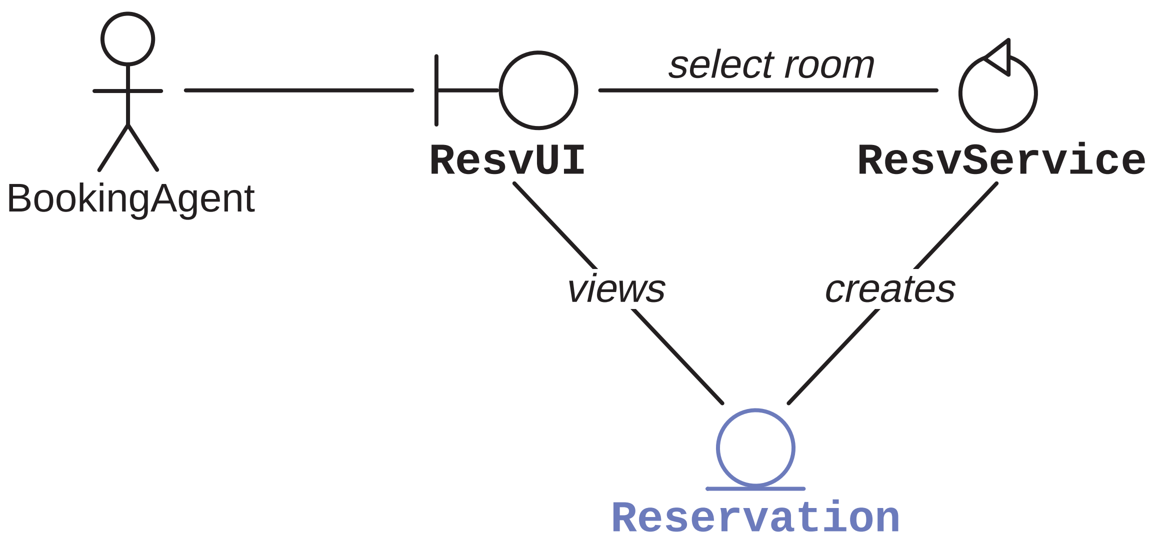
"Control (Service) classes (components) represent coordination, sequencing, transactions, and control of other objects and are often used to encapsulate control related to a specific use case." (Jacobson, Booch, and Rumbaugh page 185)



- 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)



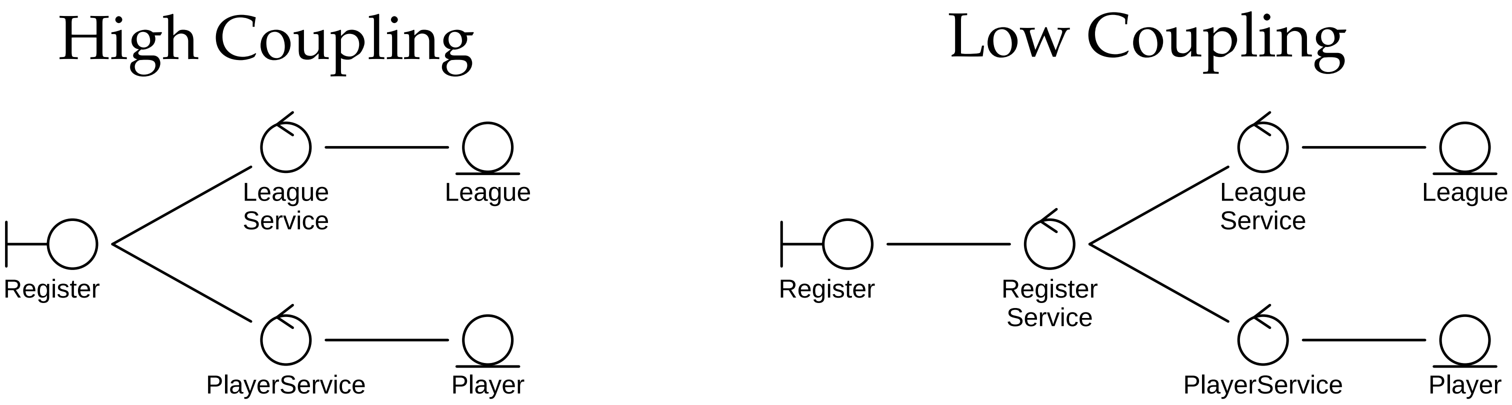
- Entities usually correspond to domain objects.
- Most entities are persistent.
- Entities can have complex behavior.

Describing the Robustness Analysis Process

1. Select a use case.
2. Construct a Collaboration 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 Collaboration diagram into a Sequence diagram for an alternate view (optional).

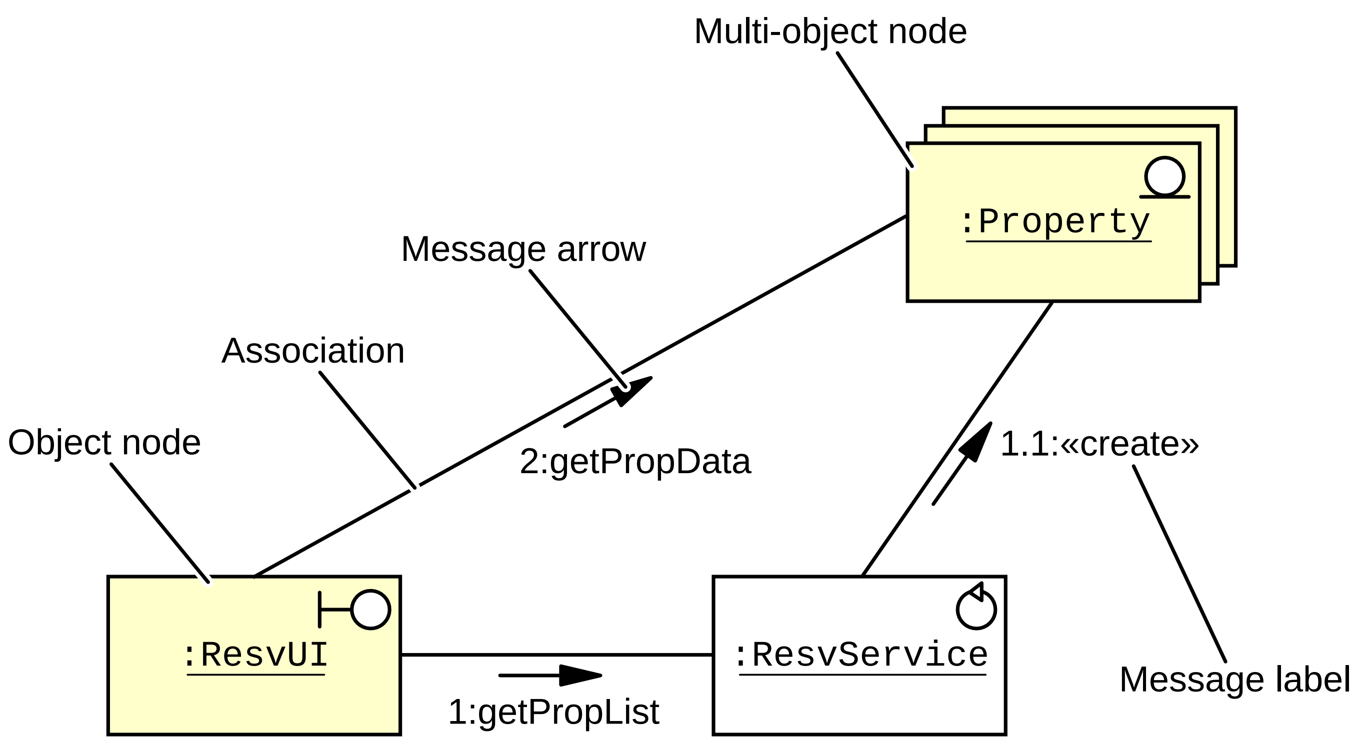
Façade Service

A façade service might be used to reduce coupling between boundary components and other services.



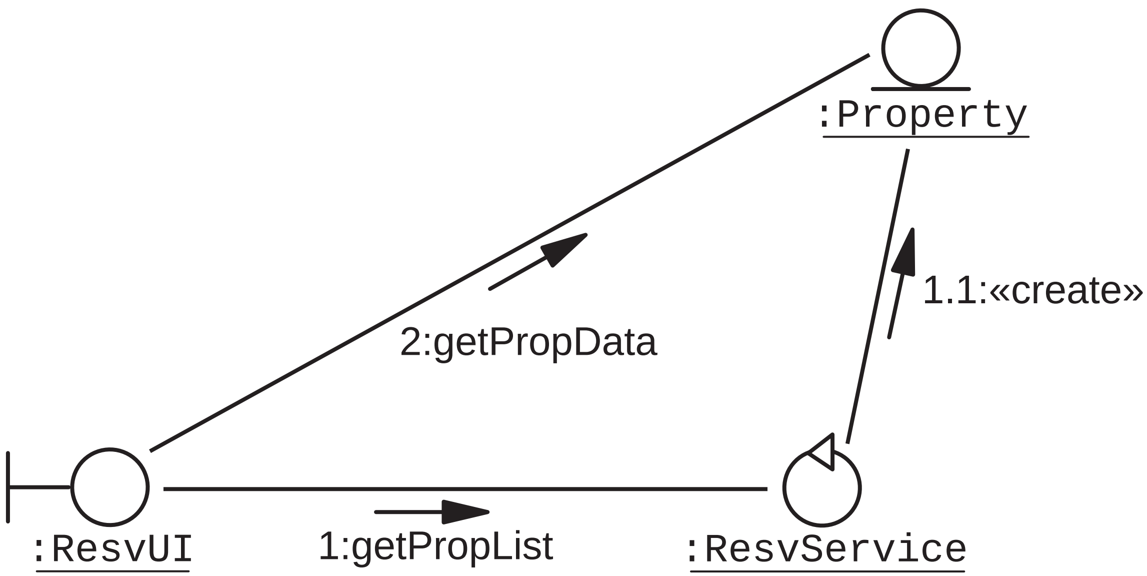
Identifying the Elements of a Collaboration Diagram

A UML Collaboration diagram is composed of the following elements:



Identifying the Elements of a Collaboration Diagram

A variation on the previous Collaboration diagram:



Identifying the Elements of a Collaboration Diagram

Message arrows can indicate:

- A method call
- Remote method invocation
- An asynchronous message

Sequence labels indicate:

- The order of the message
- The activity the message will invoke

Multi-objects represent a collection of related objects.

Performing Robustness Analysis

1. Select an appropriate use case.
2. Place the actor in the Collaboration diagram.
3. Analyze the use case (Activity diagram).
   For every action in the use case:
   a. Identify and add Boundary components.
   b. Identify and add Service components.
   c. Identify and add Entity components.
   d. Draw the associations between these components.
   e. Label the actions performed by each component to satisfy the interactions in the use case.

Step 1 — Select a Use Case

Select a Use Case: Create Reservation

1. Customer calls BookingAgent
2. BookingAgent selects "Make Reservation" icon
3. BookingAgent enters search criteria
   3.1 BookingAgent enters arrival and departure dates
   3.2 BookingAgent enters type of room
4. BookingAgent presses the "Search" button

...

11. BookingAgent enters customer name
12. BookingAgent presses the "Search" button
13. If a customer match is not found
    13.1 BookingAgent enters address info
    13.2 BookingAgent enters phone info
    13.3 BookingAgent presses "Add New Customer"
14. Else
    14.1 The system display match list
    14.2 BookingAgent selects the correct customer
    14.3 The System populates the GUI with customer info

...

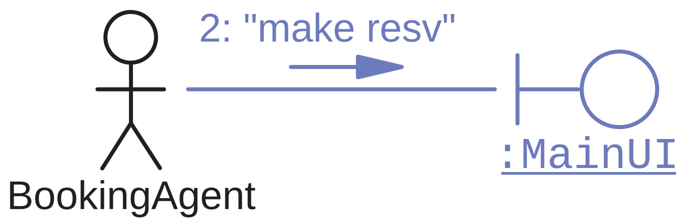
21. The System saves the reservation and displays ResvID
22. BookingAgent presses "Done"

Step 2 — Place the Actor in the Diagram

Place actor in the Collaboration diagram:

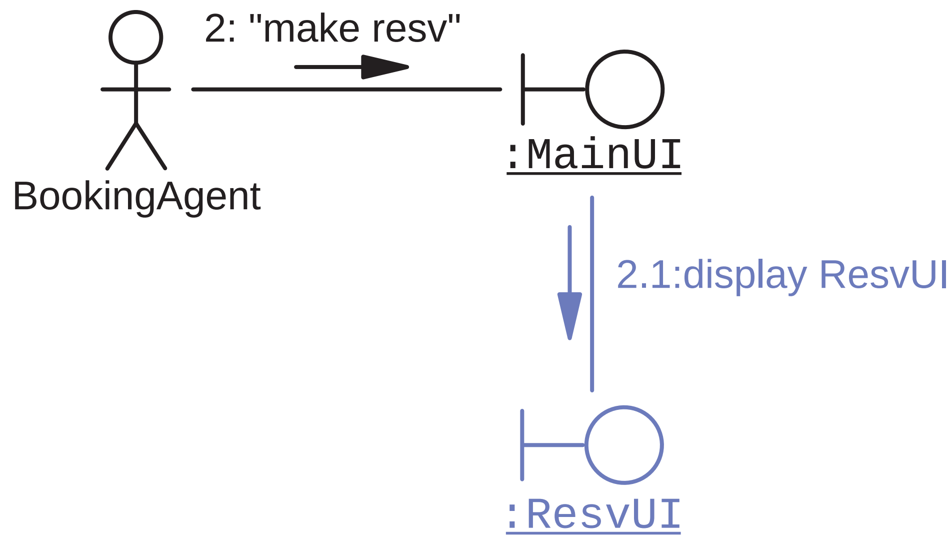
Step 3a — Identify Boundary Components

Activity: 2. BookingAgent selects "Make Reservation"



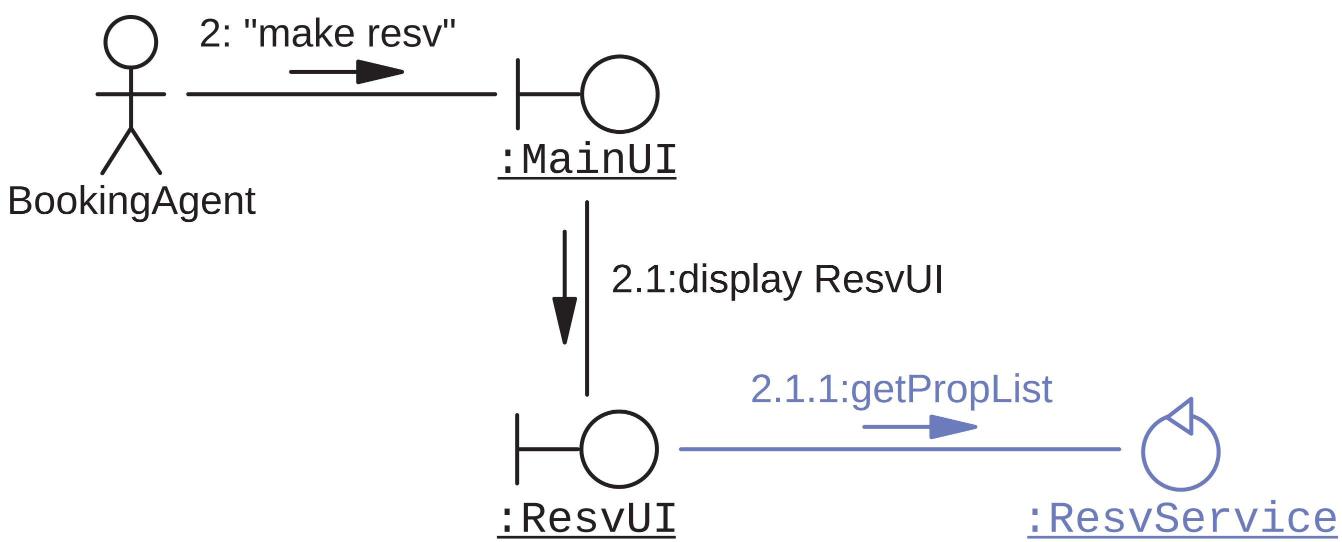
Step 3a — Identify Boundary Components

Activity: 2. BookingAgent selects “Make Reservation”



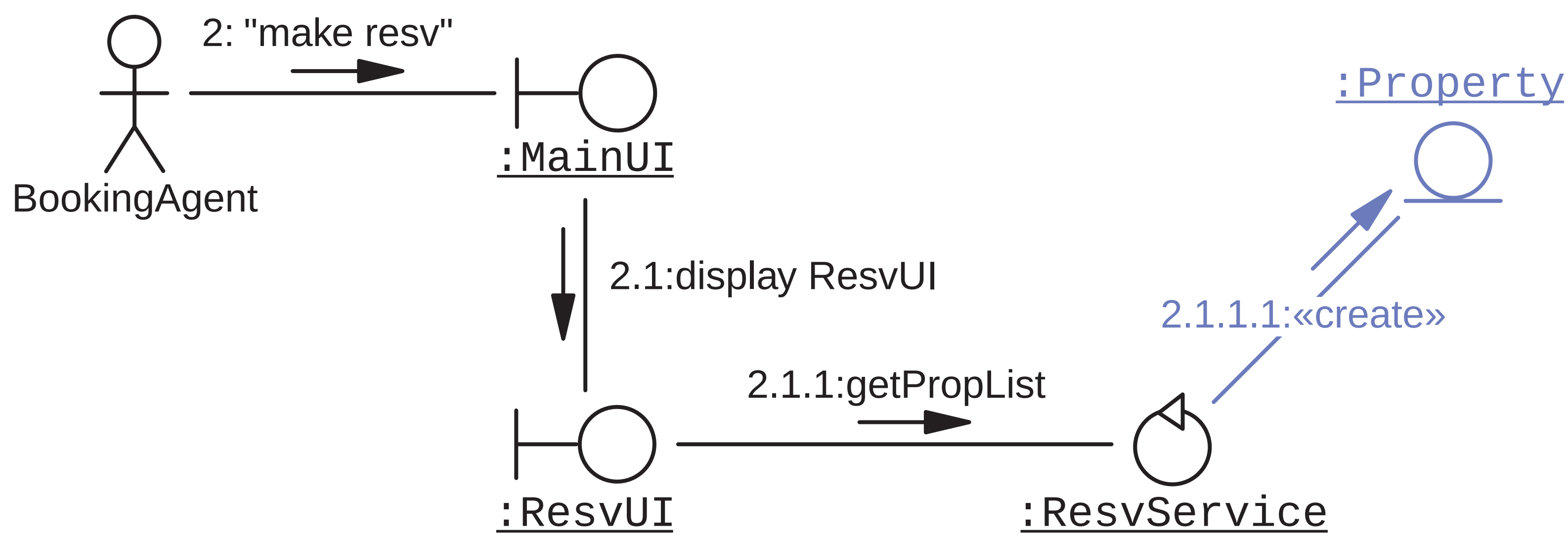
Step 3b — Identify Service Components

Activity: 2. BookingAgent selects “Make Reservation”



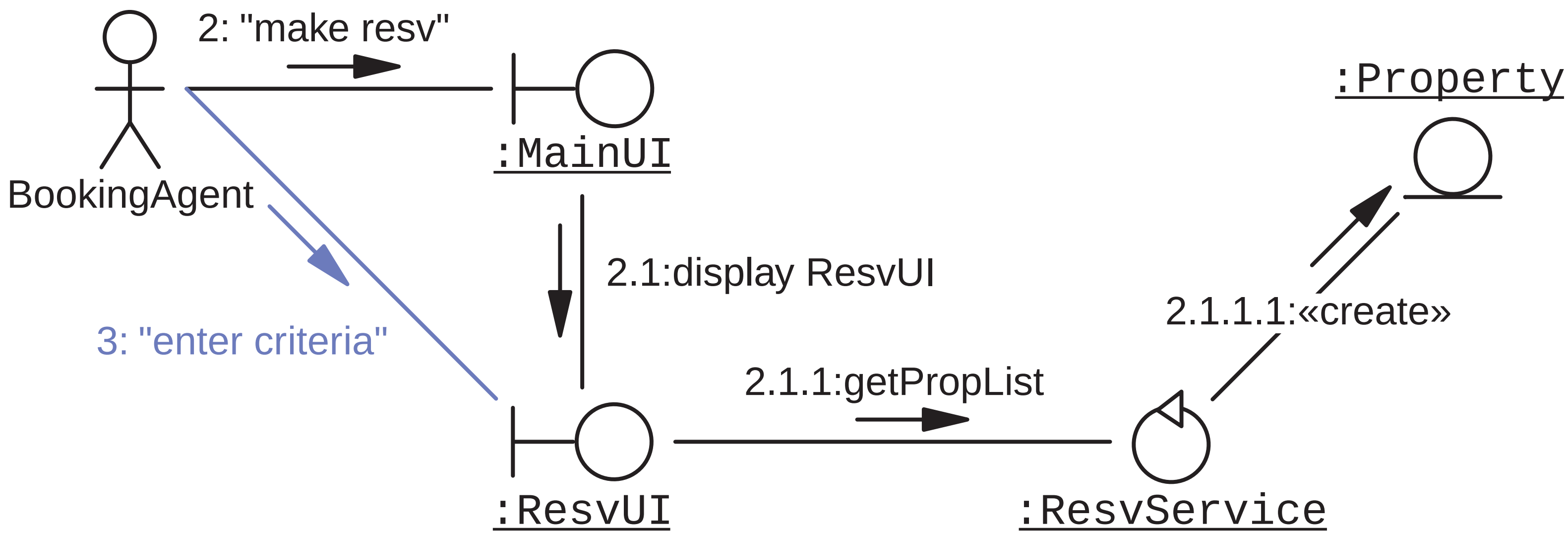
Step 3c — Identify Entity Components

Activity: 2. BookingAgent selects "Make Reservation"



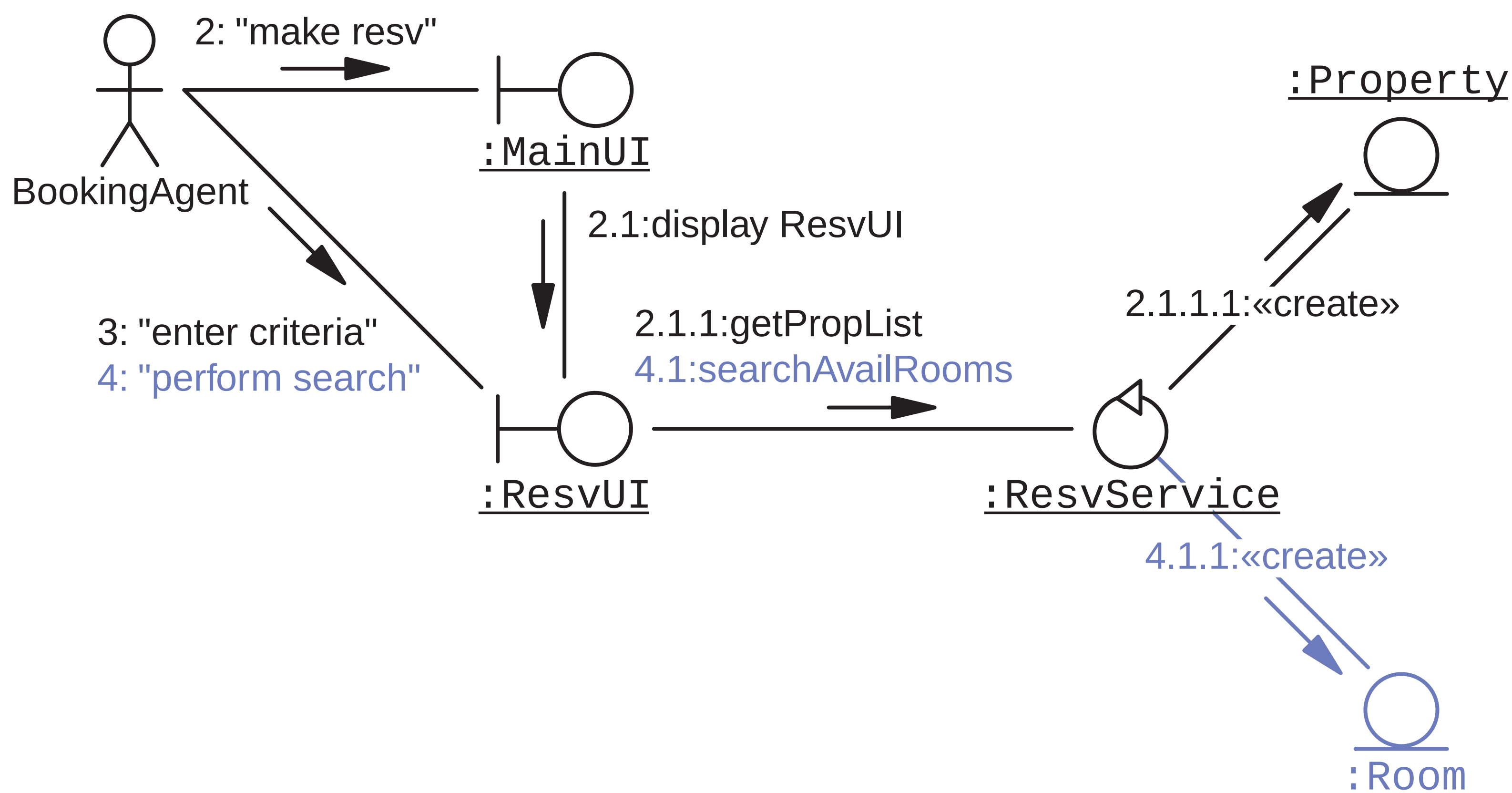
Analyze All Actions in the Activity Diagram

Activity: 3. BookingAgent enters search criteria



Analyze All Actions in the Activity Diagram

Activity: 4. BookingAgent presses the “Search” button

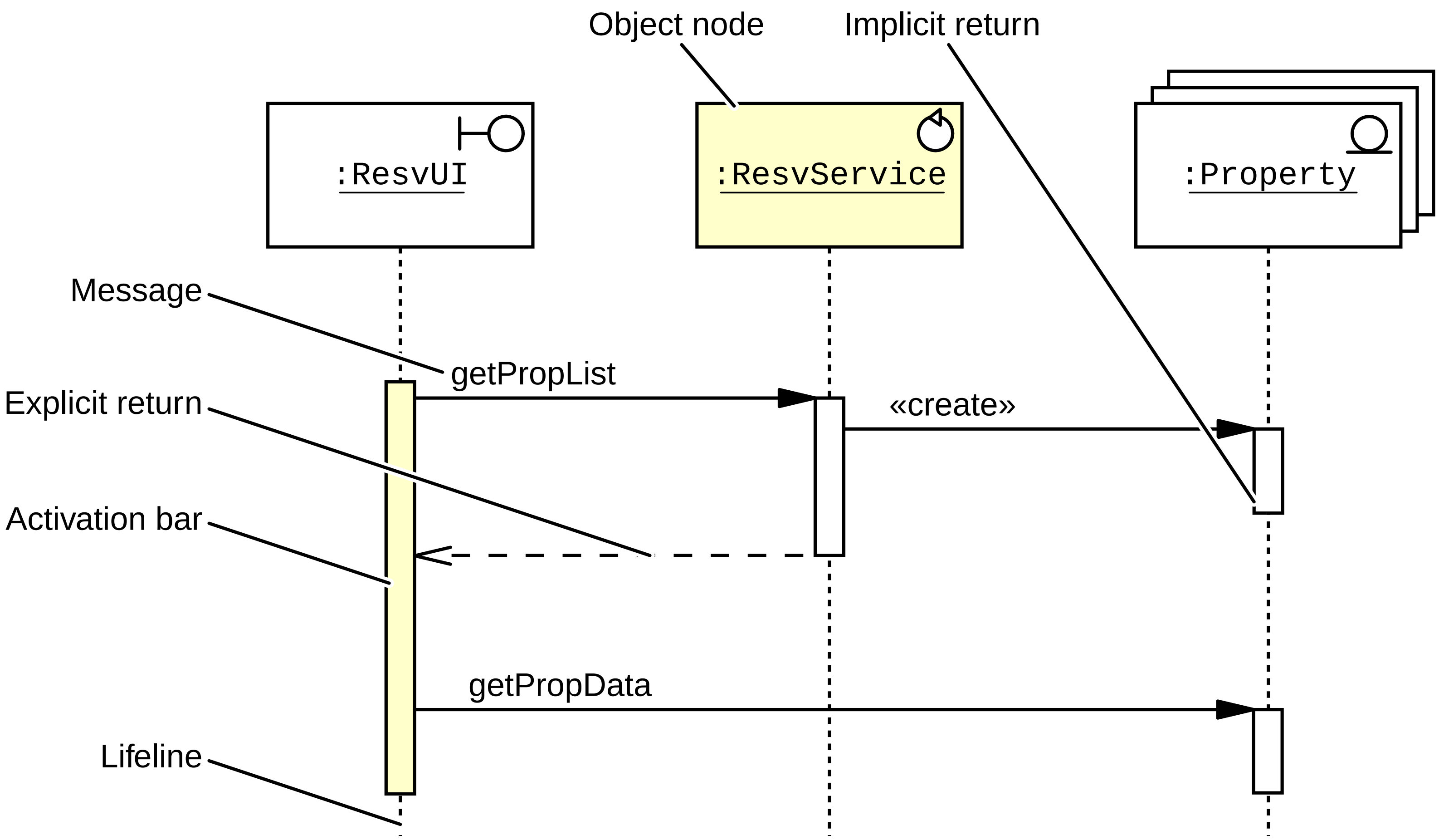


Converting the Collaboration Diagram into a Sequence Diagram

To provide a different perspective on the Robustness model you can convert the Collaboration diagram into a Sequence diagram. This diagram tends to be more useful for developers.

The next section describes UML Sequence diagrams.

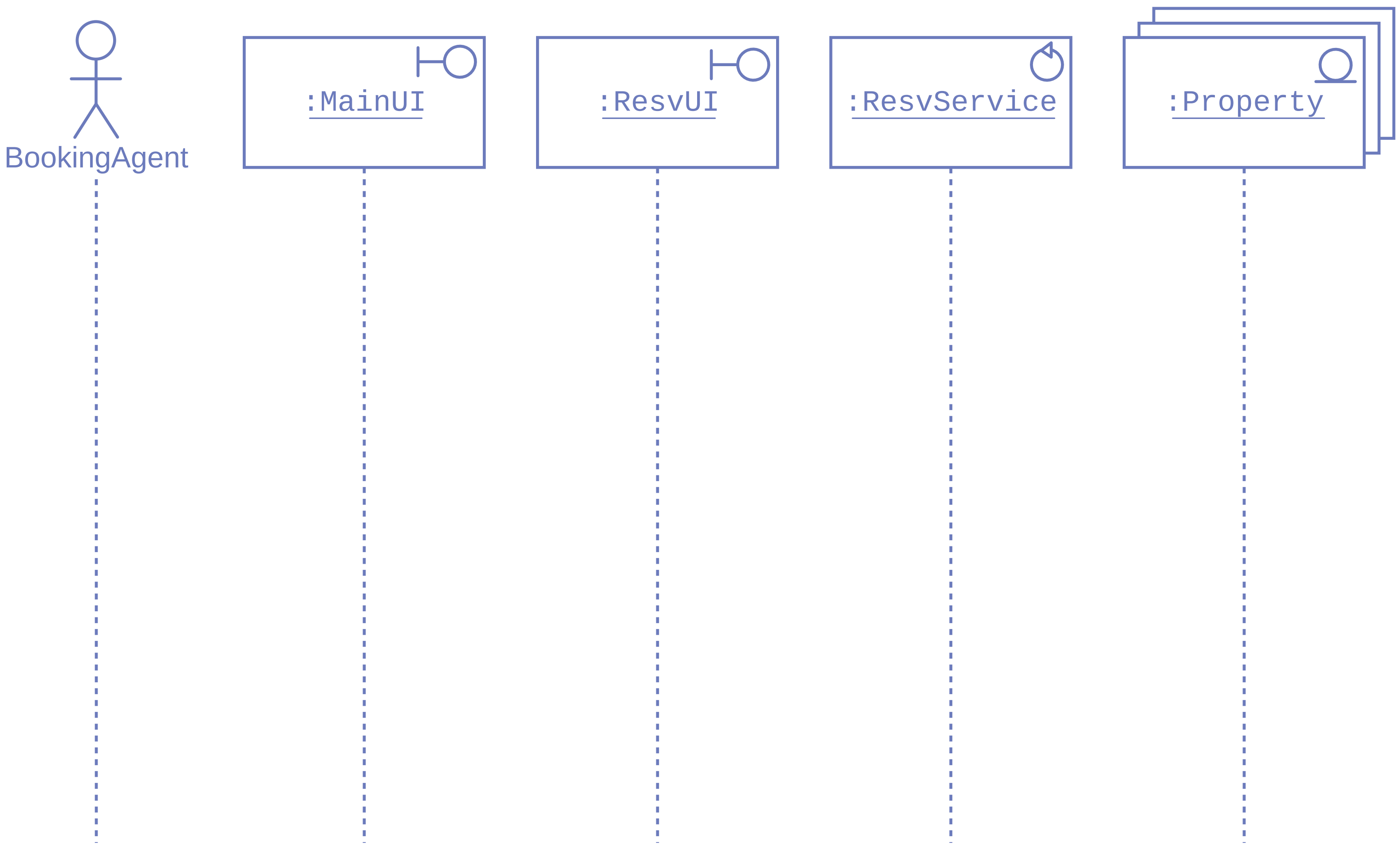
Identifying the Elements of a Sequence Diagram



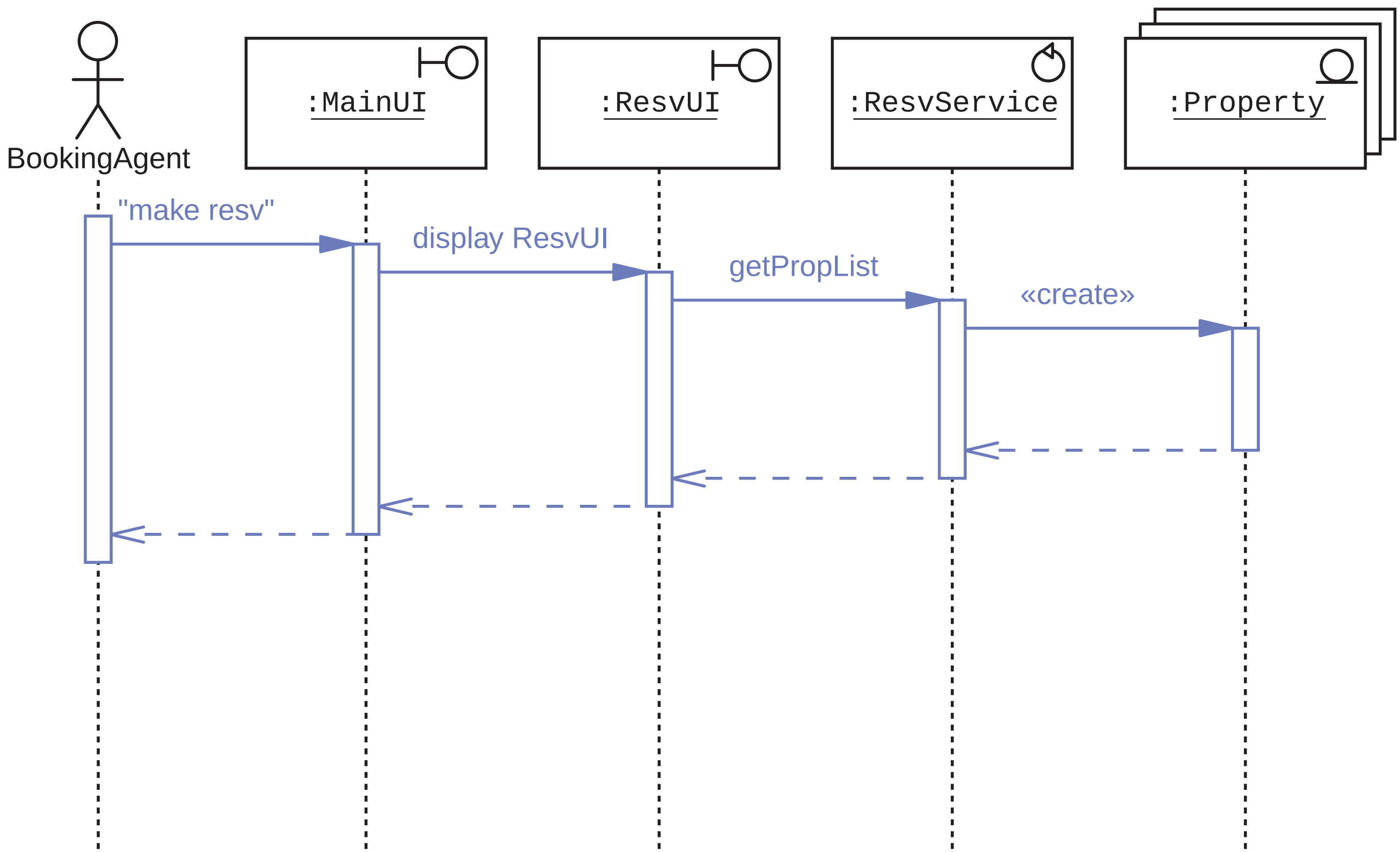
Clarifying the Design Model Using a Sequence Diagram

1. Arrange the collaborators at the top of the Sequence diagram to reflect the time-order of the first activity.
2. Add message links and activation bars for each message in the first activity.
3. Repeat Step 2 for each activity in the use case until the conversion is complete.

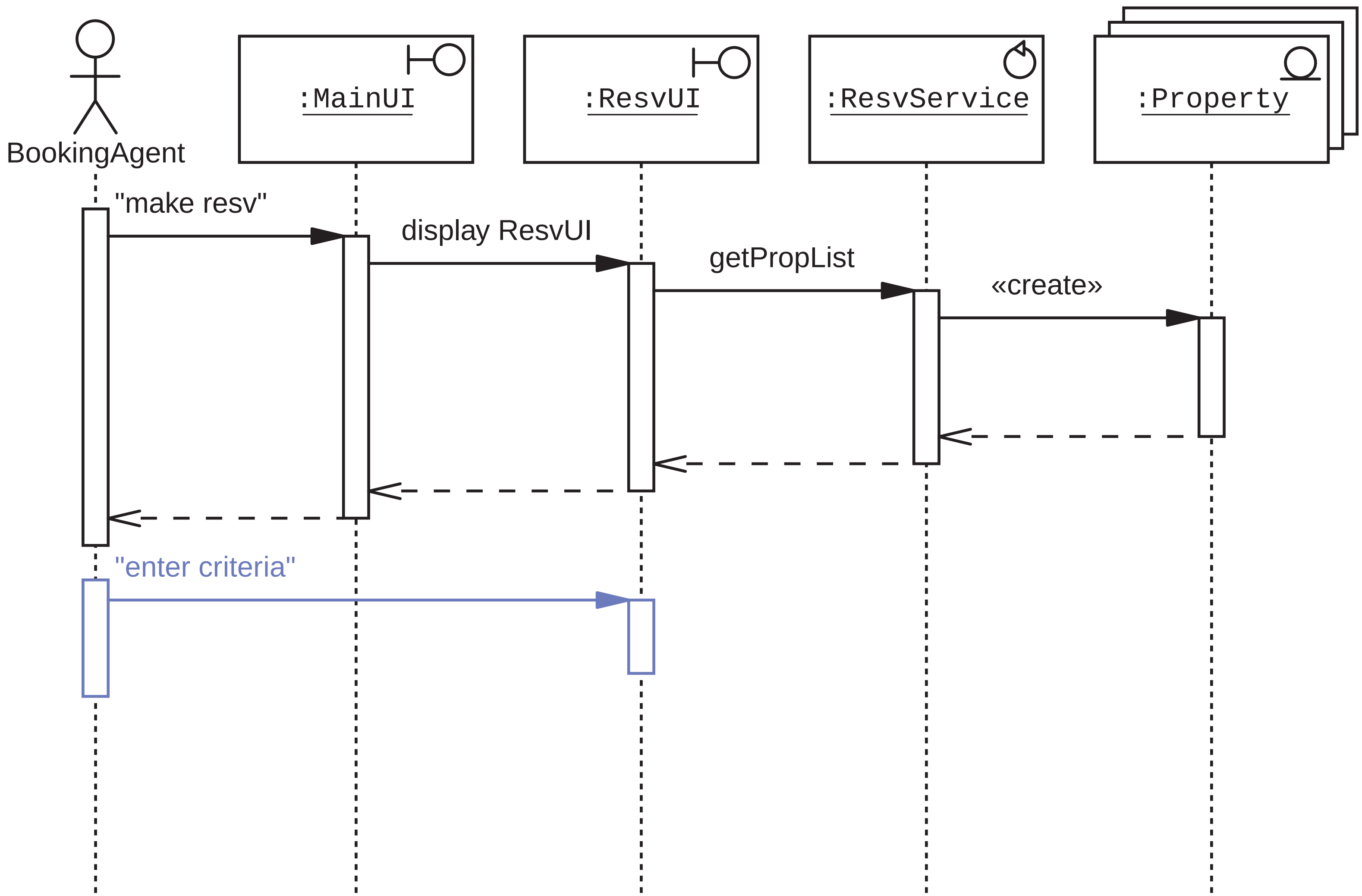
Step 1 – Arrange Components for the First Activity



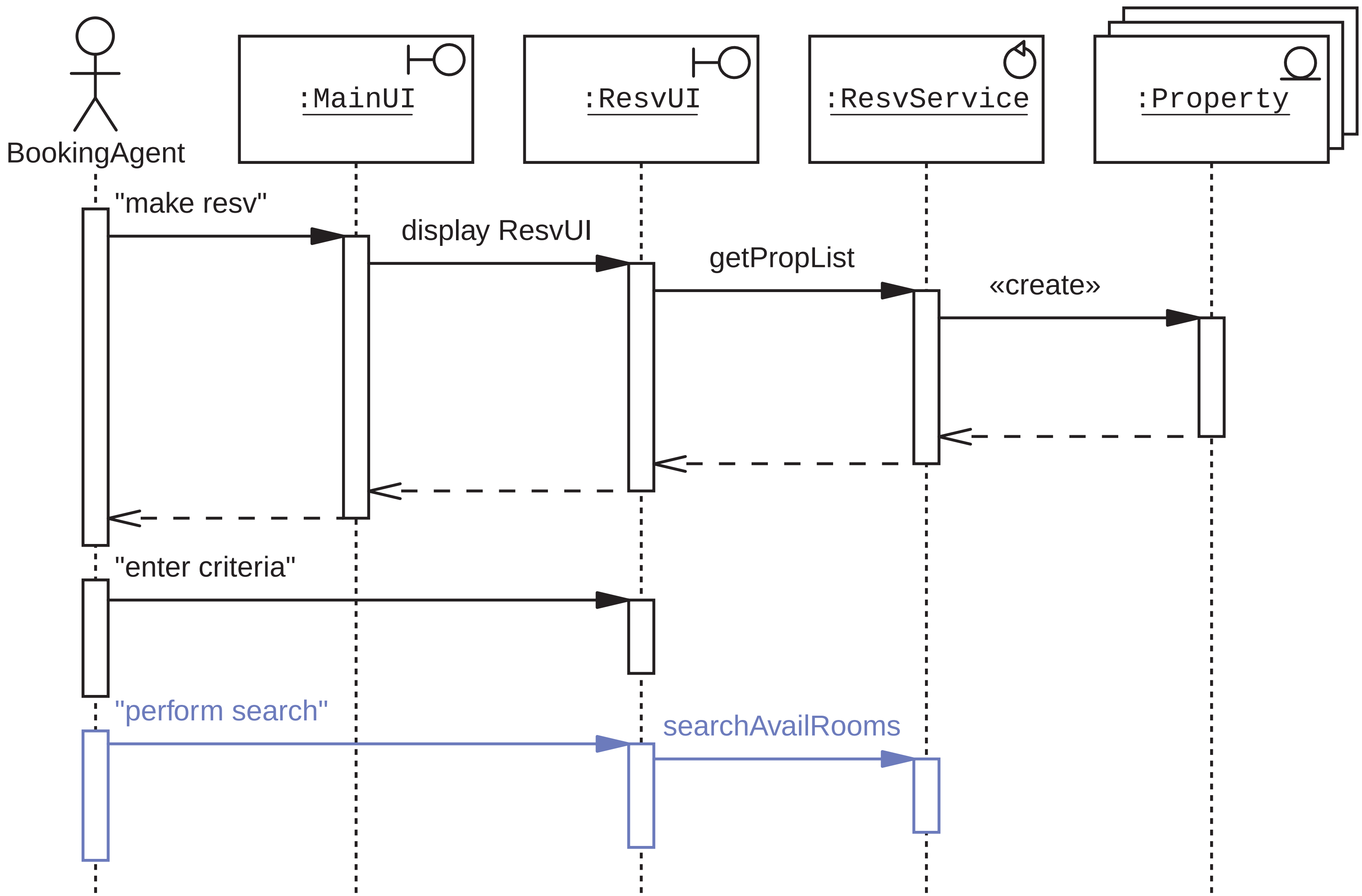
Step 2 – Add Message Links and Activation Bars



Step 3 — Repeat Step 2 For Each Activity



Step 3 — Repeat Step 2 For Each Activity



Summary

- Robustness analysis creates a model of the design components that satisfy a use case. This is called the Design model.
- The Design model is usually visualized with a UML Collaboration diagram.
- The Design model can be converted into a Sequence diagram to provide a another view of use case collaboration.

Module 11

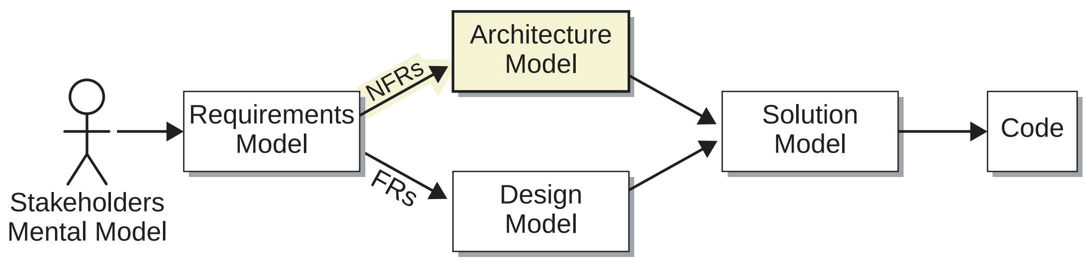
Introducing Fundamental Architectural Concepts

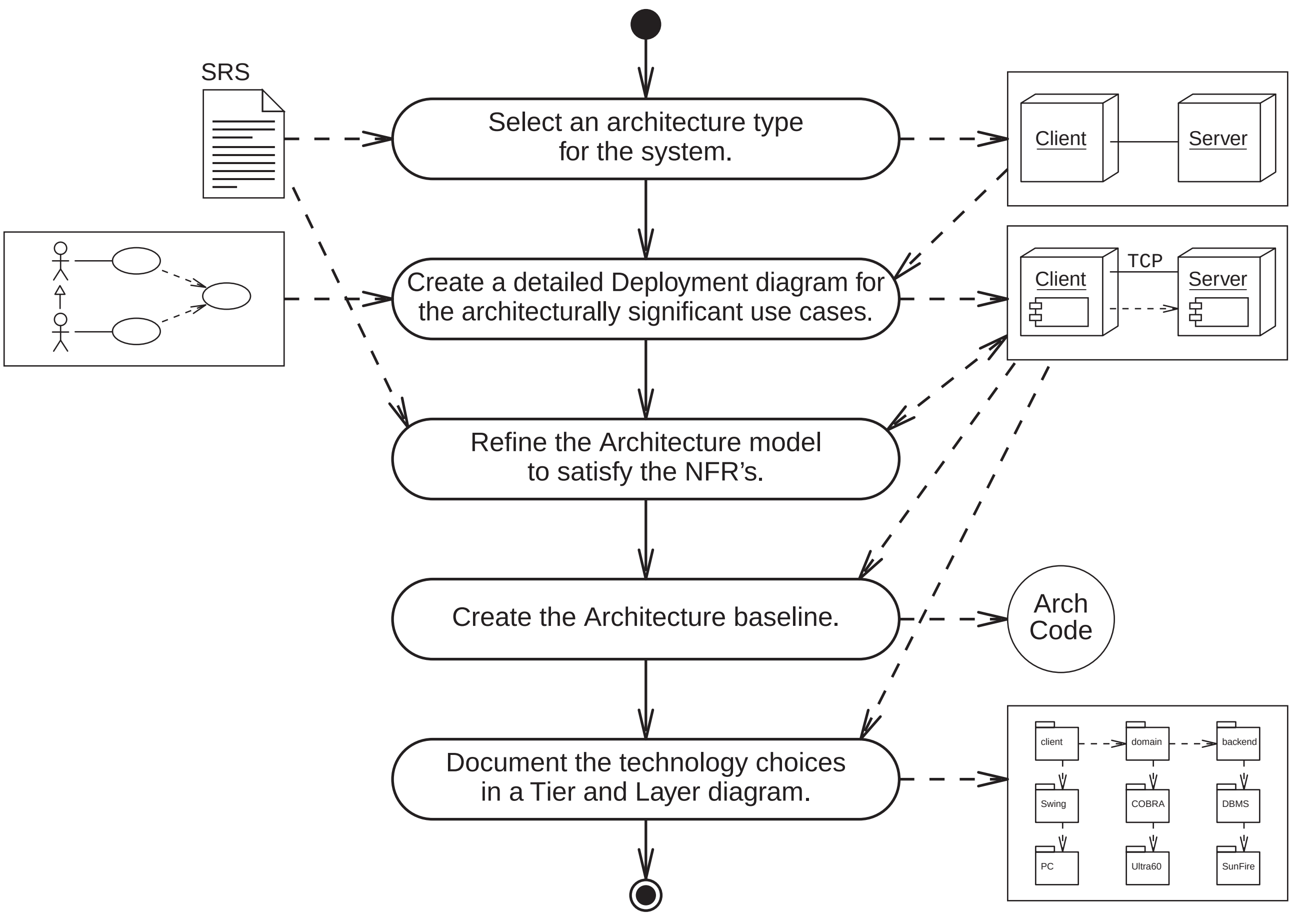
Objectives

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

- Justify the need for the architect role
- Distinguish between architecture and design
- Describe the SunTone Architecture Methodology

Process Map


Architecture Model
NFRs
Requirements Model
Solution Model
Code
FRs
Design Model
Stakeholders Mental Model


SRS
Select an architecture type for the system.
Client
Server
Create a detailed Deployment diagram for the architecturally significant use cases.
TCP
Client
Server
Refine the Architecture model to satisfy the NFR's.
Create the Architecture baseline.
Arch Code
Document the technology choices in a Tier and Layer diagram.
client
domain
backend
Swing
COBRA
DBMS
PC
Ultra60
SunFire

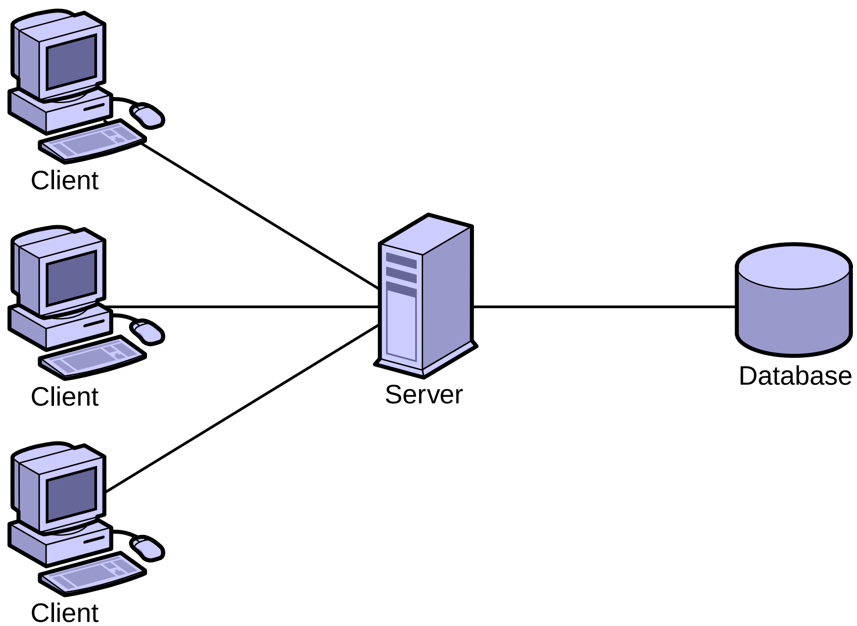
Justifying the Need for the Architect Role

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

- Scale
- Distribution

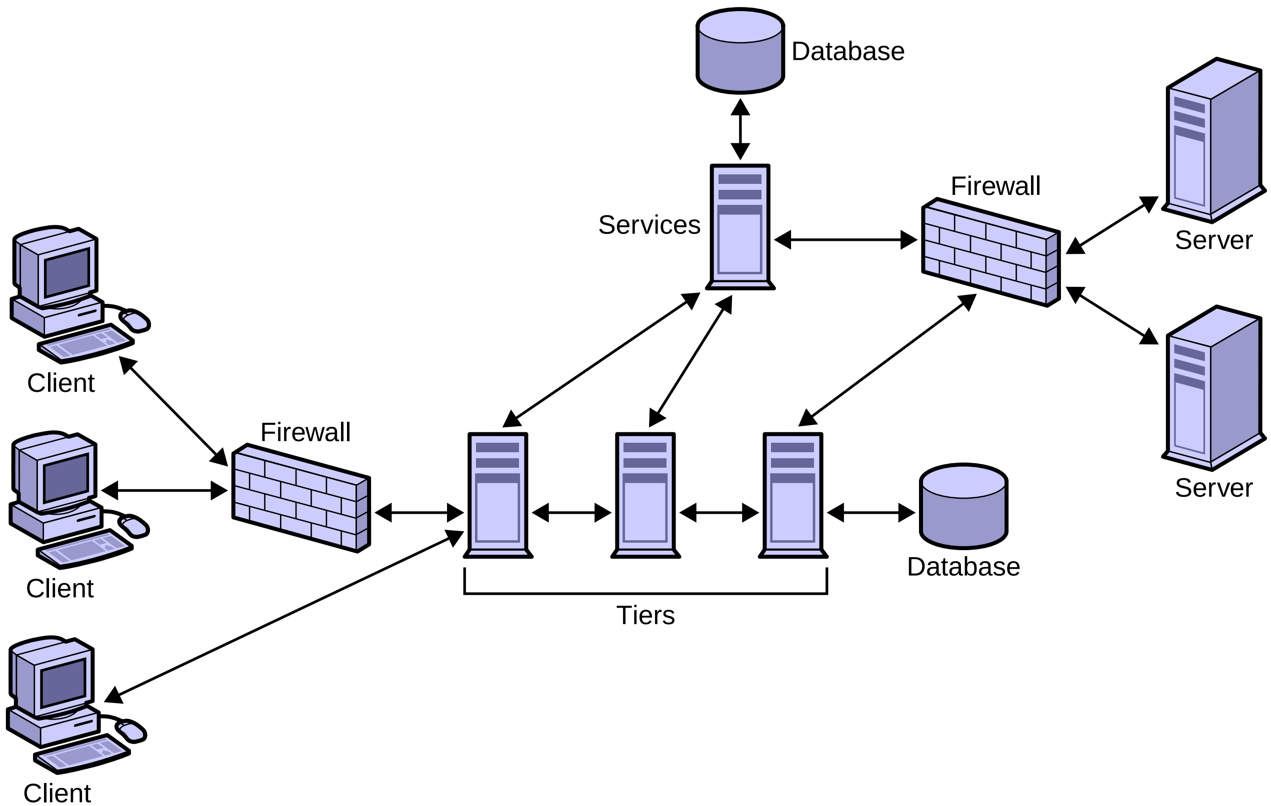
Risks Associated With Large-Scale, Distributed Enterprise Systems

Minimally distributed systems (such as client-server)



Risks Associated With Large-Scale, Distributed Enterprise Systems

Highly distributed systems



Quality of Service

- Project failure
- Non-functional requirements:
  - Constraints
  - Systemic qualities

Risk Evaluation and Control

- Risk evaluation:
  - Evaluate risks and their mitigation strategies
  - Compare risks based on cost and probability of occurrence
- Cost analysis:
  - Some options might cost little, but leave some risk
  - Other options might cost more, but control the risk completely
  - *Do nothing* option – Assume the risk is realized and examine the cost impact

The Role of the Architect

- Technology responsibilities:
  - List of assumptions and constraints
  - Risk identification and mitigation plan
  - Deployment environment description
  - Interaction diagrams
- Management responsibilities:
  - Convince other stakeholders of the validity of decisions
  - Mentor other team members

Distinguishing Between Architecture and Design

The table shows how architects differ from designers.

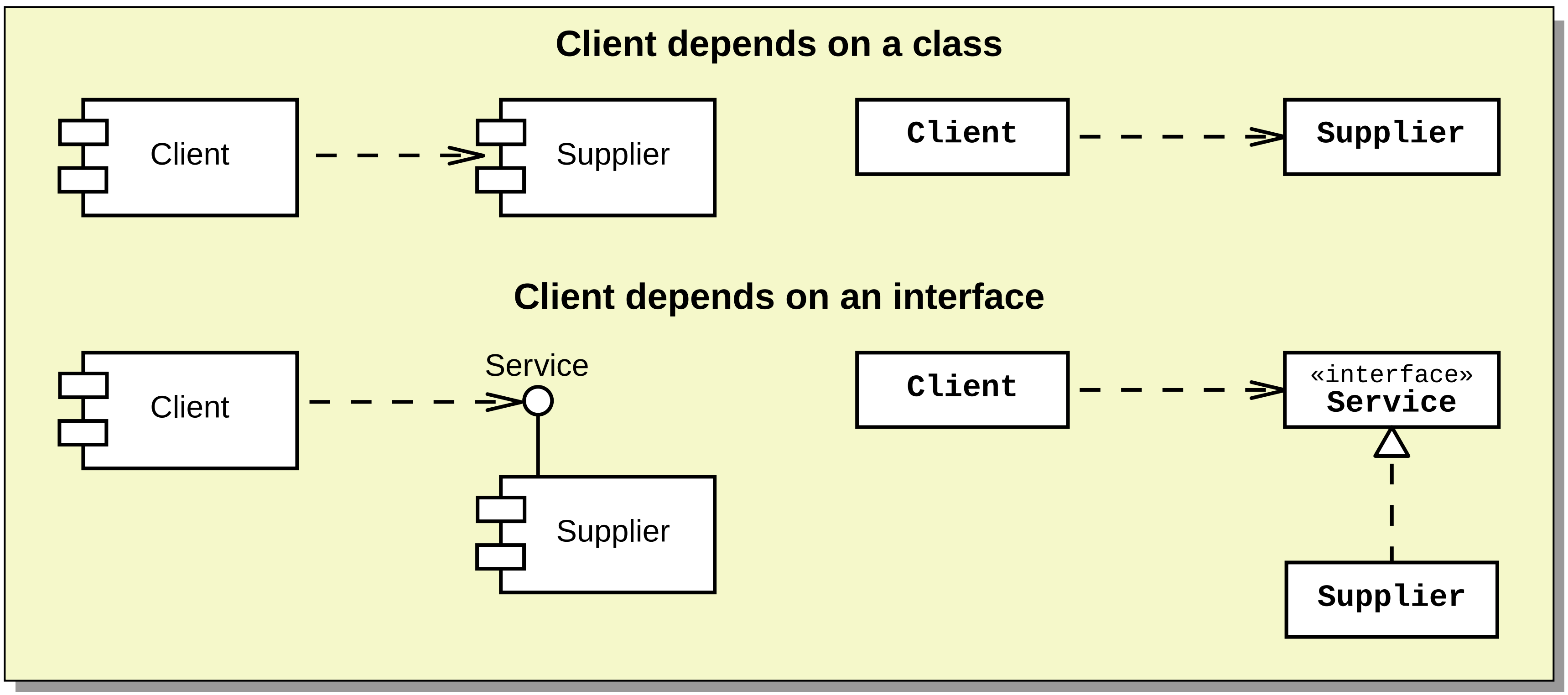
	Architect	**Designer**
Abstraction level	High/broad Focus on few details	Low/specific Focus on many details
Deliverables	System and subsystem plans, architectural prototype	Component designs, code specifications
Area of focus	Nonfunctional requirements, risk management	Functional requirements

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

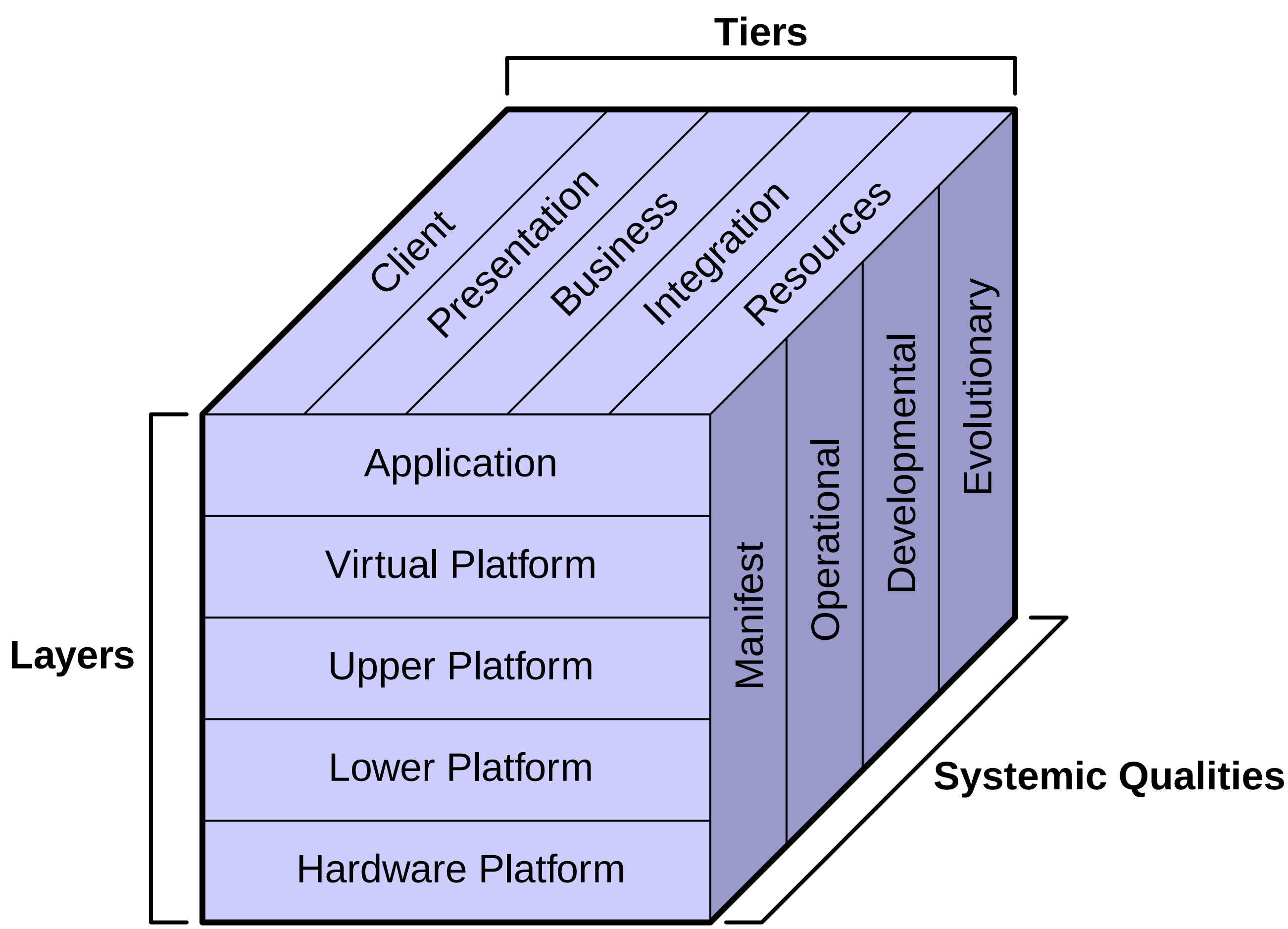
- An architect plans systems using a pattern-based reasoning process.
- An architect must be familiar with a variety of pattern catalogs to be effective.
- Types of patterns:
  - Design patterns define structure and behavior to construct effective and reusable OO software components to support functional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

Applying the SunTone Architecture Methodology

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.

Summary

- The role of the Architect is to:
  - Identify and mitigate project risks
  - Create an Architectural baseline
- Difference between architecture and design:
  - Design provides a set of components to implement a use case.
  - Architecture provides a template into which the designed components are realized.
- The SunTone Architecture Methodology organizes the Architecture model across three dimensions: tiers, layers, and systemic qualities.

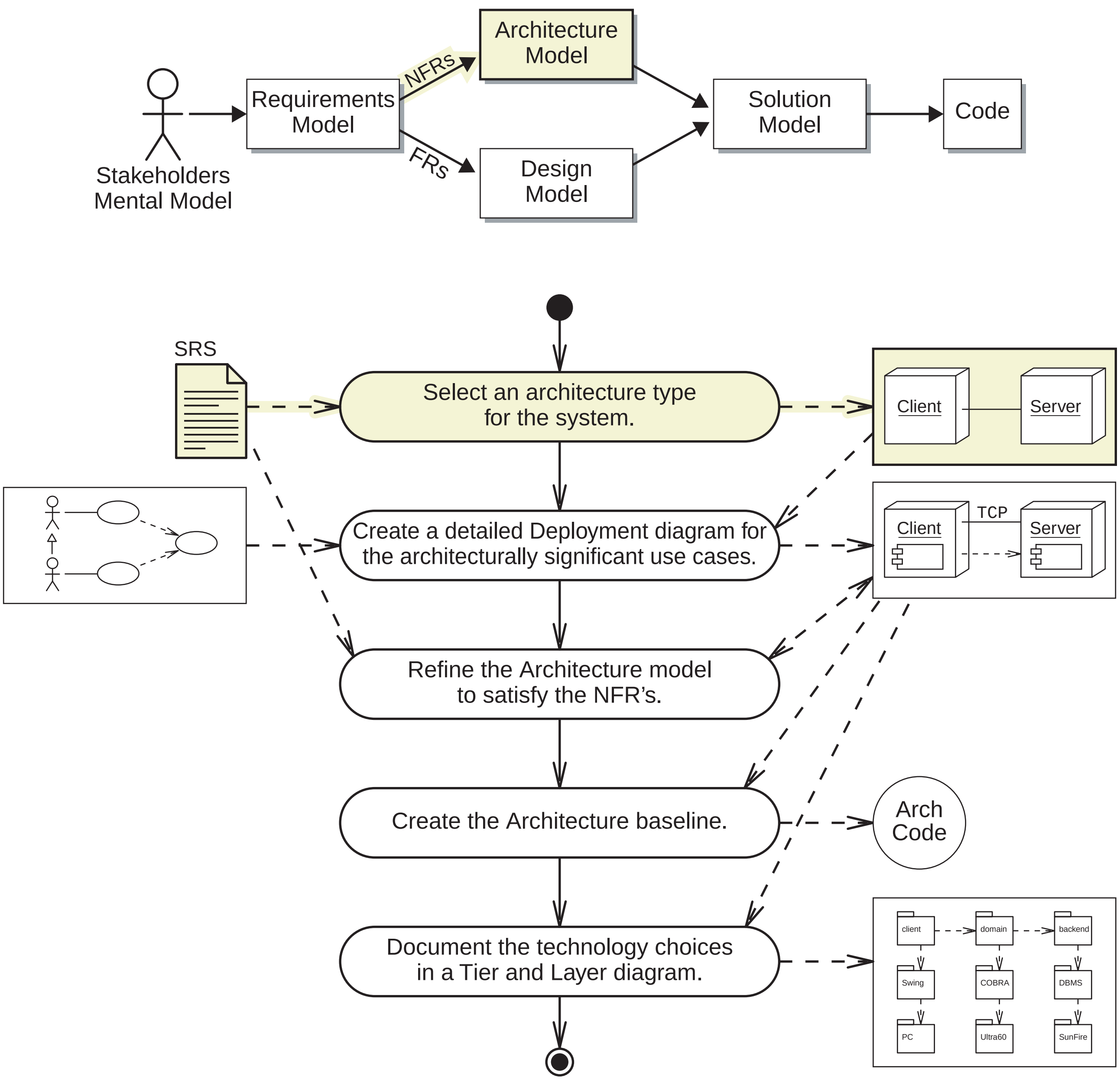
Module 12

Exploring the Architecture Workflow

Objectives

- Describe the Architecture workflow
- Describe the diagrams of the key architecture views
- Select the Architecture type
- Create the Architecture workflow artifacts

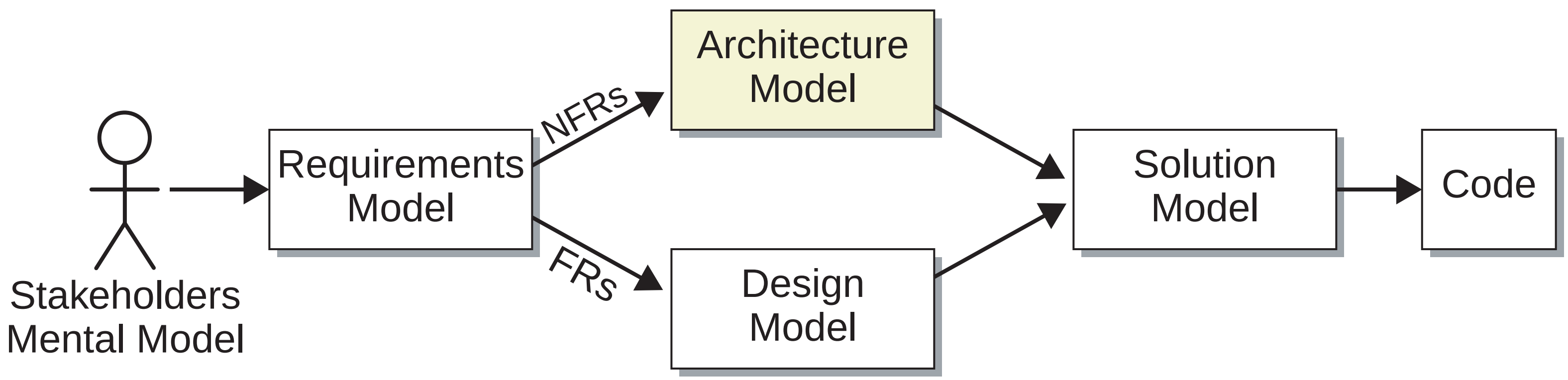
Process Map



Stakeholders Mental Model
Requirements Model
NFRs
FRs
Architecture Model
Design Model
Solution Model
Code
SRS
Select an architecture type for the system.
Create a detailed Deployment diagram for the architecturally significant use cases.
Refine the Architecture model to satisfy the NFR's.
Create the Architecture baseline.
Document the technology choices in a Tier and Layer diagram.
Client
Server
TCP
Arch Code
client
domain
backend
Swing
COBRA
DBMS
PC
Ultra60
SunFire

Exploring the Architecture Workflow

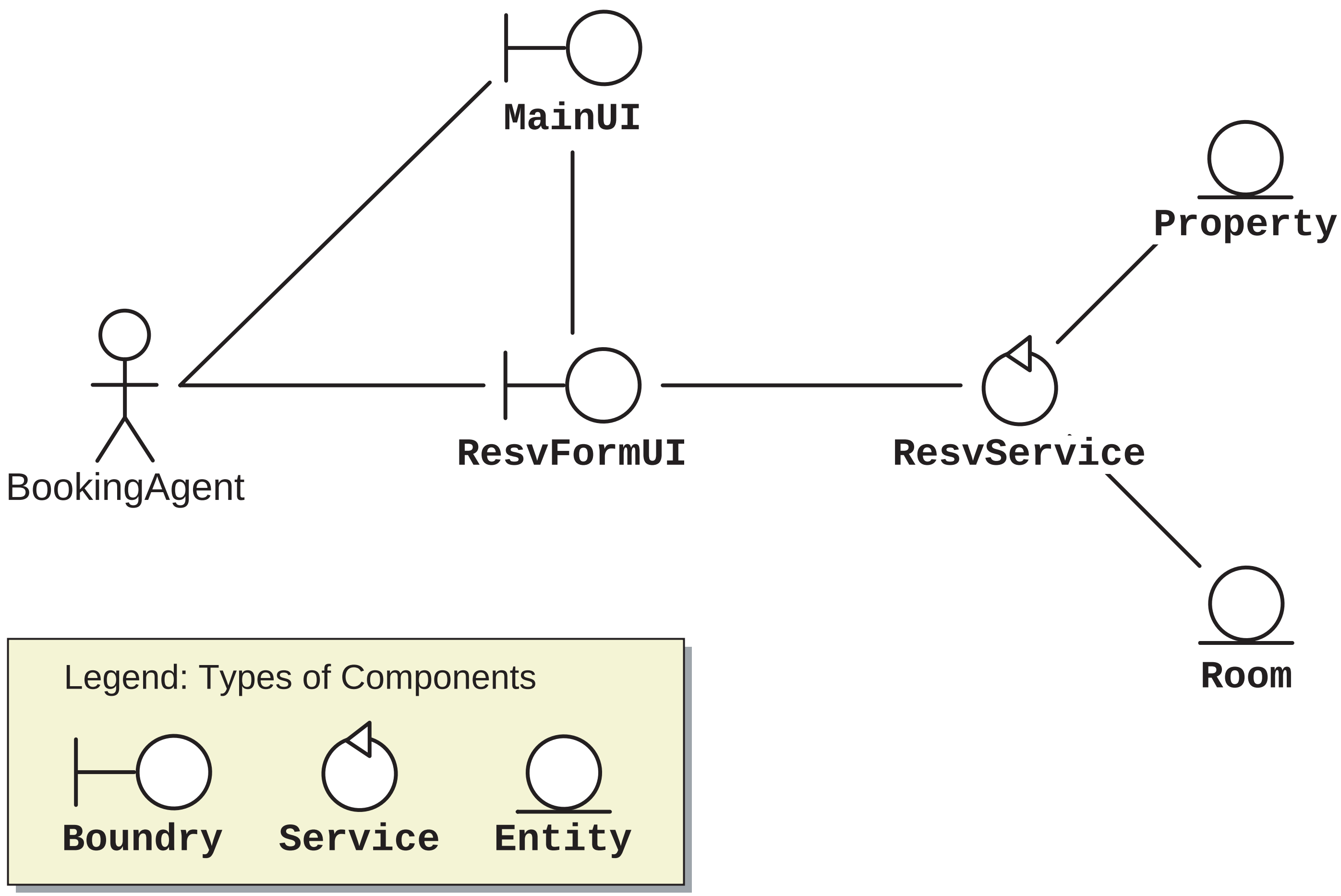
The Architecture model is essential to the creation of the Solution model:



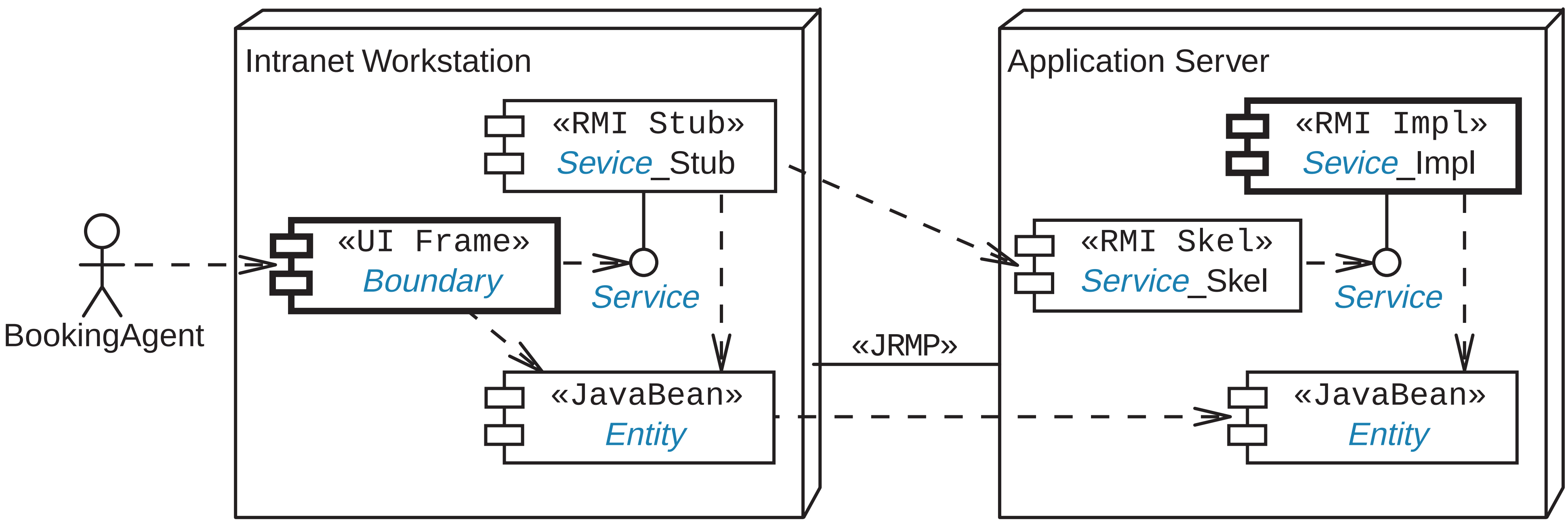
Introducing the Architecture Workflow

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.

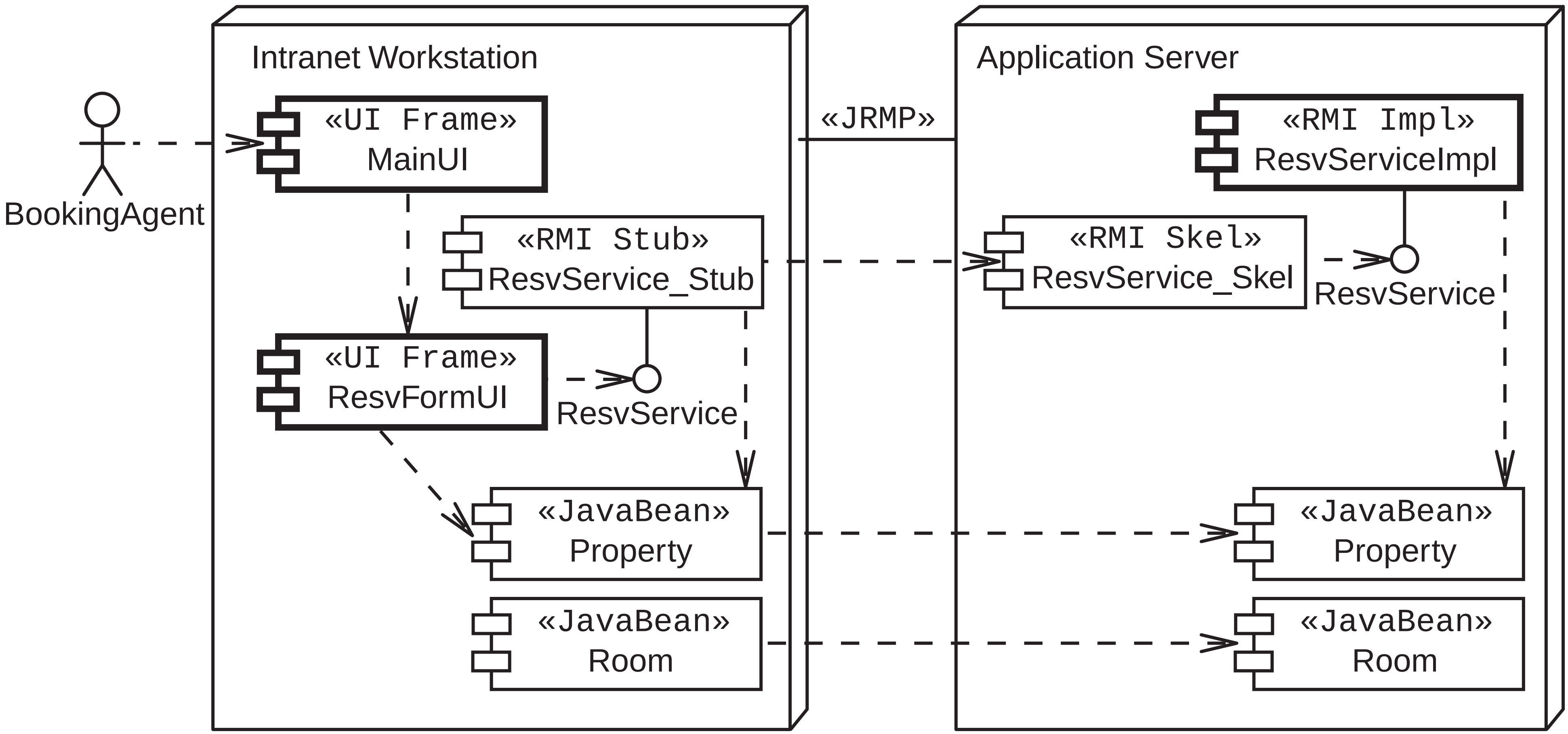
Example Design Model



Example Architecture Template



Example Solution Model



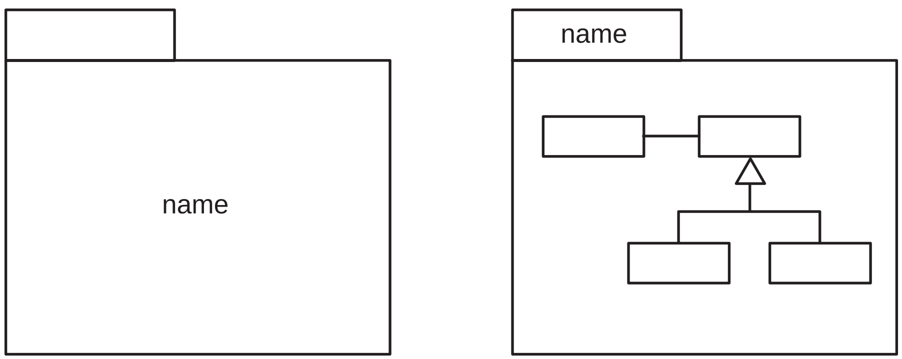
Architectural Views

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

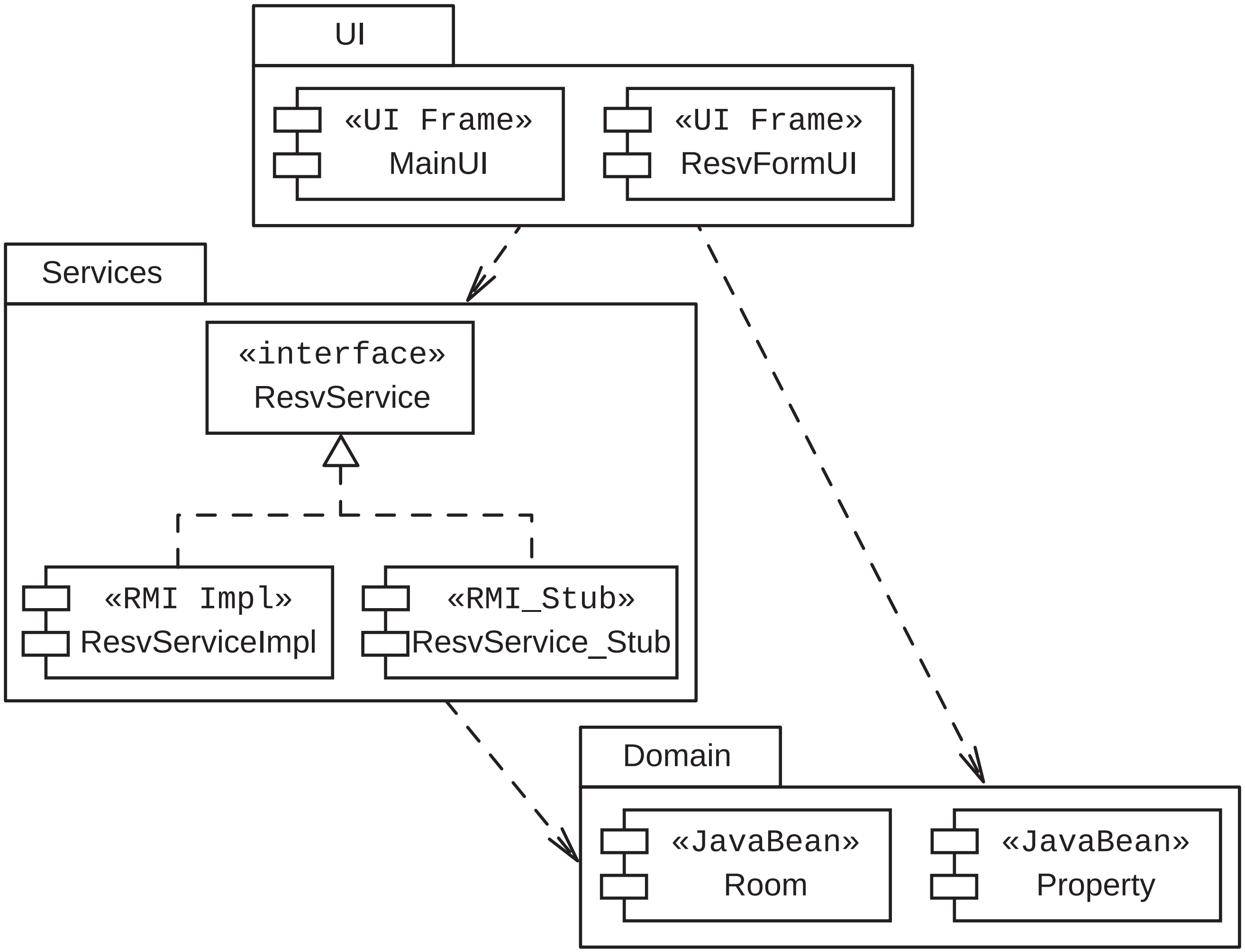
Identifying the Elements of a Package Diagram

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

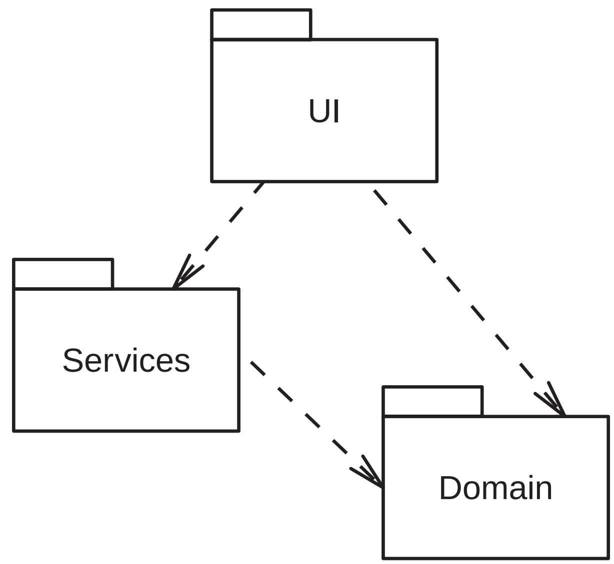


  - The package name can be placed in the body box or in the name box.
  - You can place any UML entity in a package, including other packages.

Example Package Diagram

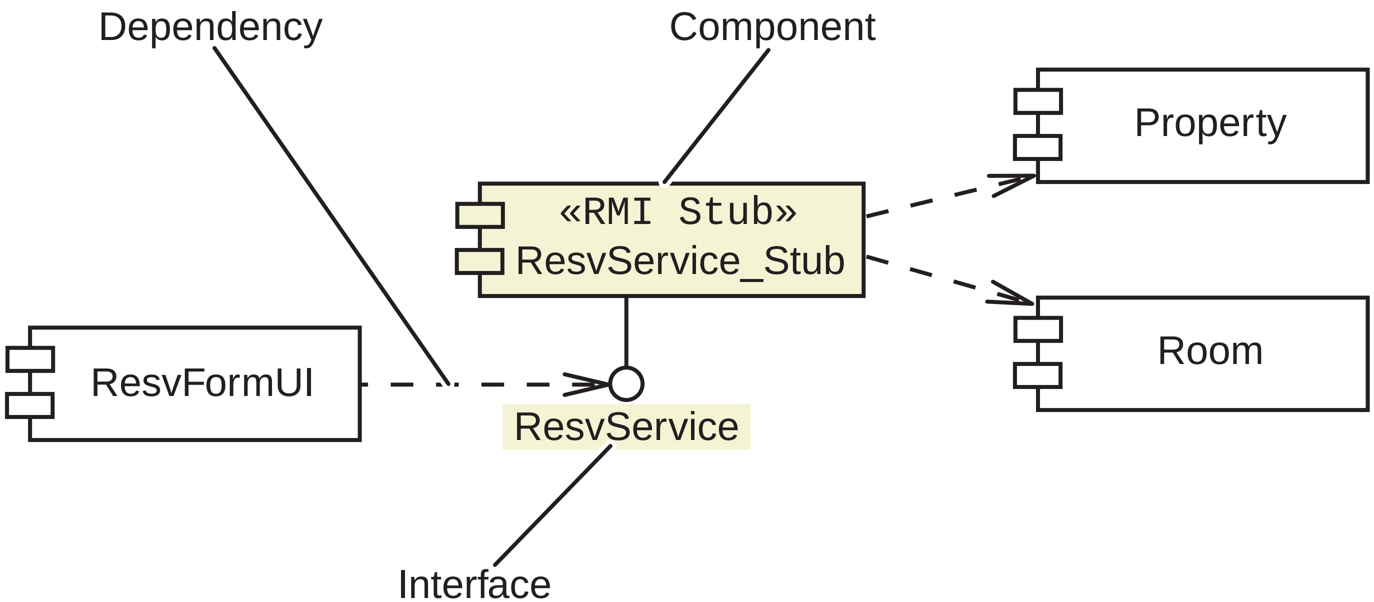


An Abstract Package Diagram



Identifying the Elements of a Component Diagram

A UML Component diagram is composed of the following elements:

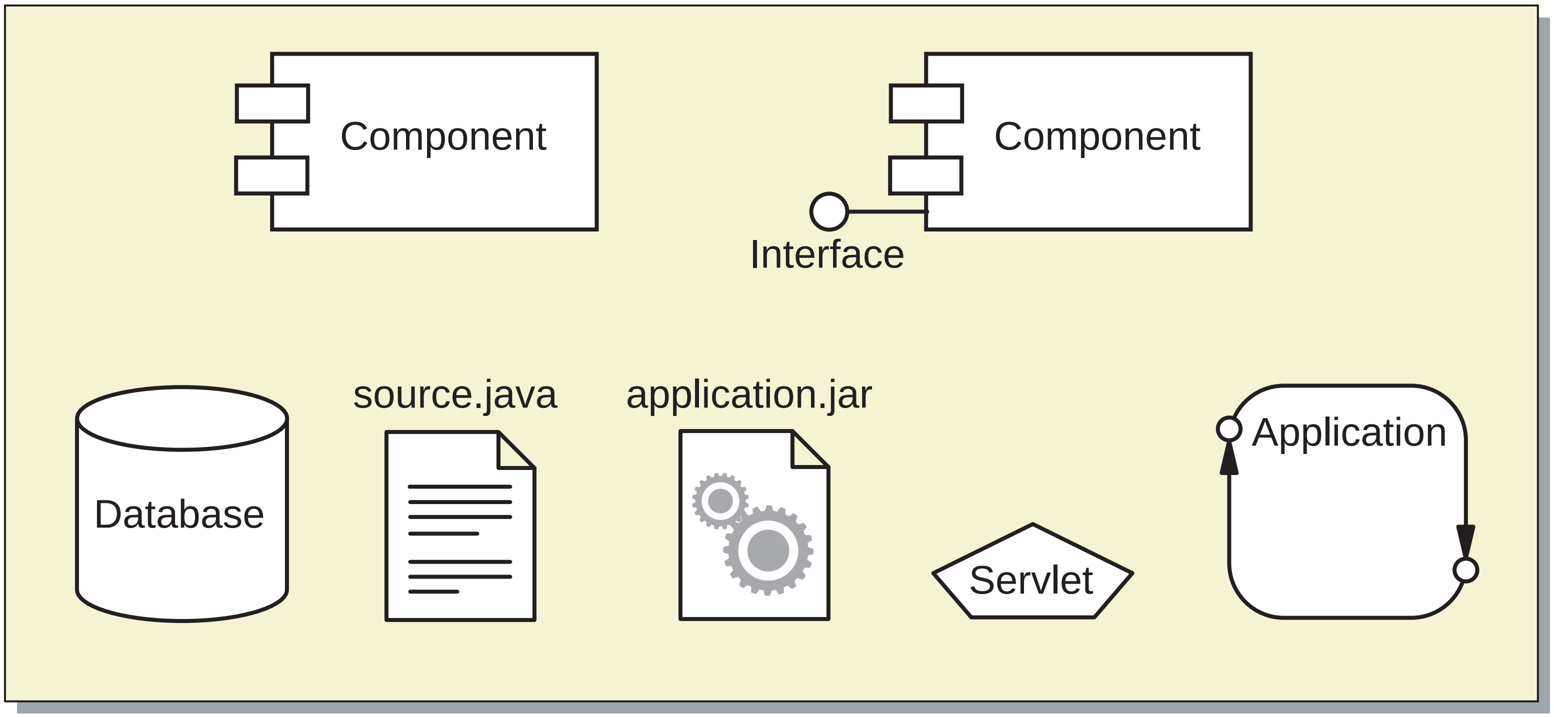


Characteristics of a Component

- 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.

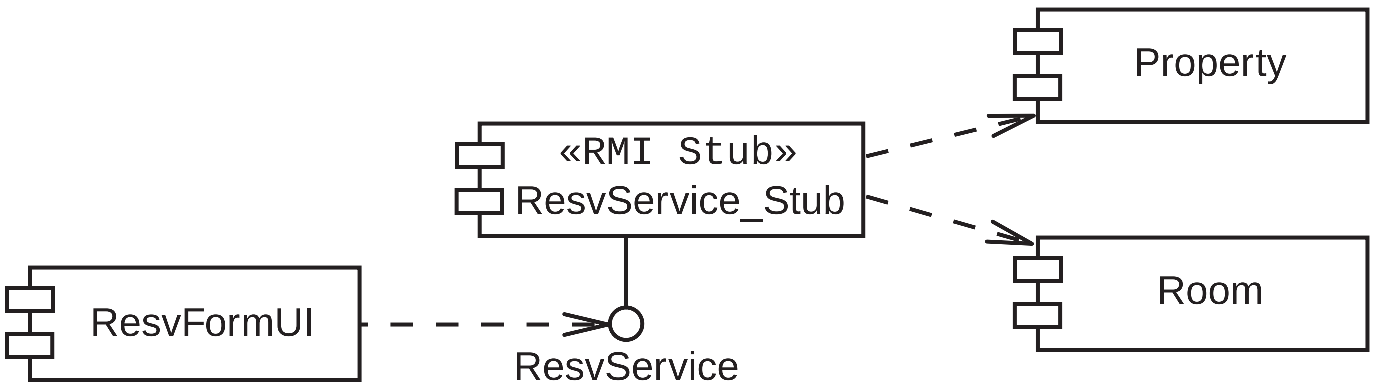
Types of Components

A component is any physical software unit:



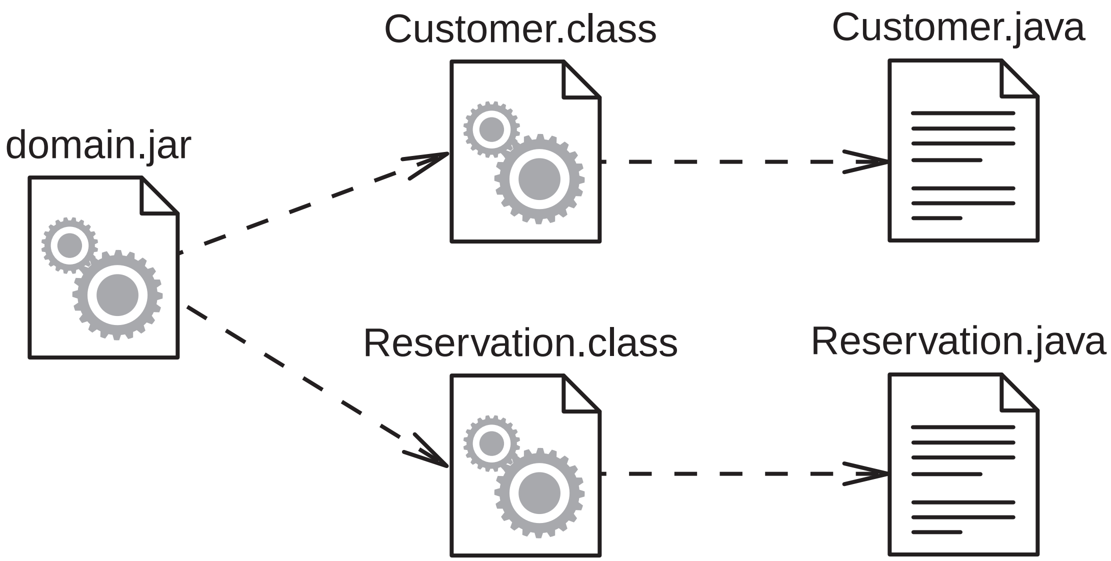
Example Component Diagrams

Component diagrams can show software dependencies:

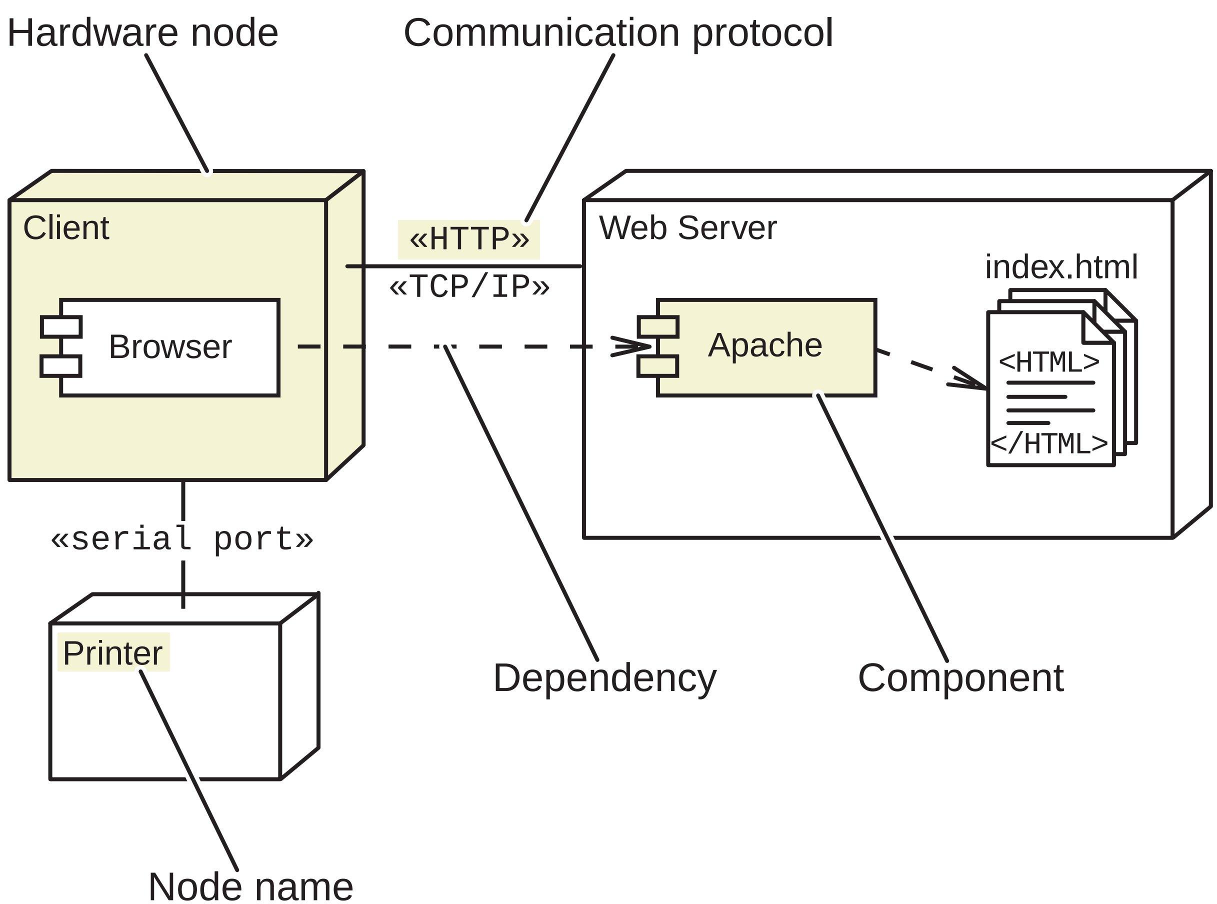


Example Component Diagrams

Component diagrams can represent build structures:



Identifying the Elements of a Deployment Diagram



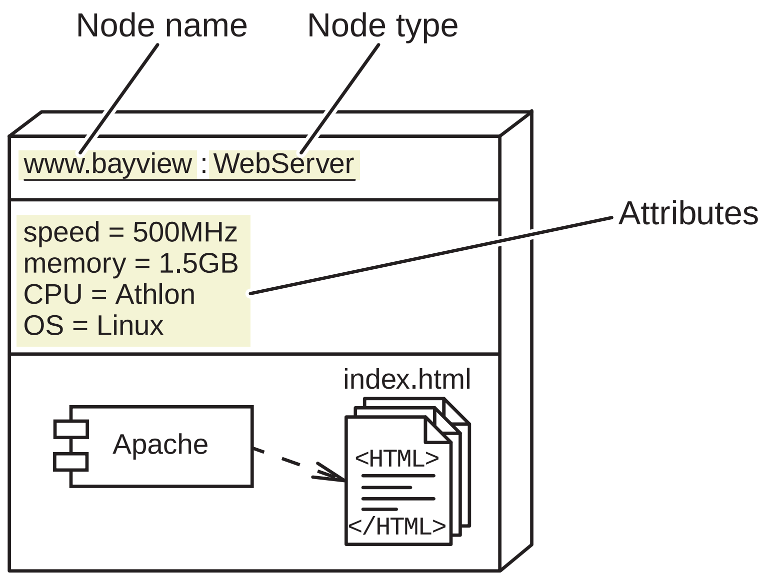
The Purpose of a Deployment Diagram

- 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:

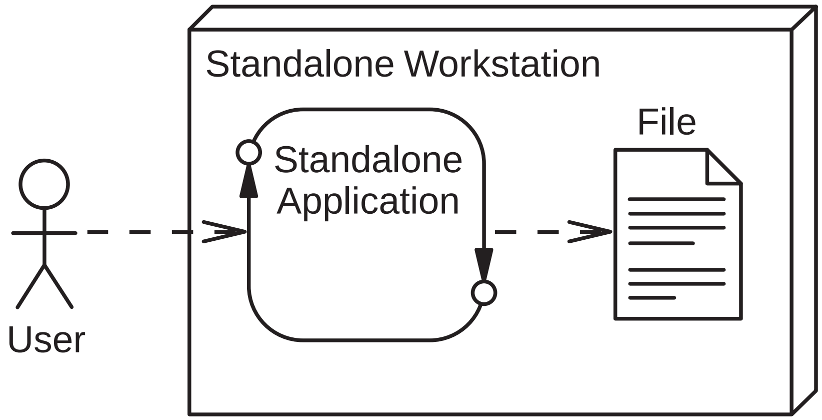
- The platform constraints in the system requirements
- The modes of user interaction
- The persistence mechanism
- Data and transactional integrity

Selecting the Architecture Type

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

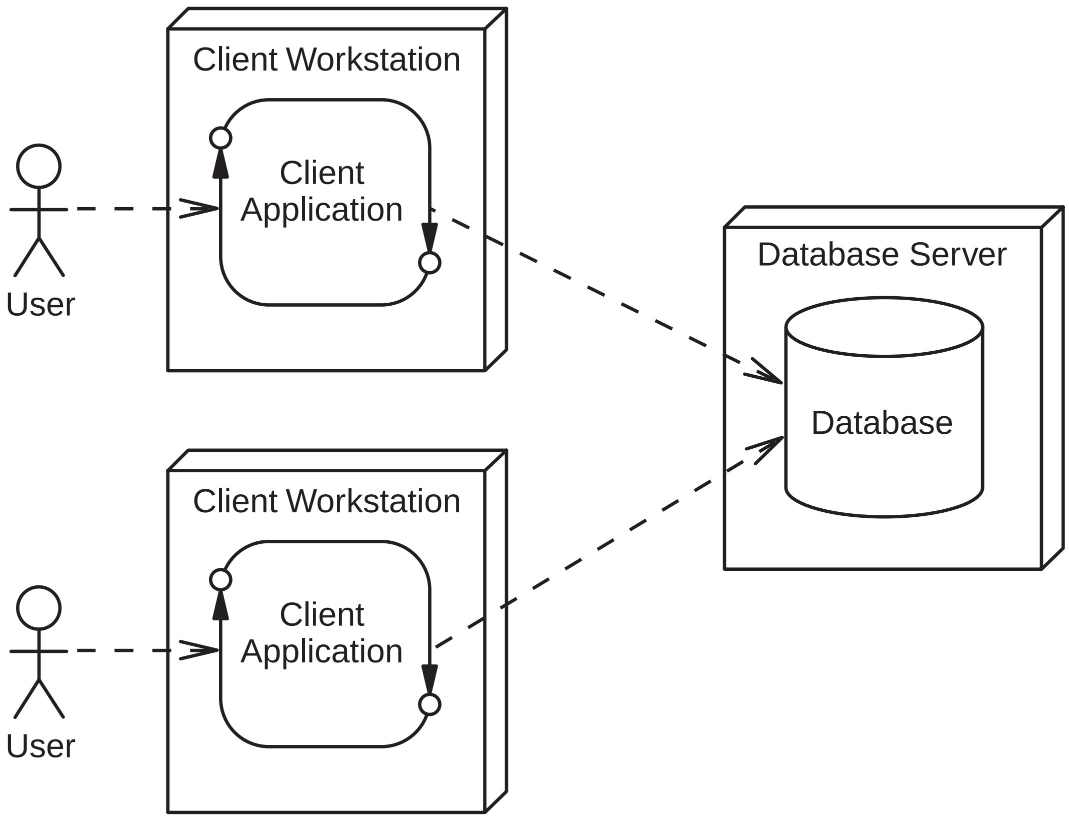
- Standalone applications
- Client/Server (2-tier) applications
- N-tier applications
- Web-centric n-tier applications
- Enterprise n-tier applications

Standalone Applications



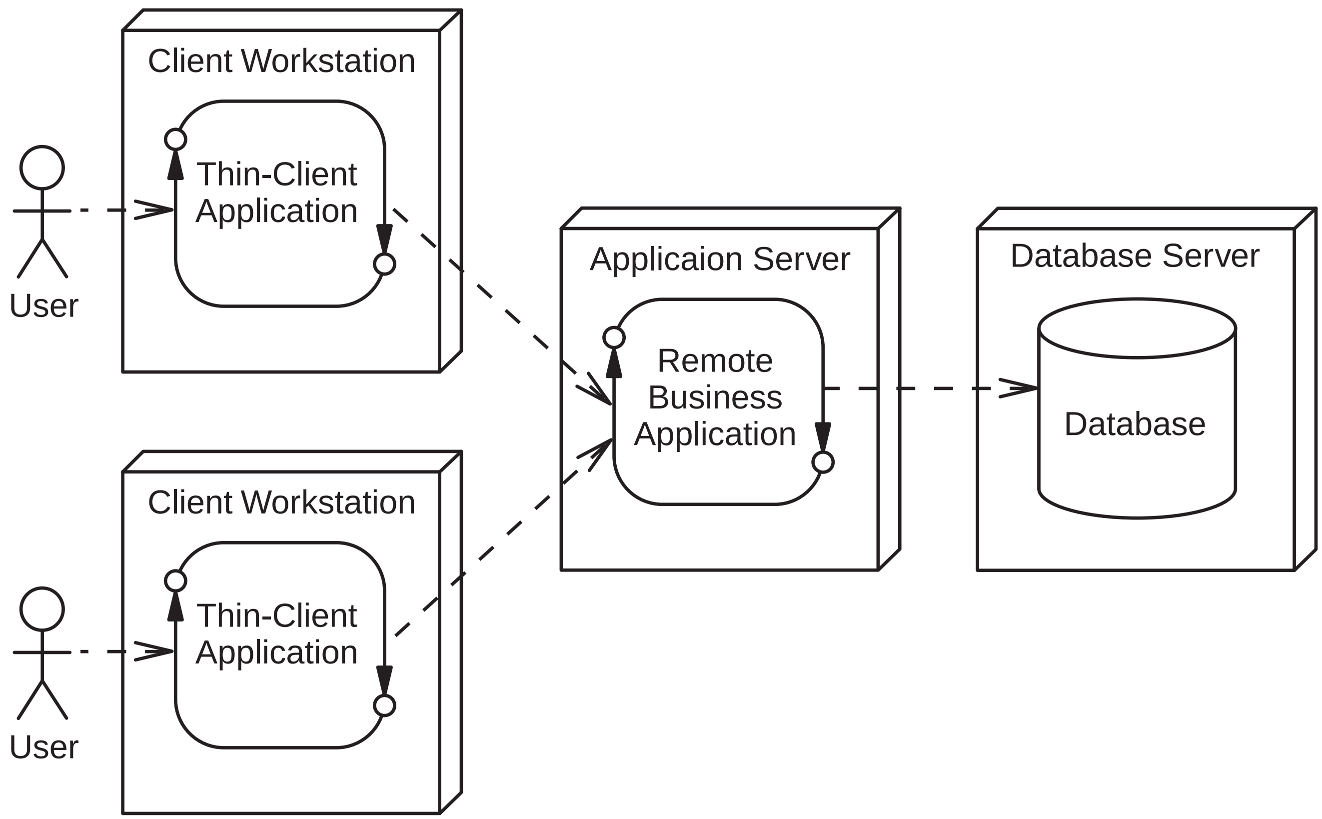
- No external data sources (all application data exists on a file server)
- No network communication (all application components exist on one machine)

Client/Server (2-Tier) Applications



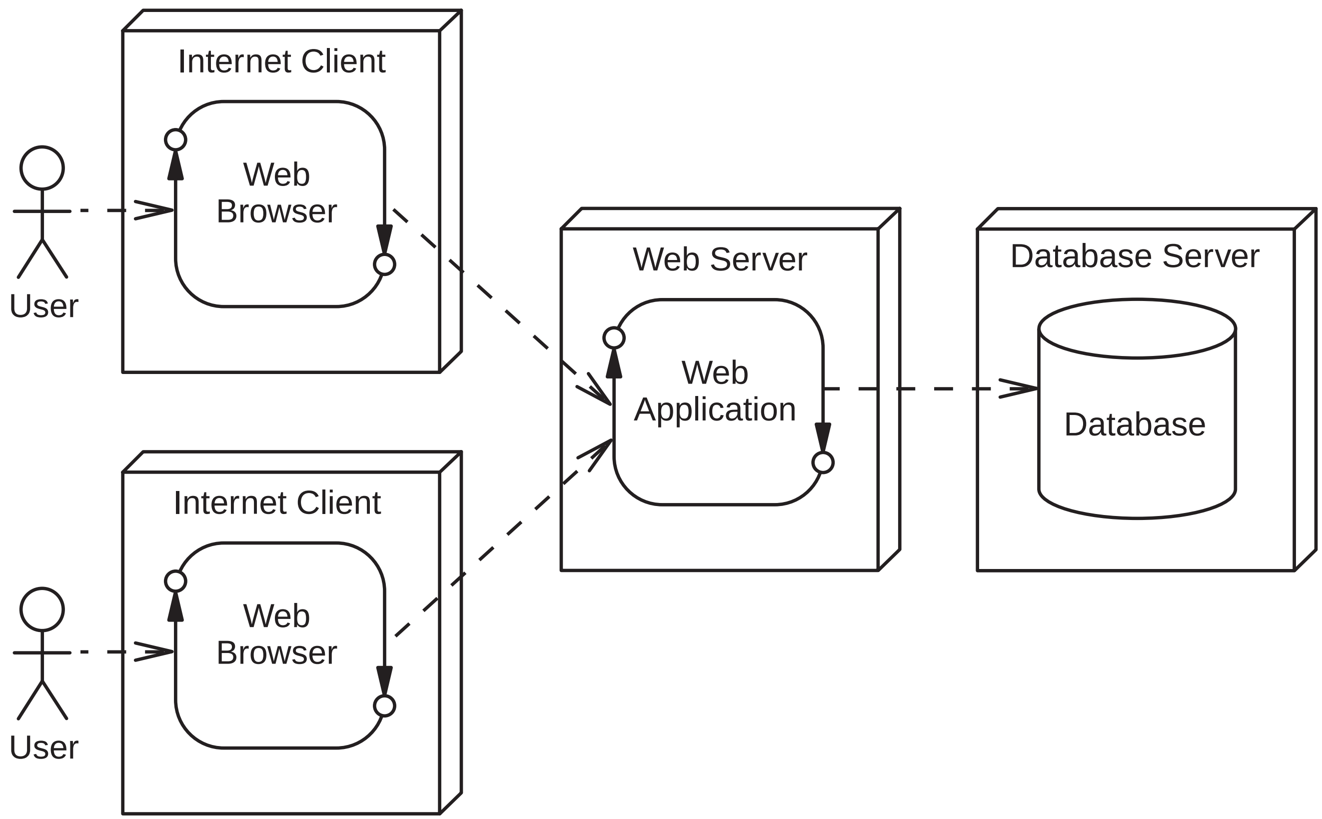
- Thick client (with business logic in the client tier)
- Data store manages data integrity

N-Tier Applications



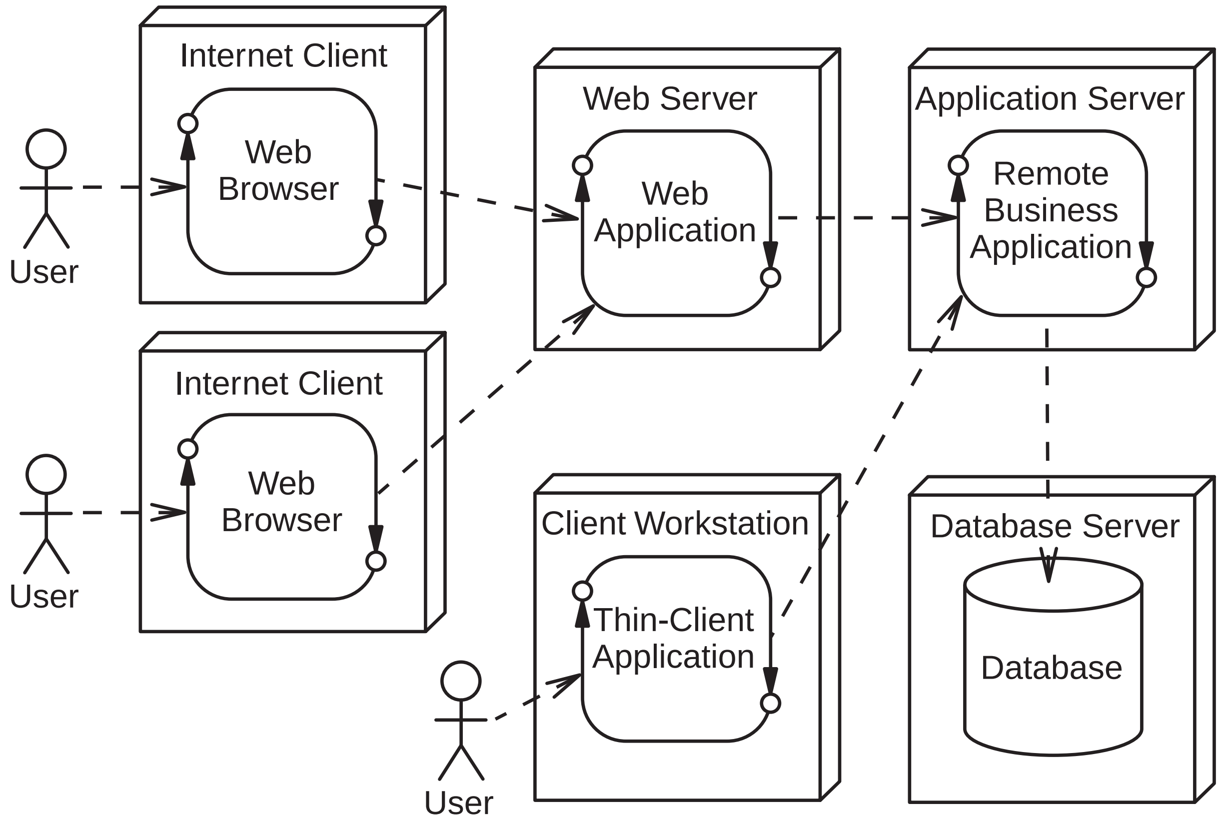
- Thin client (business logic is in the application server)
- Application server manages data integrity

Web-Centric (N-Tier) Applications



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

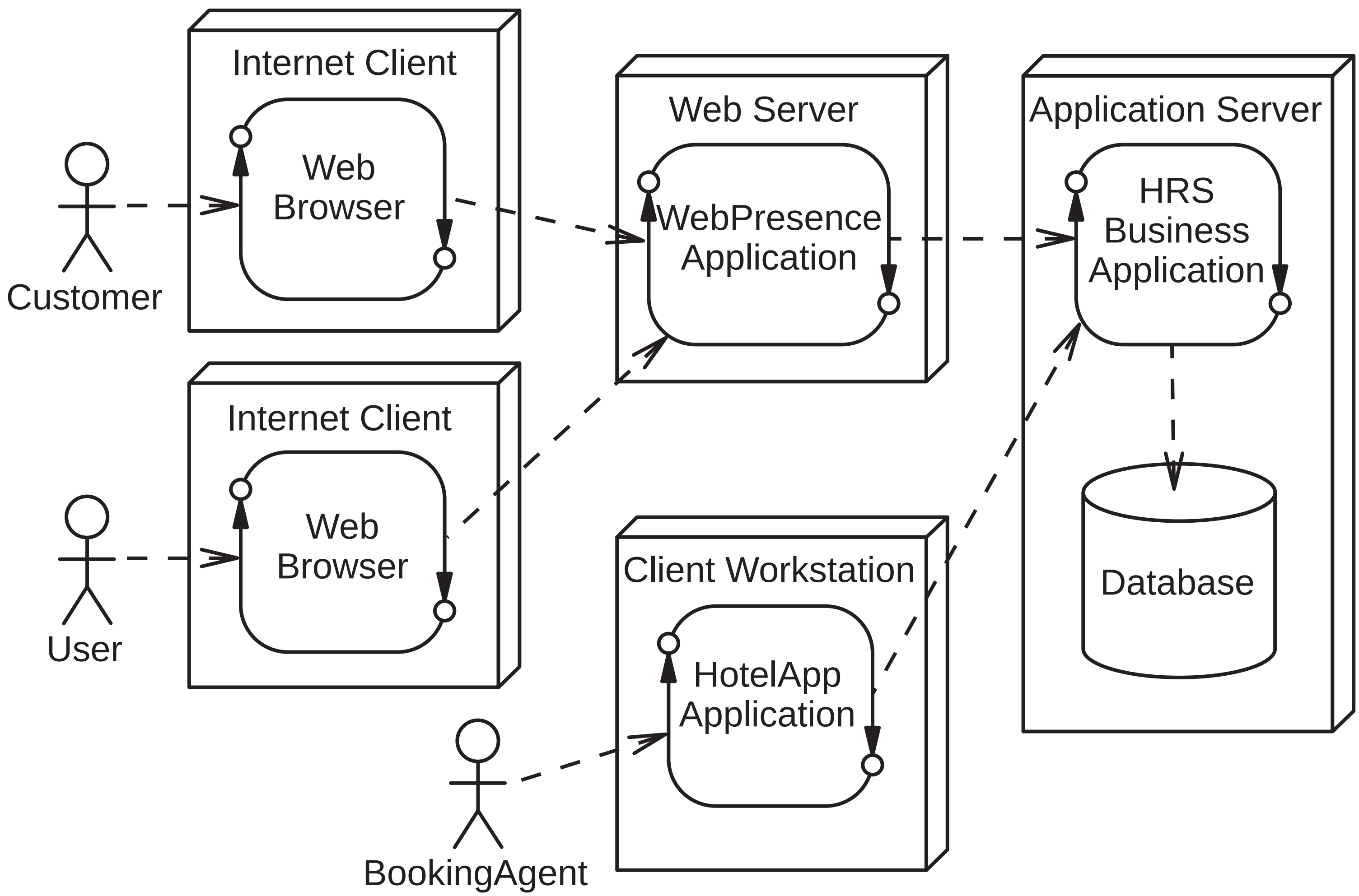
Enterprise (N-Tier) Architecture Type



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

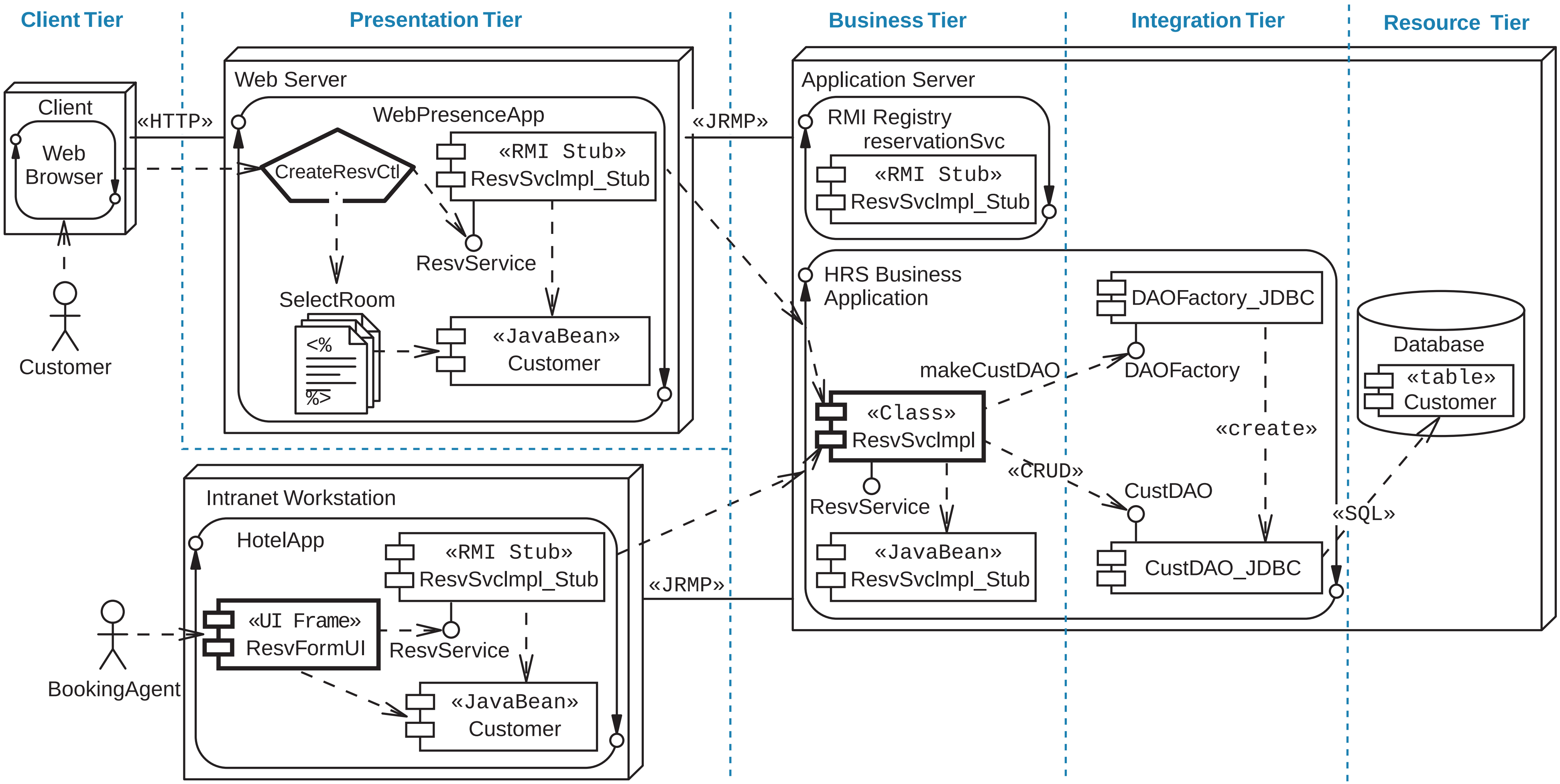
Hotel Reservation System Architecture



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.

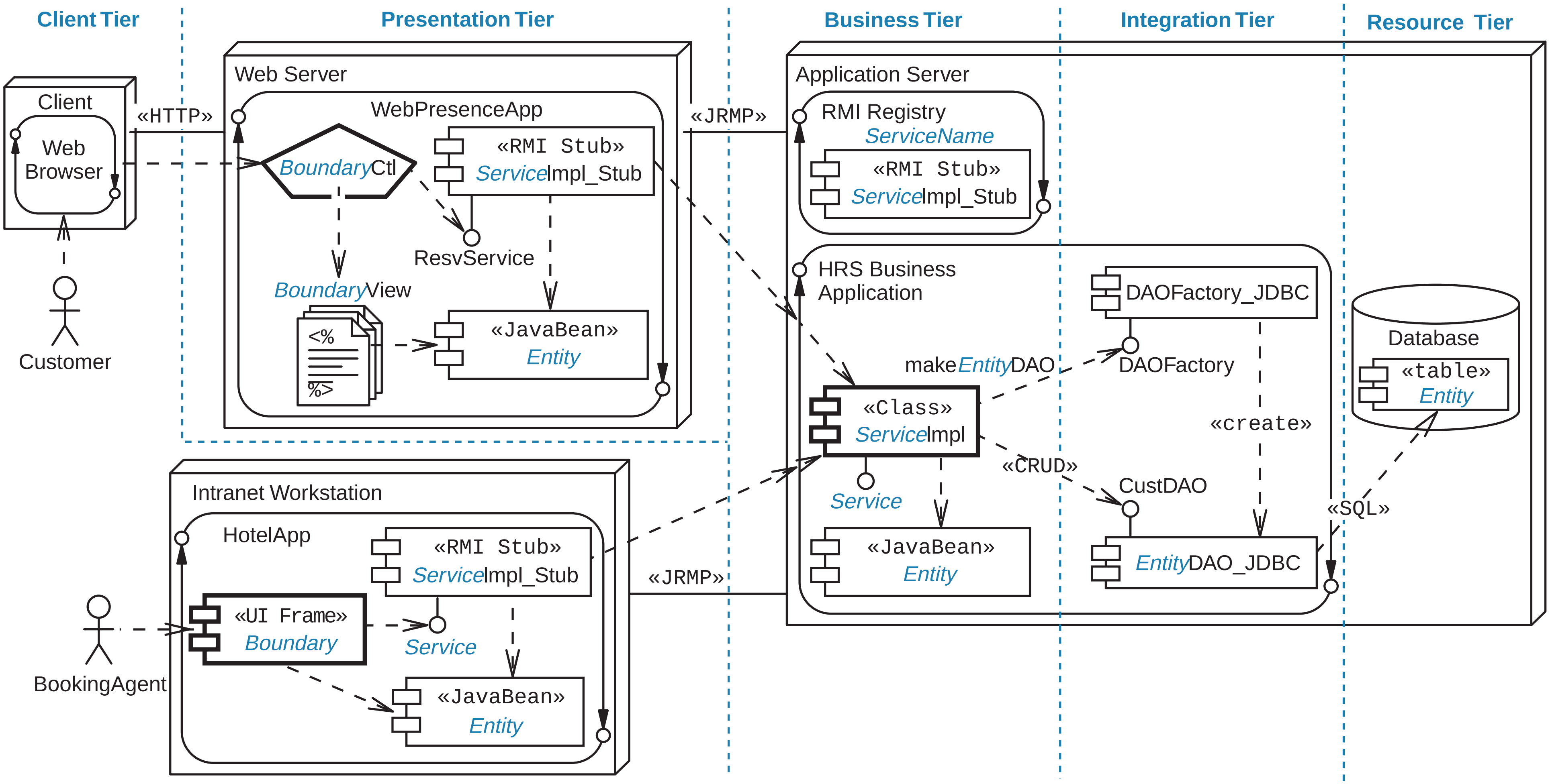
Example Detailed Deployment Diagram



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

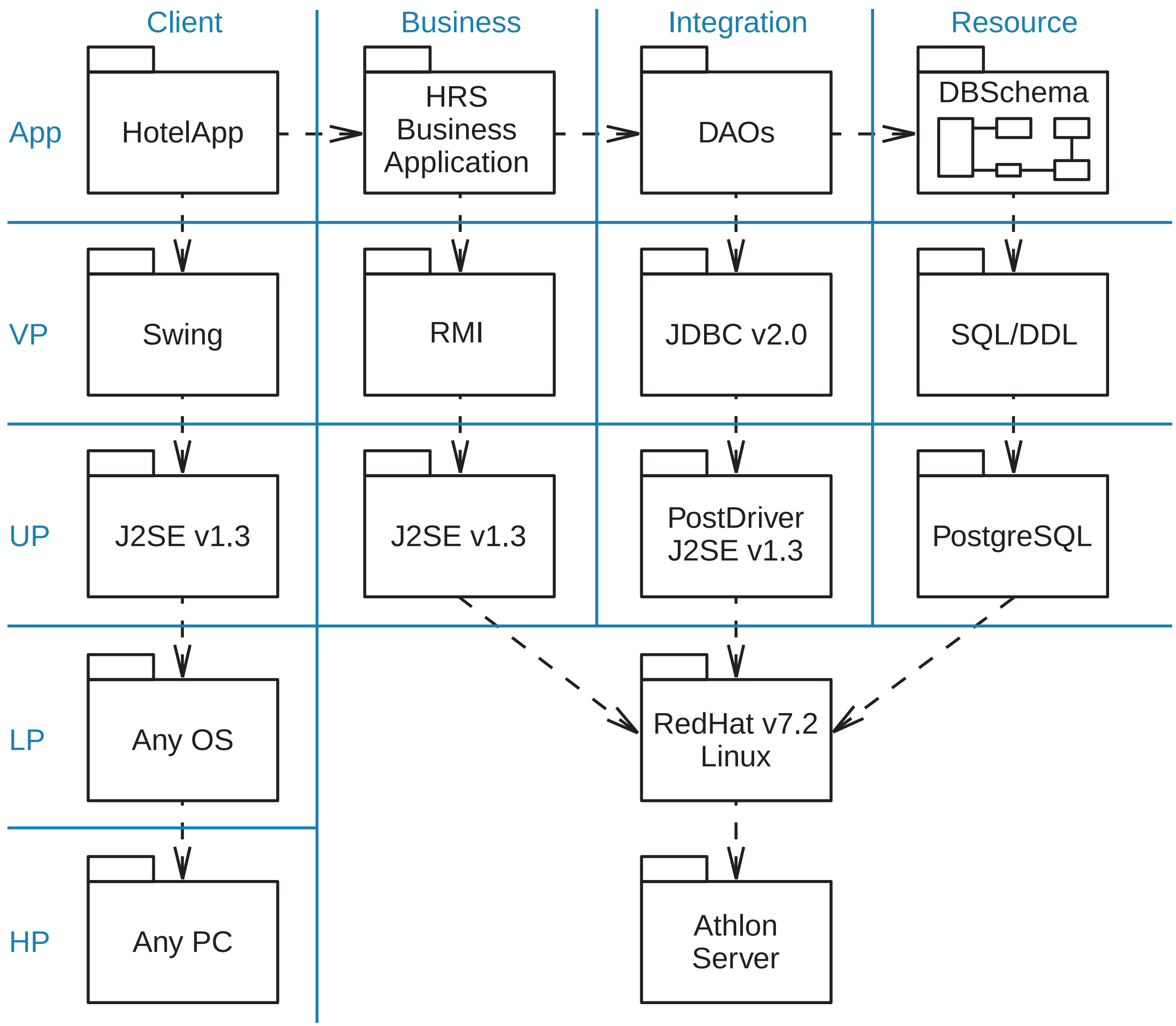


Creating the Tiers and Layers Package Diagram

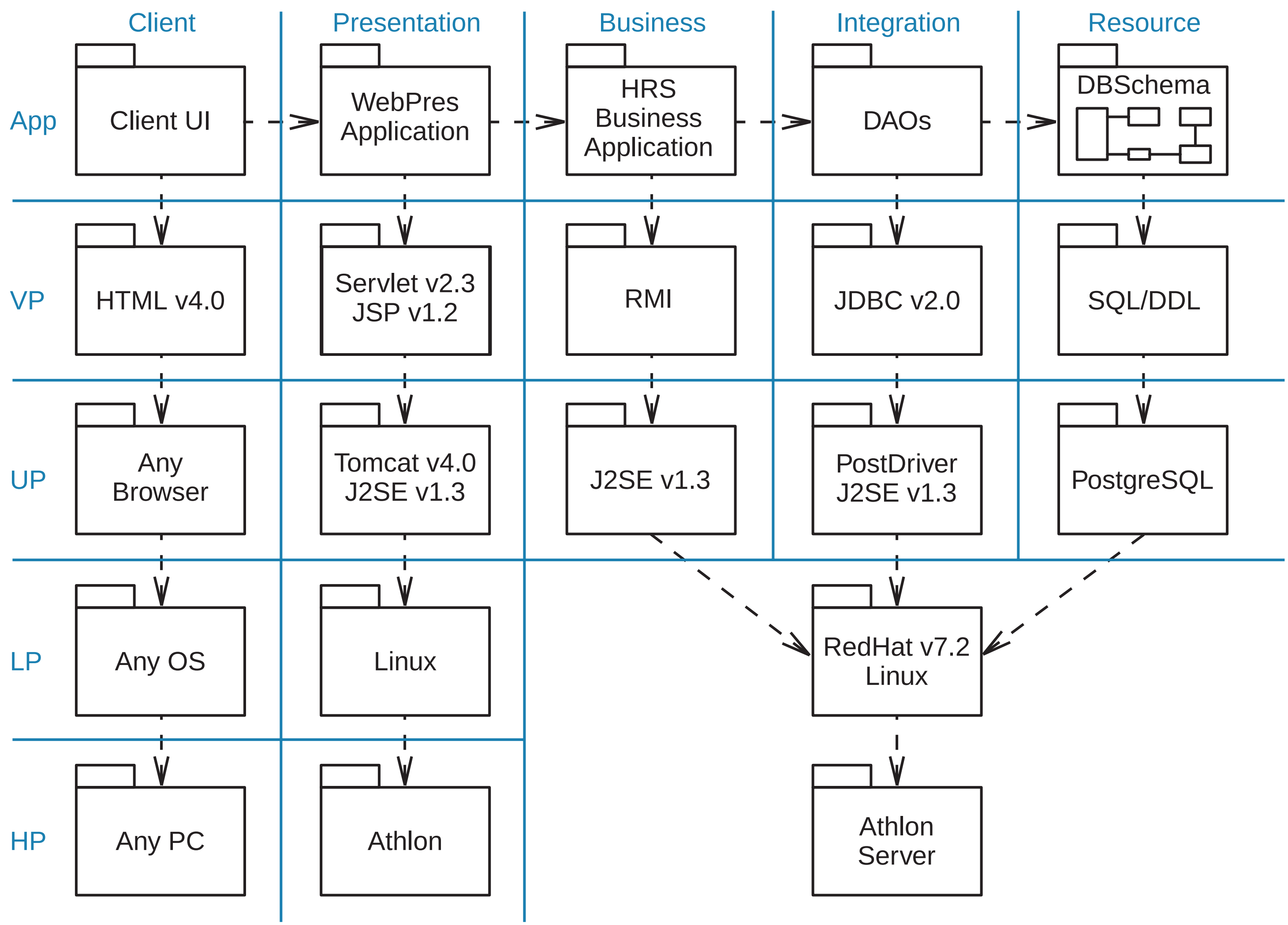
For each tier:

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 WebPresenceApp



Summary

The Architecture workflow includes the following tasks:

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

Summary

The Architecture workflow produces the following artifacts:

- The high-level Deployment diagram
- The detailed Deployment diagram
- The Architecture template
- The tiers and layers Package diagram

The following UML diagrams provide views of the Architecture model: Package, Component, and Deployment diagrams.

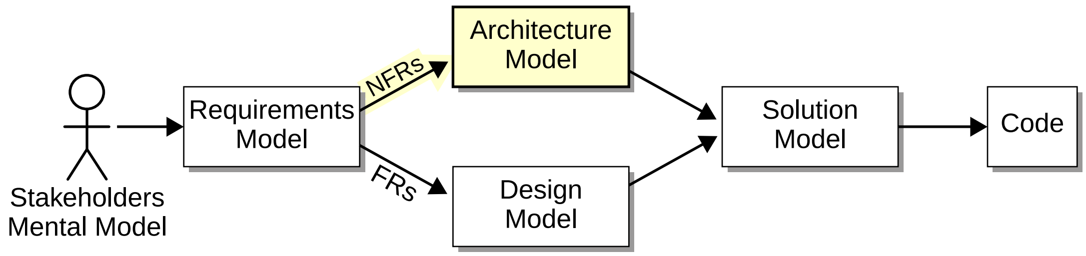
Module 13

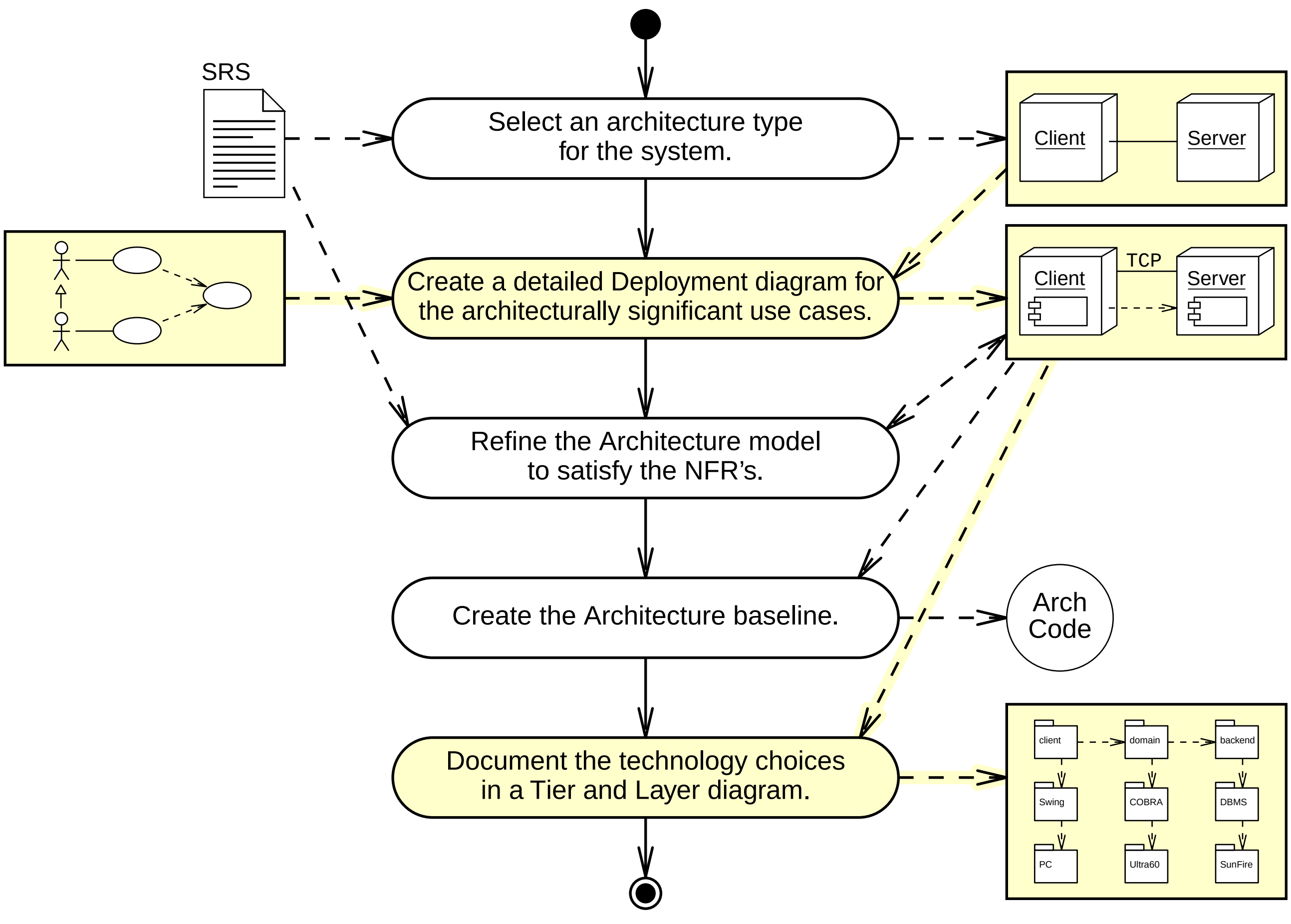
Creating an Architectural Model for the Client and Presentation Tiers

Objectives

- Exploring user interfaces
- Document a graphical user interface (GUI) application in the Client tier of the Architecture model
- Document a web user interface (WebUI) application in the Presentation tier of the Architecture model

Process Map


Stakeholders Mental Model
Requirements Model
NFRs
FRs
Architecture Model
Design Model
Solution Model
Code


SRS
Select an architecture type for the system.
Create a detailed Deployment diagram for the architecturally significant use cases.
Refine the Architecture model to satisfy the NFR's.
Create the Architecture baseline.
Document the technology choices in a Tier and Layer diagram.
Client
Server
TCP
Arch Code
client
domain
backend
Swing
COBRA
DBMS
PC
Ultra60
SunFire

Exploring User Interfaces

Why is the architect interested in UIs?

- UI technologies are many and varied.
- Usability is critical to the success of the system.

UIs provide:

- User input and actions that manipulate the system
- Visual presentations that represent the state of the system

User Interface Prototypes

Creating a UI prototype provides:

- Immediate visualization of the system to the stakeholders
- Workflow analysis
- Usability analysis

User Interface Technologies

The primary UI types are:

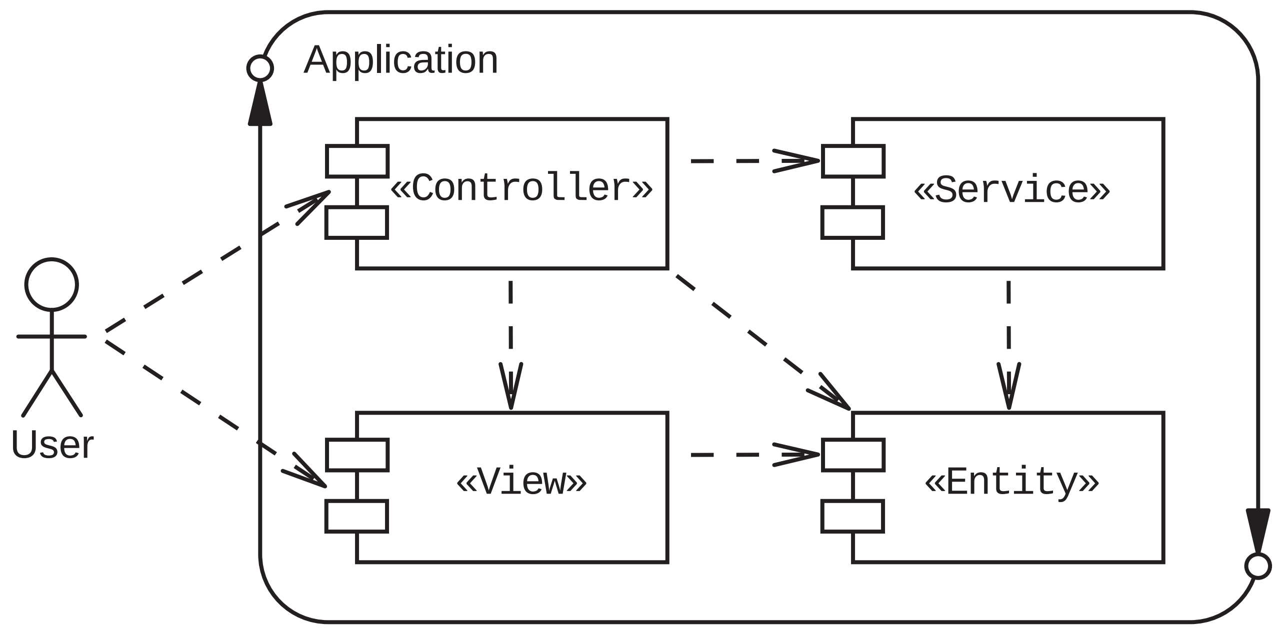
- Graphical user interface (GUI)
- Web user interface (Web UI)

Other types of UIs exist as well:

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

Generic Application Components

There are four fundamental types of application components:



Exploring Graphical User Interfaces

A GUI provides a window-based UI. GUIs have the following characteristics:

- The system must handle many small user actions.
- One screen can handle multiple use cases.
- The system enables an unrestricted flow of user actions.
- Multiple screens are accessible at one time.

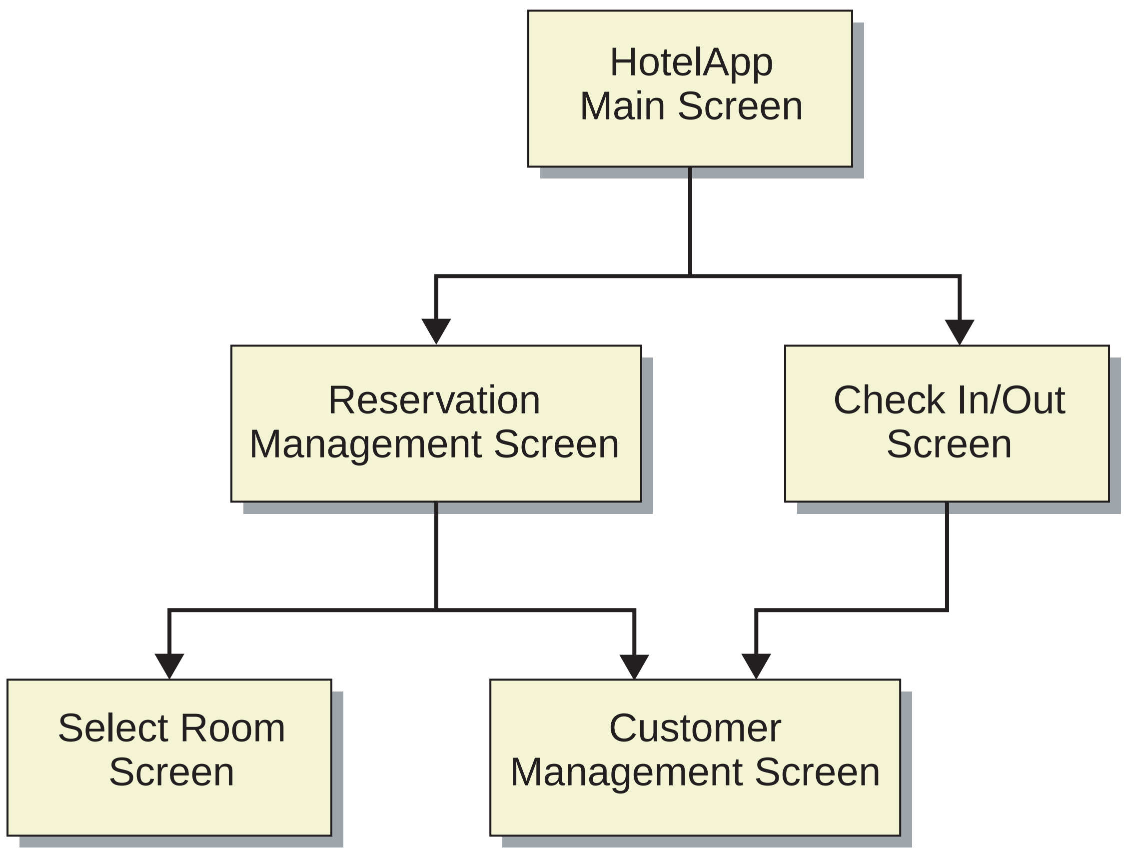
GUI Design

Designing any GUI should be done by a professional UI designer. However, these are the issues that you (as the software designer) should know:

- A GUI tends to be constructed as a hierarchy of related screens.
- Each screen is a hierarchy of GUI components.
- A GUI screen presents the user's view of the domain model and also presents the user's action controls.
- A single screen can support multiple use cases.

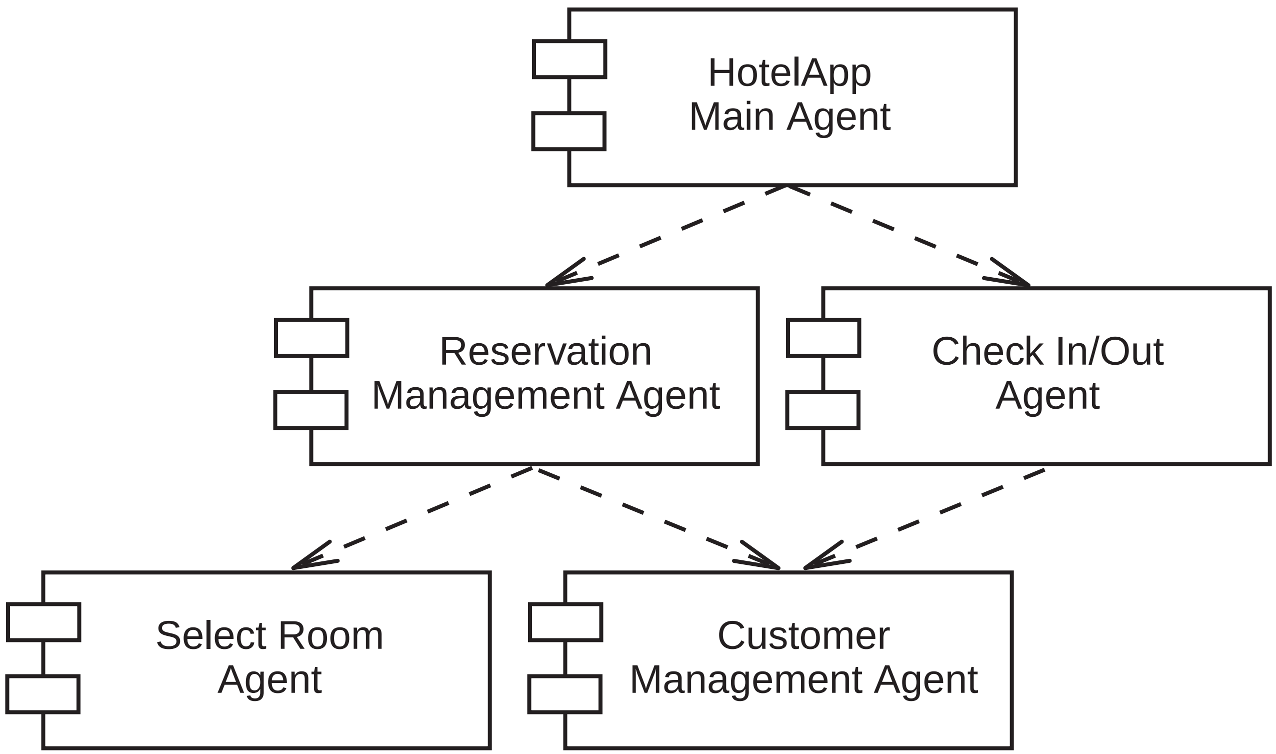
HotelApp Screen Hierarchy

The HotelApp program can be viewed as a hierarchy of UI screens:



The PAC Pattern

PAC defines a hierarchy of interacting agents:

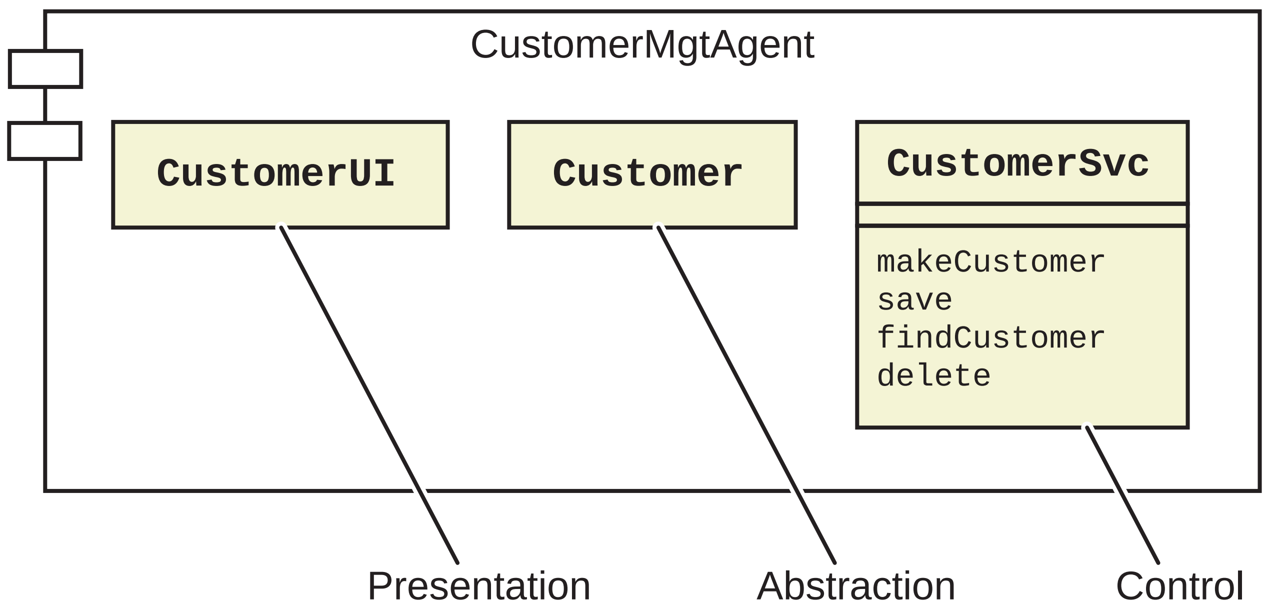


Elements of a PAC Agent

Each agent is a combination of three subcomponents:

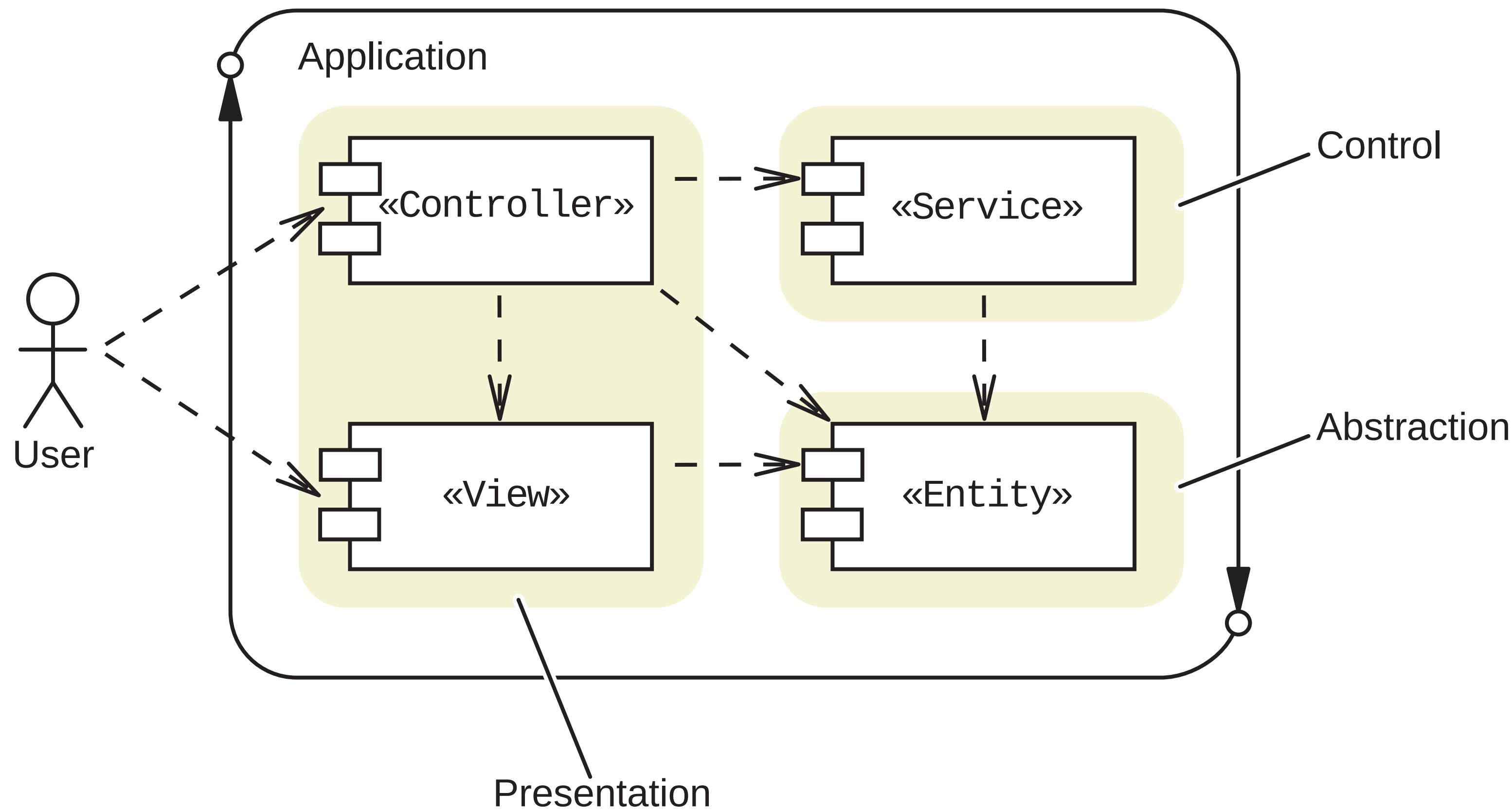
- Presentation – Presents a view of the Abstraction to the user and facilitates user actions to manipulate the Control component.
- Abstraction – Represents an entity in the system.
- Control – Represents a service of the system.

An Example PAC Agent



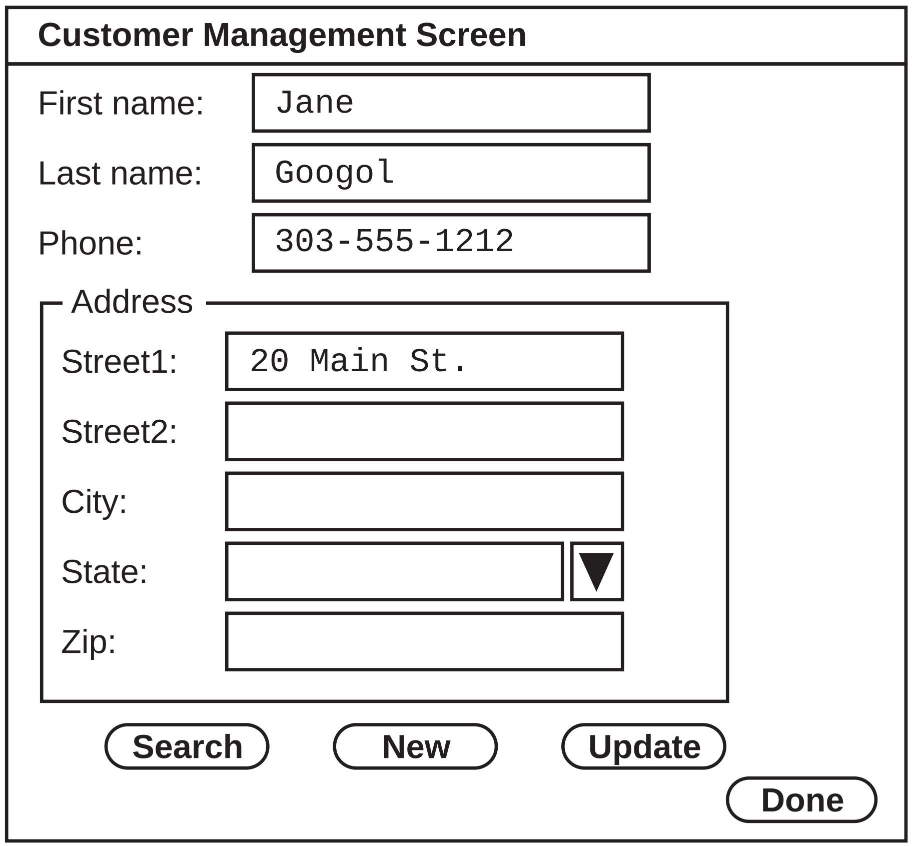
The PAC Component Types

PAC groups Controller and View components into a single component called Presentation:

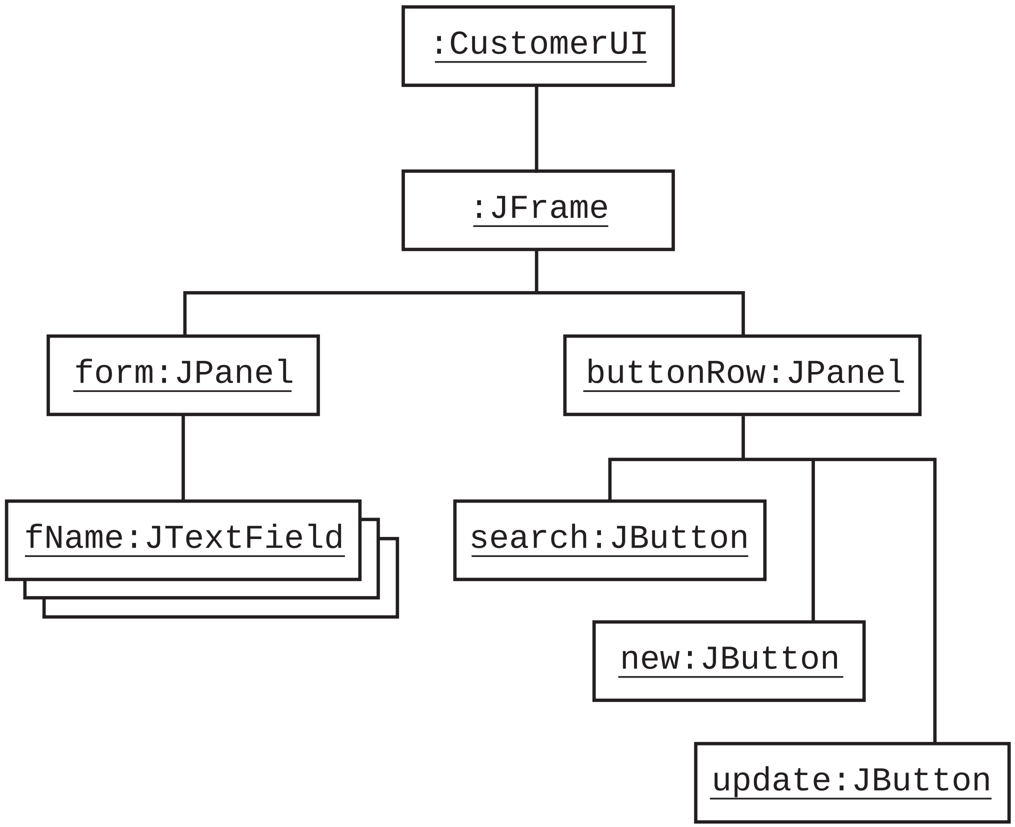


GUI Screen Design

An example GUI screen design:

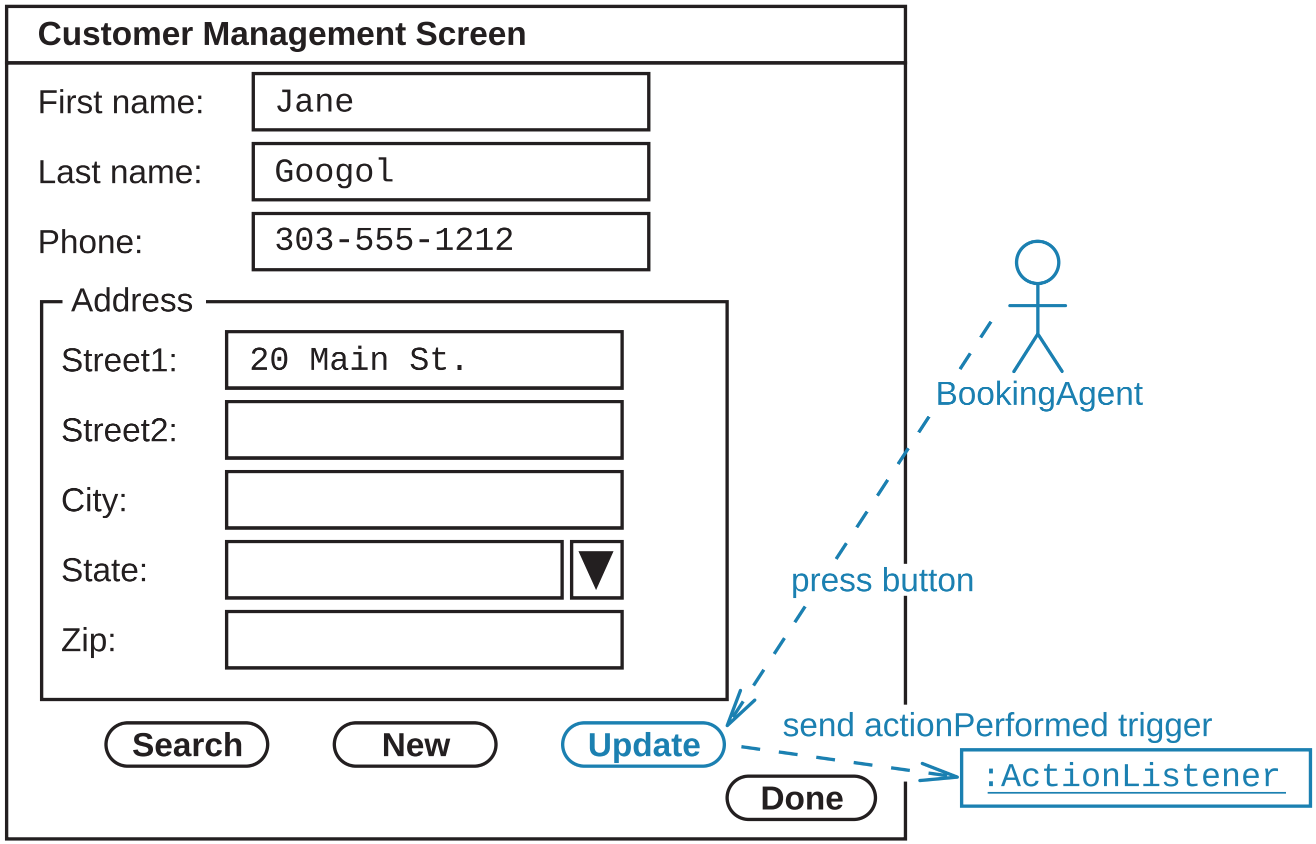


Customer GUI Component Hierarchy

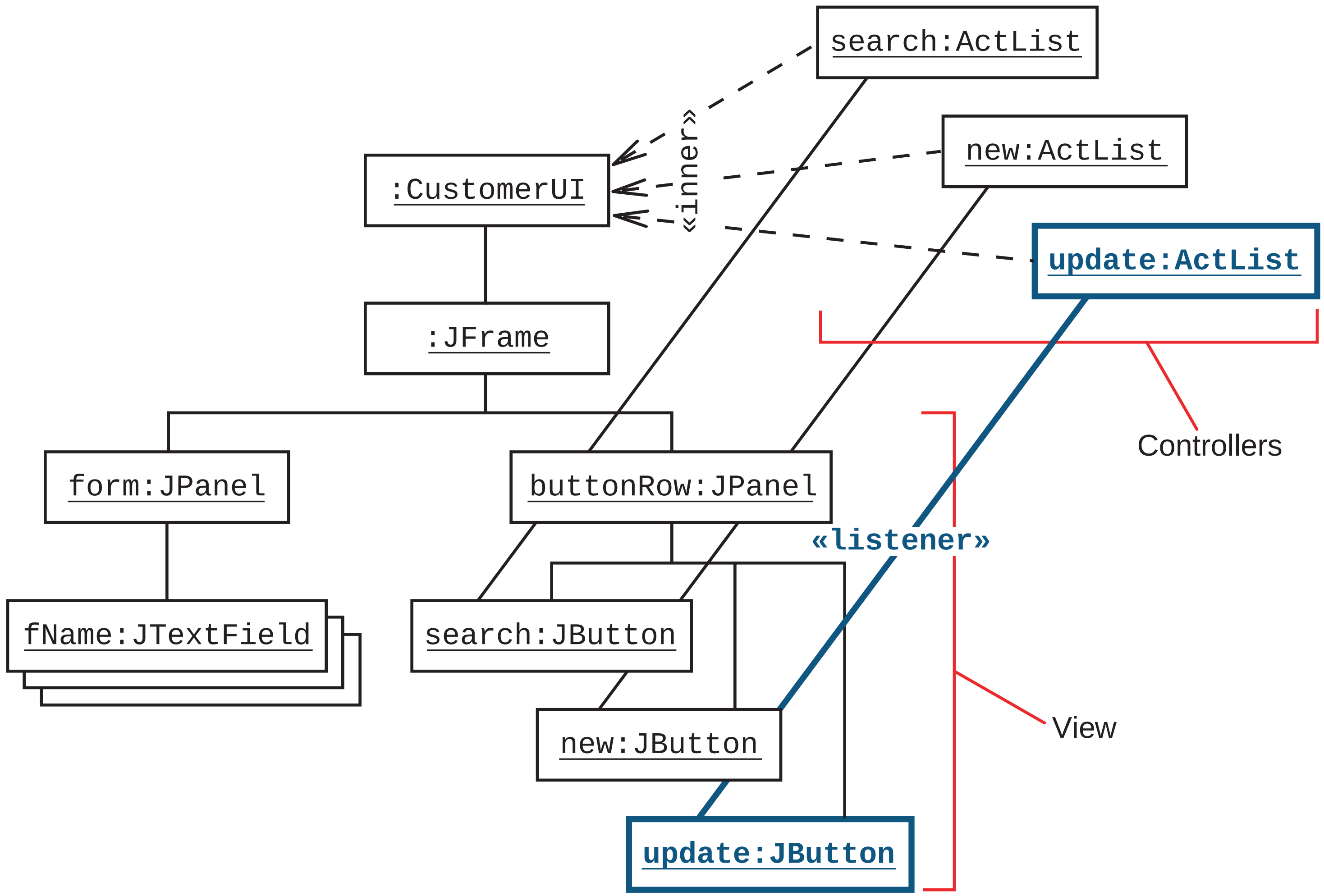


GUI Event Model

Java™ technology uses an event-listener mechanism:

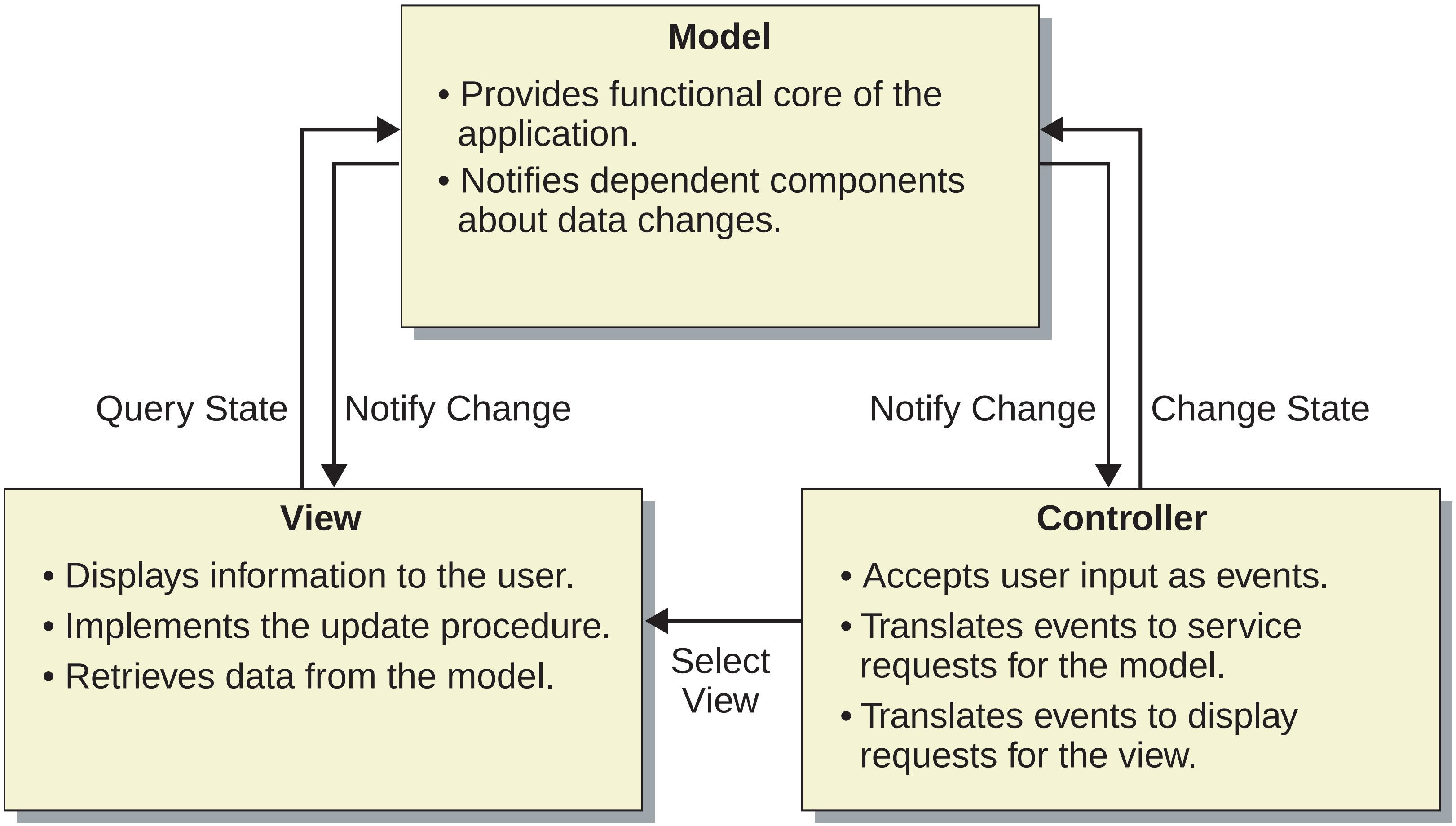


GUI Listeners as Controller Elements

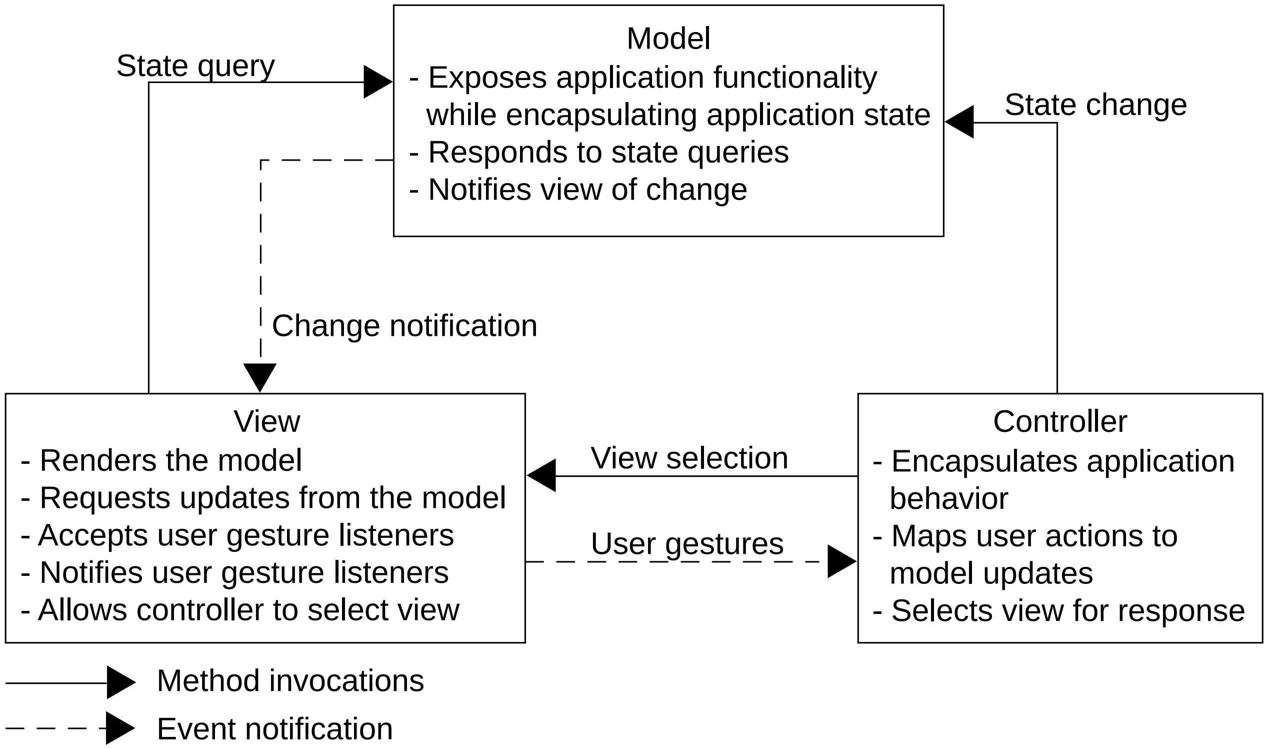


The MVC Pattern

MVC separates Views and Controllers from the Model.

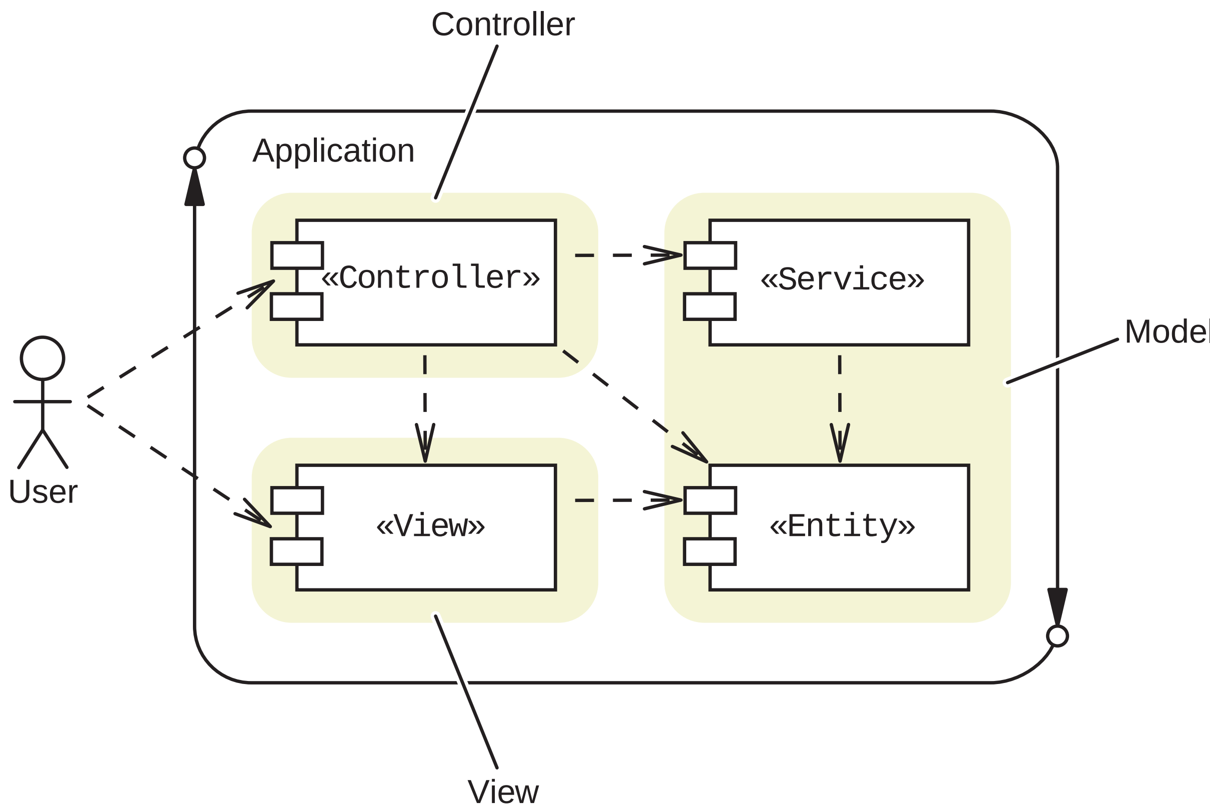


Relationships Among MVC Participants



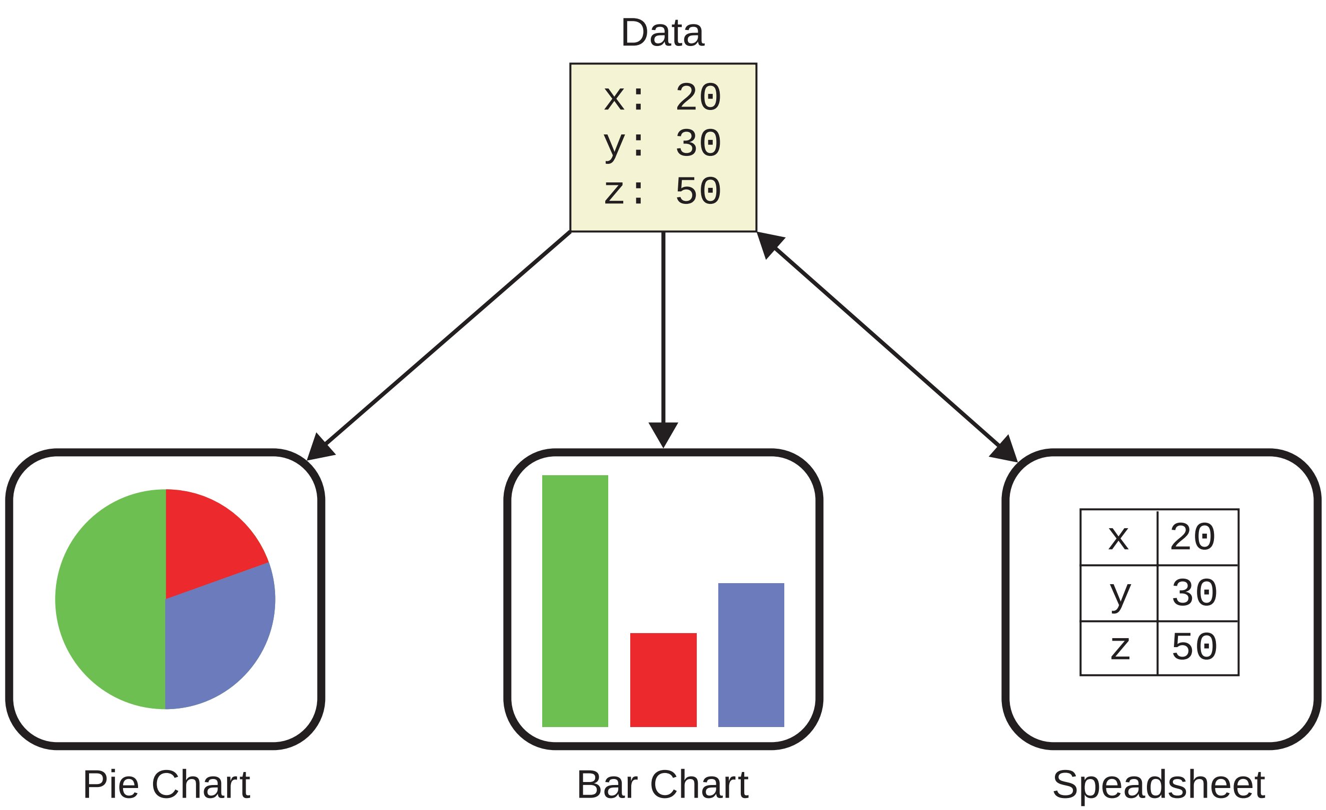
The MVC Component Types

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



Example Use of the MVC Pattern

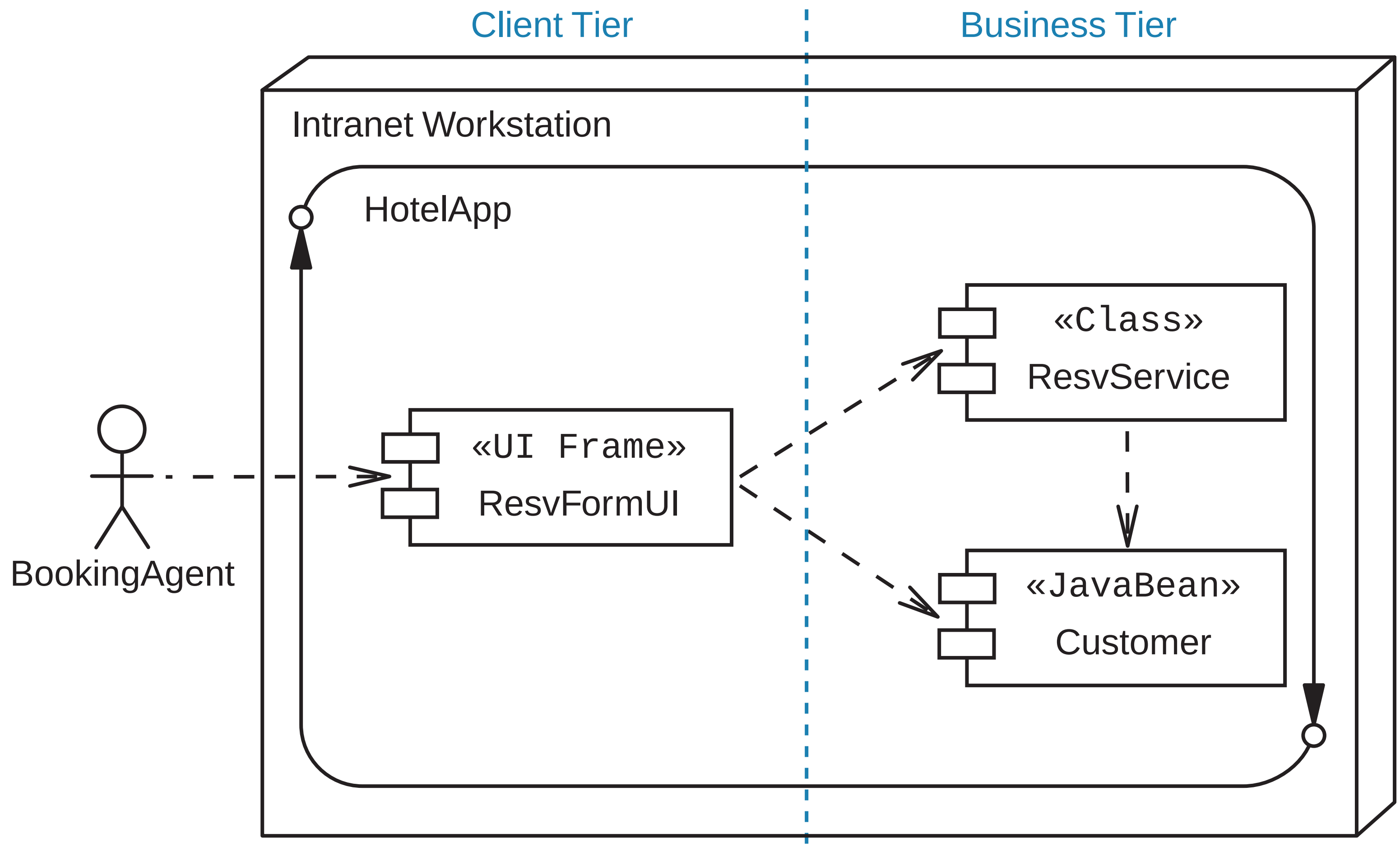
Multiple views can be used on the same data:



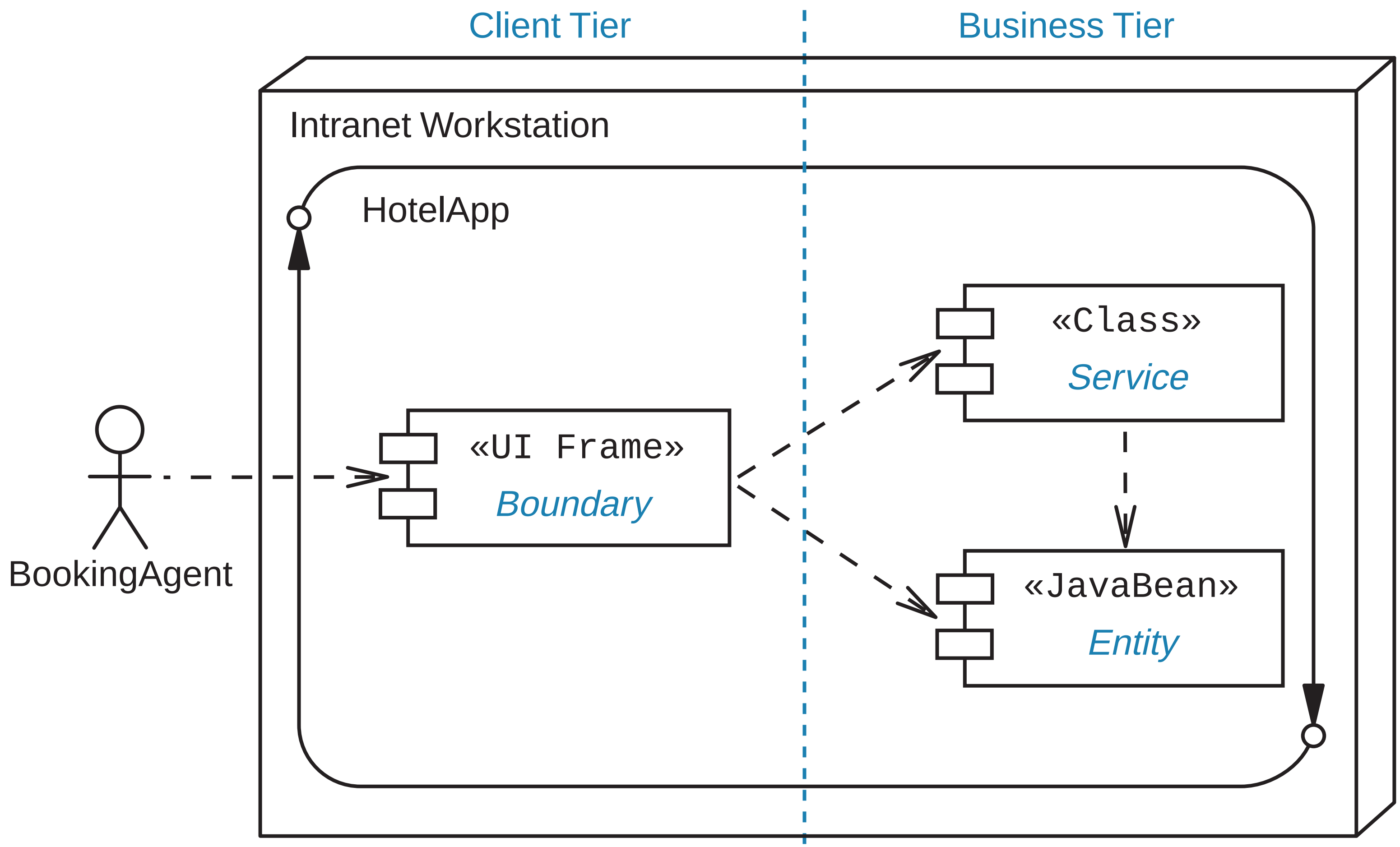
Recording the Client Tier in the Architecture Model

- Populate the detailed Deployment diagram with the Client tier components
- Create the Architecture template from the detailed Deployment diagram
- Populate the tiers and layers Package diagram with the Client tier technologies

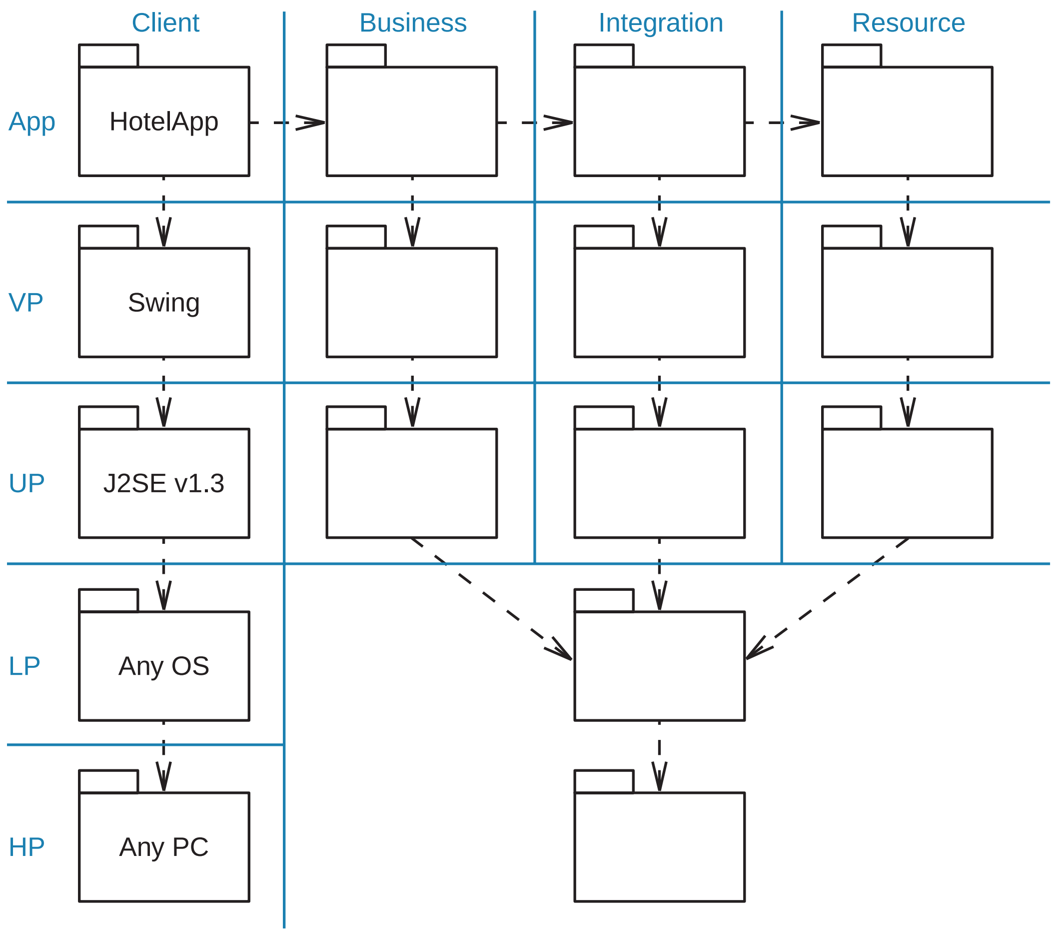
Populating the Detailed Deployment Diagram



Creating the Architecture Template



Populate the Tiers and Layers Pkg Diagram



Exploring Web User Interfaces

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

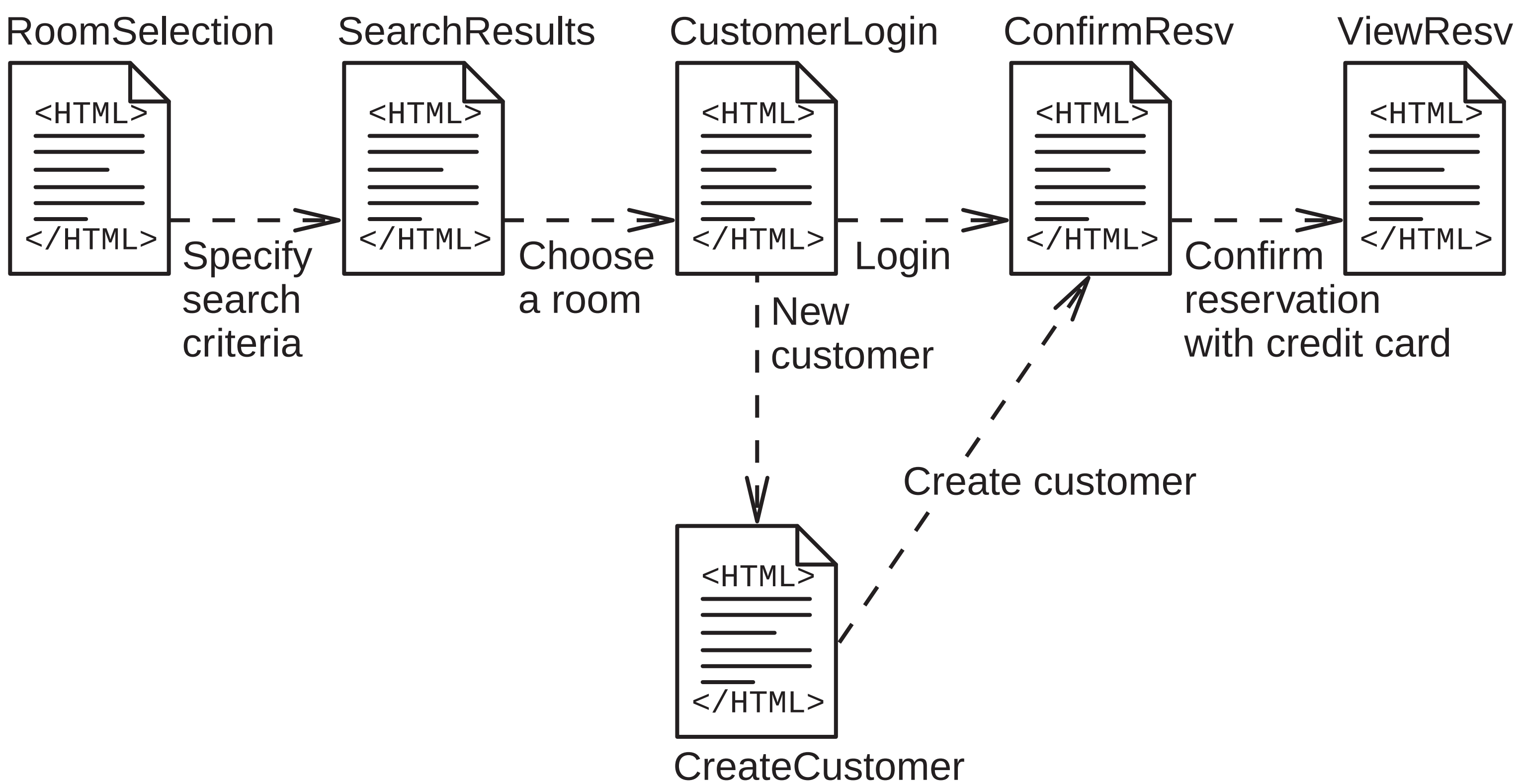
- Perform a few large user actions (HTTP requests).
- A single use case is usually broken into multiple screens.
- There is often a single path through the screens.
- Only one screen is usually open at a time.

Web UI 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:



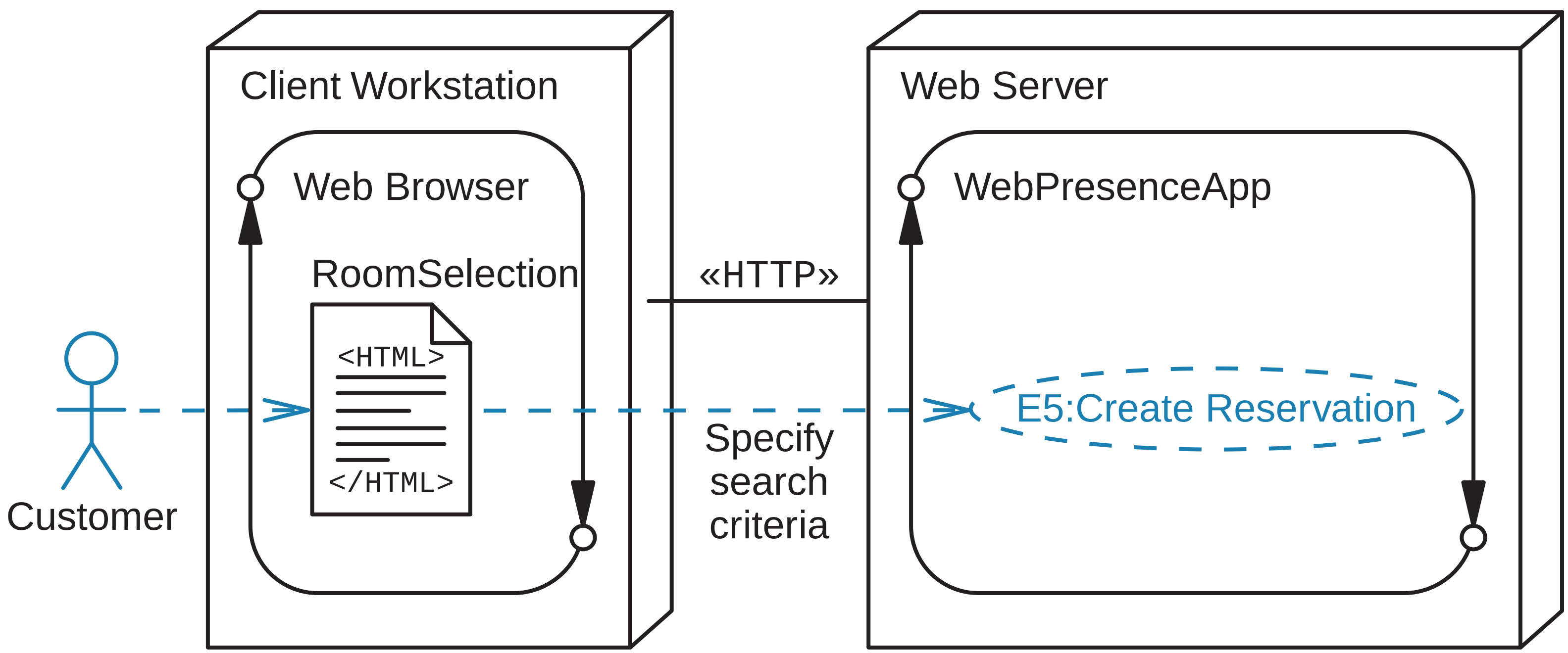
Partial Web UI Form Example

Web UI components are based on HTML forms. For example:

```
<FORM ACTION='makeResv' METHOD='POST'>
  <INPUT TYPE='hidden' NAME='action' VALUE='roomSearch'>
  Enter arrival date: <INPUT TYPE='text' NAME='arrivalDate'>
  <BR>
  Enter departure date: <INPUT TYPE='text' NAME='departureDate'>
  <BR>
  Select room type:
  <SELECT NAME='roomType'>
    <OPTION VALUE='Single'> Single
    <OPTION VALUE='Double'> Double
    <OPTION VALUE='Suite'> Suite
  </SELECT>
  <BR>
  <INPUT TYPE='submit' VALUE='Search...'>
</FORM>
```

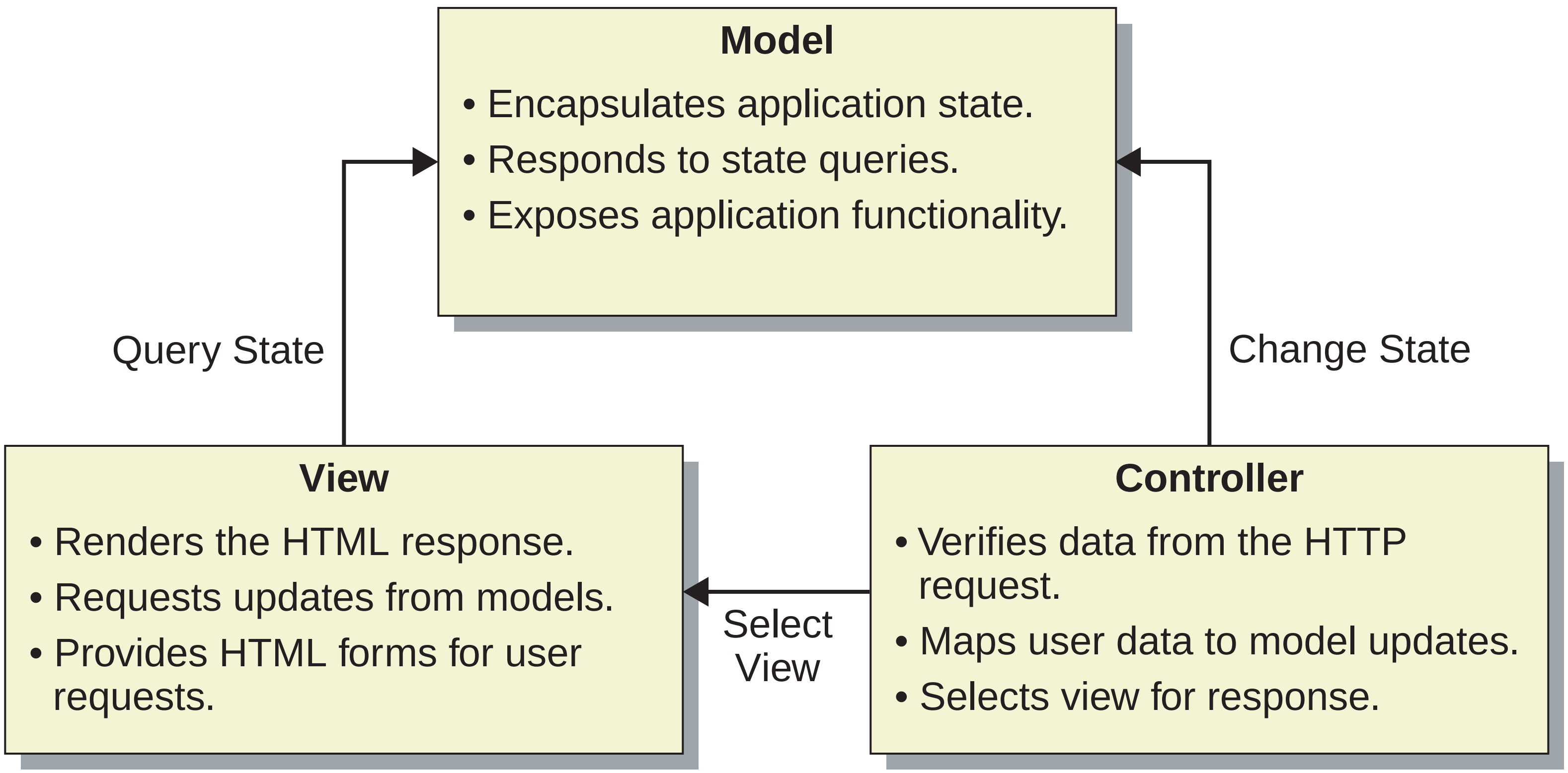
Web UI Event Model

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

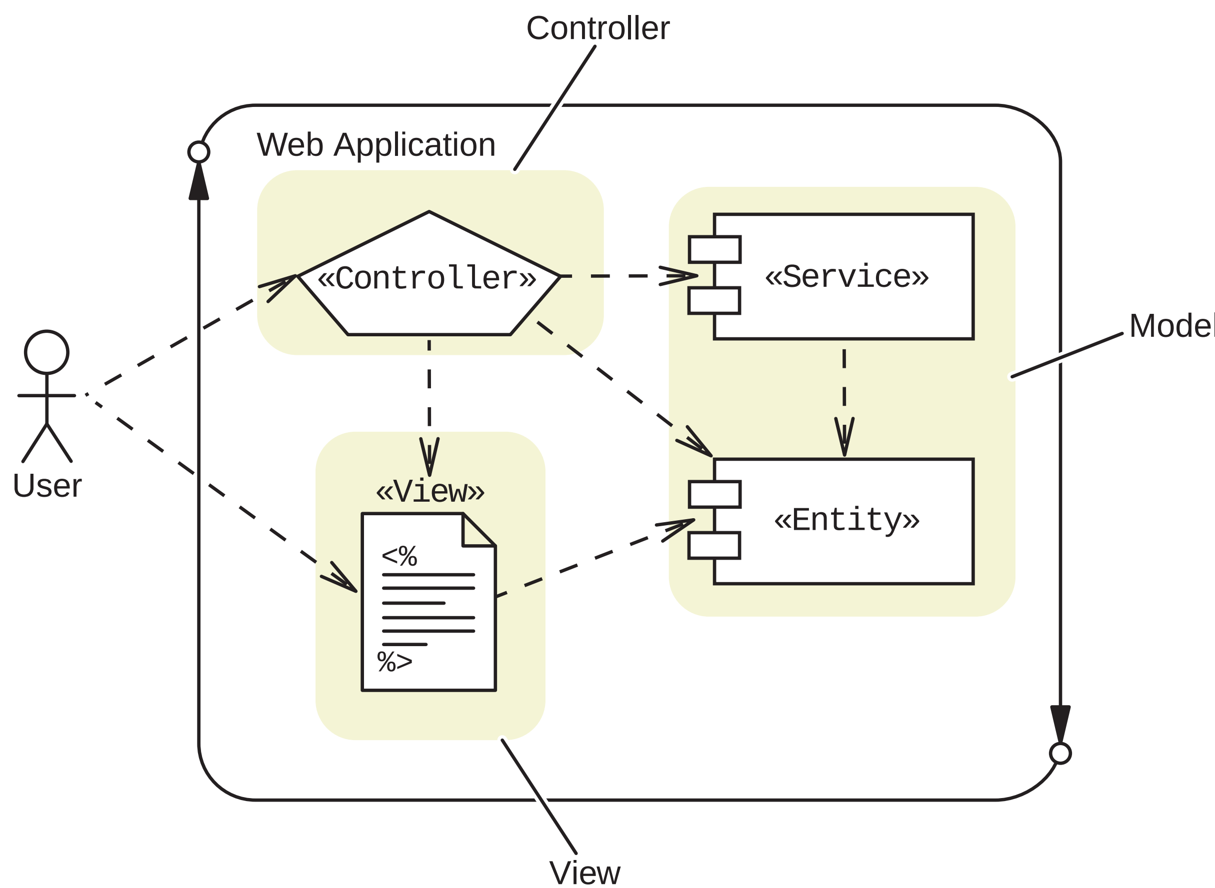


The WebMVC Pattern

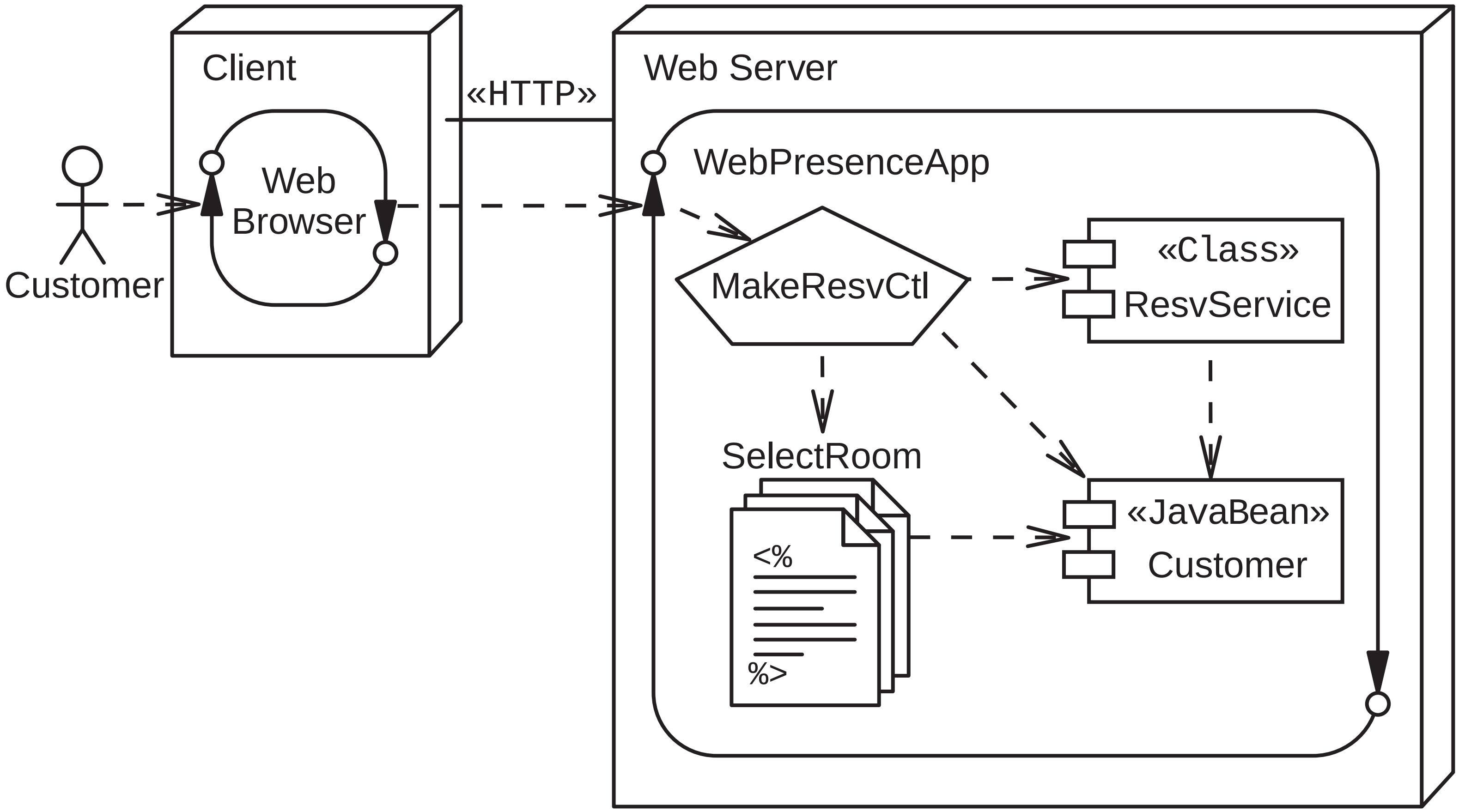
WebMVC is based on MVC, but with no Model to View updates:



The WebMVC Pattern Component Types



An Example Java Technology Web Application



The WebMVC Pattern

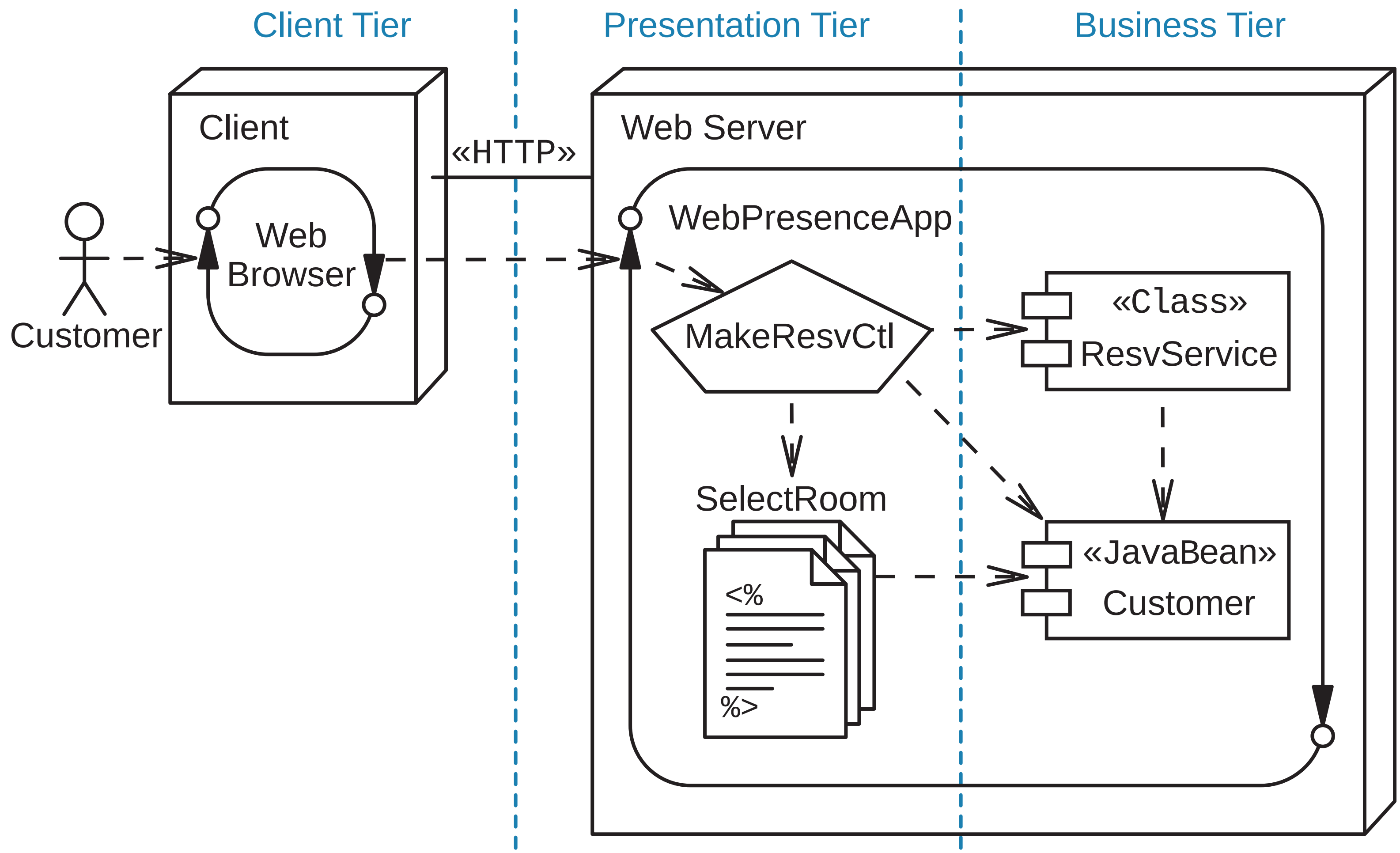
The Model 2 architecture use these components:

- 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.

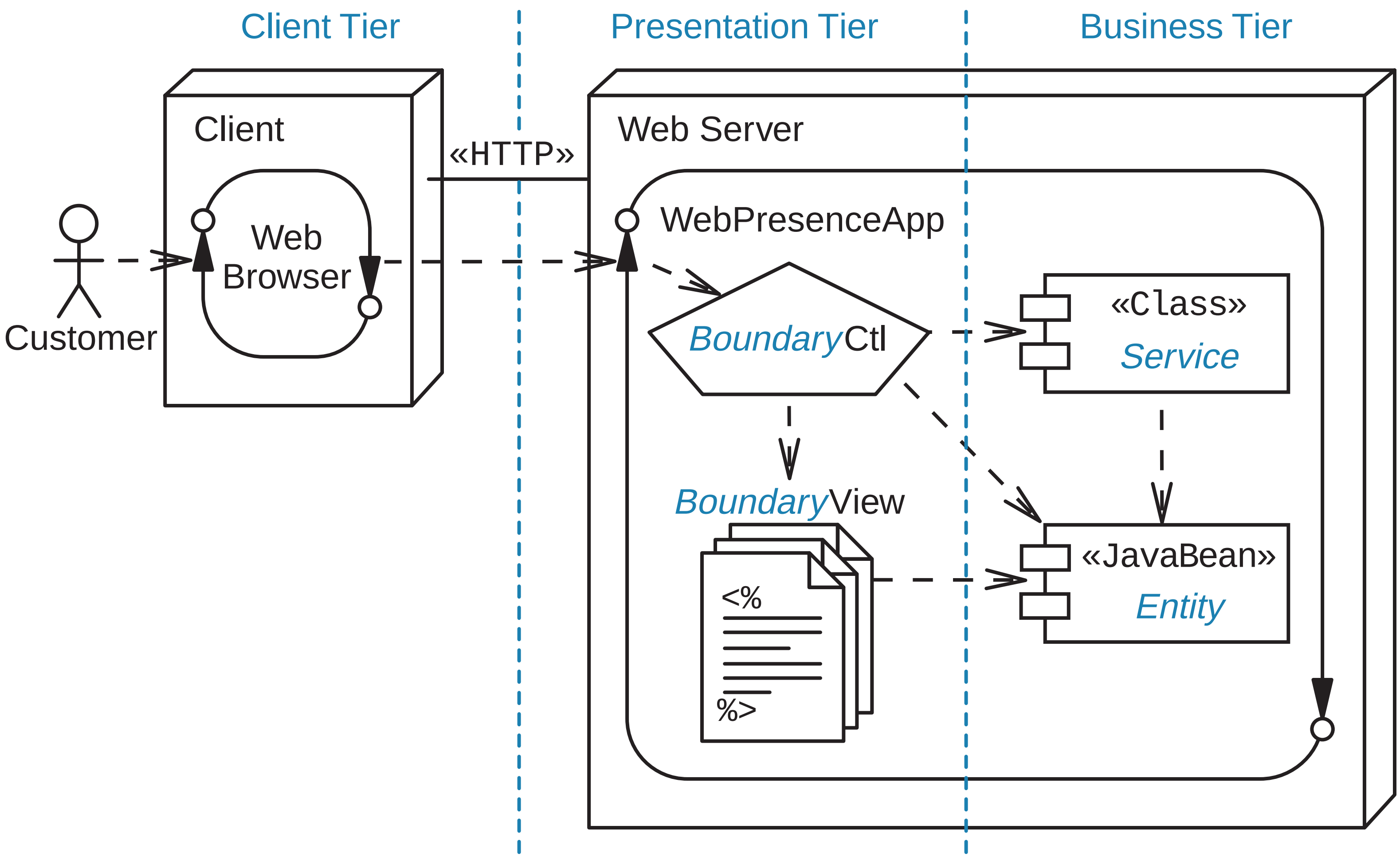
Recording the Presentation Tier in the Architecture Model

1. Populate the detailed Deployment diagram with the Presentation tier components
2. Create the Architecture template from the detailed Deployment diagram
3. Populate the tiers and layers Package diagram with the Presentation tier technologies

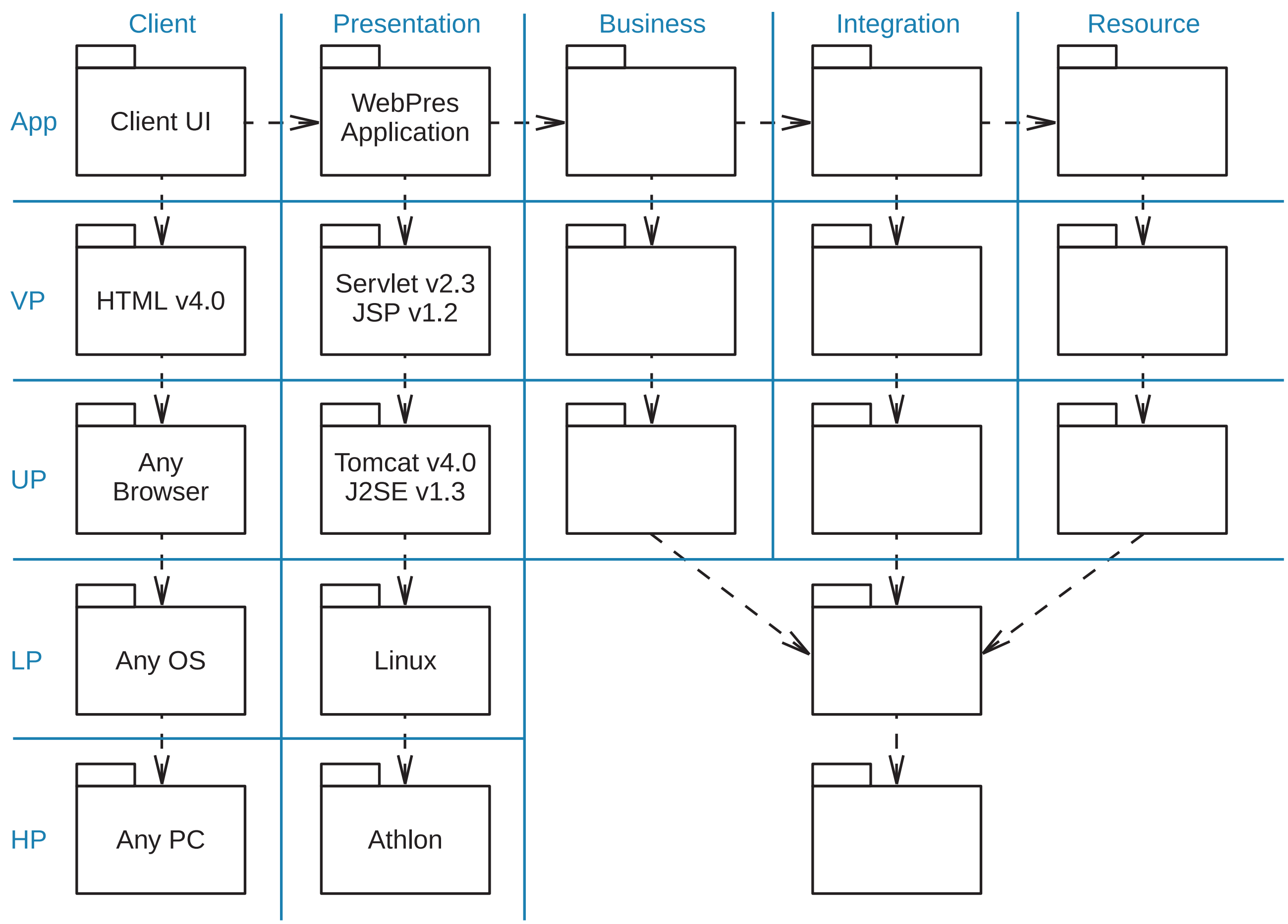
Populate the Detailed Deployment Diagram



Create the Architecture Template



Populate the Tiers and Layers Pkg Diagram



Summary

GUI Characteristics	Web UI Characteristics
Many small user actions must be handled by the system.	Mostly a few large user actions (HTTP requests); small user actions can be handled using JavaScript technology.
One screen can usually handle multiple use cases.	A single use case is usually broken into multiple screens.
The system usually allows an unrestricted flow of user actions.	There is often a single flow through the screens.
Multiple screens are accessible at one time.	The user deals with one screen at a time.

Summary

- Characteristics of GUI technologies are:
  - Use swing components for Views.
  - Use listener classes for Controllers.
  - These low-level components are combined into a single Presentation component in the PAC pattern.
  - Make up the Client tier for a GUI application
- Characteristics of Web UI Technologies are:
  - Use JavaServer Pages technology components for Views.
  - Use Servlet components for Controllers.
  - These components make up the Presentation tier.

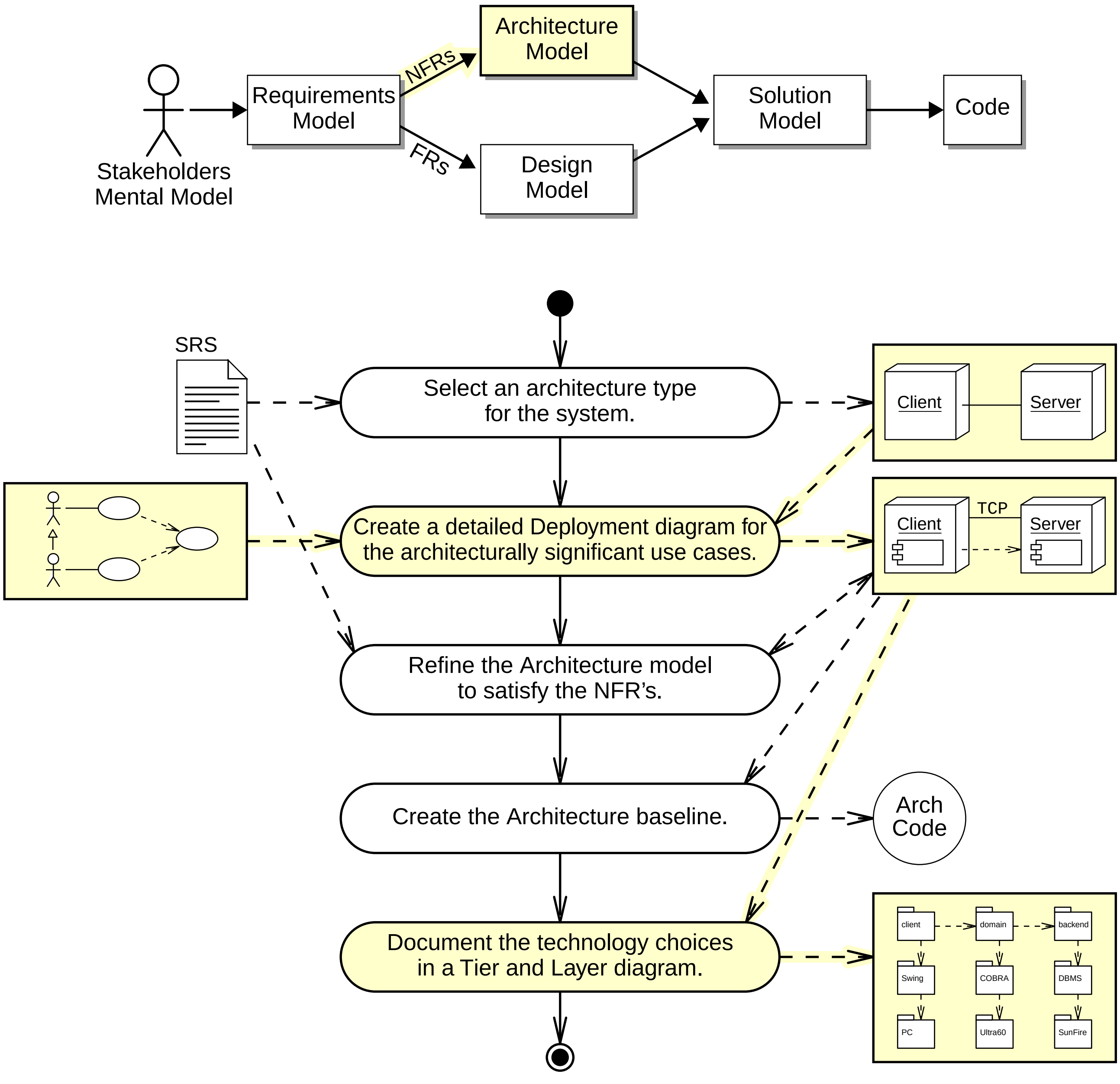
Module 14

Creating an Architectural Model for the Business Tier

Objectives

- Explore distributed object-oriented computing
- Document the Business tier in the Architecture model

Process Map


Stakeholders Mental Model
Requirements Model
NFRs
FRs
Architecture Model
Design Model
Solution Model
Code
SRS
Select an architecture type for the system.
Create a detailed Deployment diagram for the architecturally significant use cases.
Refine the Architecture model to satisfy the NFR's.
Create the Architecture baseline.
Document the technology choices in a Tier and Layer diagram.
Client
Server
TCP
Arch Code
client
domain
backend
Swing
COBRA
DBMS
PC
Ultra60
SunFire

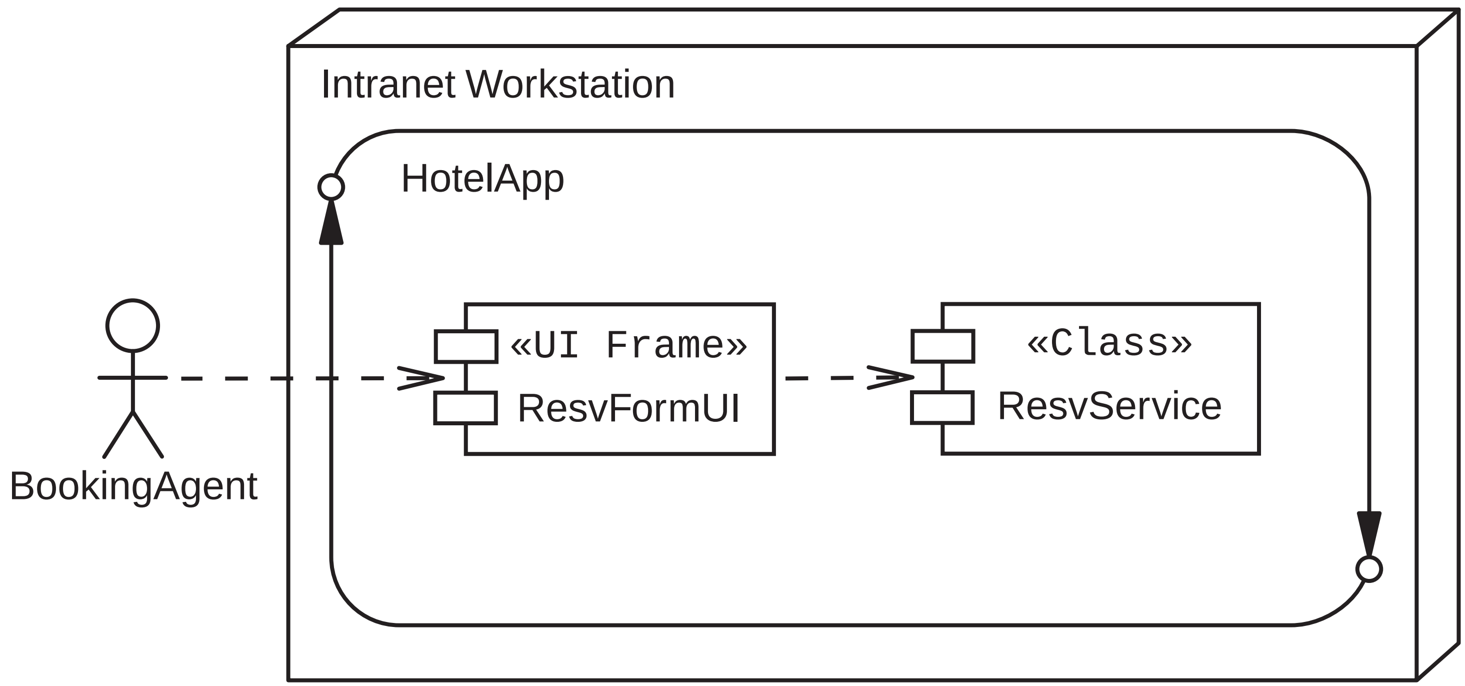
Exploring Distributed Object-Oriented Computing

Purpose: To invoke methods on a remote service (object).

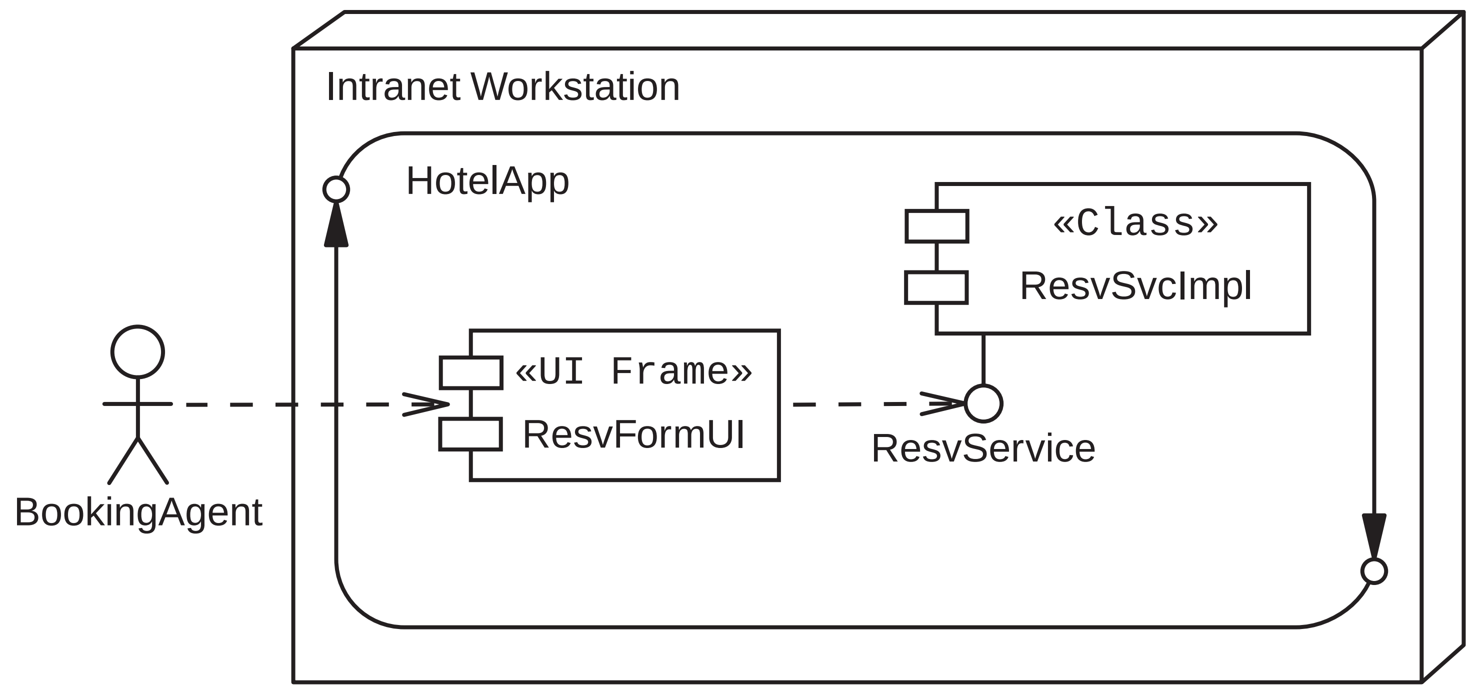
Types of technologies:

- CORBA
- RMI
- EJB™ technology
- Web services (SOAP)

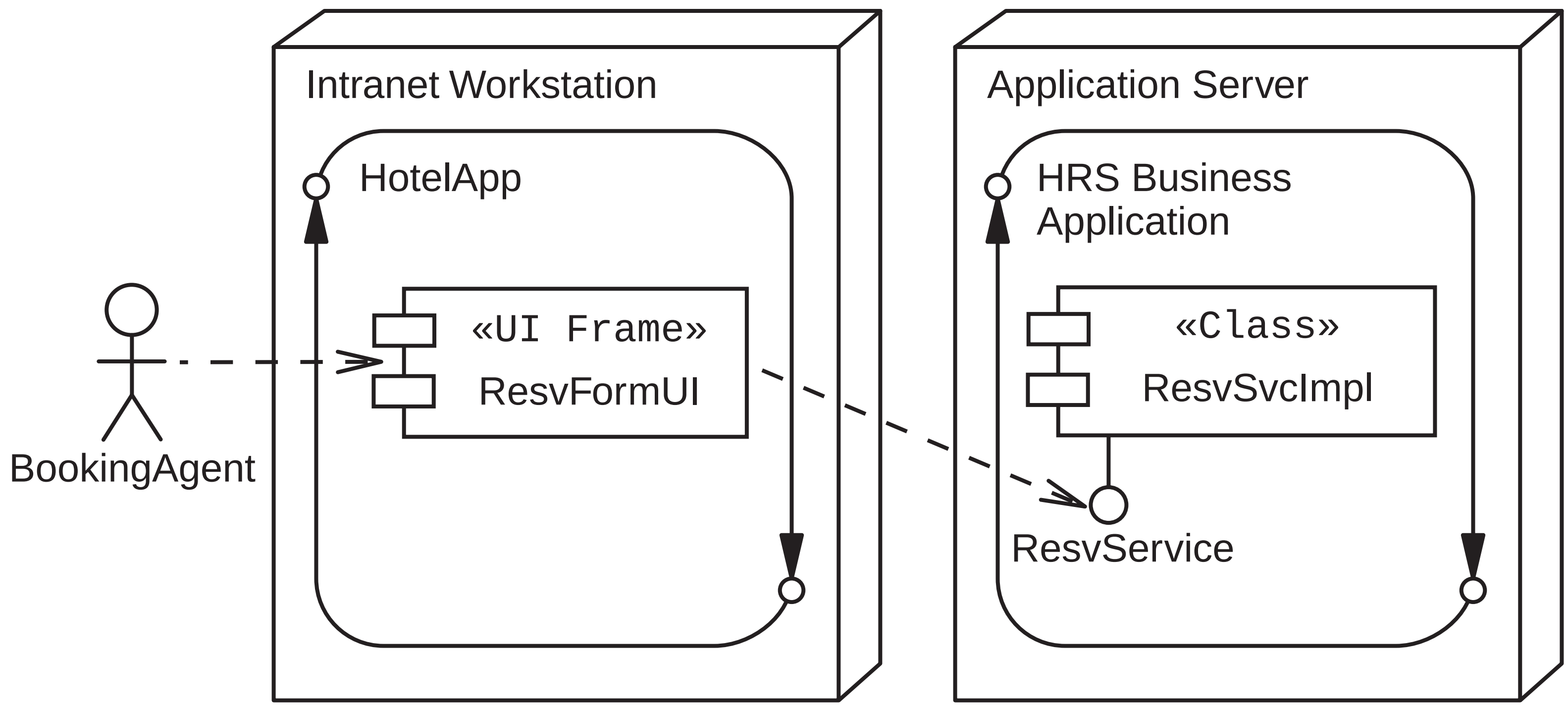
Local Access to a Service Component



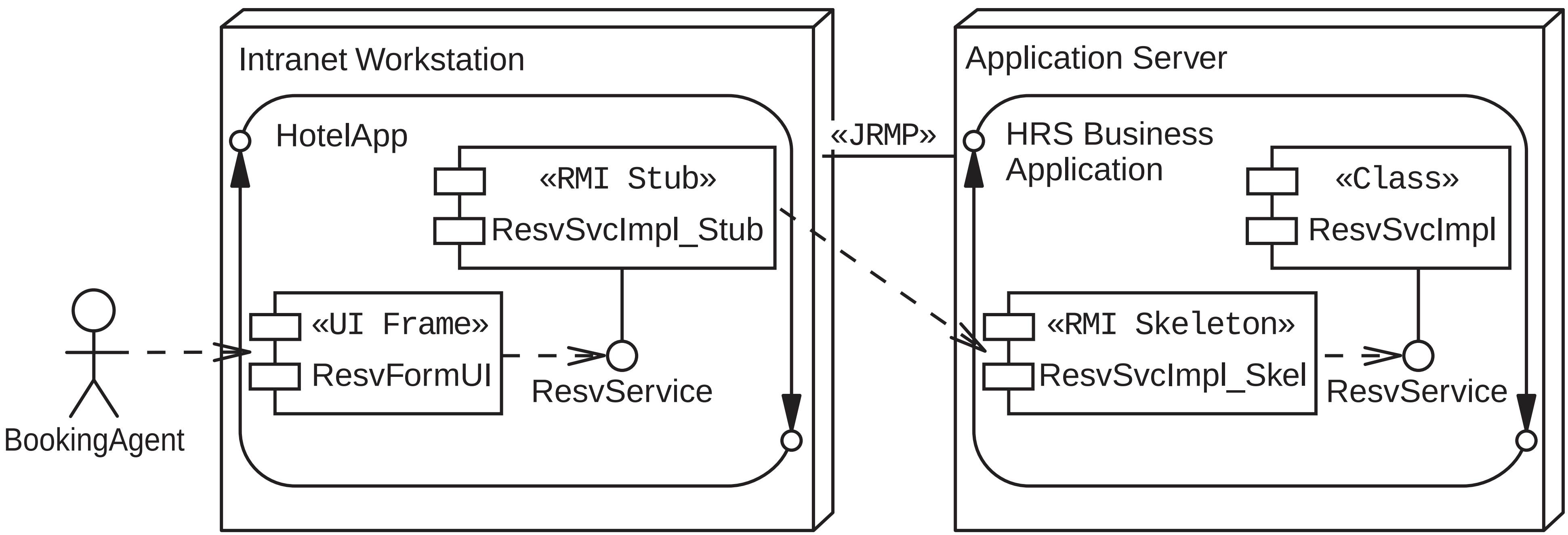
Applying Dependency Inversion Principle



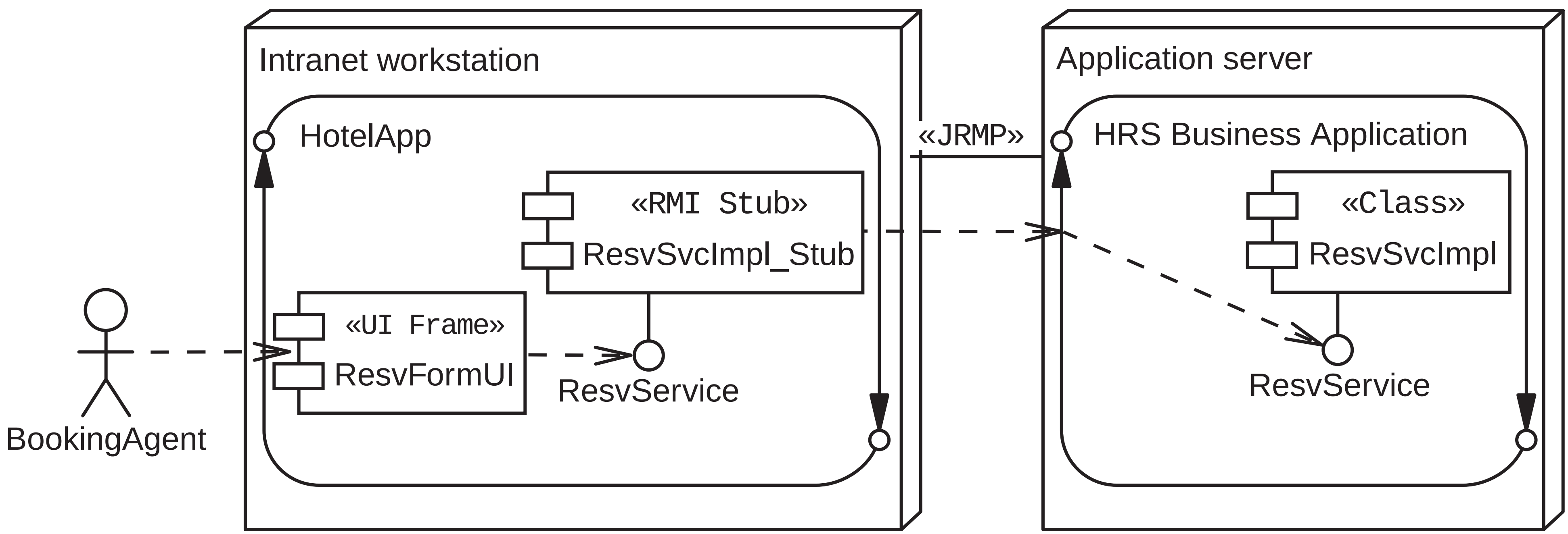
An Abstract Version of Accessing a Remote Service



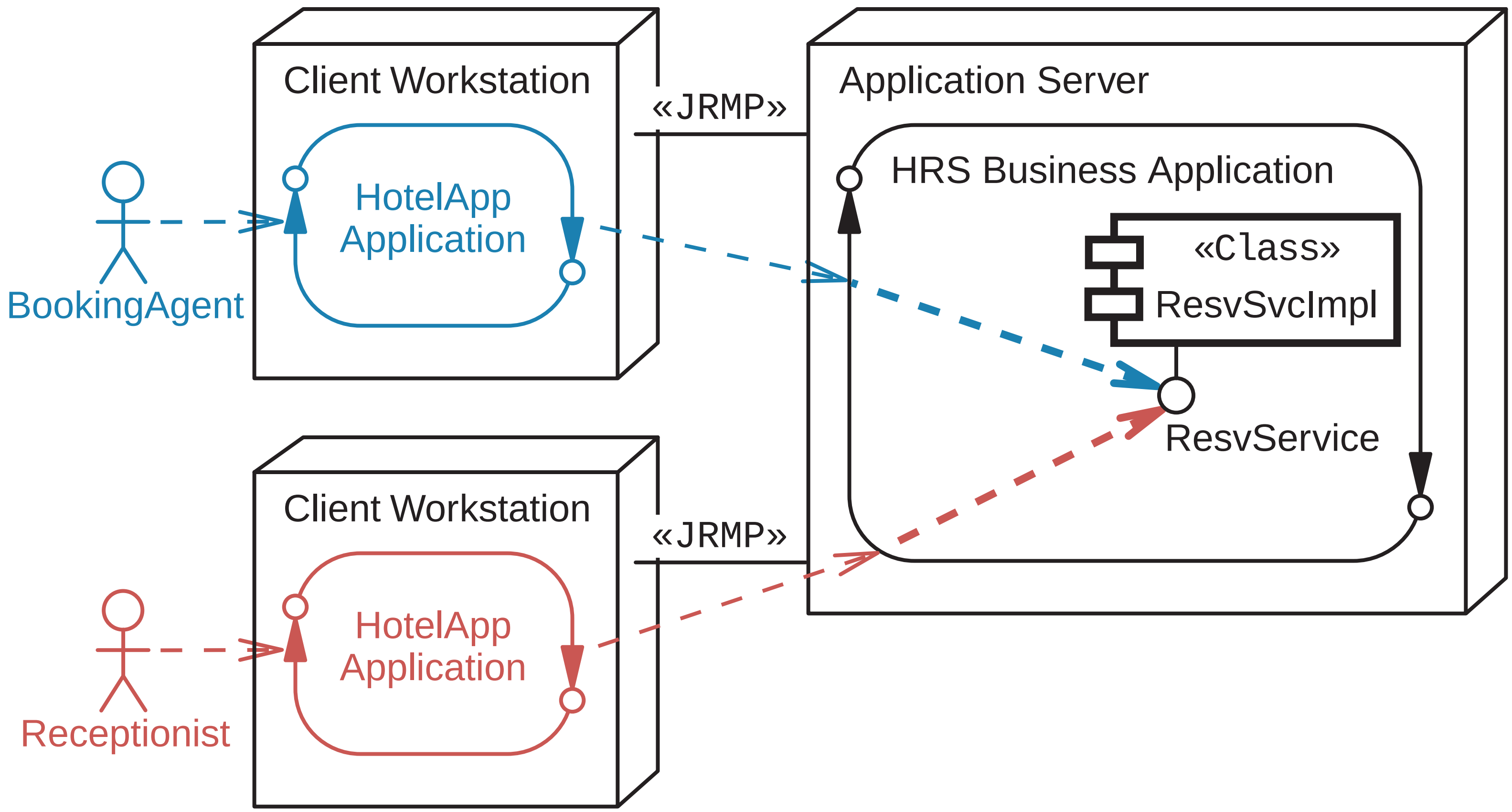
Accessing a Remote Service With RMI Infrastructure



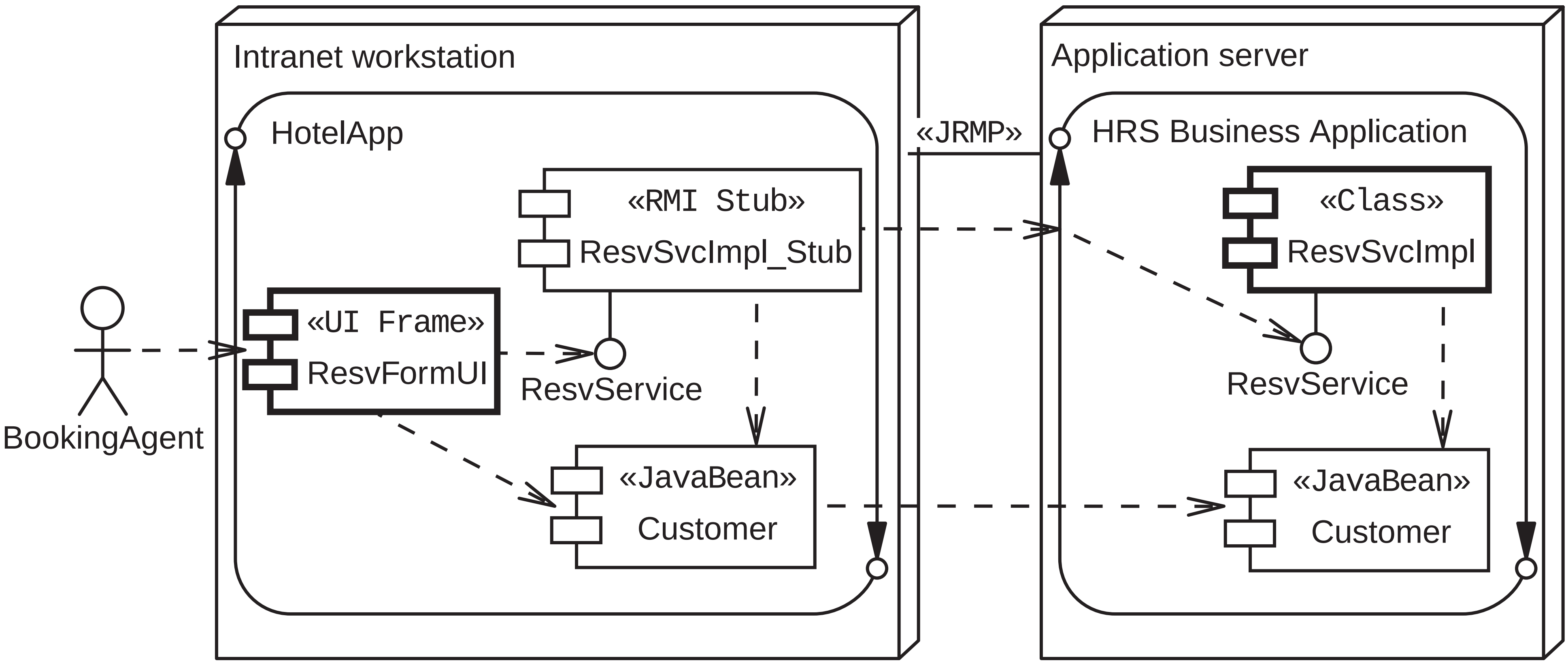
Accessing a Remote Service Without a Skeleton Component



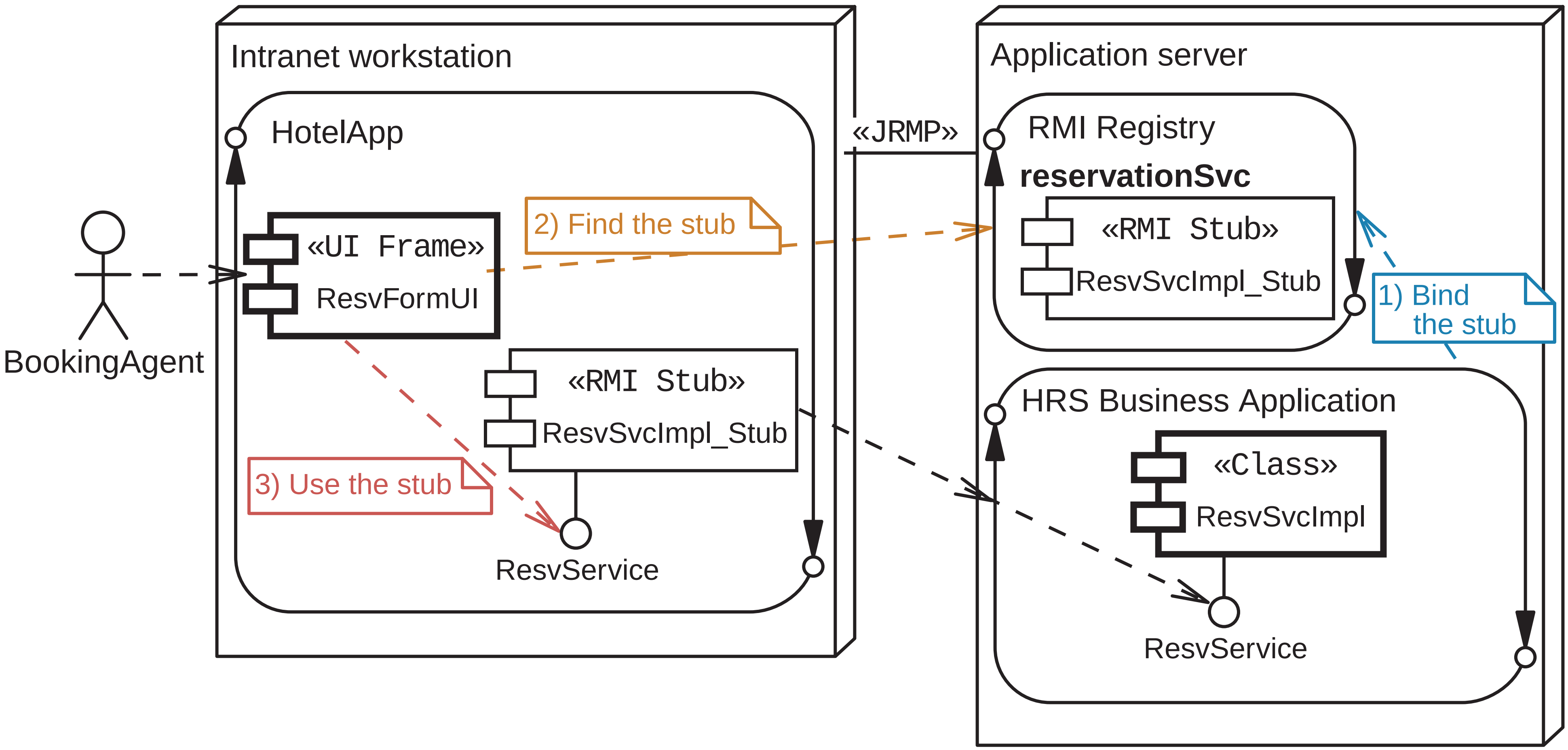
A Remote Service Is an Active Component



RMI Uses Serialization to Pass Parameters



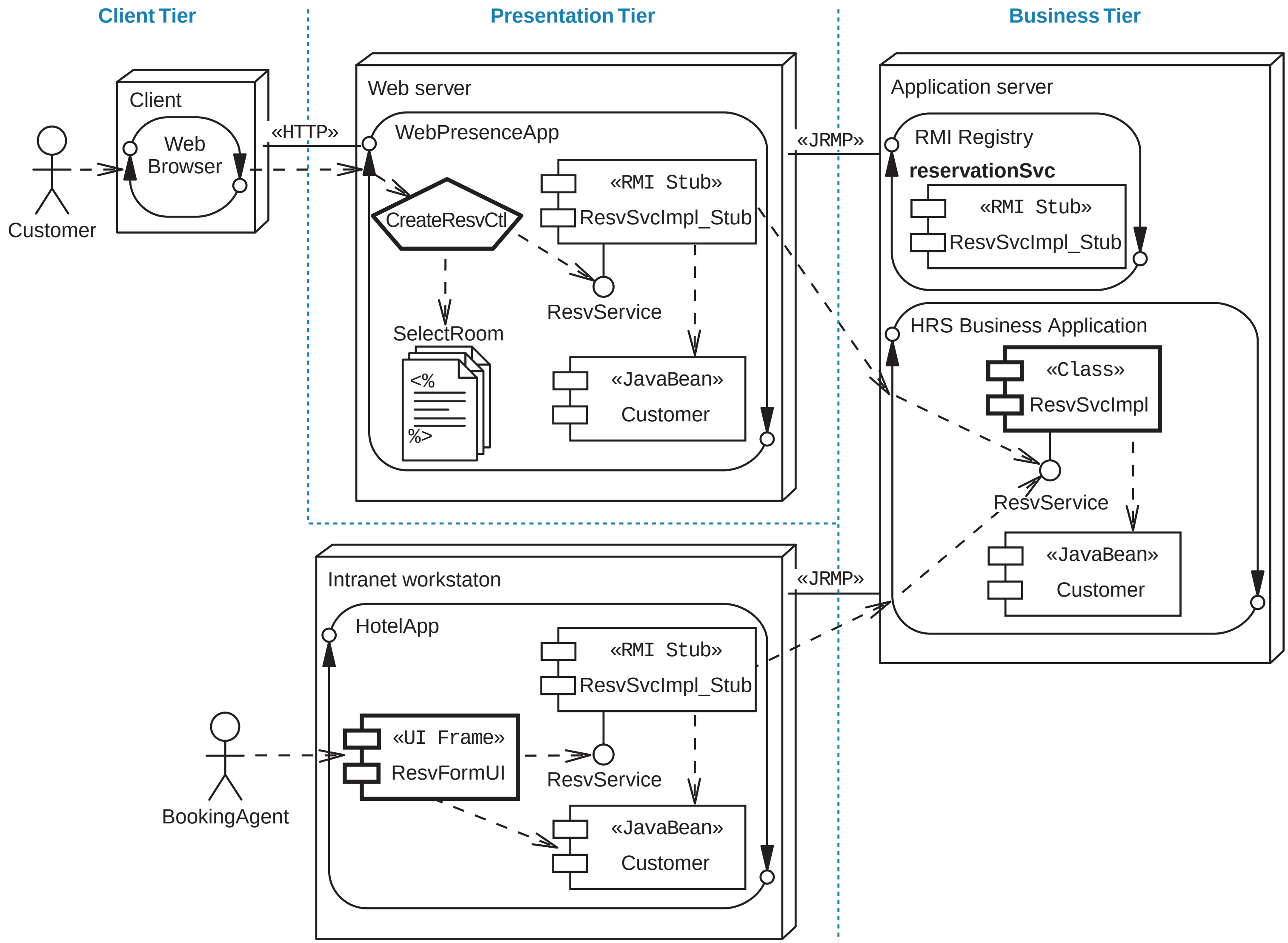
The RMI Registry Stores Stubs for Remote Lookup



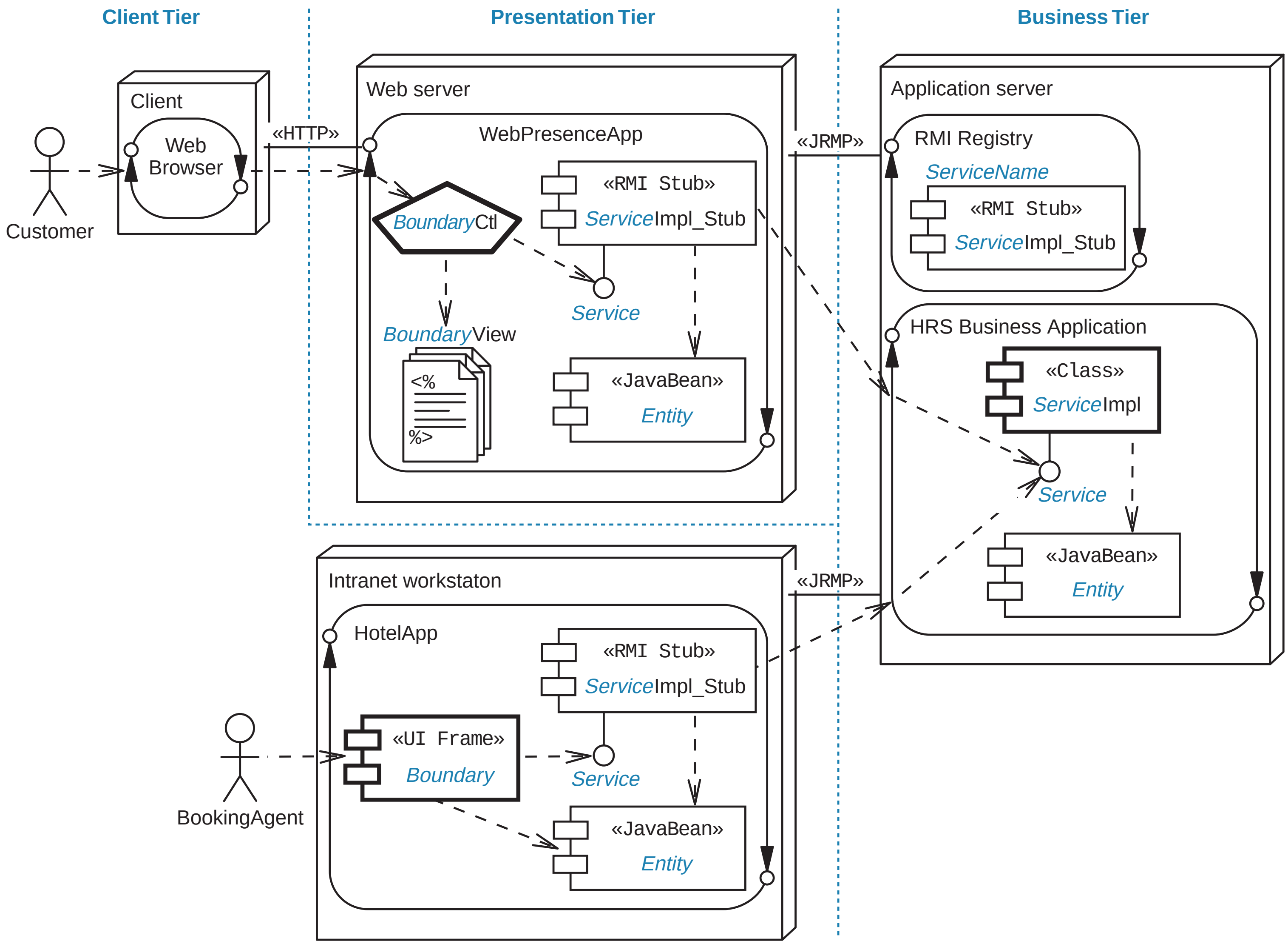
Documenting the Business Tier in the Architecture Model

- Populate the detailed Deployment diagram with the Business tier components.
- Create the Architecture template from the detailed Deployment diagram.
- Populate the tiers and layers Package diagram with the Business tier technologies.

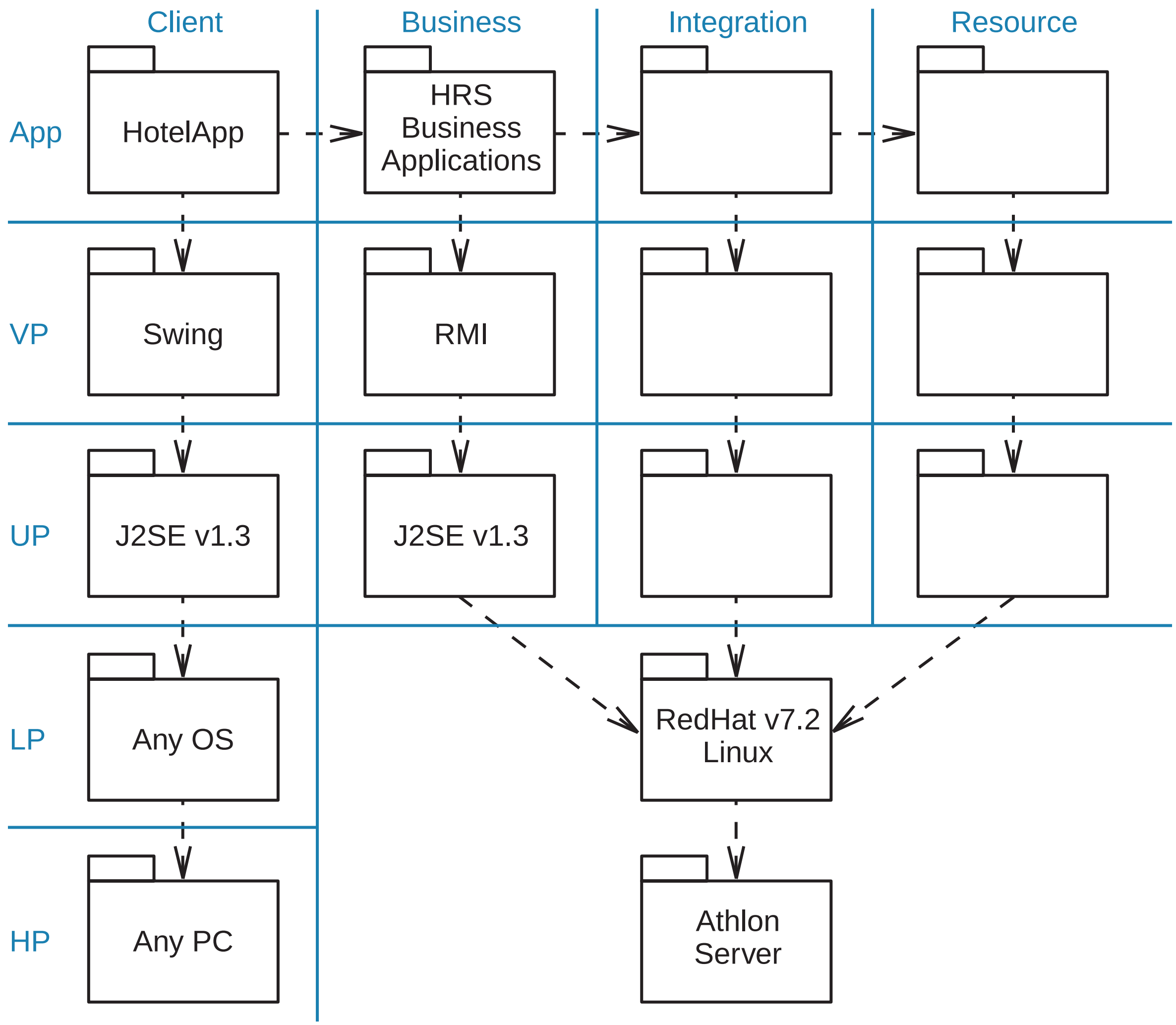
Populating the Detailed Deployment Diagram



Creating the Architecture Template



Populating the Tiers and Layers Pkg Diagram



Summary

- Distributed computing provides the necessary mechanisms to support scalable, distributed n-tier applications.
- RMI is a simple, yet powerful tool for implementing distributed object computing.

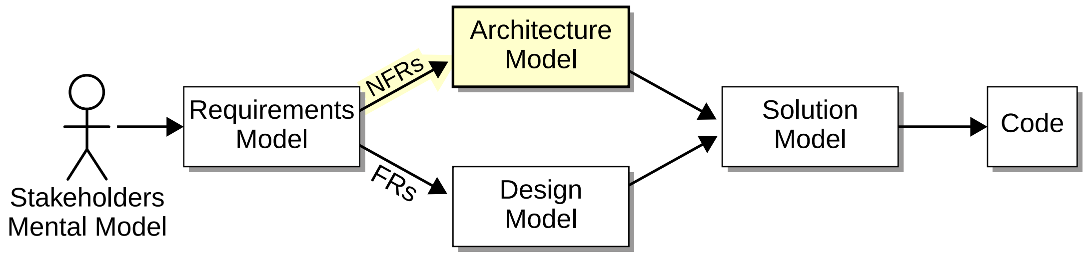
Module 15

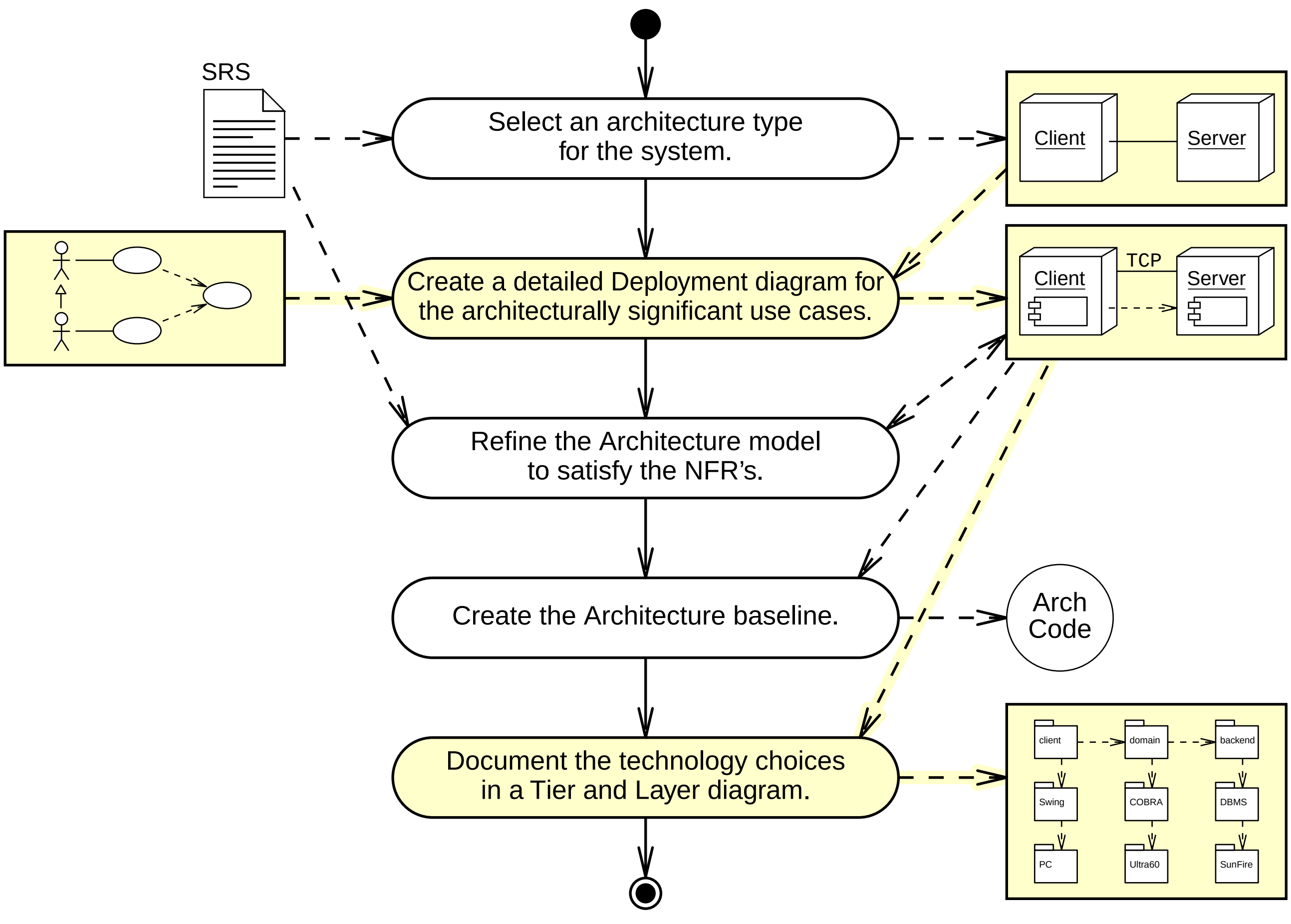
Creating an Architectural Model for the Resource and Integration Tiers

Objectives

- Document the persistence mechanism in the Resource tier of the Architecture model
- Document the persistence integration mechanism in the Integration tier of the Architecture model

Process Map


Architecture Model
NFRs
Requirements Model
Solution Model
Code
FRs
Design Model
Stakeholders Mental Model


SRS
Select an architecture type for the system.
Client
Server
Create a detailed Deployment diagram for the architecturally significant use cases.
Client
TCP
Server
Refine the Architecture model to satisfy the NFR's.
Create the Architecture baseline.
Arch Code
Document the technology choices in a Tier and Layer diagram.
client
domain
backend
Swing
COBRA
DBMS
PC
Ultra60
SunFire

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

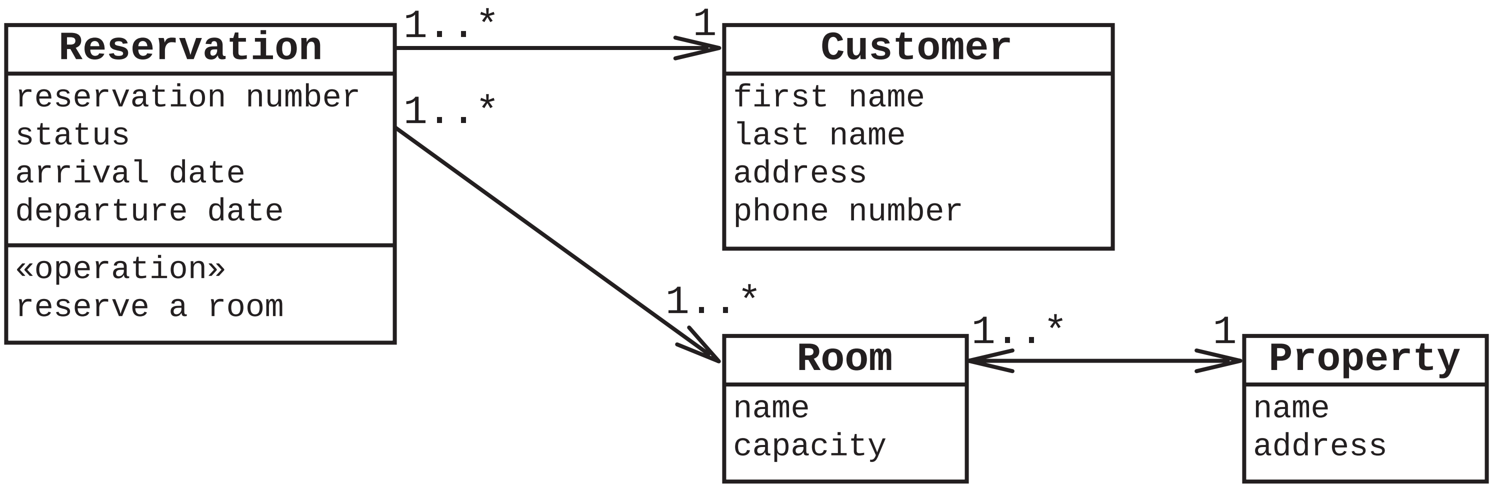
Creating a Database Schema for the Domain Model

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.

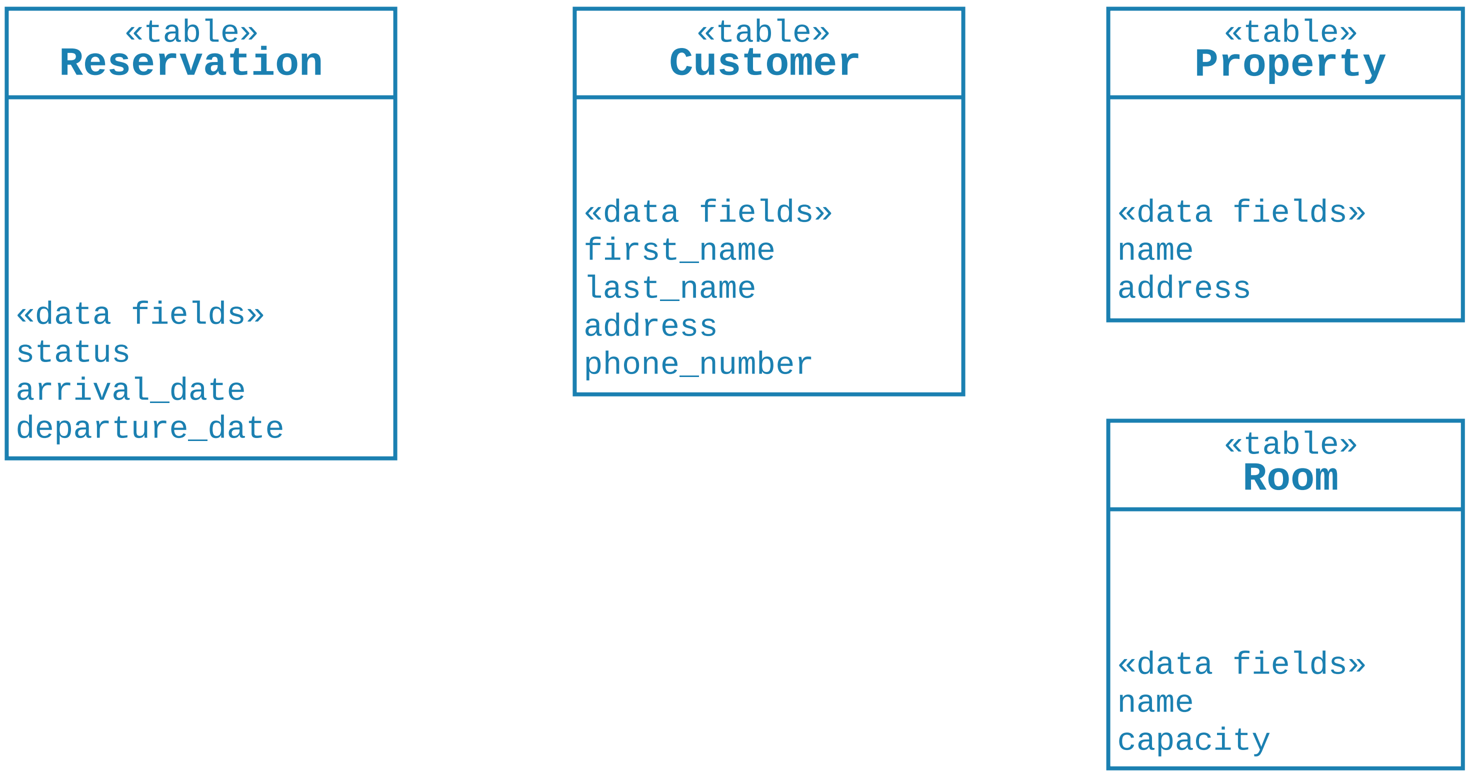
Simplified HRS Domain Model

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

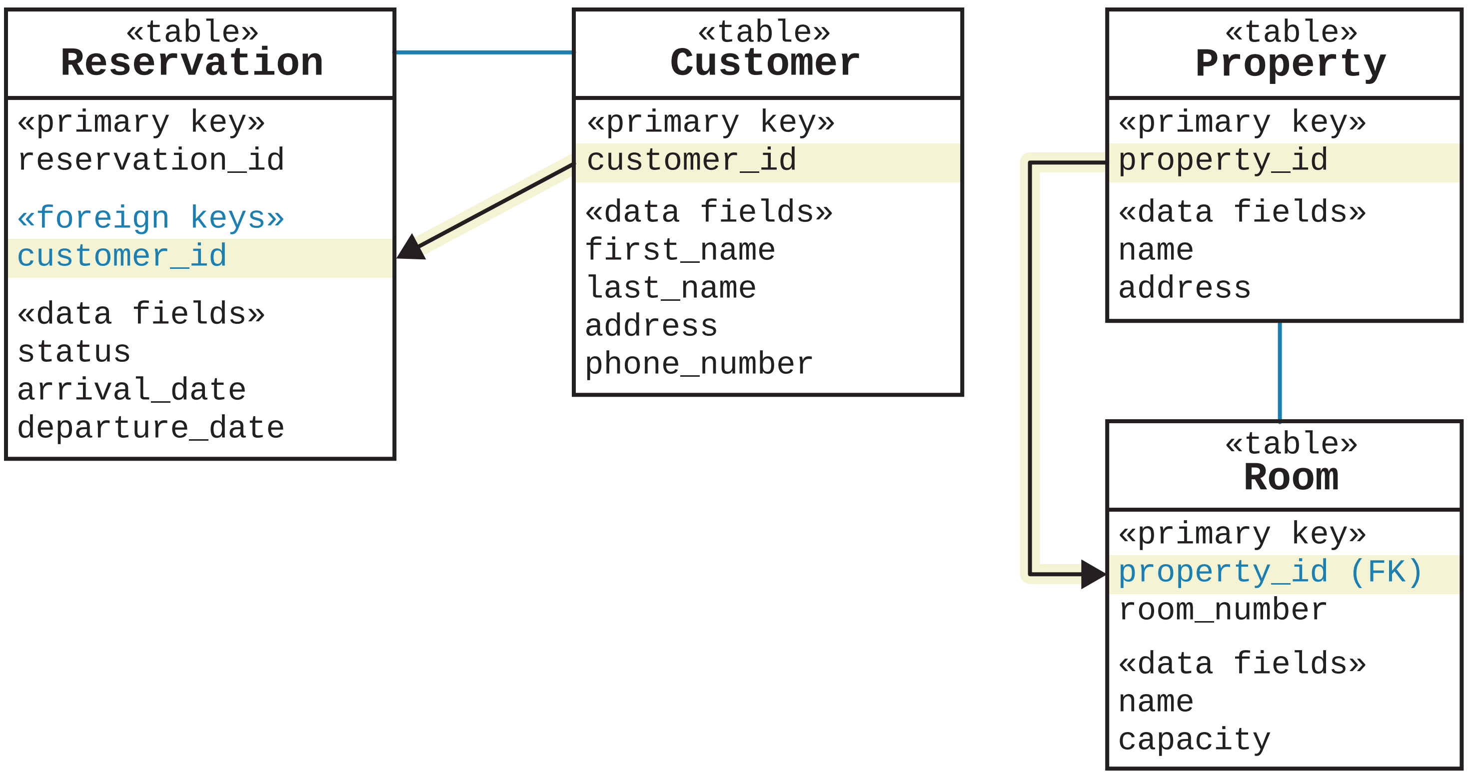
«table»
Room

«primary key»
property_id (FK)
room_number

«data fields»
name
capacity

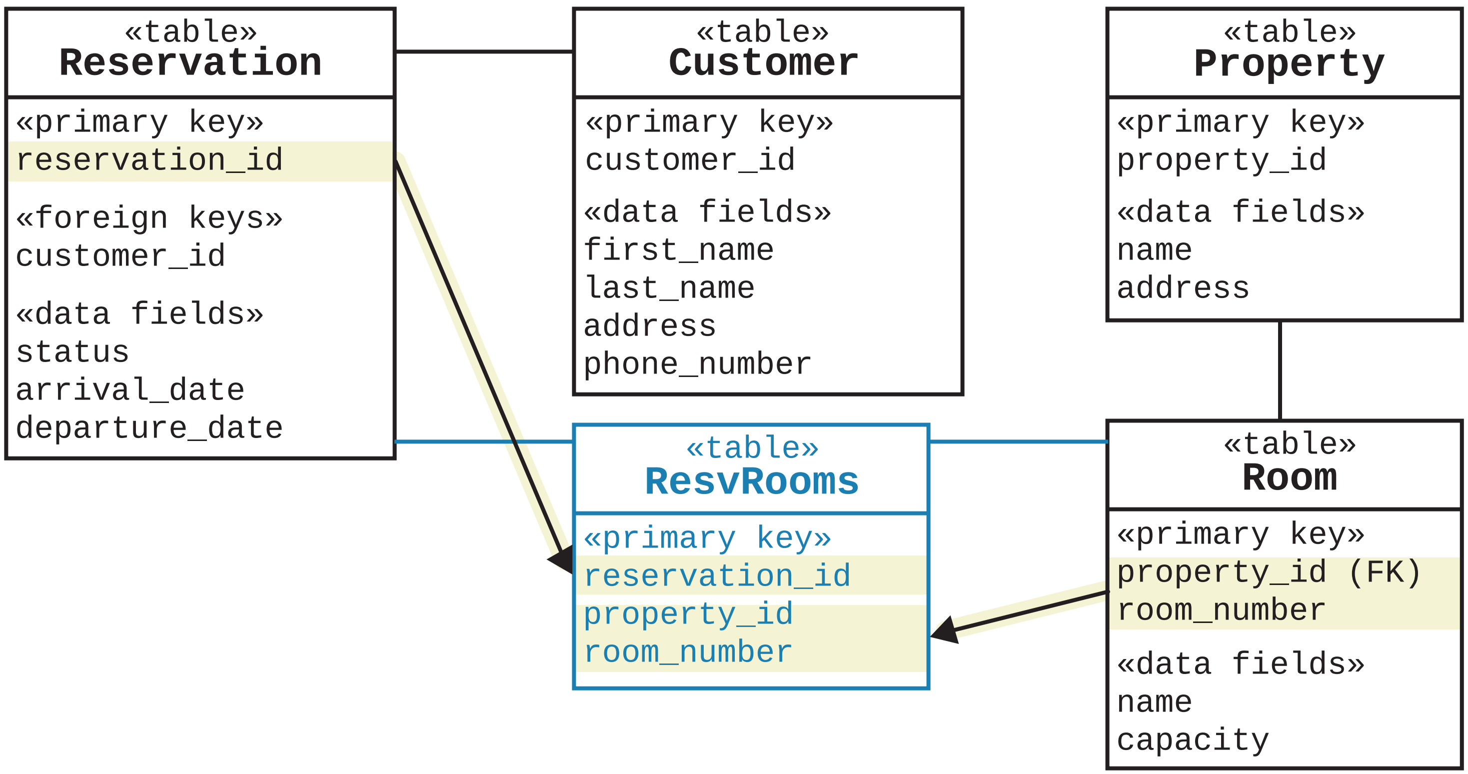
Step 3 – Specify One-to-Many Relationships

Use foreign keys:



Step 3 – Specify Many-to-Many Relationships

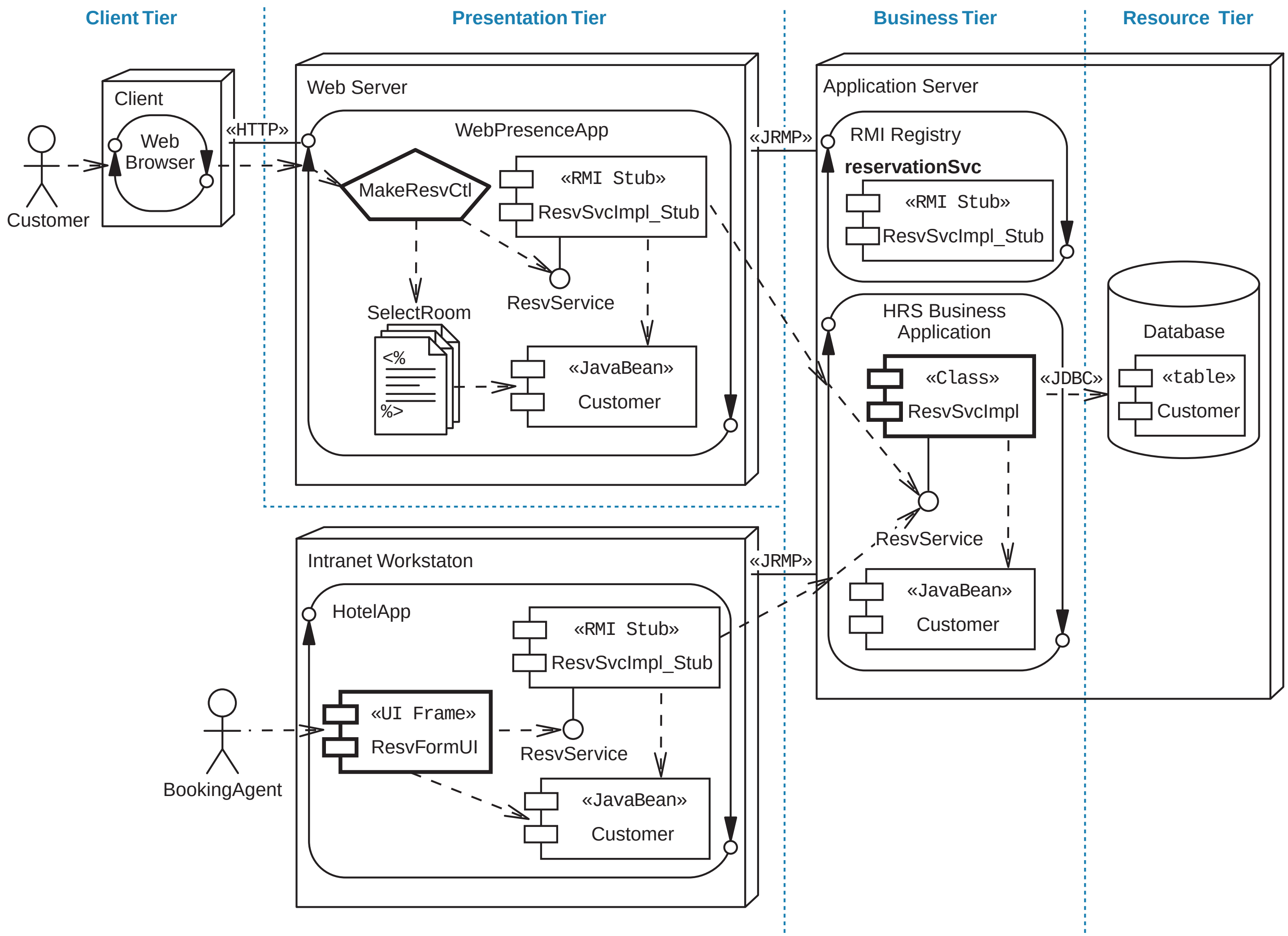
Introduce a resolution table:



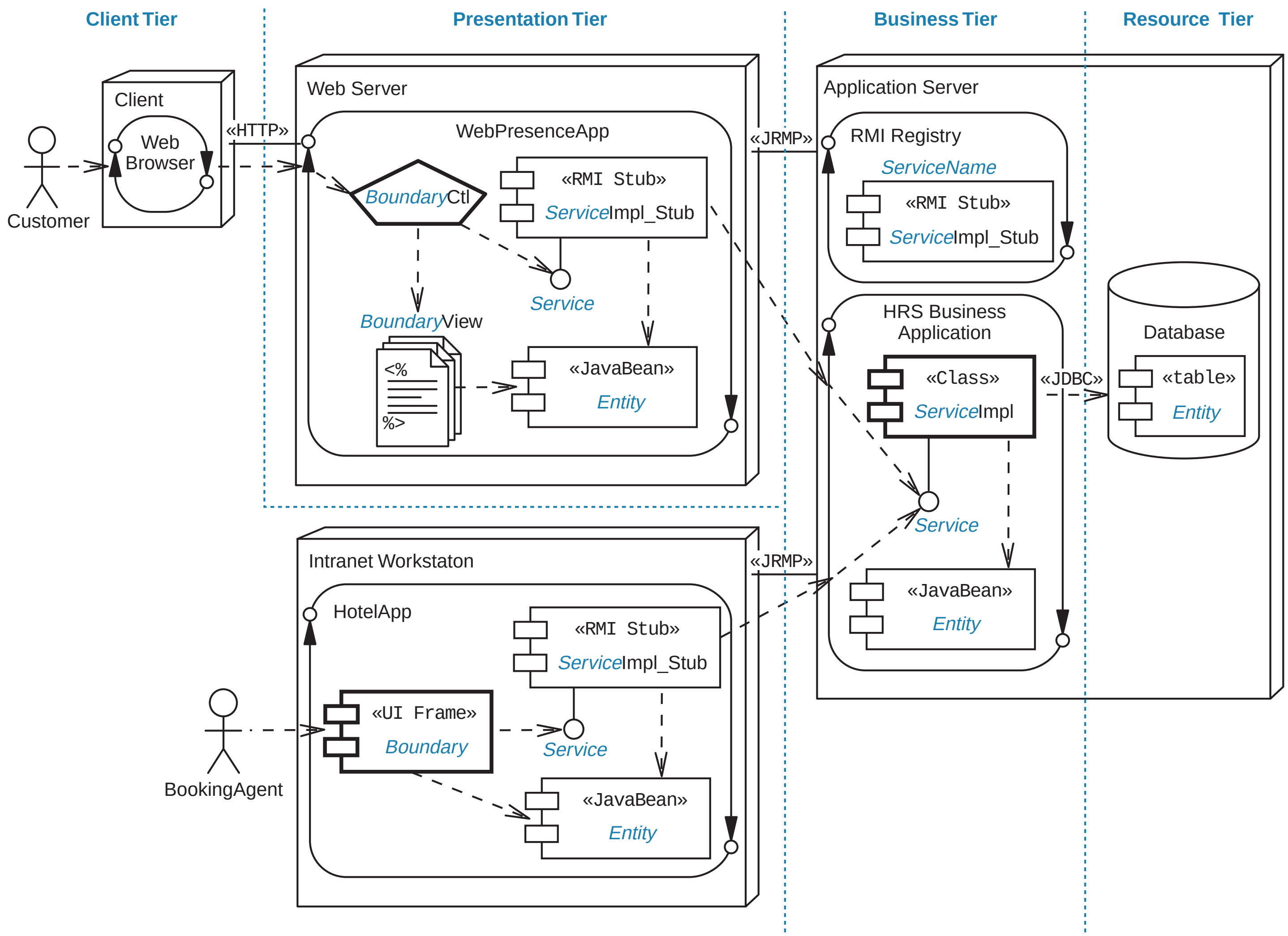
Recording the Resource Tier in the Architecture Model

1. Populate the detailed Deployment diagram with the Resource tier components.
2. Create the Architecture template from the detailed Deployment diagram.
3. Populate the tiers and layers Package diagram with the Resource tier technologies.

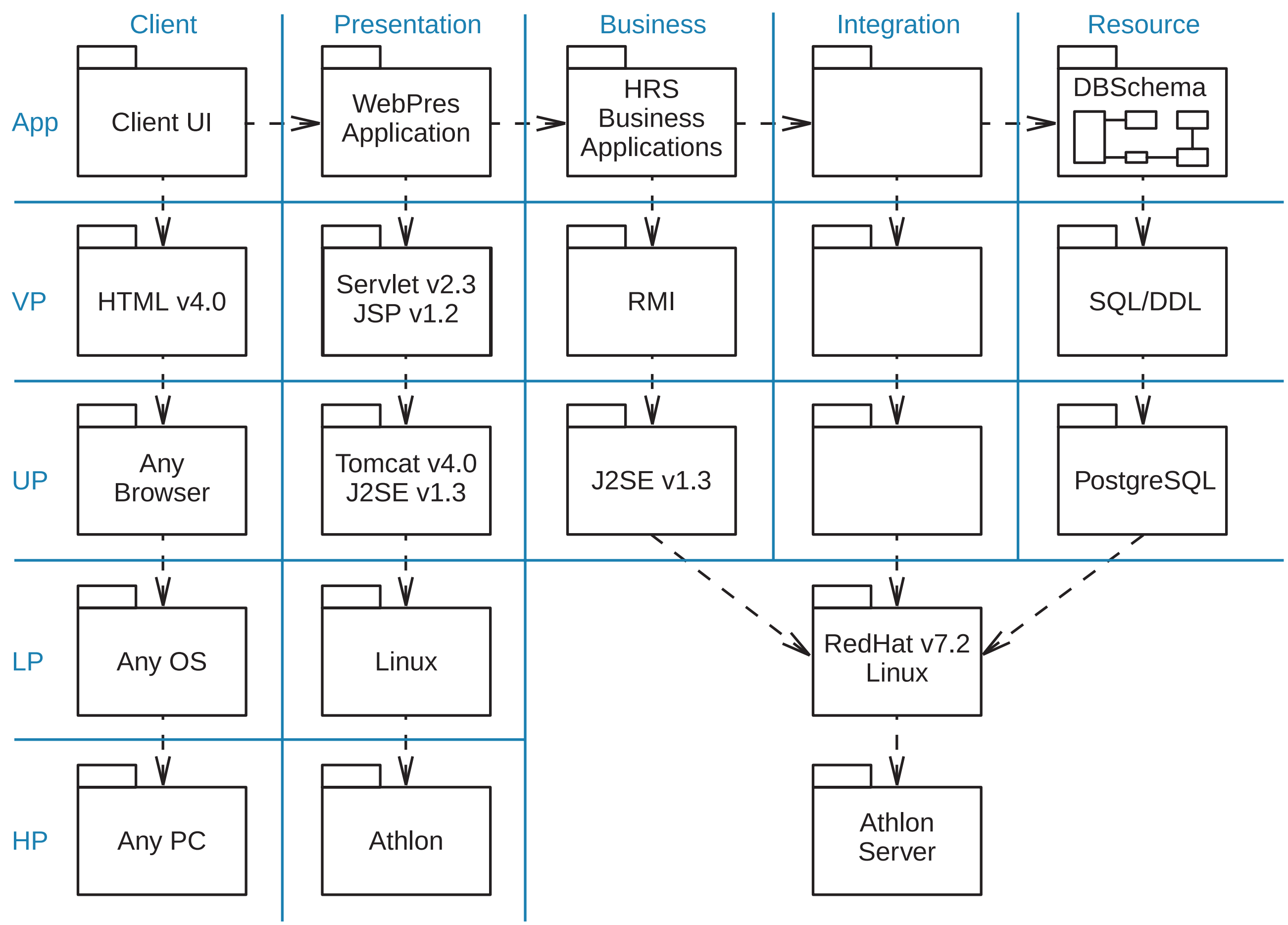
Populating the Detailed Deployment Diagram



Creating the Architecture Template



Populating the Tiers and Layers Pkg Diagram



Exploring Integration Tier Technologies

Resources that require integration are:

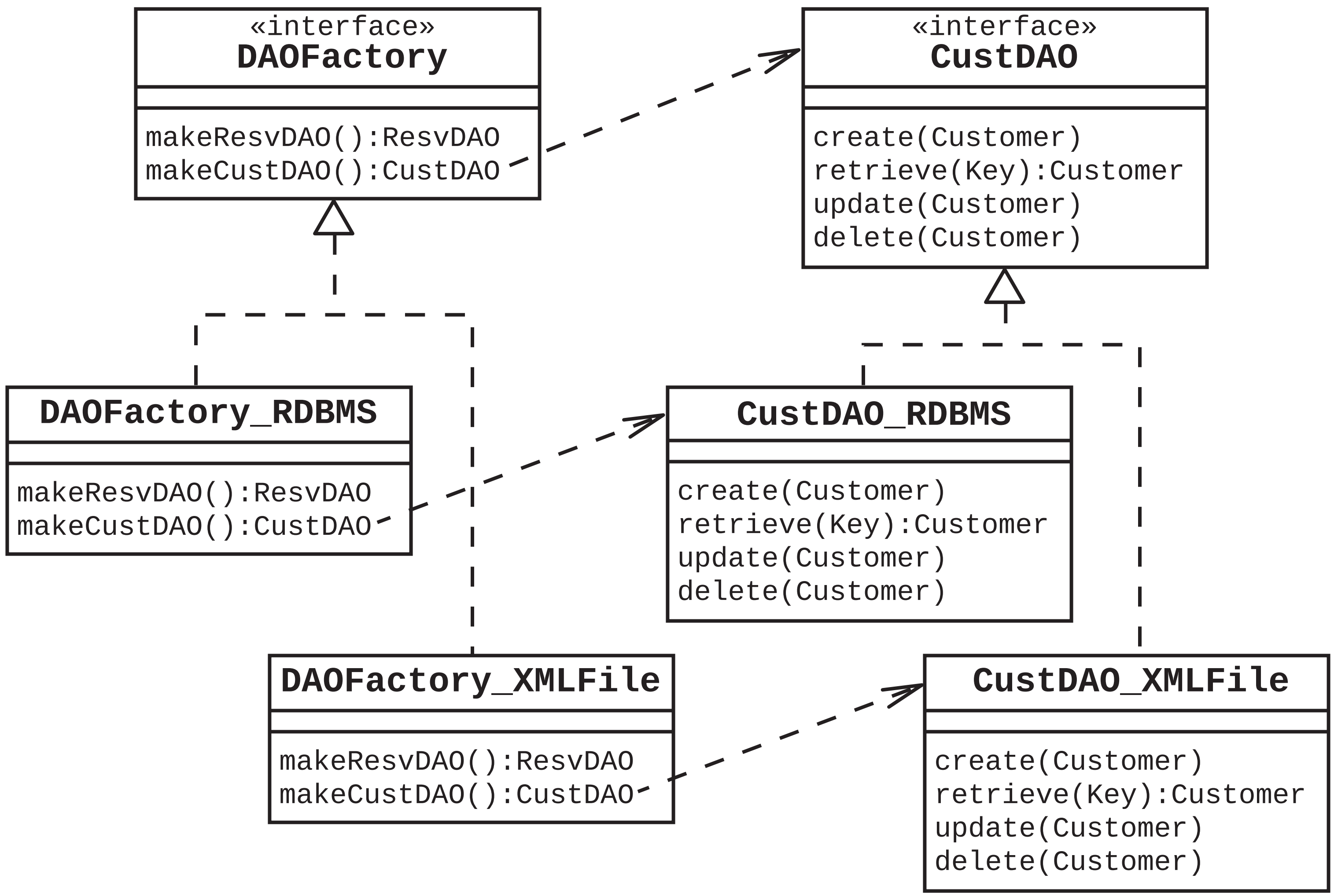
- Data sources
- Enterprise Information Systems (EIS)
- Computation libraries
- Message services
- B2B services

The DAO Pattern

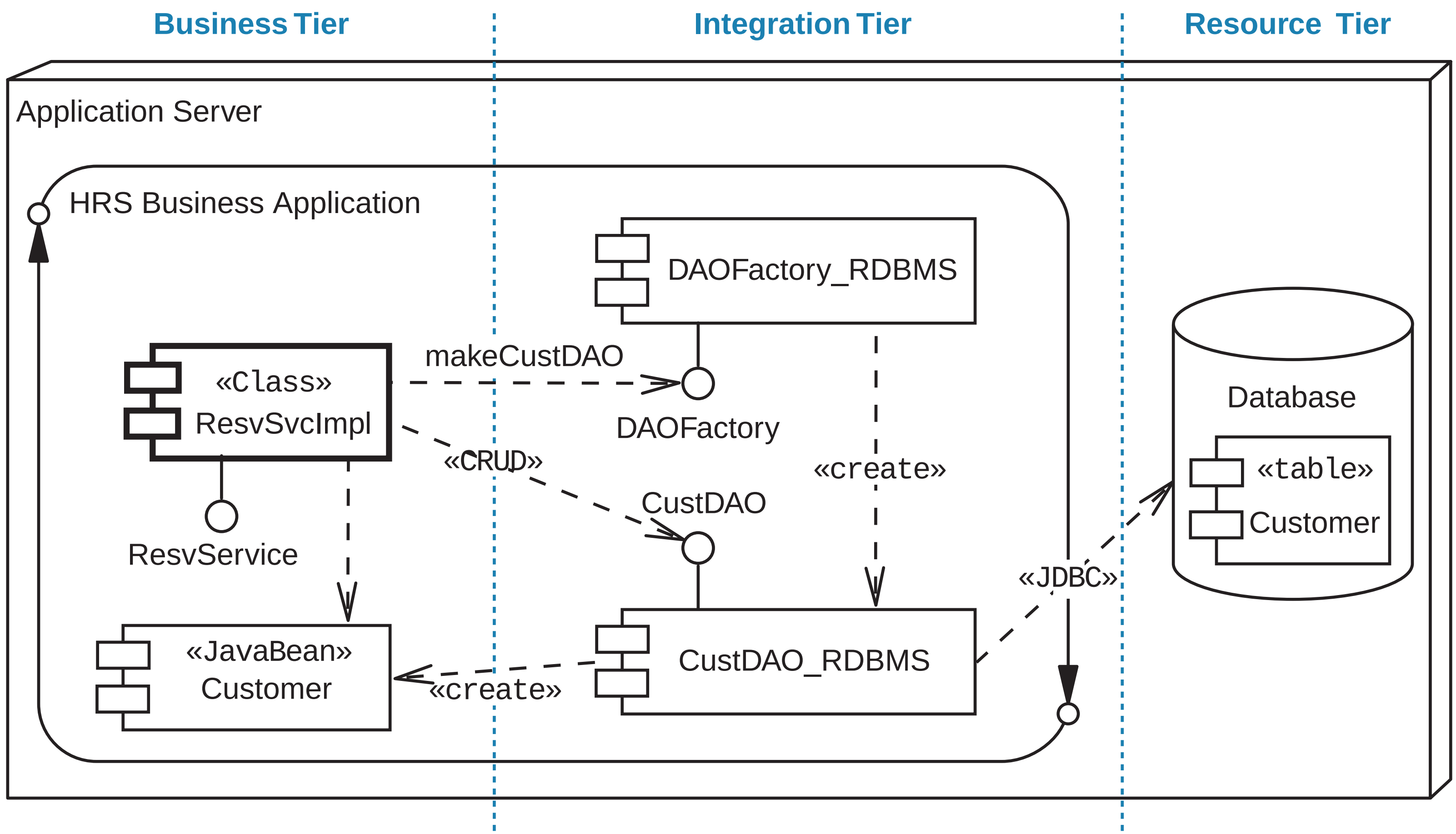
The Data Access Object (DAO) Pattern:

- 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



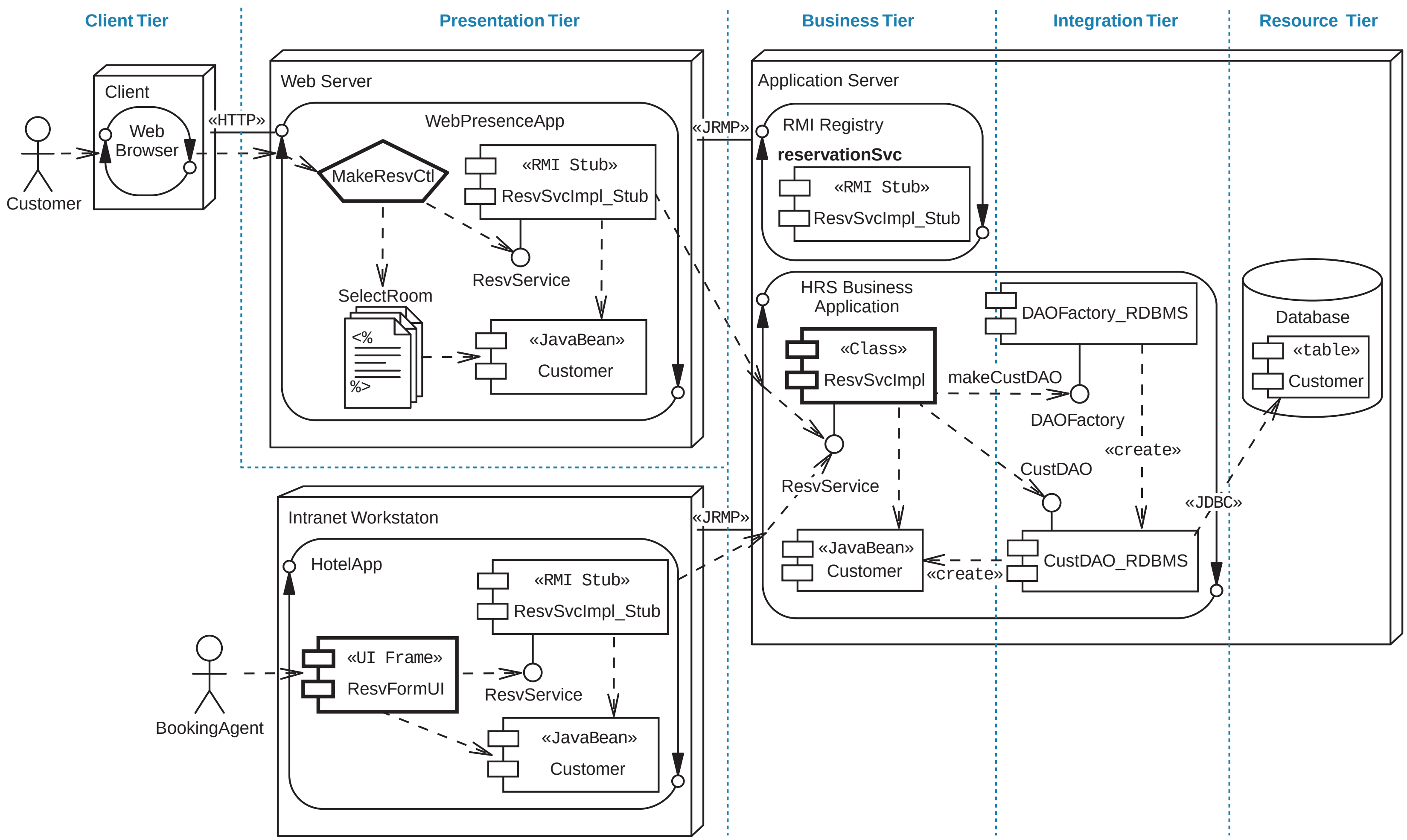
The DAO Pattern



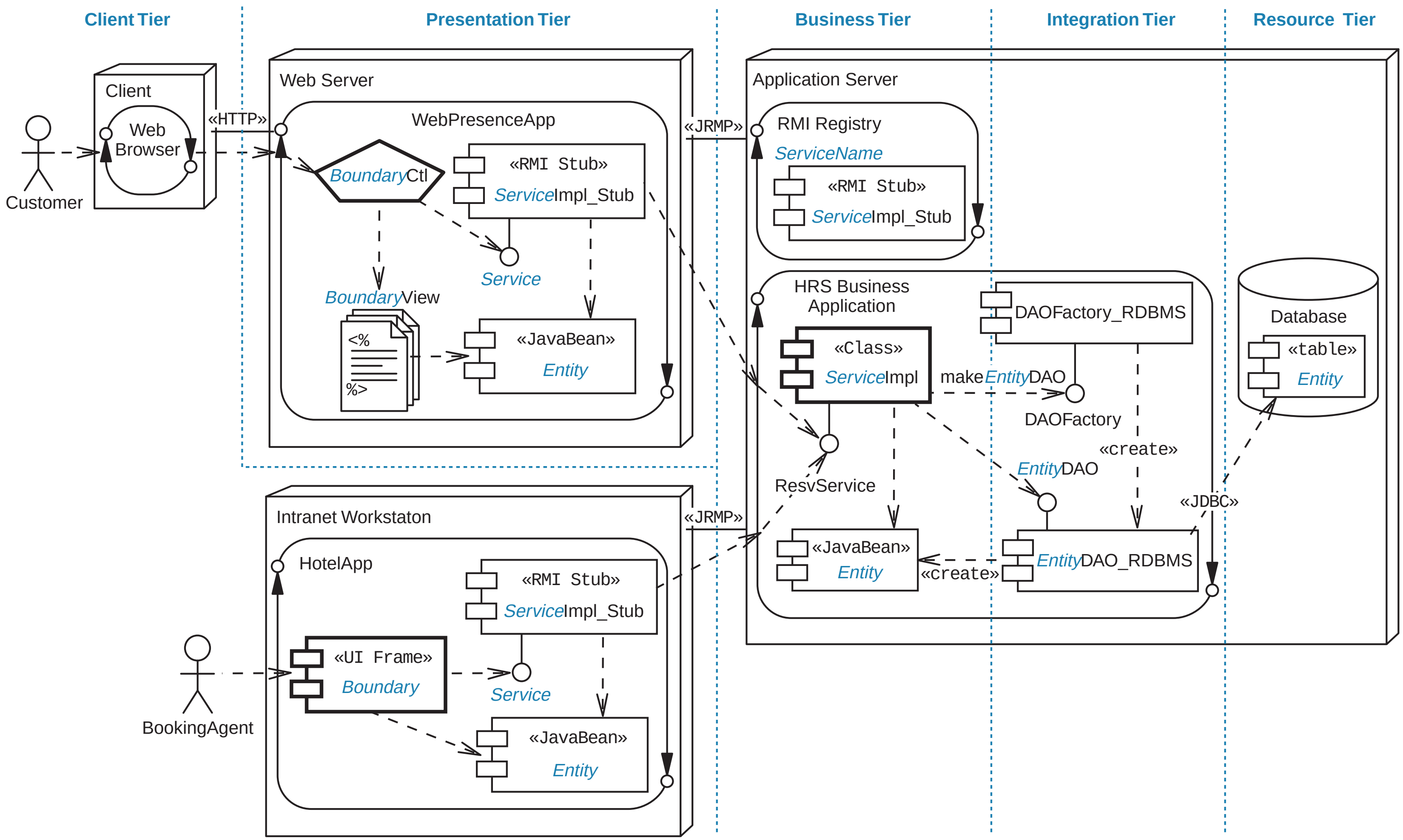
Recording the Integration Tier in the Architecture Model

1. Populate the detailed Deployment diagram with the Integration tier components.
2. Create the Architecture template from the detailed Deployment diagram.
3. Populate the tiers and layers Package diagram with the Integration tier technologies.

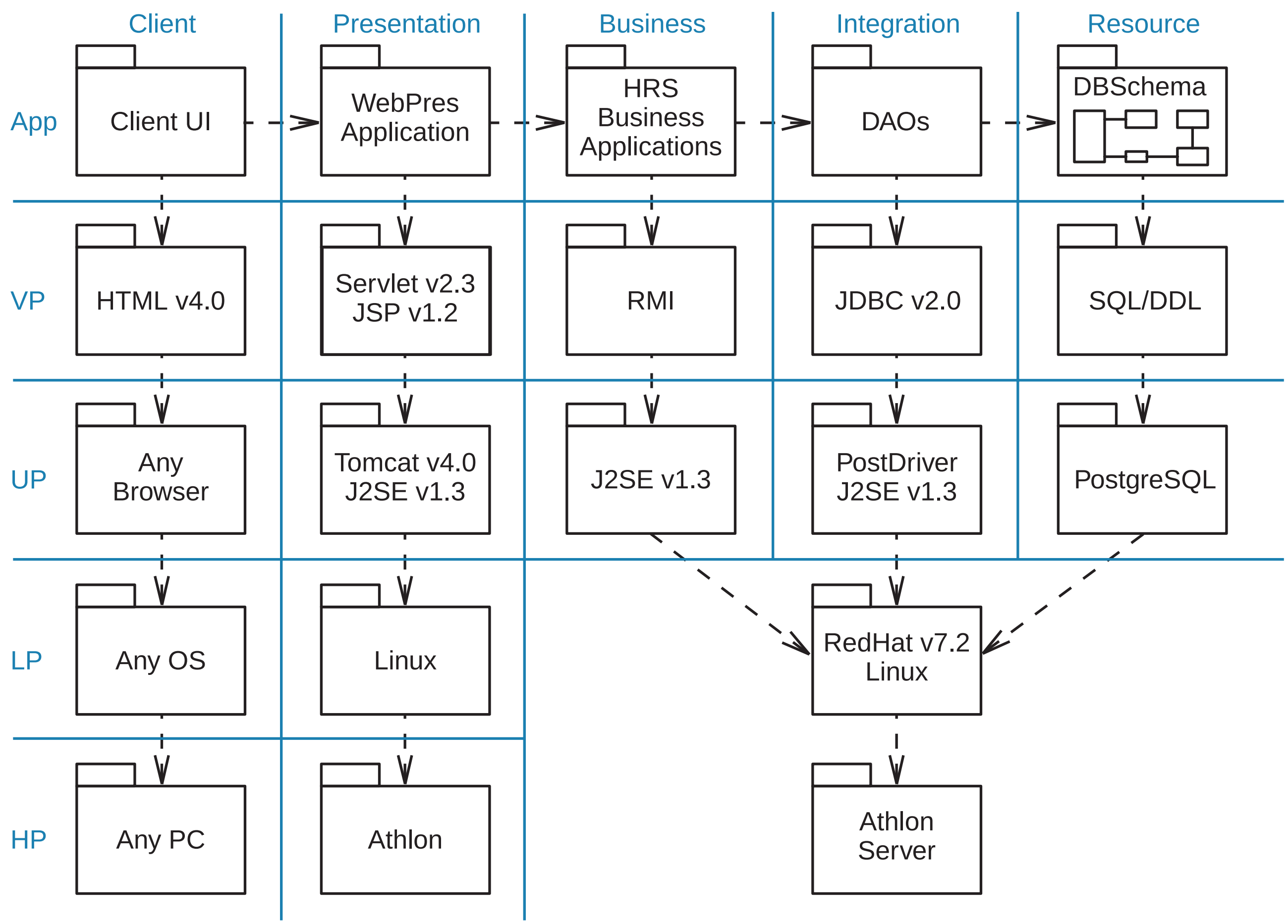
Populating the Detailed Deployment Diagram



Creating the Architecture Template



Populating the Tiers and Layers Pkg Diagram



Summary

- The Resource tier usually includes one or more data sources such as an RDBMS, OODBMS, EIS, or files.
- The Integration tier includes the components that provide CRUD operations to the Business tier. These components can be designed using the DAO pattern to support the Separation of Concerns between the Business and Resource tiers.

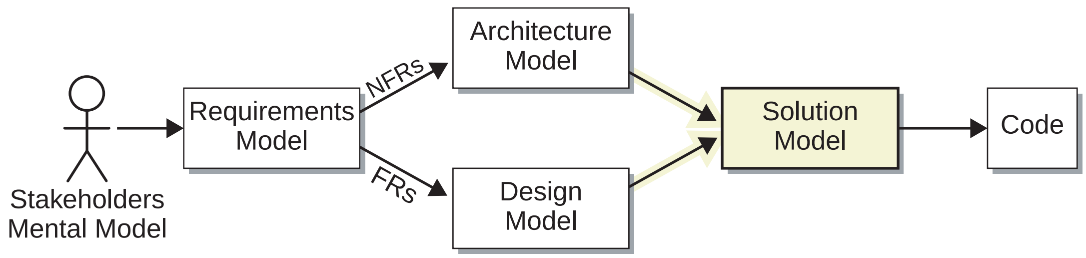
Module 16

Creating the Solution Model

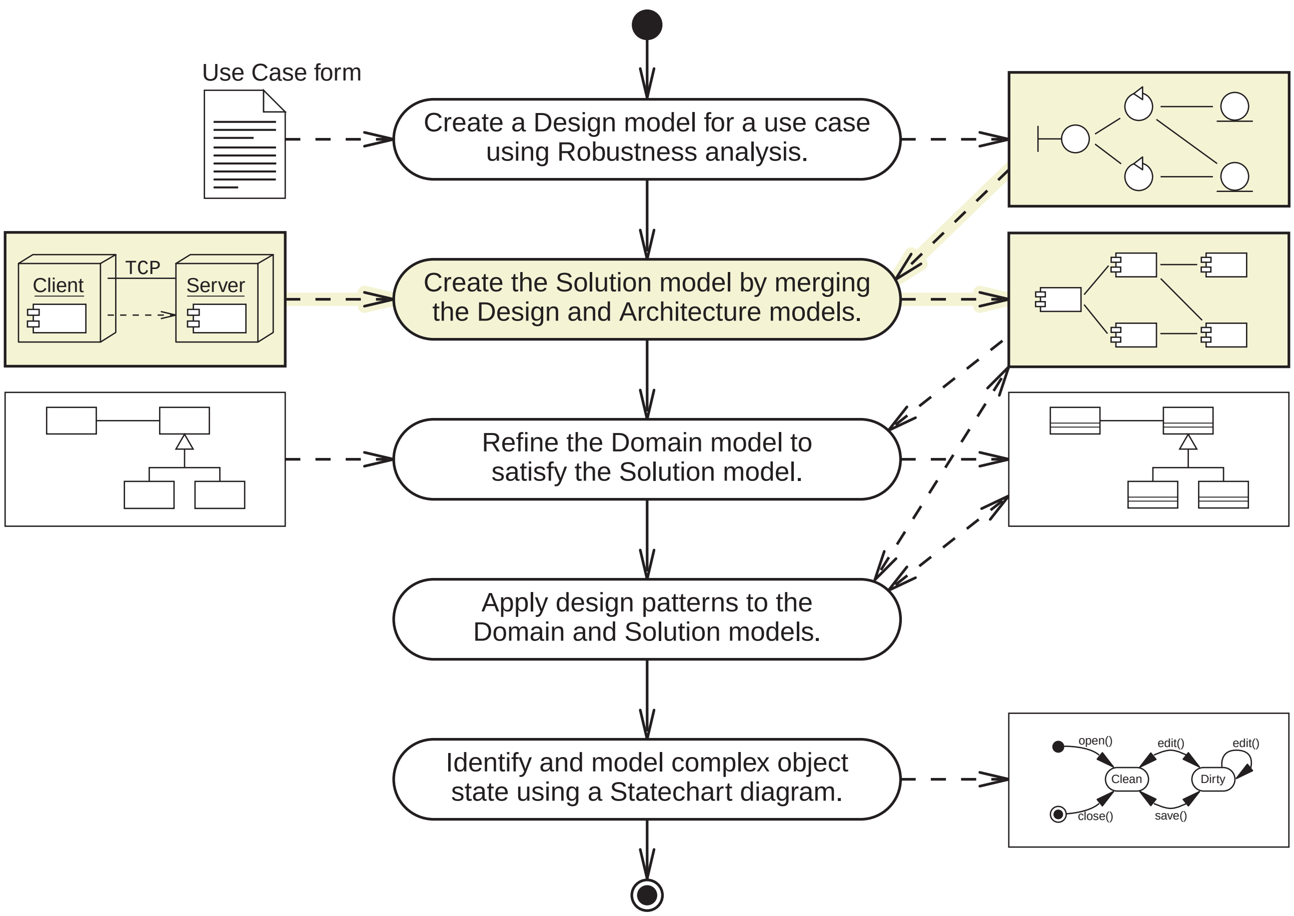
Objectives

- Create a Solution model for a GUI application
- Create a Solution model for a Web UI application

Process Map



Architecture Model
NFRs
Requirements Model
Solution Model
Code
FRs
Stakeholders Mental Model
Design Model

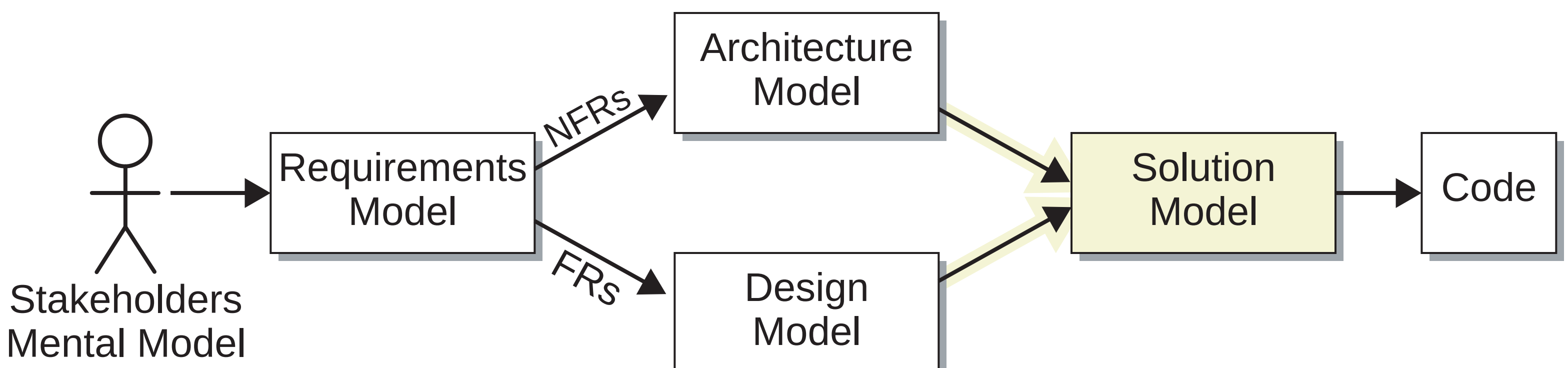


Use Case form
Create a Design model for a use case using Robustness analysis.
Client
TCP
Server
Create the Solution model by merging the Design and Architecture models.
Refine the Domain model to satisfy the Solution model.
Apply design patterns to the Domain and Solution models.
Identify and model complex object state using a Statechart diagram.
open()
edit()
edit()
Clean
Dirty
close()
save()

Introducing the Solution Model

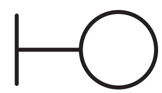
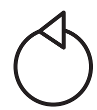
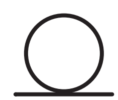
The Solution model is the basis upon which the development team will construct the code of the system solution.

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

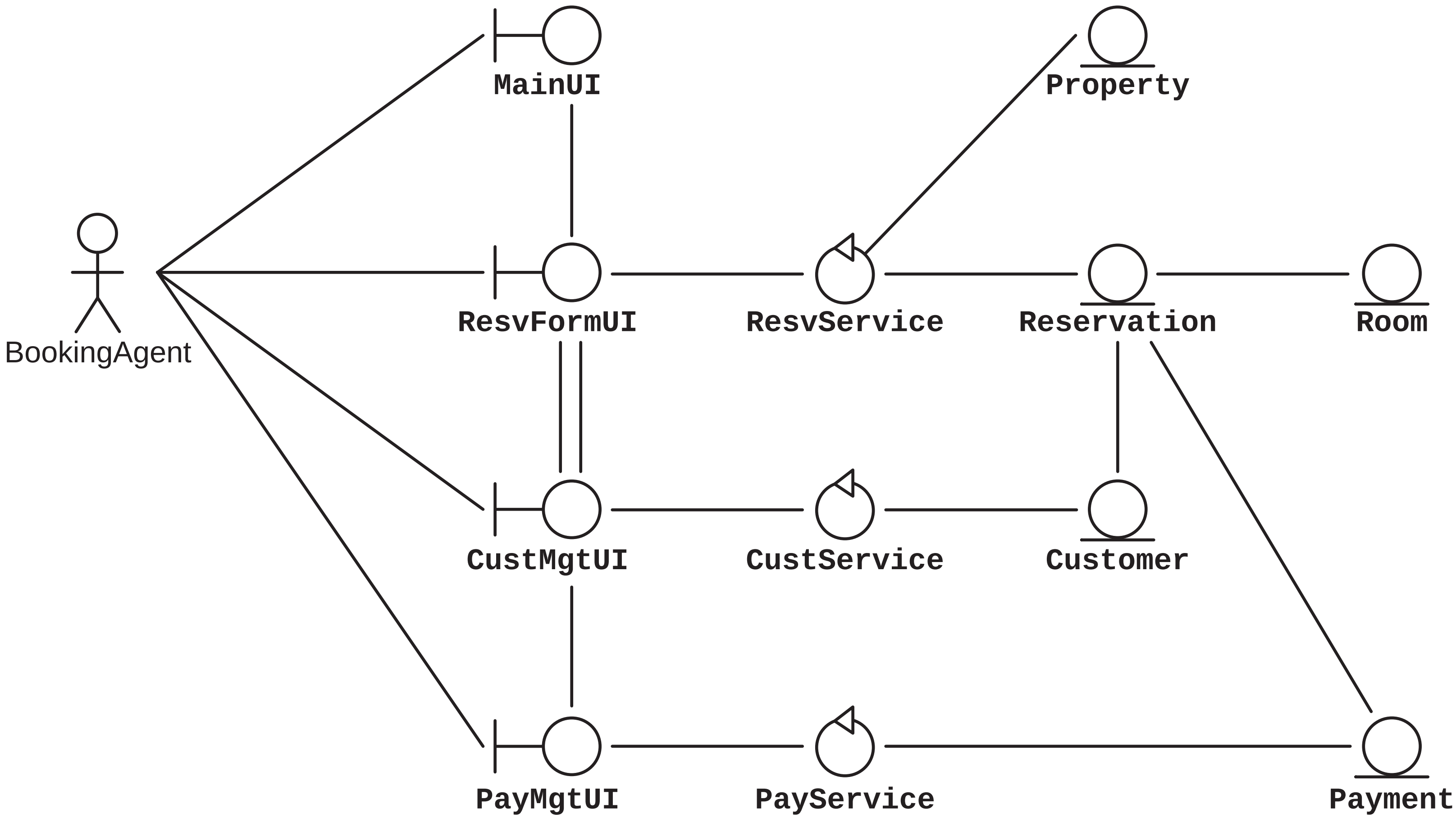


Creating the Solution Model for GUI Applications

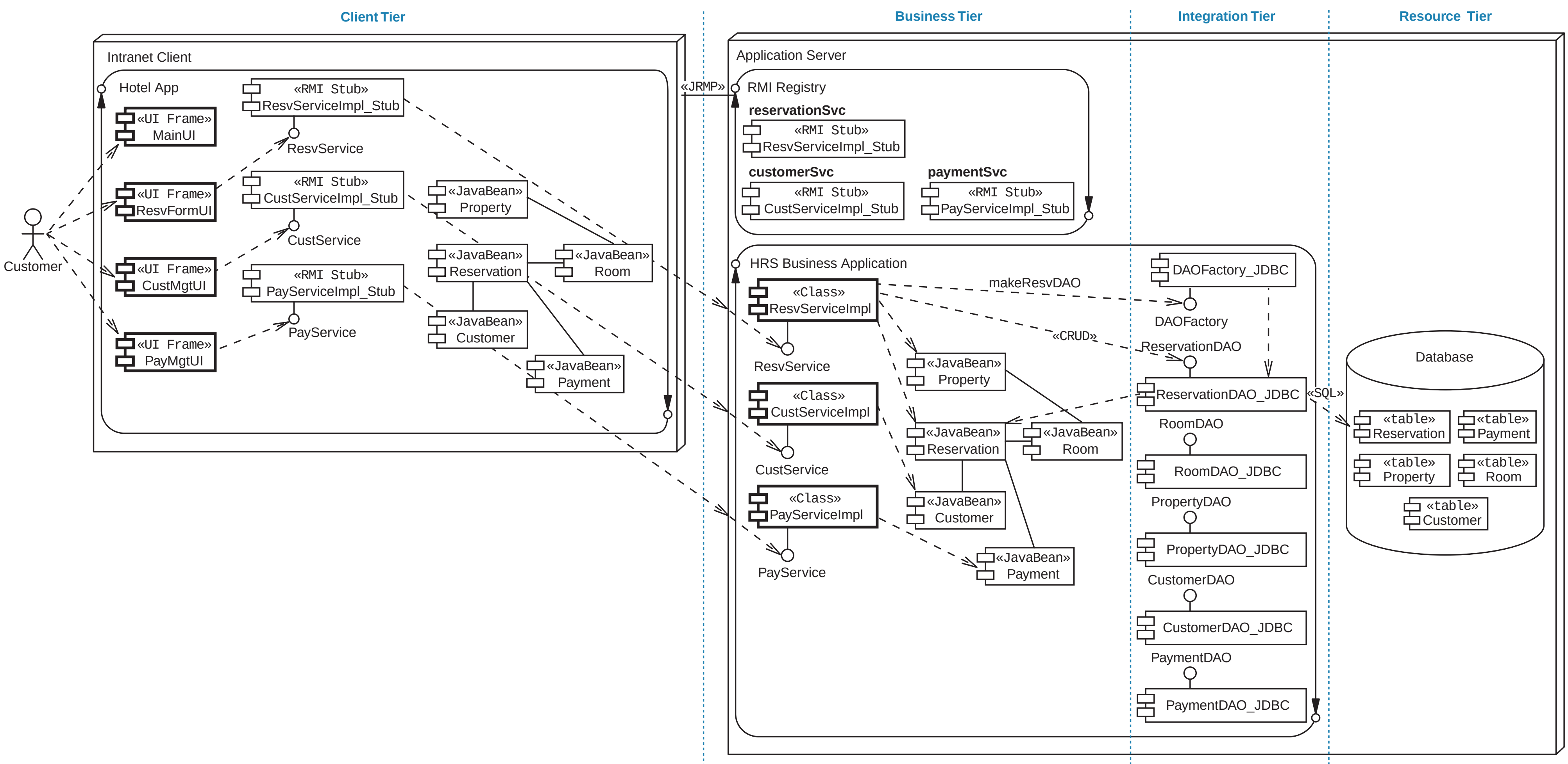
GUI applications use the standard set of design components.

Entity	Icon
Boundary	
Service	
Entity	

A Complete Design Model for the Create Reservation Use Case

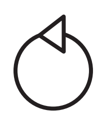
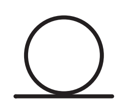


A Complete Solution Model for the Create Reservation Use Case

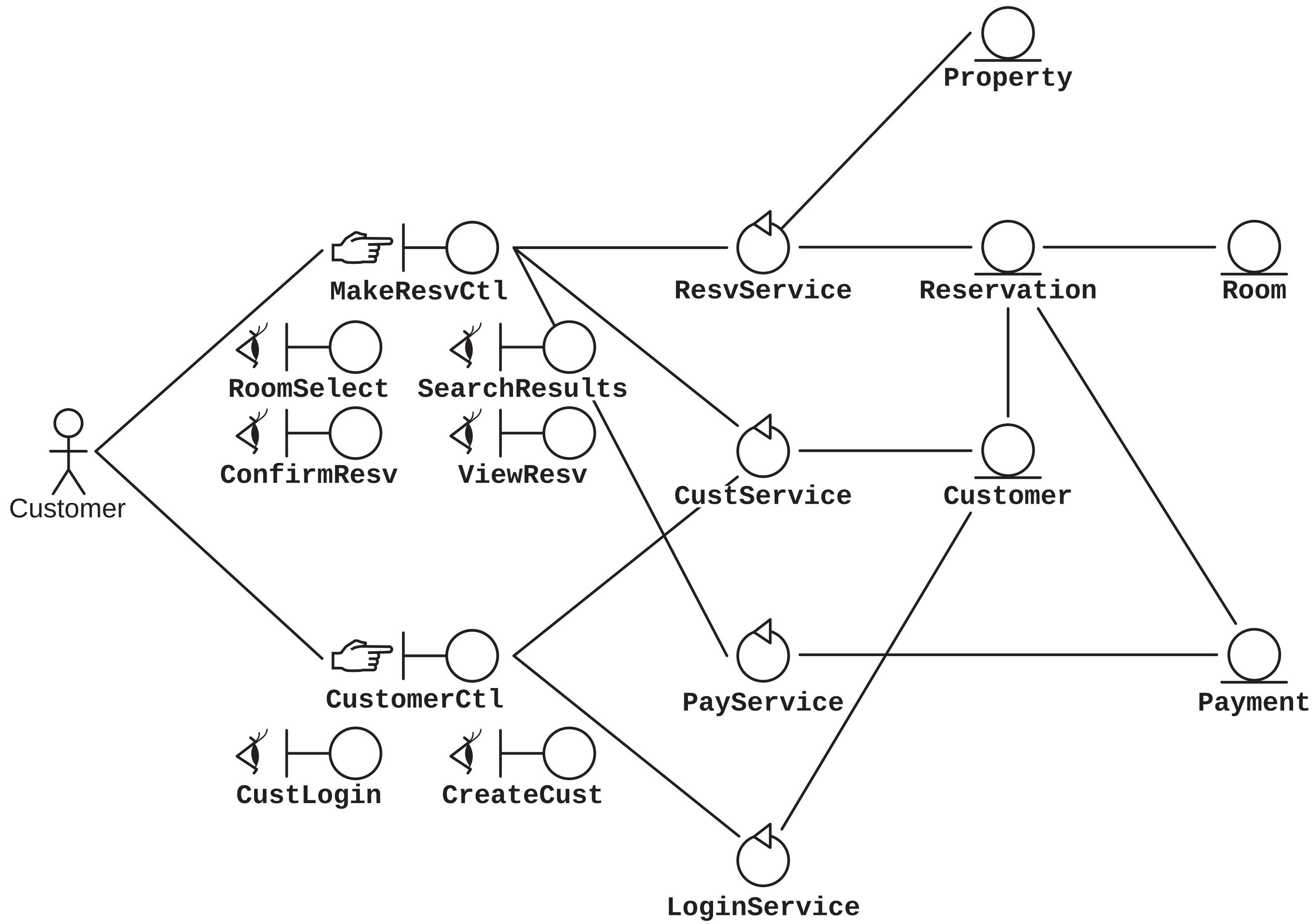


Creating the Solution Model for WebUI Applications

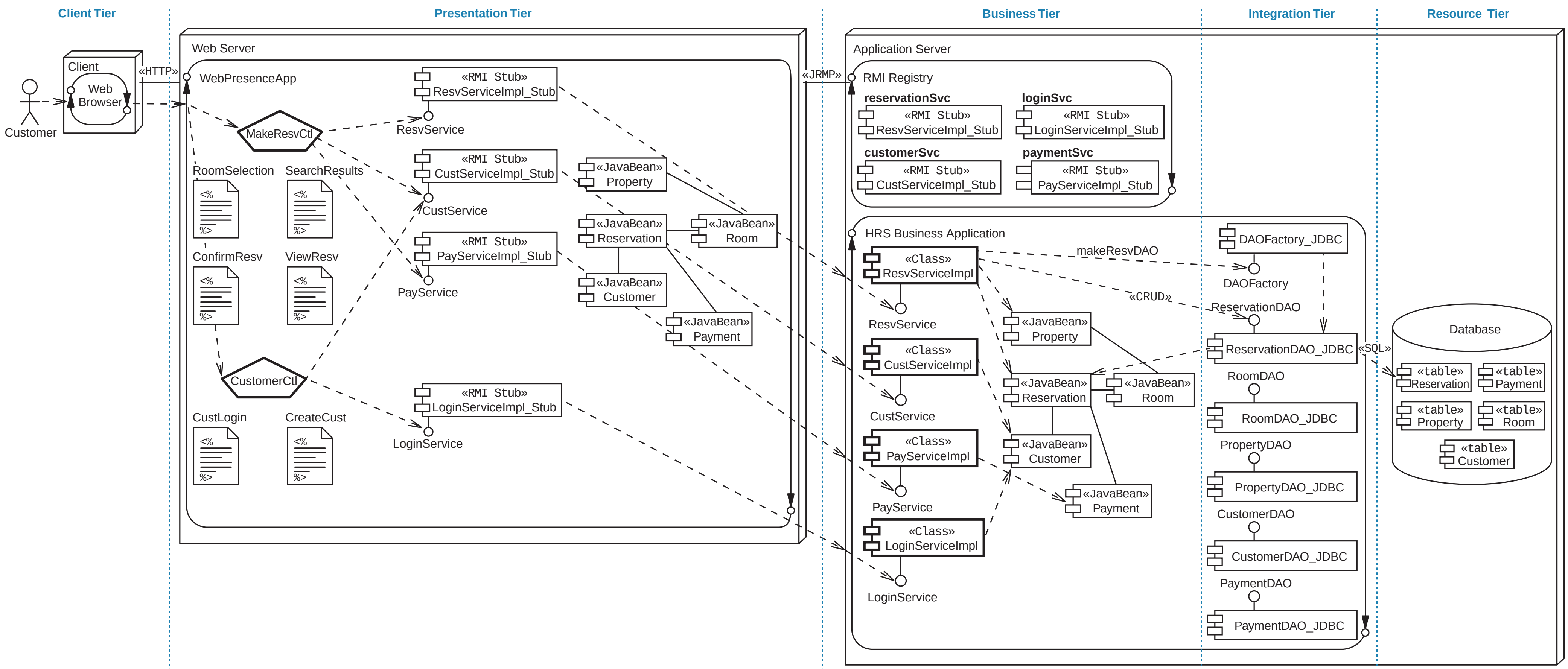
Web applications split the Boundary component into two separate components: Views and Controllers.

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 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).
- GUI applications use the standard set of Design components.
- WebUI applications use an alternate set of Design components, with the Boundary component split into Views and Controllers.

Module 17

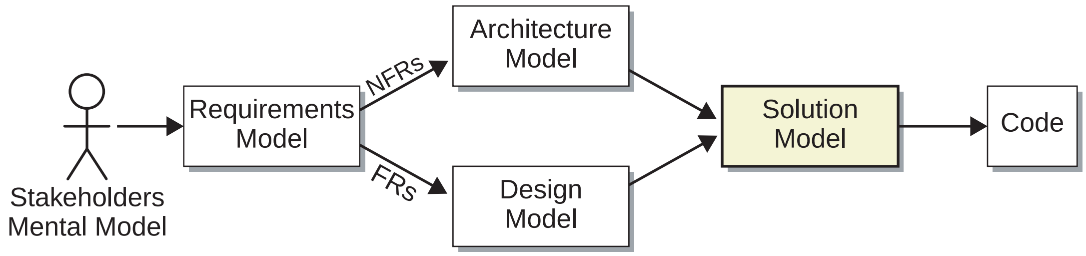
Refining the Domain Model

Objectives

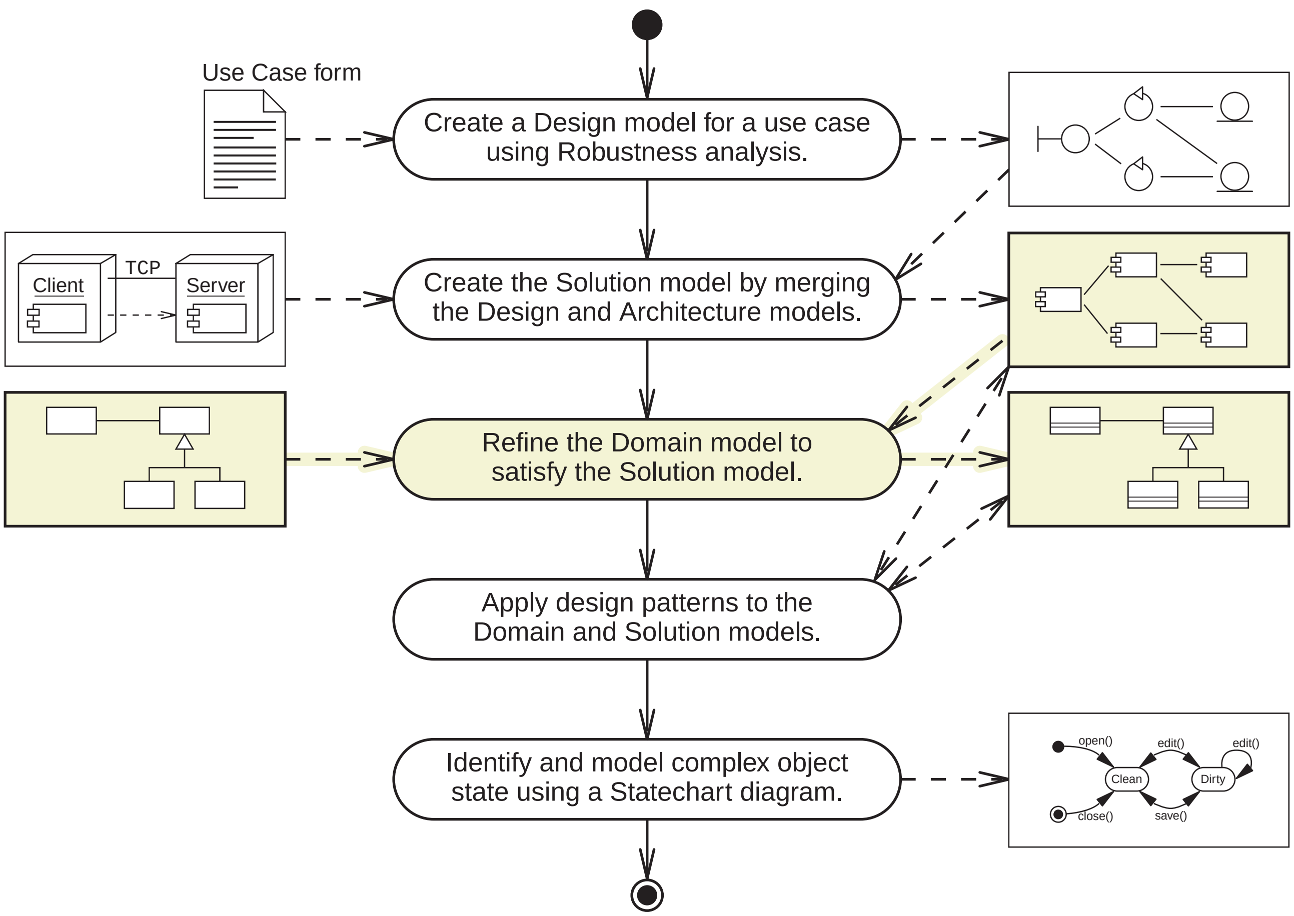
Upon completion of this module, you should be able to:

- 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

Process Map



Stakeholders Mental Model
Requirements Model
NFRs
FRs
Architecture Model
Design Model
Solution Model
Code



Use Case form
Create a Design model for a use case using Robustness analysis.
Client
TCP
Server
Create the Solution model by merging the Design and Architecture models.
Refine the Domain model to satisfy the Solution model.
Apply design patterns to the Domain and Solution models.
Identify and model complex object state using a Statechart diagram.
open()
edit()
edit()
Clean
Dirty
close()
save()

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

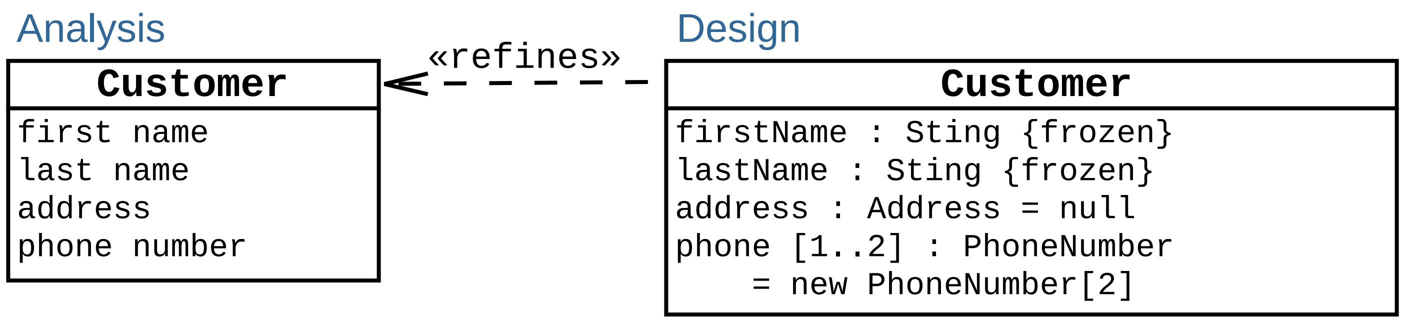
An attribute declaration in UML Class diagrams includes the following:

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

Refining the Attribute Metadata

Syntax:

`[visibility] name [multiplicity] [: type] [= init-value] [{property-string}]`



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

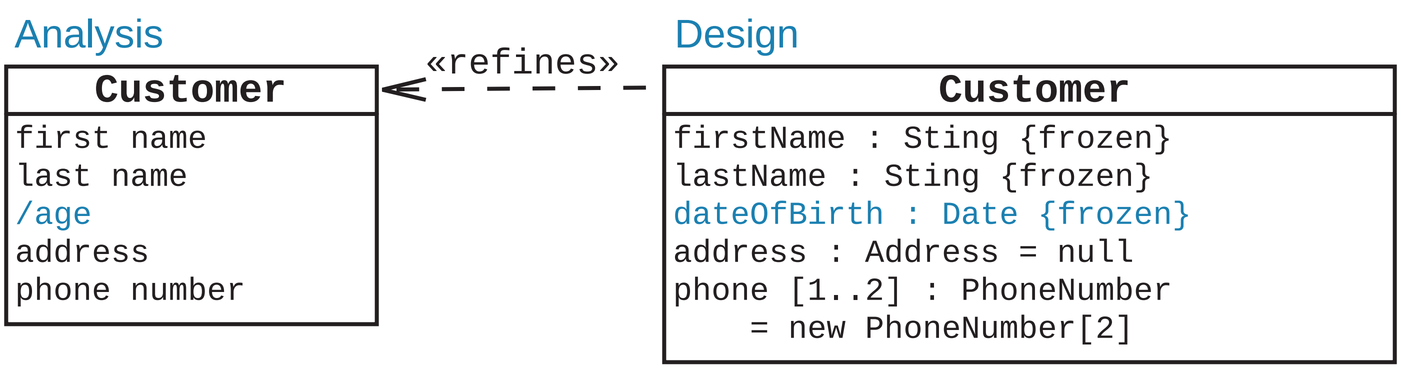
For a phone number attribute:

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 represenationally 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

```
Customer
-firstName : Sting {frozen}
-lastName : Sting {frozen}
-address : Address = new Address()
-phone [1..2] : PhoneNumber = null

«accessors»
+getFirstName() : String
+getLastName() : String
+getAddress() : Address
+getPhoneNumber(:int) :PhoneNumber)

«mutators»
+setAddress(:Address)
+setPhoneNumber(:int, :PhoneNumber)
```

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:

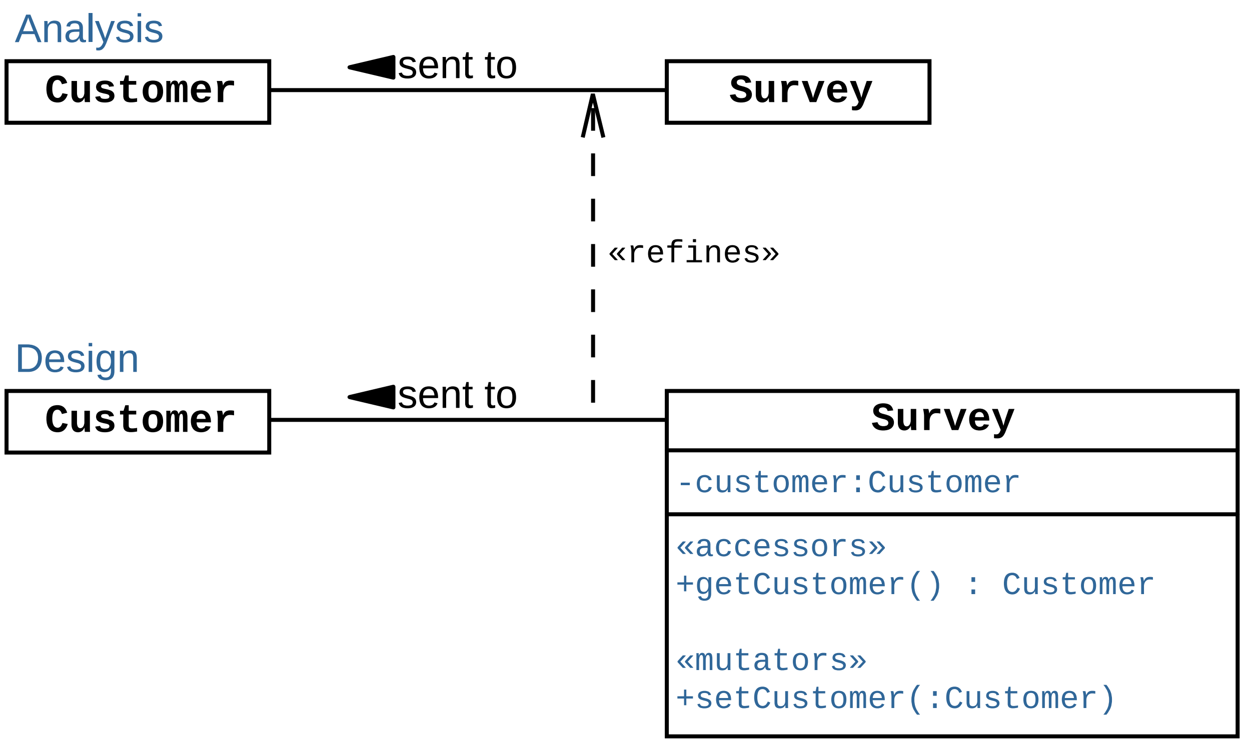
- Association
- Aggregation
- Composition

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.

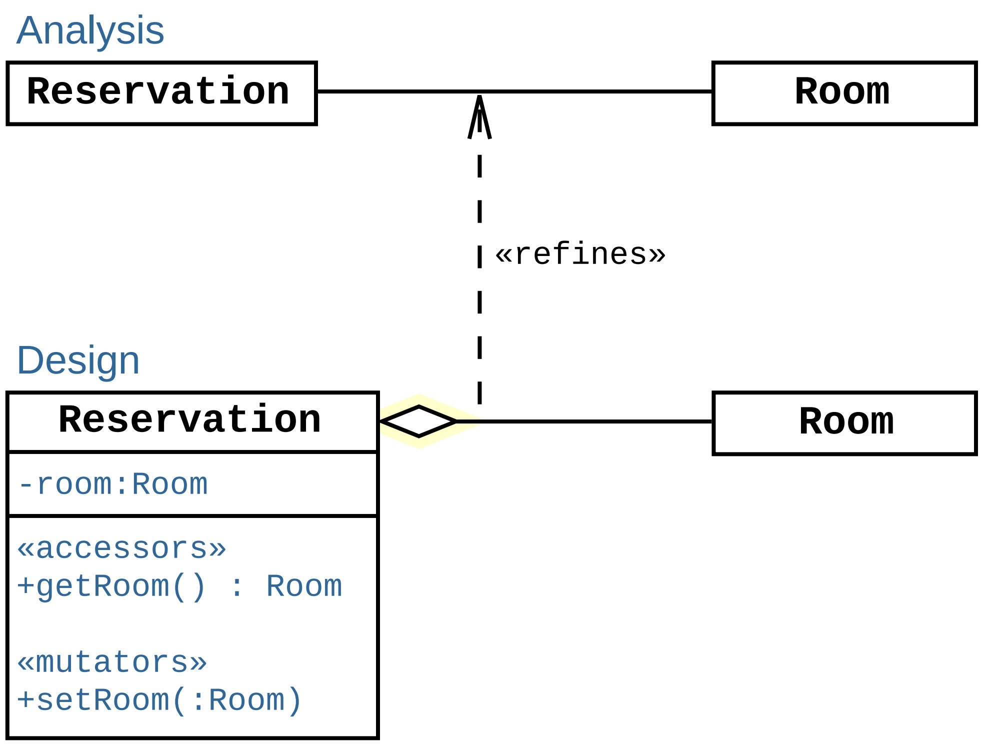
Association

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



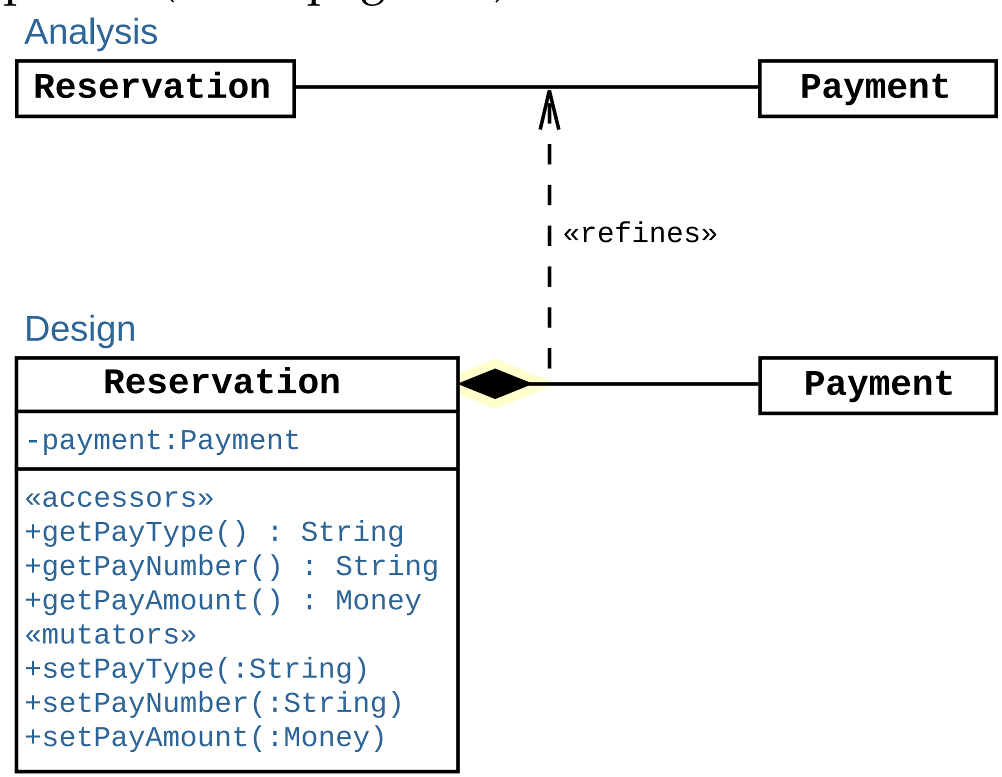
Aggregation

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



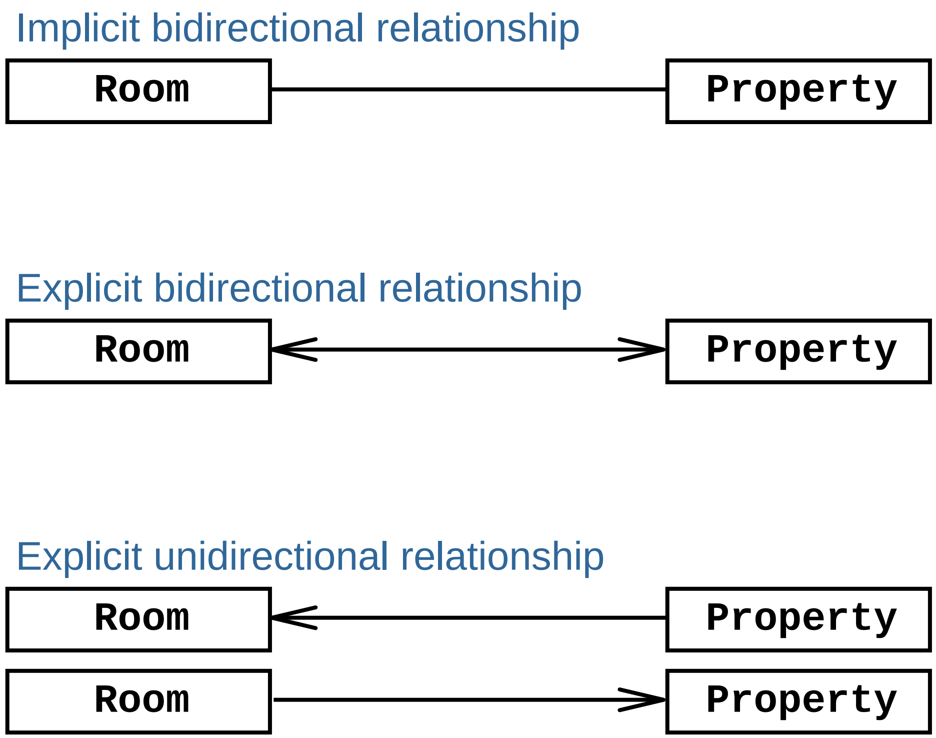
Composition

“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)



Navigation

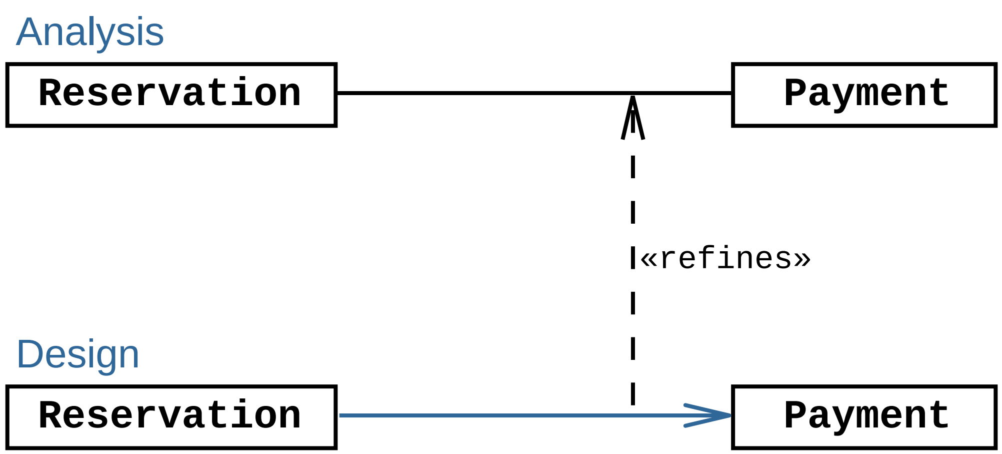
A navigation arrow shows the direction of object traversal at runtime.



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.

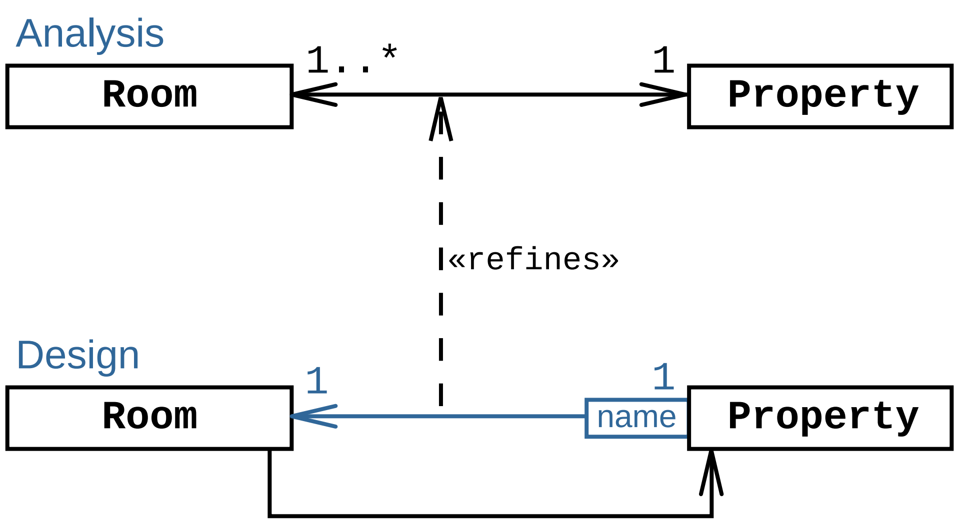
For example:



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.

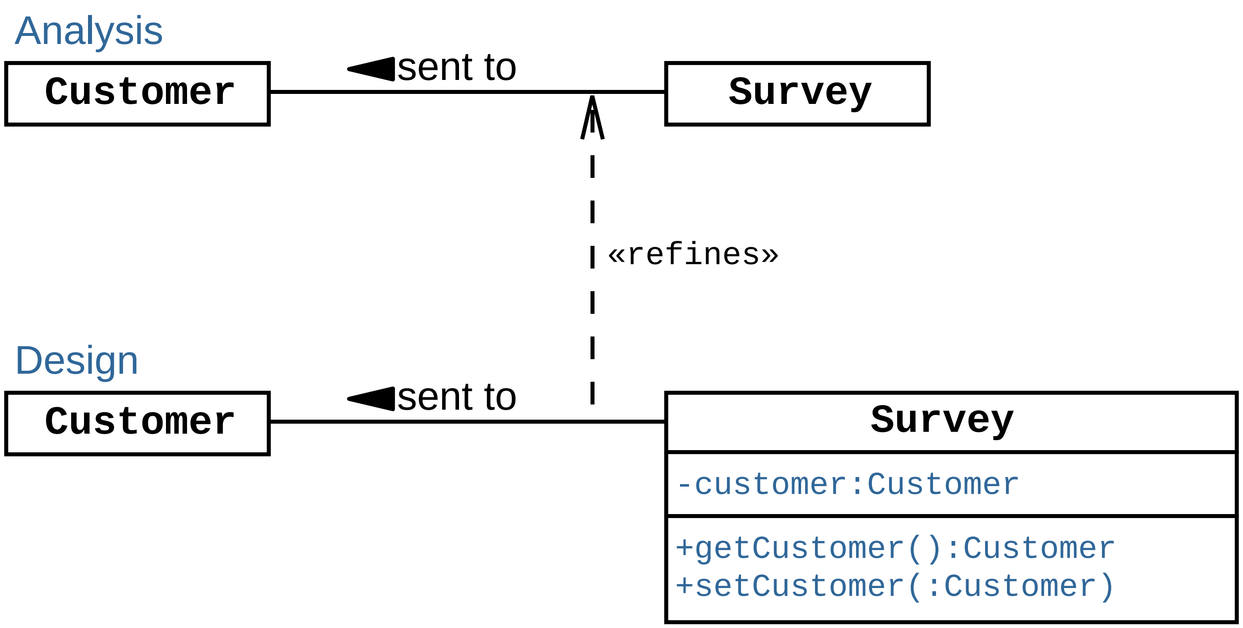
For example:



Relationship Methods

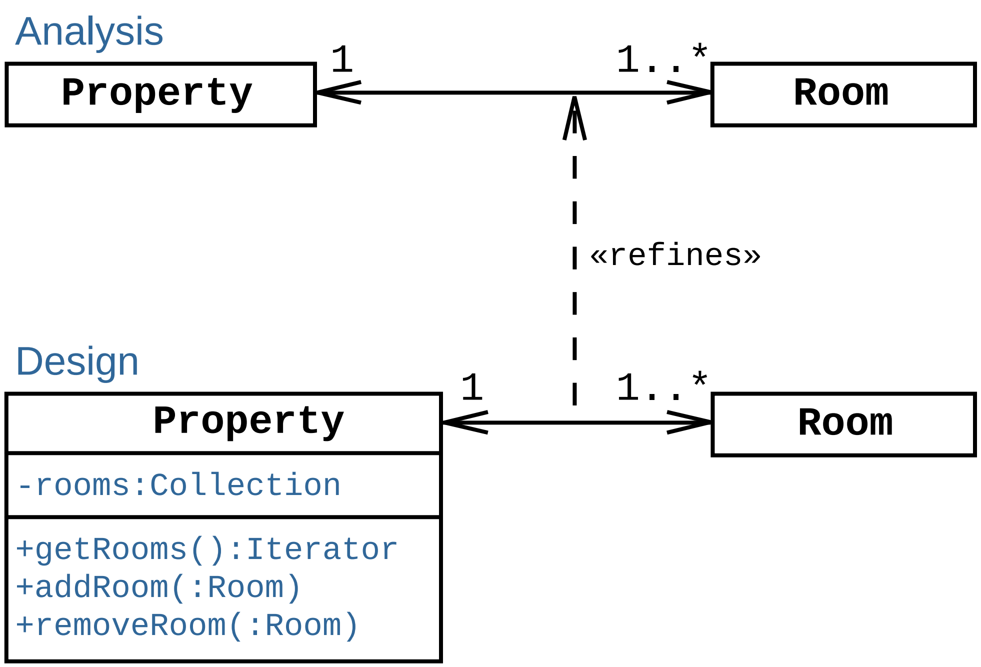
Association methods enable the client to access and change associated objects. There are three cases: one-to-one, one-to-many, and many-to-many.

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



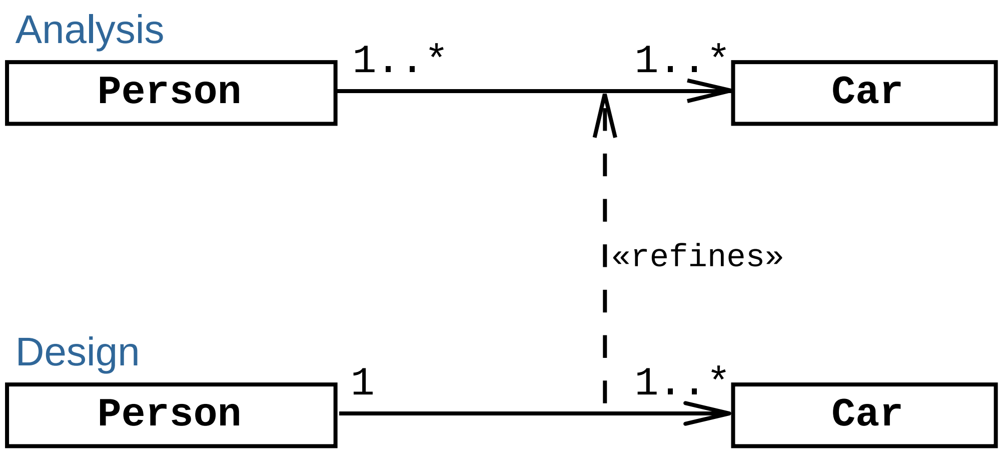
Relationship Methods

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



Resolving Many-to-Many Relationships

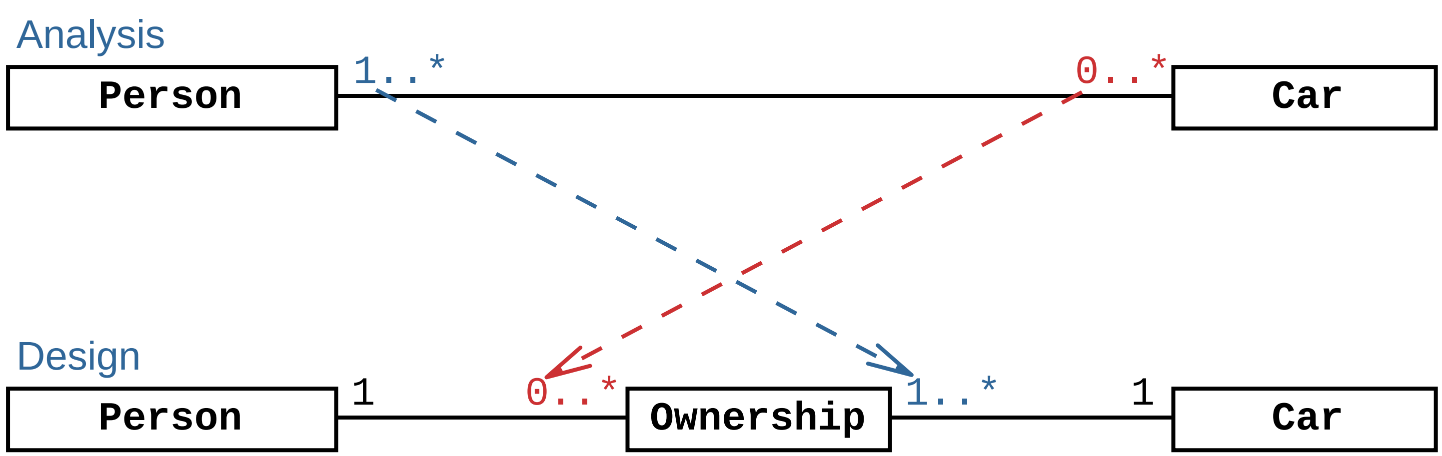
Managing many-to-many associations is challenging. Consider dropping this requirement at design-time.



Resolving Many-to-Many Associations

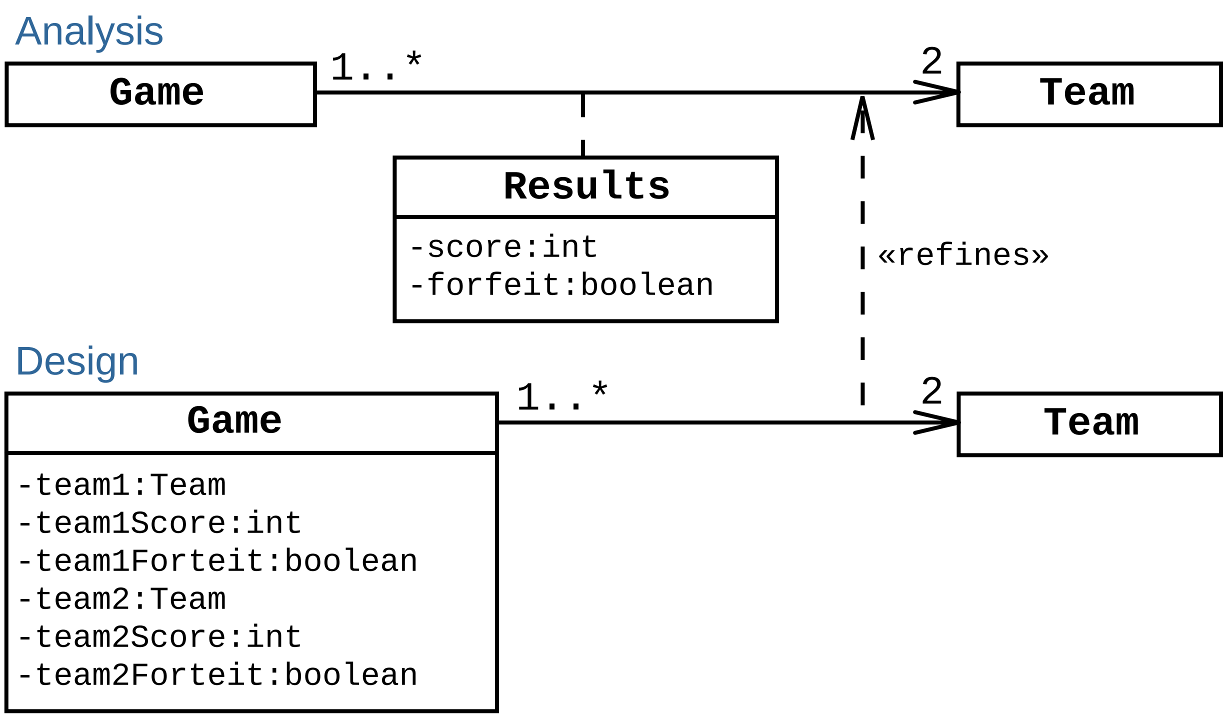
If the many-to-many association must be preserved, you can sometimes add a class in between that reduces the single many-to-many association to two one-to-many associations.

For example:



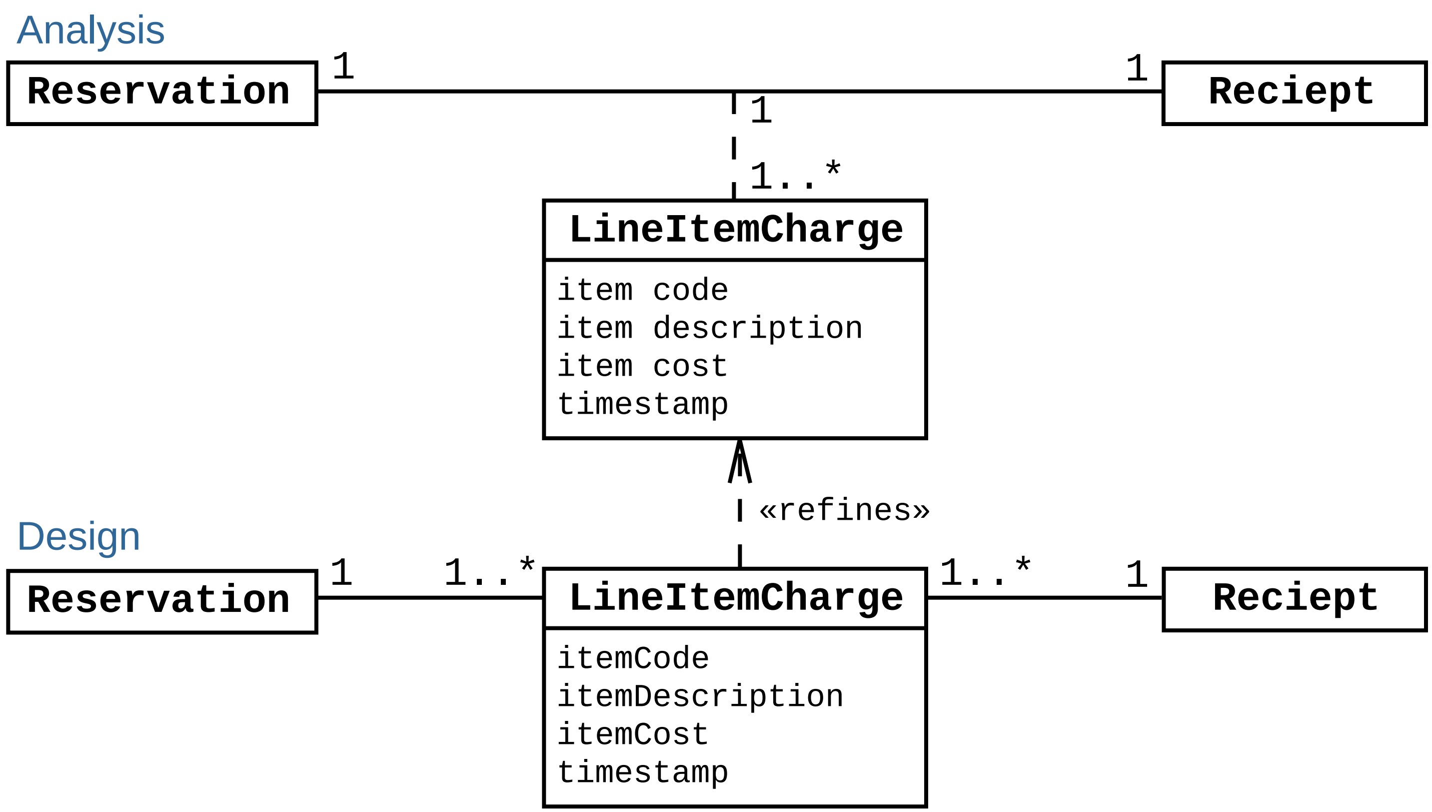
Resolving Association Classes

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



Resolving Association Classes

For many-to-many, 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

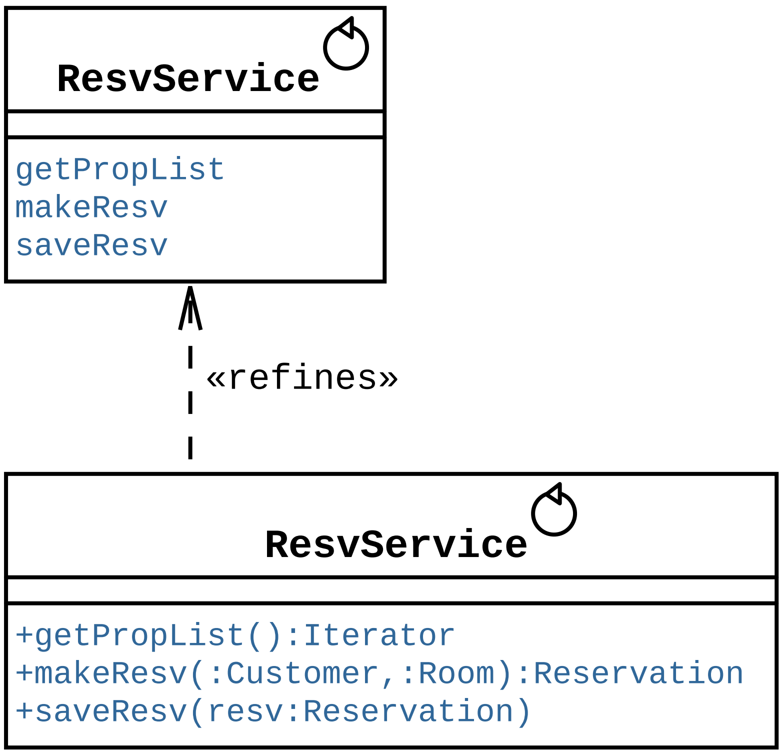
Other types of methods:

- Object management
- Unit testing
- Recovery and inverse operations

Refining Methods

Syntax:

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



Declaring Constructors

Constructors initialize an object and a syntax similar to methods:

```
Customer
-----------------------------------------------
-firstName : String {frozen}
-lastName : String {frozen}
-address : Address = new Address()
-phone [1..2] : PhoneNumber = null
-----------------------------------------------
«constructors»
+Customer(fName:String,1Name:String)
+Customer(fName:String,1Name:String
         addr:Address)

«accessors»
+getFirstName() : String
+getLastName() : String
+getAddress() : Address
+getPhoneNumber(:int) : PhoneNumber

«mutators»
+setAddress(:Address)
+setPhoneNumber(:int, :PhoneNumber)
```

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, and constructors.

Module 18

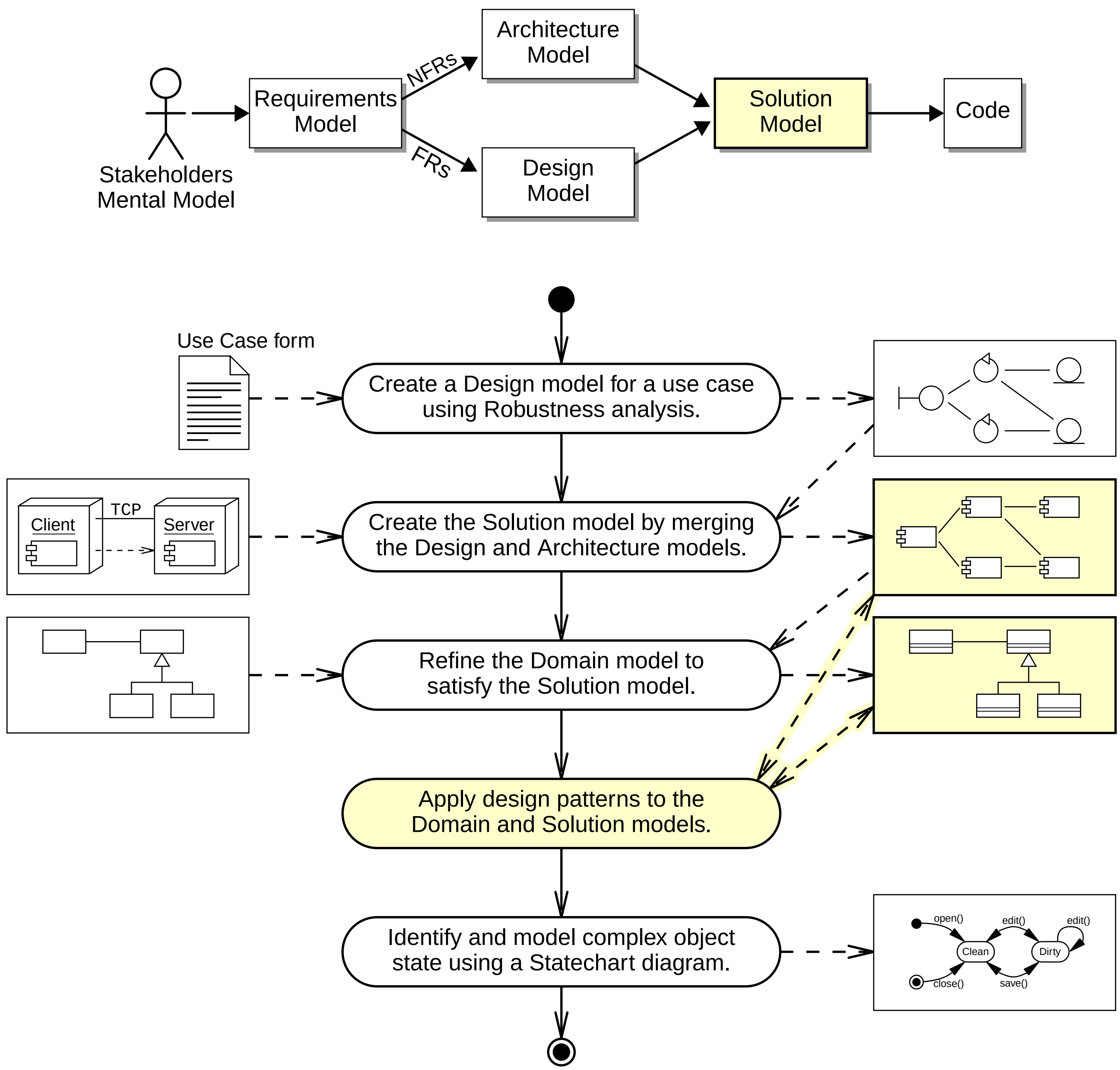
Applying Design Patterns to the Solution Model

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

Process Map


Requirements Model
NFRs
FRs
Architecture Model
Design Model
Solution Model
Code
Stakeholders Mental Model
Use Case form
Create a Design model for a use case using Robustness analysis.
Client
TCP
Server
Create the Solution model by merging the Design and Architecture models.
Refine the Domain model to satisfy the Solution model.
Apply design patterns to the Domain and Solution models.
Identify and model complex object state using a Statechart diagram.
open()
edit()
edit()
Clean
Dirty
close()
save()

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

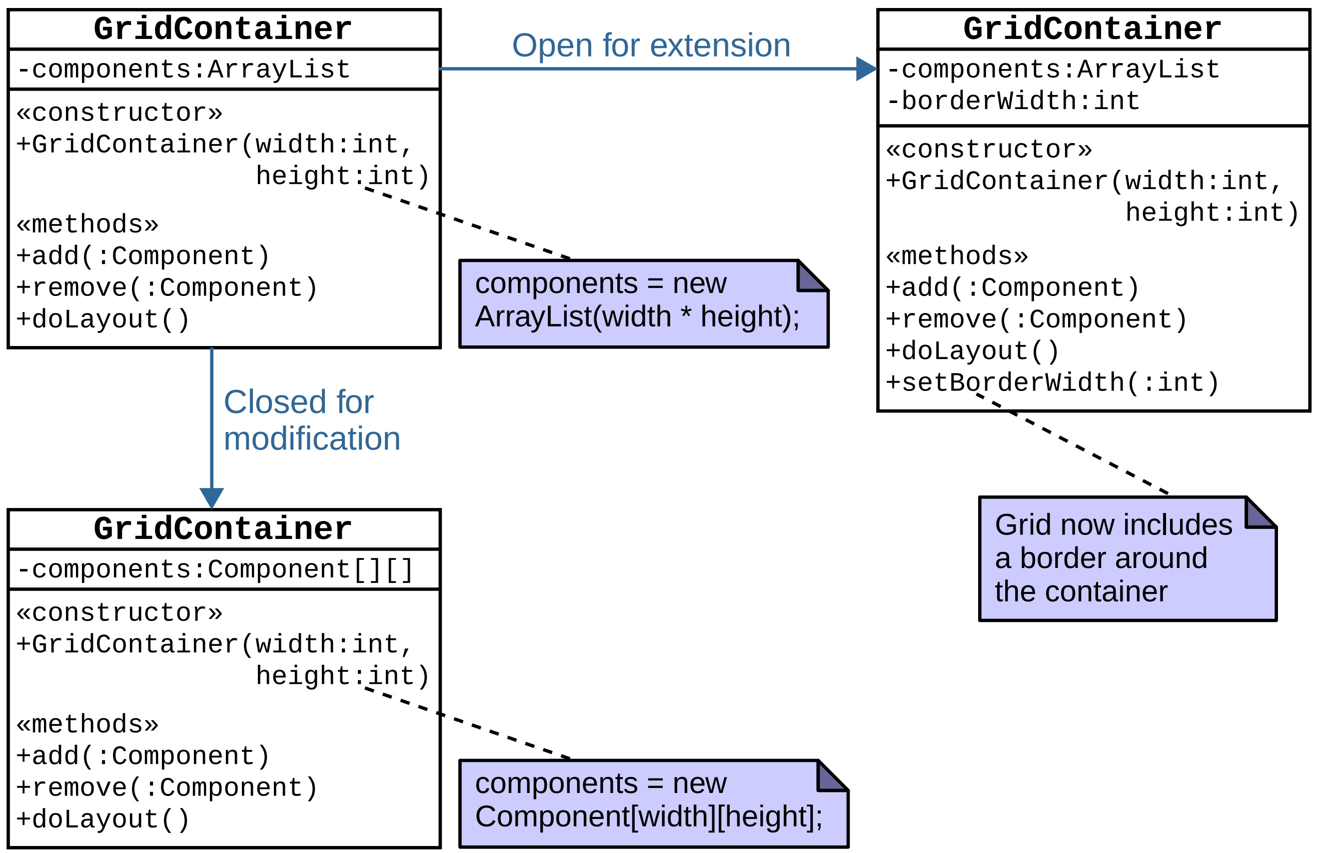
GridContainer
-components:ArrayList
«constructor» +GridContainer(width:int, height:int) «methods» +add(:Component) +remove(:Component) +doLayout()

```
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	.	=	/

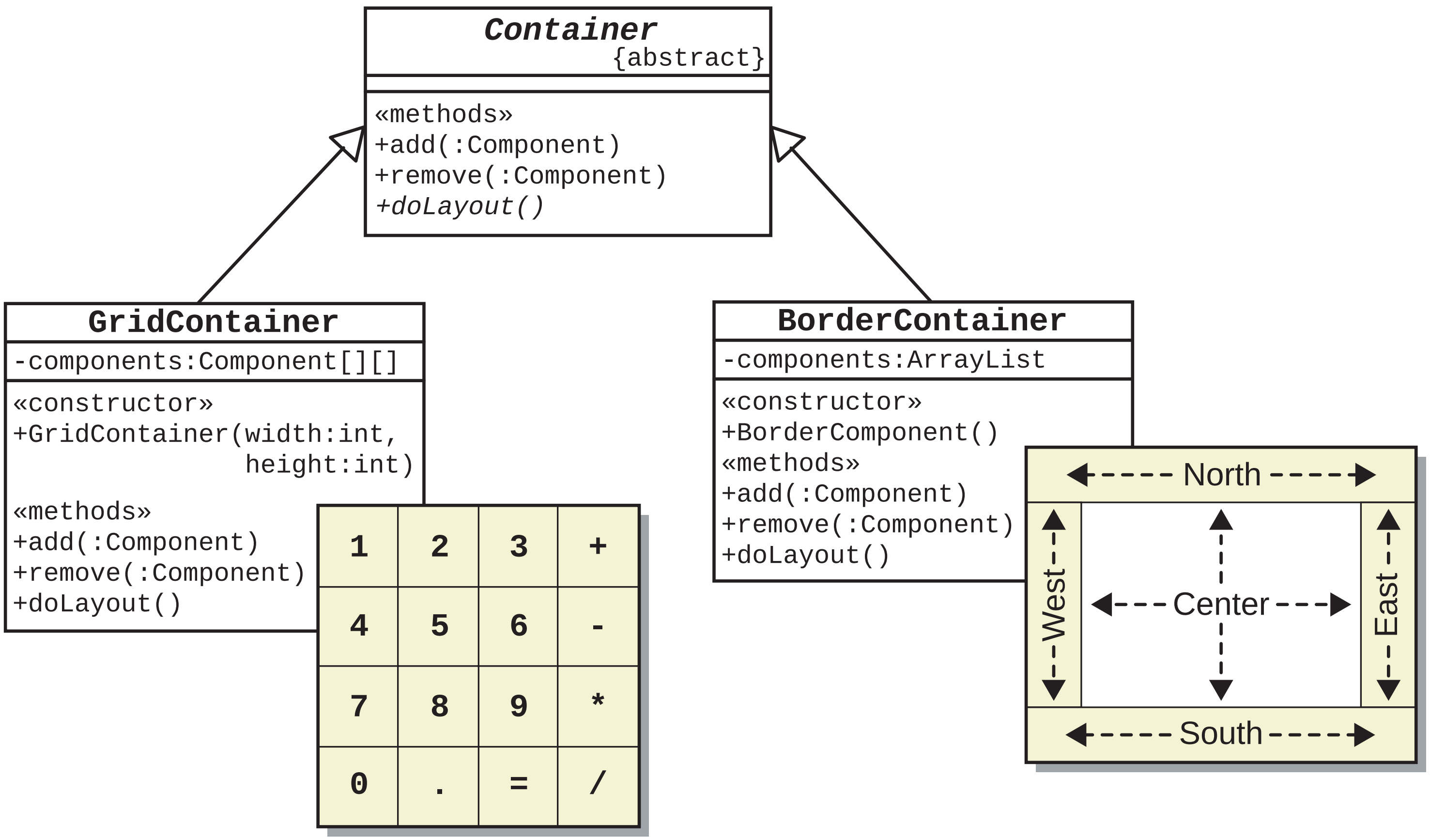
Open Closed Principle

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



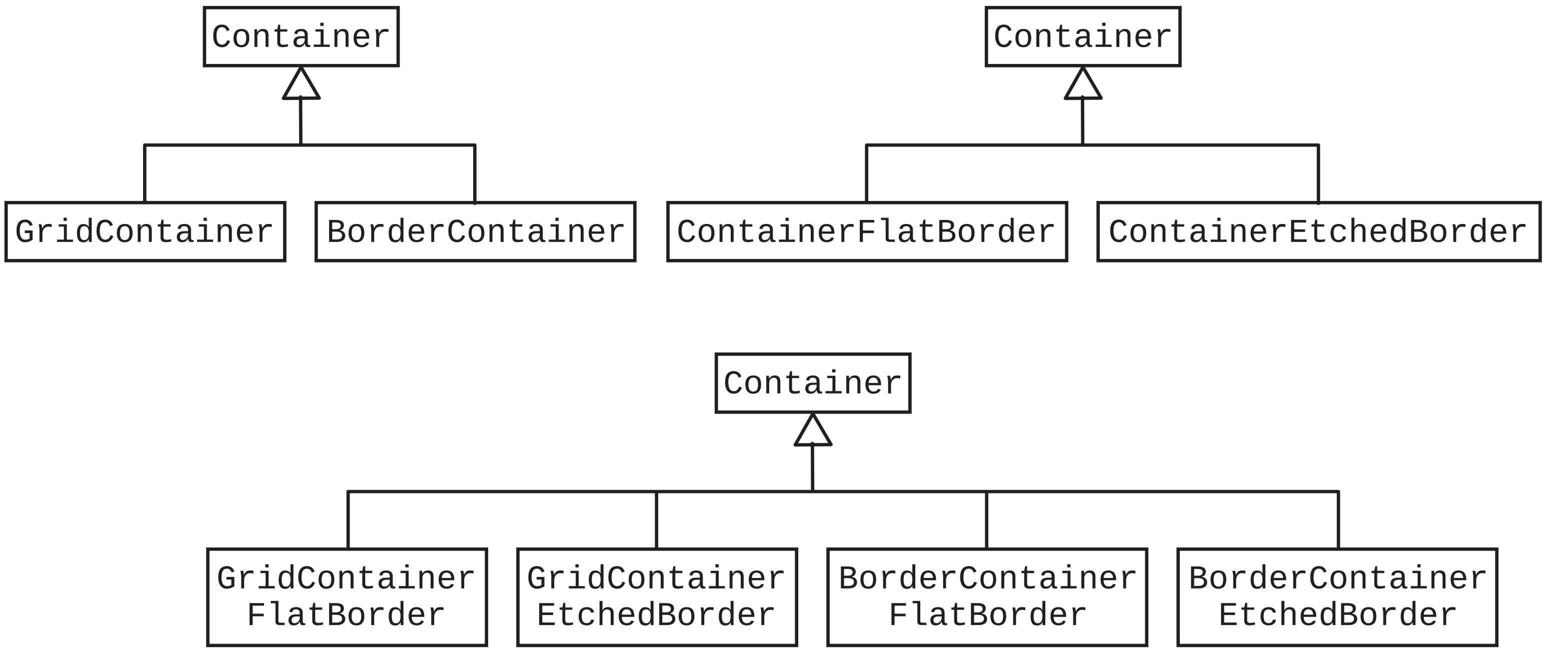
Composite Reuse Principle

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



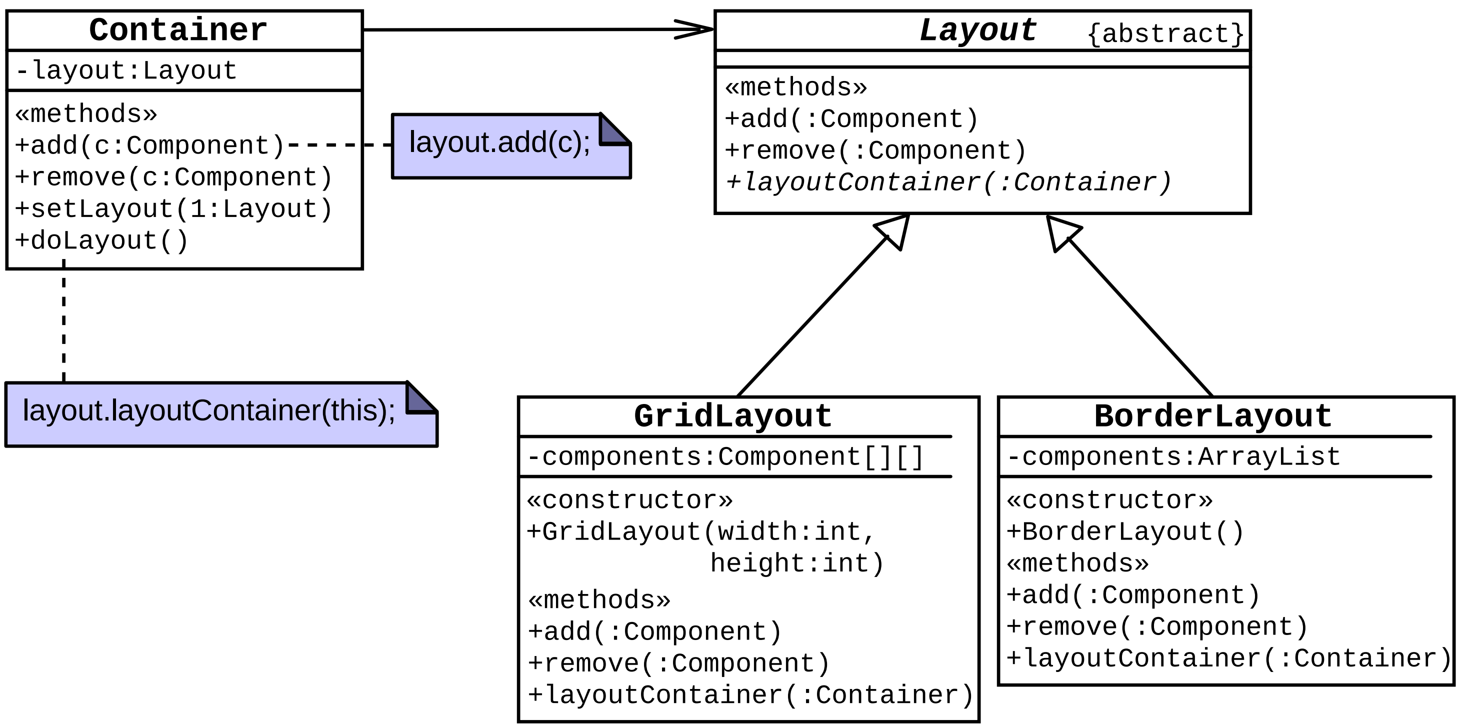
Composite Reuse Principle

Excessive inheritance leads to brittle and large hierarchies:



Composite Reuse Principle

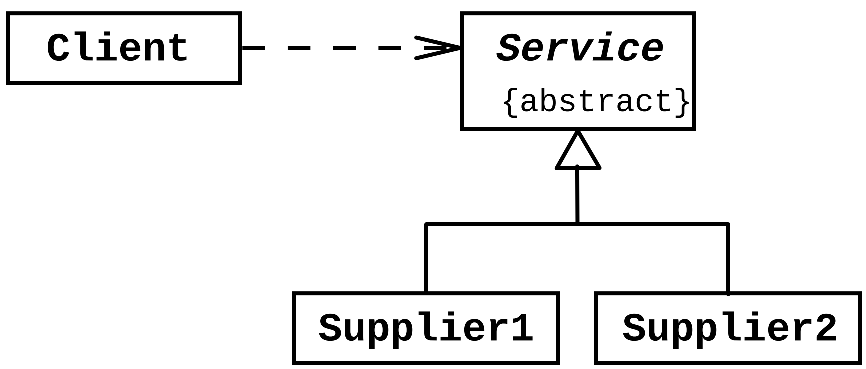
CRP leads to flexible reuse through delegation:



Dependency Inversion Principle

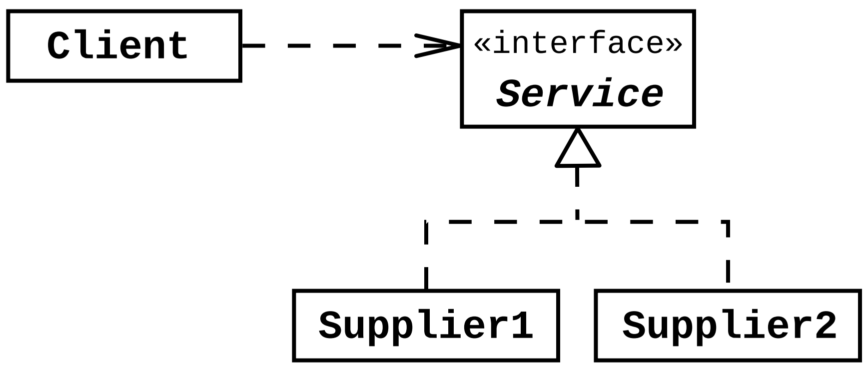
"Depend on abstractions. Do not depend on concretions." (Knoernschild page 12)

An abstraction can be an abstract class:



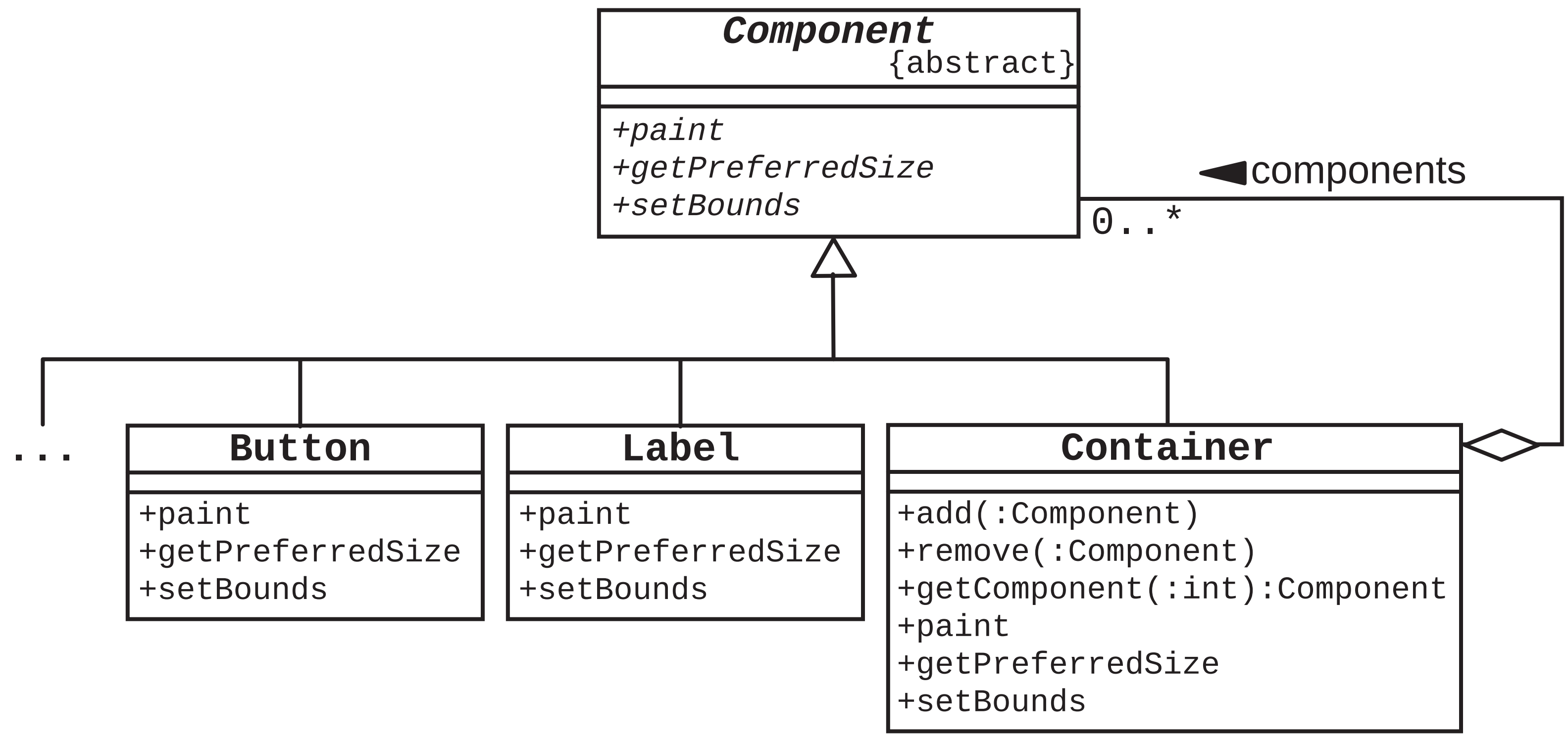
Dependency Inversion Principle

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:

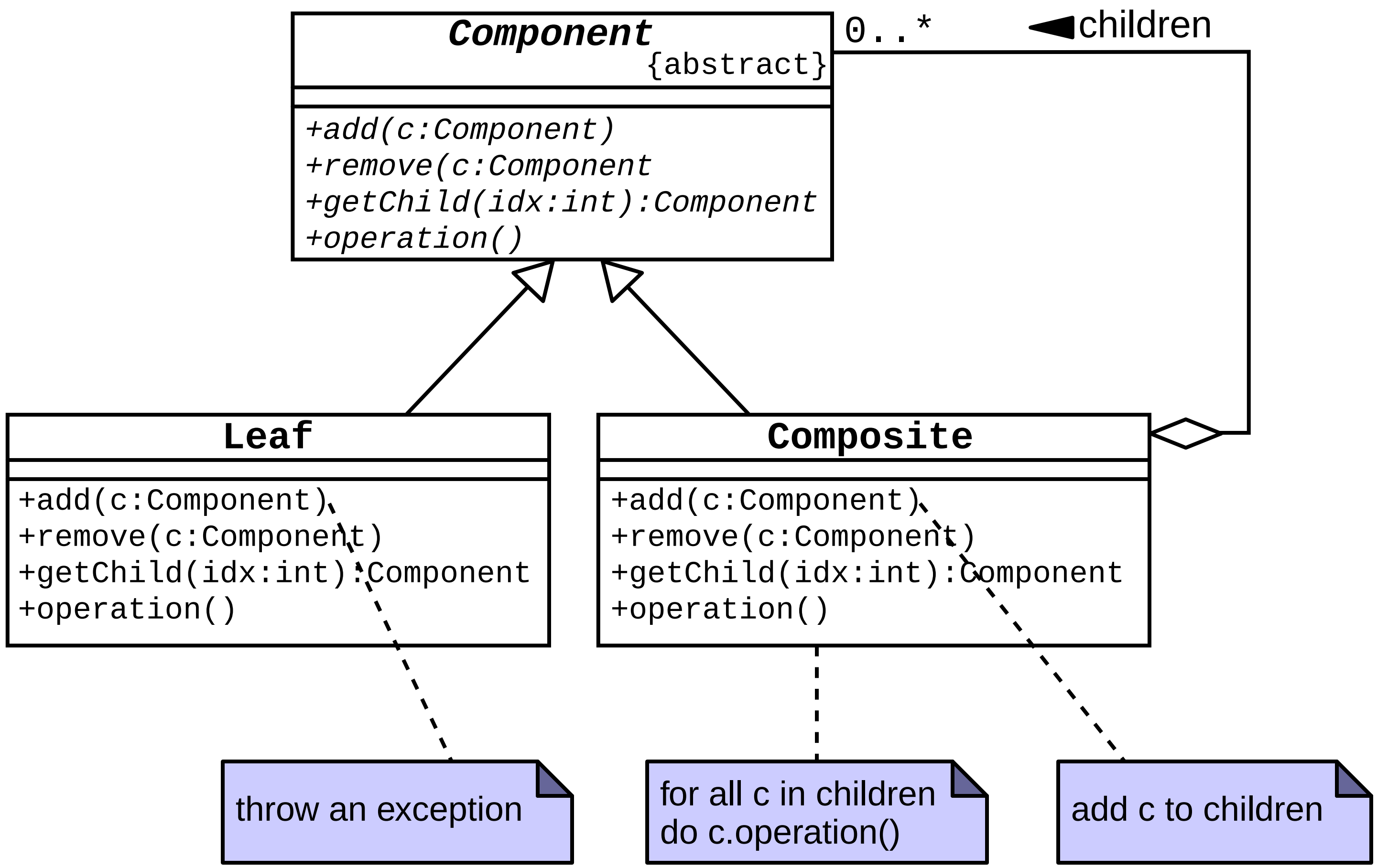
- You want to represent whole-part hierarchies of objects
- You want to use the same interface on the assemblies and the components in an assembly

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.

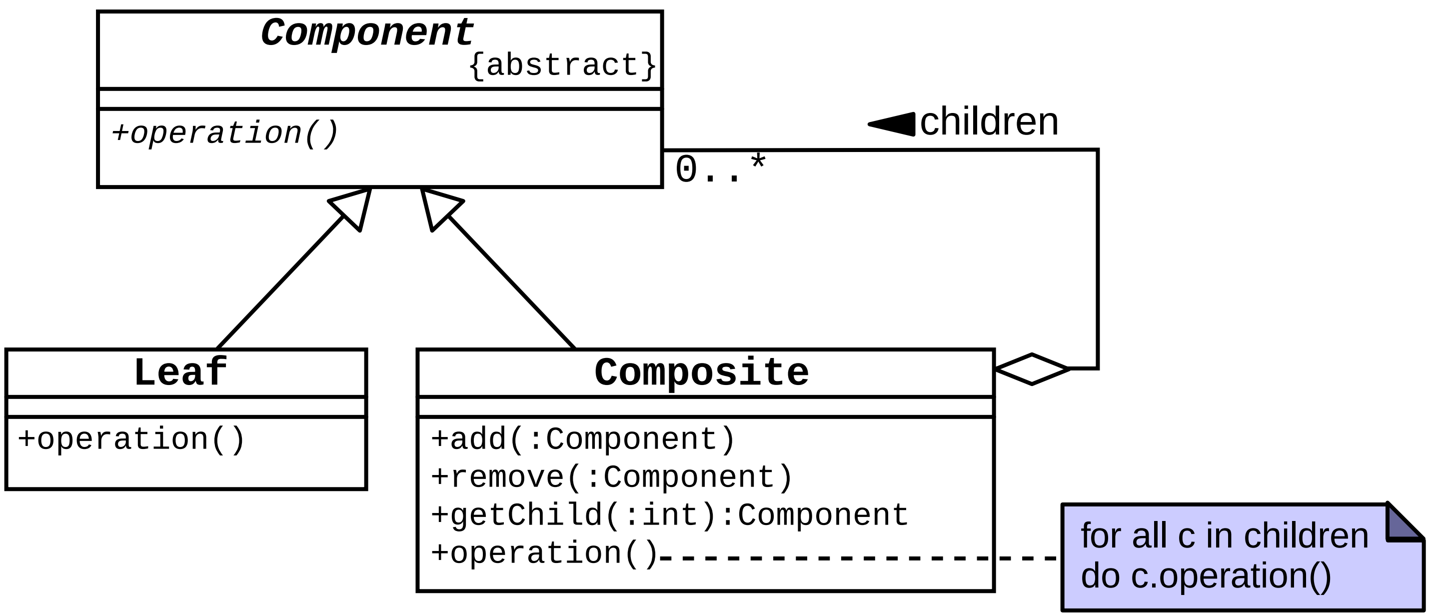
Describing the Composite Pattern

Composite (GoF) Model:



Describing the Composite Pattern

Alternate Model:



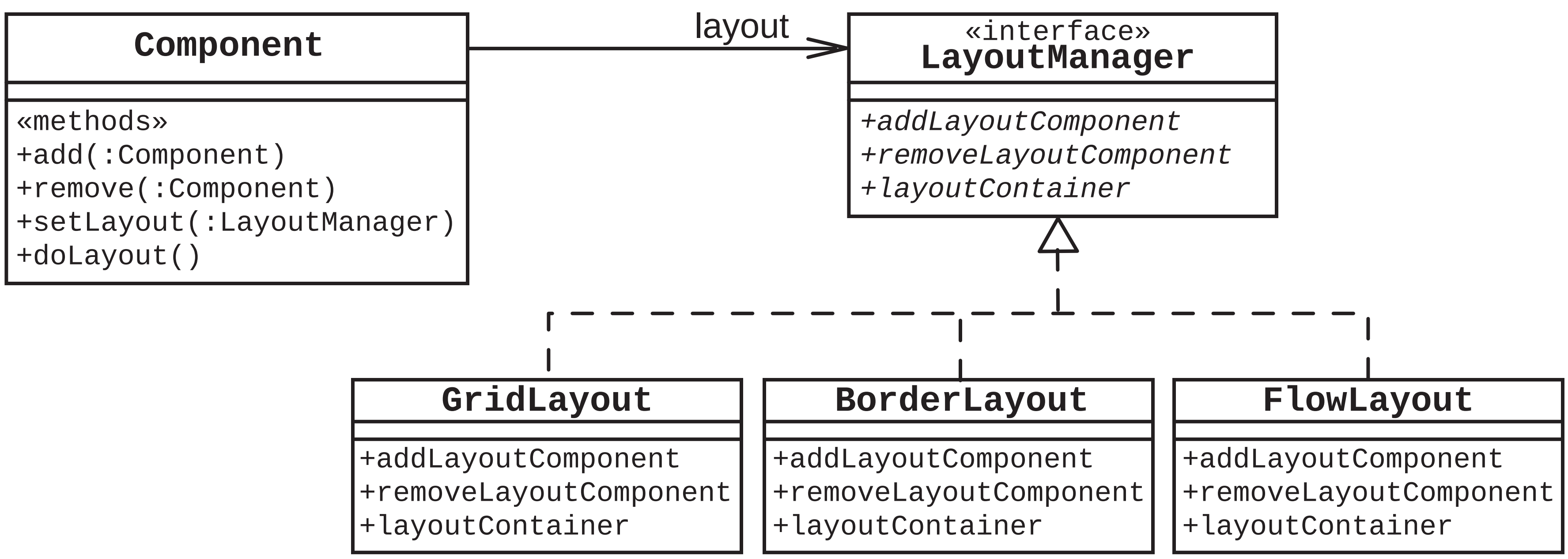
Describing the Composite Pattern

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:

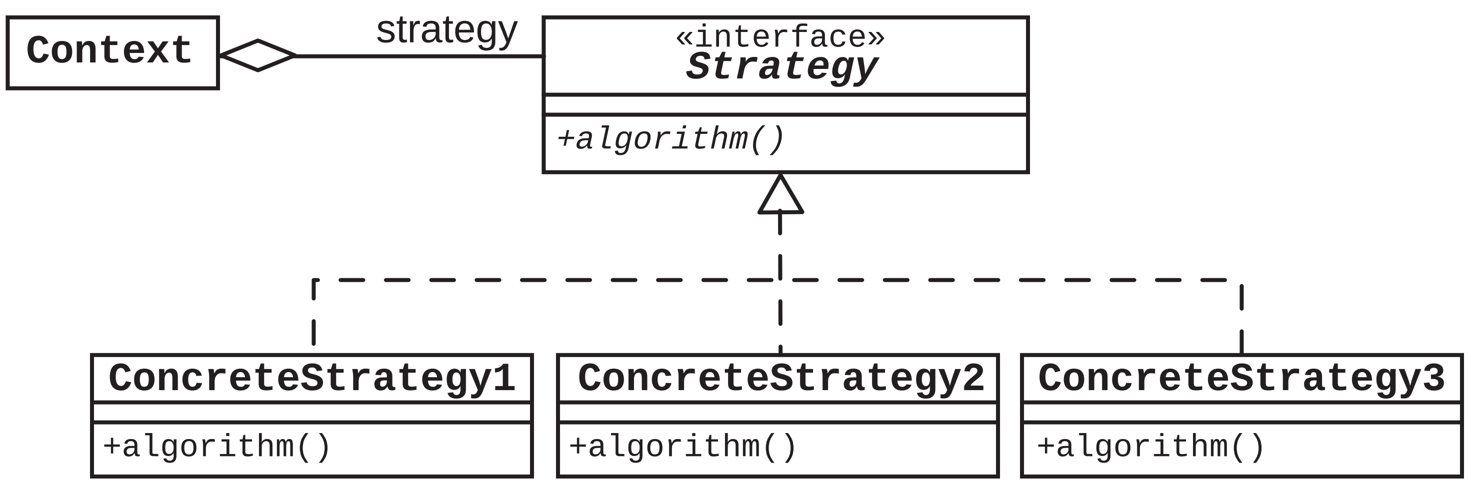
- You have a set of classes that are only different in the algorithms that they use
- You want to change algorithms at runtime

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:



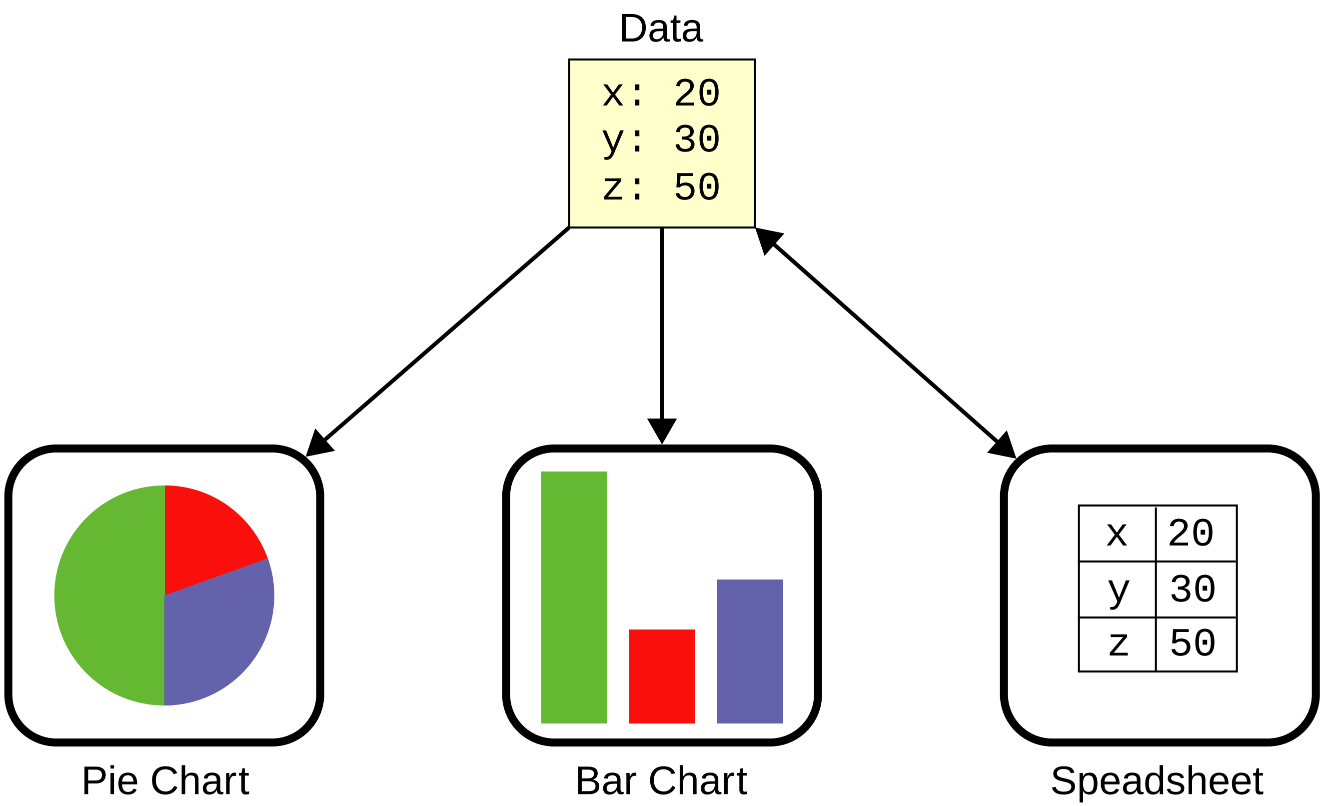
Describing the Strategy Pattern

Consequences:

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

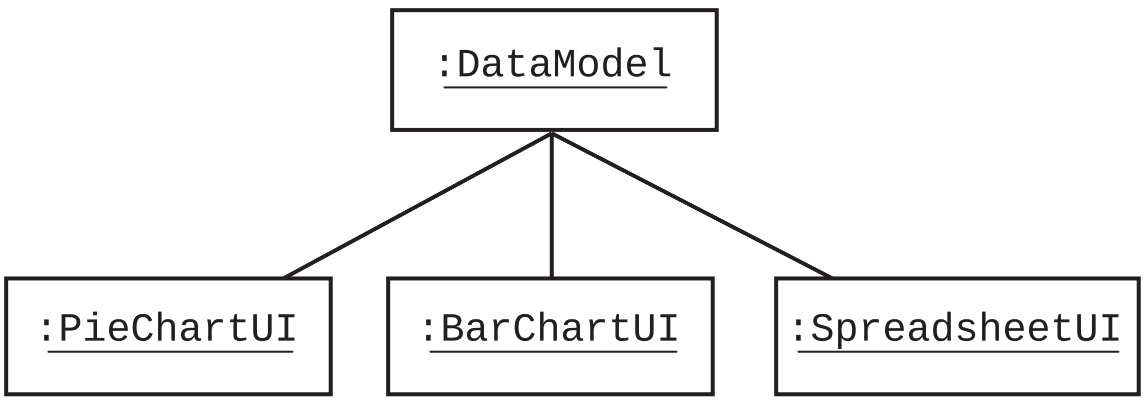
Describing the Observer Pattern

"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)

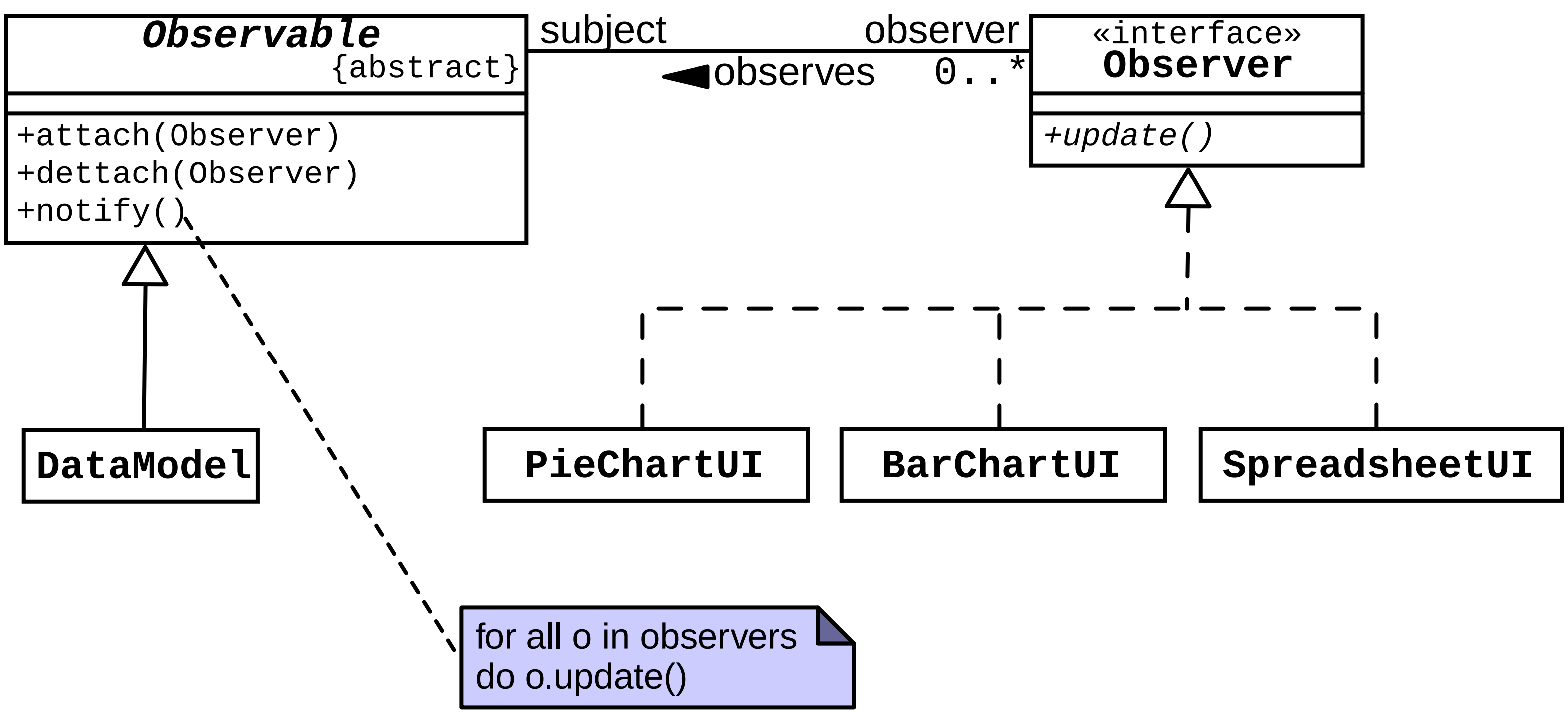


Describing the Observer Pattern

- 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:

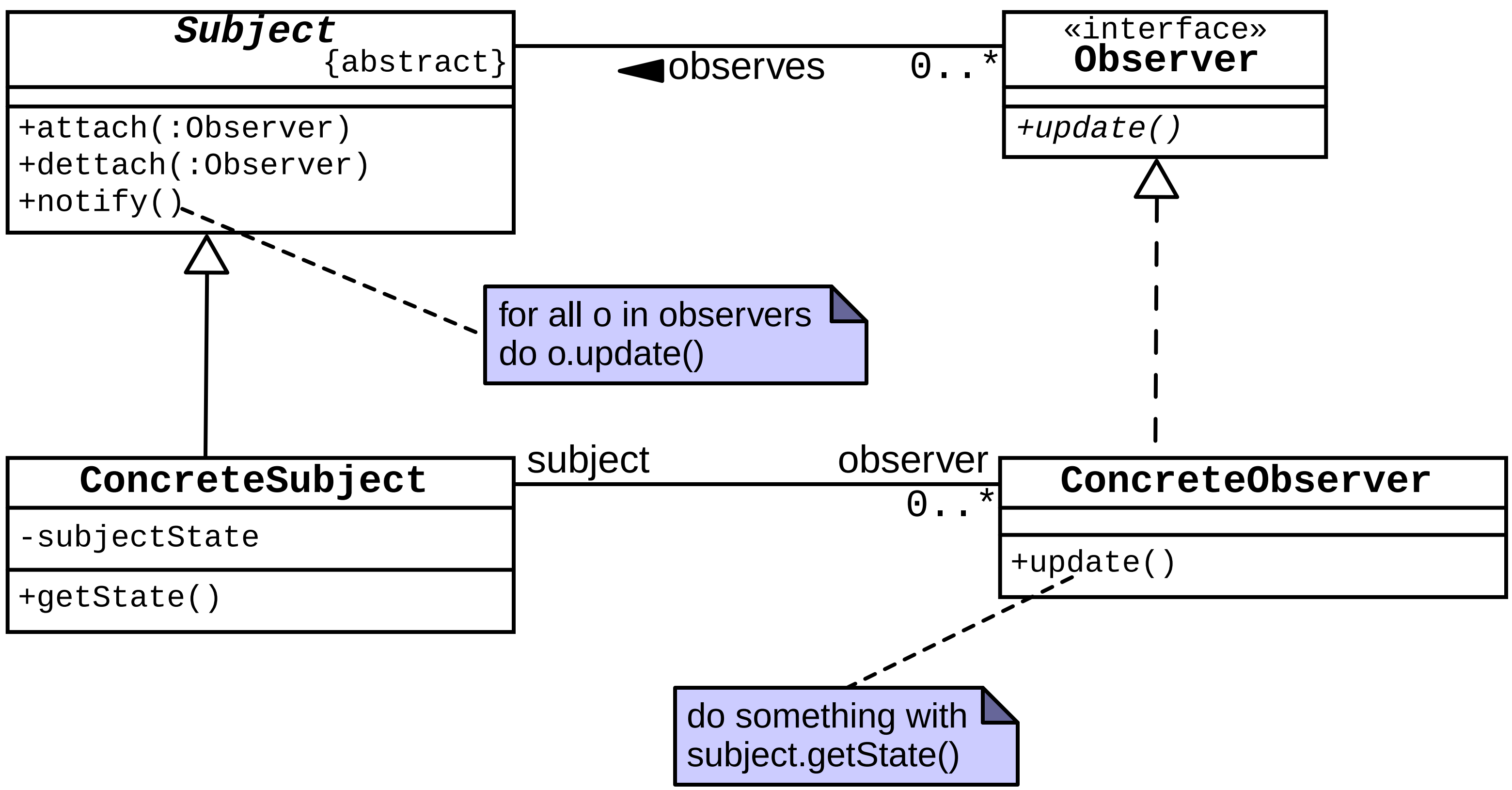
- You need to notify a set of objects that an event has occurred.
- The set of observing objects can change at runtime.

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:



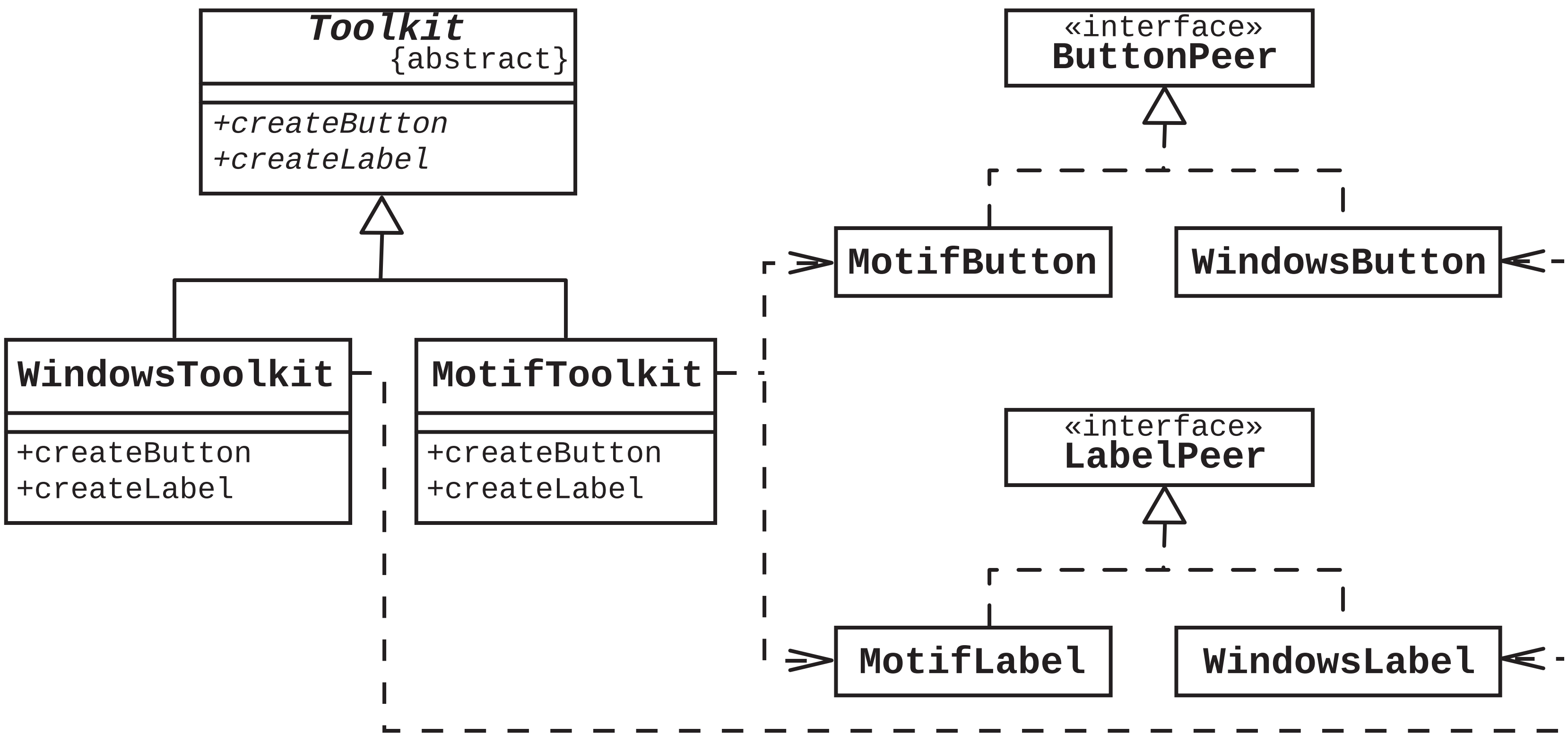
Describing the Observer Pattern

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:

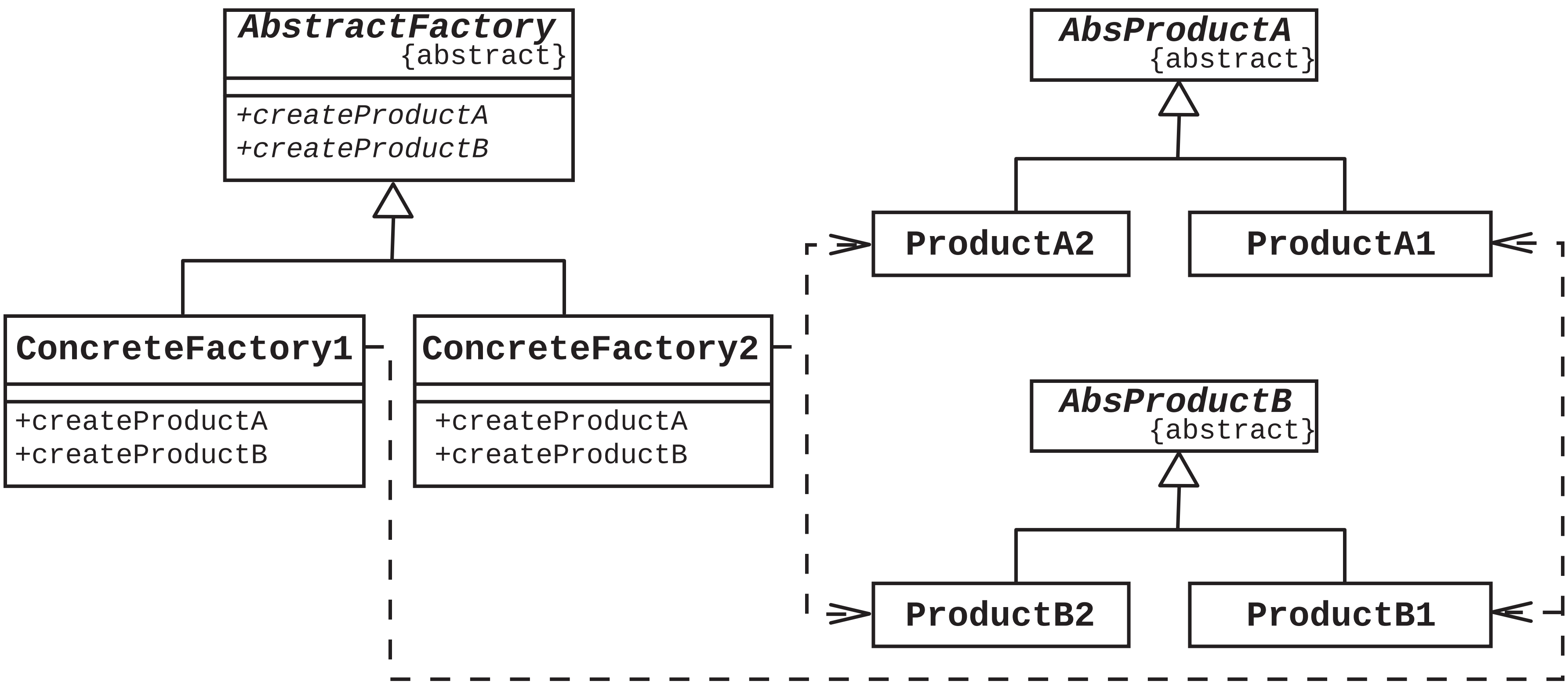
- A system has multiple families of products.
- Each product family is designed to be used together.
- You do not want to reveal the implementation classes of the product families.

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

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 19

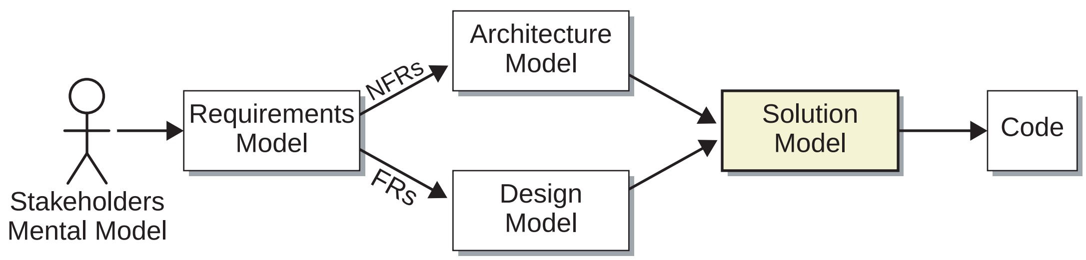
Modeling Complex Object State Using Statechart Diagrams

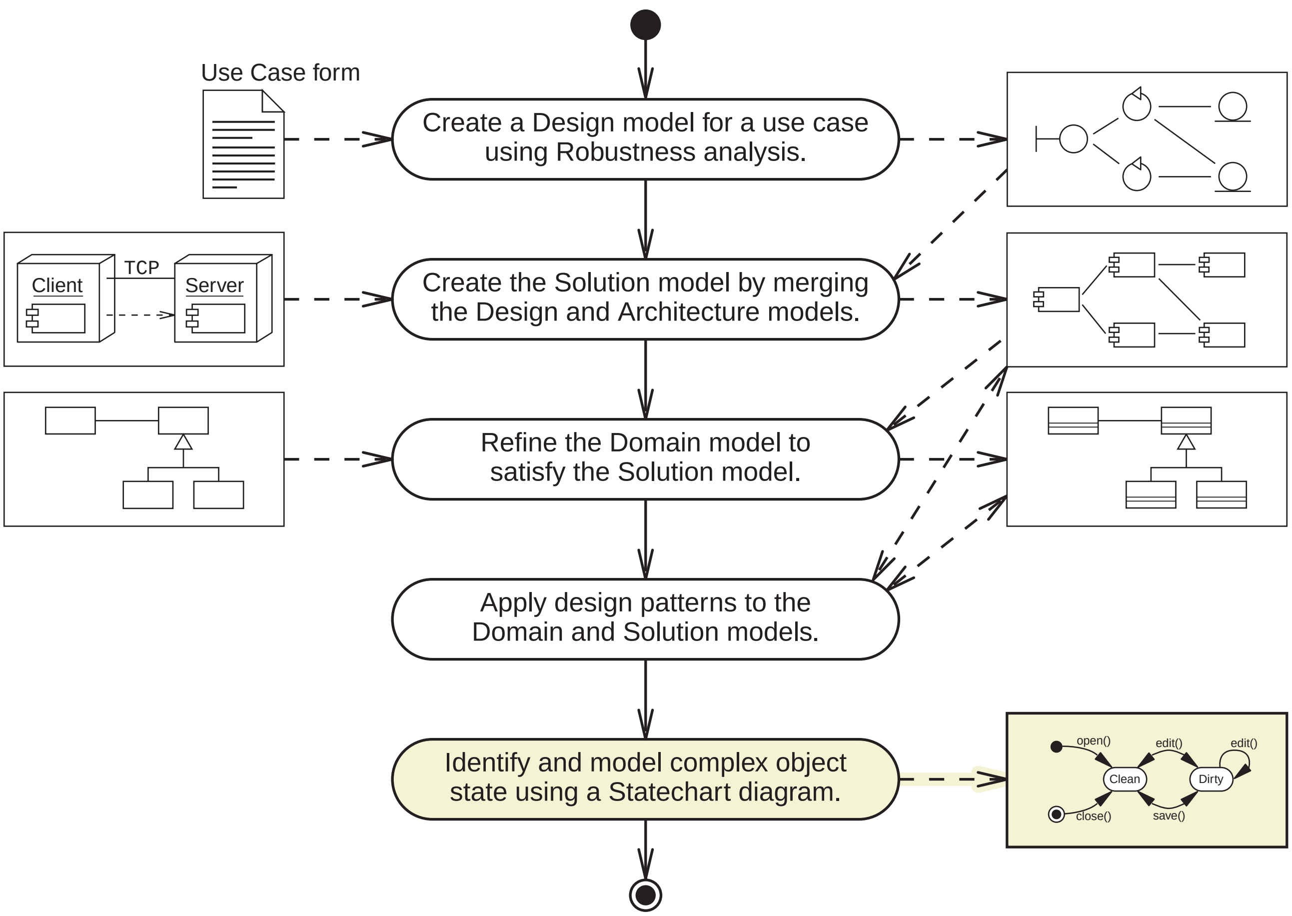
Objectives

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

- Model object state
- Describe techniques for programming complex object state

Process Map


Stakeholders Mental Model
Requirements Model
NFRs
FRs
Architecture Model
Design Model
Solution Model
Code


Use Case form
Create a Design model for a use case using Robustness analysis.
Client
TCP
Server
Create the Solution model by merging the Design and Architecture models.
Refine the Domain model to satisfy the Solution model.
Apply design patterns to the Domain and Solution models.
Identify and model complex object state using a Statechart diagram.
open()
edit()
edit()
Clean
Dirty
close()
save()

Introducing Object State

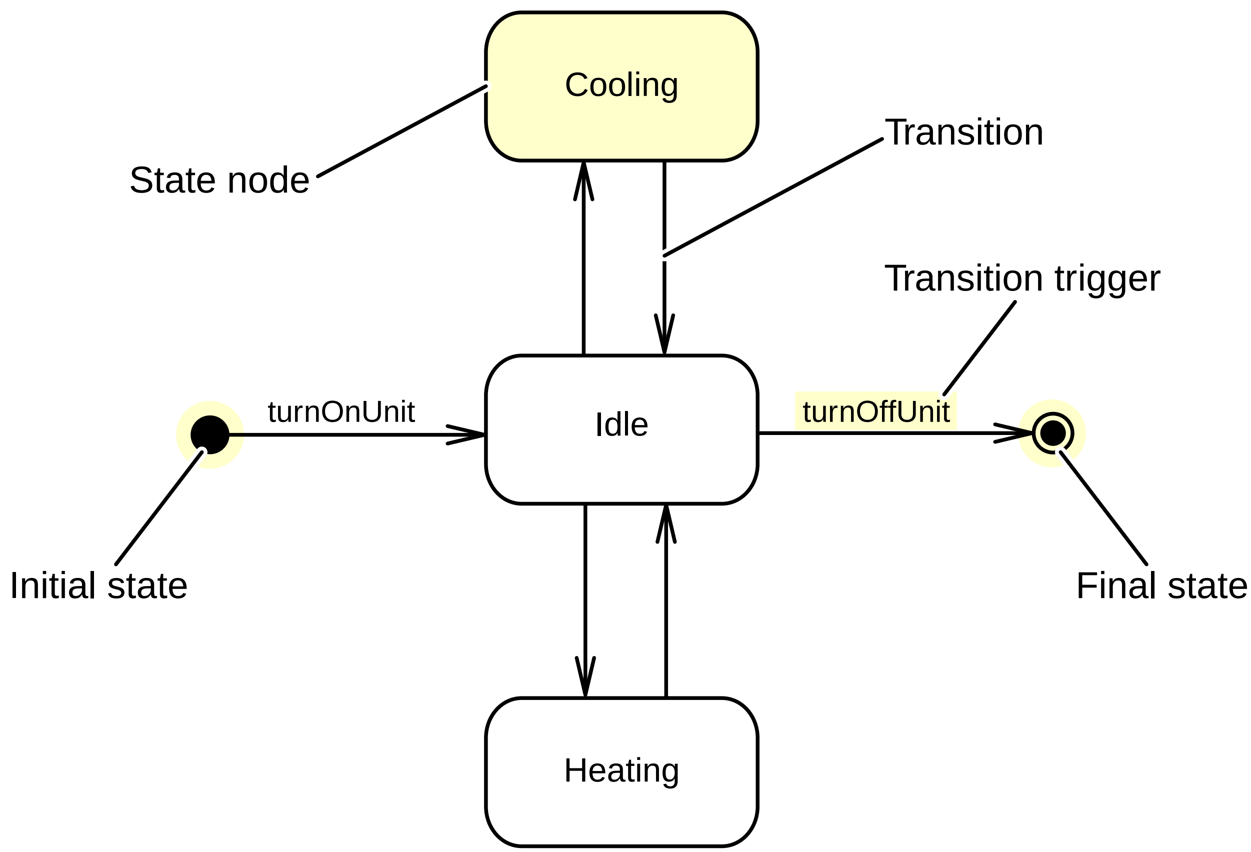
State is "1a: mode or condition of being" (Webster)

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.

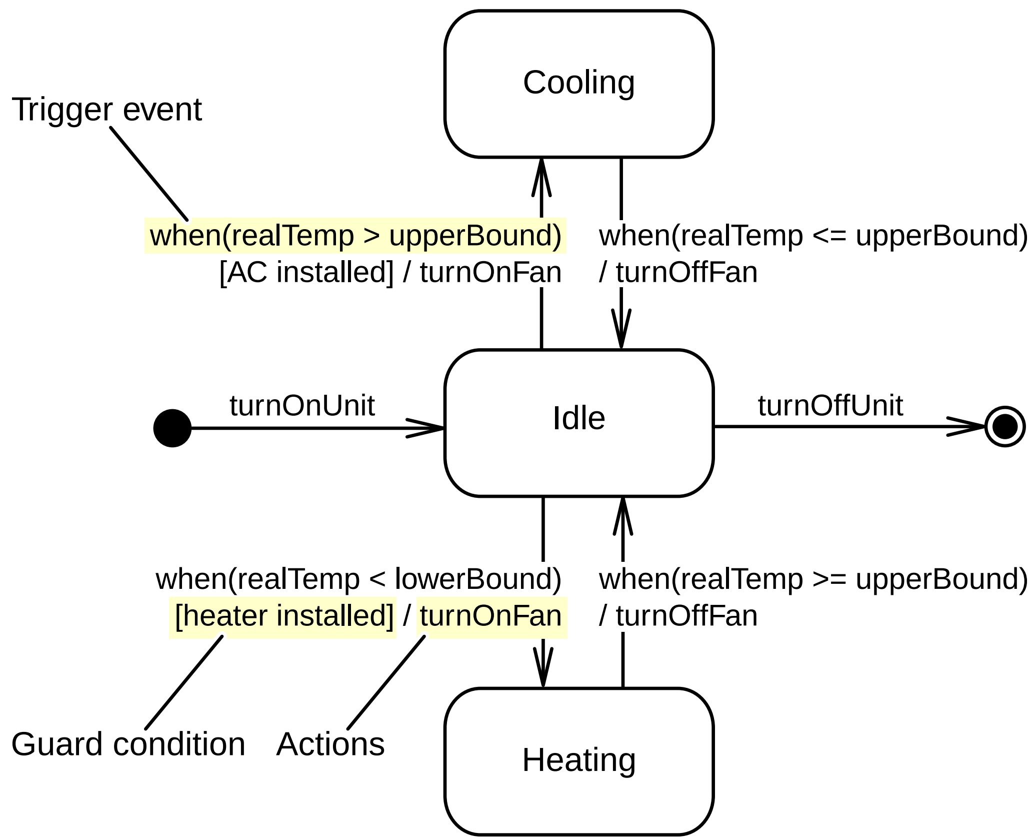
This module considers the second definition.

Identifying the Elements of a Statechart Diagram



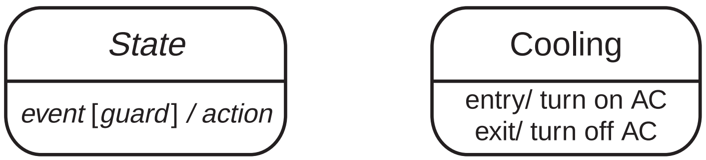
State Transitions

A state transition represents a change of state at runtime.



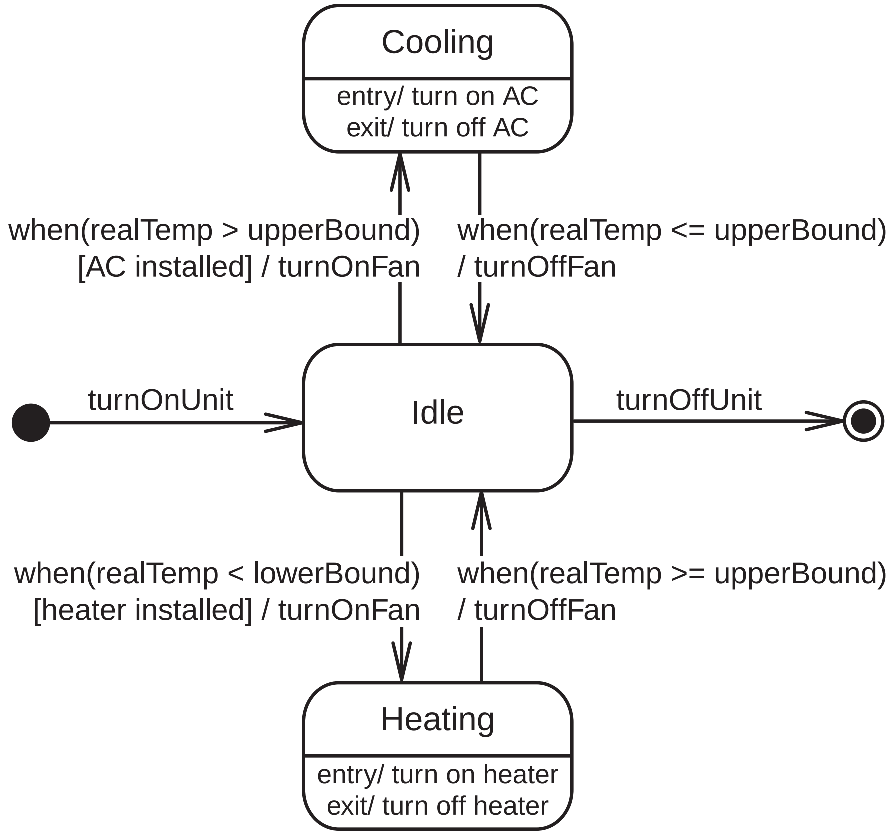
Internal Structure of State Nodes

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.

The Complete HVAC Statechart Diagram

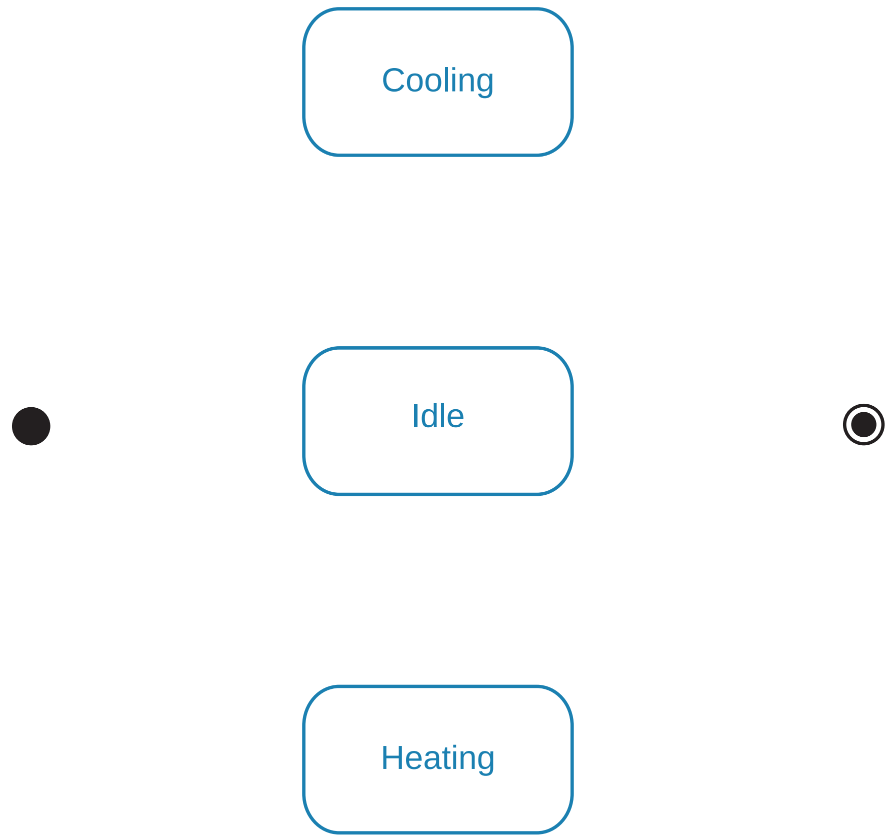


Creating a Statechart Diagram for a Complex Object

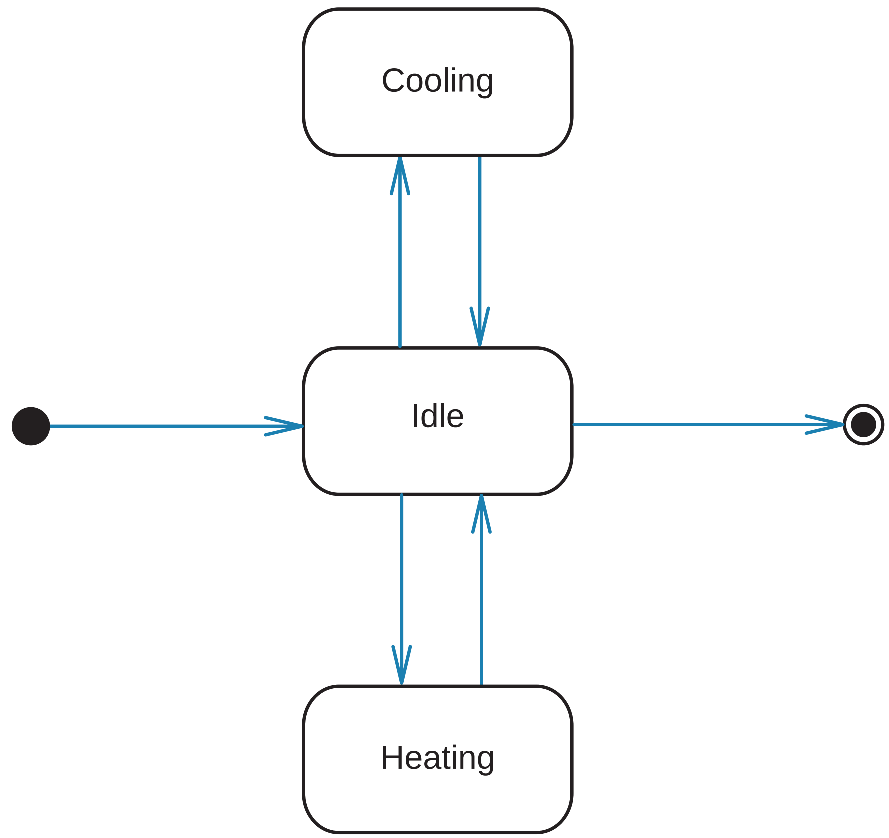
1. Draw the initial and final state for the object.
2. Draw the stable states of the object.
3. Specify the partial ordering of stable states over the lifetime of the object.
4. Specify the events that trigger the transitions between the states of the object. Specify transition actions (if any).
5. Specify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

Step 1 – Start With the Initial and Final States

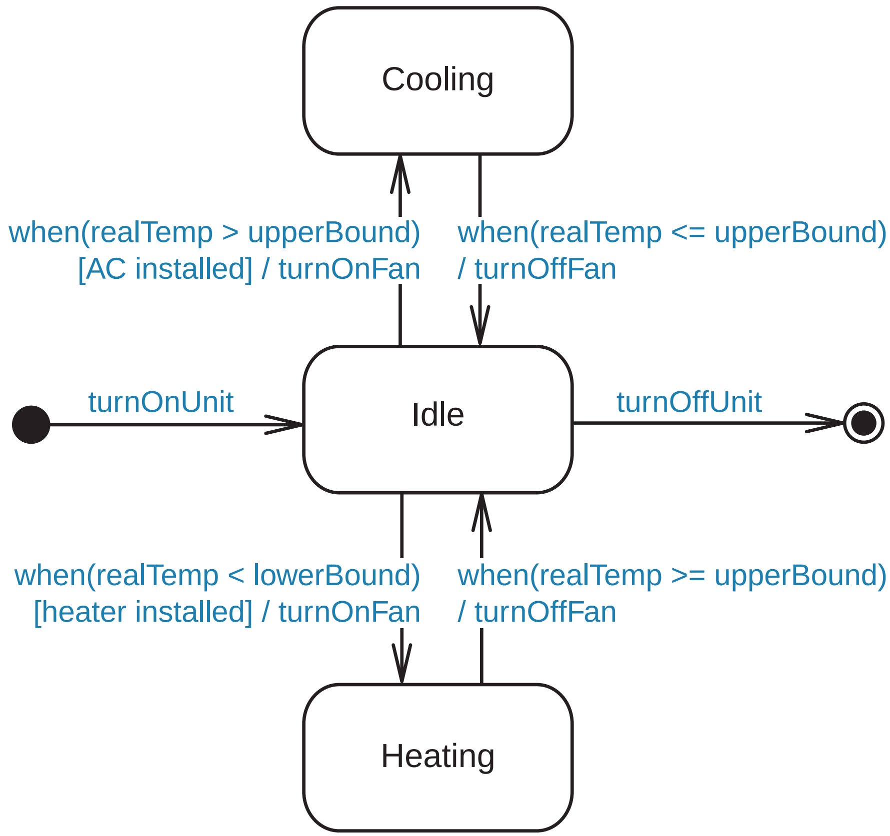
Step 2 – Determine Stable Object States



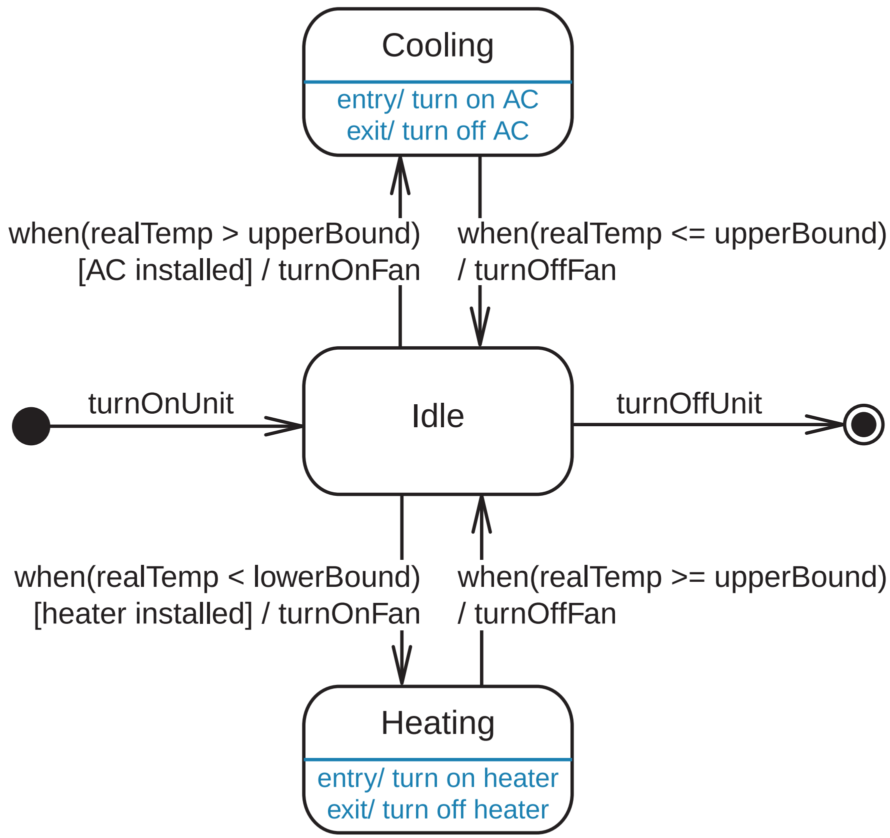
Step 3 – Specify the Partial Ordering of States



Step 4 – Specify the Transition Events and Actions



Step 5 – Specify the Actions Within a State



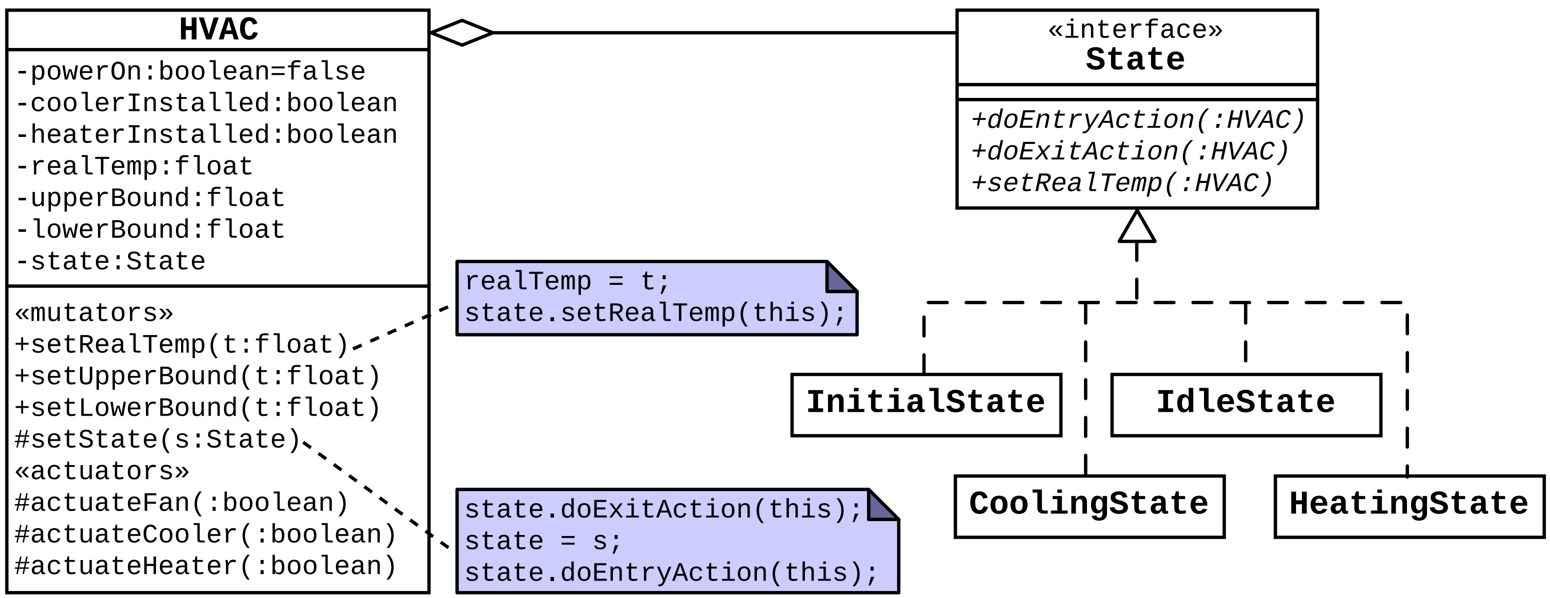
Coding a Complex Object

HVAC
-powerOn:boolean=false -coolerInstalled:boolean -heaterInstalled:boolean -realTemp:float -upperBound:float -lowerBound:float
«mutators» +setRealTemp(t:float) +setUpperBound(t:float) +setLowerBound(t:float) «actuators» #actuateFan(:boolean) #actuateCooler(:boolean) #actuateHeater(:boolean)

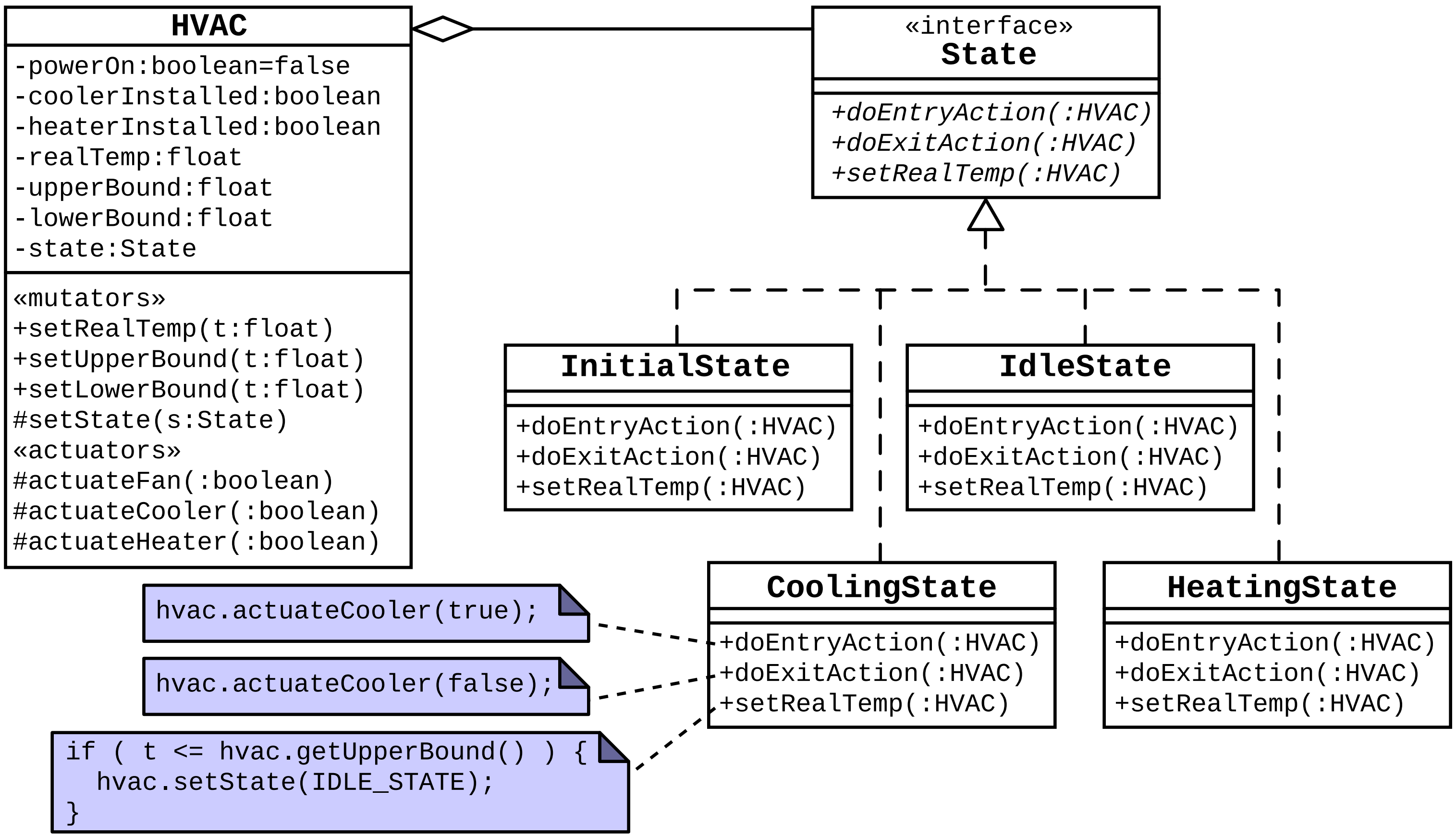
```
if ( powerOn ) {
  if ( realTemp > upperBound ) {
    // cooling
    actuateFan(true);
    if ( coolerInstalled ) {
      actuateCooler(true);
    }
  } else if ( realTemp < lowerBound ) {
    // heating
    actuateFan(true);
    if ( heaterInstalled ) {
      actuateHeater(true);
    }
  } else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

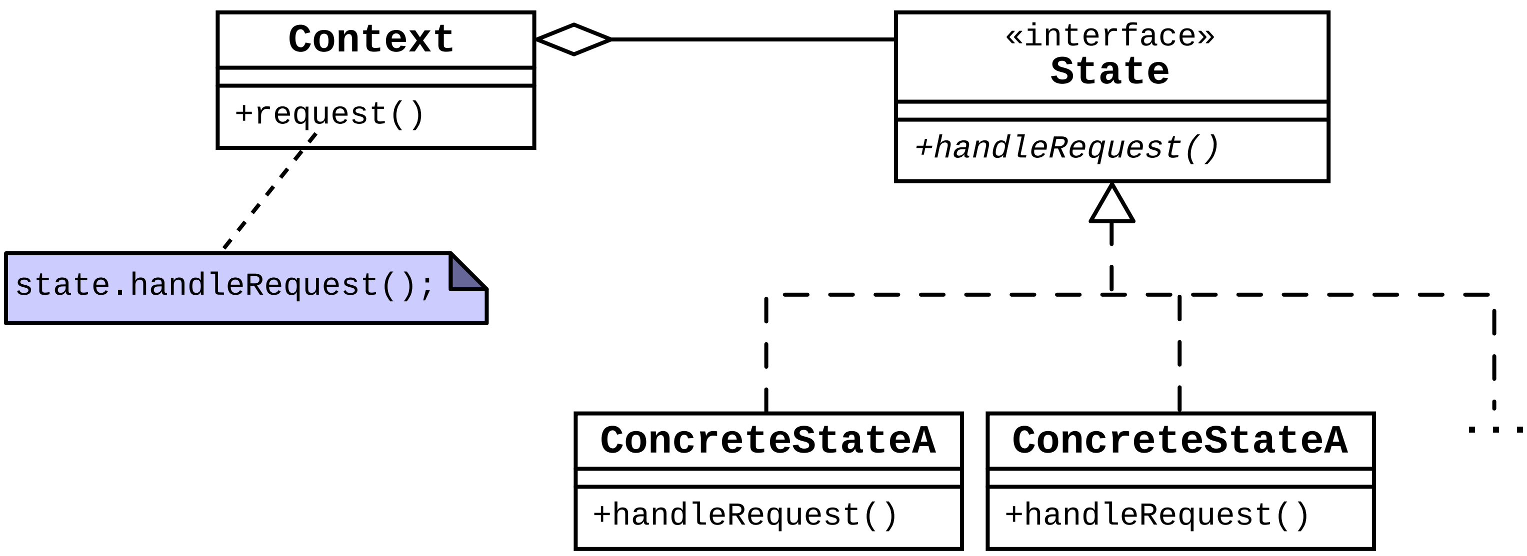
- An object's runtime behavior depends on its state.
- Operations have large, multipart conditional statements that depend on the object's state.

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

- A complex object might have states which define unique behaviors for the object.
- The Statechart diagram provides a mechanism for modeling the states and transitions of an object.
- The State design pattern provides an implementation mechanism for objects with complex state behavior.

Course Contents

About This Course .... Preface-iv
Course Goals .... Preface-v
Course Map .... Preface-vi
Course Map .... Preface-vii
Topics Not Covered .... Preface-viii
How Prepared Are You? .... Preface-ix
Introductions .... Preface-x
How to Use the Icons .... Preface-xi
Typographical Conventions and Symbols .... Preface-xii

Introducing the Software Development Process 1-1
Objectives 1-2
Describing Software Methodology 1-3
The OOSD Hierarchy 1-4
Listing the Workflows of the OOSD Process 1-5
Comparing the Procedural and OO Paradigms 1-6
Describing the Software Team Job Roles 1-7
Exploring the Benefits of Modeling Software 1-8
What is a Model? 1-9
Why Model Software? 1-10
OOSD as Model Transformations 1-11
Defining the UML 1-12
UML Elements 1-13
UML Diagrams 1-14
Views 1-15
What UML Is and Is Not 1-16

UML Tools 1-17
Exploring the Requirements Gathering Workflow 1-18
Activities and Artifacts of the Requirements Gathering Workflow 1-19
Exploring the Requirements Analysis Workflow 1-20
Activities and Artifacts of the Requirements Analysis Workflow 1-21
Exploring the Architecture Workflow 1-22
Activities and Artifacts of the Architecture Workflow 1-23
Exploring the Design Workflow 1-24
Activities and Artifacts of the Design Workflow 1-25
Exploring the Construction Workflow 1-26
Activities and Artifacts of the Construction Workflow 1-27
Summary 1-28

Examining Object-Oriented Technology 2-1
Objectives 2-2
Examining Object-Oriented Principles 2-3
Software Complexity 2-5
Software Decomposition 2-6
Software Costs 2-8
Surveying the Fundamental OO Principles 2-9
Objects 2-10
Objects: Example 2-11
Classes 2-12
Classes: Example 2-13
Abstraction 2-14
Abstraction: Example 2-15
Cohesion 2-16
Cohesion: Example 2-17
Encapsulation 2-18
Encapsulation: Example 2-19

Inheritance 2-20
Inheritance: Example 2-21
Abstract Classes 2-22
Abstract Classes: Example 2-23
Interfaces 2-24
Interfaces: Example 2-25
Polymorphism 2-26
Polymorphism: Example 2-27
Coupling 2-29
Object Associations 2-30
Object Associations: Example 2-31
Summary 2-32

Choosing an Object-Oriented Methodology 3-1
Objectives 3-2
Reviewing Software Methodology 3-3
Exploring Methodology Best Practices 3-4
Use-Case-Driven 3-5
Systemic-Quality-Driven 3-6
Architecture-Centric 3-7
Iterative and Incremental 3-8
Model-Based 3-9
Design Best Practices 3-10
Surveying Several Methodologies 3-11
Waterfall 3-12
Unified Software Development Process 3-14
Rational Unified Process 3-16
SunTone Architecture Methodology 3-18
eXtreme Programming 3-20
Choosing a Methodology 3-22

Choosing Waterfall 3-23
Choosing UP 3-24
Choosing RUP 3-25
Choosing SunTone Architecture Methodology 3-26
Choosing XP 3-27
Summary 3-28

Determining the Project Vision 4-1
Objectives 4-2
Process Map 4-3
Interviewing Business Owners 4-4
Types of Requirements 4-5
Interview Skills 4-6
Vision Interview Focus 4-7
Business Case Questions 4-8
Questions Used to Discover Functional Requirements 4-9
Questions Used to Discover Risks 4-10
Questions Used to Discover Constraints 4-11
Questions Used to Discover Stakeholders 4-12
Analyzing the Vision Interview 4-13
Identifying NFRs 4-14
Identifying Risks 4-15
Political Risks 4-16
Technology Risks 4-17
Resource Risks 4-18
Skills Risks 4-19
Requirement Risks 4-20
Creating the Vision document 4-21
Writing a Problem Statement 4-22
Documenting the Business Opportunity 4-23

Documenting the Proposed Solution 4-24
List the Identified Risks 4-26
List the Identified Constraints 4-27
Summary 4-28

Gathering the System Requirements 5-1
Objectives 5-2
Process Map 5-3
Planning for Requirements Gathering 5-4
Identifying Sources of Requirements 5-5
Identifying Stakeholders 5-6
Stakeholder List 5-7
Preparing for the Stakeholder Interviews 5-8
Detailed FR Questions 5-9
Requirements Elicitation Issues 5-11
Elicitation Issues: Deletion 5-12
Elicitation Issues: Distortion 5-13
Elicitation Issues: Generalization 5-14
Detailed NFR Questions 5-15
Systemic Qualities 5-16
Performance 5-17
Scalability 5-19
Usability 5-21
Actor Information 5-22
Usability Considerations 5-23
Security 5-24
Creating the SRS Document 5-25
Writing the Introduction 5-26
Writing the Functional Requirements 5-27
Actors Section 5-28

Use Cases Section 5-30
Applications Section 5-31
Detailed Requirements Section 5-32
Writing a Detailed Requirement 5-33
The Importance of Traceability 5-34
Writing the Non-Functional Requirements Section 5-35
Writing the Project Glossary 5-36
Summary 5-37

Creating the Initial Use Case Diagram 6-1
Objectives 6-2
Process Map 6-3
Justifying the Need for a Use Case Diagram 6-4
Identifying the Elements of a Use Case Diagram 6-5
Actors 6-6
Use Cases 6-7
System Boundary 6-8
Use Case Associations 6-9
Developing a Use Case Diagram 6-10
Create the System Boundary 6-11
Add the Customer Actor and Use Cases 6-12
Add the Booking Agent Actor 6-13
Add the Receptionist Actor 6-14
Storing the Use Case Diagram 6-15
Recording Use Case Scenarios 6-16
Selecting Use Case Scenarios 6-17
Writing a Use Case Scenario 6-18
Use Case Scenario Example 6-19
Storing the Use Case Scenarios 6-23
Summary 6-24

Refining the Use Case Diagram 7-1
Objectives 7-2
Process Map 7-3
Analyzing a Use Case 7-4
Use Case Forms 7-5
Creating a Use Case Form 7-7
Step 1– Fill in Values Specified in the SRS Document 7-8
Step 2 – Determine the Pre-Conditions From Scenarios 7-9
Step 3 – Determine the Trigger From Scenarios 7-10
Step 4 – Determine the Flow of Events From the Primary Scenario 7-11
Step 5 – Determine the Alternate Flows From the Secondary Scenarios 7-12
Step 6 – Determine the Post-Conditions 7-13
Expanding High-Level Use Cases 7-14
Analyzing Inheritance Patterns 7-18
Actor Inheritance 7-19
Use Case Specialization 7-20
Analyzing Use Case Dependencies 7-21
The «`include`» Dependency 7-22
The «`extend`» Dependency 7-25
A Combined Example From the Hotel Reservation System 7-27
Validating a Use Case With an Activity Diagram 7-28
Identifying the Elements of an Activity Diagram 7-29
Activities 7-31
Flow of Control 7-32
Branching 7-33
Iteration 7-34
Concurrent Flow of Control 7-35
Creating an Activity Diagram for a Use Case 7-36
Use Case Activities 7-37
Branching 7-38

Concurrent Flow 7-39
Summary 7-40

Determining the Key Abstractions 8-1
Objectives 8-2
Process Map 8-3
Introducing Key Abstractions 8-4
Identifying Candidate Key Abstractions 8-5
SRS Nouns 8-6
Candidate Key Abstractions Form 8-7
Candidate Key Abstractions Form (Example) 8-8
Project Glossary 8-9
Discovering Key Abstraction Using CRC Analysis 8-10
Selecting a Key Abstraction Candidate 8-11
Identifying a Relevant Use Case 8-13
Determining Responsibilities and Collaborators 8-15
Documenting a Key Abstraction Using a CRC Card 8-17
Updating the Candidate Key Abstractions Form 8-19
Summary 8-21

Constructing the Problem Domain Model 9-1
Objectives 9-2
Process Map 9-3
Introducing the Domain Model 9-4
Identifying the Elements of a Class Diagram 9-5
Class Nodes 9-6
Class Node Compartments 9-7
Associations 9-8
Multiplicity 9-9
Navigation 9-10

Association Classes 9-11
Creating a Domain Model 9-12
Step 1 – Draw the Class Nodes 9-13
Step 2 – Draw the Associations 9-14
Step 3 – Label the Associations and Role Names 9-15
Step 4 – Label the Association Multiplicity 9-16
Step 5 – Draw the Navigation Arrows 9-17
Step 6 – Draw the Association Classes 9-18
Validating the Domain Model (Intro) 9-19
Identifying the Elements of an Object Diagram 9-20
Object Nodes 9-21
Links 9-22
Validating the Domain Model Using Object Diagrams 9-23
Step 1 – Create Reservation Scenario 1 9-24
Step 2 – Create Reservation Scenario 1 9-25
Step 3 – Create Reservation Scenario 1 9-26
Step 4 – Create Reservation Scenario 1 9-27
Step 5 – Create Reservation Scenario 1 9-28
Step 6 – Create Reservation Scenario 1 9-29
Create Reservation Scenario No. 2 9-30
Comparing Object Diagrams to Validating the Domain Model 9-31
Revised Domain Model for the Hotel Reservation System 9-32
Summary 9-33

Creating the Design Model Using Robustness Analysis 10-1
Objectives 10-2
Process Map 10-3
Introducing the Design Model 10-4
Comparing Analysis and Design 10-5
Robustness Analysis 10-6

Boundary Components 10-8
Service Components 10-9
Entity Components 10-10
Describing the Robustness Analysis Process 10-11
Identifying the Elements of a Collaboration Diagram 10-12
Performing Robustness Analysis 10-15
Step 1 — Select a Use Case 10-16
Step 2 — Place the Actor in the Diagram 10-17
Step 3a — Identify Boundary Components 10-18
Step 3b — Identify Service Components 10-20
Step 3c — Identify Entity Components 10-21
Analyze All Actions in the Activity Diagram 10-22
Converting the Collaboration Diagram into a Sequence Diagram 10-24
Identifying the Elements of a Sequence Diagram 10-25
Clarifying the Design Model Using a Sequence Diagram 10-26
Step 1 – Arrange Components for the First Activity 10-27
Step 2 – Add Message Links and Activation Bars 10-28
Step 3 — Repeat Step 2 For Each Activity 10-29
Summary 10-31

Introducing Fundamental Architectural Concepts 11-1
Objectives 11-2
Process Map 11-3
Justifying the Need for the Architect Role 11-4
Risks Associated With Large-Scale, Distributed Enterprise Systems 11-5
Quality of Service 11-7
Risk Evaluation and Control 11-8
The Role of the Architect 11-9
Distinguishing Between Architecture and Design 11-10
Architectural Principles 11-11

Dependency Inversion Principle 11-12
Architectural Patterns and Design Patterns 11-13
Applying the SunTone Architecture Methodology 11-14
Tiers 11-16
Layers 11-17
Systemic Qualities 11-18
Summary 11-19

Exploring the Architecture Workflow 12-1
Objectives 12-2
Process Map 12-3
Exploring the Architecture Workflow 12-4
Introducing the Architecture Workflow 12-5
Example Design Model 12-6
Example Architecture Template 12-7
Example Solution Model 12-8
Architectural Views 12-9
Identifying the Elements of a Package Diagram 12-10
Example Package Diagram 12-11
An Abstract Package Diagram 12-12
Identifying the Elements of a Component Diagram 12-13
Characteristics of a Component 12-14
Types of Components 12-15
Example Component Diagrams 12-16
Identifying the Elements of a Deployment Diagram 12-18
The Purpose of a Deployment Diagram 12-19
Types of Deployment Diagrams 12-20
Selecting the Architecture Type 12-21
Standalone Applications 12-23
Client/Server (2-Tier) Applications 12-24

N-Tier Applications 12-25
Web-Centric (N-Tier) Applications 12-26
Enterprise (N-Tier) Architecture Type 12-27
Hotel Reservation System Architecture 12-29
Creating The Detailed Deployment Diagram 12-30
Example Detailed Deployment Diagram 12-31
Creating the Architecture Template 12-32
Example Architecture Template 12-33
Creating the Tiers and Layers Package Diagram 12-34
Tiers and Layers Diagram for the HotelApp 12-35
Tiers and Layers Diagram for the WebPresenceApp 12-36
Summary 12-37
Summary 12-38

Creating an Architectural Model for the Client and Presentation Tiers 13-1
Objectives 13-2
Process Map 13-3
Exploring User Interfaces 13-4
User Interface Prototypes 13-5
User Interface Technologies 13-6
Generic Application Components 13-7
Exploring Graphical User Interfaces 13-8
GUI Design 13-9
HotelApp Screen Hierarchy 13-10
The PAC Pattern 13-11
Elements of a PAC Agent 13-12
An Example PAC Agent 13-13
The PAC Component Types 13-14
GUI Screen Design 13-15
Customer GUI Component Hierarchy 13-16

GUI Event Model 13-17
GUI Listeners as Controller Elements 13-18
The MVC Pattern 13-19
The MVC Component Types 13-20
Example Use of the MVC Pattern 13-21
Recording the Client Tier in the Architecture Model 13-22
Populating the Detailed Deployment Diagram 13-23
Creating the Architecture Template 13-24
Populate the Tiers and Layers Pkg Diagram 13-25
Exploring Web User Interfaces 13-26
Web UI Design 13-27
Example Web Page Flow 13-28
Partial Web UI Form Example 13-29
Web UI Event Model 13-30
The WebMVC Pattern 13-31
The WebMVC Pattern Component Types 13-32
An Example Java Technology Web Application 13-33
The WebMVC Pattern 13-34
Recording the Presentation Tier in the Architecture Model 13-35
Populate the Detailed Deployment Diagram 13-36
Create the Architecture Template 13-37
Populate the Tiers and Layers Pkg Diagram 13-38
Summary 13-39

Creating an Architectural Model for the Business Tier 14-1
Objectives 14-2
Process Map 14-3
Exploring Distributed Object-Oriented Computing 14-4
Local Access to a Service Component 14-5
Applying Dependency Inversion Principle 14-6

An Abstract Version of Accessing a Remote Service 14-7
Accessing a Remote Service With RMI Infrastructure 14-8
Accessing a Remote Service Without a Skeleton Component 14-9
A Remote Service Is an Active Component 14-10
RMI Uses Serialization to Pass Parameters 14-11
The RMI Registry Stores Stubs for Remote Lookup 14-12
Documenting the Business Tier in the Architecture Model 14-13
Populating the Detailed Deployment Diagram 14-14
Creating the Architecture Template 14-15
Populating the Tiers and Layers Pkg Diagram 14-16
Summary 14-17

Creating an Architectural Model for the Resource and Integration Tiers 15-1

Objectives 15-2
Process Map 15-3
Exploring Object Persistence 15-4
Persistence Issues 15-5
Creating a Database Schema for the Domain Model 15-6
Simplified HRS Domain Model 15-8
Step 1 – Map OO Entities to DB Tables 15-9
Step 2 – Specify the Primary Keys 15-10
Step 3 – Specify One-to-Many Relationships 15-11
Step 3 – Specify Many-to-Many Relationships 15-12
Recording the Resource Tier in the Architecture Model 15-13
Populating the Detailed Deployment Diagram 15-14
Creating the Architecture Template 15-15
Populating the Tiers and Layers Pkg Diagram 15-16
Exploring Integration Tier Technologies 15-17
The DAO Pattern 15-18
Recording the Integration Tier in the Architecture Model 15-21

Populating the Detailed Deployment Diagram 15-22
Creating the Architecture Template 15-23
Populating the Tiers and Layers Pkg Diagram 15-24
Summary 15-25

Creating the Solution Model 16-1
Objectives 16-2
Process Map 16-3
Introducing the Solution Model 16-4
Creating the Solution Model for GUI Applications 16-5
A Complete Design Model for the Create Reservation Use Case 16-6
A Complete Solution Model for the Create Reservation Use Case 16-7
Creating the Solution Model for WebUI Applications 16-8
A Complete Design Model for the Create Reservation Use Case 16-9
A Complete Solution Model for the Create Reservation Online Use Case 16-10
Summary 16-11

Refining the Domain Model 17-1
Objectives 17-2
Process Map 17-3
Refining Attributes of the Domain Model 17-4
Refining the Attribute Metadata 17-5
Choosing an Appropriate Data Type 17-7
Choosing an Appropriate Data Type 17-8
Creating Derived Attributes 17-9
Applying Encapsulation 17-10
An Encapsulation Example 17-11
Refining Class Relationships 17-12
Relationship Types 17-13
Association 17-14

Aggregation 17-15
Composition 17-16
Navigation 17-17
Qualified Associations 17-19
Relationship Methods 17-20
Resolving Many-to-Many Relationships 17-22
Resolving Association Classes 17-24
Refining Methods 17-26
Declaring Constructors 17-28
Summary 17-29

Applying Design Patterns to the Solution Model 18-1
Objectives 18-2
Process Map 18-3
Explaining Software Patterns 18-4
Levels of Software Patterns 18-5
Design Principles 18-6
Open Closed Principle 18-8
Composite Reuse Principle 18-9
Dependency Inversion Principle 18-12
Describing the Composite Pattern 18-14
Describing the Strategy Pattern 18-19
Describing the Observer Pattern 18-23
Describing the Abstract Factory Pattern 18-29
Summary 18-33

Modeling Complex Object State Using Statechart Diagrams 19-1
Objectives 19-2
Process Map 19-3
Introducing Object State 19-4

Identifying the Elements of a Statechart Diagram 19-5
State Transitions 19-6
Internal Structure of State Nodes 19-7
The Complete HVAC Statechart Diagram 19-8
Creating a Statechart Diagram for a Complex Object 19-9
Coding a Complex Object 19-15
Describing the State Pattern 19-16
Summary 19-21