Software Engineering I

Chapters 6 & 7

Architecture and Design

Chapter 6
Architectural Design

Architectural design

- Architectural design is concerned with understanding how a software system should be organized and designing the overall structure of that system.
- Architectural design is the critical link between design and requirements engineering, as it identifies the main structural components in a system and the relationships between them.
- The output of the architectural design process is an architectural model that describes how the system is organized as a set of communicating components.
- It is generally accepted that an early stage of agile processes is to design an overall systems architecture.
- Refactoring the system architecture is usually expensive because it affects so many components in the system

Architectural abstraction

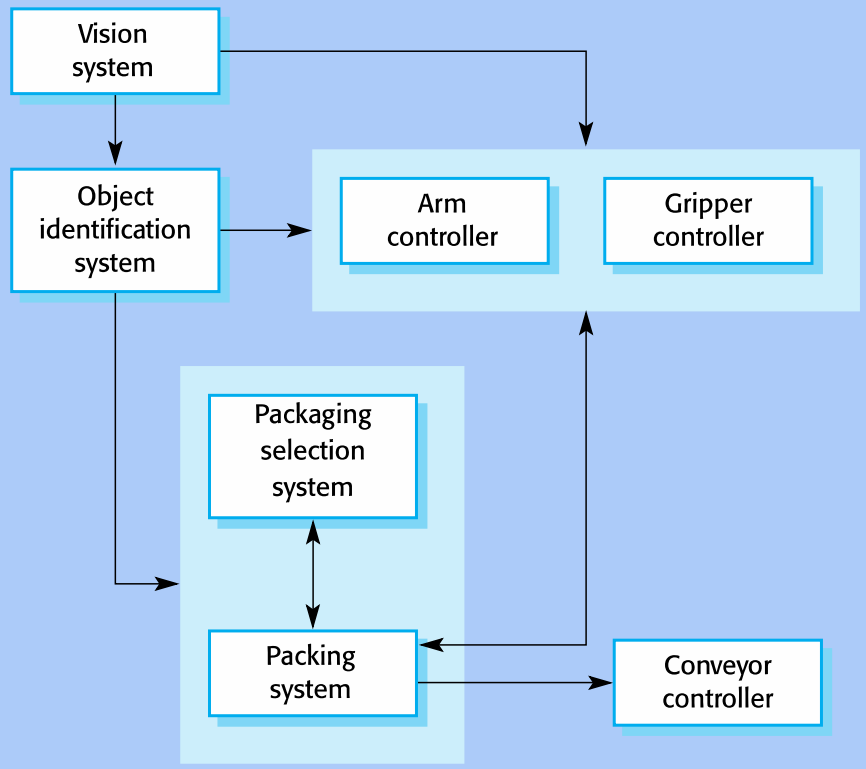
- Architecture in the small is concerned with the architecture of individual programs. At this level, we are concerned with the way that an individual program is decomposed into components.
- Architecture in the large is concerned with the architecture of complex enterprise systems that include other systems, programs, and program components. These enterprise systems are distributed over different computers, which may be owned and managed by different companies.

Advantages of explicit architecture

- Stakeholder communication
  - Architecture may be used as a focus of discussion by system stakeholders.
- System analysis
  - Means that analysis of whether the system can meet its non-functional requirements is possible.
- Large-scale reuse
  - The architecture may be reusable across a range of systems
  - Product-line architectures may be developed.

Architectural representations

- Simple, informal block diagrams showing entities and relationships are the most frequently used method for documenting software architectures.
- But these have been criticized because they lack semantics, do not show the types of relationships between entities nor the visible properties of entities in the architecture.
- Depends on the use of architectural models. The requirements for model semantics depends on how the models are used.
  - Very abstract - do not show nature of component relationships nor externally visible properties of the sub-systems.



- However, useful for communication with stakeholders and for project planning.

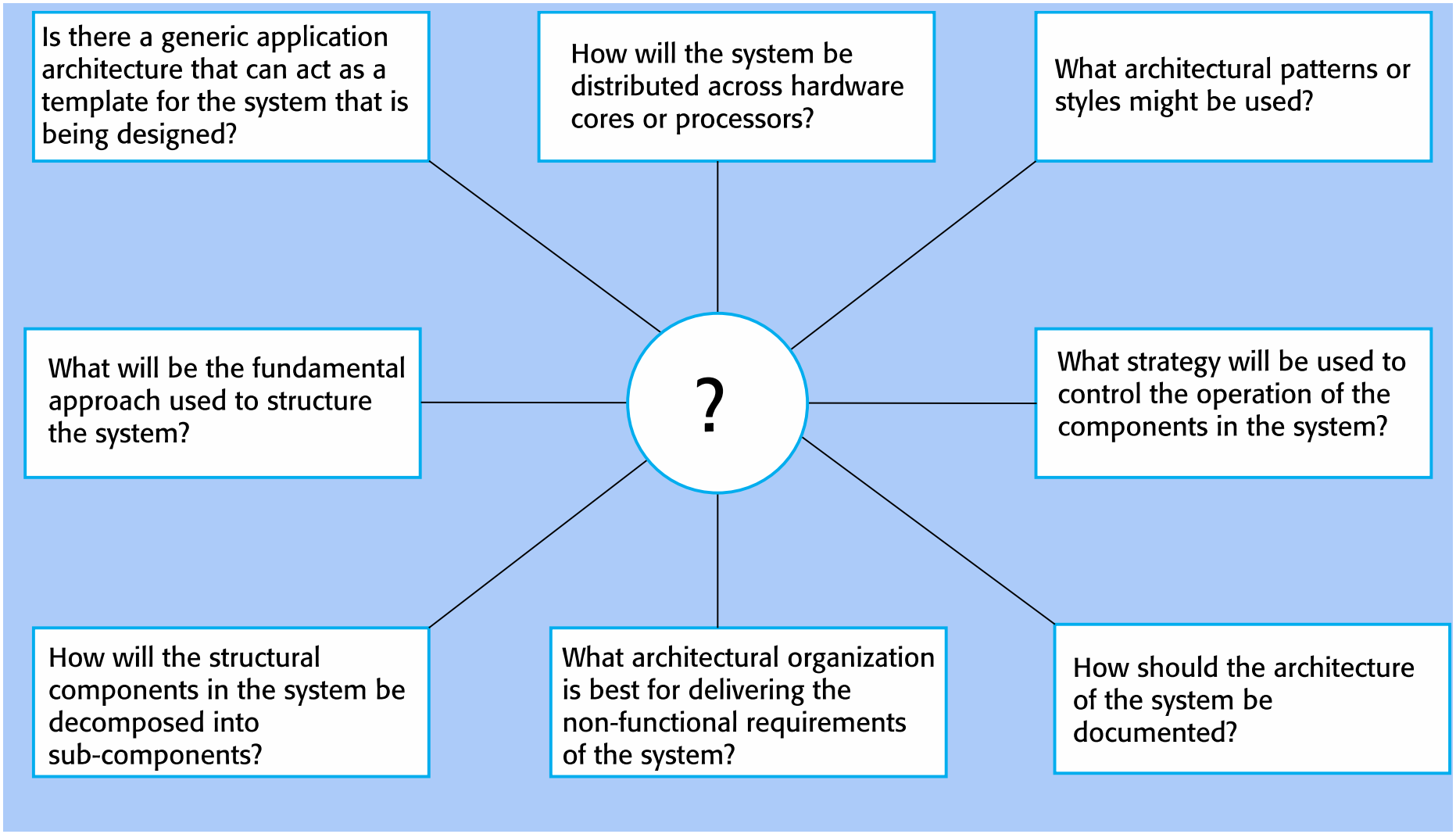
Use of architectural models

- As a way of facilitating discussion about the system design
  - A high-level architectural view of a system is useful for communication with system stakeholders and project planning because it is not cluttered with detail. Stakeholders can relate to it and understand an abstract view of the system. They can then discuss the system as a whole without being confused by detail.
- As a way of documenting an architecture that has been designed
  - The aim here is to produce a complete system model that shows the different components in a system, their interfaces and their connections.

Architectural design decisions

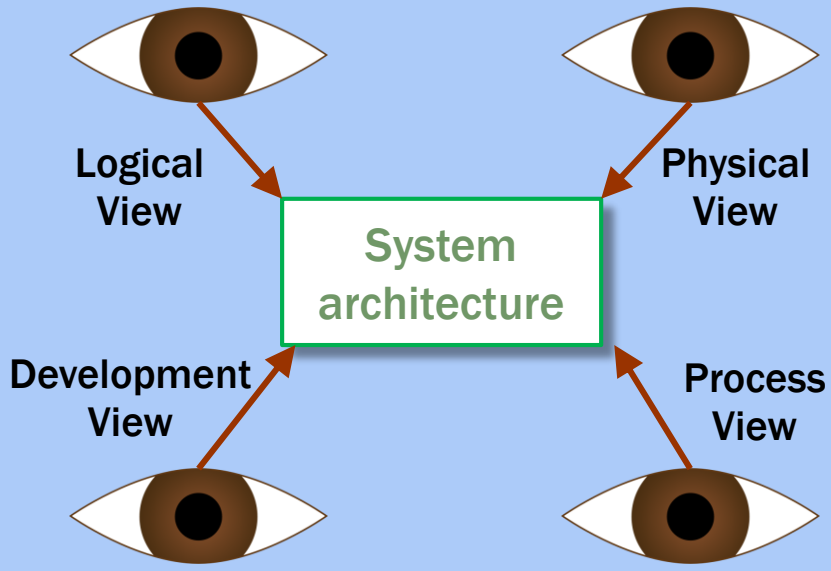
- Architectural design is a creative process so the process differs depending on the type of system being developed.
- However, a number of common decisions span all design processes and these decisions affect the non-functional characteristics of the system.

Architectural design decisions



Architectural views

Architectural views



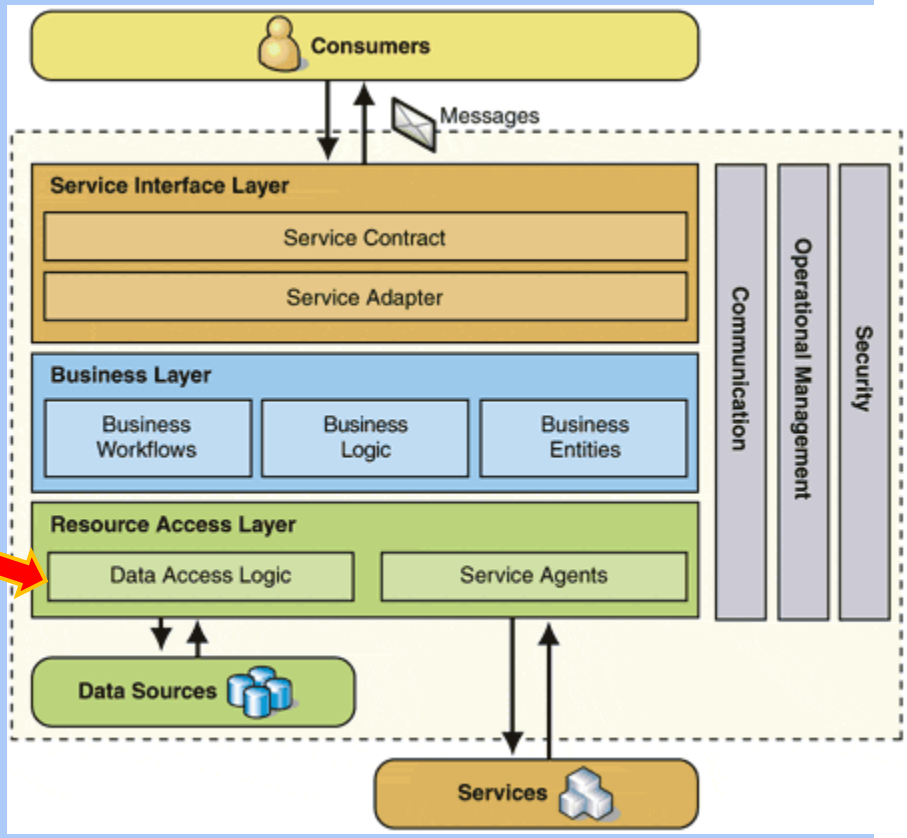
- What views or perspectives are useful when designing and documenting a system's architecture?
- What notations should be used for describing architectural models?
- Each architectural model only shows one view or perspective of the system.
  - It might show how a system is decomposed into modules, how the run-time processes interact or the different ways in which system components are distributed across a network.
  - For both design and documentation, you usually need to present multiple views of the software architecture.

Logical View

Logical View

System architecture

- Shows key abstractions in the system as objects

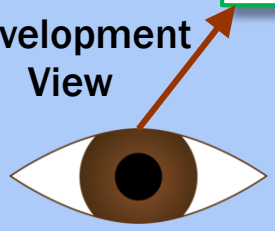


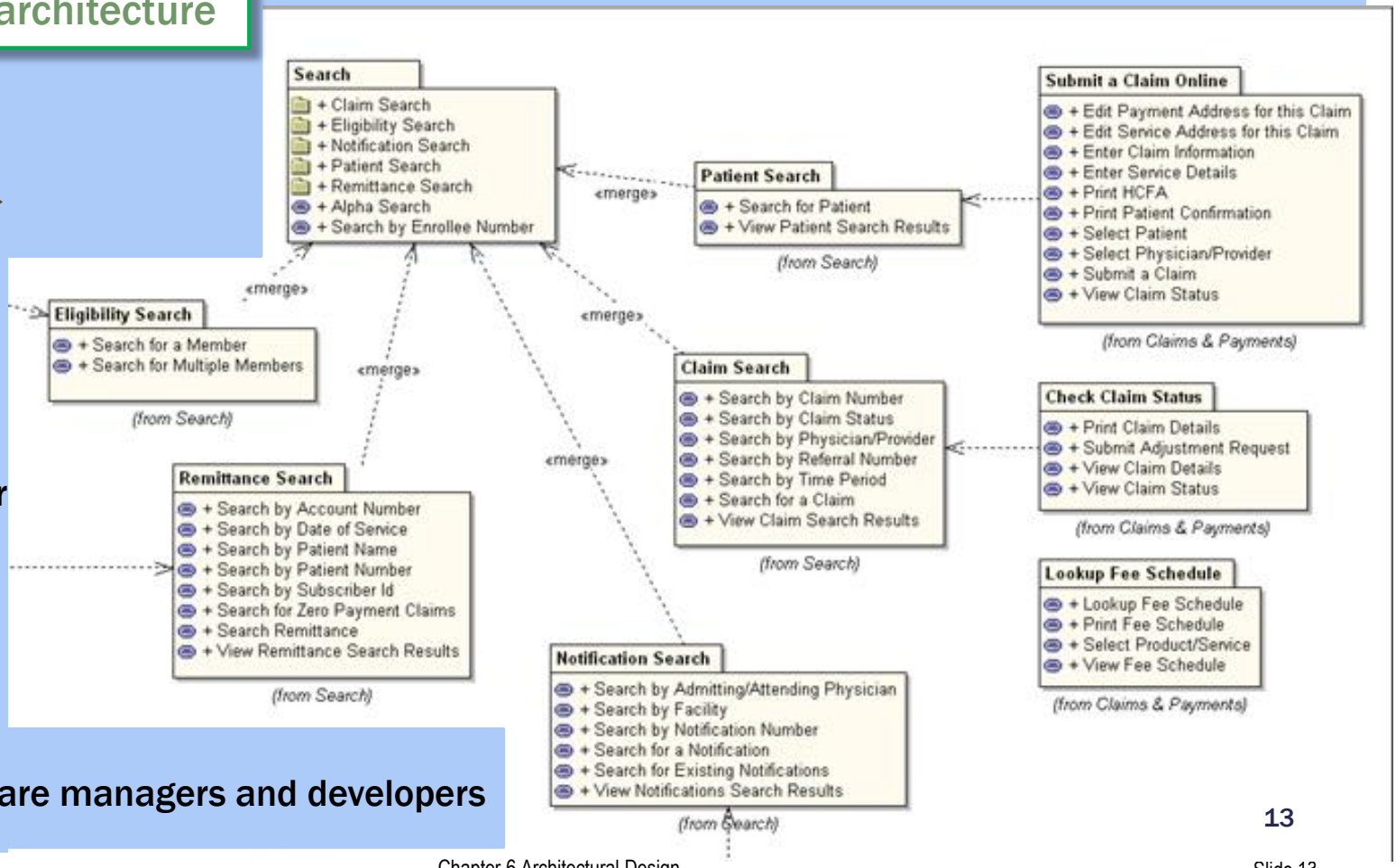
4.1.28
A user should be able to retrieve paintings by location (for example, all paintings in (art galleries of) Italy). The search results returned should be sorted alphabetically according to the name of the art galleries in the location and should be links that take the user to the particular art gallery information.

Development View

System architecture

Development View

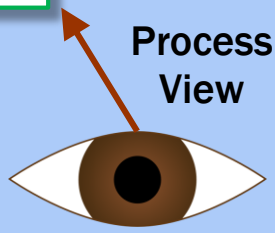




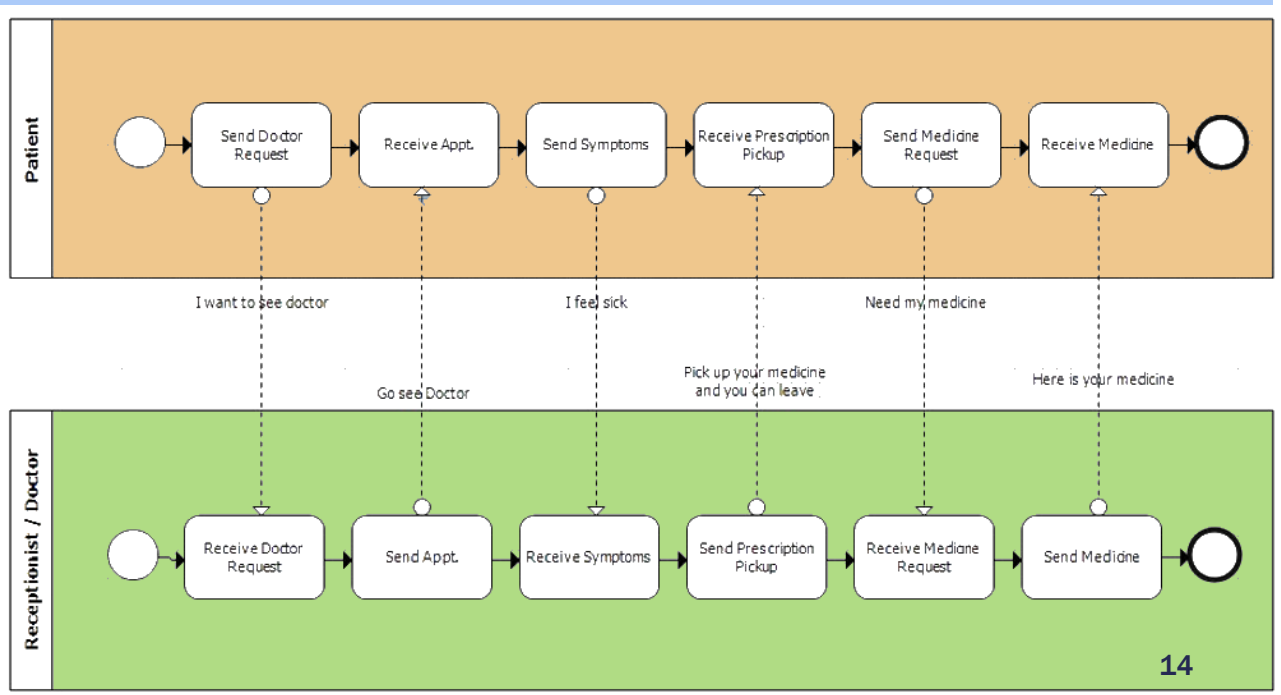
- Shows how the software is decomposed for development, – the breakdown of software into components
- Useful for software managers and developers

Process View

System architecture



- Shows how, at runtime, the system is composed of interacting processes



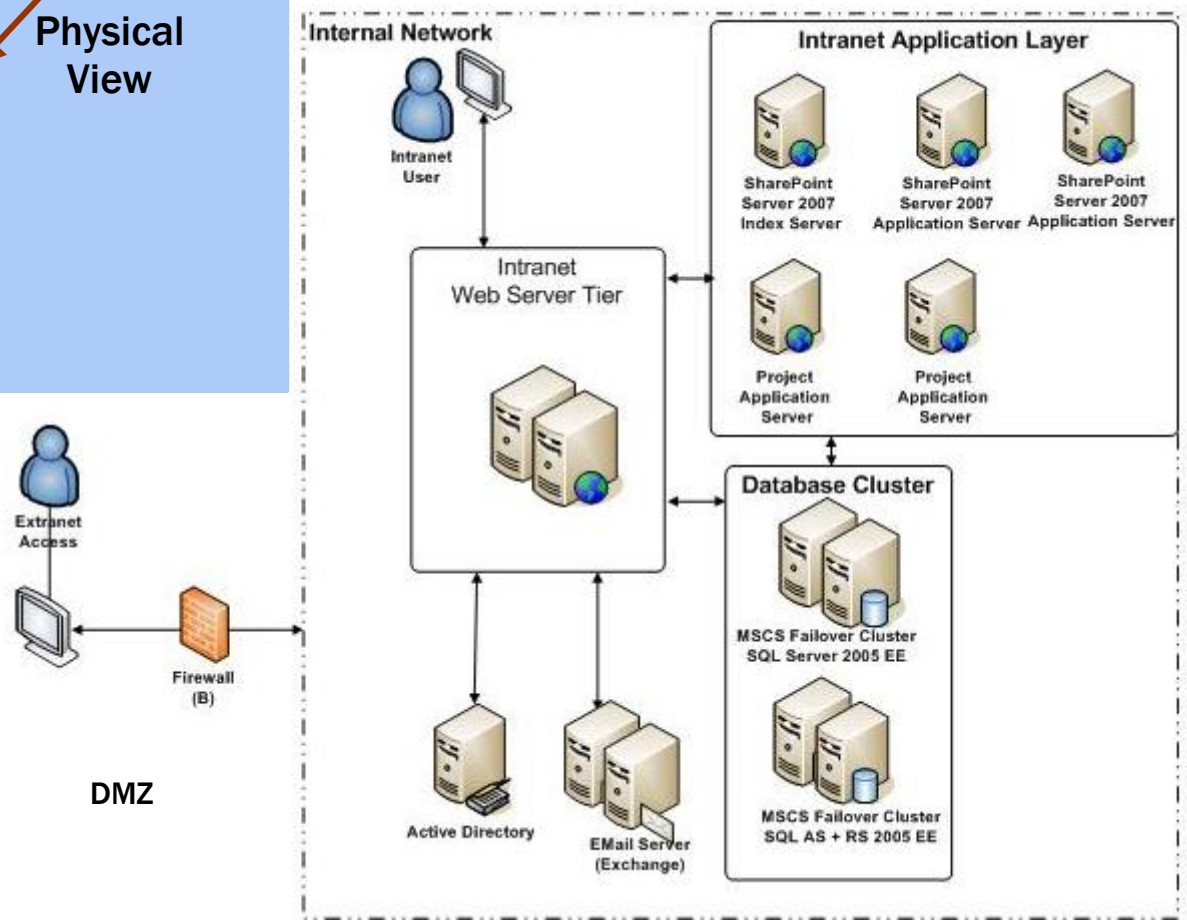
Process chart made with QPR Process Designer

Physical View

Physical View

System architecture

- Shows system hardware and how software components are distributed across processors
- Useful to system engineers



4 + 1 view model of software architecture

- A logical view, which shows the key abstractions in the system as objects or object classes.
- A process view, which shows how, at run-time, the system is composed of interacting processes.
- A development view, which shows how the software is decomposed for development.
- A physical view, which shows the system hardware and how software components are distributed across the processors in the system.
- Related using use cases or scenarios (+1)

Representing architectural views

- Some people argue that the Unified Modeling Language (UML) is an appropriate notation for describing and documenting system architectures
- Your author disagrees with this as he does not think that the UML includes abstractions appropriate for high-level system description.
- Architectural description languages (ADLs) have been developed but are not widely used

Layered architecture

- Used to model the interfacing of sub-systems.
- Organises the system into a set of layers (or abstract machines) each of which provide a set of services.
- Supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.
- However, often artificial to structure systems in this way.

Chapter 7

Design and Implementation

Design and implementation

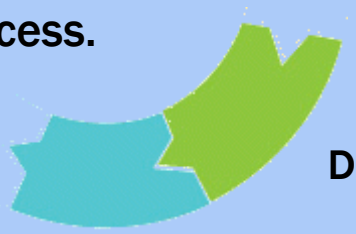
- Software design and implementation is the stage in the software engineering process at which an executable software system is developed.
- Software design and implementation activities are invariably inter-leaved.
  - Software design is a creative activity in which you identify software components and their relationships, based on a customer's requirements.
  - Implementation is the process of realizing the design as a program.

Build or buy

- In a wide range of domains, it is now possible to buy off-the-shelf systems (COTS) that can be adapted and tailored to the users' requirements.
  - For example, if you want to implement a medical records system, you can buy a package that is already used in hospitals. It can be cheaper and faster to use this approach rather than developing a system in a conventional programming language.
- When you develop an application in this way, the design process becomes concerned with how to use the configuration features of that system to deliver the system requirements.

Software design and development

- The activities of design and development are closely related and may be inter-leaved.
- The software is created either by developing a program or programs or by configuring an application system.
- Programming is an individual activity with no standard process.
- Debugging is the activity of finding program faults and correcting these faults.



This begs the question:

Does the Design drive the Development?
Or does the Development drive the Design?

UML

Current Spec is UML 2.5

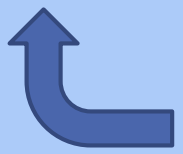
http://www.omg.org/spec/UML/index.htm

Types of UML Diagrams

- **Structural Models**
  - Capture the static features of a system.
  - Represent the framework for the system and this framework is the place where all other components exist.
  - Class diagram
  - Object diagram
  - Package diagram
  - Analysis Class diagram
- **Behavioral Models**
  - Describe the interaction in the system by depicting the interaction among the structural diagrams.
  - Show the dynamic nature or "flow" of the system.
  - Use case diagram
  - *Interaction diagrams*
    - Communication diagram
    - Sequence diagram
  - *Analysis diagrams*
    - Activity diagrams
  - State diagram
- **Architectural Models**
  - Represents the overall framework of the system – contain both structural and behavioral elements of the system.
  - The "blue print" of the entire system.
  - Component diagram
  - Deployment diagram

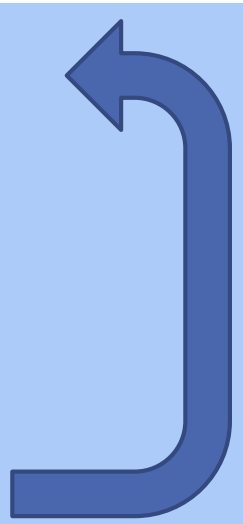
Types of UML Diagrams Sommerville's View

- Structural Models
- Behavioral Models



A system context model is a structural model that demonstrates the other systems in the environment of the system being developed.

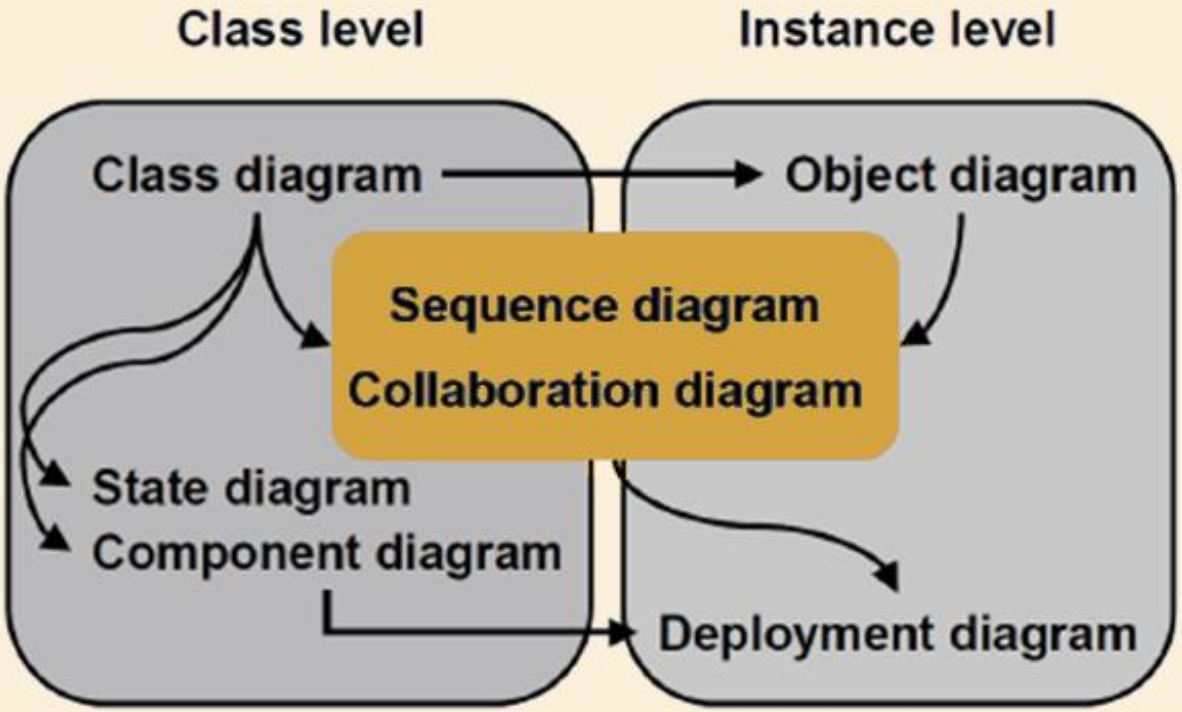
An interaction model is a dynamic model that shows how the system interacts with its environment as it is used.



- Understanding the relationships between the software that is being designed and its external environment is essential for deciding how to provide the required system functionality and how to structure the system to communicate with its environment.

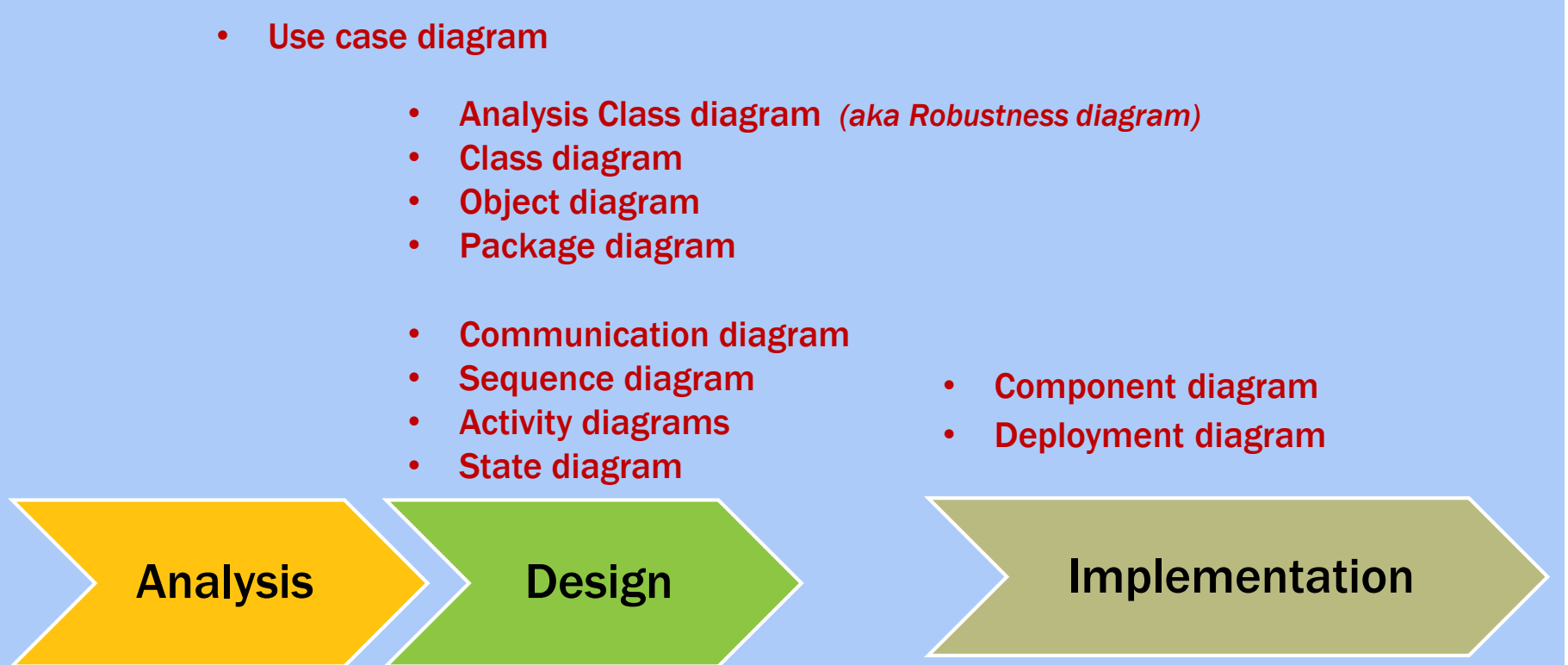
- Understanding of the context also lets you establish the boundaries of the system. Setting the system boundaries helps you decide what features are implemented in the system being designed and what features are in other associated systems.

UML Diagrams: Class level vs Instance Level



Design Artifacts
UML Diagrams by SDLC Phase

- Use Cases (Diagrams and Descriptions) as well as Non-Functional Requirements bridge the Requirements and the Design Phases.



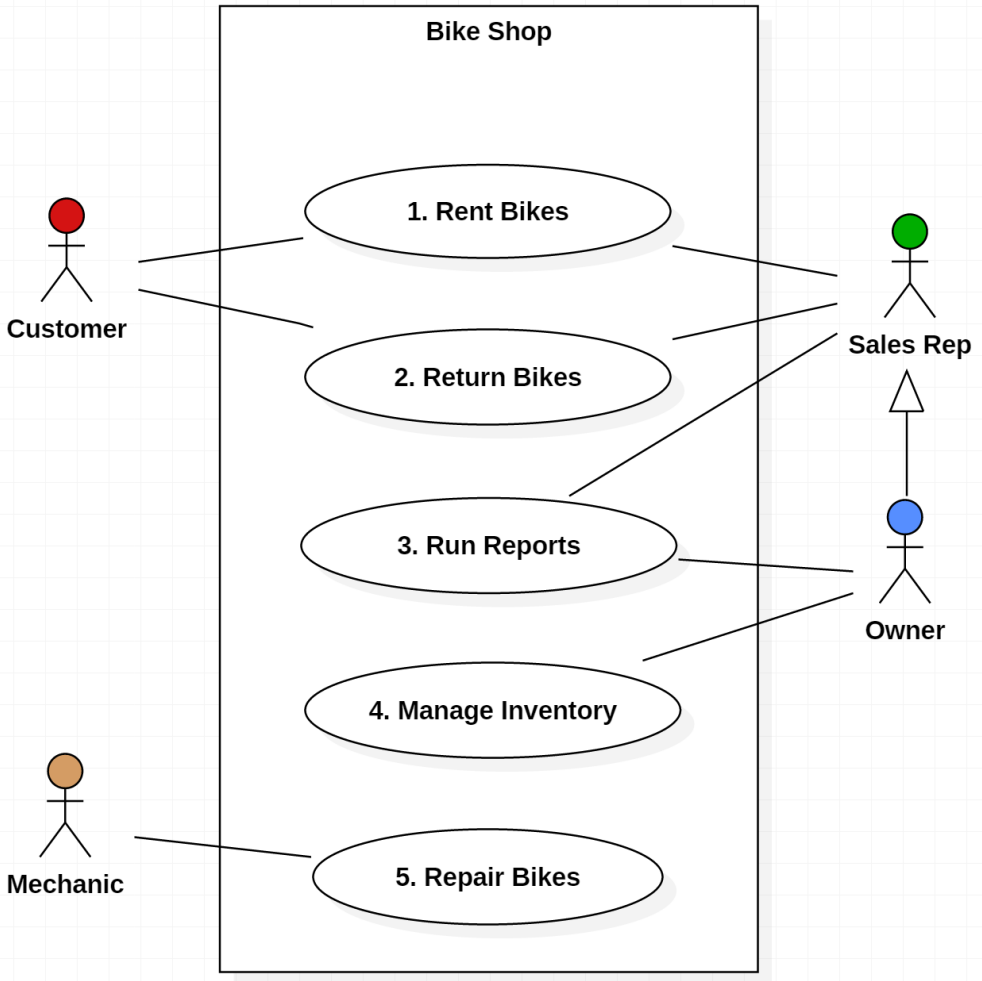
UML Stereotypes

- A stereotype is one of three types of extensibility mechanisms in the Unified Modeling Language (UML),
- A Stereotype provides the capability to create new kind of modeling elements. Stereotypes must be based on elements that are part of the UML meta-model. Some common stereotypes for a class are entity, boundary, control, utility and exception.
- Stereotypes are a way to group classes under a common purpose and are represented by a pair of guillemets (pronounced GEE-may) « »

Use Case Diagrams

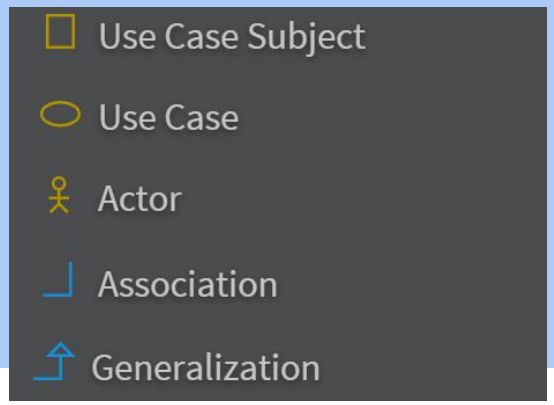
Use Case Diagrams (1)

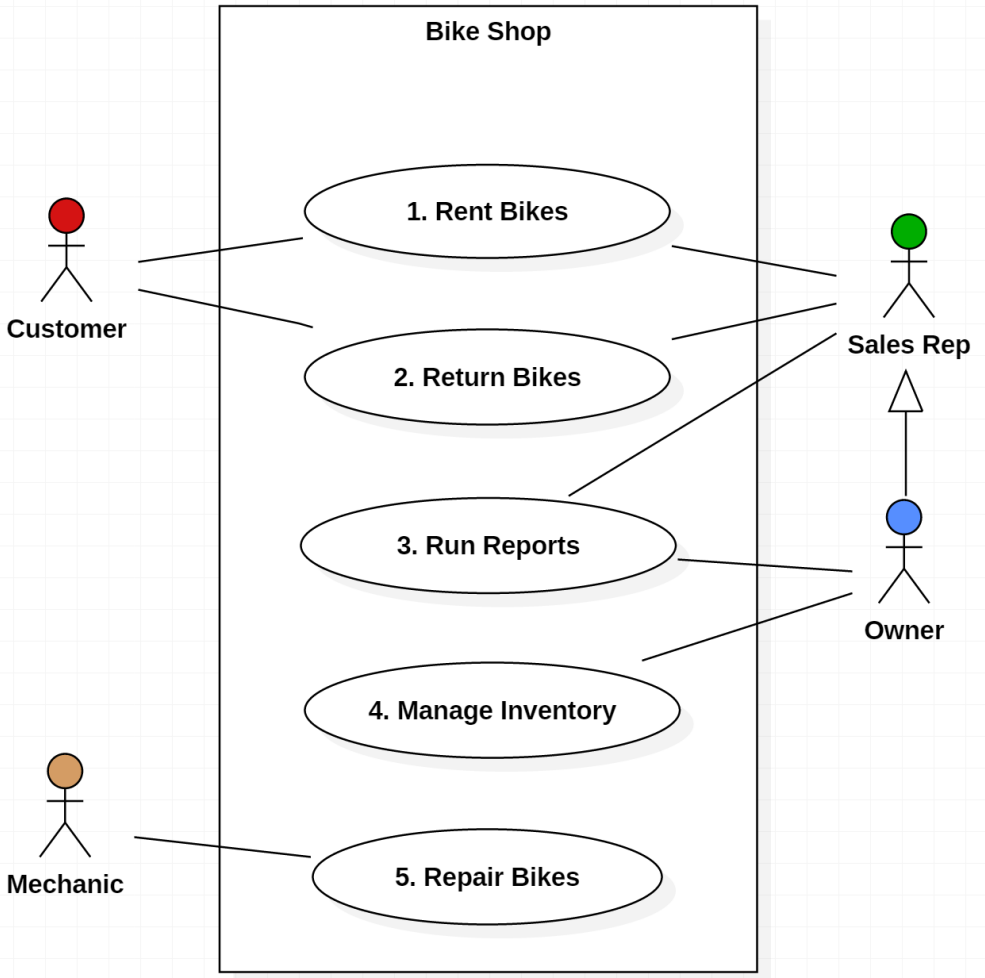
- Purpose
  - Visualize the different types of roles in a system
  - Show how those roles interact with the system
- Usage
  - Requirements Document (simple Use Case Diagrams)
  - Design Document (advanced Use Case Diagrams



Use Case Diagrams (2)

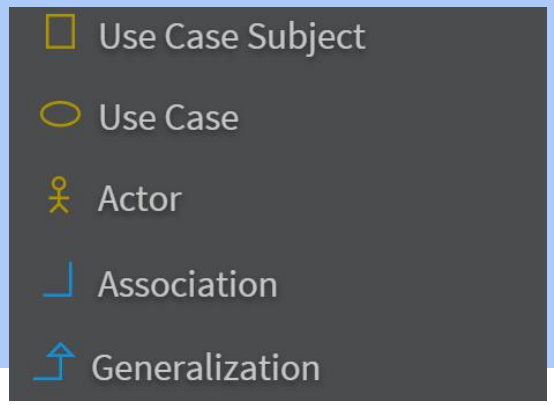
- **Boundary** (aka "Use Case Subject" or "Scope"): defines the considered scope of the diagram.
- **Actor**: any entity that performs a role in a system. Could be a person, organization or an external system.
- **Use Case**: a function or an action within the system.
- **Association**: links an Actor to a Use Case
- **Generalization**: one Actor can inherit (or "act as" the role of another Actor

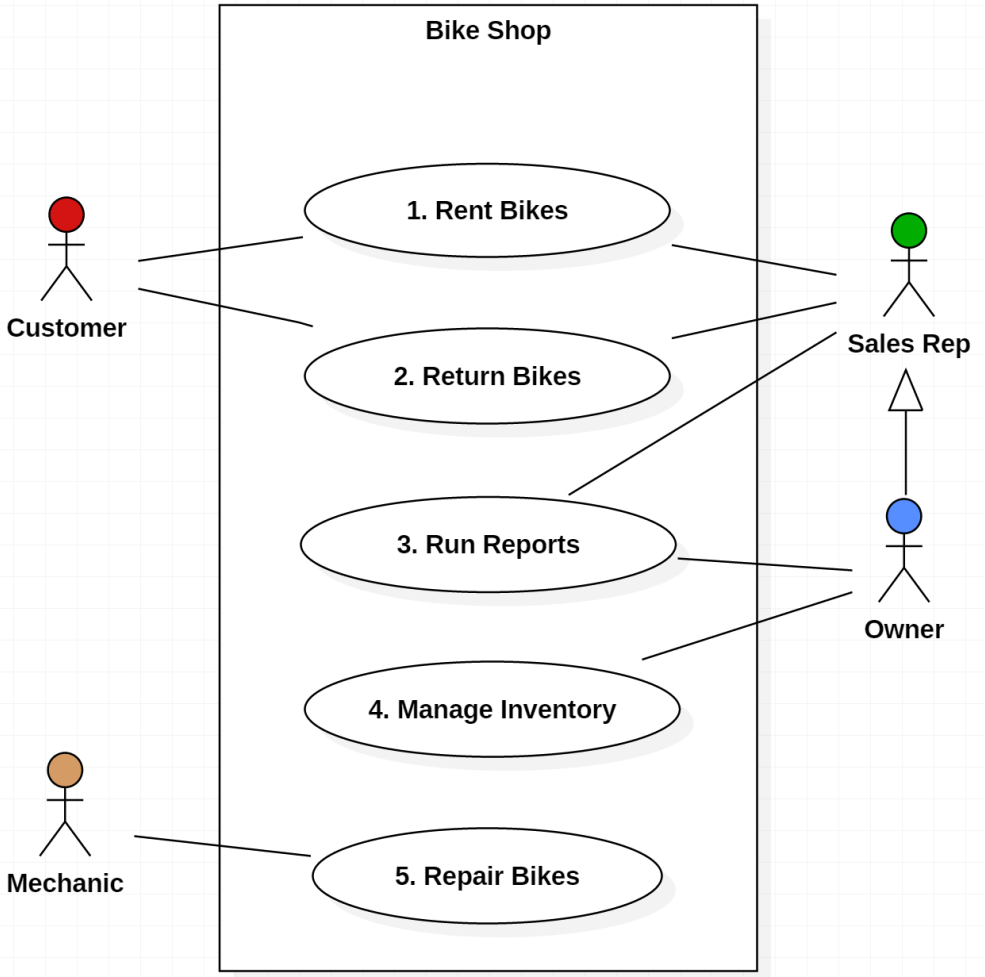




Use Case Diagrams (3)

- Note that the Owner of the Bike Shop can act as a Sales Rep if needed through Generalization
- Both the Sales Rep and the Owner are associated with "Run Reports." Is this redundant?
  - No, there are certain reports a Sales Rep runs that an Owner could also run when he is assuming that role.
  - But the separate Association between the Owner and Run Reports implies that there are some reports only the Owner can run.





Use Case Diagrams

Use Case diagrams consist of

- Actors
  - External entities that interact with the system.
  - Examples of actors include a user role (e.g., a system administrator, a bank customer, a bank teller) or another system (e.g., a central database, a fabrication line).
  - Actors have unique names and descriptions.
- Use cases
  - Describe the behavior of the system as seen from an actor's point of view.
  - A use case describes a function provided by the system as a set of events that yields a visible result for the actors. Actors initiate a use case to access system functionality. The use case can then initiate other use cases and gather more information from the actors. When actors and use cases exchange information, they are said to ***communicate***

Use Case Description (1)

The Overview

Rent Bike Use Case

- Customers have the option to browse or search the database at a customer kiosk to select available bike(s) for rental and specify the rental period for each of the selected bikes. Alternatively, they can have an employee of the bike shop browse or search the database, make customer suggestions with which the customer can agree or disagree, and record the respective rental periods. The selected bikes are then physically examined by an employee to ensure their suitability for rental. If any bikes are unsuitable, they are marked for repair (see Update Bike use case) and alternative bike selections are made by either the customer or employee. Then the system will display to the employee any special discounts that might apply to frequent customers. The employee informs the customer of the final price (including all bike rates, rental durations, rental deposits and discounts) for all bikes rented, accepts the payment for the rental and authorizes the rental. The employee will check out the bikes by either associating the bike(s) with the pre-existing customer record, or by creating a new customer record and then associating that record with the bike(s). A receipt will be printed for the customer which lists the details and cost of each bike rental. A rental request may be cancelled at any time during this process, up until the bikes are associated with the customer. After that, the Return Bike use case will apply.

Use Case Description (2)
The Scenarios

Rent Bike Use Case

Primary Flow "Typical rental"
(consists of subordinate use cases)
- Employee Selects Bikes
- Employee Selects Customer
- Employee Confirms Reservation

Alternate Flow "Customer selects bikes rental"
(consists of subordinate use cases)
- Customer Selects Bikes
- Employee Selects Rental Log
- Employee Selects Customer
- Employee Confirms Reservation

Alternate Flow "Cancellation after bike selection"
(consists of subordinate use cases)
- Employee Selects Bikes
- Cancel Reservation

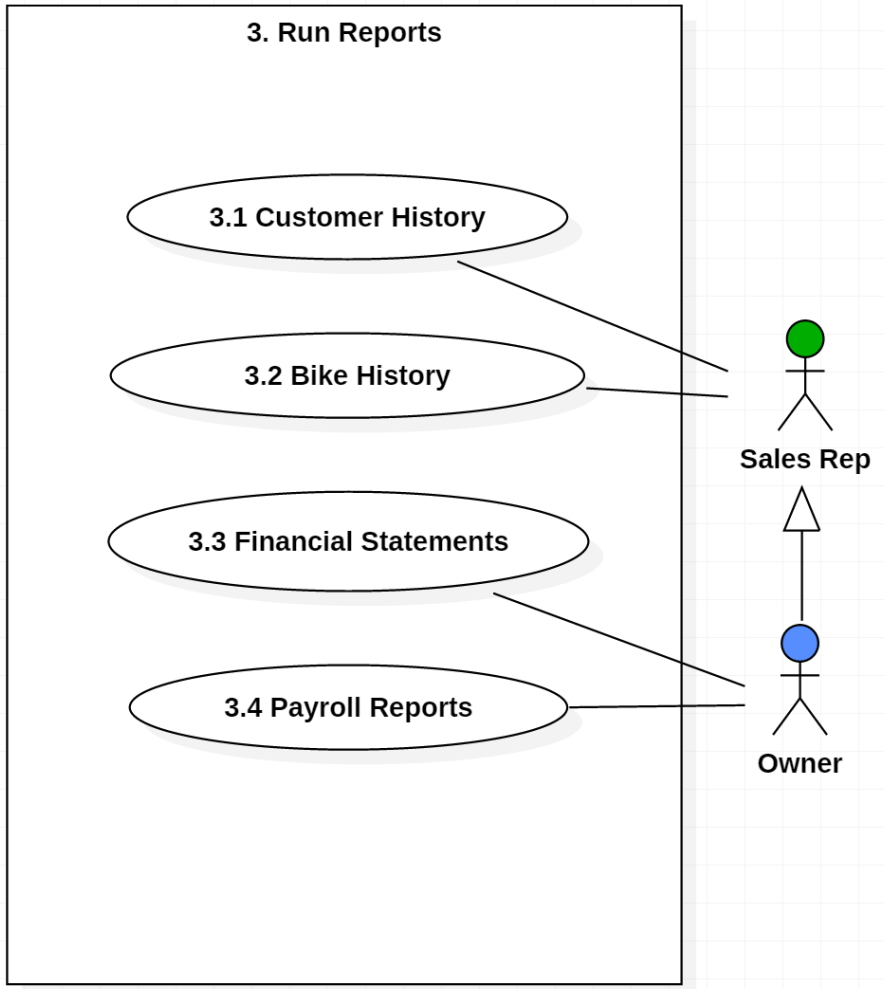
Alternate Flow "Cancellation after bike selection" *(consists of subordinate use cases)*
- Customer Selects Bikes
- Cancel Reservation

Alternate Flow "Cancellation after rental log selection" *(consists of subordinate use cases)*
- Customer Selects Bikes
- Employee Selects Rental Log
- Cancel Reservation

Alternate Flow "Cancellation after customer selection" *(consists of subordinate use cases)*
- Employee Select Bikes
- Employee Selects Customer
- Cancel Reservation

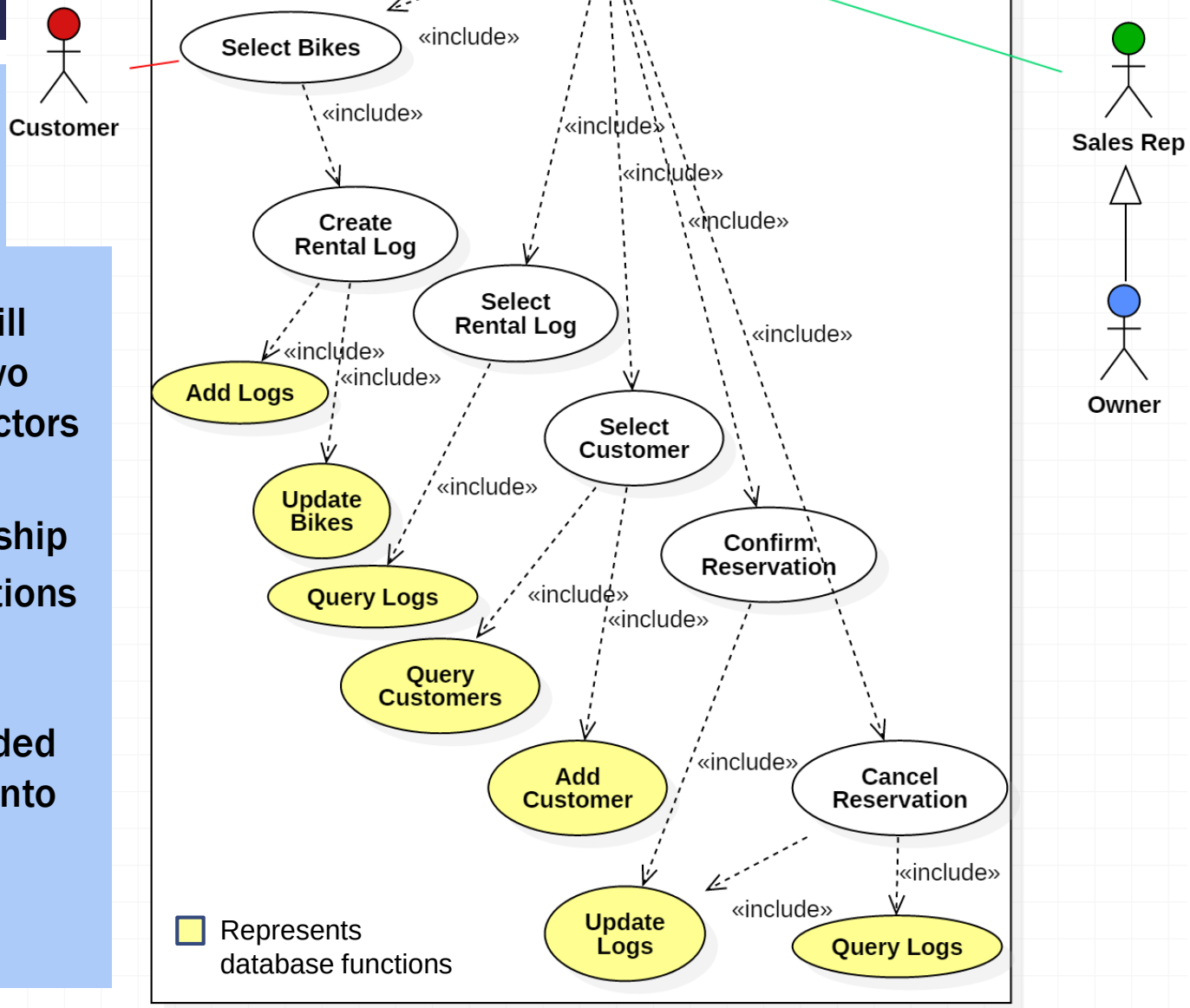
Decomposed Use Case Diagrams

- Use case diagrams can be decomposed to lower levels
- This is why it is a good idea to number your Use Cases



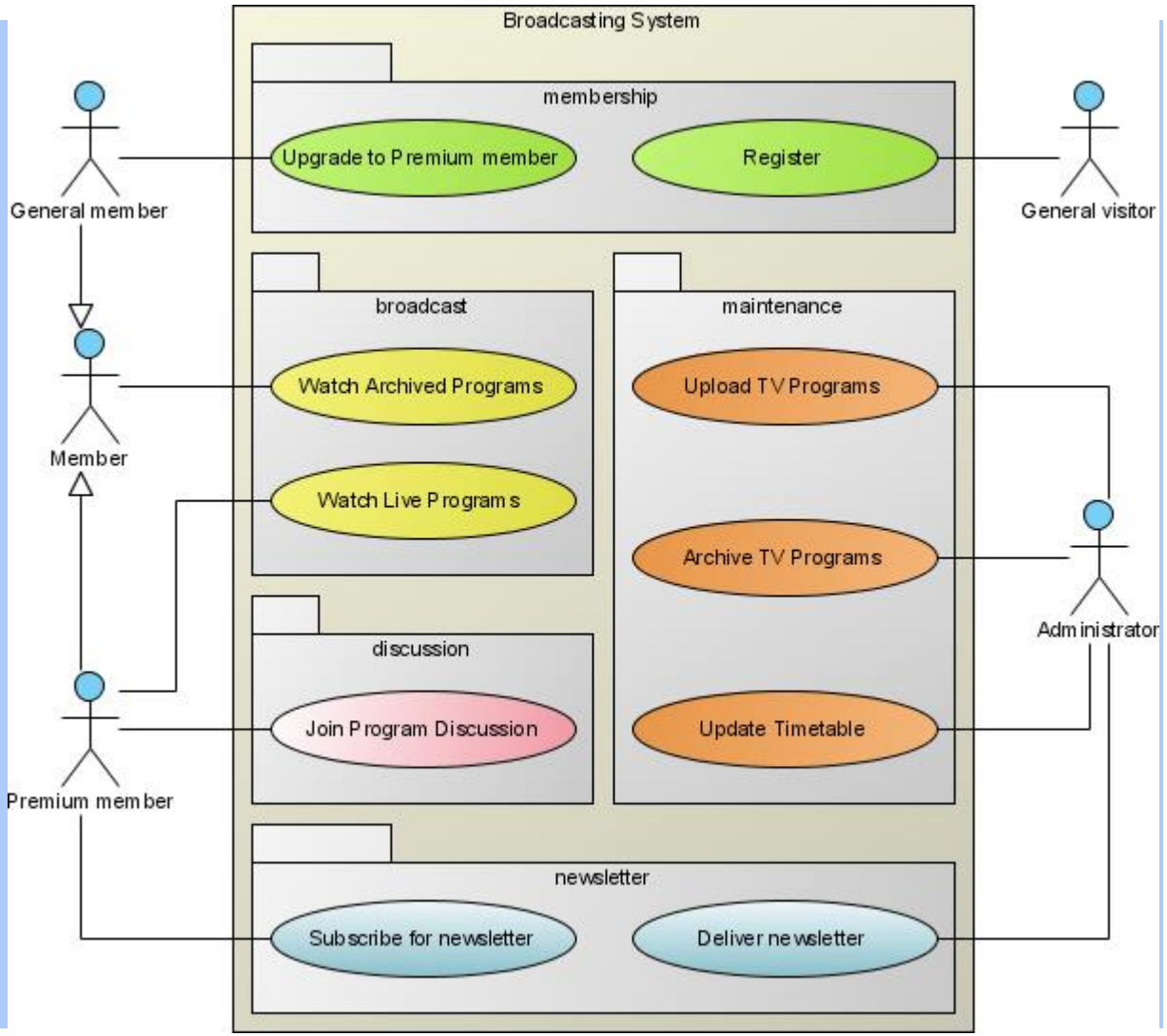
More Advanced Use Case Diagrams

- This Use Case Diagram belongs in a Design Document
- The Use Cases are still functions, but only two are exposed to the Actors
- The «include» relationship indicates which functions are included in other functions, i.e., the behavior of the included use case is inserted into the behavior of the including use case



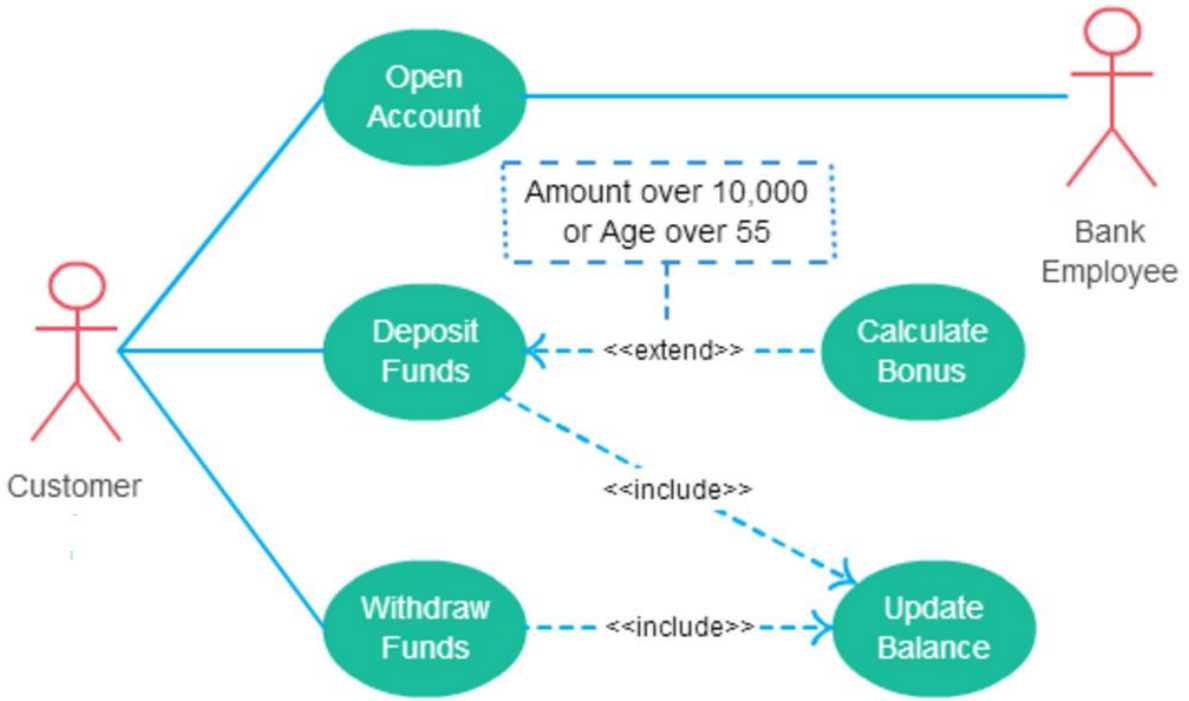
More Use Case Symbols (1)

- **Package** is an optional element that is extremely useful in complex diagrams. Packages are **used to group together use cases.**



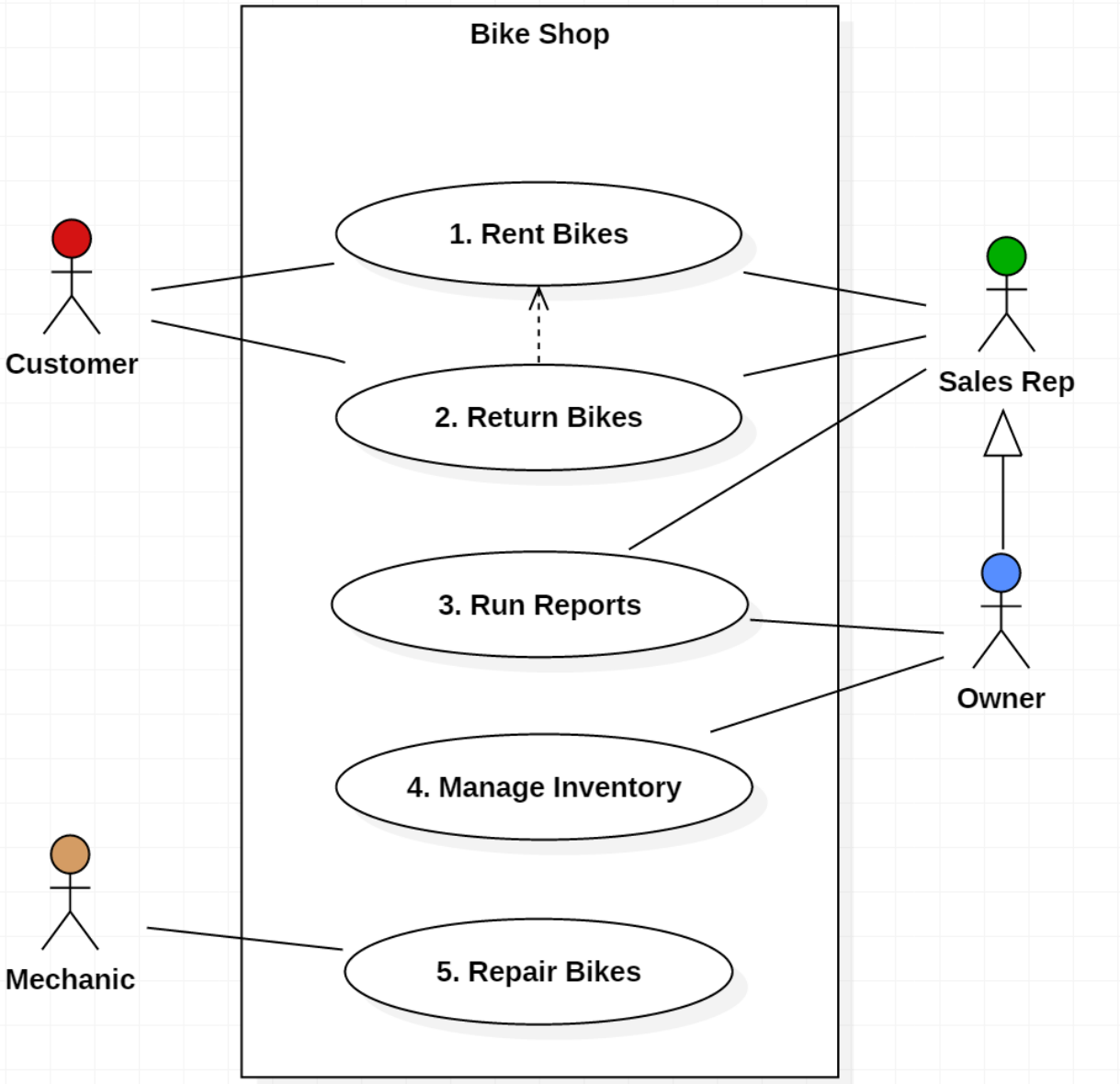
More Use Case Symbols (2)

- **Extend** is used instead of includes when one use case invokes another use case **conditionally. (see diagram)**



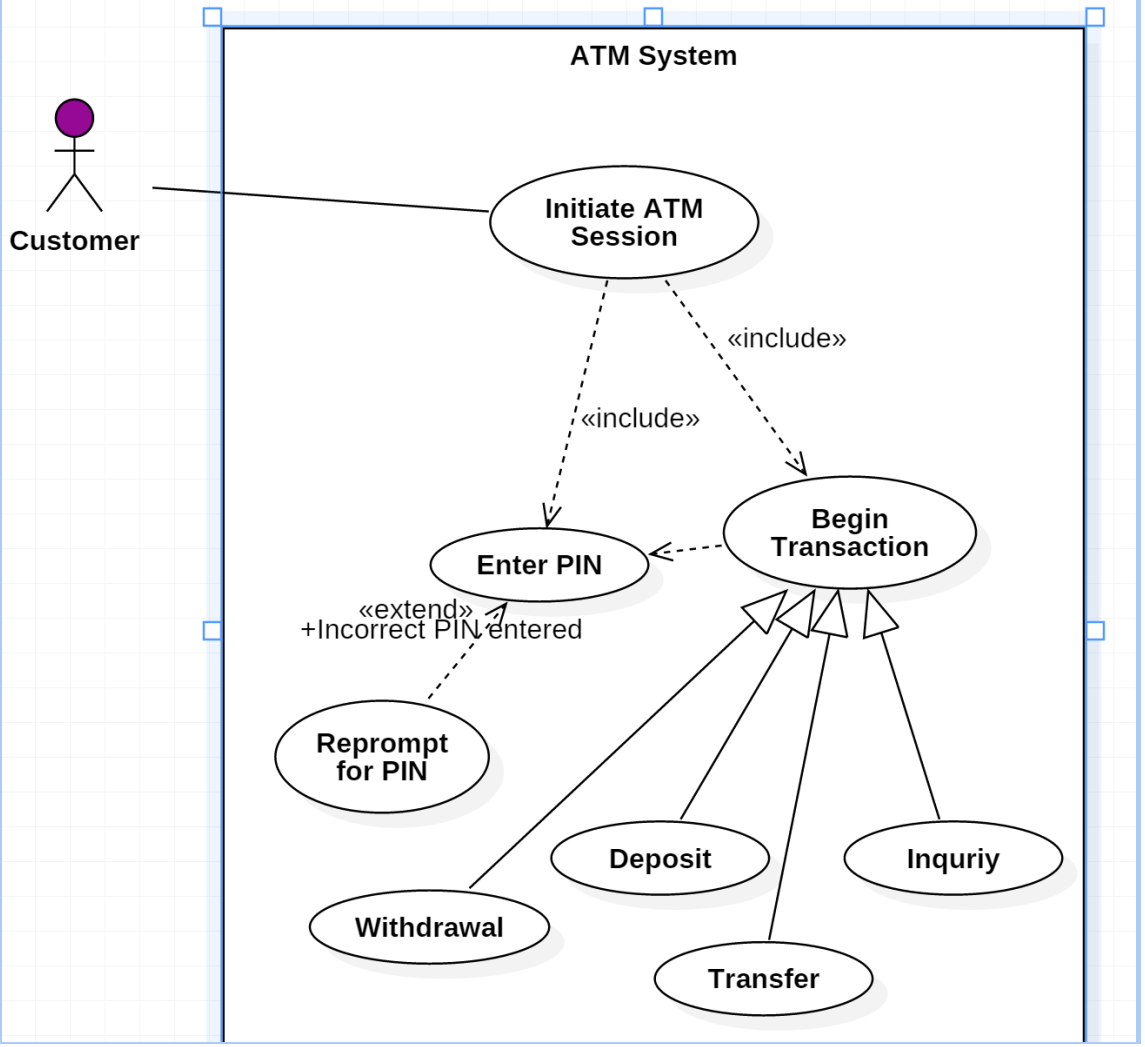
More Use Case Symbols (3)

- **Dependency**: the extending use case is dependent on the base use case as a one way dependency, i.e. the base use case doesn't need any reference to the extending use case in the sequence.
- You cannot return bikes that have not been first rented, but you can rent bikes without returning them



One More Example

- A different example of
  - Includes
  - Extends
  - Dependency
  - Generalization



from
Use Case Diagrams
to
Analysis Class Diagrams

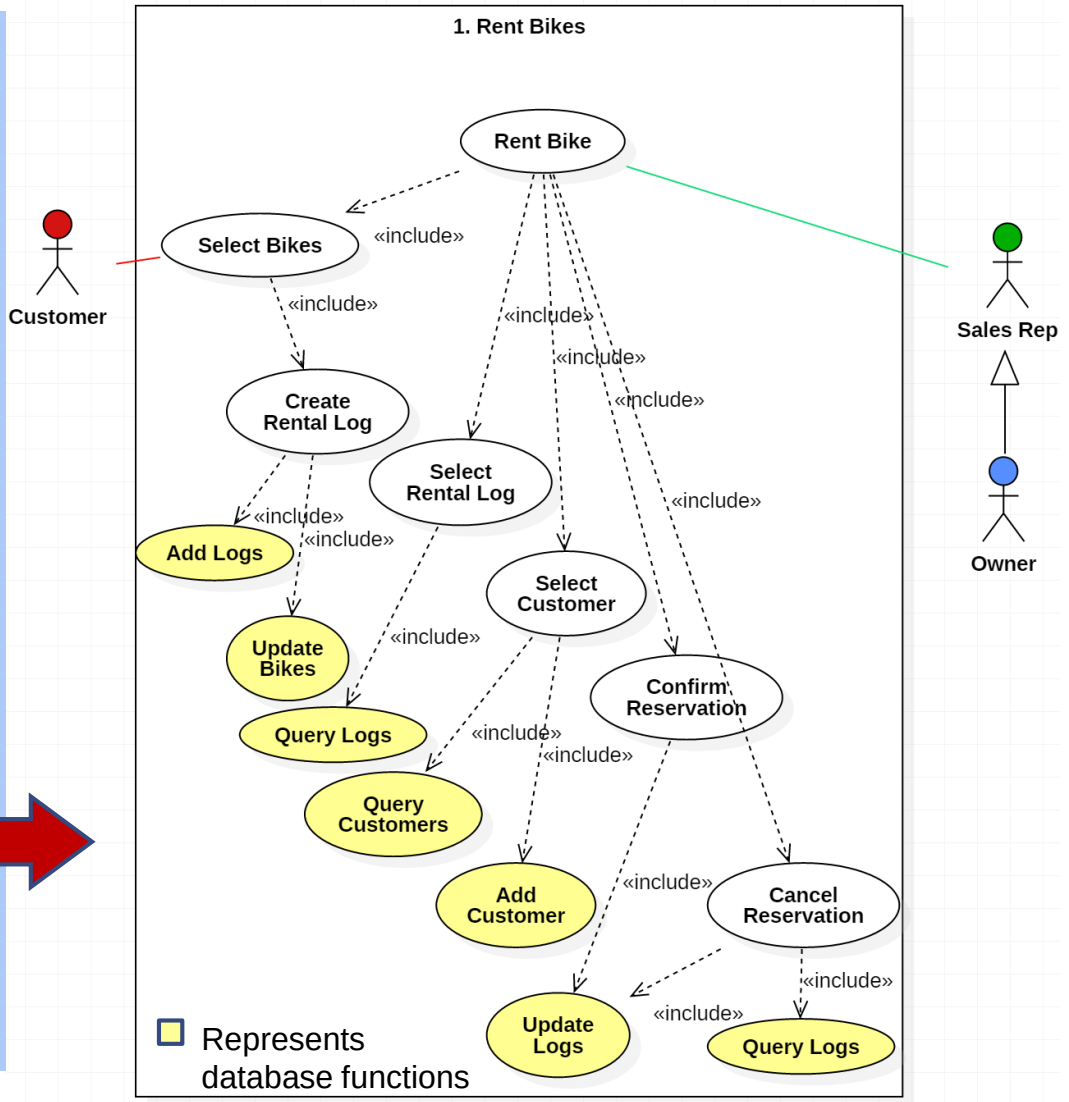
Analysis Class Diagram

- A robustness diagram is basically a simplified UML communication / collaboration diagram which uses the symbols / stereotypes
  - Actors. This is the same concept as actors on a UML use case diagram.
  - Boundary elements. These represent software elements such as screens, reports, HTML pages, or system interfaces that actors interact with. Also called interface elements.
  - Control elements. These serve as the glue between boundary elements and entity elements, implementing the logic required to manage the various elements and their interactions. Also known as process elements or simply as controllers. It is important to understand that you may decide to implement controllers within your design as something other than objects - many controllers are simple enough to be implemented as a method of an entity or boundary class for example.
  - Entity elements. These are entity types that are typically found in your conceptual model, such as Student and Seminar.
  - Use cases (optional). Because use cases can invoke other use cases you need to be able to depict this on your robustness diagrams. This functions as a cross reference.

The Analysis Class Diagram

- Analysis Class Diagrams are also known as
  - Robustness Diagrams
  - Ideal Object Model
- They resemble the more detailed Use Case Diagrams (see diagram) except the main objects in an Analysis Class Diagram are ***classes*** not ***use cases (or functions)***

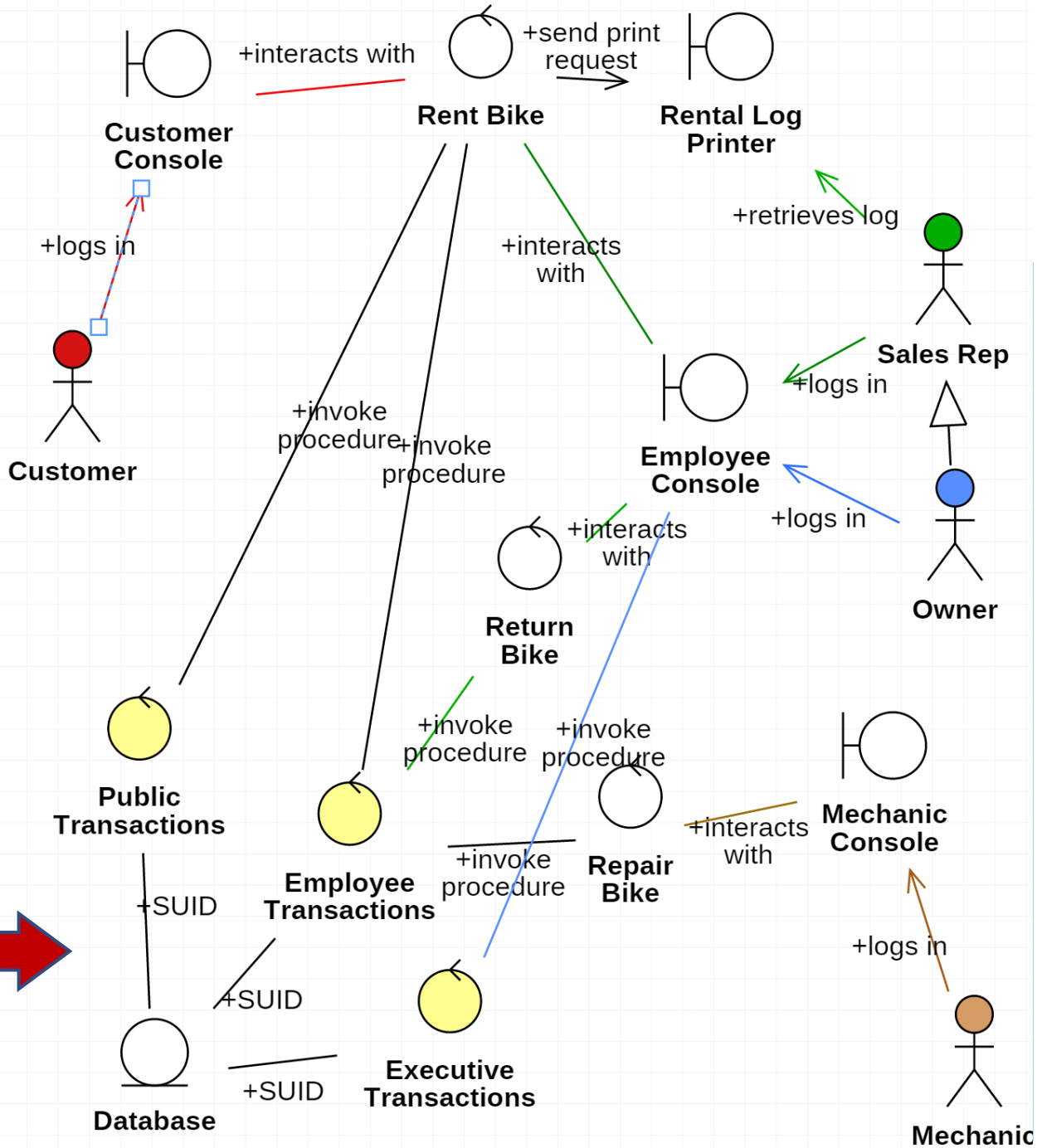
Use Case Diagram



Analysis Class Diagram

- No longer are functions (use cases) represented, but actual classes

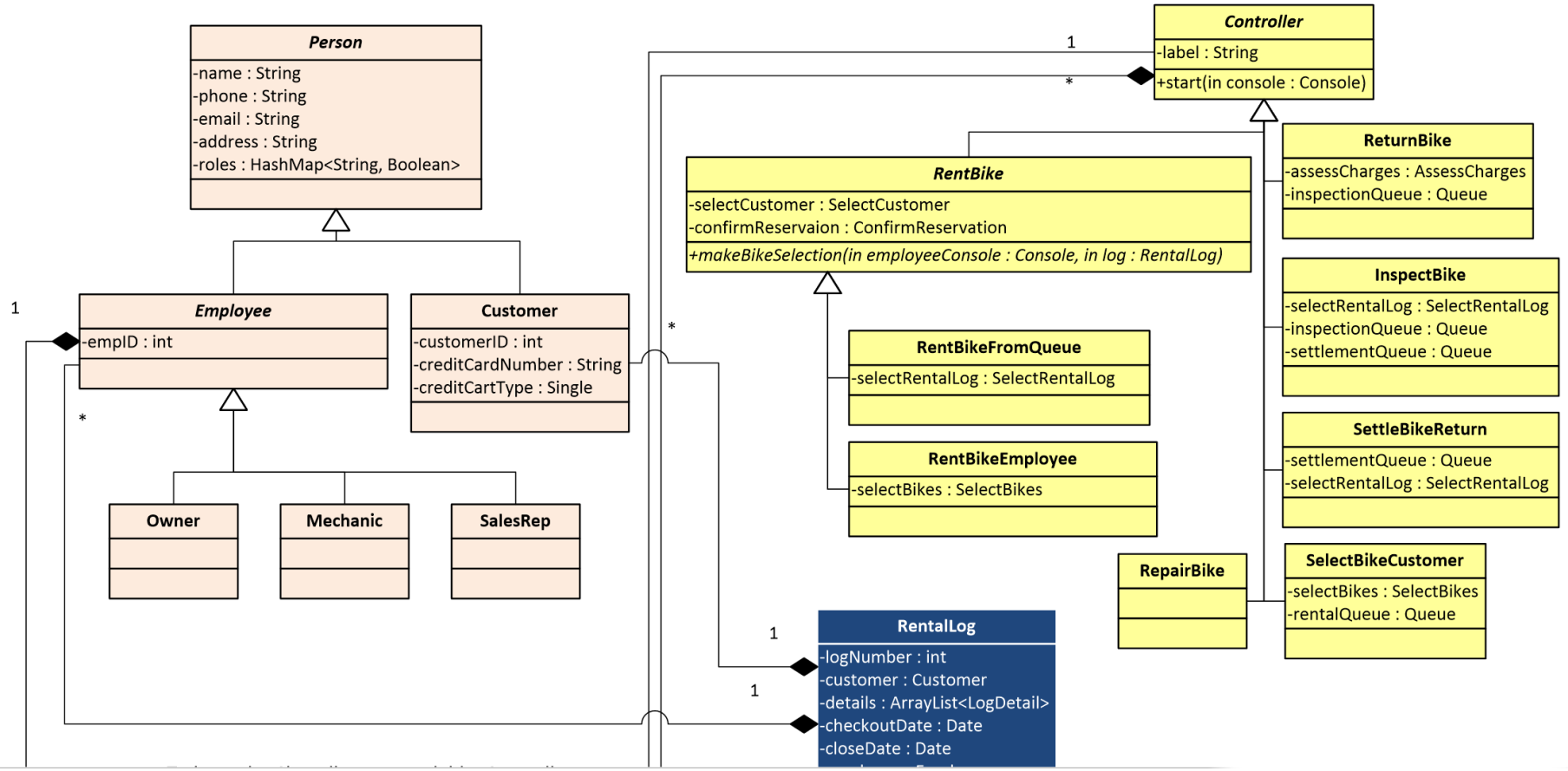
Analysis Class Diagram



Class and Object Diagrams

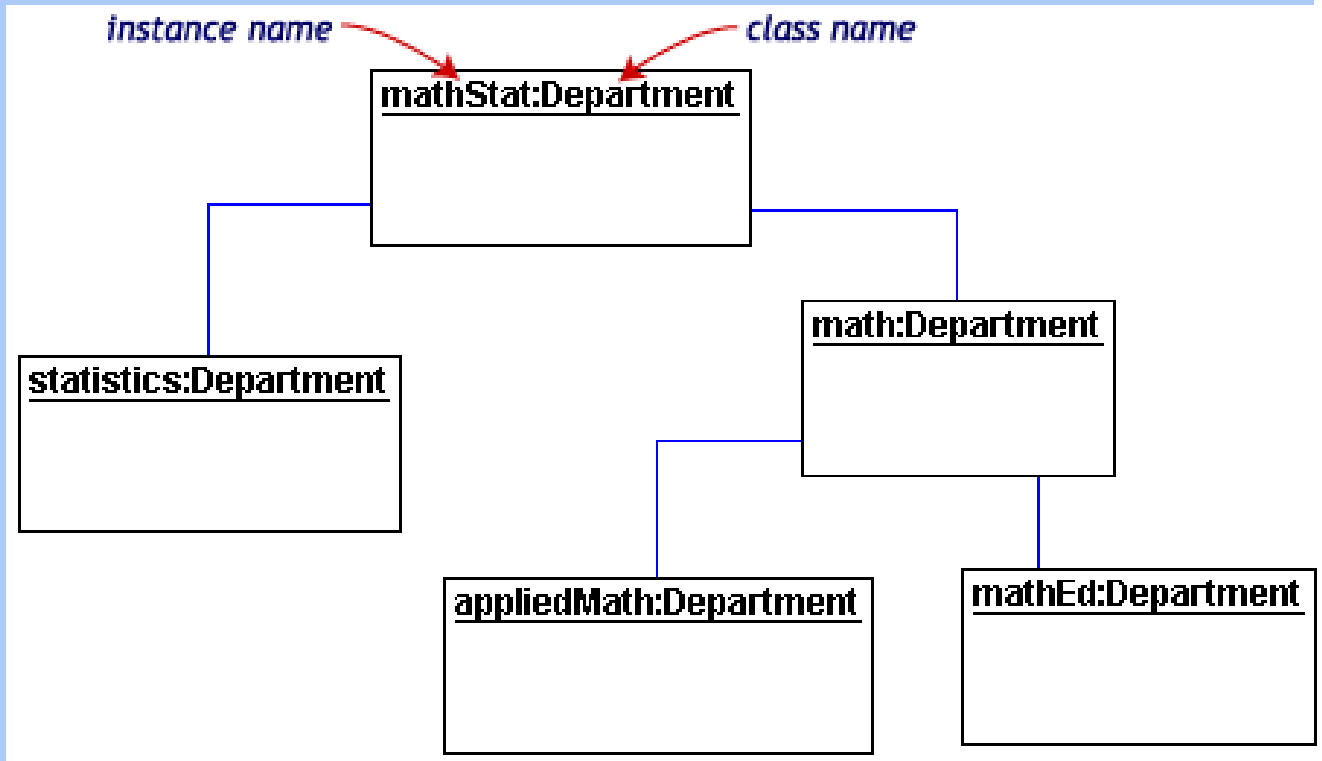
Class Diagrams

- **Class diagram: Describes the structure of a system by showing the classes of the systems, their attributes, and the relationships among the classes.**



Object Diagrams

- Object diagram: A view of the structure of example instances of modeled concepts at a specific time, possibly including property values.
- The syntax is similar to class diagrams, but objects are identified thorough underlined names.
- Objects can be identified by the couple object name and class name, by the simple object name, or by the class name they are instance of.



Activity Diagrams

Activity Diagrams

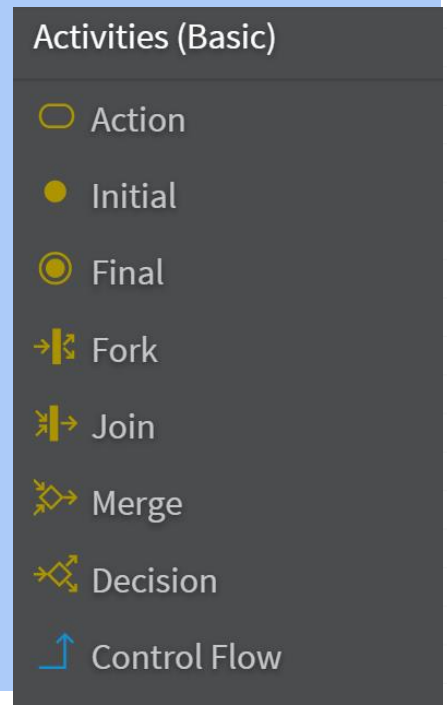
- There are many kinds of diagrams that depct a process, such as "flow charts" or "workflow diagrams"
- The Activity Diagram is one such diagram, except it has been formalized in UML

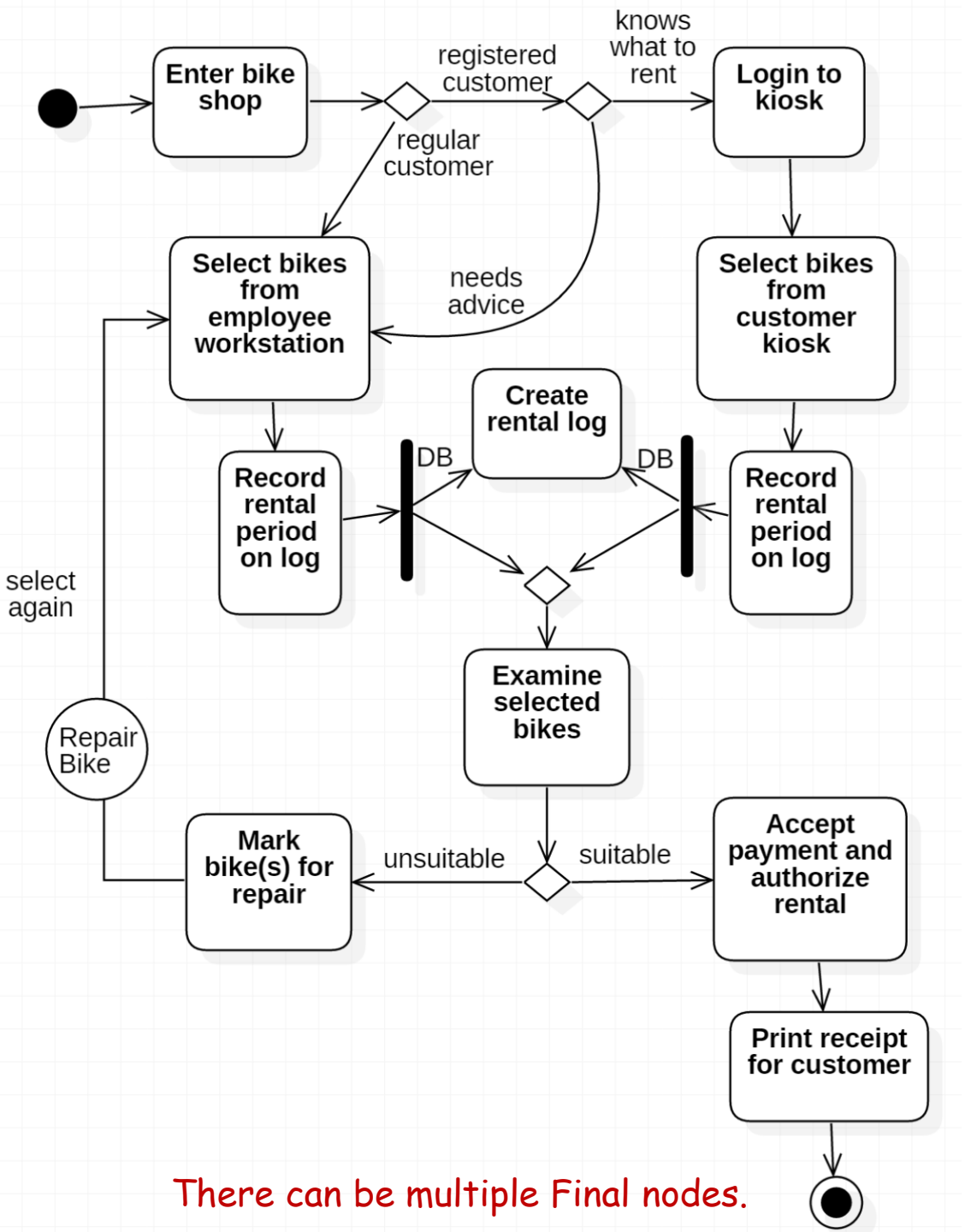
Activity Diagram

- Activity diagrams are used to model system behaviors, and the way in which these behaviors are related in an overall flow
  - of the system
  - of the data
  - of control for performing a task
- Describe the step-by-step workflows of activities to be performed in a system for reaching a specific goal.
- The logical paths a process follows, based on various conditions, concurrent processing, data access, interruptions and other logical path distinctions, are all used to construct a process, system or procedure.
- Diagrams are an oriented graph where nodes represent the activities.

Bike Shop

- Join vs Merge
  - Merge automatically advances the flow
  - Join requires that both Actions complete before flow is advanced, (example: document being approved by two dept heads)
- Connector
  - Circle designates another Activity Diagram

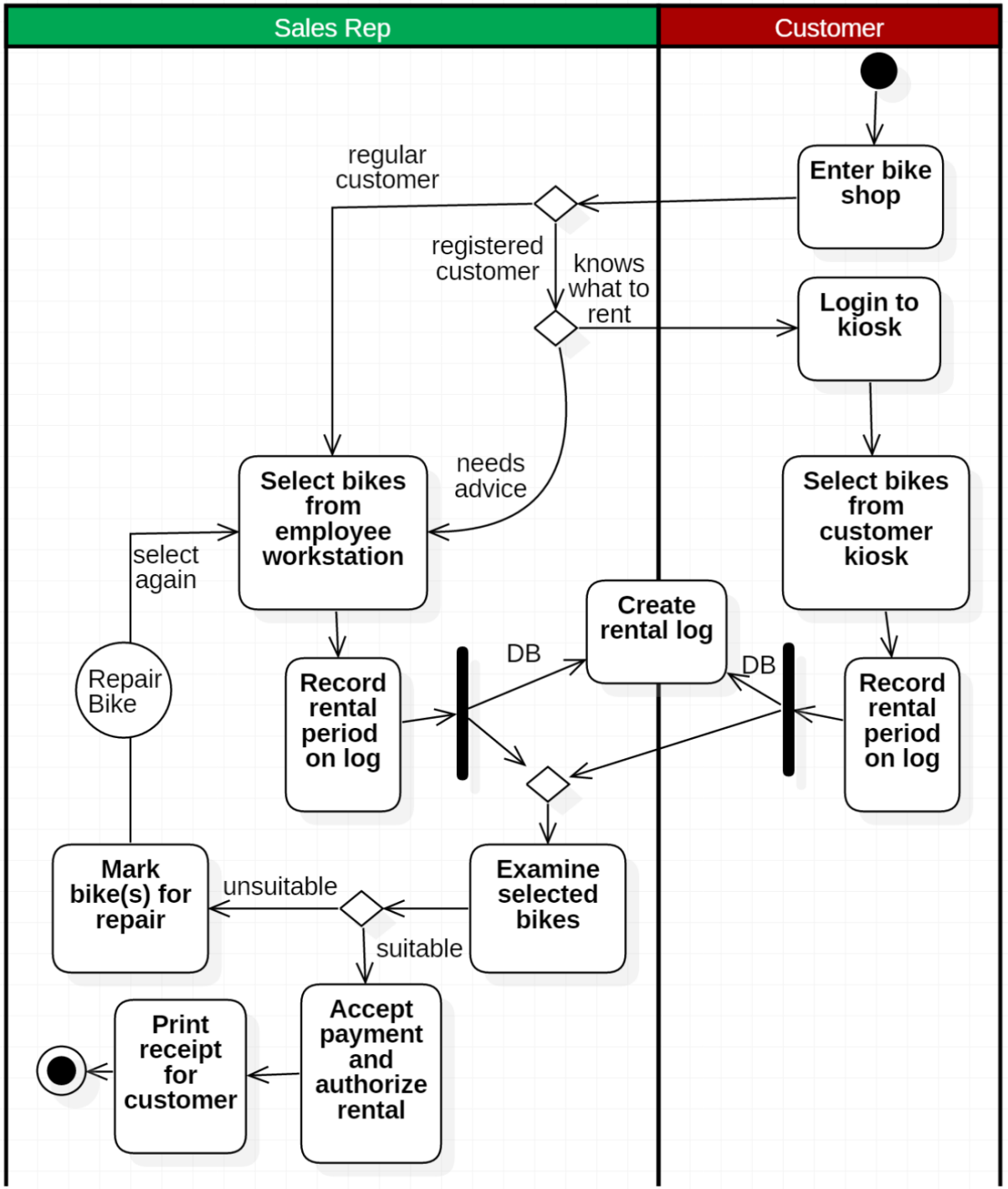




There can be multiple Final nodes.

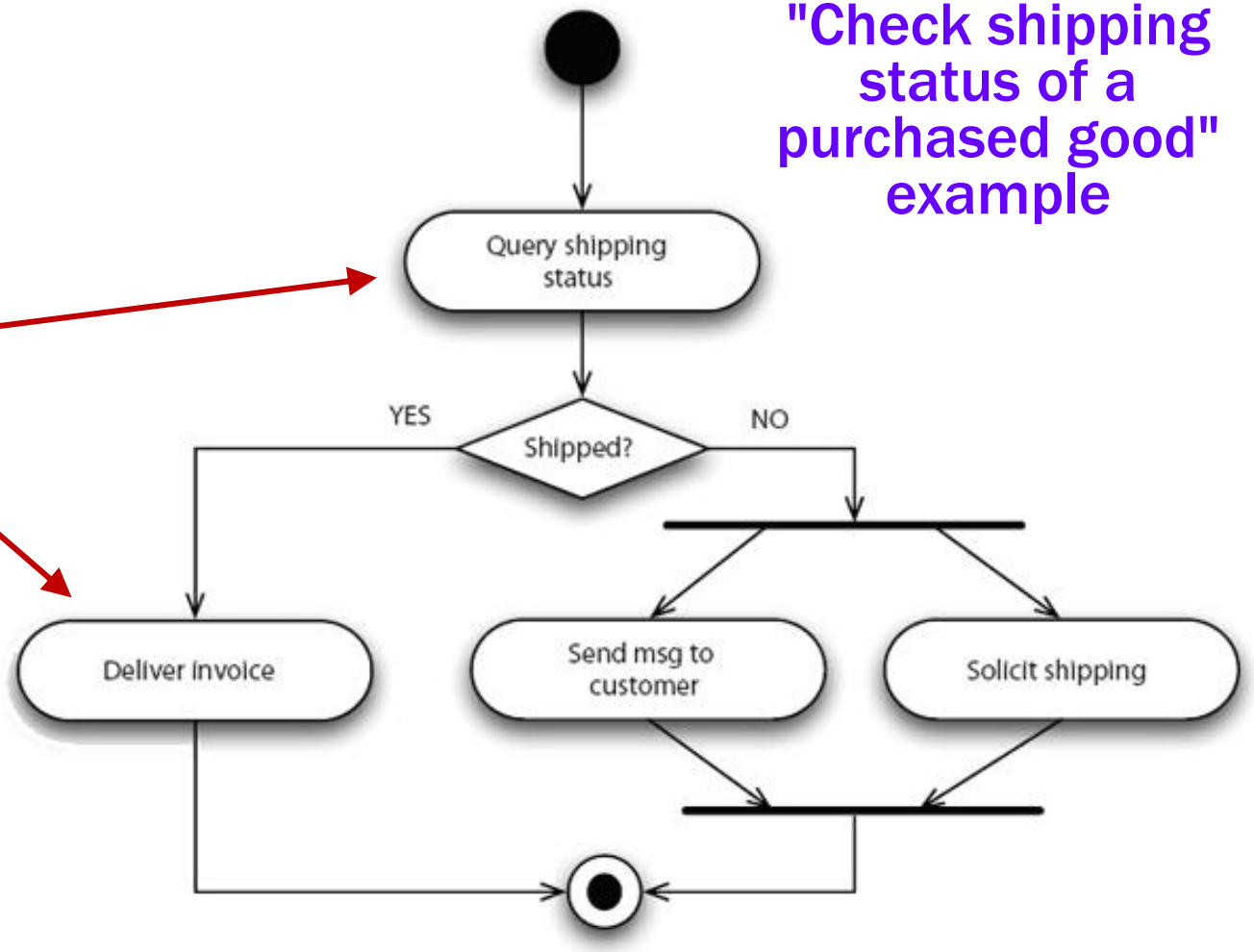
Swimlanes

- Swimlanes (horizontal or vertical) can clearly differentiate which Actors are performing which function in an Activity Diagram



Activity Diagrams are essentially flow charts

- Diagrams are an oriented graph where nodes represent the activities.



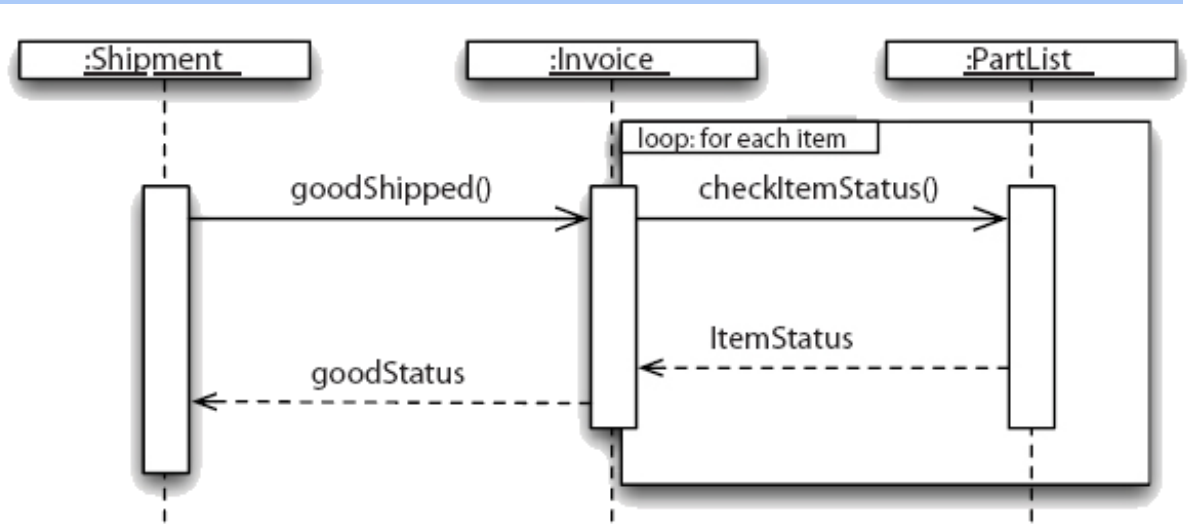
Interaction Diagrams

Interaction Diagrams

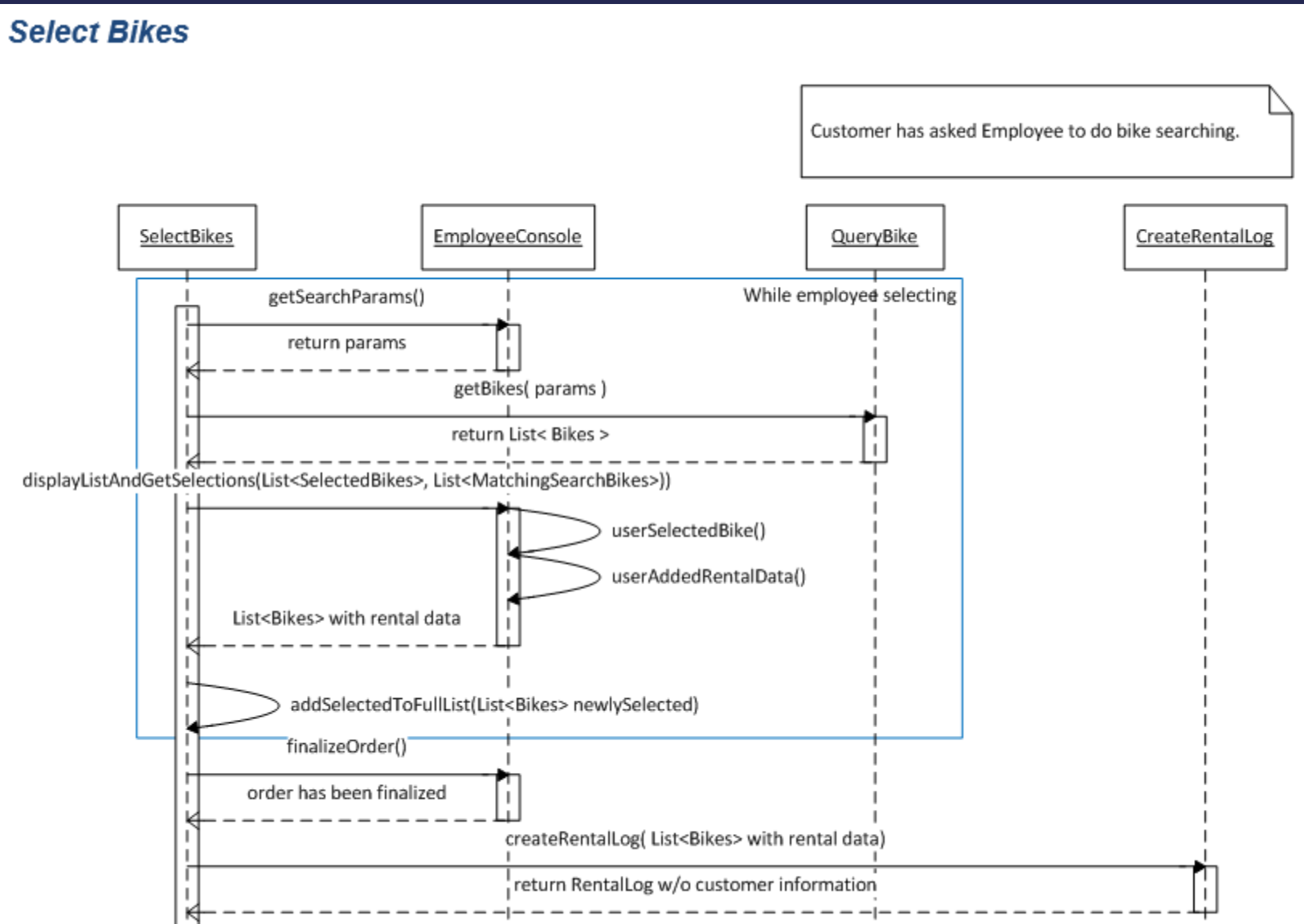
- UML defines two types of Interaction Diagram:
  - the Sequence Diagram
  - the Communication Diagram (formerly known as Collaboration Diagram)
- Sequence and communication diagrams both aim at describing the dynamic interactions between objects. The information you can describe are basically the same, but the two models have a different focus:

Interaction Diagrams
Sequence Diagram

- Sequence diagrams highlight more the temporal aspect of interactions
  - Show invocation and responses along a (vertical) timeline
  - Explicitly show the activation time of objects.
- Sequence diagrams show how objects communicate with each other in terms of a temporal sequence of messages.
- The time flow is the most visible aspect in these diagrams, as messages are sequenced according to a vertical timeline and also the lifespan of objects associated to those messages is reported.

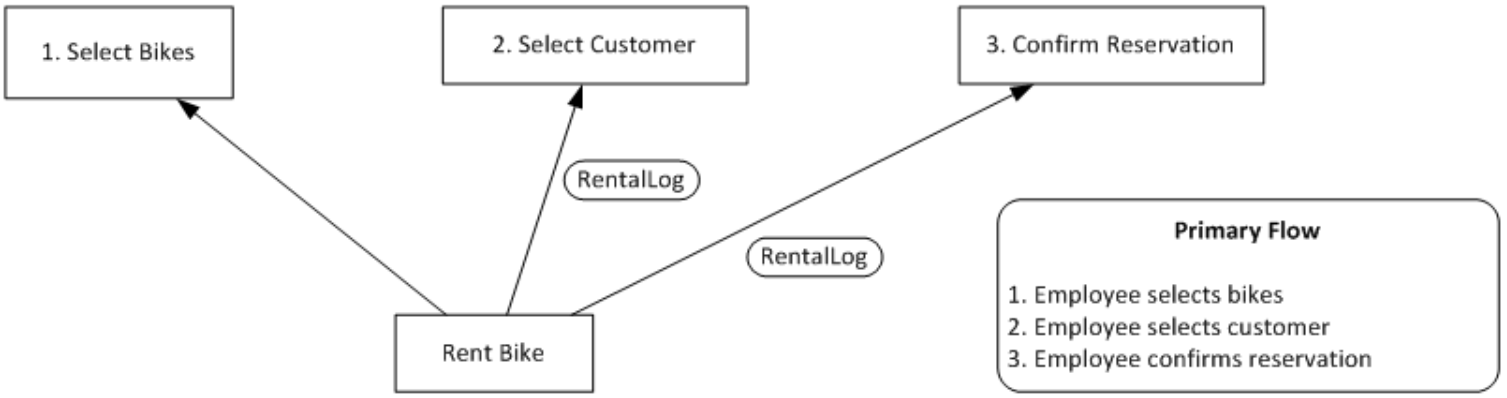


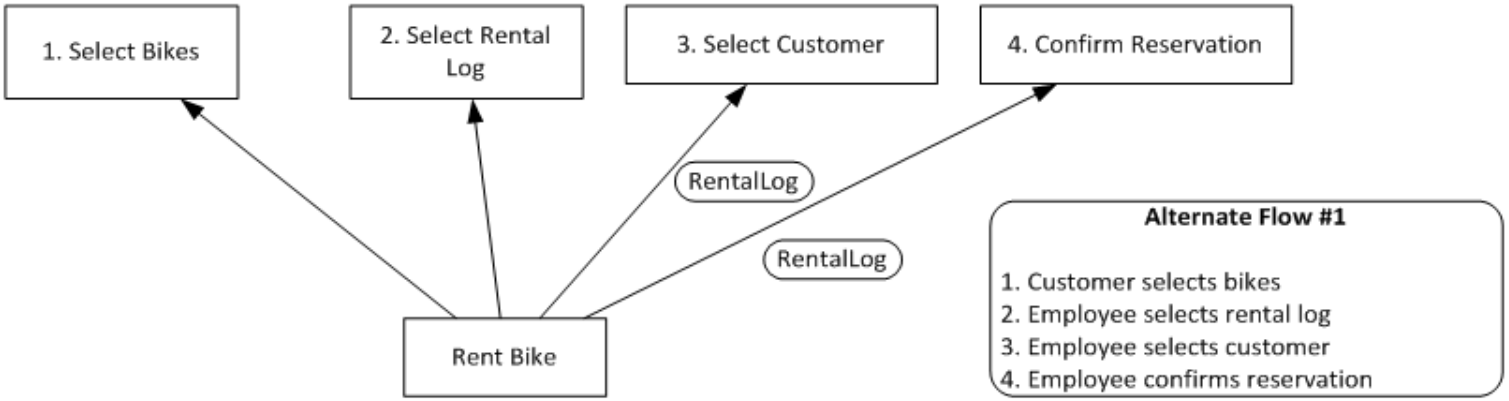
Interaction Diagrams
Sequence Diagram



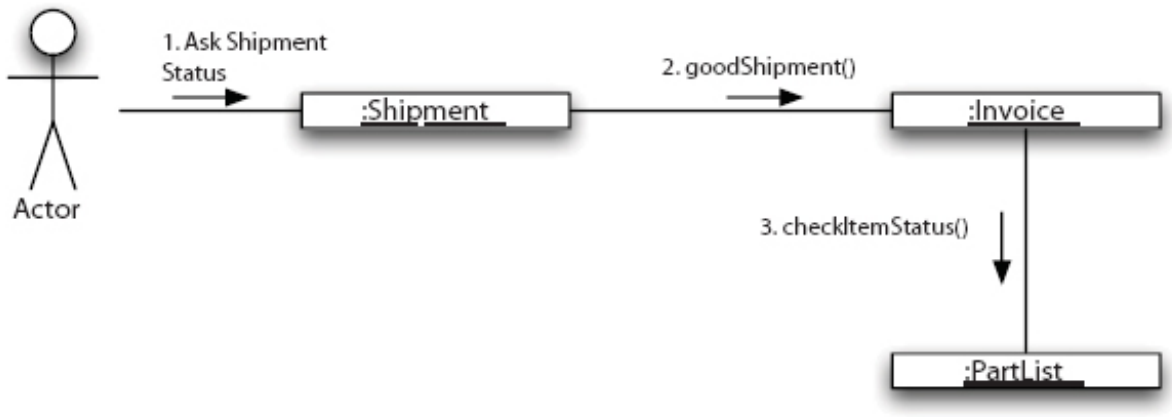
Interaction Diagrams
Communication Diagram

Rent Bike (Interaction Diagrams)

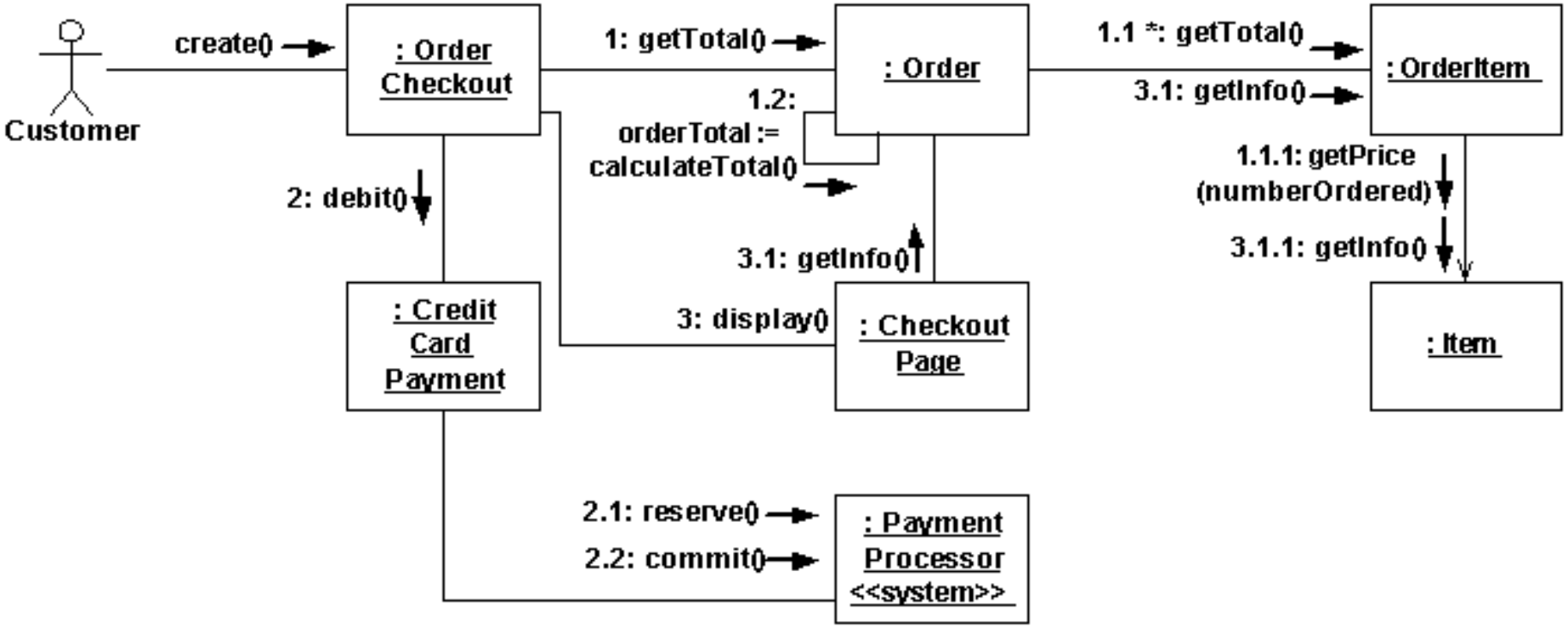




Interaction Diagrams
Communication Diagram

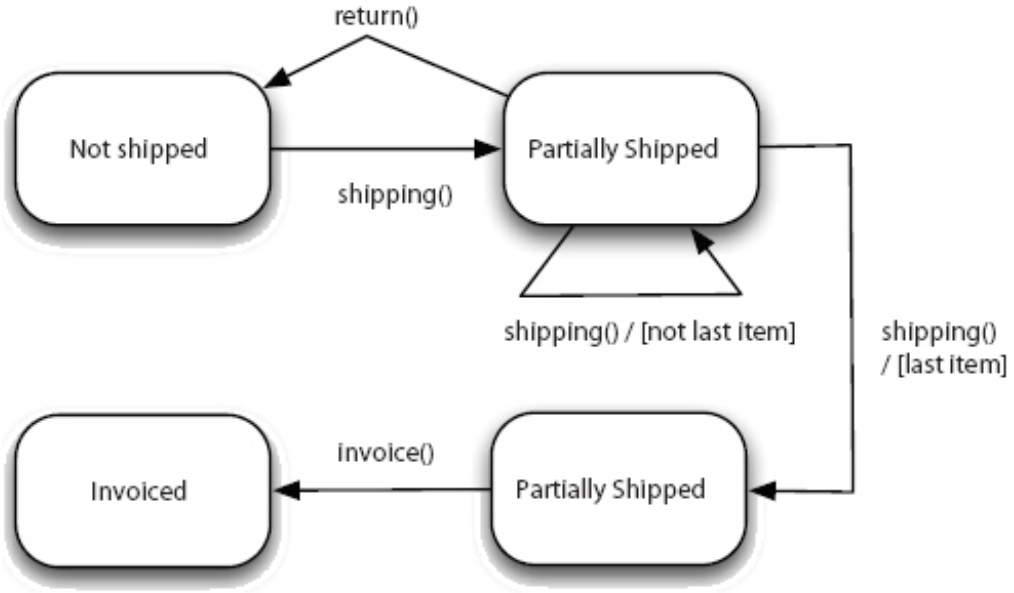


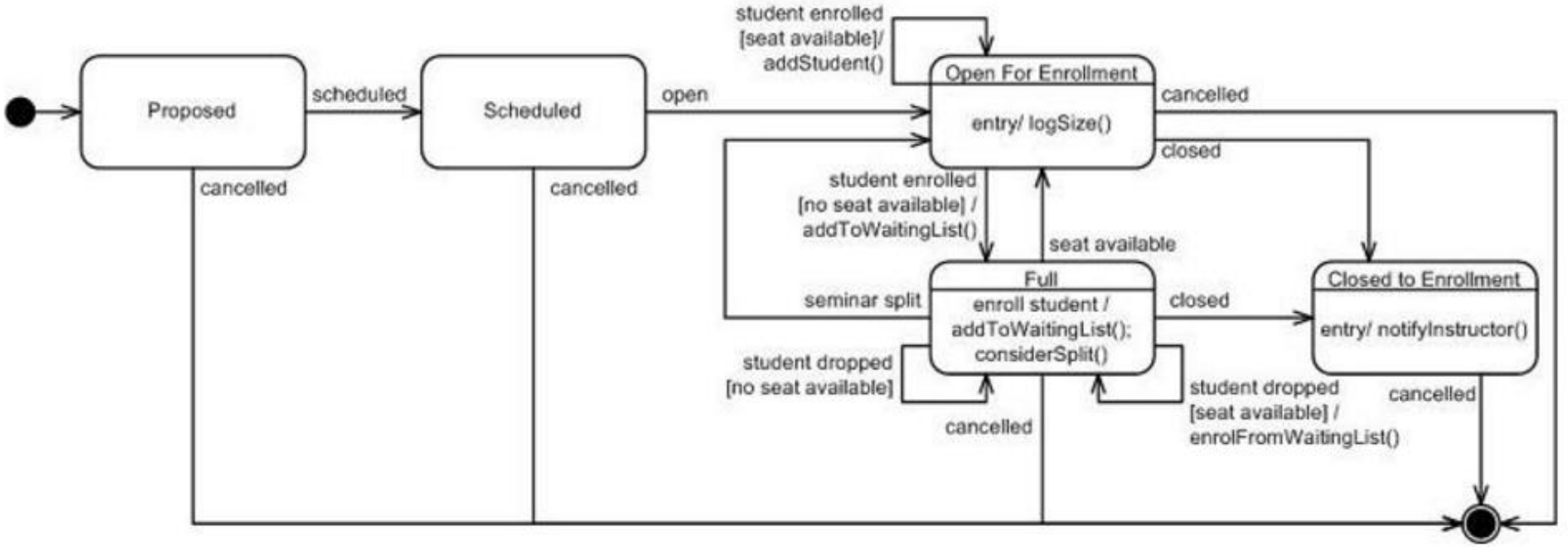
- More examples
  - Shipment of purchased goods
  - Customer checkout



State Diagrams

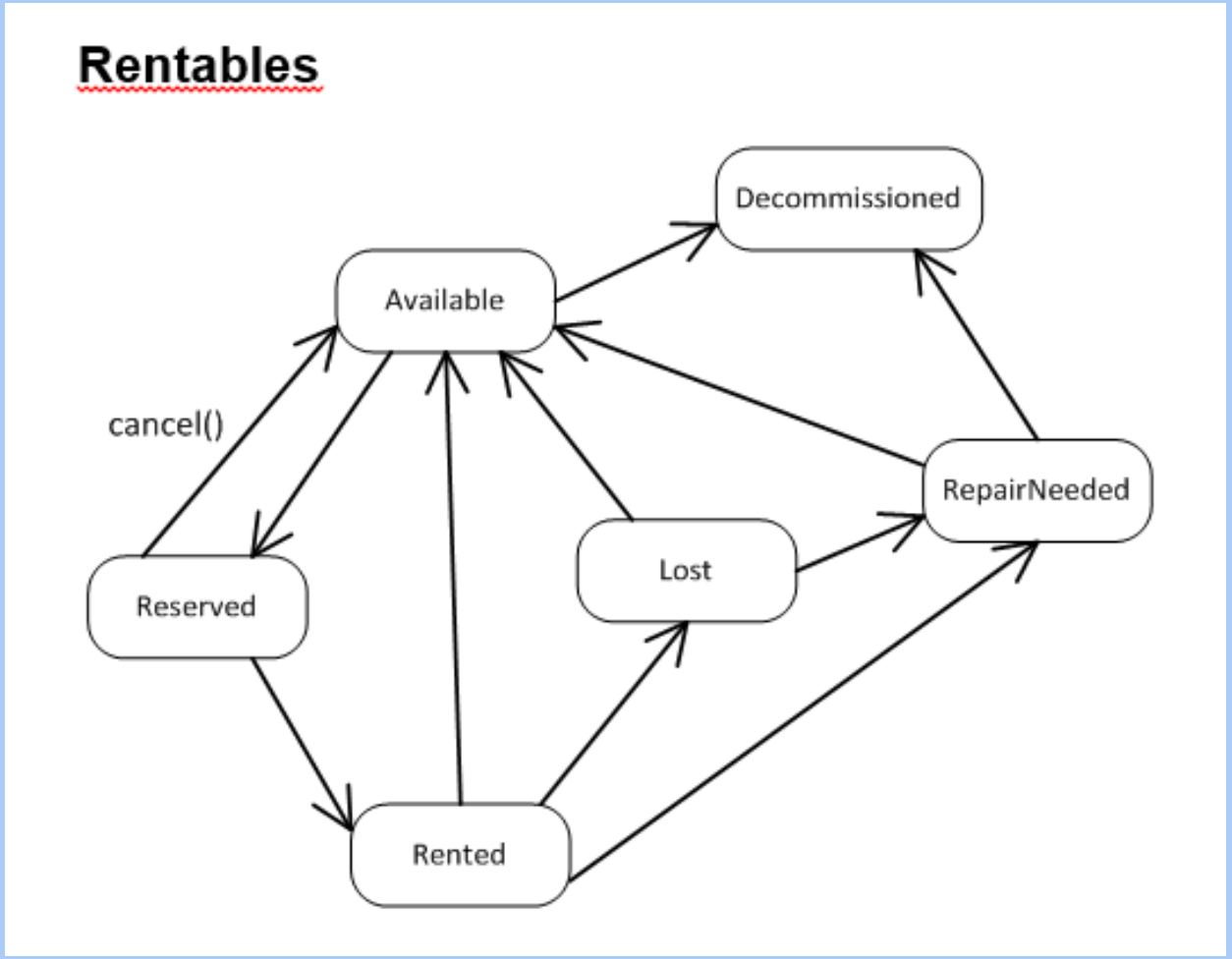
- Sufficiently complex objects warrant State Diagrams
- Examples
  - Purchased Good states
  - Seminar states





State Diagrams

- Bike shop example

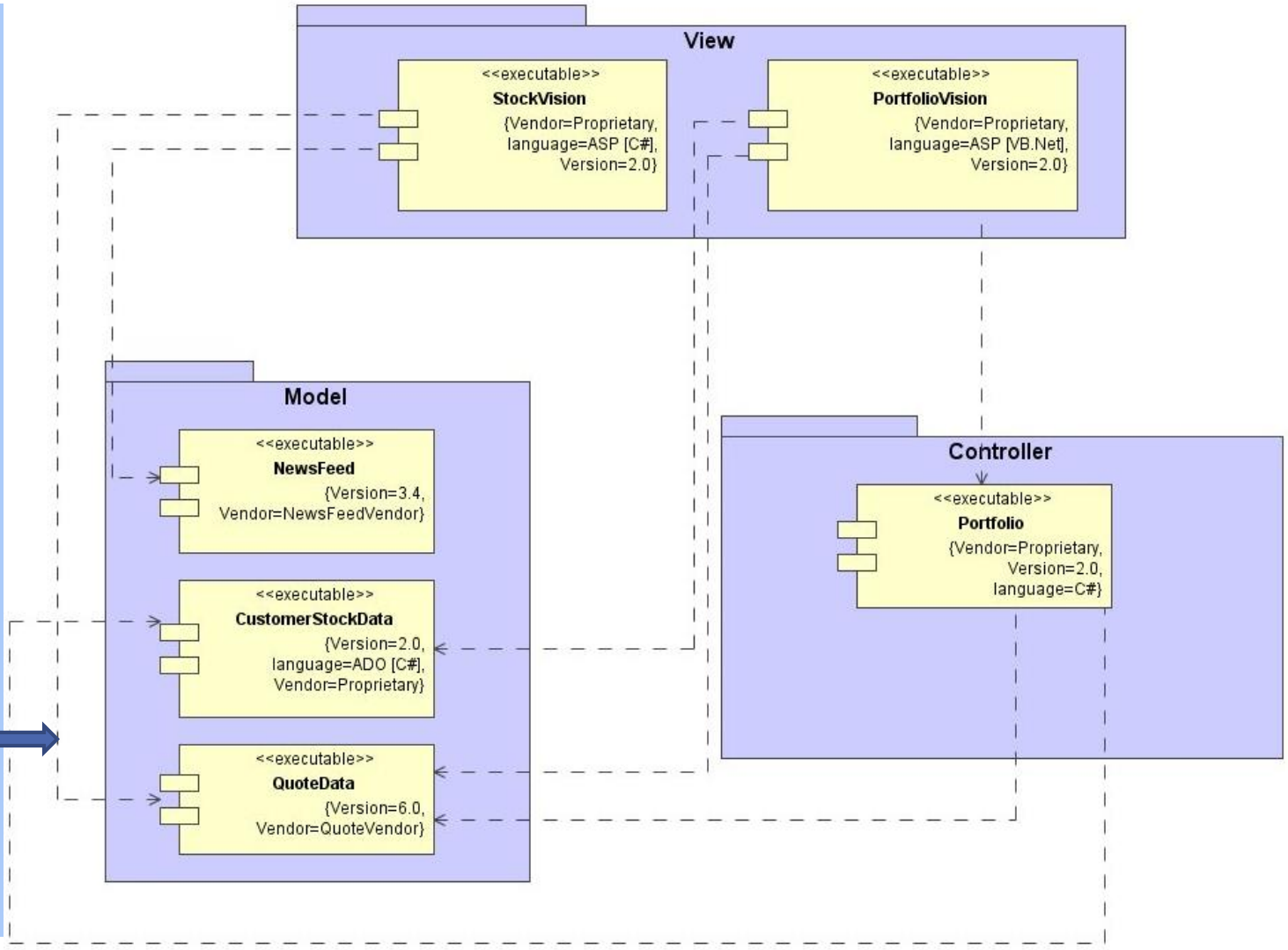


Component Diagrams

- Shows how the design is organised into logically related groups of objects.
- In the UML, these are shown using packages - an encapsulation construct. This is a logical model. The actual organisation of objects in the system may be different.

Subsystem Decomposition with Component Diagrams

- A subsystem is a replaceable part of the system with well-defined interfaces that encapsulates the state and behavior of its contained classes.
- UML component and package conventions



Deployment Diagrams

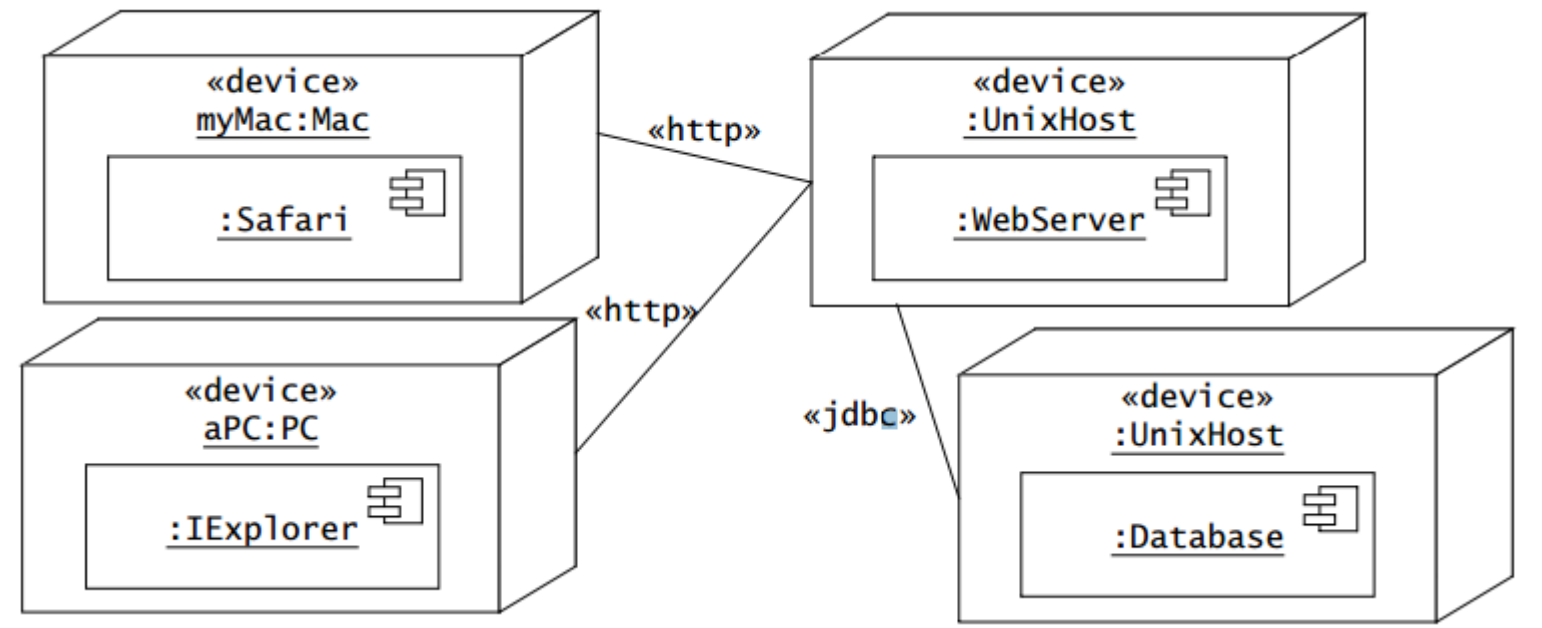
- Deployment diagrams are used to visualize the topology of the physical components of a system where the software components are deployed.
- Deployment diagrams are used by the system engineers and describe:
  - The physical components (hardware),
  - Their distribution
  - Their association
- The purpose of deployment diagrams:
  - Visualize hardware topology of a system.
  - Describe the hardware components used to deploy software components.
  - Describe runtime processing nodes.

Elements of a Deployment Diagram

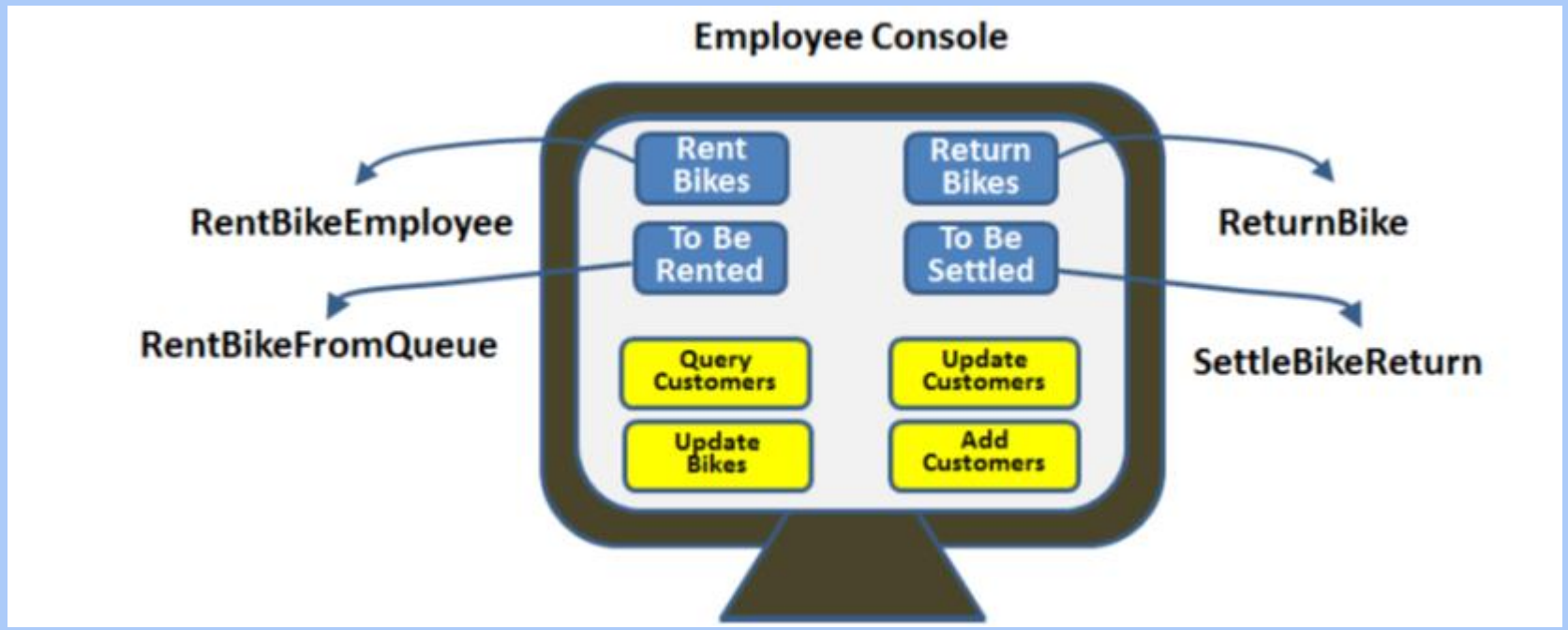
- UML deployment diagrams are used to depict the relationship among run-time components and nodes
- Components
  - Components are self-contained entities that provide services to other components or actors.
    - A Web server, for example, is a component that provides services to Web browsers.
    - A Web browser such as Safari is a component that provides services to a user.
- Nodes
  - A node is a physical device or an execution environment in which components are executed.
  - A system is composed of interacting run-time components that can be distributed among several nodes.
  - A node can contain another node, for example, a device can contain an execution environment.
  - Represented by boxes containing component icons.

An example

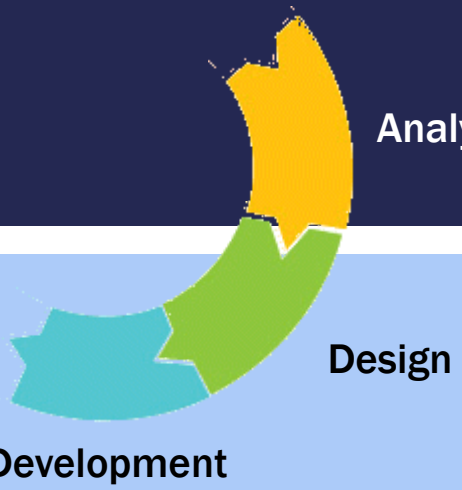
- Nodes are represented by boxes containing component icons.
- Nodes can be stereotyped to denote physical devices or execution environments.
- Communication paths between nodes are represented by solid lines. The protocol used by two nodes to communicate can be indicated with a stereotype on the communication path.



Any Diagram that Explains How the System Works is a Good One



Design Artifacts



Analysis

System Design

Object Design

Development

Nonfunctional Requirements

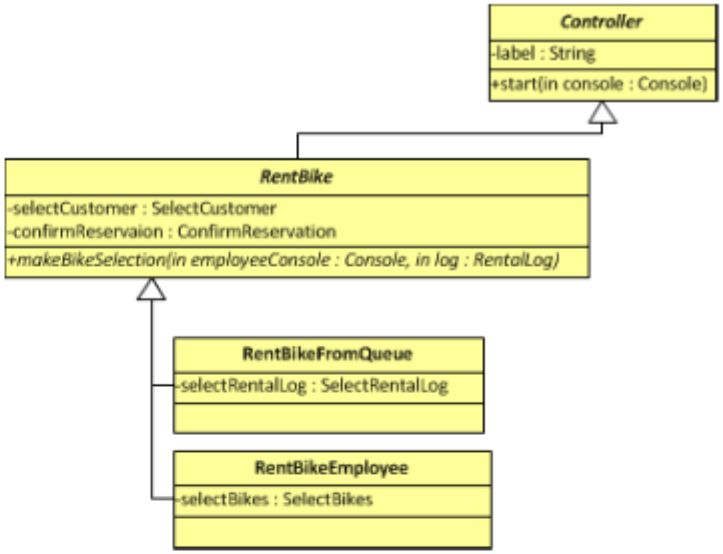
- Subsystem Decomposition Diagrams

DYNAMIC MODEL
- Sequence Diagrams
- State Machine Diagrams

ANALYSIS OBJECT MODEL
- Class Diagrams
- ER Diagrams and EER Diagrams

Using the Template Pattern for RentBike

The reuse of Subcontrollers lends itself very well to the **Template** pattern. An abstract Controller RentBike was created. As the behavior of selecting a customer and confirming a reservation is the same regardless of whether the rental was employee- or customer-initiated, those subcontrollers are properties of RentBike.



RentBike's start() method is the template method. It would have code such as:

```
public void start() {
      RentalLog log = makeBikeSelection(employeeConsole, log);
      log = selectCustomer.execute(employeeConsole, log);
      log = confirmReservation.execute(employeeConsole, log);
      employeeConsole.reset();
```

Explain any Design Patterns used

The two concrete subclasses would define "makeBikeSelection(employeeConsole, log)" differently.

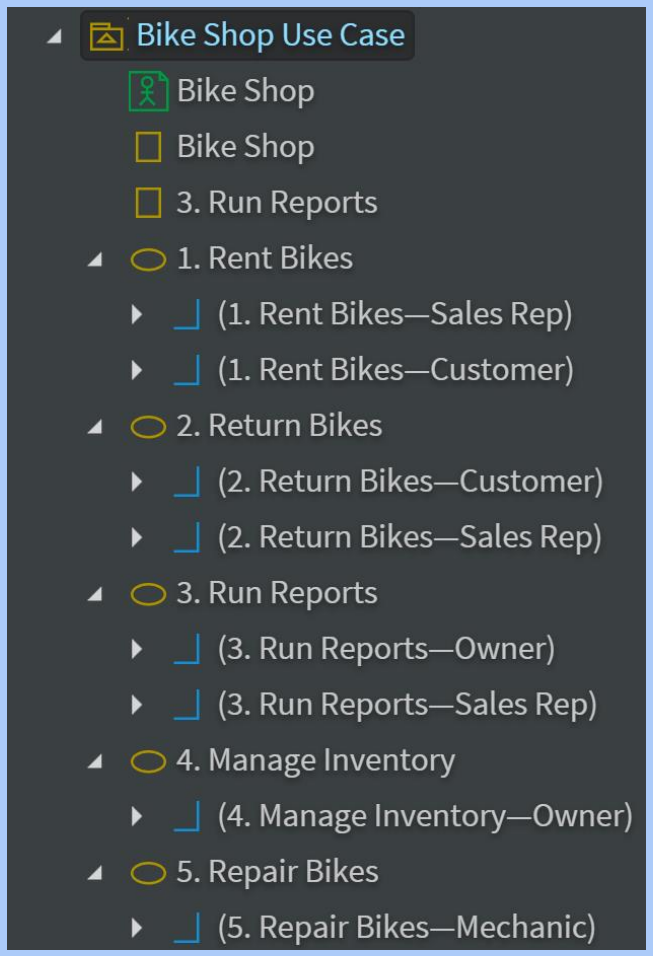
```
// Subclass RentBikeFromQueue
public void makeBikeSelection(employeeConsole, log) {
      return selectRentalLog.execute(employeeConsole, log);
}

// Subclass RentBikeEmployee
public void makeBikeSelection(employeeConsole, log) {
      return selectBikes.execute(employeeConsole, log);
}
```

StarUML Tips

StarUML tips

- To move a selected object use Ctrl-*<arrow key>*
- For Use Case Diagrams drag Association from Use Case to Actor
- For Analysis Class Diagrams create a Use Case Diagram if you want Actors. Copy Actors to a Class Diagram. Utilize the "Classes (Basic)" and the "Robustness" icon libraries
- Activity Diagram Forks and Joins will be vertical or horizontal depending on how you draw them

- StarUML model explorer

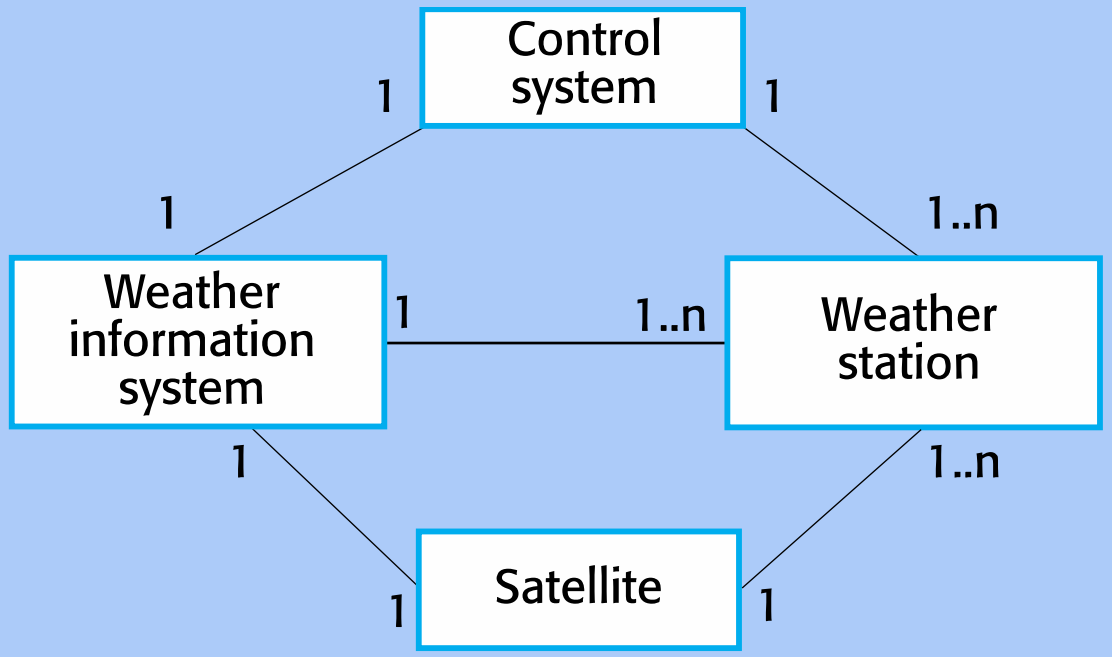


The Weather Station Example

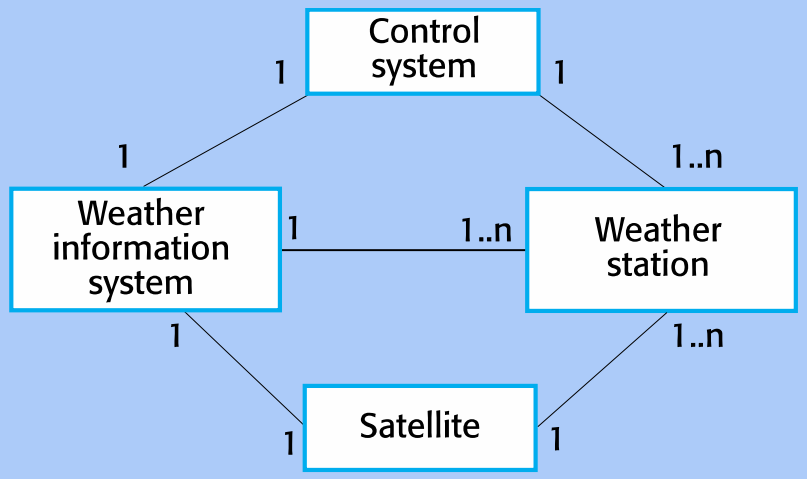
An object-oriented design process, implemented in stages

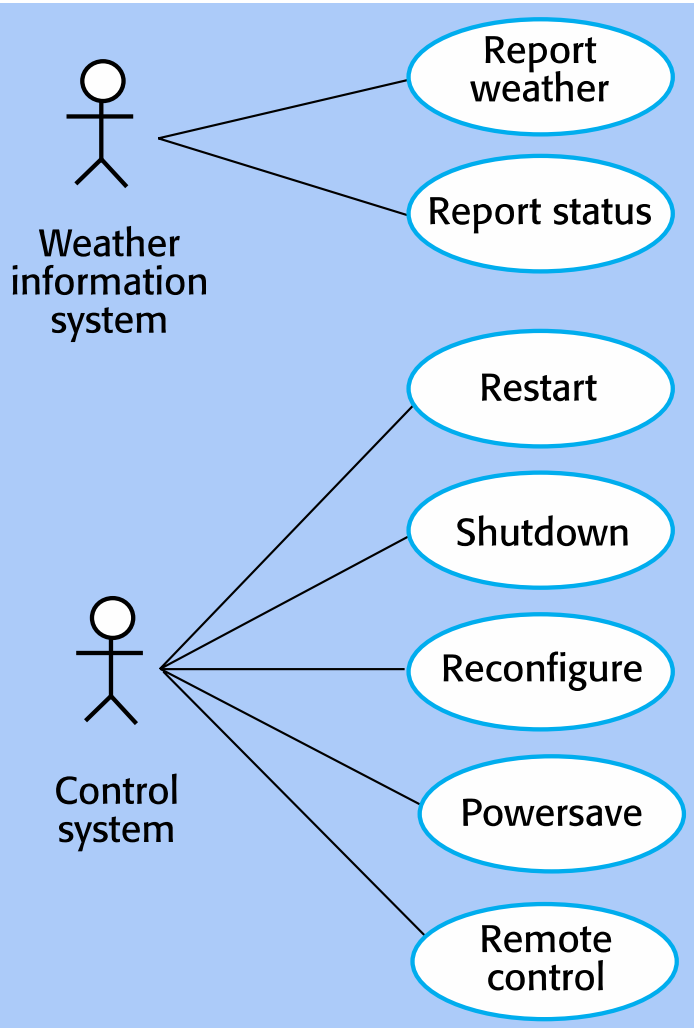
- Structured object-oriented design processes involve developing a number of different system models.
- They require a lot of effort for development and maintenance of these models and, for small systems, this may not be cost-effective.
- However, for large systems developed by different groups design models are an important communication mechanism.
- There are a variety of different object-oriented design processes that depend on the organization using the process.
- Common activities in these processes include:
  - Define the context and modes of use of the system;
  - Design the system architecture;
  - Identify the principal system objects;
  - Develop design models;
  - Specify object interfaces.

System context for the weather station



Weather station use cases





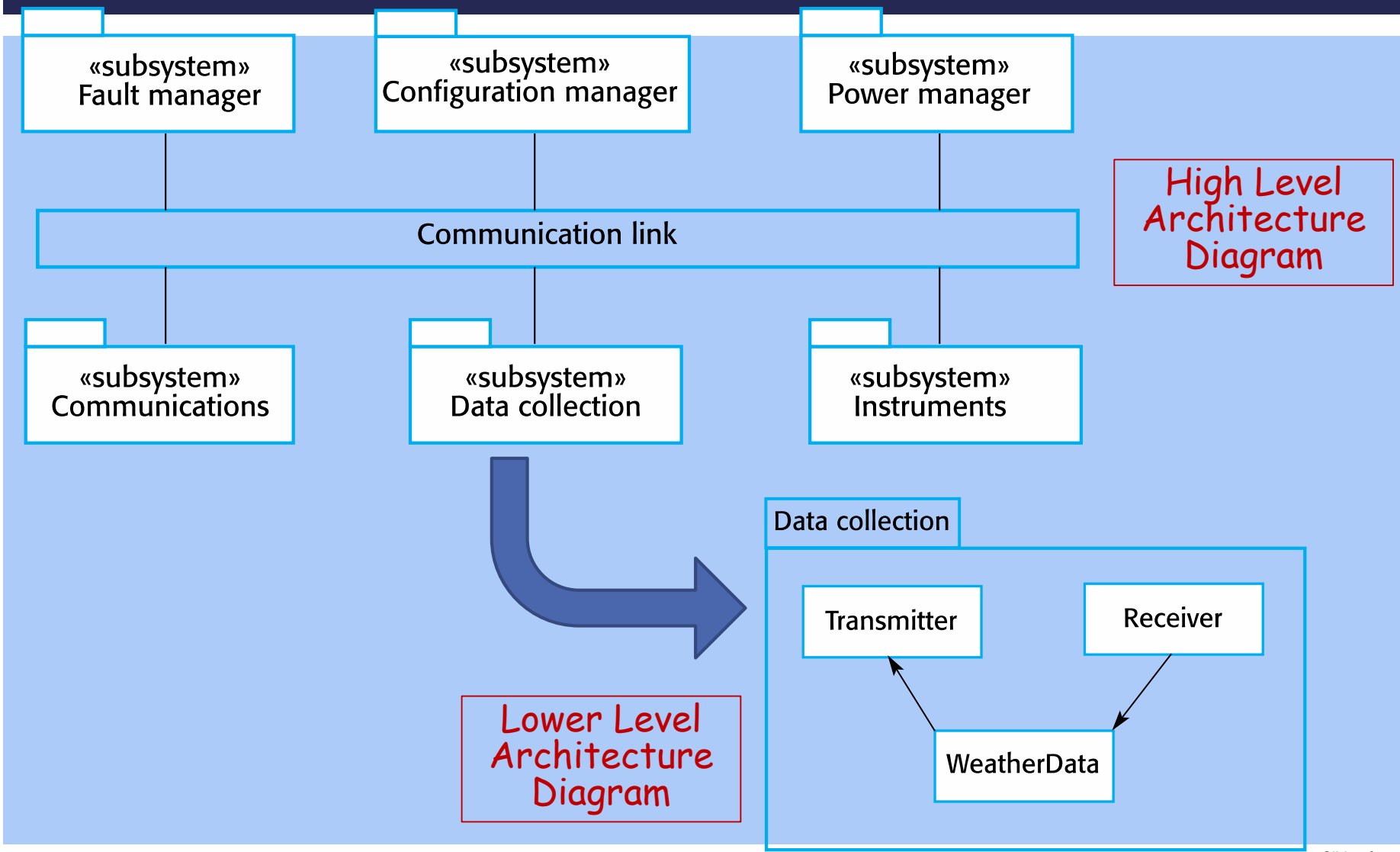
Use case description—Report weather

System	Weather station
Use case	Report weather
Actors	Weather information system, Weather station
Description	The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather information system. The data sent are the maximum, minimum, and average ground and air temperatures; the maximum, minimum, and average air pressures; the maximum, minimum, and average wind speeds; the total rainfall; and the wind direction as sampled at five-minute intervals.
Stimulus	The weather information system establishes a satellite communication link with the weather station and requests transmission of the data.
Response	The summarized data is sent to the weather information system.
Comments	Weather stations are usually asked to report once per hour but this frequency may differ from one station to another and may be modified in the future.

Architectural design

- Once interactions between the system and its environment have been understood, you use this information for designing the system architecture.
- You identify the major components that make up the system and their interactions, and then may organize the components using an architectural pattern such as a layered or client-server model.
- The weather station is composed of independent subsystems that communicate by broadcasting messages on a common infrastructure.

Architectures of the Weather Station



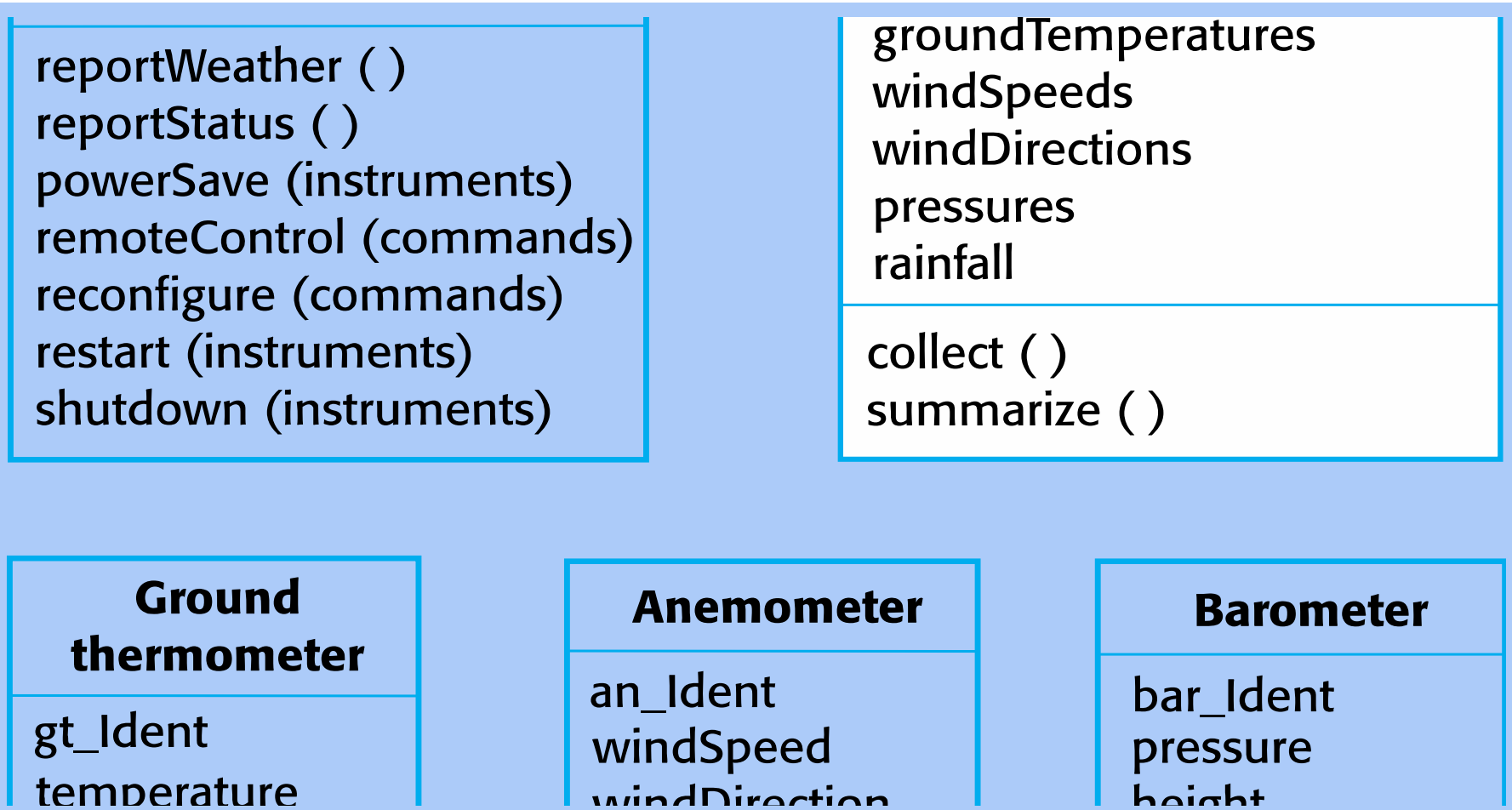
Approaches to identification

- Use a grammatical approach based on a natural language description of the system.
- Base the identification on tangible things in the application domain.
- Use a behavioural approach and identify objects based on what participates in what behaviour.
- Use a scenario-based analysis. The objects, attributes and methods in each scenario are identified.

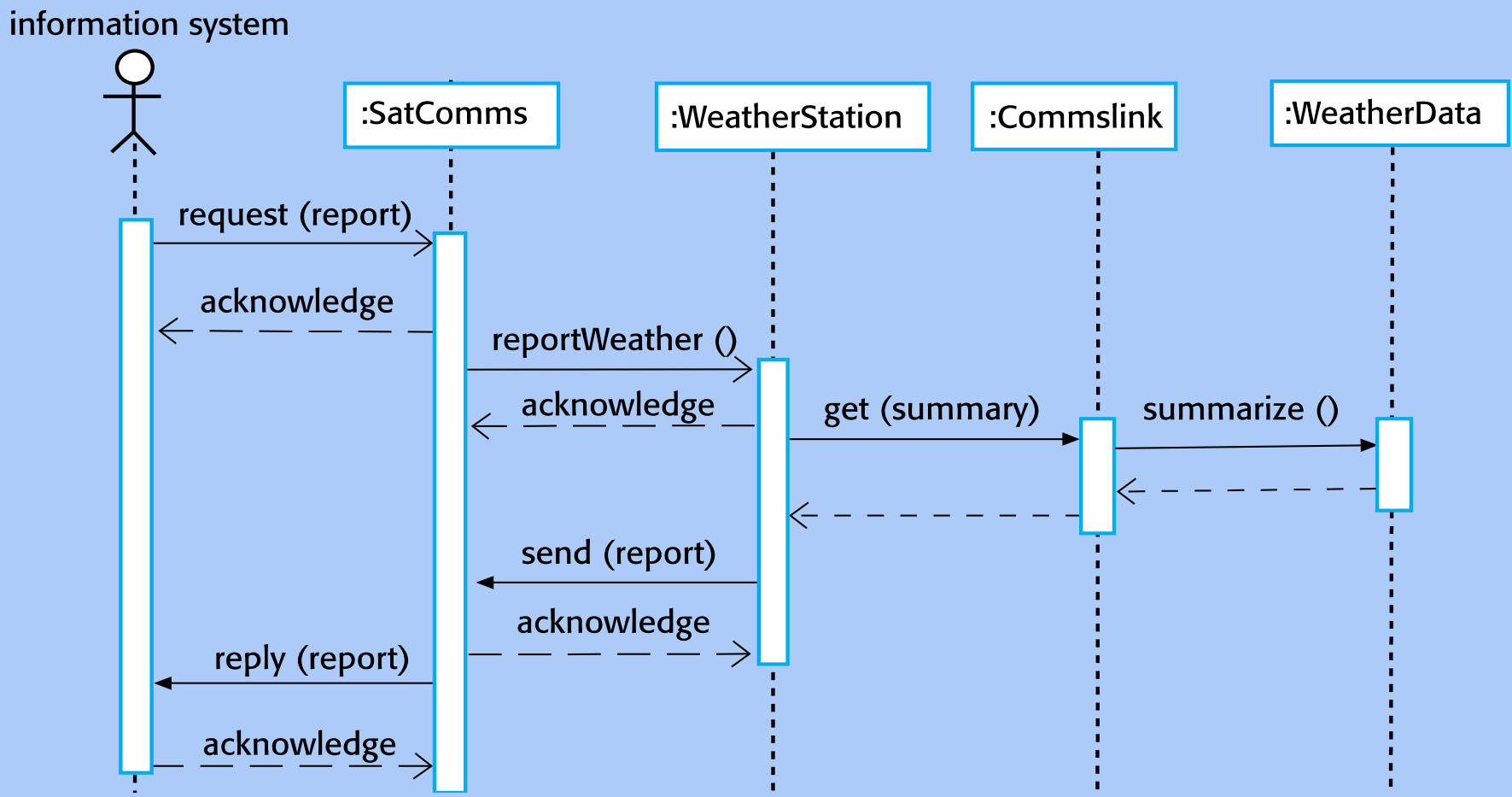
Weather station object classes

- Object class identification in the weather station system may be based on the tangible hardware and data in the system:
  - Ground thermometer, Anemometer, Barometer
    - Application domain objects that are 'hardware' objects related to the instruments in the system.
  - Weather station
    - The basic interface of the weather station to its environment. It therefore reflects the interactions identified in the use-case model.
  - Weather data
    - Encapsulates the summarized data from the instruments.

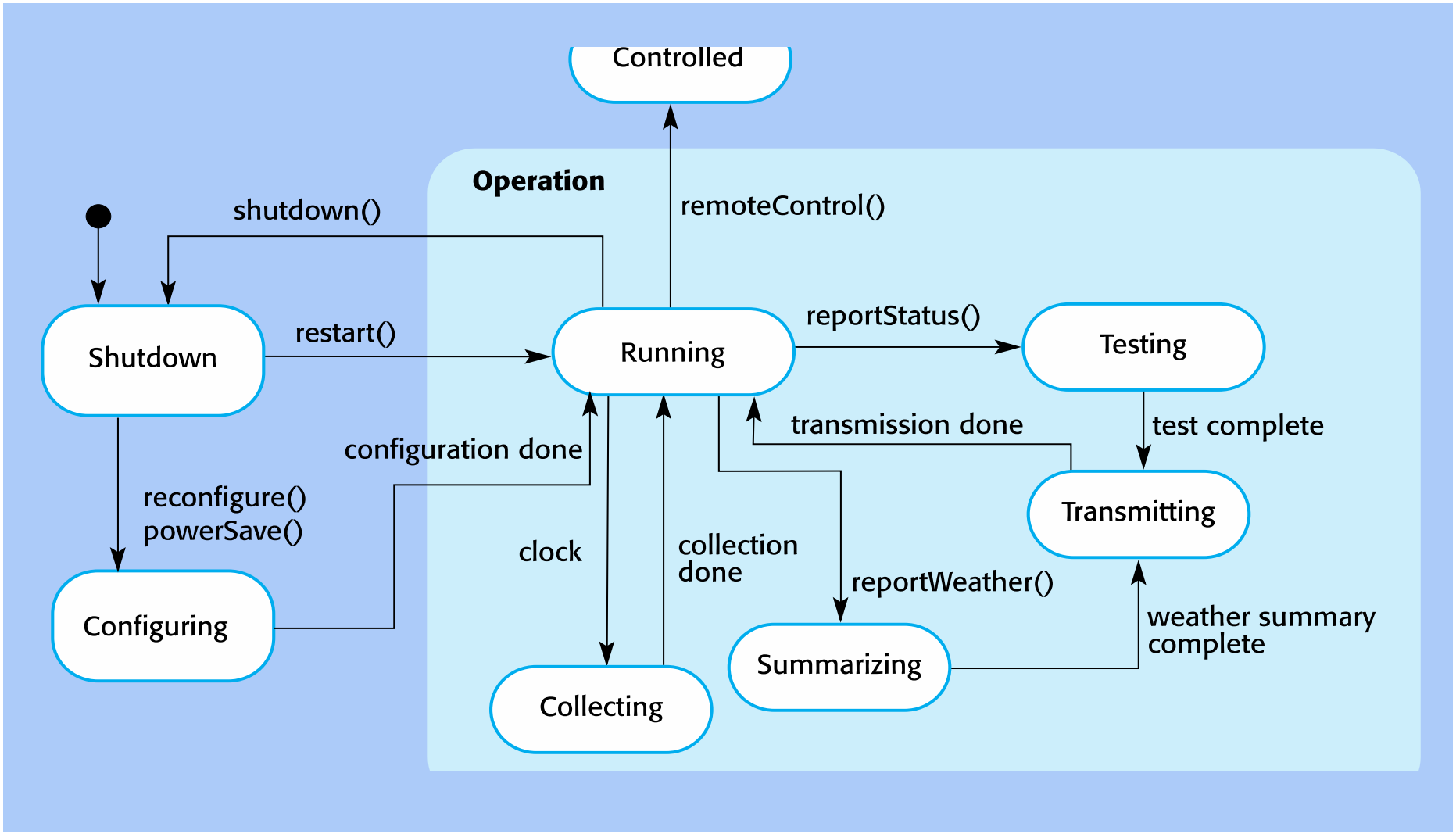
Weather station object classes



Sequence diagram describing data collection



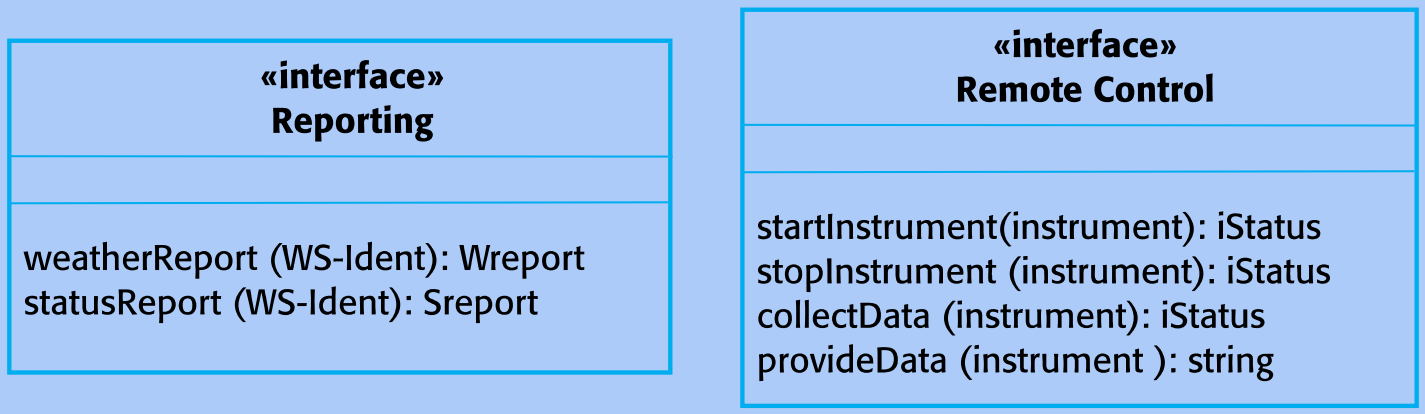
Weather station state diagram



Interface specification

- Object interfaces have to be specified so that the objects and other components can be designed in parallel.
- Designers should avoid designing the interface representation but should hide this in the object itself.
- Objects may have several interfaces which are viewpoints on the methods provided.
- The UML uses class diagrams for interface specification but Java may also be used.

Weather station interfaces



Implementation issues

Implementation issues

- Focus here is not on programming, although this is obviously important, but on other implementation issues that are often not covered in programming texts:
  - Reuse Most modern software is constructed by reusing existing components or systems. When you are developing software, you should make as much use as possible of existing code.
  - Configuration management During the development process, you have to keep track of the many different versions of each software component in a configuration management system.
  - Host-target development Production software does not usually execute on the same computer as the software development environment. Rather, you develop it on one computer (the host system) and execute it on a separate computer (the target system).

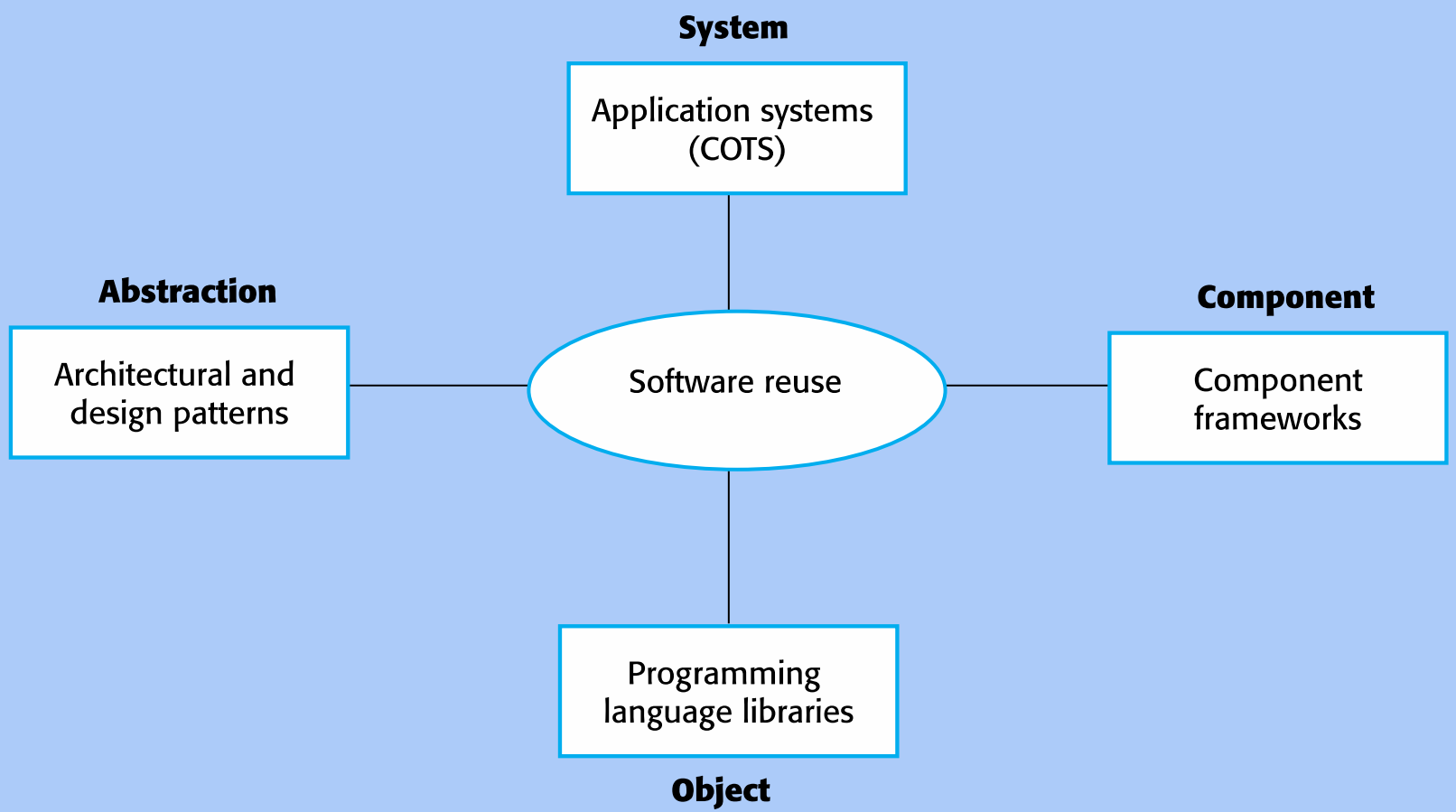
Reuse

- From the 1960s to the 1990s, most new software was developed from scratch, by writing all code in a high-level programming language.
  - The only significant reuse or software was the reuse of functions and objects in programming language libraries.
- Costs and schedule pressure mean that this approach became increasingly unviable, especially for commercial and Internet-based systems.
- An approach to development based around the reuse of existing software emerged and is now generally used for business and scientific software.

Reuse levels

- The abstraction level
  - At this level, you don't reuse software directly but use knowledge of successful abstractions in the design of your software.
- The object level
  - At this level, you directly reuse objects from a library rather than writing the code yourself.
- The component level
  - Components are collections of objects and object classes that you reuse in application systems.
- The system level
  - At this level, you reuse entire application systems.

Software reuse



Reuse costs

- The costs of the time spent in looking for software to reuse and assessing whether or not it meets your needs.
- Where applicable, the costs of buying the reusable software. For large off-the-shelf systems, these costs can be very high.
- The costs of adapting and configuring the reusable software components or systems to reflect the requirements of the system that you are developing.
- The costs of integrating reusable software elements with each other (if you are using software from different sources) and with the new code that you have developed.

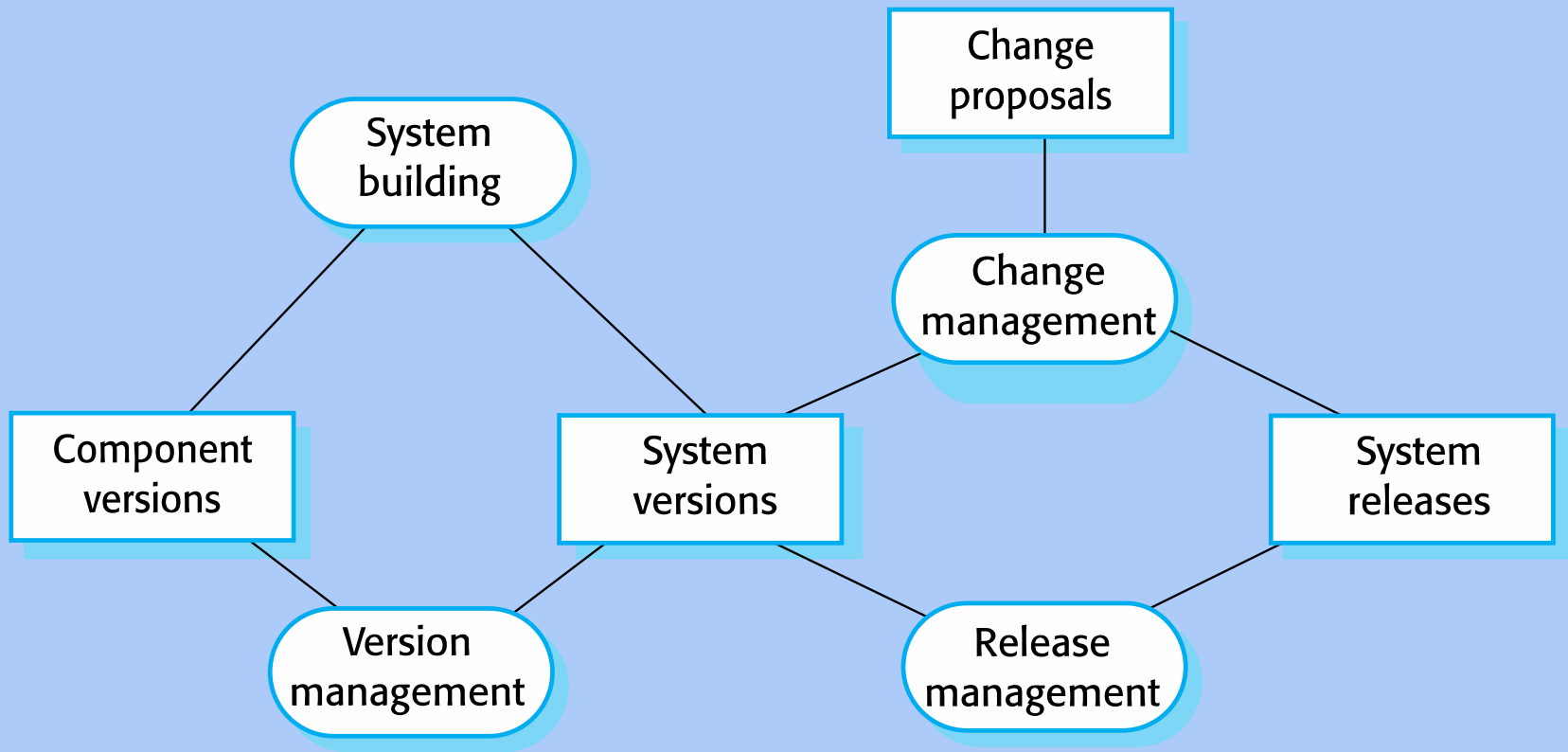
Configuration management

- Configuration management is the name given to the general process of managing a changing software system.
- The aim of configuration management is to support the system integration process so that all developers can access the project code and documents in a controlled way, find out what changes have been made, and compile and link components to create a system.
- See also Chapter 25.

Configuration management activities

- Version management, where support is provided to keep track of the different versions of software components. Version management systems include facilities to coordinate development by several programmers.
- System integration, where support is provided to help developers define what versions of components are used to create each version of a system. This description is then used to build a system automatically by compiling and linking the required components.
- Problem tracking, where support is provided to allow users to report bugs and other problems, and to allow all developers to see who is working on these problems and when they are fixed.

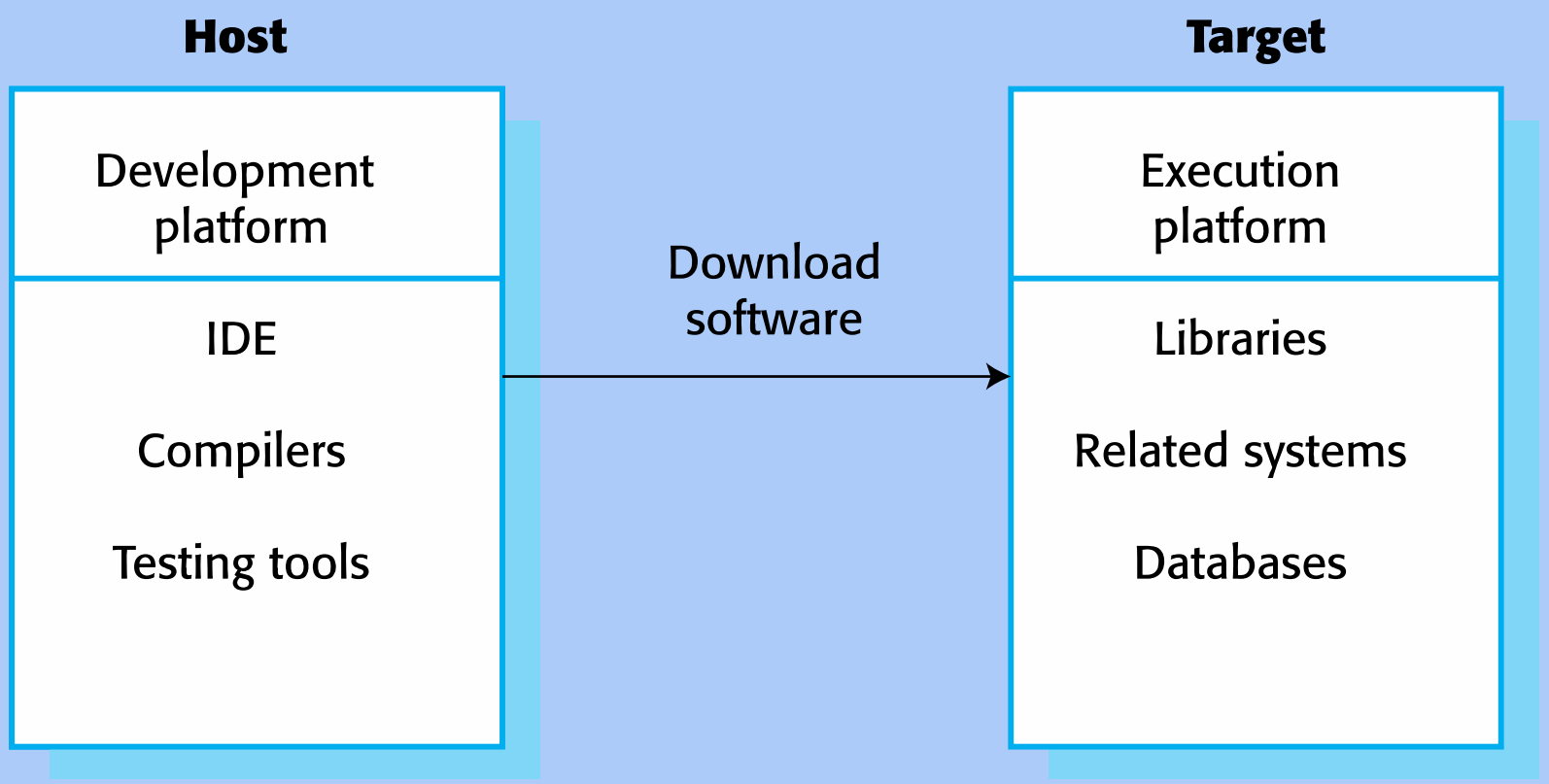
Configuration management tool interaction



Host-target development

- Most software is developed on one computer (the host), but runs on a separate machine (the target).
- More generally, we can talk about a development platform and an execution platform.
  - A platform is more than just hardware.
  - It includes the installed operating system plus other supporting software such as a database management system or, for development platforms, an interactive development environment.
- Development platform usually has different installed software than execution platform; these platforms may have different architectures.

Host-target development



Development platform tools

- An integrated compiler and syntax-directed editing system that allows you to create, edit and compile code.
- A language debugging system.
- Graphical editing tools, such as tools to edit UML models.
- Testing tools, such as Junit that can automatically run a set of tests on a new version of a program.
- Project support tools that help you organize the code for different development projects.

Integrated development environments (IDEs)

- Software development tools are often grouped to create an integrated development environment (IDE).
- An IDE is a set of software tools that supports different aspects of software development, within some common framework and user interface.
- IDEs are created to support development in a specific programming language such as Java. The language IDE may be developed specially, or may be an instantiation of a general-purpose IDE, with specific language-support tools.

Component/system deployment factors

- If a component is designed for a specific hardware architecture, or relies on some other software system, it must obviously be deployed on a platform that provides the required hardware and software support.
- High availability systems may require components to be deployed on more than one platform. This means that, in the event of platform failure, an alternative implementation of the component is available.
- If there is a high level of communications traffic between components, it usually makes sense to deploy them on the same platform or on platforms that are physically close to one other. This reduces the delay between the time a message is sent by one component and received by another.

Open source development

Open source development

- Open source development is an approach to software development in which the source code of a software system is published and volunteers are invited to participate in the development process
- Its roots are in the Free Software Foundation (www.fsf.org), which advocates that source code should not be proprietary but rather should always be available for users to examine and modify as they wish.
- Open source software extended this idea by using the Internet to recruit a much larger population of volunteer developers. Many of them are also users of the code.

Open source systems

- The best-known open source product is, of course, the Linux operating system which is widely used as a server system and, increasingly, as a desktop environment.
- Other important open source products are Java, the Apache web server and the mySQL database management system.

Open source issues

- Should the product that is being developed make use of open source components?
- Should an open source approach be used for the software's development?

Open source business

- More and more product companies are using an open source approach to development.
- Their business model is not reliant on selling a software product but on selling support for that product.
- They believe that involving the open source community will allow software to be developed more cheaply, more quickly and will create a community of users for the software.

Open source licensing

- A fundamental principle of open-source development is that source code should be freely available, this does not mean that anyone can do as they wish with that code.
  - Legally, the developer of the code (either a company or an individual) still owns the code. They can place restrictions on how it is used by including legally binding conditions in an open source software license.
  - Some open source developers believe that if an open source component is used to develop a new system, then that system should also be open source.
  - Others are willing to allow their code to be used without this restriction. The developed systems may be proprietary and sold as closed source systems.

License models

- The GNU General Public License (GPL). This is a so-called 'reciprocal' license that means that if you use open source software that is licensed under the GPL license, then you must make that software open source.
- The GNU Lesser General Public License (LGPL) is a variant of the GPL license where you can write components that link to open source code without having to publish the source of these components.
- The Berkley Standard Distribution (BSD) License. This is a non-reciprocal license, which means you are not obliged to re-publish any changes or modifications made to open source code. You can include the code in proprietary systems that are sold.

License management

- Establish a system for maintaining information about open-source components that are downloaded and used.
- Be aware of the different types of licenses and understand how a component is licensed before it is used.
- Be aware of evolution pathways for components.
- Educate people about open source.
- Have auditing systems in place.
- Participate in the open source community.

Key points

- Software design and implementation are inter-leaved activities. The level of detail in the design depends on the type of system and whether you are using a plan-driven or agile approach.
- The process of object-oriented design includes activities to design the system architecture, identify objects in the system, describe the design using different object models and document the component interfaces.
- A range of different models may be produced during an object-oriented design process. These include static models (class models, generalization models, association models) and dynamic models (sequence models, state machine models).
- Component interfaces must be defined precisely so that other objects can use them. A UML interface stereotype may be used to define interfaces.

Key points

- When developing software, you should always consider the possibility of reusing existing software, either as components, services or complete systems.
- Configuration management is the process of managing changes to an evolving software system. It is essential when a team of people are cooperating to develop software.
- Most software development is host-target development. You use an IDE on a host machine to develop the software, which is transferred to a target machine for execution.
- Open source development involves making the source code of a system publicly available. This means that many people can propose changes and improvements to the software.