Week 13 - Monday

COMP 2000

Last time

- What did we talk about last time?
- Sorting
- **`Arrays.sort()`**
- **`Collections.sort()`**
- **`Comparable<T>`** interface
- Custom **`Comparator<T>`** objects

Questions?

Project 4

Software Engineering

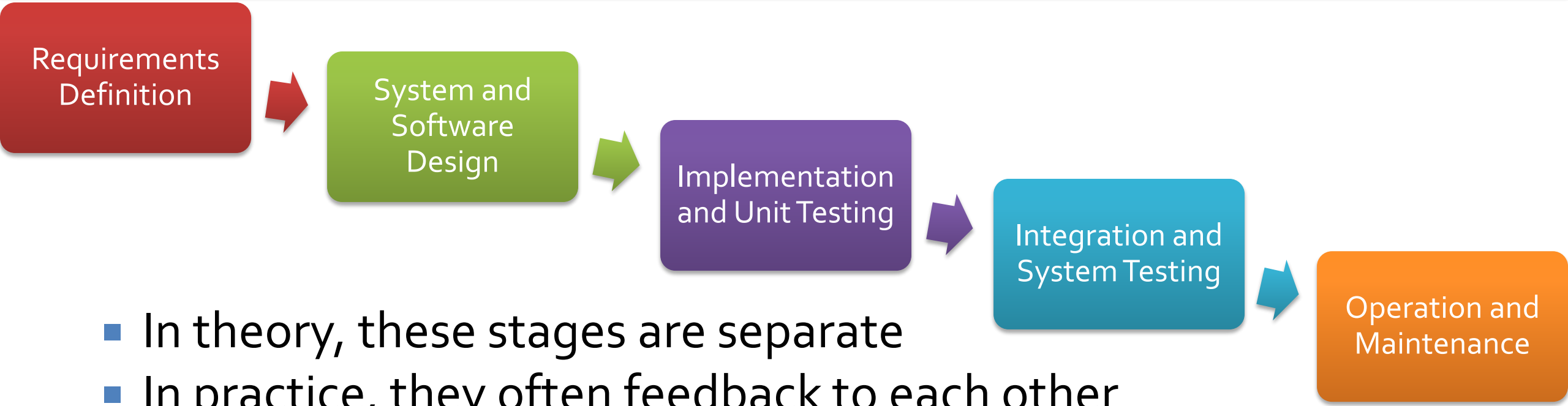
A few things we want from software engineering

Characteristic	Description
Maintainability	We can update the code to add in new requirements and features.
Dependability and security	Software is reliable, secure, and safe. Systems failures don't cause physical or economic damage. Hackers can't break in or damage the system.
Efficiency	Software uses processors and memory efficiently. Software is responsive.
Acceptability	The users of the software can understand and use the software, and it's compatible with other tools they use.

Why software engineering is important

- People need software
  - It's everywhere, in every facet of life
  - If it doesn't work correctly or is vulnerable to attack, people can be hurt, die, suffer financial losses, etc.
- It's cheaper to engineer it the right way
  - Hacking stuff together seems faster and cheaper…at first
  - But for large, long term projects, a well-managed development process ends up saving money and time

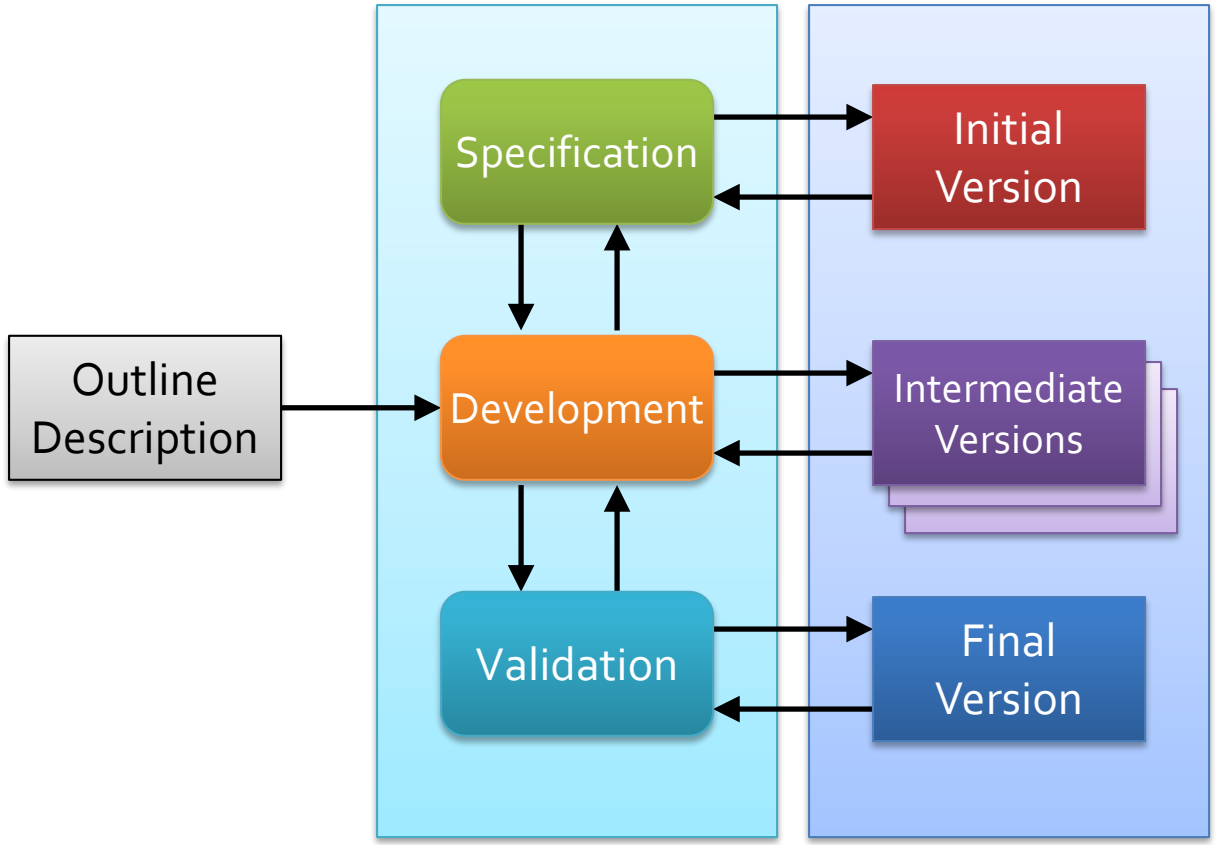
Waterfall model



- In theory, these stages are separate
- In practice, they often feedback to each other
- It's very expensive if mistakes are discovered in later stages
  - One rule of thumb is that mistakes costs 10 times as much to fix than they would have at a previous stage

Incremental development

- Incremental software development starts with an initial version that evolves with user feedback
- Specification, development, and validation happen continually and concurrently
- Incremental development is a cornerstone of Agile development



Pros and cons of incremental development

PROS

- The cost of changing customer requirements is smaller
- It's easier to get customer feedback
- Rapid delivery and deployment of usable software is possible

CONS

- There's less documentation since it's time-prohibitive to document each rapidly changing version
- Structure tends to worsen over time as more code is added
  - Time must be spent on refactoring

UML

Modeling

- At both the requirements stage and the design stage, modeling can be useful
- **Modeling** mostly means drawing boxes and arrows
- We want high-level descriptions of:
  - What the thing is supposed to do
  - What parts it's composed of
  - How it does what it does

System modeling

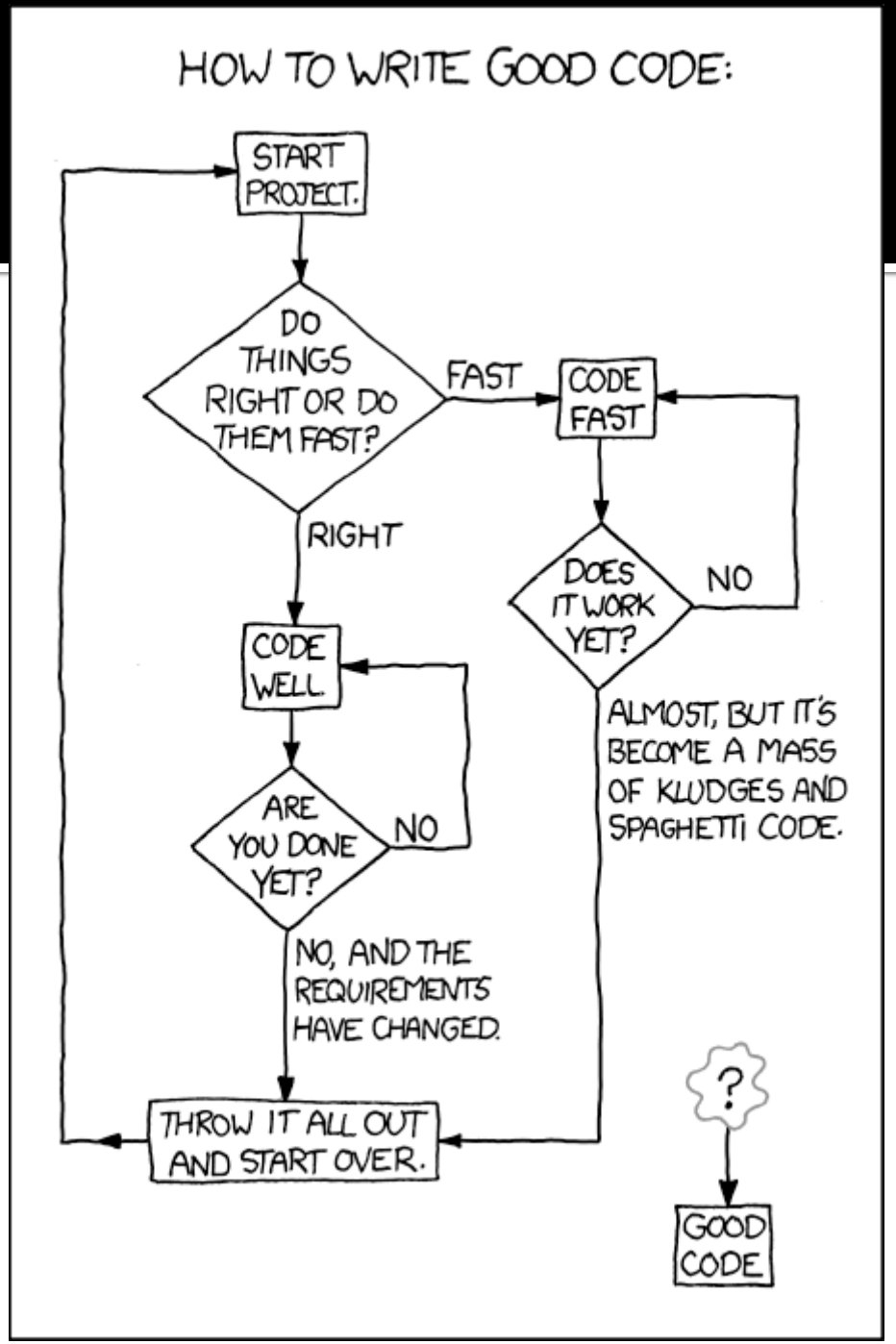
- **Models leave out details**
- Models are useful to help understand a complex system
  - During requirements engineering, models clarify what an existing system does
  - Or models could be used to plan out a new system
- Models can represent different perspectives of a system:
  - **External:** the context of a system
  - **Interaction:** the interactions within the system or between it and the outside
  - **Structural:** organization of a system
  - **Behavior:** how the system responds to events

UML

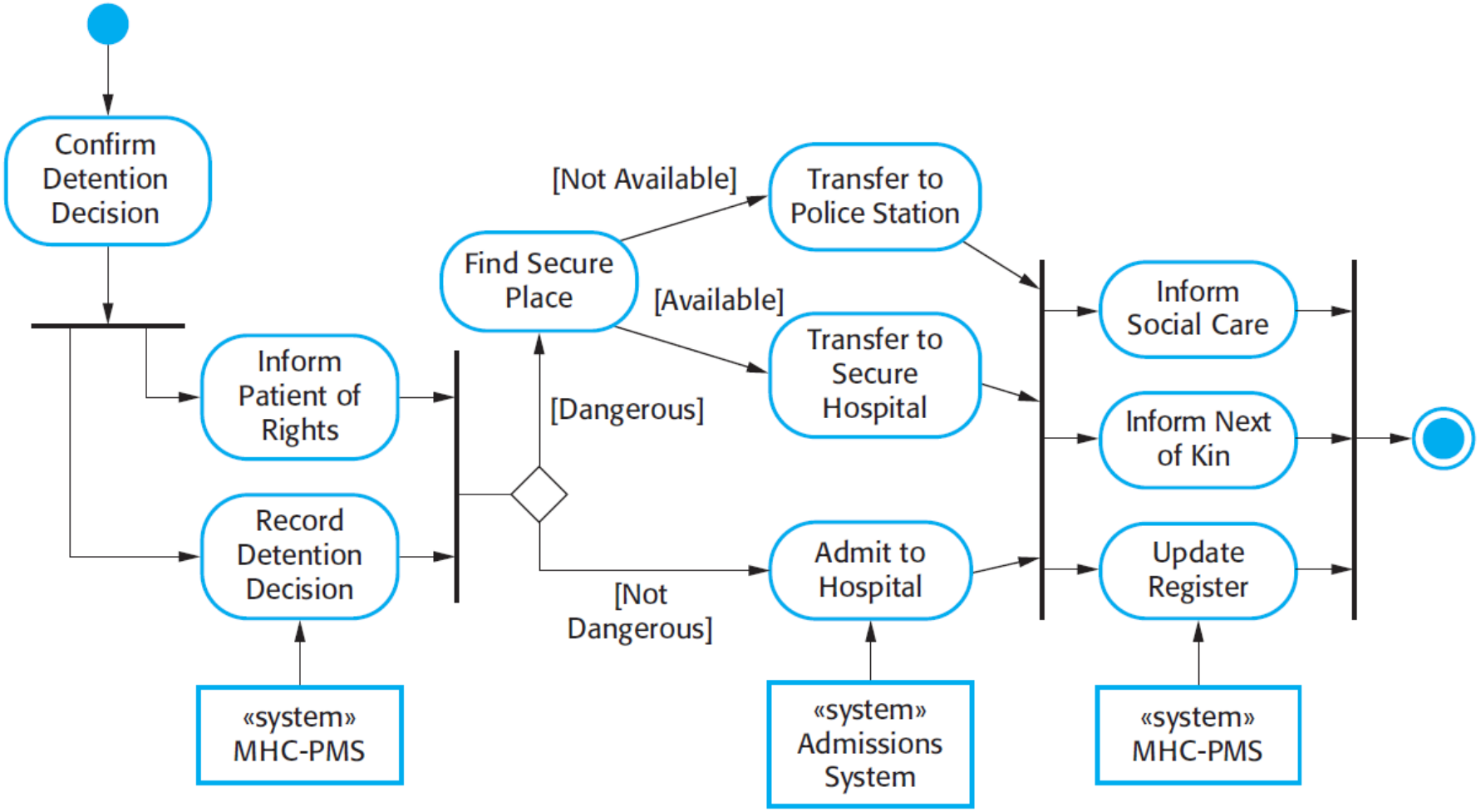
- The **Unified Modeling Language** (UML) is an international standard for graphical models of software systems
- A few useful kinds of diagrams:
  - Activity diagrams
  - Use case diagrams
  - Sequence diagrams
  - State diagrams
- Class diagrams are important enough that we'll talk about them in greater detail

Activity diagrams

- Activity diagrams show the workflow of actions that a system takes
- XKCD of an activity diagram for writing good code
  - From: https://xkcd.com/844/
- Formally:
  - Rounded rectangles represent actions
  - Diamonds represent decisions
  - Bars represent starting or ending concurrent activities
  - A black circle represents the start
  - An encircled black circle represents the end

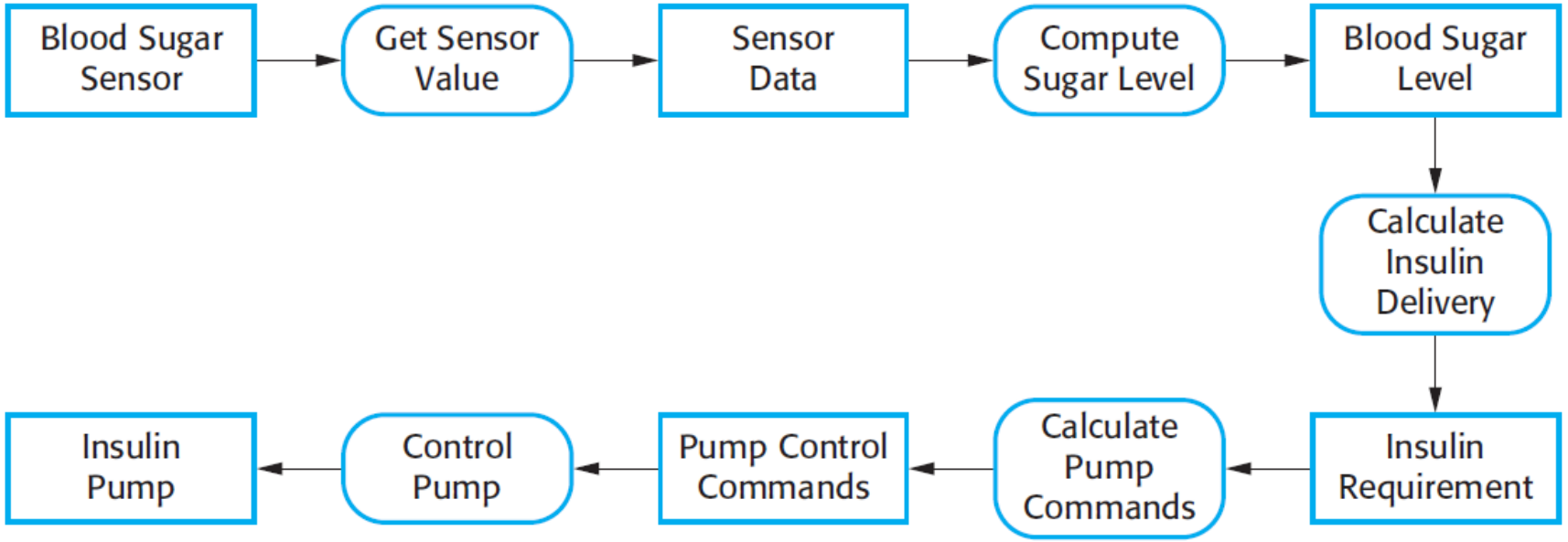


More detailed activity model



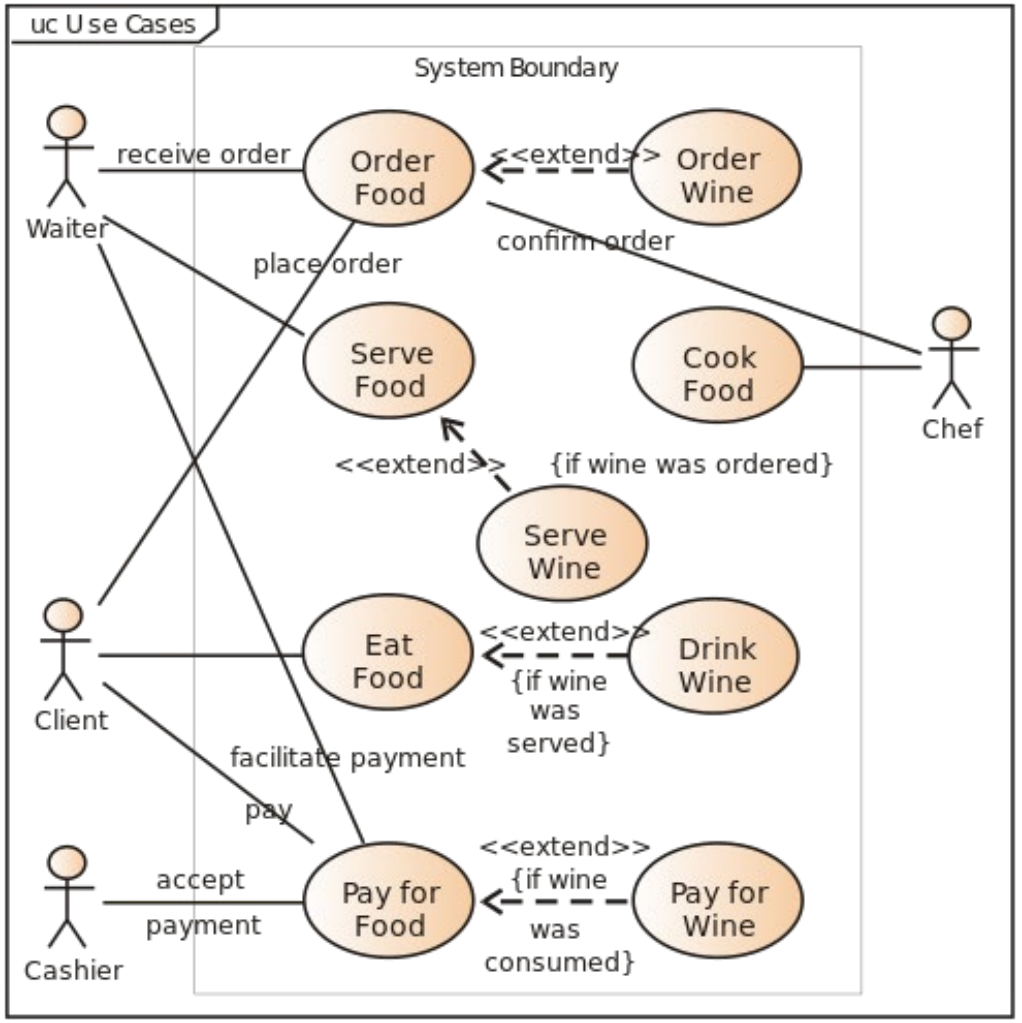
Data-driven modeling

- Data-driven models show how input data is processed to generate output data
- The following is an activity diagram that shows how blood sugar data is processed by a system to deliver the right amount of insulin



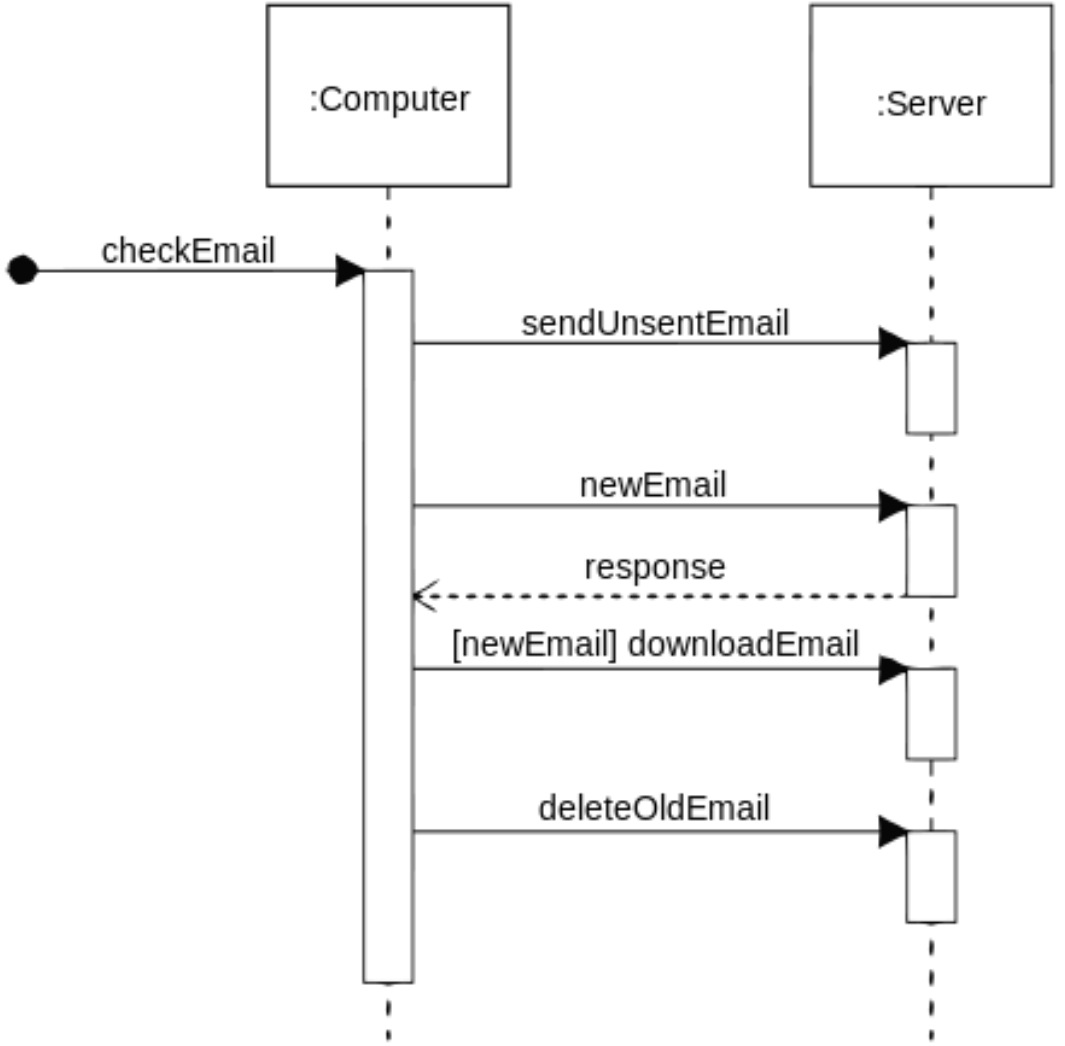
Use case diagrams

- Use case diagrams show relationships between users of a system and different use cases where the user is involved
- Example from Wikipedia:



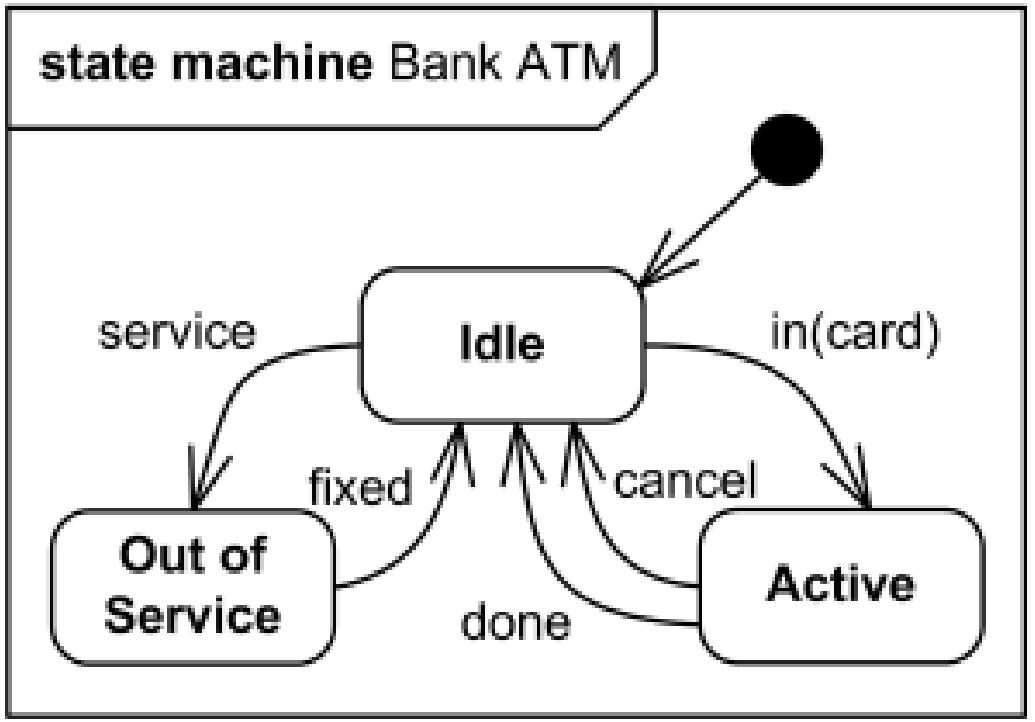
Sequence diagrams

- Sequence diagrams show system object interactions over time
- These messages are visualized as arrows
  - Solid arrow heads are synchronous messages
  - Open arrow heads are asynchronous messages
  - Dashed lines represent replies
- Example from Wikipedia:



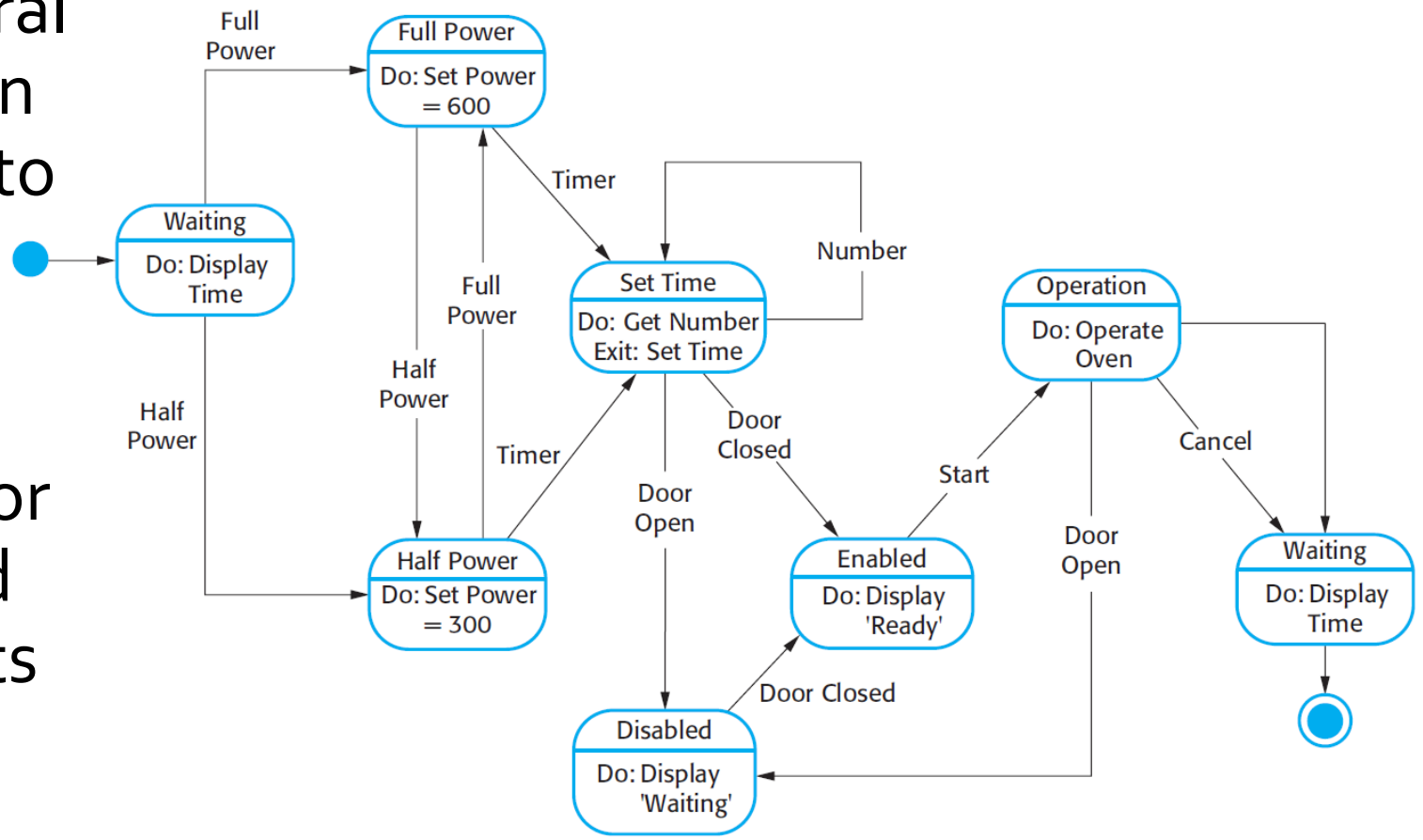
State diagrams

- State diagrams are the UML generalization of finite state automata from discrete math
- They describe a series of states that a system can be in and how transitions between those states happen
- Example from uml-diagrams.org:



Event-driven modeling

- Event-driven modeling is another kind of behavioral modeling that focuses on how a system responds to events rather than on processing a stream of data
- Here's a state diagram for a microwave oven based on various outside events



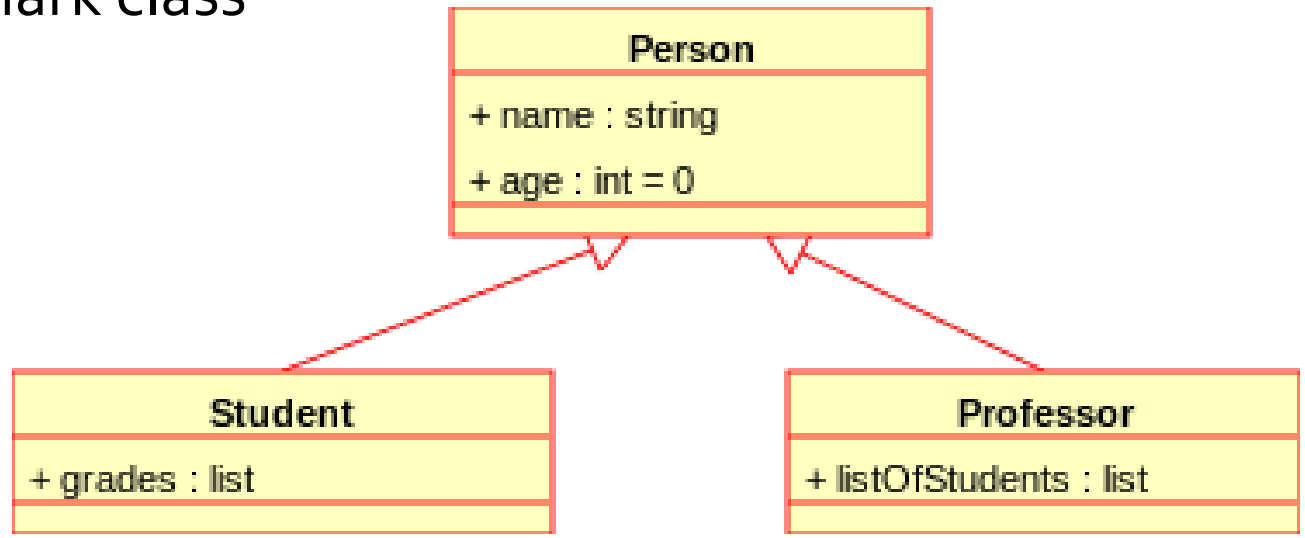
Class Diagrams

Structural models

- Structural models show how a system is organized in terms of its components and their relationships
- UML class diagrams are used for structural models, but they can be used in many different ways:
  - Relationships
  - Generalization
  - Aggregation

Class diagrams

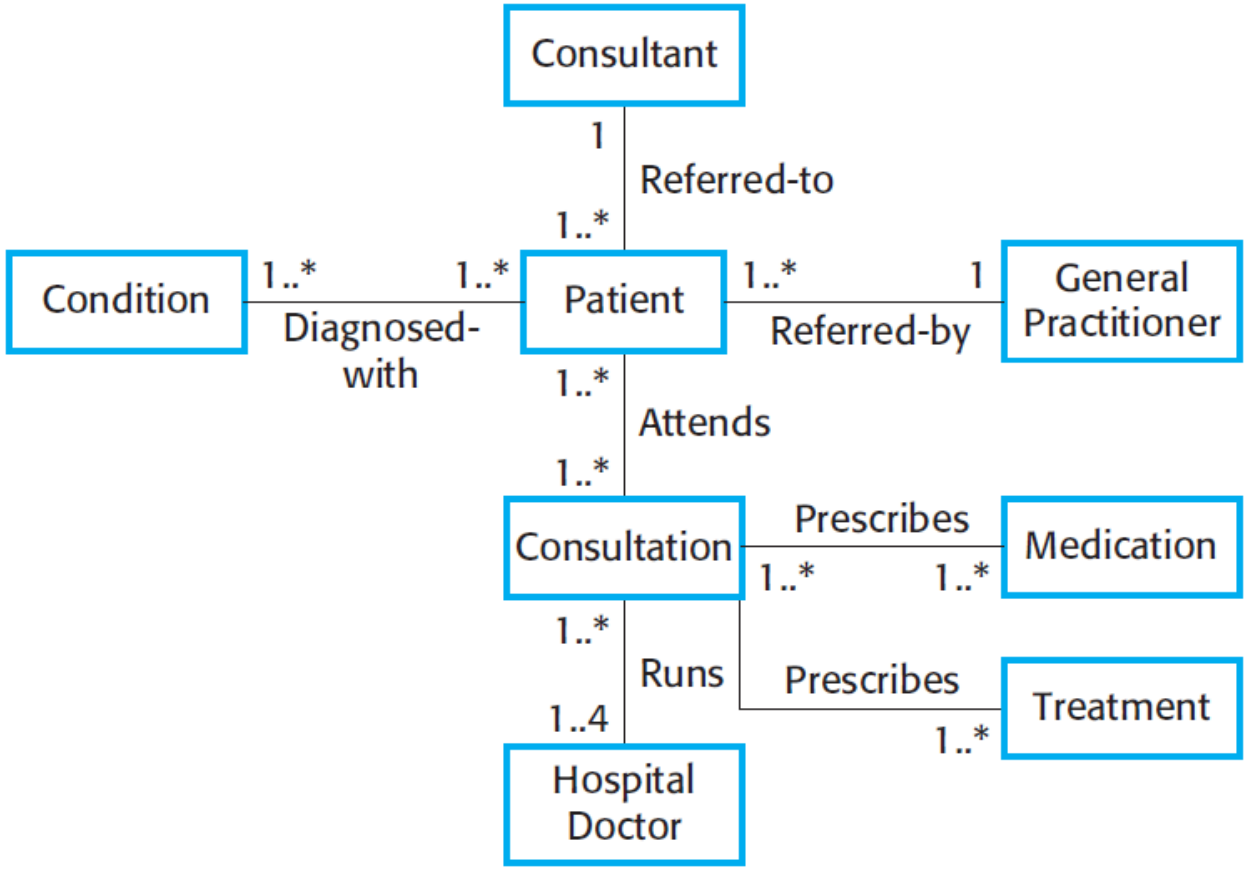
- Class diagrams show many kinds of relationships
- The **classes** being described often (but not always) map to classes in object-oriented languages
- The following symbols are used to mark class members:
  - **+** Public
  - **–** Private
  - **#** Protected
  - **/** Derived
  - **~** Package
  - ***** Random
- Example from Wikipedia:



Relationships

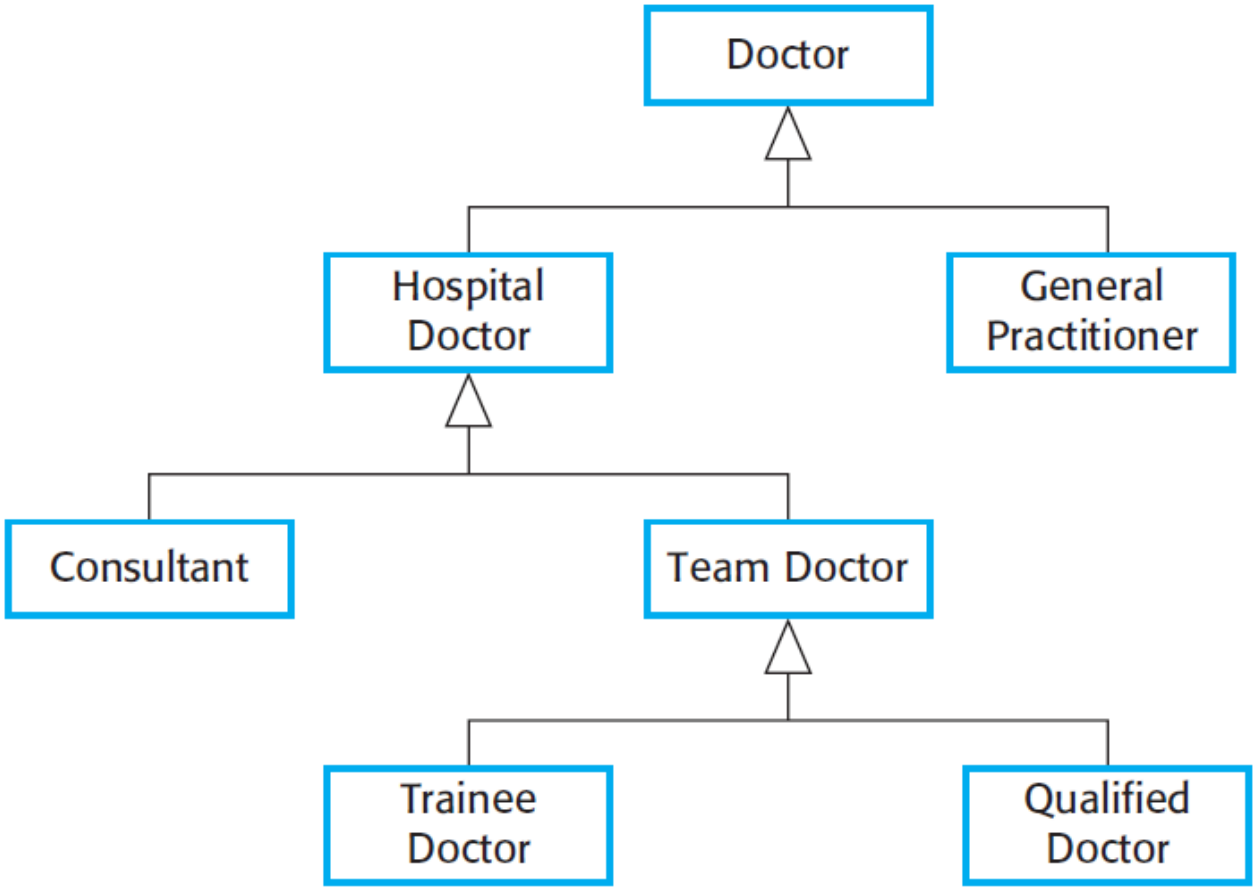
- Associations between classes can be drawn with a line in a class diagram

- Notations can be used to mark relationships as one to one, many to one, many to many, etc.
- These kinds of relationships are particularly important when designing a database



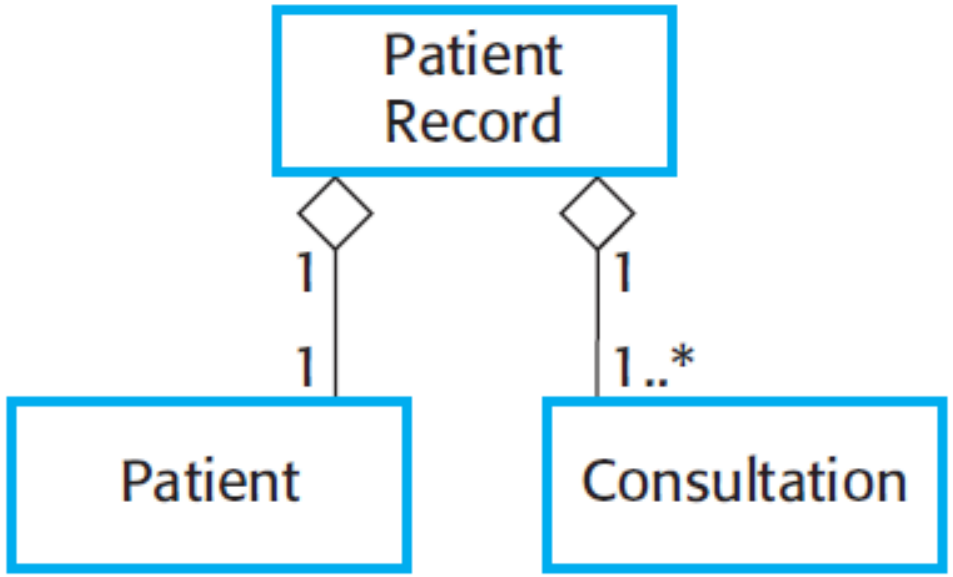
Generalization

- Classes can be listed with their attributes
- However, there are often classes that share attributes with each other
- Some classes are specialized versions of other classes, with more attributes and abilities
- This relationship between general classes and more specialized classes is handled in Java by the mechanic of **inheritance**



Aggregation

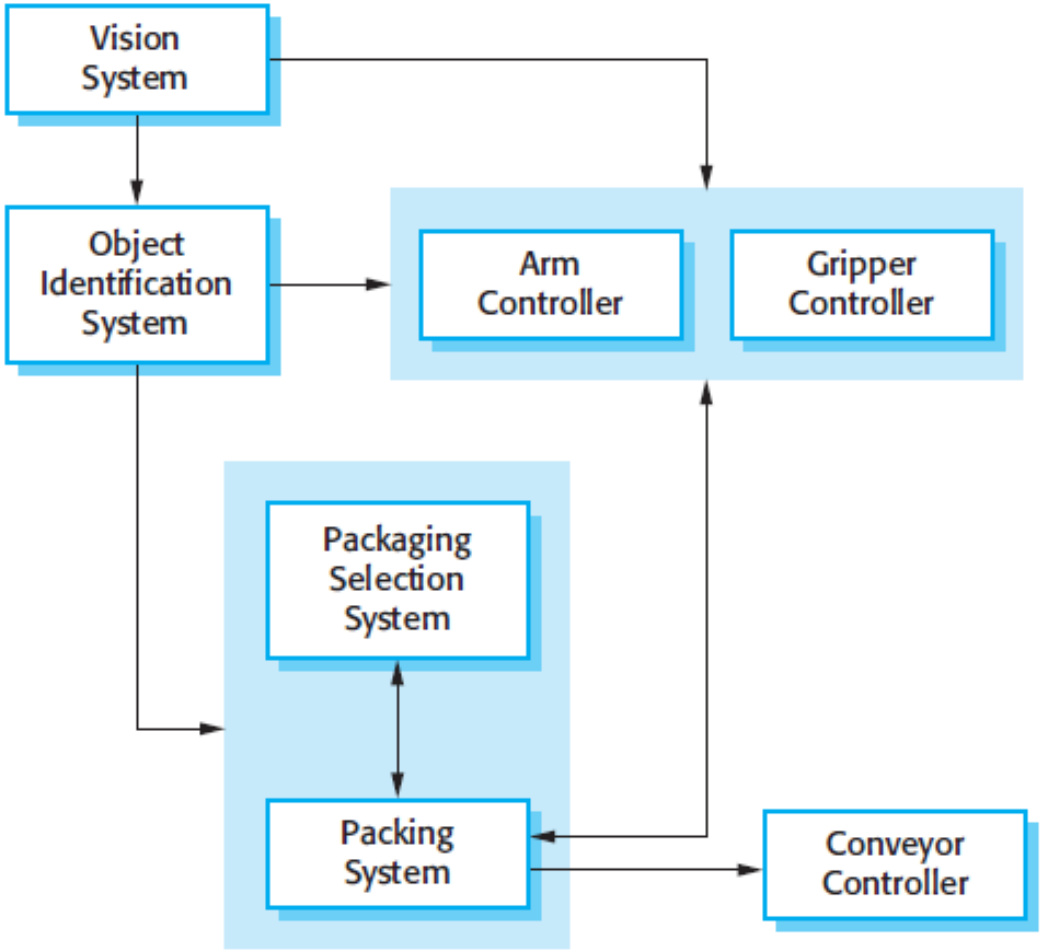
- Another way of using class diagrams is to show that some objects or classes are made up of smaller parts represented by other classes
- A diamond shape is used to mark a class that is the whole, and its parts are connected to the diamond



Architectural Design

Architecture

- Architecture describes the main structural components in a system and the relationships between them
- Architectural design is somewhat freeform
  - It's hard to follow a recipe for architectural design
- There's overlap between requirements engineering and architectural design
- **Block diagrams** are commonly used to describe architecture:



Advantages of well-documented architecture

- Stakeholder communication
  - Everyone involved in the project can understand the system at a high level
- System analysis
  - Creating architecture requires some (hopefully useful) analysis
- Large-scale reuse
  - Architecture describes how a system is organized and how the components interoperate
  - Since system architectures are similar for systems with similar requirements, it may be possible to choose an off-the-shelf system with the right architecture

Architectural design decisions

- Since architecture is somewhat free-form, a good way to guide the design is by asking questions
- Since non-functional requirements often relate to the system as a whole, which non-functional requirements should the architecture focus on?
  - Performance
  - Security
  - Safety
  - Availability
  - Maintainability
- Emphasis on one area may hurt other areas
  - For example, greater security usually comes at the cost of performance

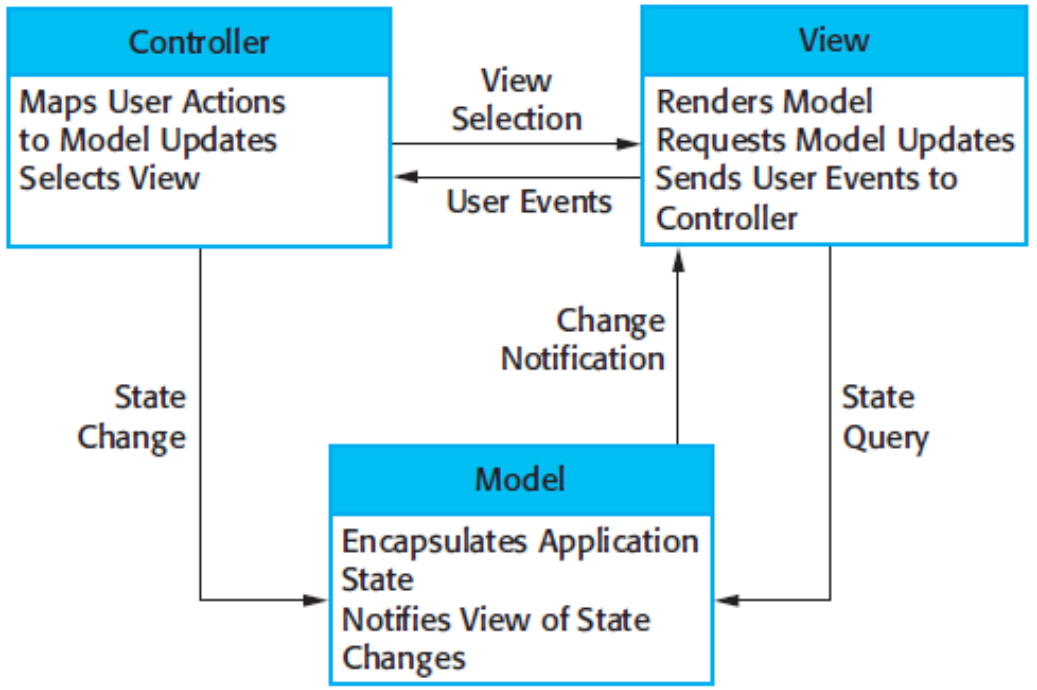
Architectural Patterns

Architectural patterns

- Even though architectural design is somewhat free-form, architectural patterns have evolved that fit many different kinds of programs
- An **architectural pattern** is an abstract description of a system that has worked well in the past
- Examples:
  - Model-view-controller
  - Layered architecture
  - Repository architecture
  - Client-server architecture
  - Pipe and filter architecture

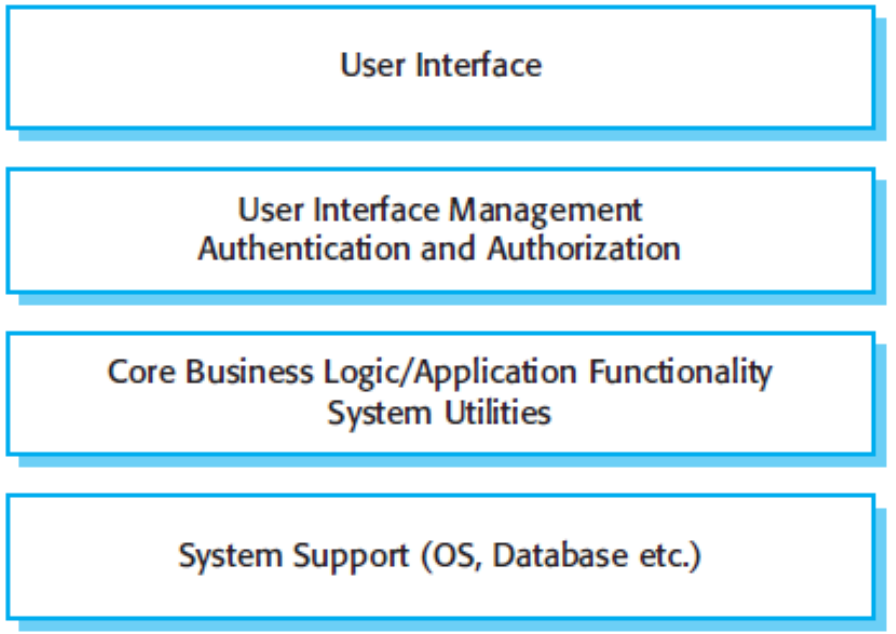
Model-View-Controller

- The **Model-View-Controller** (MVC) pattern fits many kinds of web or GUI interactions
- The **model** contains the data that is being represented, often in a database
- The **view** is how the data is displayed
- The **controller** is code that updates the model and selects which view to use
- The Java Swing GUI system is built around MVC
- **Good:** greater independence between data and how it's represented
- **Bad:** additional complexity for simple models



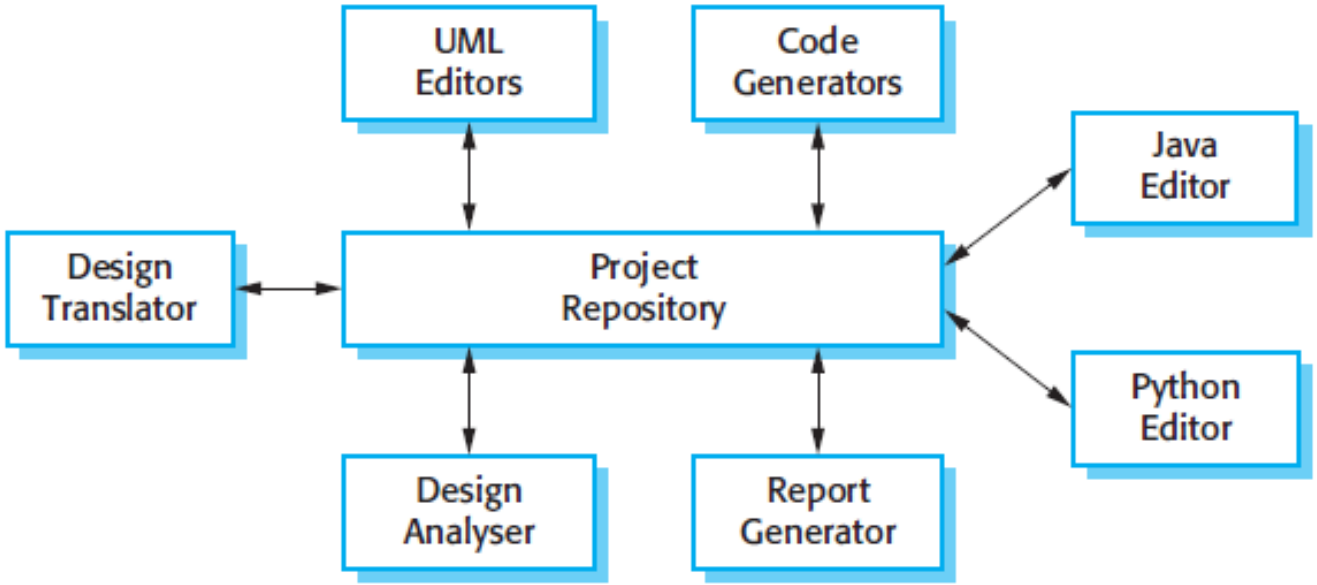
Layered architecture

- Organize the system into layers
- Each layer provides services to layers above it, with the lowest layer being the most fundamental operations
- Layered architectures work well when adding functionality on top of existing systems
- **Good:** entire layers can be replaced as long as the interfaces are the same
- **Bad:** it's hard to cleanly separate layers, and performance sometimes suffers



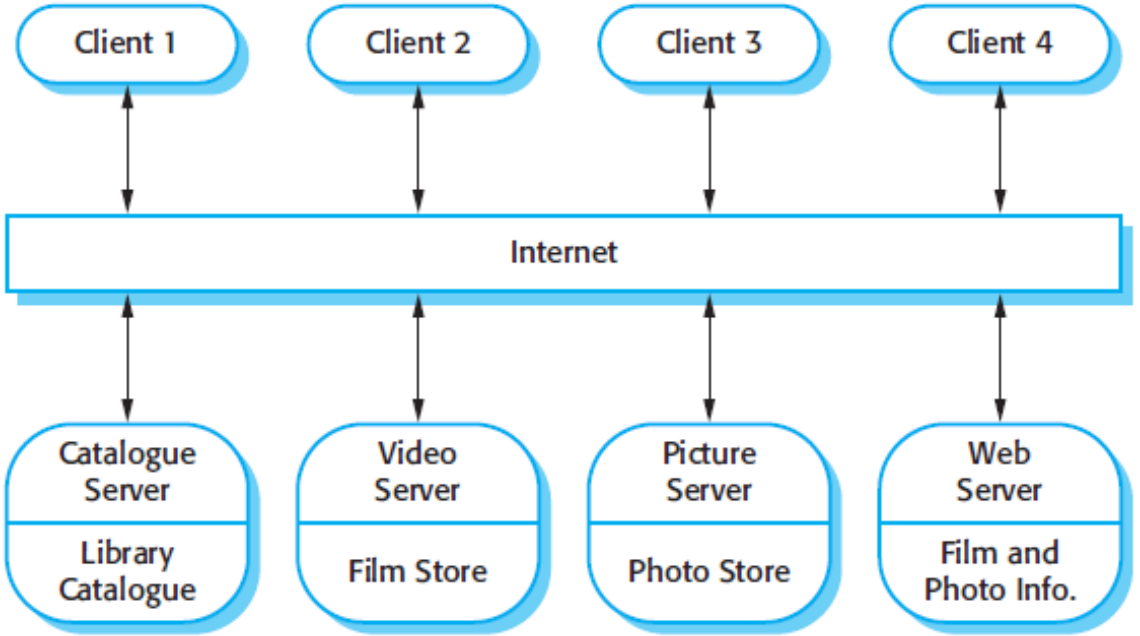
Repository architecture

- If many components share a lot of data, a Repository pattern might be appropriate
- Components interact by updating the repository
- This pattern is ideal when there is a lot of data stored for a long time
- **Good:** components can be independent
- **Bad:** the repository is a single point of failure



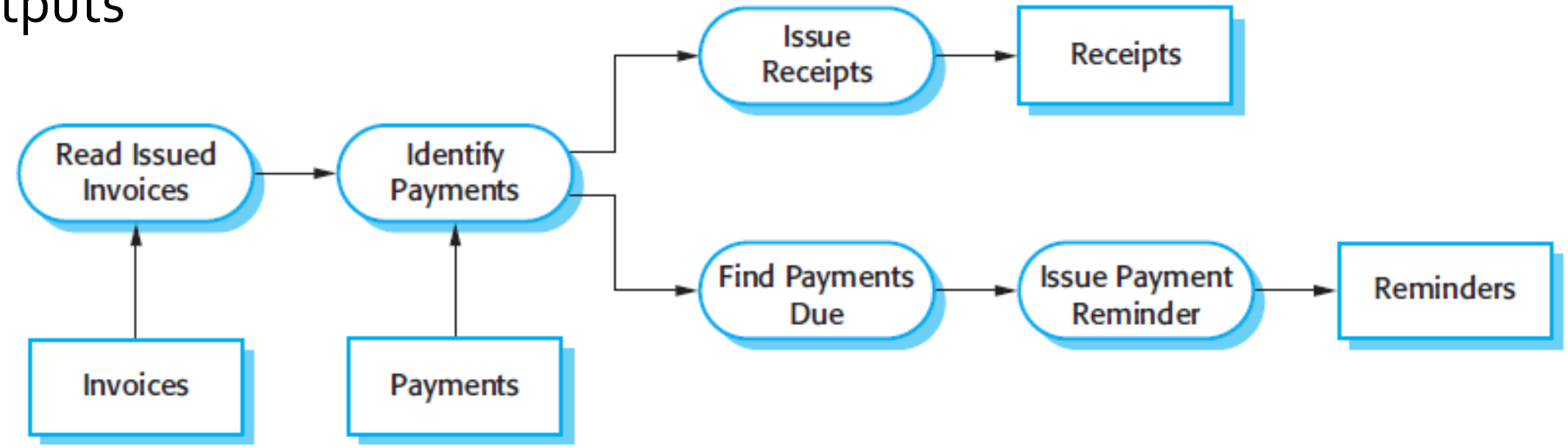
Client-server architecture

- Client-Server patterns are used for distributed systems
- Each server provides a separate service, and clients access those services
- **Good:** work is distributed, and clients can access just what they need
- **Bad:** each service is a single point of failure, and performance might be unpredictable



Pipe and filter architecture

- In the Pipe and Filter pattern, data is passed from one component to the next
- Each component transforms input into output
- **Good:** easy to understand, matches business applications, and allows for component reuse
- **Bad:** each component has to agree on formatting with its inputs and outputs



Upcoming

Next time…

- Testing
- Introduction to JUnit

Reminders

- **Work on Project 4**
- Lab is tomorrow