Week 15 - Monday

COMP 3100

Last time

- What did we talk about last time?
- Mostly work days
- Before that:
- Execution and control
  - Earned value management
  - Planned Value, Earned Value, Actual Cost
  - Burn charts

Questions?

Final exam format

- Time: Friday, 12/13/2024 2:45 - 4:45 p.m.

- The exam will have:
  - Short answer questions
  - At least one matching question
  - One or two diagrams or figures you must create
  - Two to three essay questions

Review

Managerial software engineering concerns

- Managerial concerns are about organization and control
  - Project cost
  - Time estimation
  - Scheduling and tracking
  - Team management
  - Risk management
  - Quality

Technical software engineering concerns

- Technical concerns are about what product, how to build it, and building it
  - Software requirements
  - Design
  - Programming languages and environments
  - Coding standards
  - Defect prevention, detection, and removal
  - Version control
  - Documentation
  - Maintenance

Management

Aspects of a project that must be managed

- **Scope**
  - How much the project is trying to accomplish
  - **Creep** is the tendency for the work to increase
- **Time**
  - Must be reasonable for the project size
  - Must be monitored
- **Cost**
  - Similar issues as with time
- **Quality**
  - How good is acceptable?
  - Quality assurance must be done through the project, not just at the end
- **Resources**
  - Do you have the people (and tools) to get the job done?
- **Risks**
  - Have you planned for things going wrong?

Software development methods

- Traditional methods
  - Careful planning and hierarchical leadership
  - Steps like requirement specification, design, implementation, testing, and maintenance
  - Example: Waterfall model
- Agile methods
  - Constant iteration
  - Self-directed teams
  - Minimal documentation
  - Example: Scrum
- Both methods are widely used and many successful teams use aspects of both
- The project for this class will mostly employ traditional methods because agile works best with experienced developers

Requirements and design

- **Requirements** are functions or characteristics that software has
- **Customers** or **users** determine the requirements
- **Stakeholder** is a broad term that includes customers, users, developer, managers, and maybe the public
- **Designs** specify how the software system will meet the requirements
- Designs can look at a system from different aspects
- **Design patterns** are standard solutions to problems that have been useful in the past and can help structure designs

Implementation

- After the design is made, the software must be implemented in one or more programming languages
- **Compilers** and **interpreters** are used to run the programs
- **Editors** allow people to write code
- **Version control** tools let people track the evolution of the code
- **Code checkers** see if the code is meeting certain standards
- **Debuggers** help programmers find mistakes

Version Control

Version control

- For any large software project (and even small ones), it's valuable to have a way to track changes over time
- Such tools are called **version control systems**
- They allow:
  - Changes to be tracked over time
  - Developers to check code into repositories
  - Comparison of files over time
  - Documentation of changes made
- It's more than just a glorified backup system

Repository

- A **repository** is where all the development data is stored
  - Usually called **repos** by professionals
- Repositories include the current source code as well as a history of all the changes ever made
- For source code, most version control systems use **delta compression**, meaning that only the *differences* between files are stored
- Thus, hundreds of versions of your code can be stored without taking up hundreds of times the space

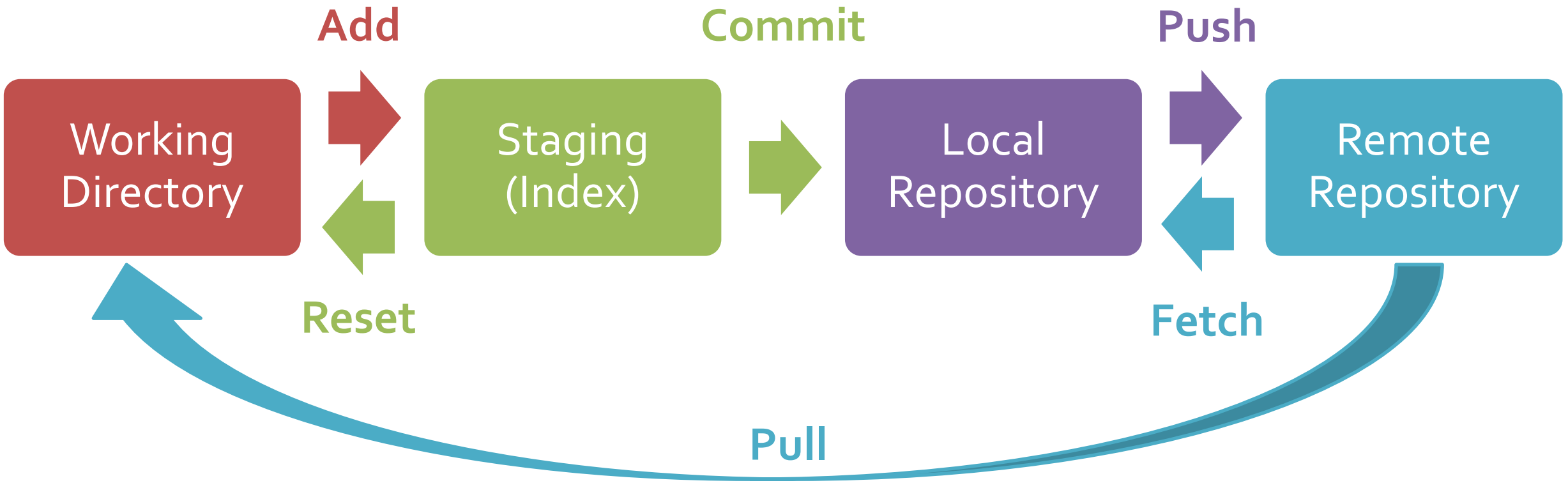
Actions

- **Committing** a file is adding its changes to a repository
- **Cloning** means creating a copy of another repository, including history
- **Merging** is combining two sets of files with independent changes into one set with changes from both
- **Pulling (or fetching)** copies the changes from an outside repository and adds them to the current repository
- **Pushing** copies the changes from the current repository to an outside repository

Philosophy of Git

- Git is a distributed VCS
- Every computer has a complete history of all the changes, ever
- There's no central server
- Programmers make changes and push them to or pull them from other repositories
- All operations are designed to be fast
- Torvalds did a pretty good job, but some common tasks are confusing

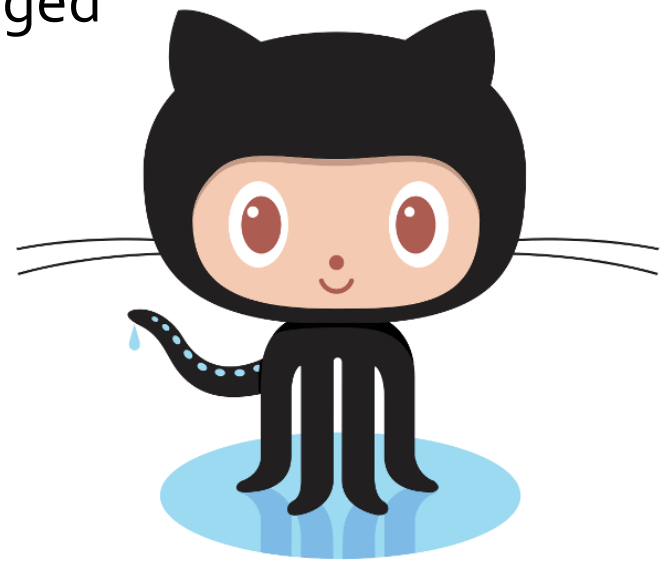
Git process



It's all possible to reset to an earlier commit, overwriting the working directory, but it's confusing to put that arrow in.

GitHub

- GitHub.com provides online repositories for code
  - Private repositories (except for education) are **not** free
  - Public repositories are free
- Git can be used without GitHub
- GitHub can even be used without Git (since it has support for SVN)
- Git has nice tools for:
  - Visualizing who's committing and how much they have changed
  - Issue tracking
  - Writing commit information and Read Me files
  - Pushing and pulling repos stored on GitHub
  - Creating webpages related to releasing software
- Ironically, Linus Torvalds hates GitHub



Requirements

Stakeholders

- **Stakeholders** are anyone affected by a product or its development
  - **Customers** are the people that pay for a product
  - **Users** are people who interact with the product
  - **Clients** are people for whom software was created (includes both customers and users)
  - **Developers** are all the people who work on the project
  - **Regulators** are responsible for ensuring that software meets standards
  - **Marketers** stand in for clients when making mass-market products

Stakeholder needs

- A **stakeholder need** is a feature that one or more stakeholders want
- Sometimes, these needs are written in descriptions called **needs specifications**
- Then, developers have to wrangle all of these conflicting, incomplete, and vague needs into a requirements specification
- Traditional methods may have a specific person who does this
  - Titles like **requirements analyst**, **requirements specialist**, **user interaction designer**

Functional requirements

- It's common to divide requirements into functional and non-functional categories
- **Functional requirements** are about how software takes input and turns it into output, its **behavior**
  - Appearance
  - User interface actions
  - Input and output processes
- Most requirements are functional requirements, and they take the most time and effort to specify

Non-functional requirements

- **Non-functional requirements** describe the **properties** software must have
  - Speed of processing
  - Amount of memory used
  - How often failures can be permitted
  - Level of security
  - Ease of modification
  - Cost of development
  - Platforms the product must run on
- Non-functional requirements are more abstract than functional requirements
- Functional requirements are tied to specific pieces of code, but non-functional requirements are properties of the whole system

Problems with requirements in traditional processes

- It's really hard to figure out all the requirements before doing any coding and looking at prototypes
- The world changes quickly, especially in technology, and people's desires change
- Writing all the requirements takes a lot of work, creates large documents, and costs a lot of money
- The waterfall process means that nothing is ready for a long time (often years) after the project starts, and some projects get canceled

Requirements in agile processes

- Agile developers try not to write requirements at all
  - But you have to start with something…
- Stakeholder needs are turned into lists called **product backlogs**
- A **product owner** adds to the product backlogs and prioritizes them
- High priority items are chosen for each **sprint**, the agile term for a development iteration

How Scrum tries to make changing requirements cheap and easy

- Delay choosing requirements as long as possible
  - Stakeholder needs can be easily added or removed from the product backlog
  - Requirements are set only for the product backlog items (PBIs) when they're implemented on a sprint
- Delay refinement as long as possible
  - PBIs are broken down until they're small enough and detailed enough for a single sprint
  - User-level requirements are refined into operational- and physical-level requirements for the sprint where they're implemented
- Avoid writing requirements altogether
  - Instead of writing down physical-level requirements, talk to the stakeholders and implement what they say in the sprint
- Determine requirements in light of current product features
  - Because agile methods iterate on an existing product, everyone can see which features would be most useful next

Stating specifications in traditional processes

- Specifications are usually made in declarative English (or appropriate natural language) sentences
- Problem: English is vague and confusing
- Rules for good technical writing:
  - Write complete, simple sentences in the active voice
  - Define terms clearly and use them consistently
  - Avoid synonyms
  - Group related material into sections
  - Use tables, lists, indentation, white space, and other formatting aids
- Use "must" or "shall" to describe behaviors the product must do

Testable requirements

- Requirements should be **testable** or **verifiable**
- This means that there can be a process for testing whether the product meets the requirement
- **Bad requirement:**
  - The product must display query results quickly.
- **Good requirement:**
  - The product must display query results in less than one second.
- The bad requirement isn't testable because "quickly" is subjective
- The good requirement is testable because we can time the finished system

Requirements traceability

- We want a clear relationship between a requirement, a part of the design, the code that implements this design, and the tests that verify it
- Being able to connect the requirements to later stages of development is called **requirements traceability**
- To make requirements more traceable, each specification should state only a single requirement
  - This kind of specification is called **atomic**
- **Non-atomic specification:**
  - The product must display a list of previous commands and the results of commands, each in its own window.
- The goal is simplicity and clarity
- A long list of simple requirements is better than a short list of confusing, complex requirements

Stating specifications in agile processes

- Agile developers have some documents like product vision statements and product backlog items
- A very common way to describe requirements is through **user stories**
- A user story describes a function that the product provides to users
- Sometimes a big story that is a huge chunk of the application is called an **epic**
- Sometimes a story that would take several sprints to implement is called a **feature**
- A story that can be implemented in a single sprint is a **sprintable story** or an **implementable story**
- **Note:** Some agile people *only* use the term user story for sprintable stories

User voice form

- A common way of expressing user stories is user voice form:
  - As a <*role*>, I want to <*activity*> so that <*benefit*>.
  - <*role*> is replaced by a user role, which is some category of user
  - <*activity*> is a function that the system does
  - <*benefit*> shows the value of the activity but is an optional part of user voice form
- Example:
  - As a payroll clerk, I want to enter salary data so that payrolls will use adjusted salaries.

Eliciting stakeholder needs in traditional processes

- It can be difficult to discover what stakeholders actually want from a product
- Some approaches:
  - **Interviews:** Ask individual stakeholders what they want and record the answers
  - **Observation:** Watch the users doing tasks, asking them to describe the actions they're taking
  - **Focus groups:** Informal discussion with six to nine people and a facilitator
  - **Workshops:** A meeting focused on documenting the desires of many stakeholders
  - **Prototypes:** Let stakeholders respond to different version of a product
  - **Document studies:** Read documents associated with the business that needs the product
  - **Competitive product studies:** Analyze similar existing products for strengths and weaknesses

Eliciting stakeholder needs in agile processes

- Agile processes don't focus on getting all the requirements up front
- Instead, a cornerstone of the agile approach is constantly getting feedback, allowing for quick responses
- The product itself becomes an evolving prototype that it easy to understand and unlikely to become obsolete
- Potential problems:
  - Stakeholders can overreact to current problems and lose sight of the big picture
  - Agile methods give a lot of power to the few stakeholders who give feedback, and others might be ignored

Requirements management in traditional processes

- Projects start with a product mission statement giving business requirements
- Requirements analysis is the process of gathering stakeholder needs and using them to turn the mission statement into a list of requirements specifications
- The result is a document called a software requirements specification (SRS)

Requirements management in agile processes

- The mission statement or other high-level needs are used to write big user stories
- Working with stakeholders, the team refines sprintable stories into operational-level and physical-level requirements
- The product owner has the responsibility to update the product backlog as the product evolves

Kinds of requirements modeling

Model	Show	Typical UML Diagram
Use Case Models	A product interacting with its environment, often **actors** who take on roles	**Use Case Diagram**
Conceptual Models	Relationships between entities	**Class Diagram**
State Diagrams	The states a product can be in and the transitions between those states	**State Diagram**
Decision Trees and Tables	What a product should do under various conditions	**Activity Diagram**
Data Flow Diagrams	How data enters, is processed, and leaves the product	**Activity Diagram or Sequence Diagram**

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

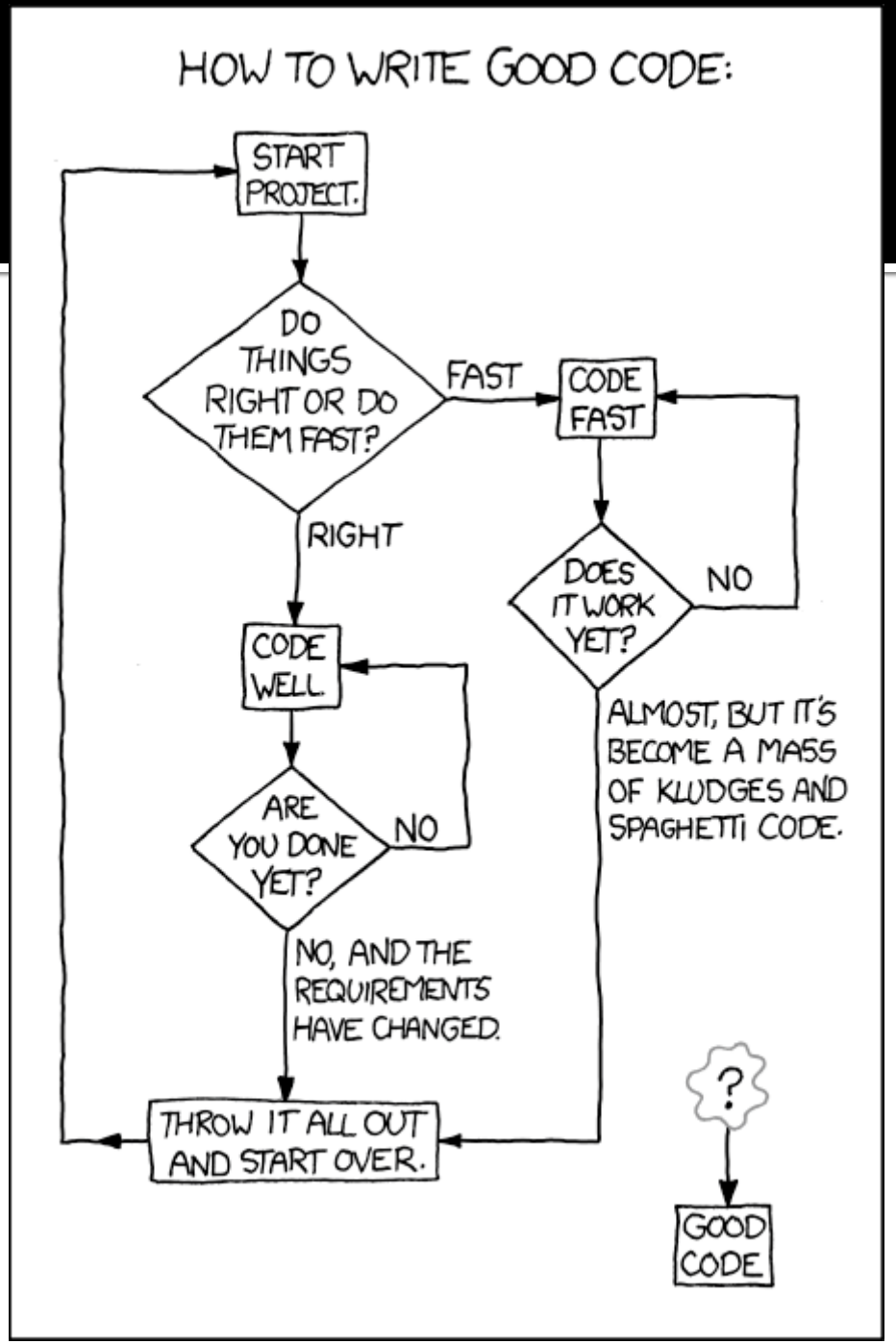
- **Models leave out details**
- Models are useful to help understand a complex system
  - During requirements engineering, models clarify what an existing system does
  - Or models could be used to plan out a new system
- Models can represent different perspectives of a system:
  - **External:** the 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

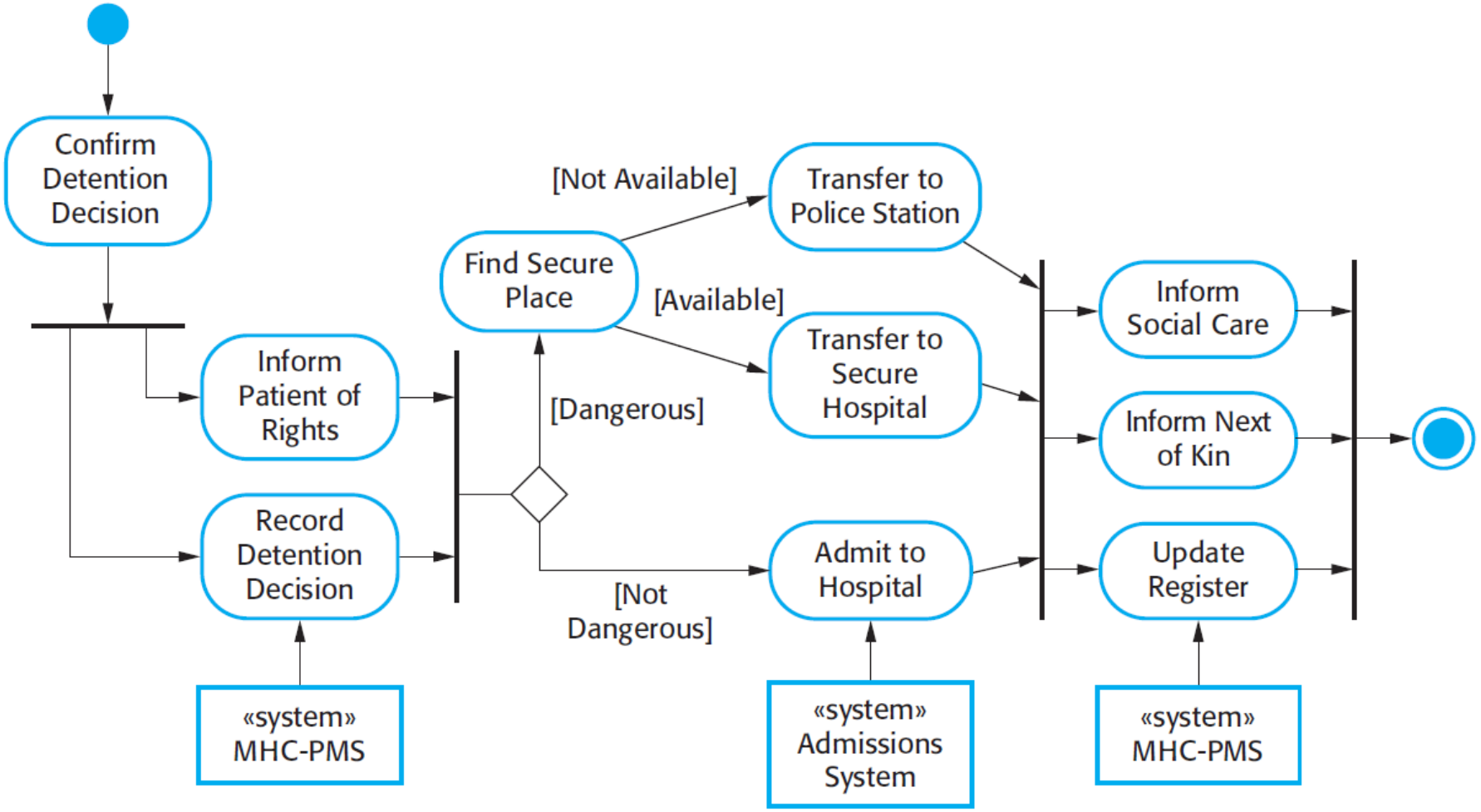
- The **Unified Modeling Language** (UML) is an international standard for graphical models of software systems
- A few useful kinds of diagrams:
  - Activity diagrams
  - Use case diagrams
  - Sequence diagrams
  - State diagrams
- Class diagrams are important enough that we'll talk about them in greater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

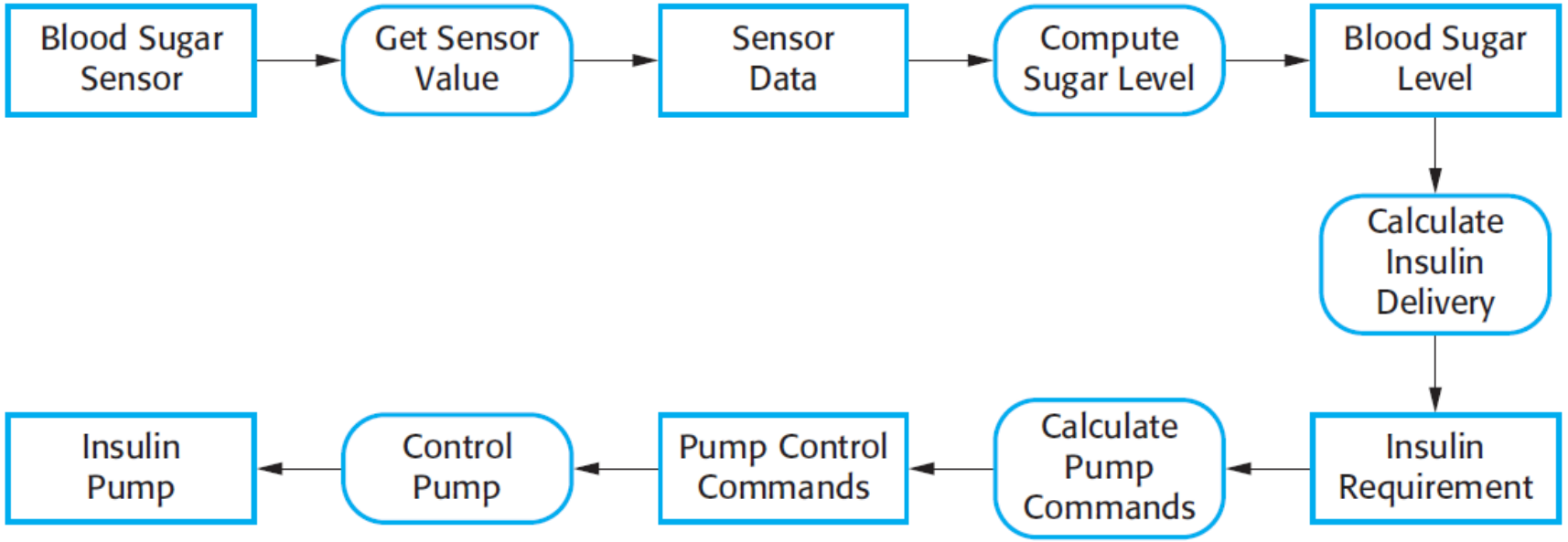


More detailed activity model



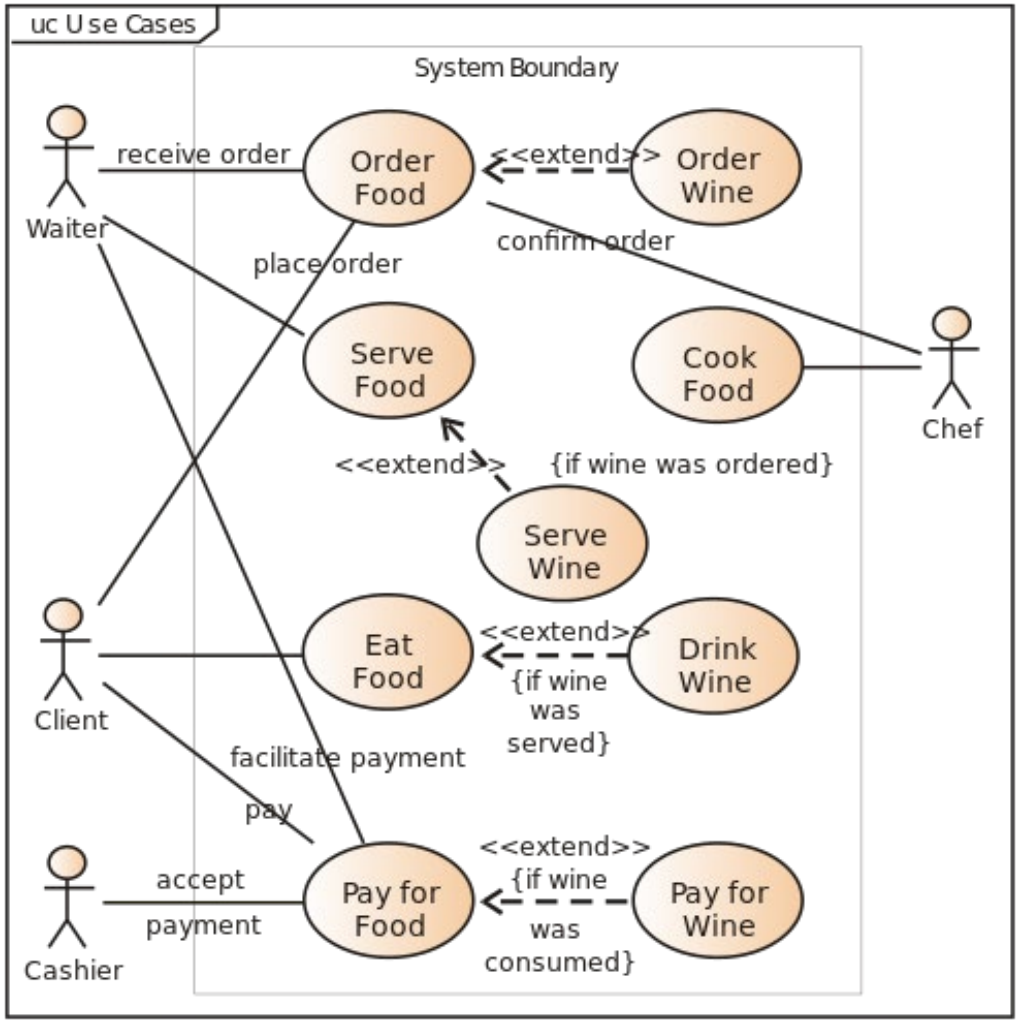
Data-driven modeling

- 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



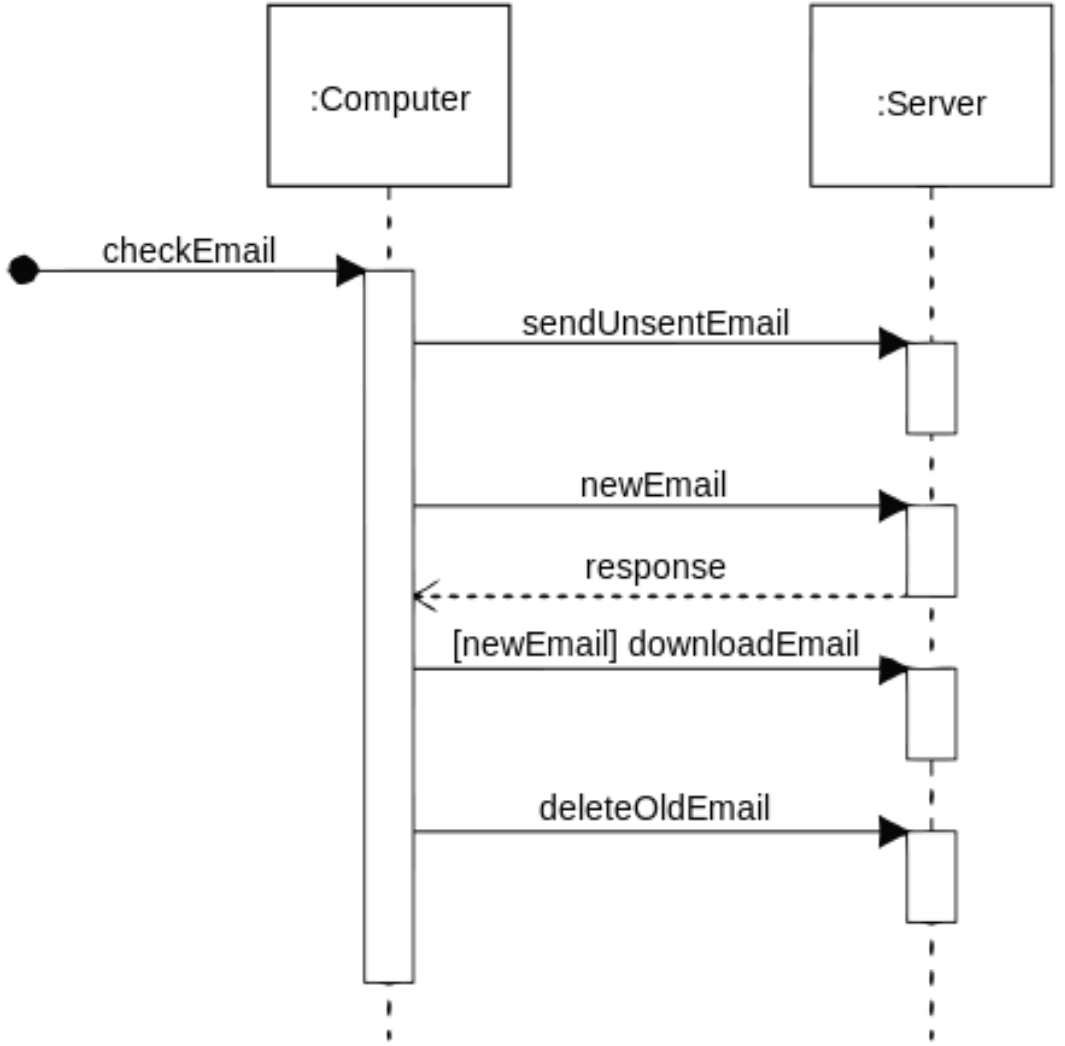
Use case diagrams

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



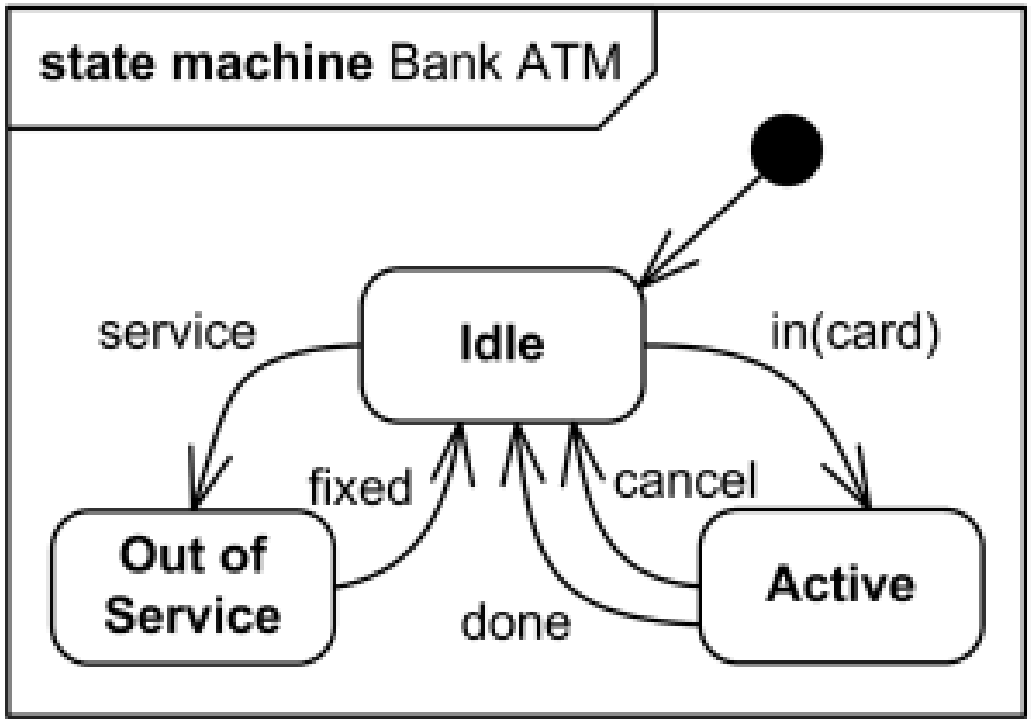
Sequence diagrams

- 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:



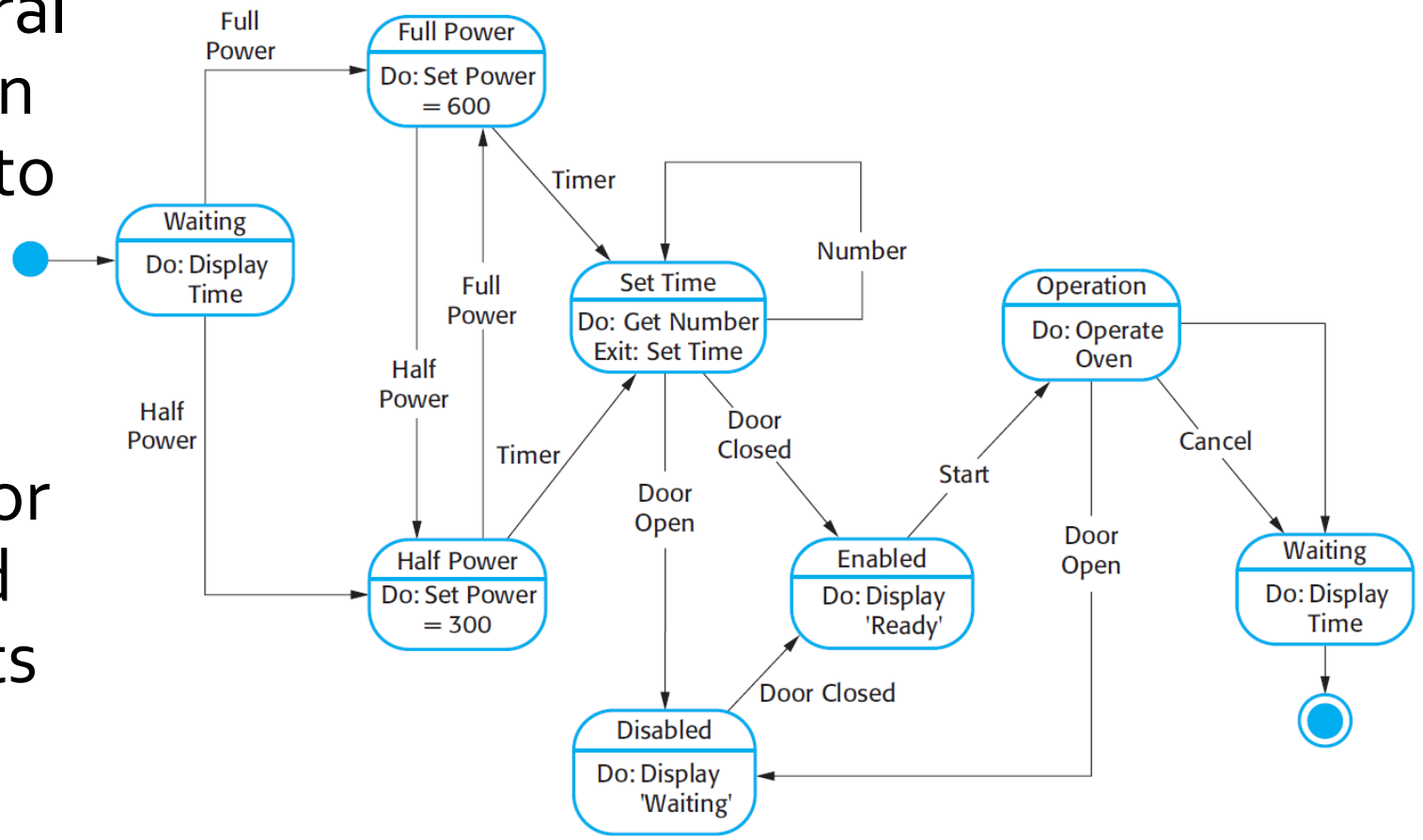
State diagrams

- 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:



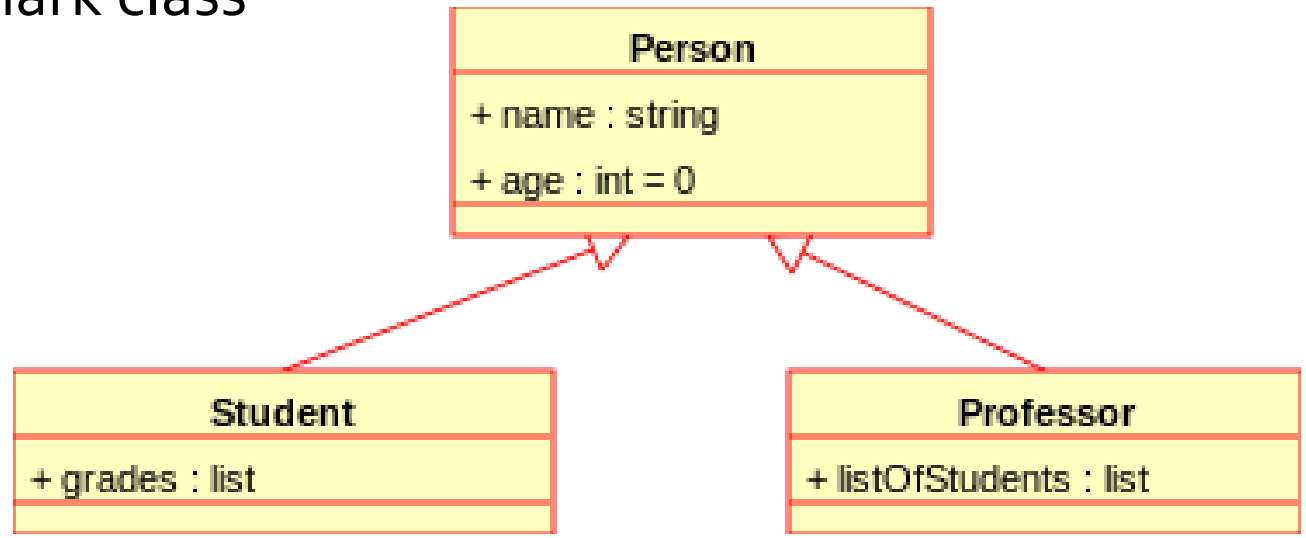
Event-driven modeling

- 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

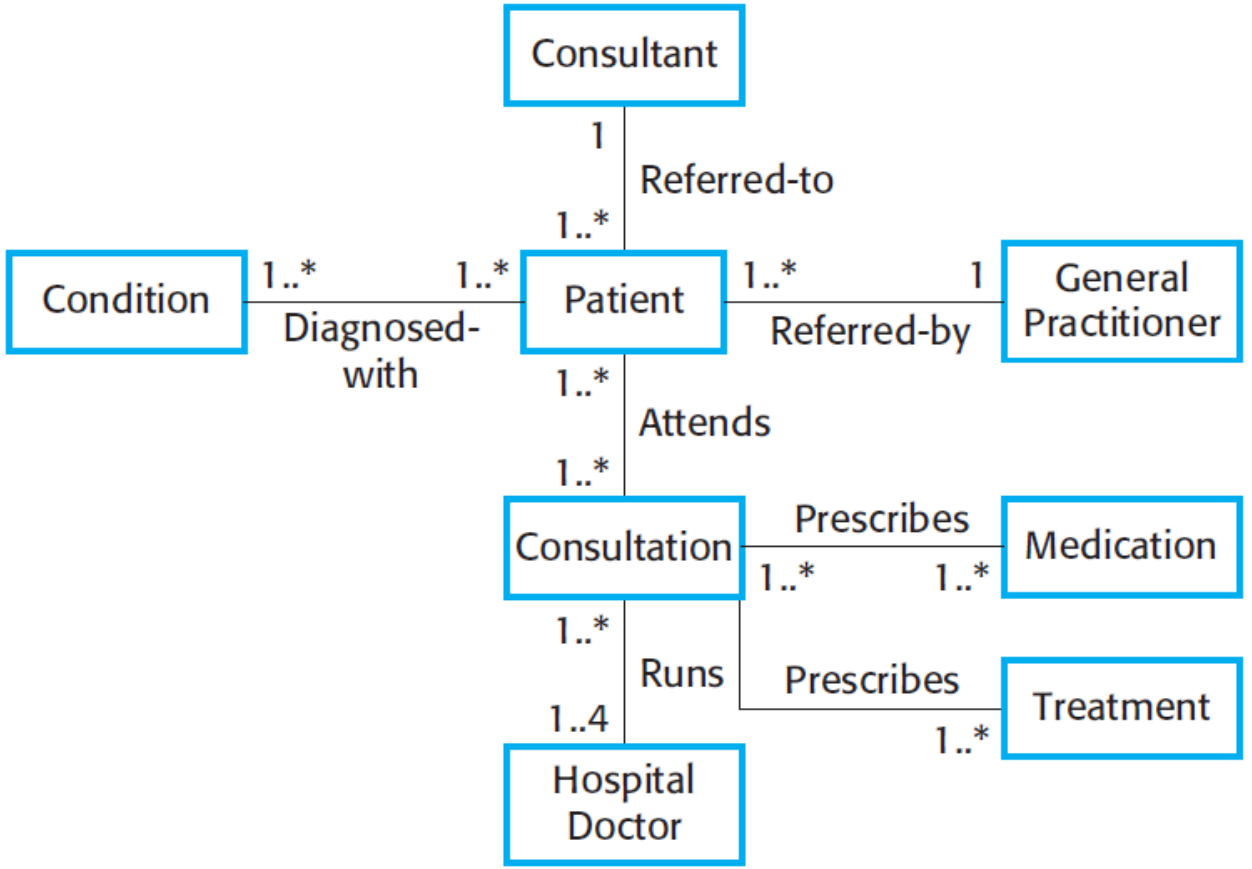
- Class diagrams show many kinds of relationships
- The **classes** being described often (but not always) map to classes in object-oriented languages
- The following symbols are used to mark class members:
  - **+** Public
  - **–** Private
  - **#** Protected
  - **/** Derived
  - **~** Package
  -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



Relationships

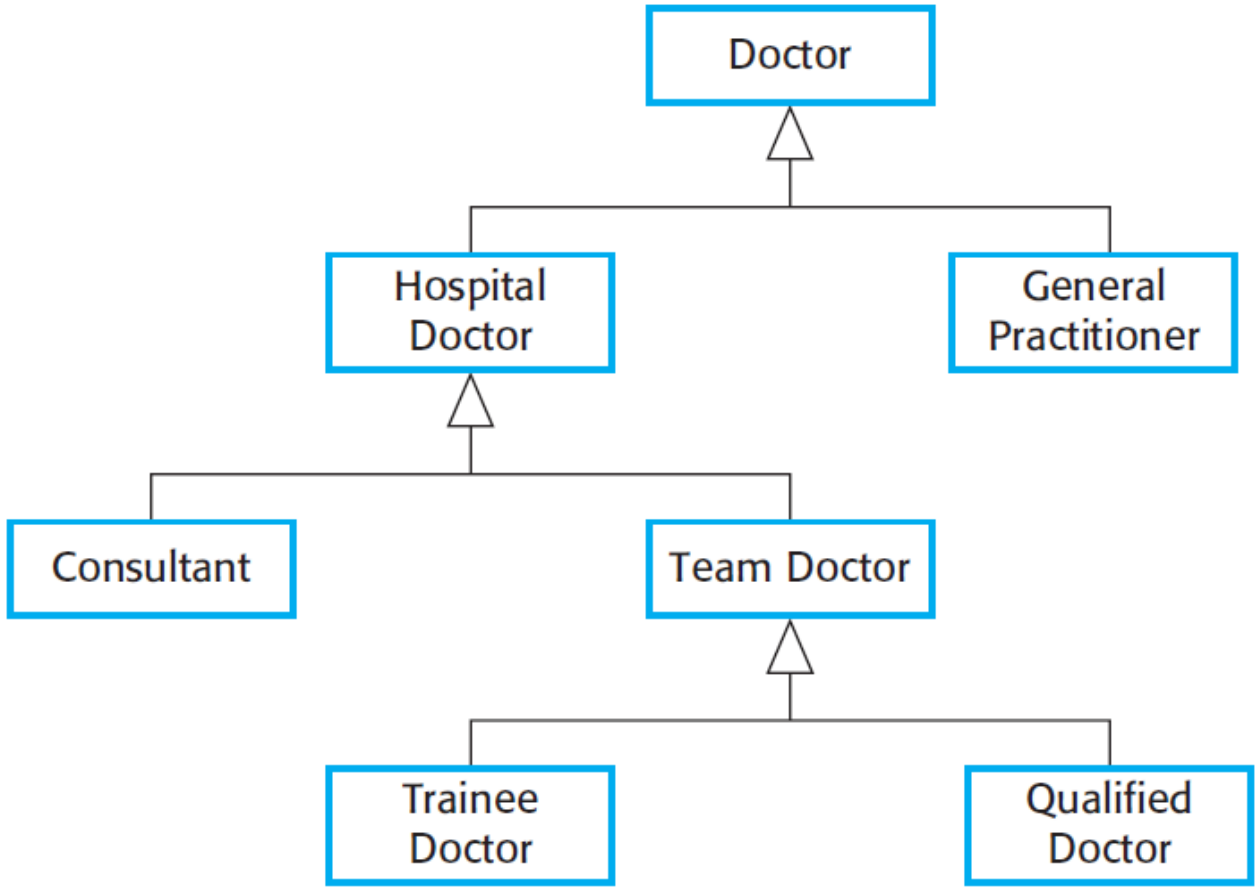
- Associations between classes can be drawn with a line in a class diagram

- 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



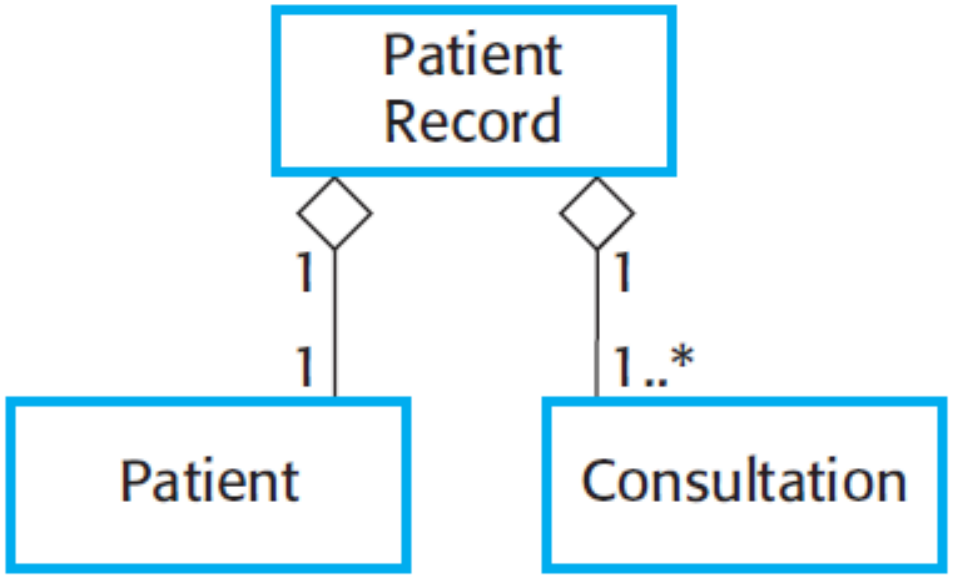
Generalization

- 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**



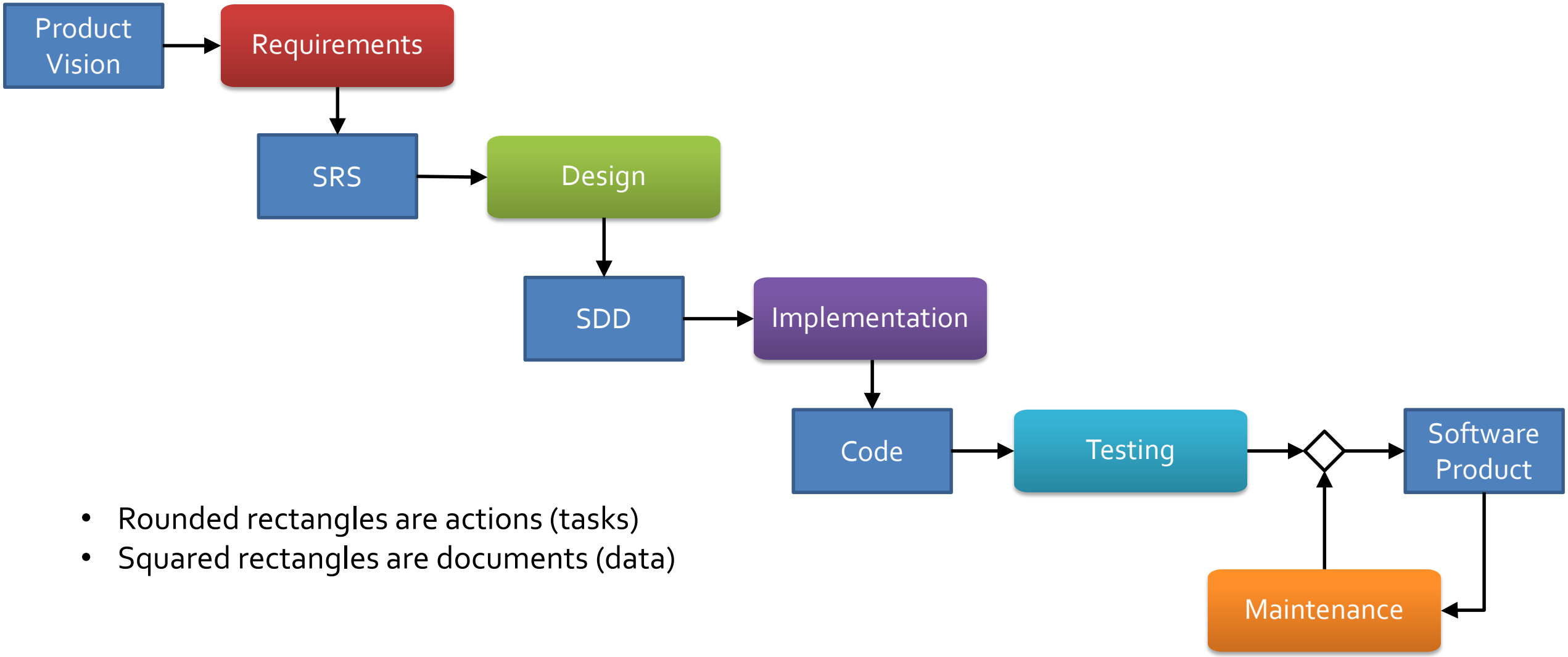
Aggregation

- 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



Software Processes

Waterfall lifecycle model



- Rounded rectangles are actions (tasks)
- Squared rectangles are documents (data)

Advantages of waterfall

- The whole product is specified
- The project to create it is planned early
- This approach is important for large and complicated products from a management perspective
  - Size, cost, delivery dates, etc.
- By comparing to the plan, it's easy to tell if a product is on-time and on-budget
- If it isn't, managers can take actions
  - Increase time, increase budget, reduce scope, etc.

More advantages of waterfall

- If each step is done completely and correctly, all mistakes are found before moving on to the next step
  - This ends up being the major disadvantage of waterfall, too, since mistakes usually propagate to future steps
- Good documentation is created for each step
  - This is really important when new people are added to the project
- Each phase is distinct, allowing it to be carried out by teams that specialize in that phase
  - For multiple projects, appropriate teams can be scheduled for maximum efficiency

Disadvantages of waterfall

- Requirements can't change
  - But they usually do
  - If requirements change, all the advantages of waterfall's predictability disappear too
- Even when requirements stay the same, it's hard to be complete and consistent in documenting them
- Creating all the documentation for waterfall is expensive
- If you have separate teams for each phase, each team has to learn what has already been done

More disadvantages of waterfall

- Because there are so many teams, a lot of management is needed
  - Drives up the cost
  - Heavyweight processes are ones with a lot of documentation and management
- There's no product until completion of the entire project
  - Could take years
  - We don't realize the problems until the product is available
  - Clients might not want the product anymore

To waterfall or not to waterfall?

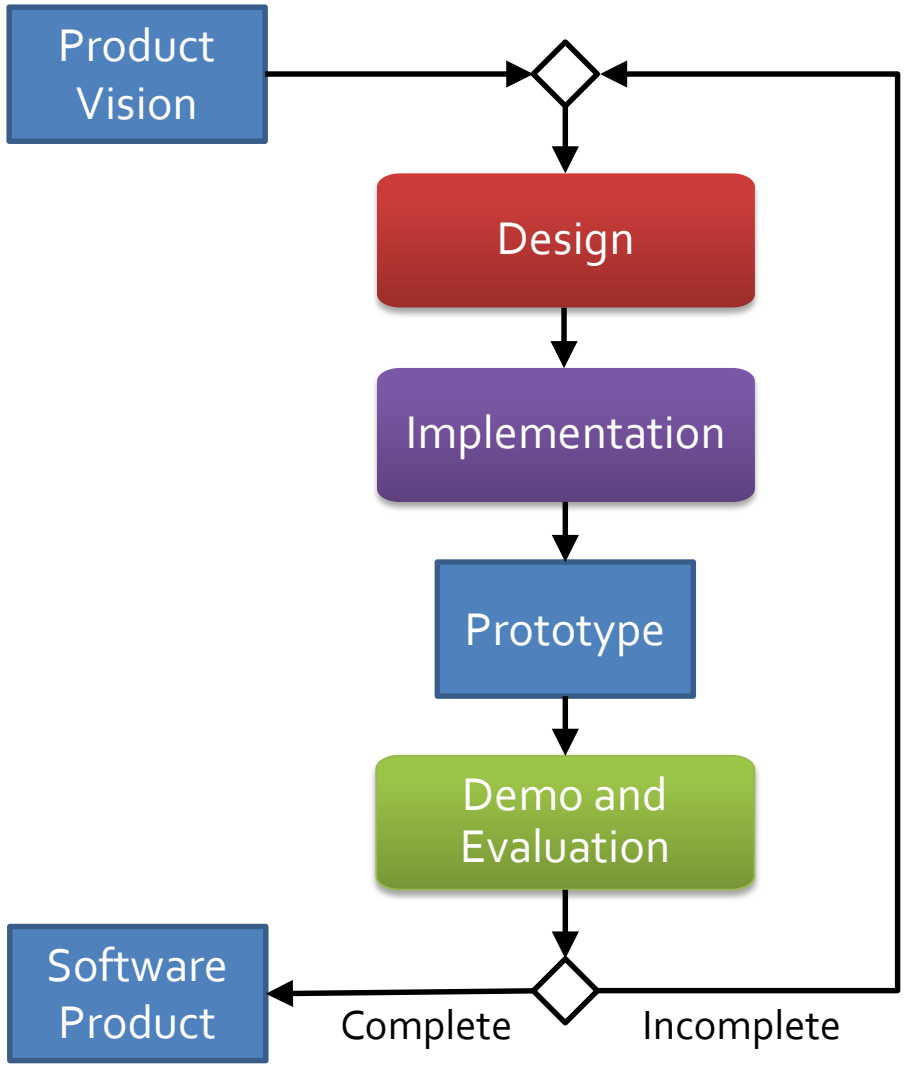
- Waterfall was the only process for a long time
- Its track record isn't great
  - Success only about 25% of the time historically, but the rate is improving
- Waterfall only works when the requirements are stable
- Waterfall has a lot of overhead
  - Might be justified for large projects
  - Isn't justified for small projects
- Use waterfall only for large projects with stable requirements or when there are very high safety, security, or reliability requirements

Prototyping

- A **prototype** is a working model of a finished product
  - It can model a part or the whole
- Prototypes can help offset problems with the waterfall model
- Prototypes are particularly helpful with testing out UI decisions
- Prototypes are easy(ish) to make and change
  - Try out several!
  - See which one is the better design
- **Throwaway prototypes** are just used for making specifications and then thrown out
- **Evolutionary prototypes** are modified into the final product

Prototyping process

- Prototypes can be used within the waterfall model
- Or they can be used for an entirely prototype-based lifecycle model:
- This idea is what incremental and agile processes are built around



Advantages of prototyping

- Changes to specifications are easy to handle
- Customers are more likely to get what they want (since they get regular opportunities for feedback)
- Customers can get (potentially) useful software quickly
- Not much documentation or management is needed
  - **Lightweight** development process

Disadvantages of prototyping

- Without the planning of a process like waterfall
  - It's hard to predict a reasonable deadline for the final product
  - It's hard to predict the budget
- Product design might be bad since the product evolved without following a plan
  - The biggest problem here is maintainability: How can new features be added?
- An undisciplined process can have poor quality control
  - The product might be unreliable or buggy

Risk management

- A **risk** is an event with negative consequences
  - Losing source code
  - Losing a team member
  - Finding an unexpected design flaw
  - Underestimating the time needed to write a piece of code
- Business people think about risk a lot
- Risk management is identifying, analyzing, controlling, or mitigating risks
- Risk management *should* be incorporated into all software lifecycle processes

Iterative and Incremental Processes

Iterative and Incremental Processes

- An **iterative process** contains repeated tasks
  - Example: While debugging code, you might run tests, do fixes, run tests, do fixes, and so on
- An **incremental process** produces output in parts
- Processes can be either iterative or incremental, both iterative and incremental, or neither
- The purest version of waterfall is *neither*
  - It's not iterative because each phase is separate and not repeated
  - It's not incremental because a working product is only available at the end

Iterative processes

- Iteration is the main way you get quality
  - It's just so hard to get it right the first time!
  - Software development still involves significant trial and error
- Even the waterfall model usually has iterative steps in practice
- Prototype evolution is iterative
- The problem with iteration is **rework**
  - Redoing or throwing out previous work

Incremental processes

- Iteration is found lurking everywhere to greater or lesser degrees, but being incremental is more binary
- To be incremental, final products must be produced along the way
- Waterfall is *not* incremental because the products produced along the way are just used for the next step

Agile

- Versions of waterfall were the only commonly used software development model until the 1990s
- A lot of people were unhappy with it
- In response, some developers created the Agile Manifesto, a statement about developing software that was diametrically opposed to waterfall
- The ideas caught on, and many developers embraced the idea, creating a series of different methods
- Sometimes businesses claimed to be changing over to agile methods but really just renamed parts of their waterfall approach

Agile manifesto

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- ***Individuals and interactions*** *over processes and tools*
- ***Working software*** *over comprehensive documentation*
- ***Customer collaboration*** *over contract negotiation*
- ***Responding to change*** *over following a plan*

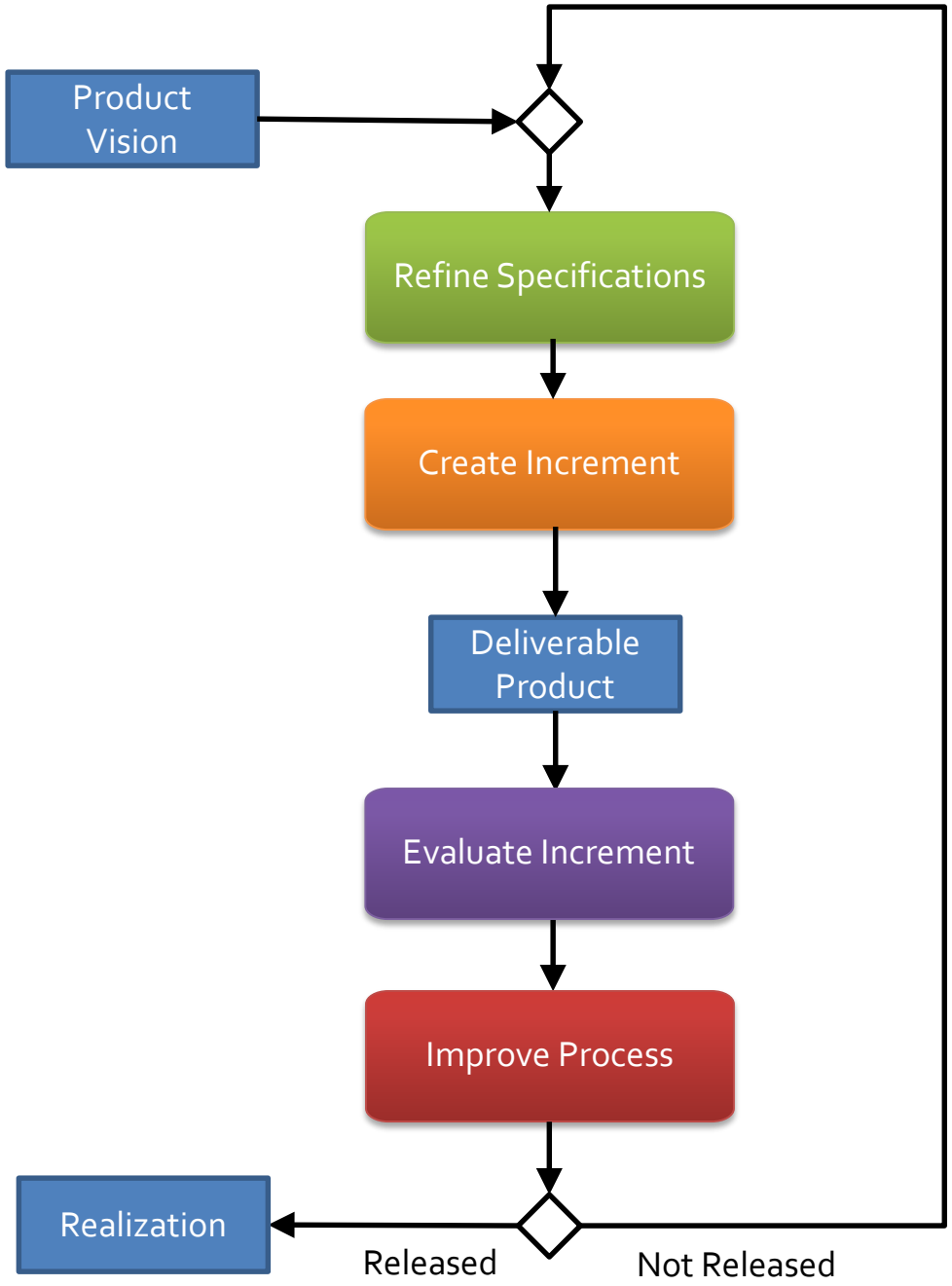
That is, while there is value in the items on the right, we value the items on the left more.

Agile characteristics

- The ideas caught on, spawning specific methods such as Extreme Programming, the Crystal Method, Dynamic System Development Method, and Scrum
- These methods all have the following characteristics:
  - Incremental process with increments ranging from a week to a few months
  - Customers are closely and continuously involved in the product
  - Lightweight process minimizing documentation and management tasks
  - Test driven, using automated test suites to avoid the problems of frequent code change

Agile lifecycle

- Agile processes are similar, following a lifecycle much like the one on the right



Agile advantages

- Product specifications can change without destroying all the work that's been done
- Customers get a software product quickly
  - With new versions coming frequently
- Bad projects can be canceled early
- Time is saved because of lightweight requirements for documentation and management
- Duplication of effort is usually reduced

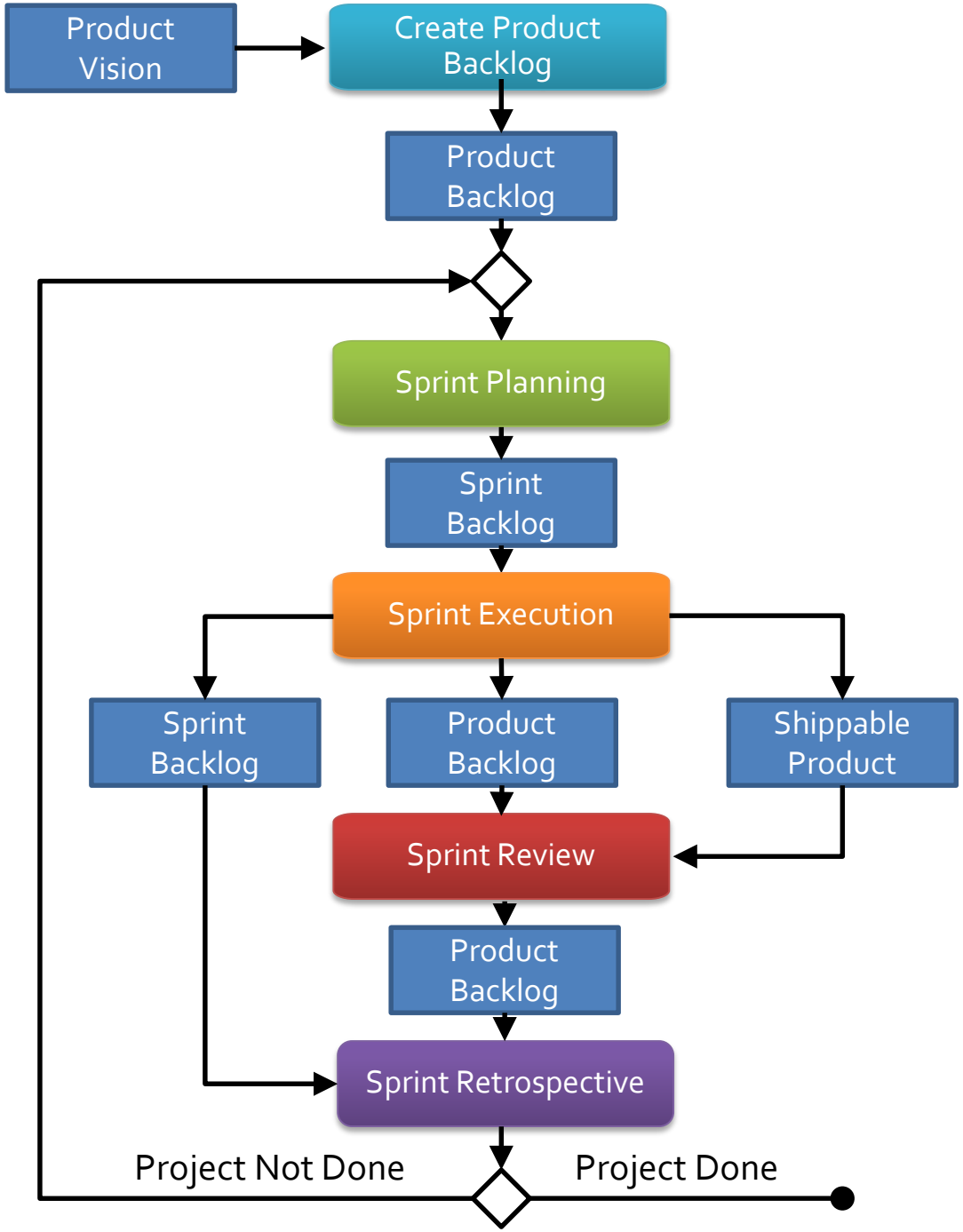
Agile disadvantages

- Customers have to be involved constantly, but most customers don't want to spend their time giving feedback
- Continuous refinement of a product can lead to a bad design through an evolution of ideas that seemed like a good idea at the time
- For large projects, it's hard to coordinate many teams on a product that is evolving unpredictably without documentation
- It's hard to predict the outcomes of agile methods

Scrum

Scrum process

- Like other workflows, Scrum can be modeled with an activity diagram showing familiar steps
- Everything is built around a cycle called a **sprint**
- Because sprints repeat, the process is iterative
- Because each sprint produces a shippable product, the process is incremental



Sprints

- Recall that agile methods are built around a product backlog, containing high-level descriptions of the desired features of the product
  - Items can be added to or removed from the product backlog at any time
- Some of the product backlog is chosen for a sprint
  - Making the **sprint backlog**
- The sprint backlog is implemented, making a new shippable product
- A **sprint review** allows customers to give feedback on the product
- The **sprint retrospective** is used to figure out how to do the next sprint better

Scrum roles

- **Product owner (PO)**
  - Responsible for what's in the product
  - Customer representative to the other developers
  - Updates the product backlog
- **Scrum master (SM)**
  - Guides the team through the Scrum process
  - Facilitator and coach
  - Protects the team from outside interference
- **Team members**
  - People who decide how to build the project and build it
  - Typically, everyone works on everything

Scrum artifacts

- **Product backlog**
  - A prioritized list of product features that haven't been implemented yet
  - **Product backlog items (PBIs)** are the elements of this list
  - Priorities are based on business value
- **Sprint backlog**
  - Subset of PBIs
  - Tasks needed to complete them
  - Estimates of effort needed for each one
- **Potentially shippable increment (PSI)**
  - Product that could be shipped to the customer (though maybe without all the desired features)
  - A PBI on the sprint backlog that wasn't finished goes back into the product backlog

Scrum activities

- **Product backlog creation**
  - The PO creates the product backlog for the first time, using customer input
- **Product backlog refinement**
  - The PO constantly adds and deletes PBIs from the product backlog based on feedback from stakeholders
- **Sprint planning**
  - The PO, SM, and other team members select PBIs, maybe with a particular sprint goal
  - PBIs are chosen by priority, taking into account how much can be done by estimating the work for the tasks for a PBI
- **Sprint execution**
  - Everyone performs the tasks to implement the sprint backlog PBIs
- **Sprint review**
  - A product demo where stakeholders discuss what was added and how they feel about it
  - Goal: improving the product
- **Sprint retrospective**
  - The team discusses what went well, what didn't, and how the next sprint can be better
  - Goal: improving the process

Managing the product backlog

- The product backlog is a prioritized list of PBIs
- Each PBI consists of
  - Specification
  - Priority
  - Estimate of effort
  - Acceptance criteria

PBI priorities

- In addition to the specification of functionality, every PBI should have a priority
- Priorities express how important the PBI is and can be expressed as a number or a rubric (low, medium, high, critical)
- The PO sets the priorities based on stakeholder feedback
- Dependencies also determine priorities: If X is needed for Y, then the priority of X must be at least as high as Y
- High-priority PBIs should be small enough to do in a single sprint

PBI effort estimates

- Each PBI must have an effort estimate
- High-priority, sprintable PBIs need precise estimates (such as person-days), to aid in sprint planning
- Low-priority, abstract PBIs are further from sprintable status and only need rough estimates (small, medium, large, gigantic)
- As PBIs are refined, their effort estimates need to become more precise

PBI acceptance criteria

- How do we know when a PBI is done?
- Acceptance criteria are checks a user can do to see if a PBI is finished and correct
- Often, these form a test suite used by developers
- Following the same pattern of steady refinement, high-priority PBIs should have detailed acceptance criteria
  - These acceptance criteria might be further refined during the sprint

Product backlog refinement

- **Refining** or **grooming** the product backlog means:
  - Adding, removing, or modifying PBIs
  - Making PBIs nearing the top of the product backlog more detailed
  - Re-estimating and re-prioritizing PBIs
  - Adding acceptance criteria to PBIs
- Refinement happens during sprint review
- It should happen at least once during a sprint to make sure there are enough sprintable stories for the next sprint
- A PO can use a spreadsheet to manage the product backlog, but there are also specialized tools

Estimating work and timeline

- Two pieces of information are needed: The size of the job and the speed of the team
- PBIs are estimated by **story points** or **ideal hours**
- One or two story points is supposed to be how much effort the smallest stories take
  - Bigger stories are estimated relative to that size
- An ideal hour or a person hour is the amount an average developer can accomplish in one uninterrupted hour of work
- Story points are more commonly used, since they're easier to estimate

Velocity

- **Velocity** is the amount of work done per sprint
- After a sprint, story points or ideal hours can be added up to see how much got done
- Past velocities can be used as a guide for how many story points can get done when planning the next sprint
- Ideally, tracking this information will help get better estimates of story points and ideal hours for other stories and also a better estimate of team velocity

Sprinting

- **Sprinting** is actually doing the implementation
- Sprinting is considered a **time-boxing** technique, where the amount of work done is based on the time available
  - Rather than letting time expand as needed to finish a task
- For a given project (and at a given company) sprints are usually the same length, somewhere between a week and a month
- Short, consistent sprints are easier to plan and track and give rapid feedback
- If PBIs can't be finished during a sprint, they go back on the product backlog
- If a team finishes all PBIs before the sprint is over, they can get another one from the PO

Sprint review

- At the end of a sprint, there is a sprint review to reflect on how the product is changing
- All stakeholders are invited
- Sprint review outline:
  - Starts with the overall sprint goal and the PBIs in the sprint backlog
  - Team lists the PBIs completed and explains why some didn't get done
  - New aspects of the product are demonstrated
  - Everyone discusses how to make the product better
- Results of the review are used for planning the next sprint

Sprint retrospective

- At the end of a sprint, there's also a sprint retrospective
- Only the development team, including the PO and the SM, are invited
- The retrospective is for analyzing how the team is working and how to improve
- Improvements tend to be clear when a new team is working on a new product
  - It may still take several sprints for an improvement to get fully integrated into the process
- Over time, the team can become comfortable with the process, but finding improvement opportunities is still important

Software Quality Assurance

Quality assurance

- **Quality assurance (QA)** is a system for making sure the product satisfies stakeholder needs
- QA focuses on two distinct goals:
- **Validation**
  - Testing if the product satisfies stakeholder needs
  - "Are we building the right product?"
  - Example: Does the customer want steak and fries?
- **Verification**
  - Testing if the product satisfies needs properly
  - "Are we building the product right?"
  - Example: Are the steak and fries cooked well?

Defect prevention

- There is no one way to prevent defects
- Instead, preventing defects must be built into the software development processes that the entire organization uses
  - **Process improvement** is making a process better
  - Training and education are necessary
- **Process guides** such as documentation standards and style guides help
- Using well-studied **design methodologies** (such as OOP) can help

Reusing ideas

- Reusing **design architectures** that have been successful in the past can prevent defects
  - Examples: MVC and pipe-and-filter
- **Design patterns** are standard patterns for OOP classes
  - Examples: decorator and factory
- Using well-studied algorithms and data structures helps a great deal
- Reusing code (often from libraries) is smart, especially since those libraries have been tested thoroughly

Formal methods and prototypes

- **Formal methods** include systems for mathematically checking that code does what it's supposed to
  - Not all code can be modeled mathematically
  - Yet some of these systems have found bugs in real software, such as TimSort, the most commonly used sort in Python and Java
- Prototypes let us explore what defects might happen before putting them in the final product
  - The opposite end of the spectrum from formal methods, since prototypes are practical rather than theoretical

Tools

- Many tools help reduce defects
- Version control tools help track code over time
- Configuration management tools allow changes in one tool to automatically update other tools
  - Examples: Puppet and Ansible
- **Integrated development environments (IDEs)**, once called computer aided software engineering (CASE) tools, can integrate many useful tools for defect prevention
  - Syntax highlighting
  - Two-way translation between code and UML models
  - Style checking

Defect detection and removal

- A good process can't keep out all defects
- Some defects will show up and must be found and removed
- Defect detection and removal techniques fall into two categories:
  - **Review and correct**
  - **Test and debug**
- Review and correct methods look at the code while test and debug methods look at the product in operation

Review and correct methods

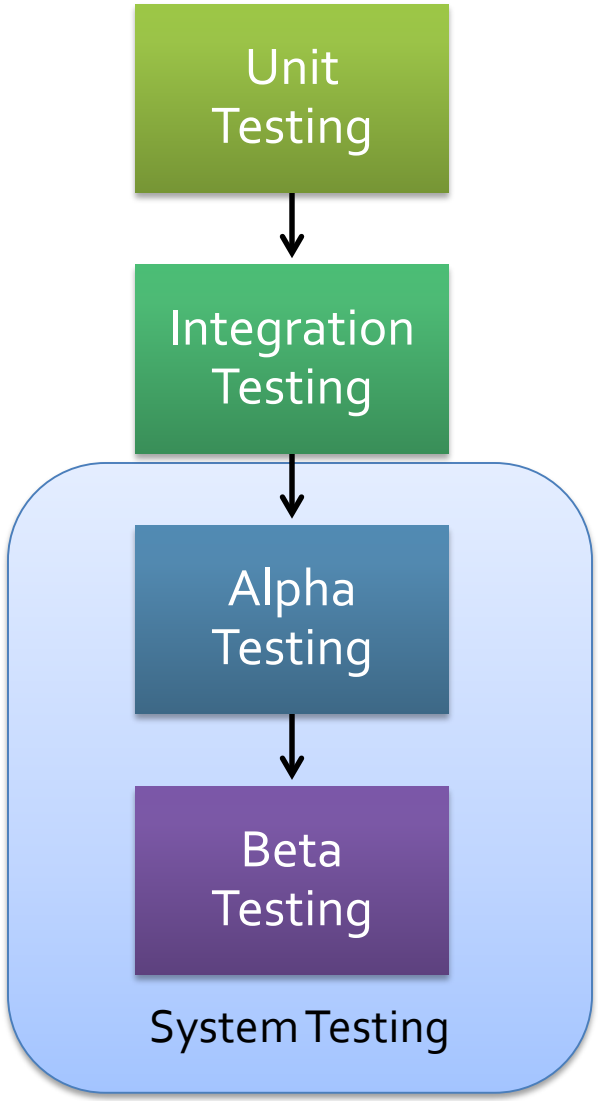
- There's a formal name for just looking at your code for errors: a **desk check**
- A **walkthrough** is when you explain your code to someone else
- An **inspection** is a more formal process with trained inspectors
- Inspection roles:
  - **Moderator** schedules and runs the meeting and distributes the code
  - **Author** of the code
  - **Reader** who guides the meeting
  - **Recorder** who takes notes
  - **Inspectors** who check code before and during the meeting

Test and debug

- **Testing** software helps find cases that are not obvious from looking at the code
- Software testing has some jargon:
  - A **failure** is a deviation between actual behavior and intended behavior
  - A **fault** is a defect that can give rise to a failure
  - A **trigger** is a condition that causes a fault to result in a failure
  - A **test case** is a set of inputs and program states
  - A collection of test cases is a **test suite**
- **Debugging** is using trigger conditions to find and fix faults

Overview of testing

- **Unit tests** test a small piece of code (method or class) in isolation from other code
  - Often done by the author
- **Integration tests** test several small pieces of code together
  - By the author, a testing team, or both
- **Alpha and beta tests** test the whole product
  - Alpha tests usually have a testing team
  - Beta tests include users



Breaking it all down

Defect Elimination

Defect Prevention

- Process Guides
- Analysis and Design Methods
- Reference Architectures
- Design Patterns
- Data Structures and Algorithms
- Software Reuse
- Prototyping
- Version Control
- Configuration Management
- IDE Tools
- Training and Education

Defect Detection and Removal

Review and Correct

Style and Standards Checkers

Spelling and Grammar Checkers

Reviews

- Desk Checks
- Walkthroughs
- Inspections

Test and Debug

Regression Testing

Unit Testing

Integration Testing

System Testing

- Alpha Testing
- Beta Testing

User Interaction Design

Interaction design

- **Interaction design** is planning out the **user experience (UX)** for a software product
- It cares about how the product looks and sounds (and, one day, smells?) and how the user gets output and puts input into it
- This field used to get little attention from computer scientists, but it's really important
  - Apple is a great posterchild for showing off the value of UX
  - Even Microsoft, maligned for its user interfaces, has invested lots of money studying how to make windows and icons easier to use
- UX is part of the field of **human computer interaction (HCI)**, which combines ergonomics, physiology, psychology, and graphic design with computer science
- The quality of a user interface is called its **usability**

User interaction design goals

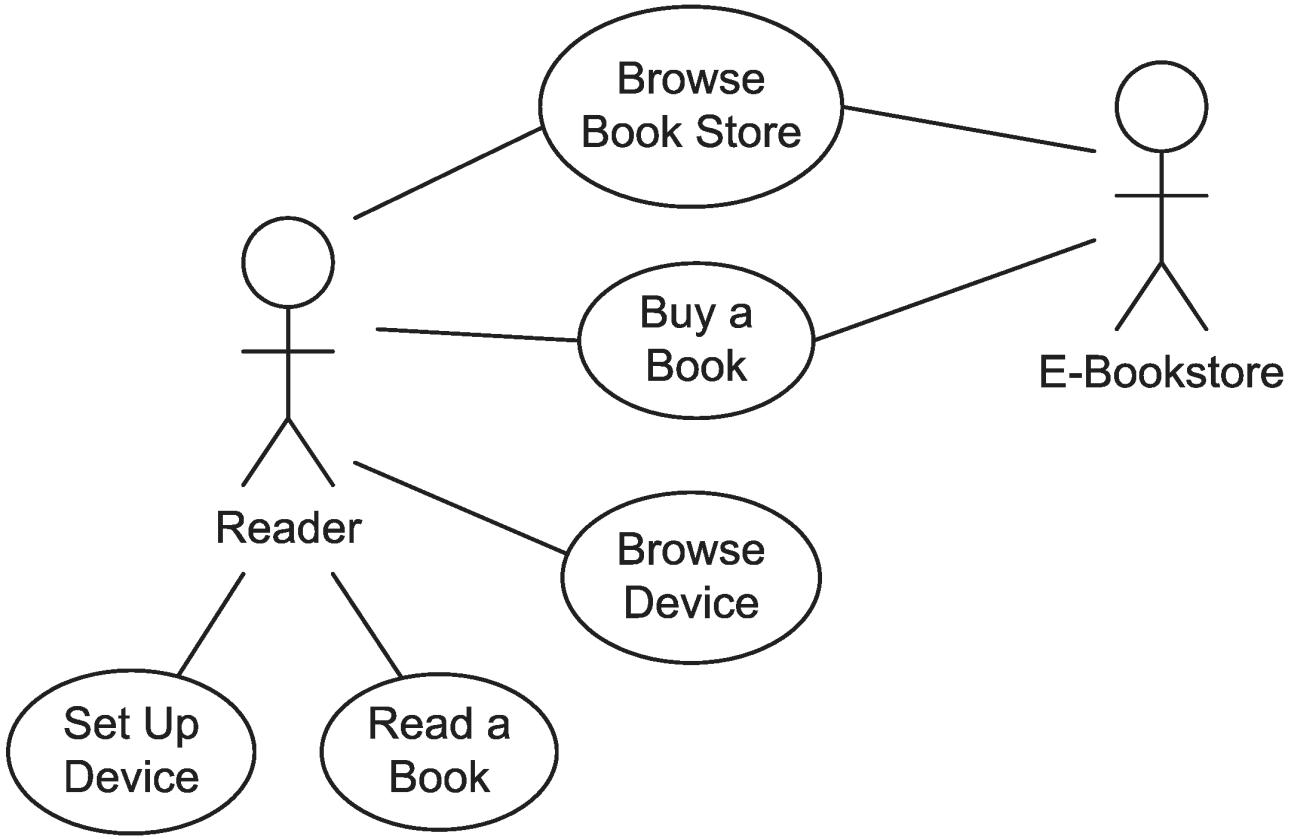
- **Effectiveness:** User can access all the features they need
- **Efficiency:** Users can achieve their goals quickly
- **Safety:** Users and computers aren't harmed
- **Learnability:** Users become proficient quickly
- **Memorability:** Users regain proficiency quickly after time away from the product
- **Enjoyability:** Users experience positive emotions when using the product
- **Beauty:** Users find the product aesthetically pleasing

Interaction design models

- Before coding the UX, models are incredibly helpful to plan out how it looks and behaves
- **Static interaction design models** show the audio and visual parts of the product that don't change during execution
- **Dynamic interaction design models** show behavior during execution
- Both are useful

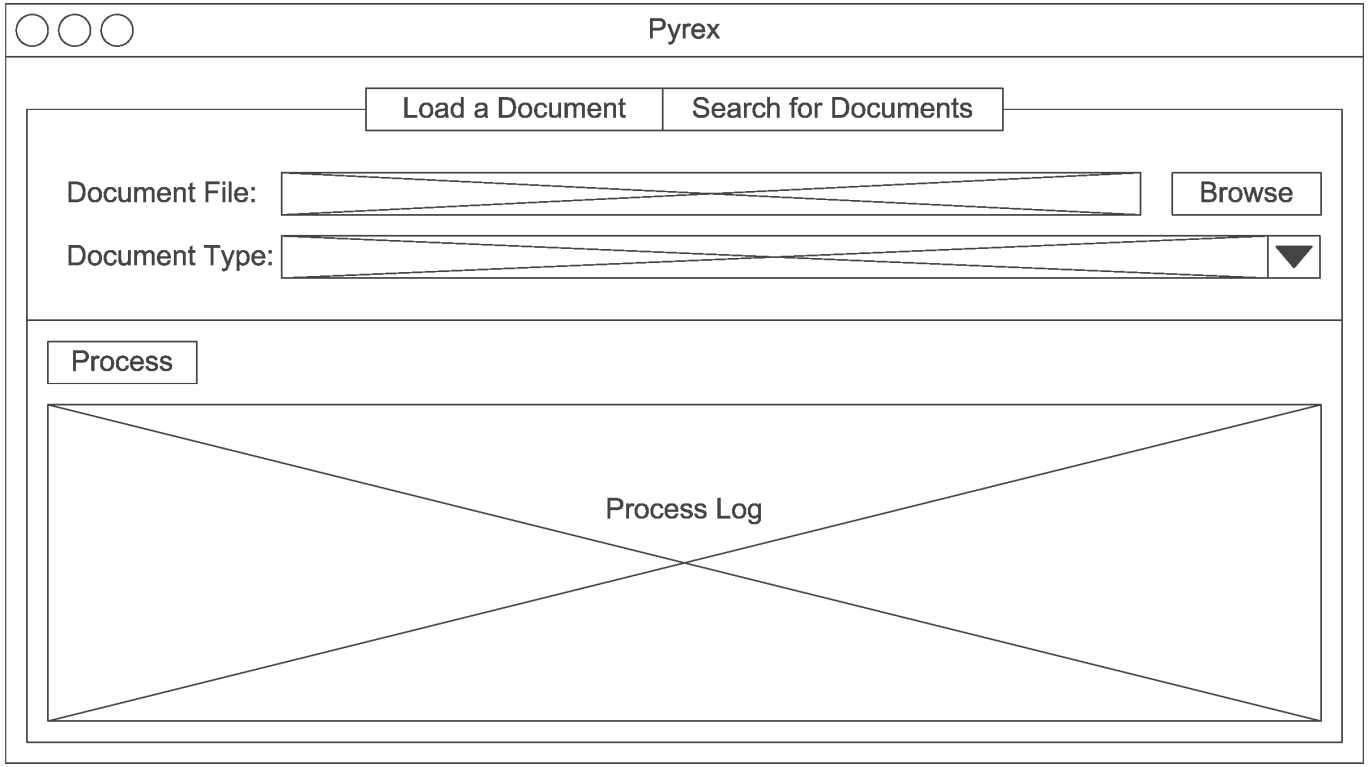
Use case diagrams

- A **use case** is an interaction between a product and its environment
- An **actor** is an agent that interacts with a product
- **Use case diagrams** (which we've seen before) are static interaction design models that represent the actors that interact with use cases



Layout diagrams

- **Screen layout diagrams** and **page layout diagrams** are drawings of a product's visual display
- A **wireframe** is a low-fidelity version that gives a rough layout without a lot of detail
- It's good to start with a wireframe and refine it with more detail later



Use case descriptions

- A use case diagram shows which actors interact with use cases
- However, it doesn't explain what they *do*
- A **use case description** is formatted text that explains the actions that an actor makes
- The use case description is a dynamic interaction design model
- Example template:

Use Case Name	To identify the use case
Actors	The agents participating in the use case
Stakeholders and Needs	What this use case does to meet stakeholder needs
Preconditions	What must be true before this use case begins
Post conditions	What will be true when this use case ends
Trigger	The event that causes this use case to begin
Basic Flow	The steps in a typical successful instance of this use case
Extensions	The steps in alternative instances of this use case due to variations in normal flow or errors

SAC principles

- **Design principles** favor certain characteristics to make a design better
- SAC principles are three general interaction design principles
- **Simplicity**
  - Simple designs are better
  - Lots of options are confusing for the user
  - It's better to make commonly used options easy and require a little more work for unusual options
- **Accessibility**
  - Designs that can be used be more people are better
  - Considerations: color blindness, things too small to see or interact with
- **Consistency**
  - Designs that present data in similar ways are better
  - Example: use consistent navigation controls

CAP principles

- CAP principles are focused on appearance
- **Contrast**
  - Designs that make different things obviously different are better
  - Example: italics and bold
  - Example: font size to distinguish headings from text
- **Alignment**
  - Designs that line up on a grid are better
  - Indentation is useful
- **Proximity**
  - Designs that group related things together are better

FeVER principles

- FeVER principles are about behavior
- **Feedback**
  - Designs that acknowledge user actions are better
  - Otherwise, how do you know if what you're doing has an effect?
- **Visibility**
  - Designs that display their state and available operations are better
  - Are we in Arm Bomb or Disarm Bomb mode?
- **Error Prevention and Recovery**
  - Designs that prevent user errors and allow error recovery are better
  - Prevention example: disable buttons that shouldn't be pressed
  - Recovery example: allow undo or ask "Are you sure?" before doing something dangerous

Software Engineering Design

Simplicity

- As with interface design, simpler designs are better
- What is simplicity?
  - Fewer lines of code
  - Fewer control structures
  - Fewer connections between different parts
  - Fewer computations with different kinds of objects
- A good rule of thumb is which design is easiest to understand
- Simplicity is a good goal, but some important algorithms in computer science are necessarily complex

Everything should be made as simple as possible, but not simpler.

-Attributed to Albert Einstein, who probably did not say it quite like that

Small modules

- Designs with small modules are better
- Smaller modules are easier to read, to write, to understand, and to test
- Miscellaneous guidelines:
  - Classes should have no more than a dozen operations (methods)
  - Classes should be no more than 500 lines long
  - Operations should be no more than 50 lines long
  - I have heard that you should be able to cover a method with your hand
- Of course, it is often impossible to follow these guidelines

Information hiding

- Each module should shield the internal details of its operation from other modules
- Declare variables with the smallest scope possible
- Use **`private`** (and **`protected`**) keywords in OOP languages to hide data (and even methods) from outside classes
- Advantages of information hiding:
  - Modules that hide their internals can change them without affecting other thins
  - Modules that hide information are easier to understand, test, and reuse because they stand on their own
  - Modules that hide information are more secure and less likely to be affected by outside errors
- This is why we use mutators and accessors instead of making members public

Minimize module coupling

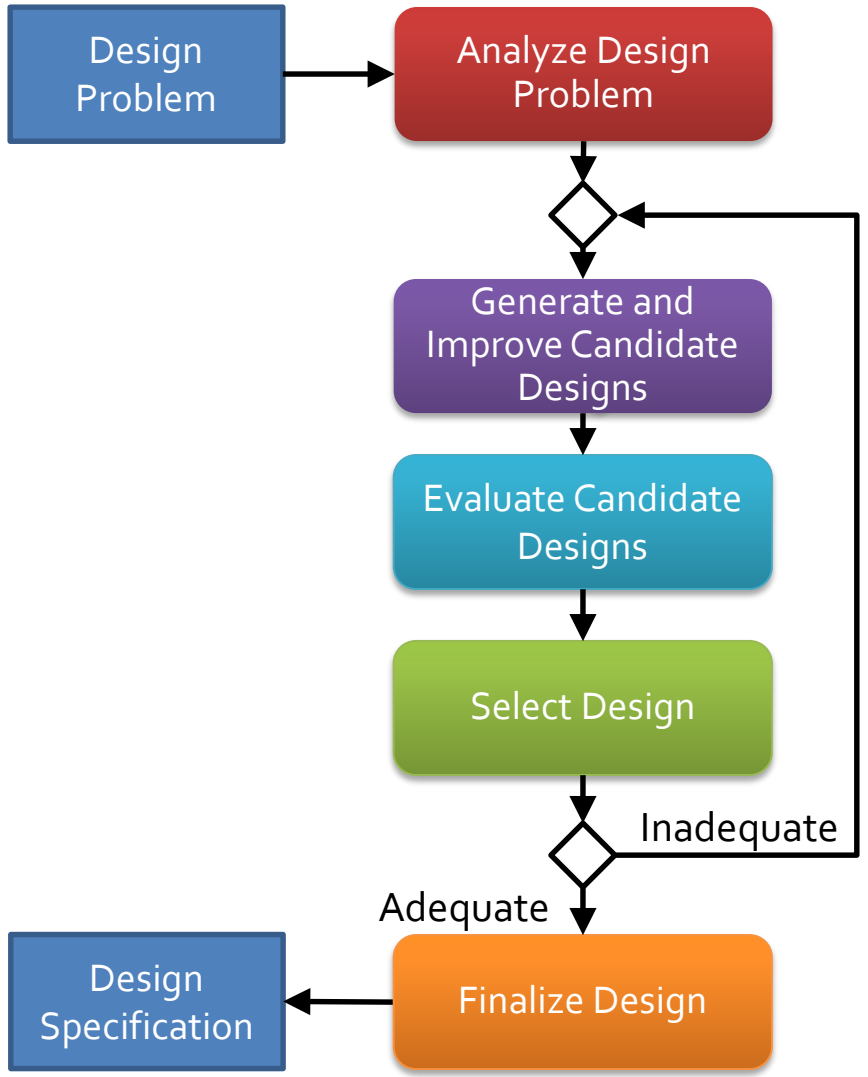
- **Module coupling** is the amount of connectivity between two modules
- Modules can be coupled in the following ways:
  - One class is an ancestor of another class
  - One class has a member whose type is another class
  - One class has an operation (method) parameter whose type is another class
  - One operation calls an operation on another class
- If there two modules have many of these couplings, we say that they are **strongly coupled** or **tightly coupled**
- When modules are strongly coupled, it's hard to use them independently and hard to change one without causing problems in the other
- Try to write classes to be as general as possible instead of tied to a specific problem or set of classes
- Using interfaces helps

Maximize module cohesion

- **Module cohesion** is how much the stuff in the module is related to the other stuff in the module
- We want everything in a class to be closely related
- It's best if a class keeps the smallest amount of information possible about other classes
- More module cohesion usually leads to looser module coupling
- Sometimes a module being hard to name suggests that its data or operations are not cohesive

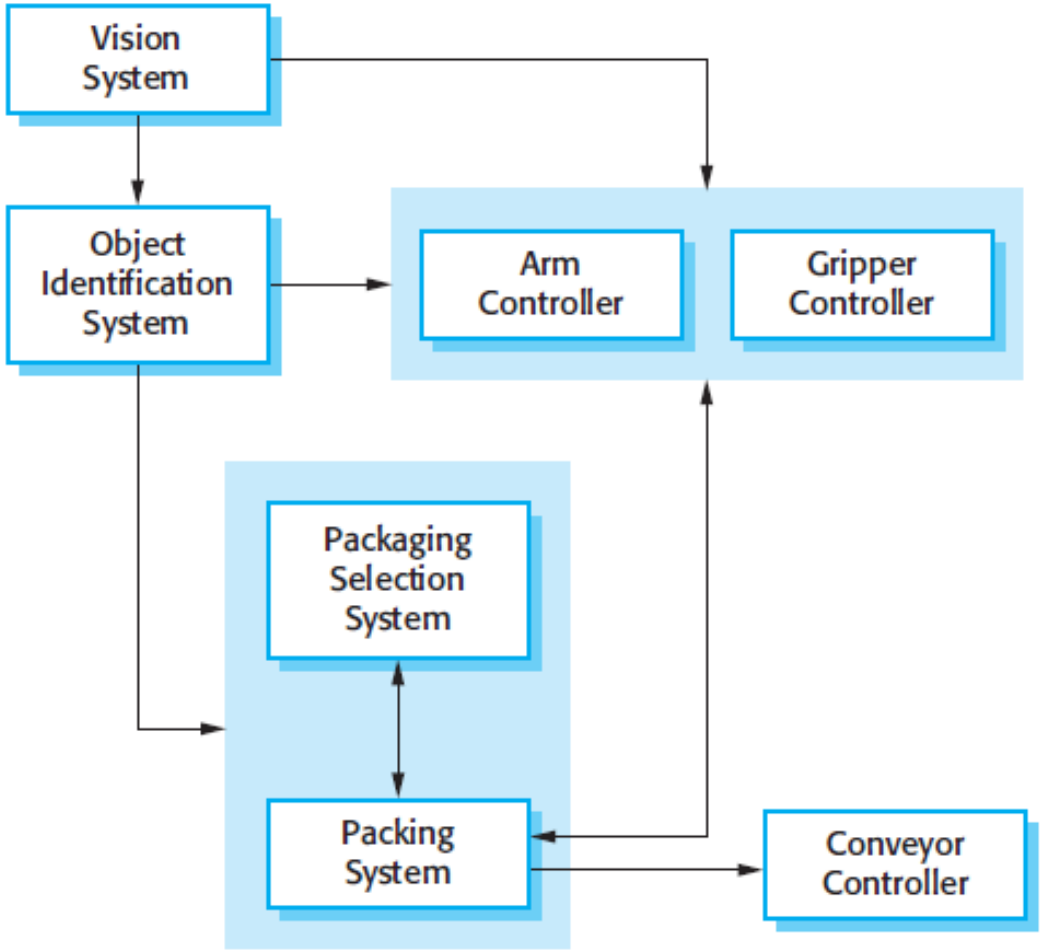
Design process

- The design process is a microcosm of the larger software development process
- The steps are analyzing the problem, proposing solutions (and looking up existing solutions to similar problems), and evaluating the solutions (perhaps combining different solutions) until a design is selected



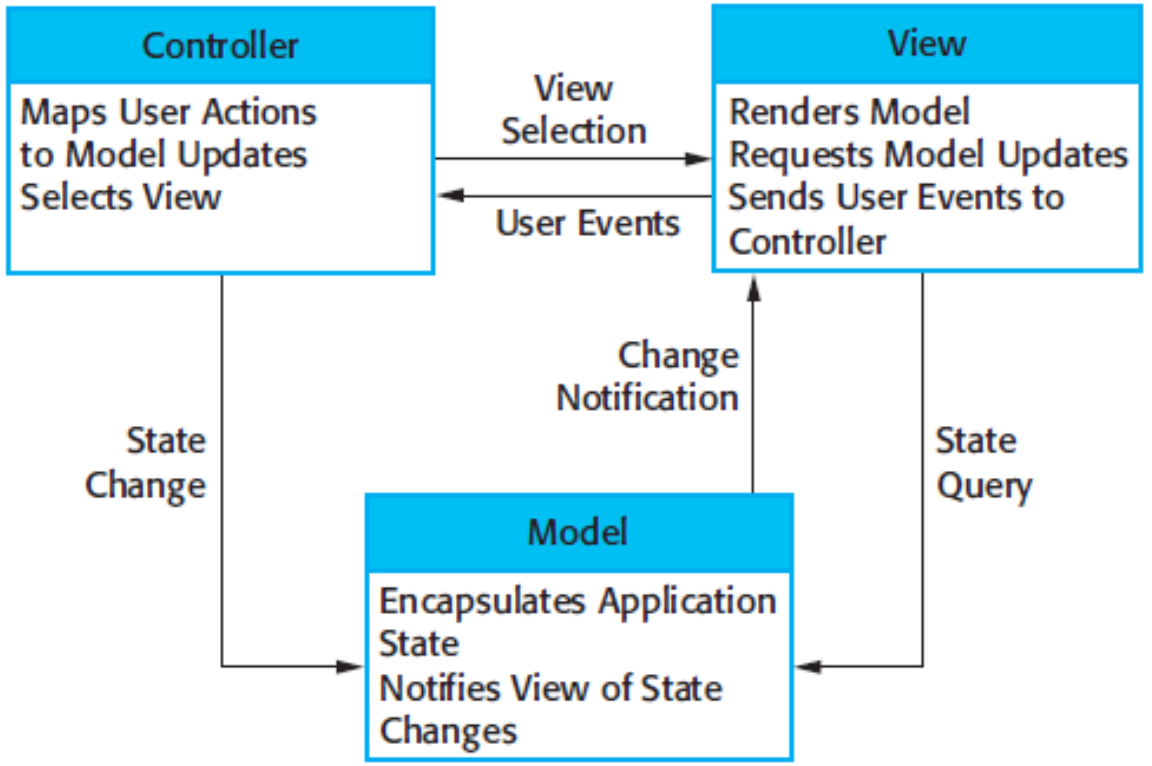
Architectural design

- **Architectural design** is specifying a program's major components
- Architectural design is often modeled with a **box-and-line diagram** (also called a block diagram)
  - Components are boxes
  - Relationships or interactions between them are lines
  - Unlike UML diagrams, box-and-line diagrams have no standards
  - Draw them in a way that communicates your design



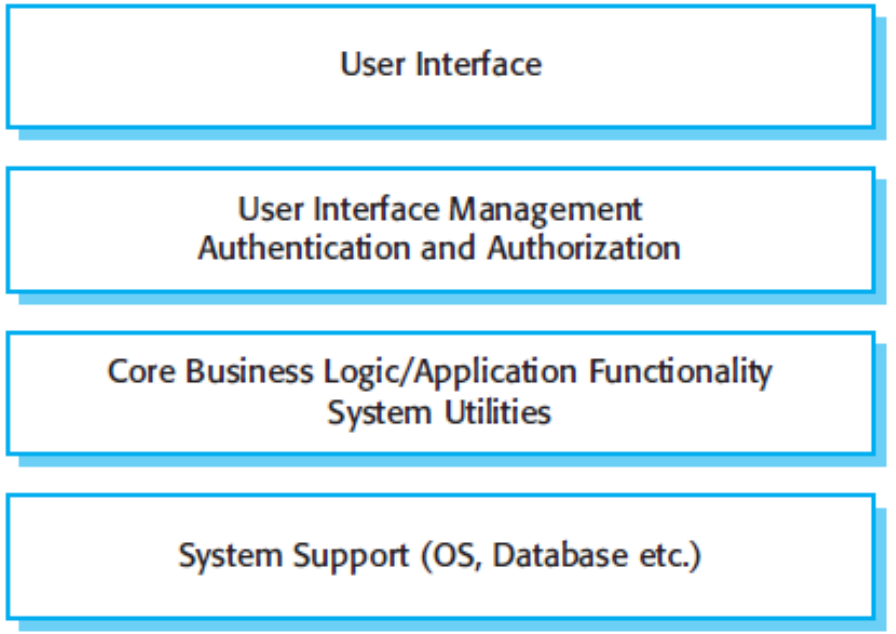
Model-View-Controller

- The **Model-View-Controller** (MVC) style 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



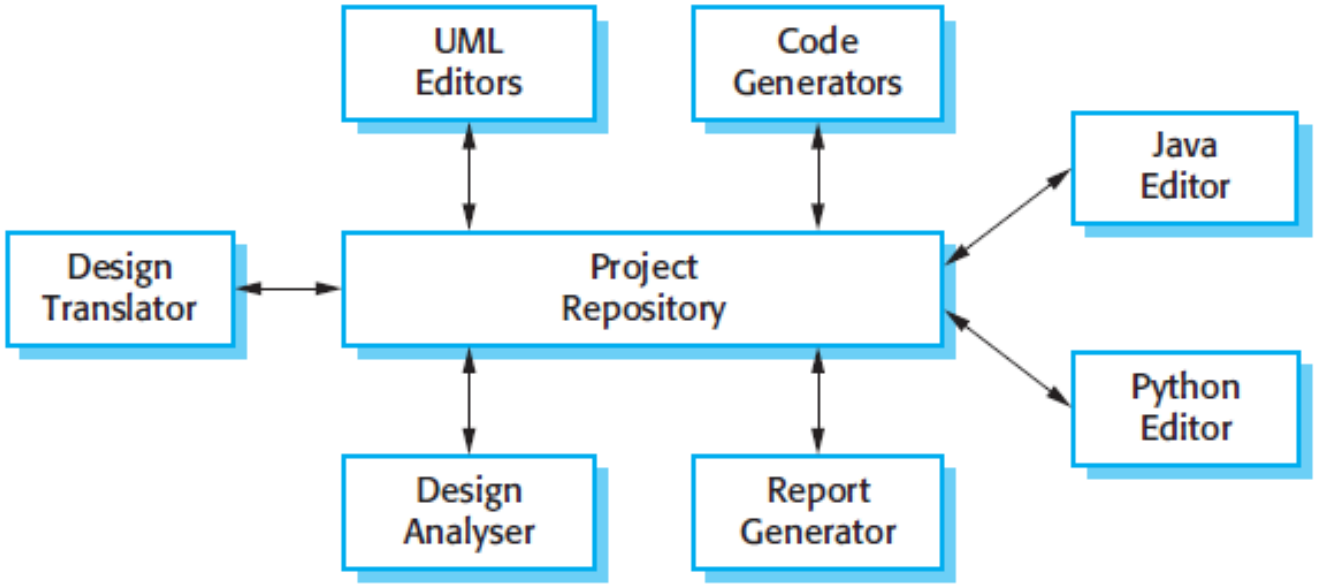
Layered style

- Organize the system into layers
- Each layer provides services to layers above it, with the lowest layer being the most fundamental operations
- Layered styles 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



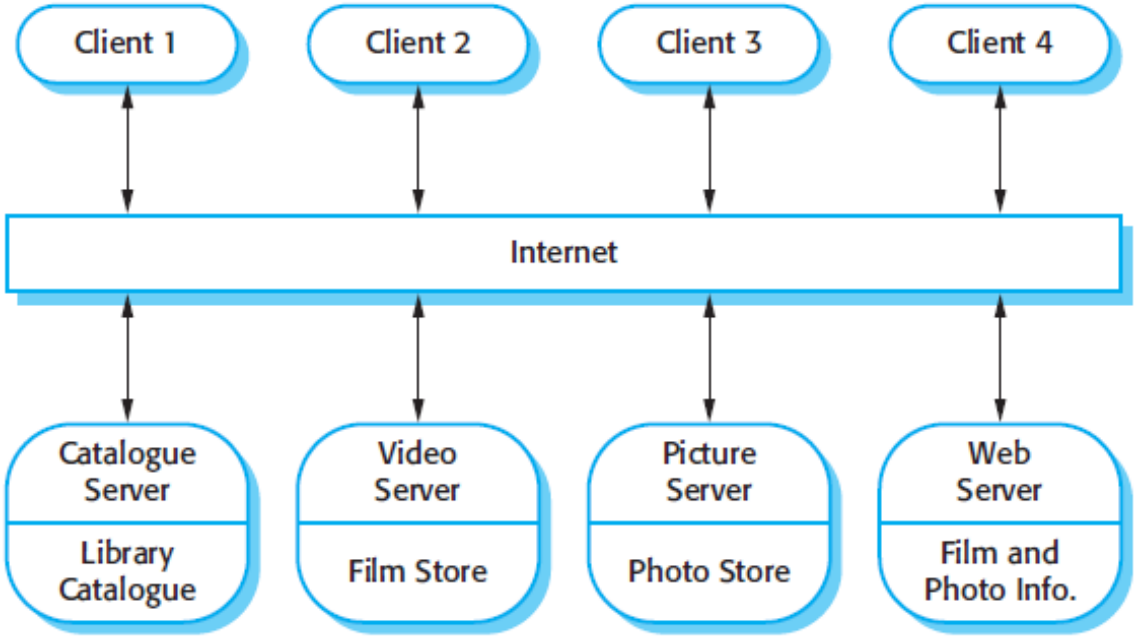
Repository style

- If many components share a lot of data, a repository style 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



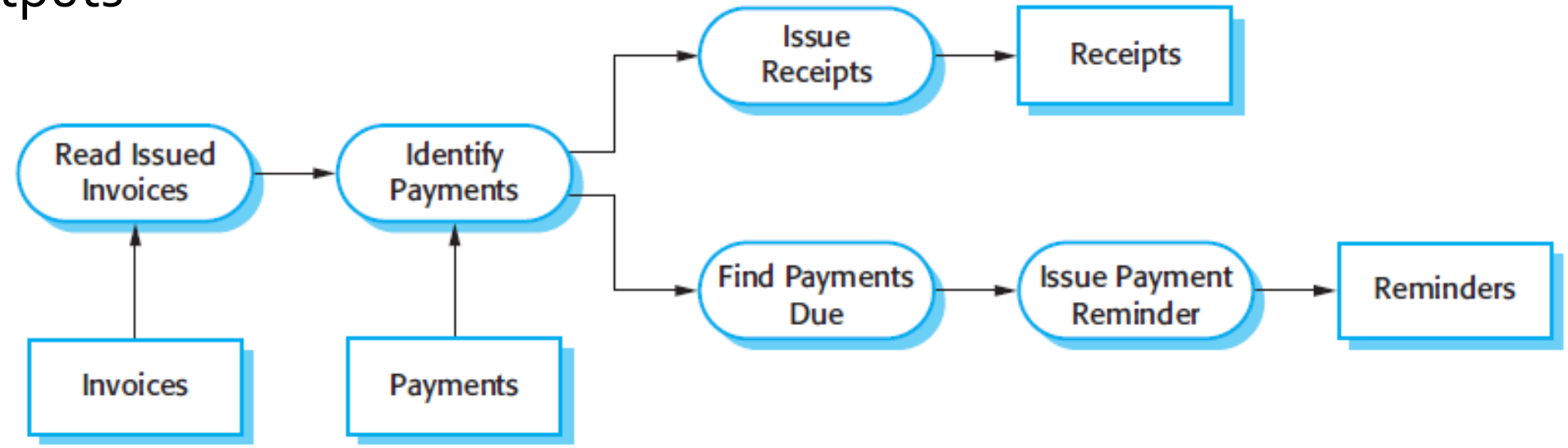
Client-server architecture

- Client-Server styles 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 style

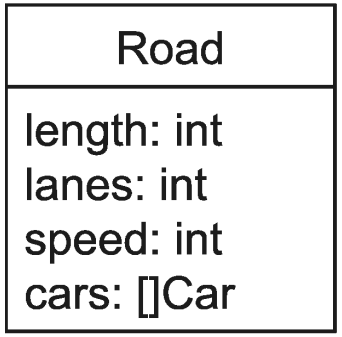
- In the pipe and filter style, data is passed from one component to the next
- Each component transforms input into output
- Good: easy to understand, matches business applications, and allows for component reuse
- Bad: each component has to agree on formatting with its inputs and outputs

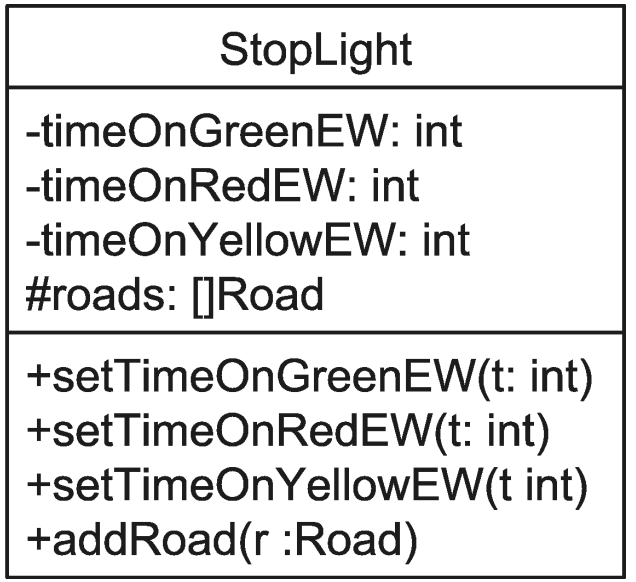


Detailed Design

More depth on class diagrams

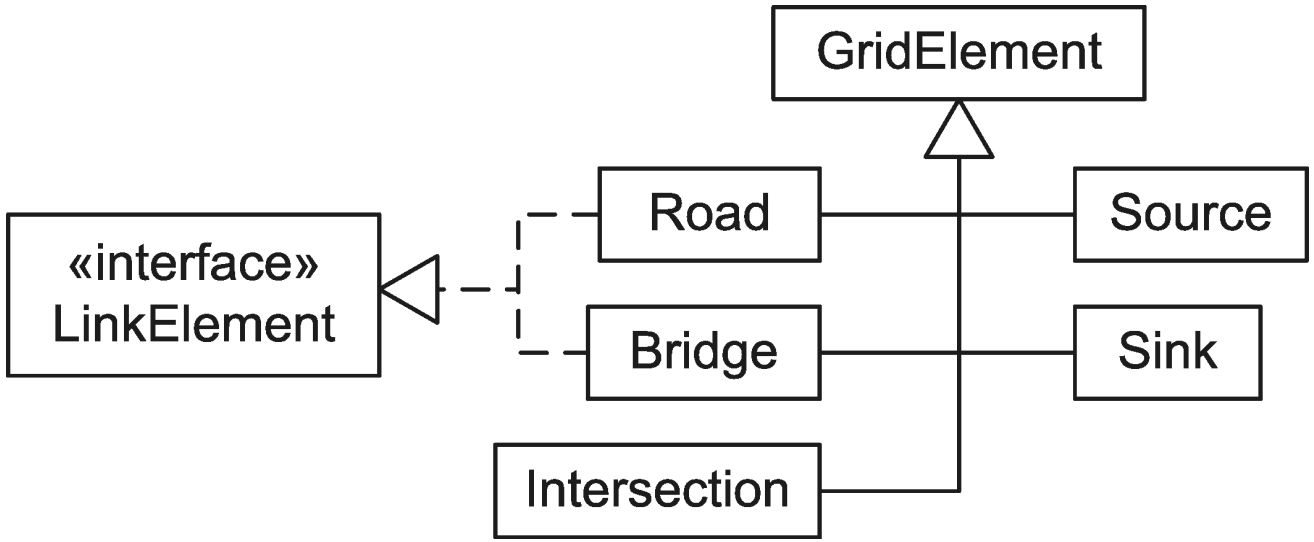
- **Class diagrams** are made up of **class symbols** (rectangles)
- These class symbols contain one or more **compartments**
- The top compartment has the class name
- A second, optional compartment often contains attributes (called member variables in Java classes)
  - Often followed by a colon with the type
- A third, optional compartment often contains operations (called methods in Java classes)
  - Sometimes followed by parameter and return types
- Visibility modifiers can be marked:
  - **+** for public
  - **#** for protected
  - **~** for package
  - **-** for private
- Only important attributes and operations need to be specified
  - Classes might contain others that aren't shown





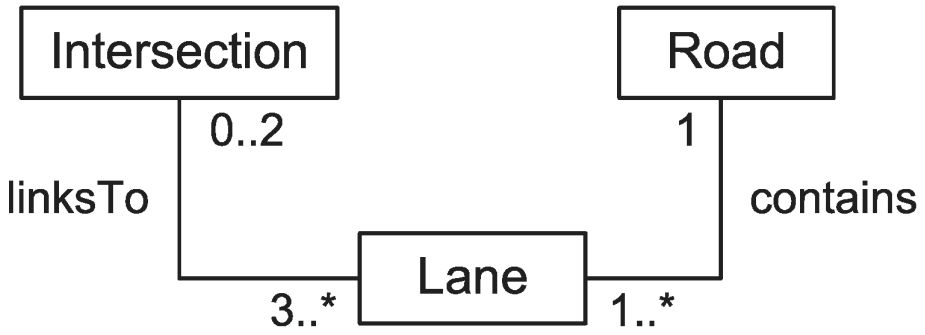
Inheritance and interfaces in class diagrams

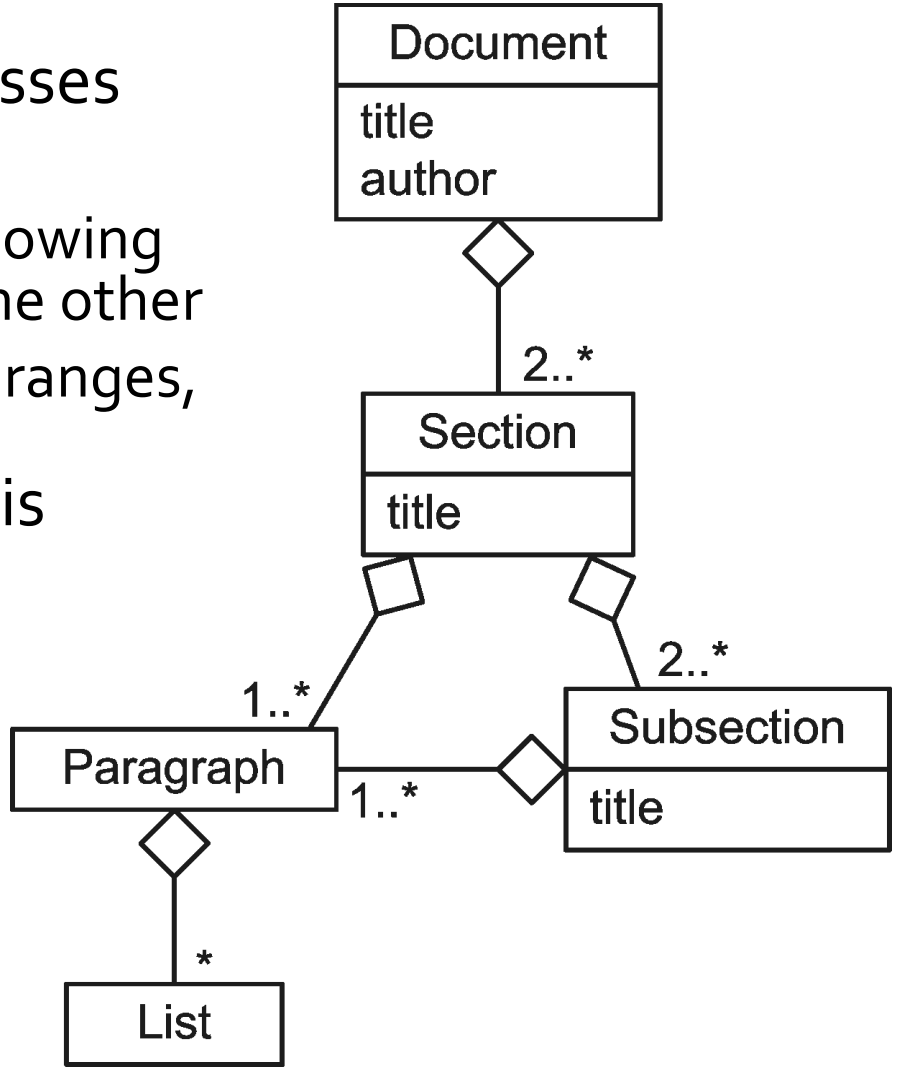
- Inheritance is shown with the **generalization** connector
  - A solid line from the child class to a solid triangle connected to the parent class
  - Confusingly, this means that children classes point at their parent classes
- Interfaces look like classes but are marked with **«interface»** above the class name
  - This kind of marking is called a **stereotype**
  - Stereotypes show extra information that wasn't part of the original UML class diagram specification
- Classes that implement interfaces have dashed lines leading to a solid triangle connected to the interface



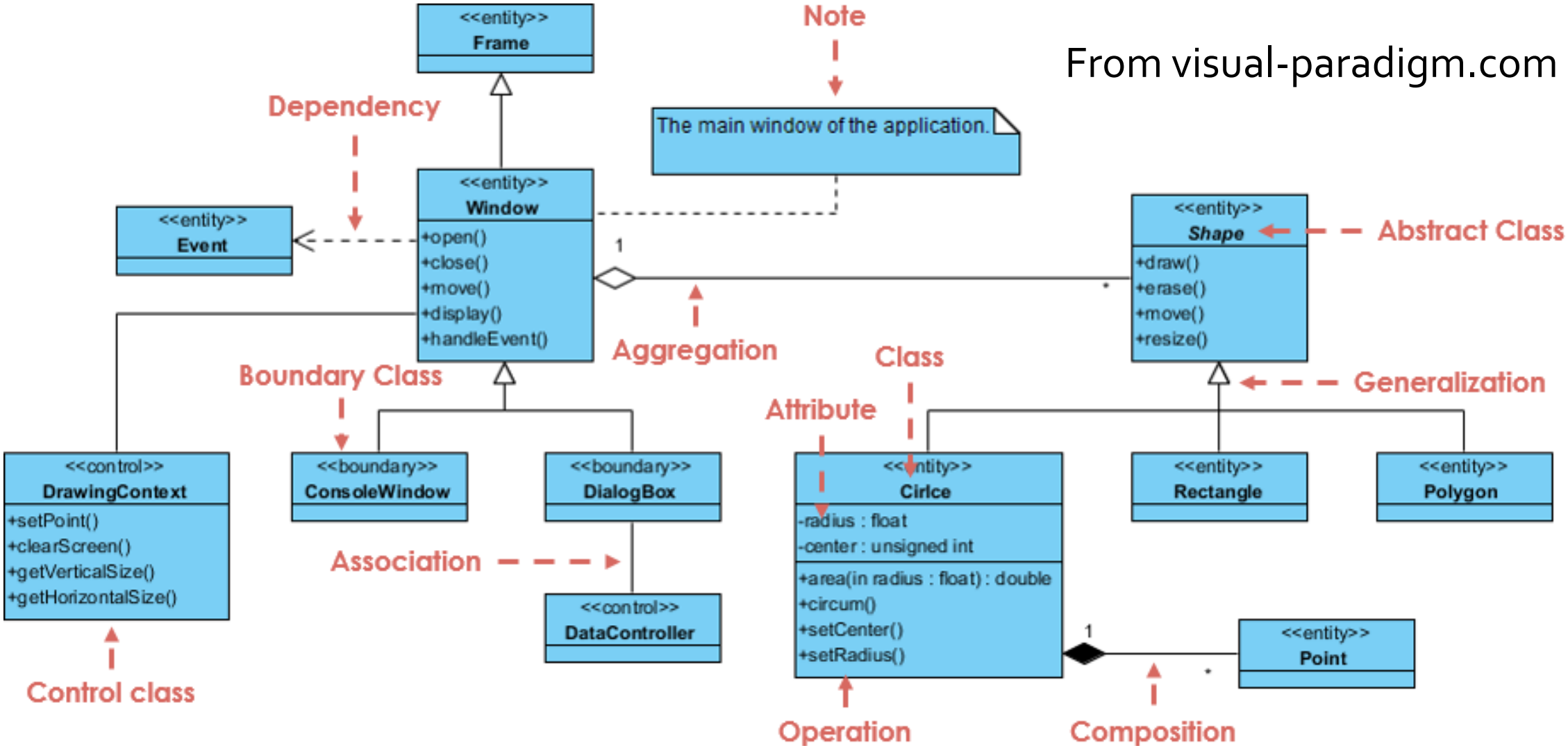
Other associations

- **Associations** are show with lines between classes
  - Associations can be labeled to explain them
  - The lines can be marked with the **multiplicity**, showing how many of each class can be associated with the other
  - The multiplicity can be comma separated lists or ranges, and * means zero or more
- When a class is part of another class, the part is connected by a line and a diamond (the **aggregation** connection) to the whole





Complex example



Upcoming

Next time…

- Finish review on Wednesday
- **Presentations on Friday!**

Reminders

- Finish Project 4 for Friday
- Don't forget Assignment 4 is **also** due on Friday!
- Review for Final Exam on Wednesday