

Week 4 - Wednesday

**COMP 3100**

# Last time

- What did we talk about last time?
- Agile
- Started Scrum

Questions?

---

# Back to Scrum

---

# User stories

- We mentioned **user stories** in the discussion about requirements
- User stories are the most popular way of specifying features in Scrum
- User story format:
  - As a *<user role>* I want to *<goal>* so that *<benefit>*.
- Examples:
  - As a course scheduler I want to determine whether students can take other sections of a course so that I can see if I can cancel a section with students already enrolled in it.
  - As a shopper I want to see whether an item is still on sale so that I can buy it more cheaply.
  - As an internet user I want to secure my devices so that I can protect my private information.
  - As an electric utility customer I want to see my usage over several years so that I can analyze it to budget my electricity costs more exactly.

# User story abstraction

- User stories come at different levels of abstraction and size
  - Large, abstract stories that would take months of coding are sometimes called **epics**
  - Medium-sized stories that would take several sprints are sometimes called **features**
  - Small, detailed stories that can be done in a single sprint are **sprintable stories** (or simply stories)
- Even sprintable stories usually aren't detailed enough to implement without additional conversations with stakeholders

# 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

# Creating the sprint backlog

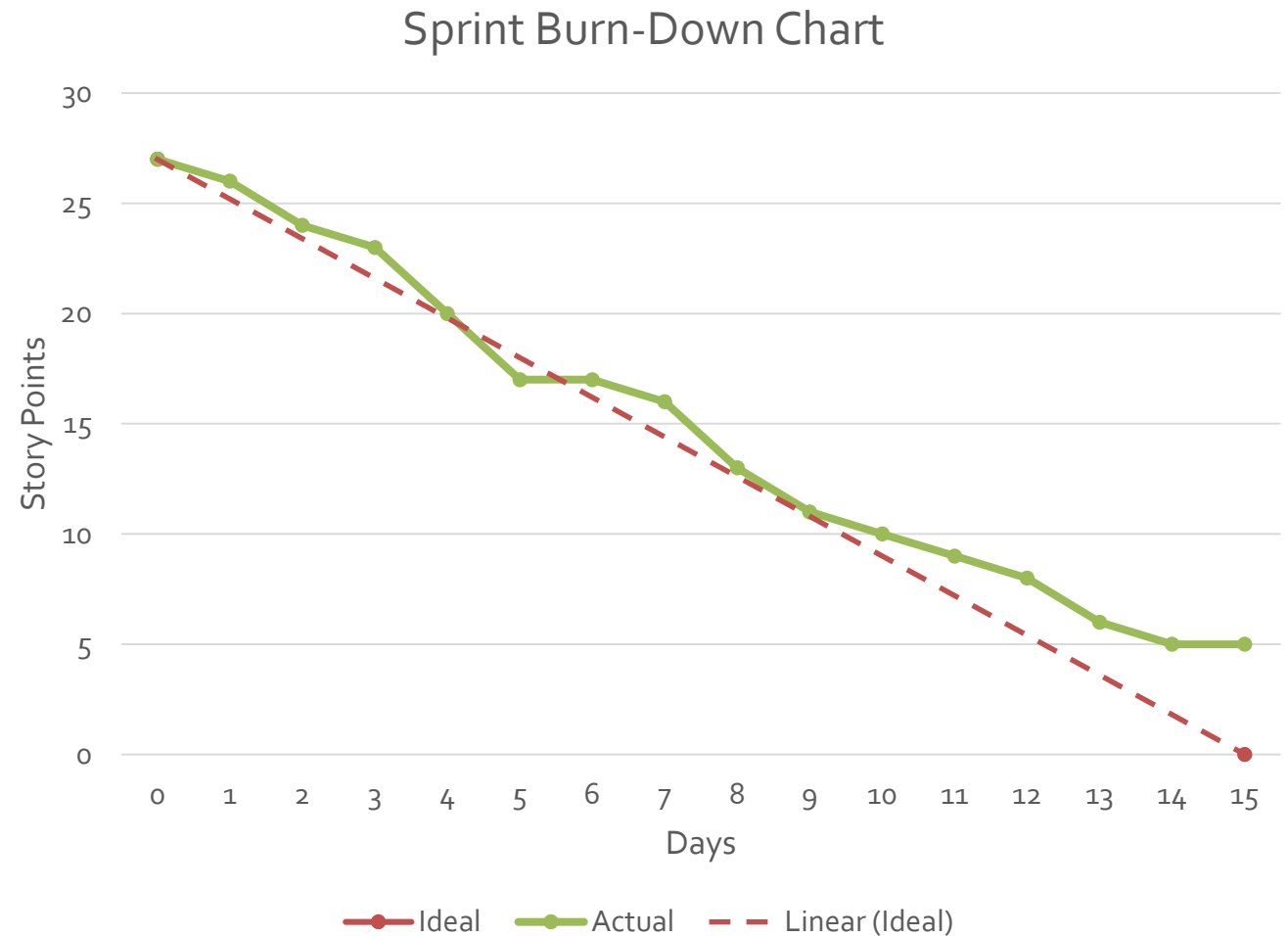
- In sprint planning, teams refine their estimates of high-priority PBIs before finalizing the sprint backlog
- For each PBI, they estimate the tasks involved in ideal hours
  - Story points can be used, but ideal hours are more precise
- The tasks can include:
  - Coding
  - Unit testing
  - Integration testing
  - Acceptance testing
  - Code inspection
  - Updating user documentation
- The final sprint backlog includes PBIs, their constituent tasks, and effort estimates for all tasks

# 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

# Burn charts

- **Burn charts** can be used to track progress on a project
  - **Burn-down charts** show how much work is left and are preferred for sprints
  - **Burn-up charts** show how much work has been done
- A burn-down chart shows the total effort in story points on the y axis and time on the x axis
- A straight line shows ideal progress while real progress is plotted against it



# Kanban boards

- To follow the progress of individual user stories, it's common to use **task boards**
- These boards are often called **kanban boards** because of the agile process Kanban that uses them
- Each row shows the progress of tasks related to the story on the left
- People use physical boards with sticky notes or electronic tools like Trello
  - Which you will be required to use in Project 3





# Definition of done

- What does *done* mean?
- Team have their own versions of done, often with the following items:
  - Design is complete and reviewed
  - Code is formatted and commented
  - Code has passed inspection
  - Code has passed PBI acceptance criteria (tests)
  - Code has passed all unit tests and regression tests
  - User documentation has been updated
  - Code has been integrated and passed all integration and systems tests
- When a PBI is truly done, it's removed from the product backlog

# 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

# Other Scrum practices

- **Daily scrum:** Short daily meeting, often called a **stand-up** (having no chairs encourages brevity)
  - What did I do since the last meeting
  - What will I do today
  - What is impeding my progress
- **Story time:** Groom the product backlog
- **Cross-functional teams:** Get non-specialists to help with specialized tasks, to get the job done and expand skills
- **Sustainable pace:** Don't overwork
- **Planning poker:** Have team members contribute their time/work estimate for a PBI
- **Bidding:** Team members bid on tasks with ideal hours
- **Pair programming:** Two people sit together to code, with one typing (the driver) and the other checking (the navigator), switching off frequently

# Review

---

# Exam format

- The exam will have:
  - Short answer questions
  - At least one diagram
  - Two essay questions
- Software Engineering is more ... philosophical than some other CS courses
  - And it's a Writing Intensive course

# 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



# Questions when building software products

- What exactly should it do? What if people disagree?
- How does this product fit into the rest of the stuff the company does?
- How will users interact with the product?
- What parts should the product have?
- What languages should it be written in?
- What standards should we use to write it?
- How do we know if the program does what it's supposed to?
- How much time and money will it take to make it?
- What kind of documentation will it need?
- How will it change in the future?
- How far along are we in the process of making it?

# Management

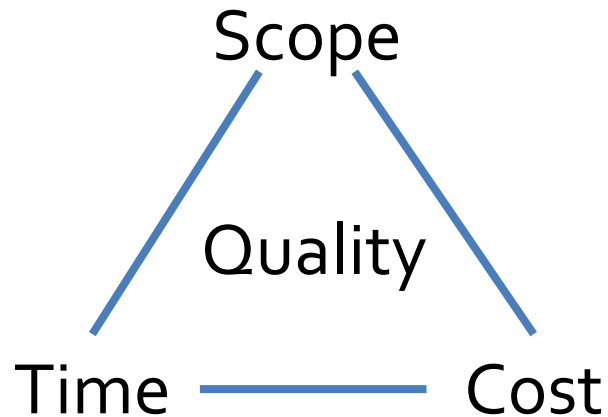
---

# 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?

# Project management iron triangle

- There's a graphical depiction of project management used imply relationships between time, scope, cost, and quality



- This triangle is intended to indicate that you can't change scope, time, or cost without affecting the other two (at least if you want to maintain quality)
- Increasing scope means increasing time or cost (or both)
- It's obvious, but manager are sometimes tempted to push workers to work faster, for example, pretending there are no consequences

# 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

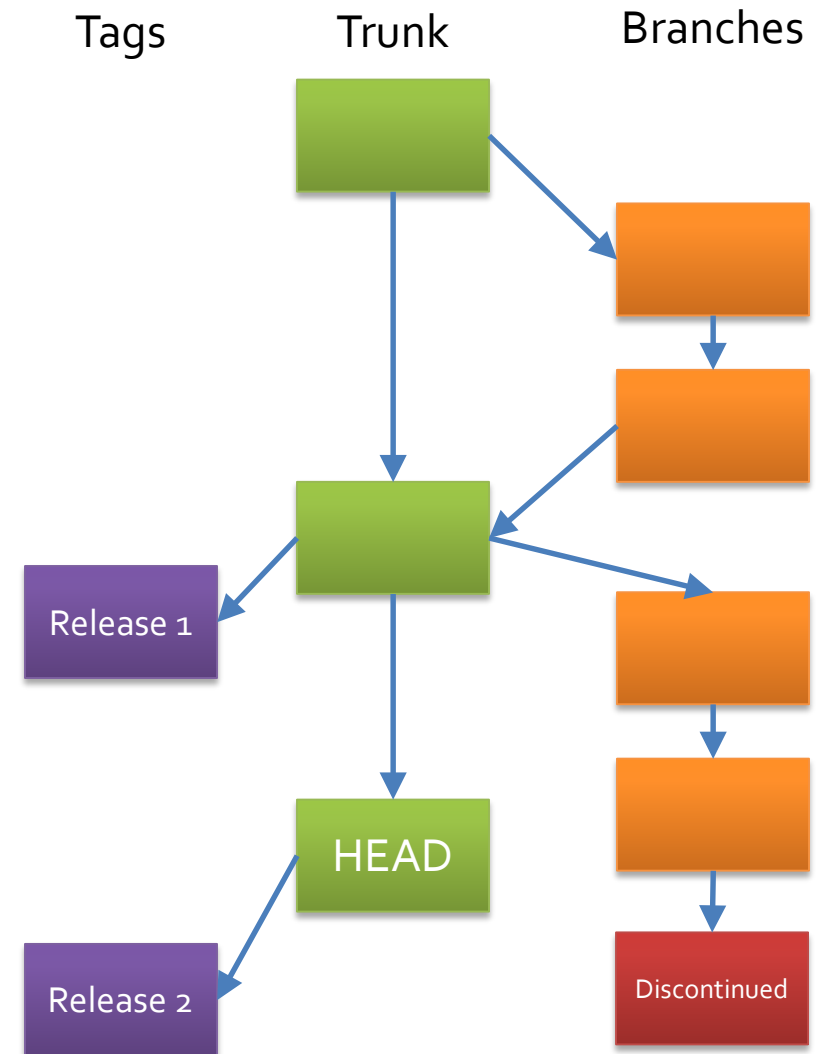
- A **repository** is where all the development data is stored
  - Usually called **repos** by professionals
- Repositories include the current source code as well as a history of all the changes ever made
- For source code, most version control systems use **delta compression**, meaning that only the *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

# Visualization of development

- Version control systems provide ways to organize the development process
- One such feature is a visualization of the development process
- The main sequence of development is called the **trunk**
- Code bases that diverge from main development (to work on a new feature) are **branches**
- **Tags** are snapshots of the code base in a particular state, often a release



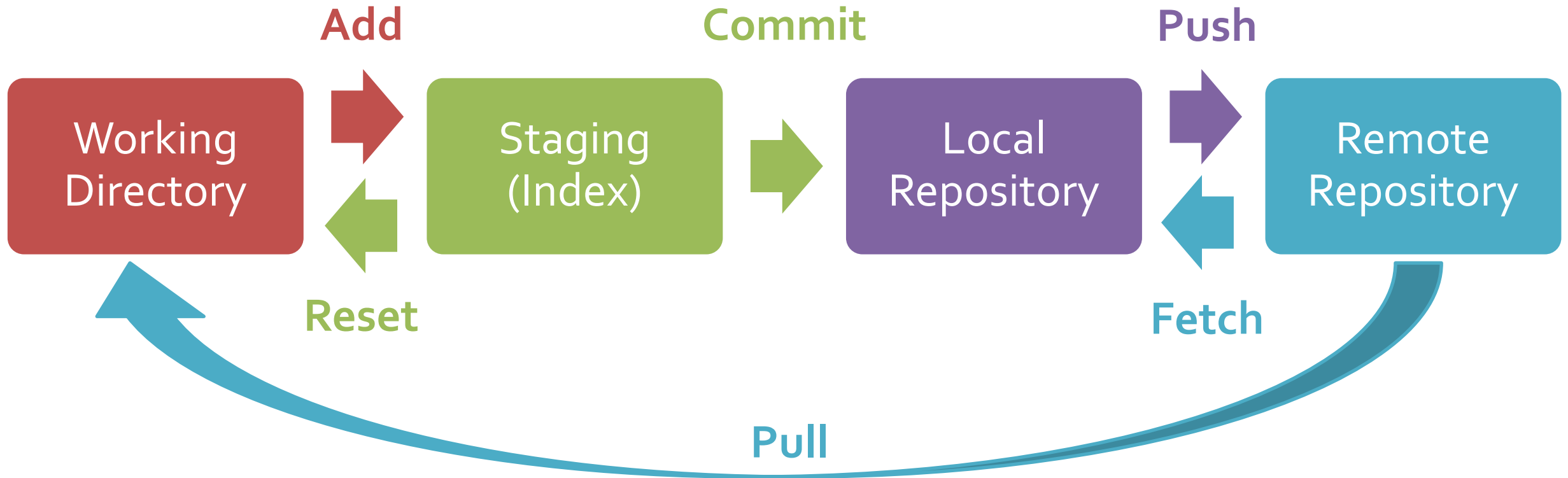
# Popular version control systems

- **Git**
  - Git is one of the most popular systems with a distributed model
- **SVN**
  - SVN is one of the most popular systems with a client-server model
- **Microsoft Team Foundation Server**
  - Microsoft always has to have its own thing
- **Mercurial**
  - Mercurial competes with Git as a distributed VCS
- **Perforce**
  - A big suite of tools that can do its own things or integrate with Git
- **CVS**
  - An old client-server tool that was popular until SVN overtook it

# 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

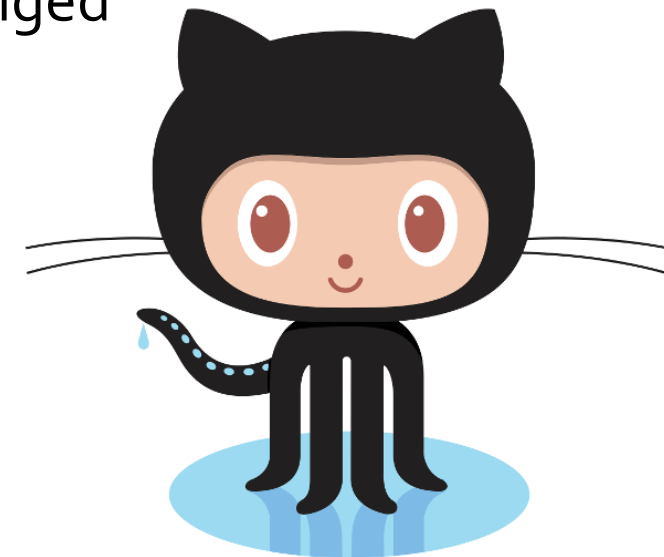
# Git process



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)
- GitHub 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

# Levels of abstraction

- **Business requirements specifications** are client objectives that must be met
  - They are abstract descriptions of the product
  - They might include deadlines or sales targets
  - They're usually non-functional requirements
- **User-level needs and requirements** are one step more concrete
  - They describe tasks or goals that the product would allow a user to perform
  - These tend to describe what the product does but not how
  - Could be functional or non-functional

# More levels of abstraction

- **Operational-level needs and requirements** describe individual inputs and outputs
  - These are more specific than user-level needs and requirements
  - Example: The product must allow users to enter polynomial equations, trigonometric equations, logarithmic equations, and exponential equations, all of one variable.
- **Physical-level needs and requirements** are about the appearance and formatting of the user interface
  - These will describe what the product actually looks like (which might be several different descriptions if the same product works on a desktop and a phone)
  - Example: The product must provide a text-box for equation input. The text box must display 50 characters and scroll vertically up to 800 lines.

# Requirements in traditional processes

- Most of the terminology we've been using comes out of traditional software development processes like waterfall
- In that paradigm, requirements must be gathered first, followed by design, followed by implementation, testing, and maintenance
- The requirements must be **frozen** so that the next steps can take place
- Then, no one wants to change the requirements because you'll have to redo everything that comes after, which is expensive



# 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 is 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

# Requirements modeling

- When software engineers say *modeling*, they usually mean drawing diagrams
- **Requirements modeling** is making representations (diagrams) that help you understand your requirements
- Both traditional and agile processes use models
- The **Unified Modeling Language (UML)** is the most common set of standards for representing such models
- Some developers use model extensively, and others use them rarely

# Kinds of requirements modeling

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

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

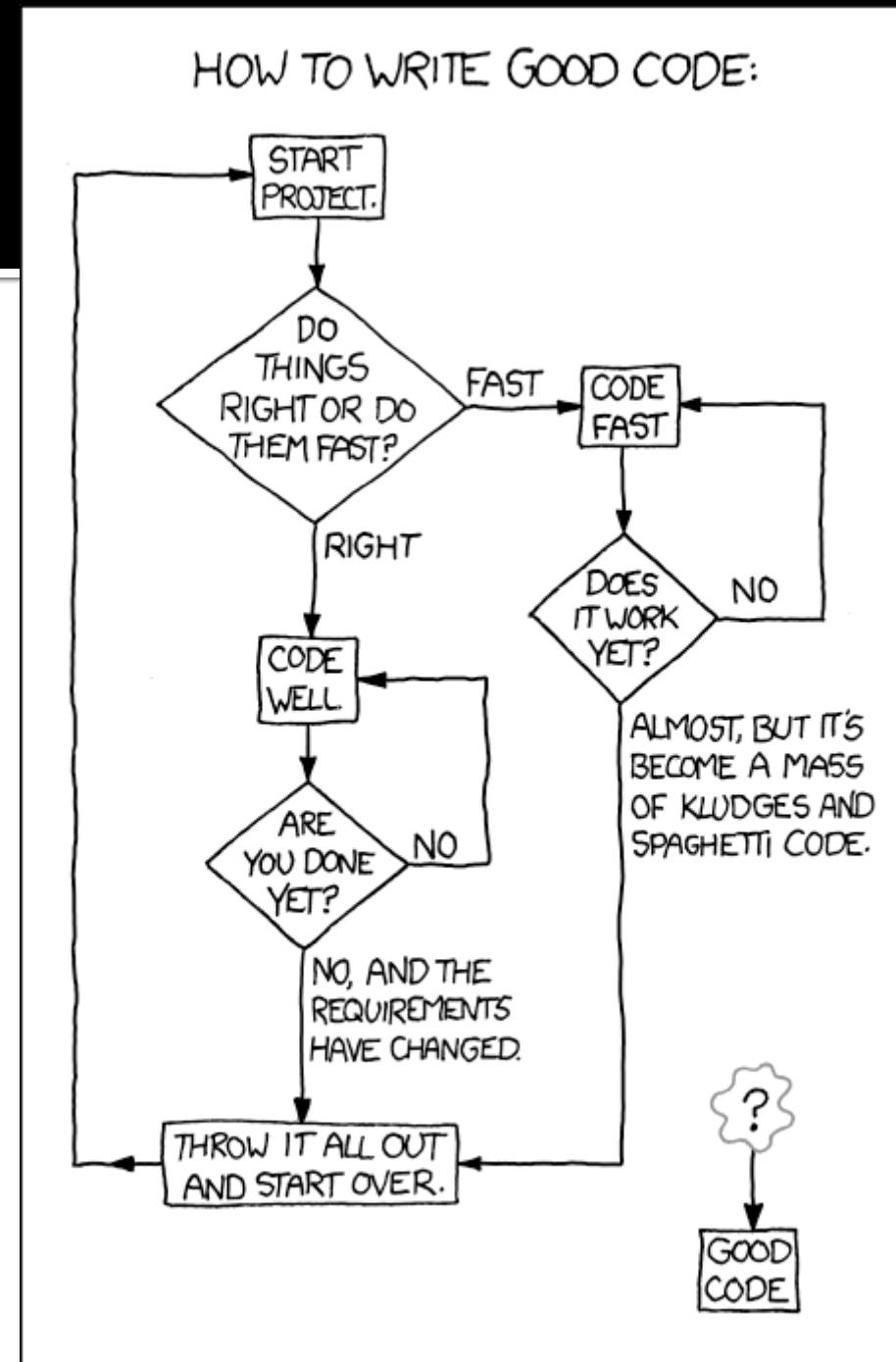
- **Models leave out details**
- Models are useful to help understand a complex system
  - During requirements engineering, models clarify what an existing system does
  - Or models could be used to plan out a new system
- Models can represent different perspectives of a system:
  - **External:** the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

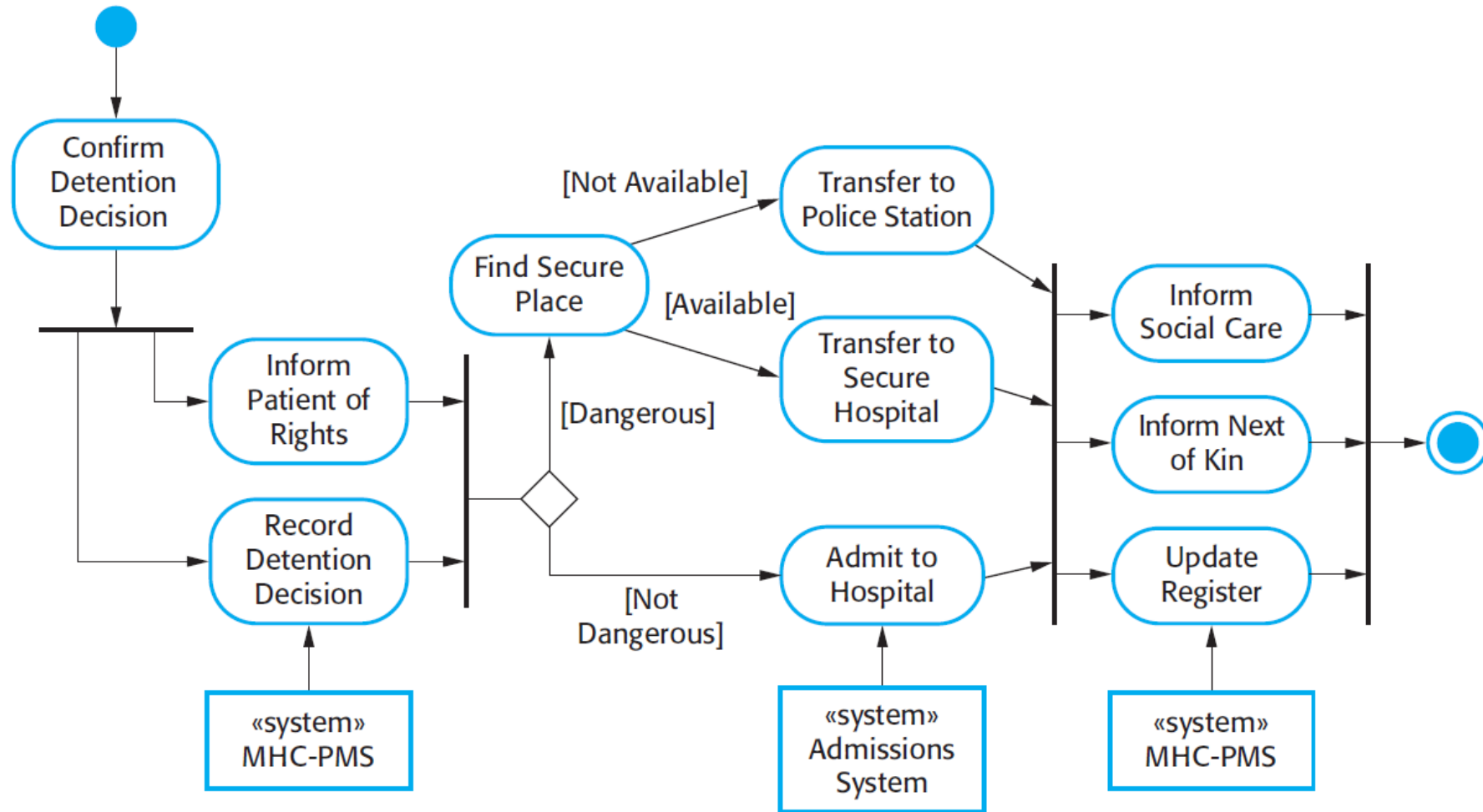
- The **Unified Modeling Language** (UML) is an international standard for graphical models of software systems
- A few useful kinds of diagrams:
  - Activity diagrams
  - Use case diagrams
  - Sequence diagrams
  - State diagrams
- Class diagrams are important enough that we'll talk about them in greater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

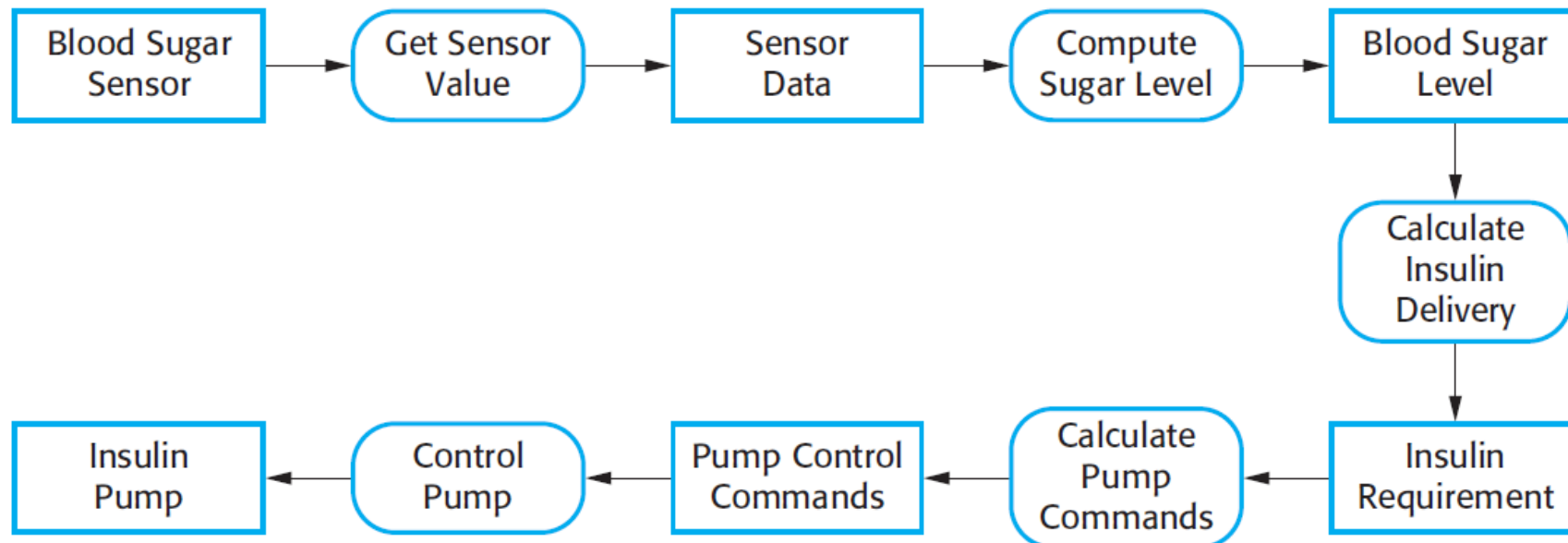


# More detailed activity model



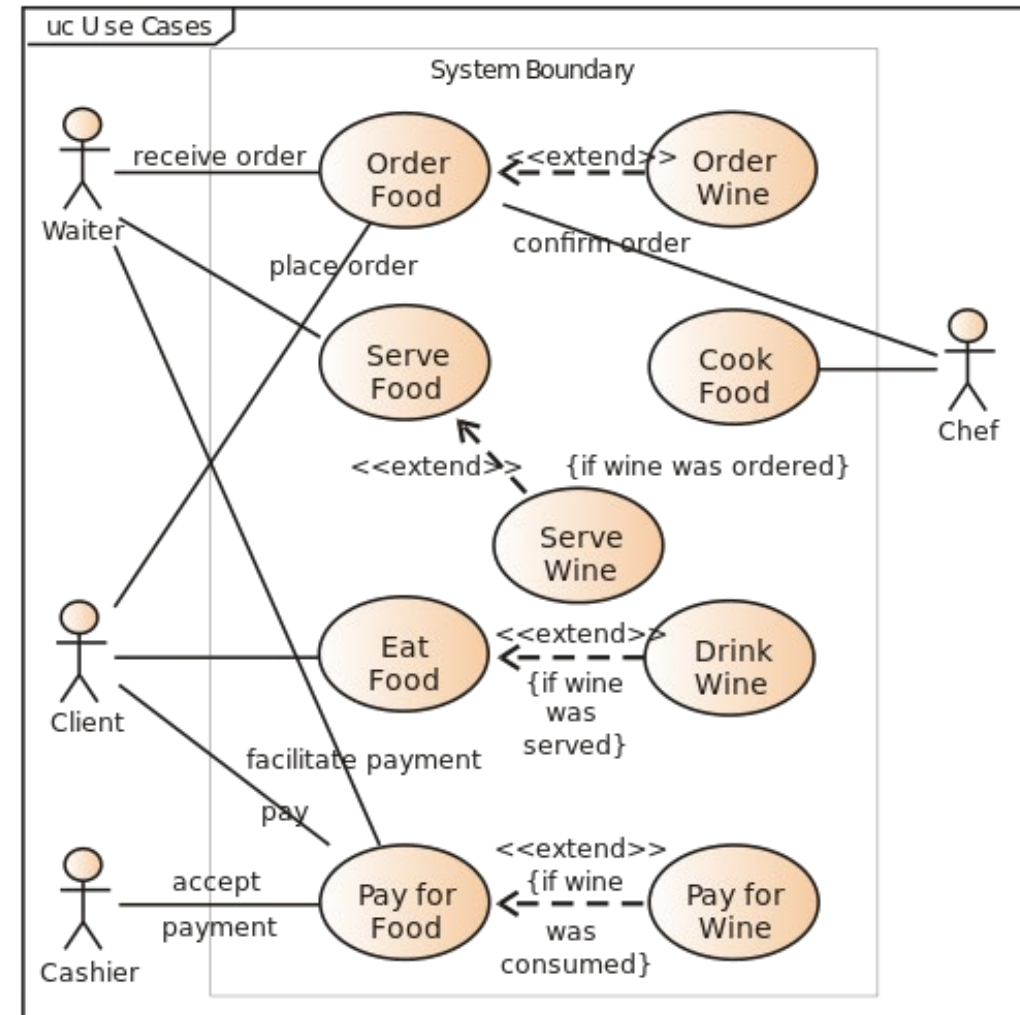
# Data-driven modeling

- 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



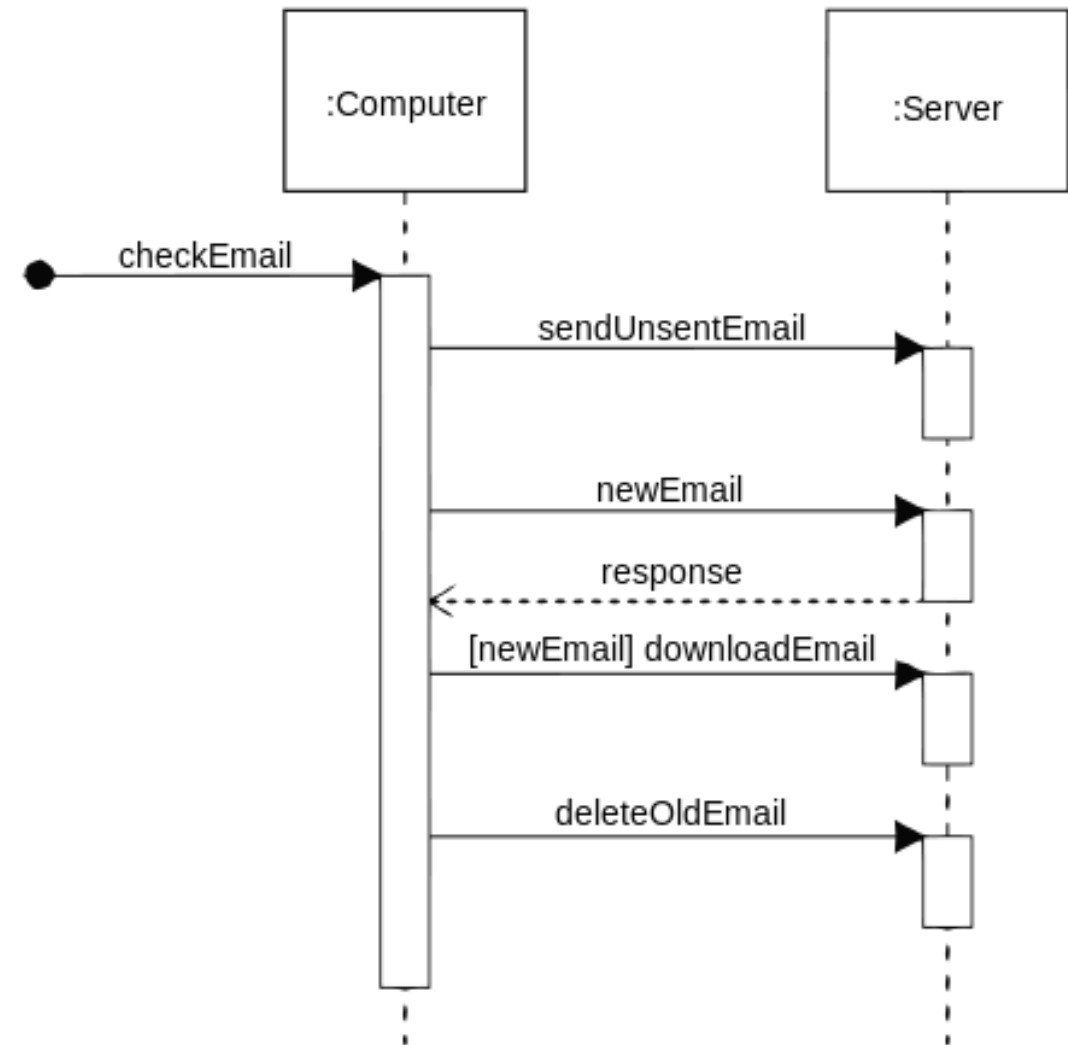
# Use case diagrams

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



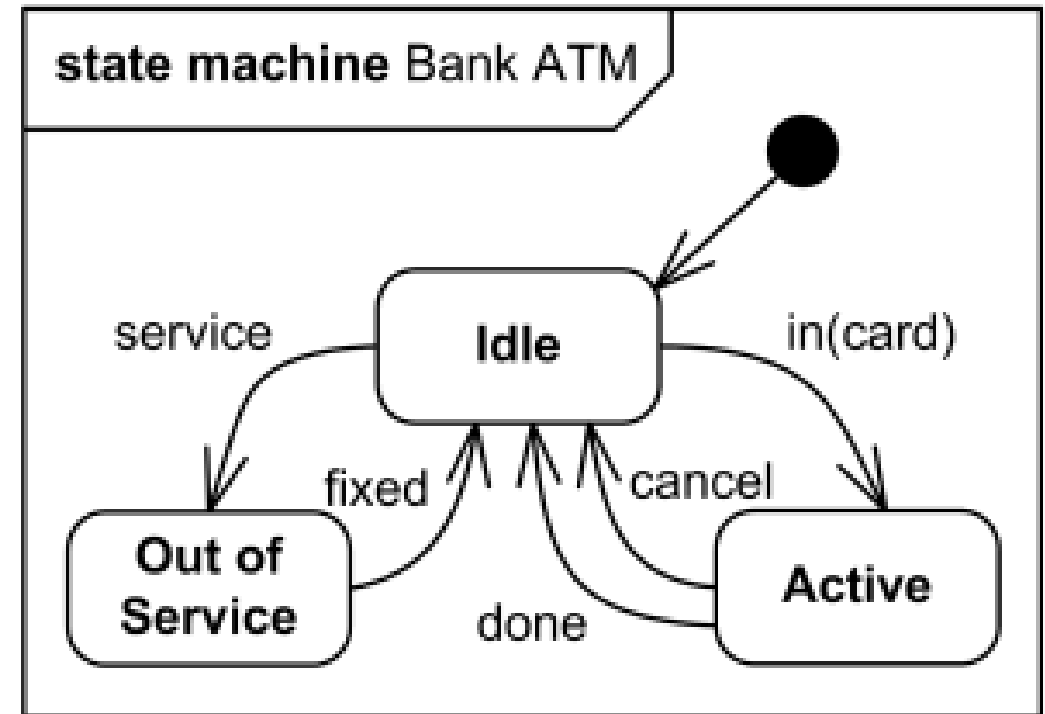
# Sequence diagrams

- 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

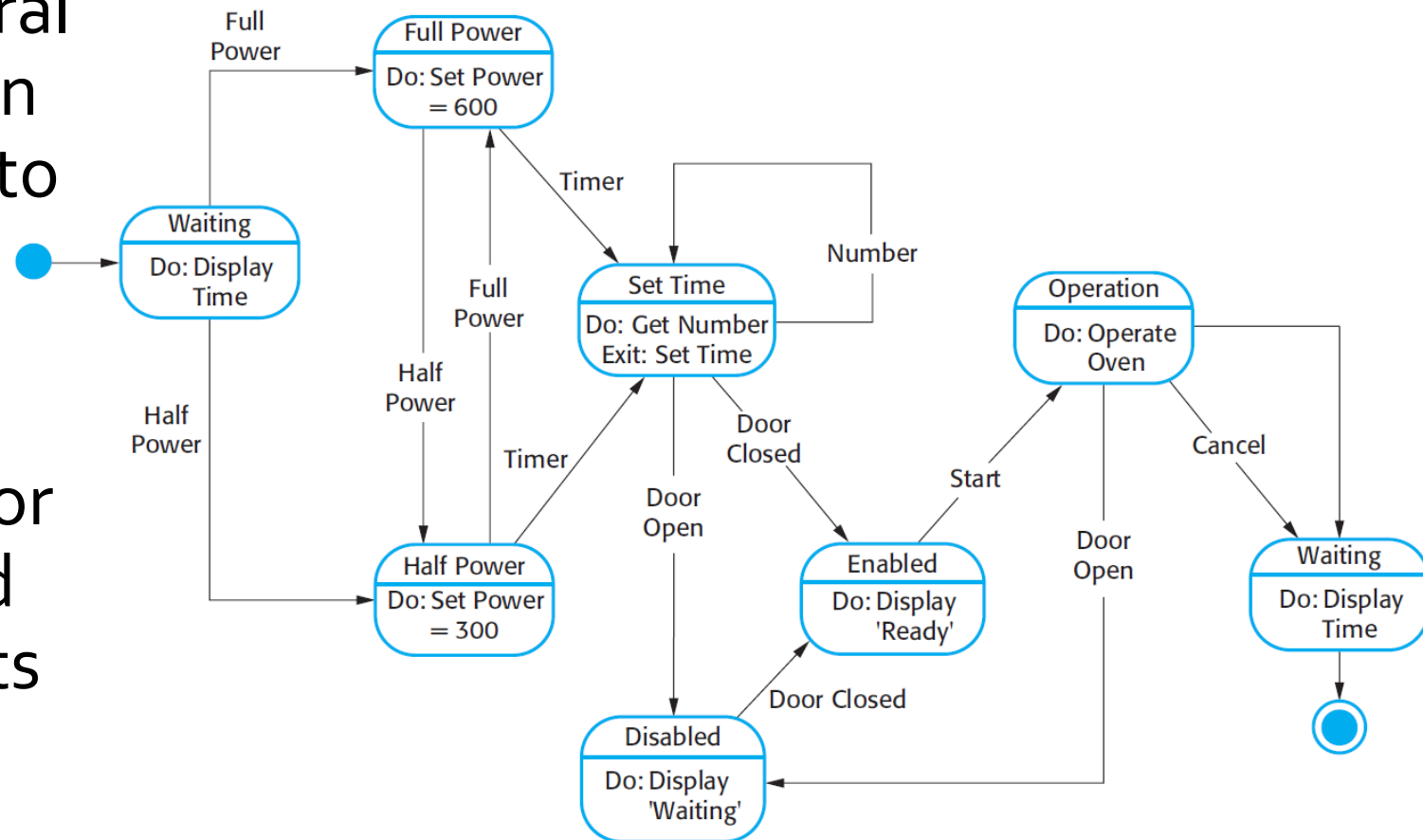
- State diagrams are the UML generalization of finite state automata from discrete math
- They describe a series of states that a system can be in and how transitions between those states happen
- Example from [uml-diagrams.org](http://uml-diagrams.org):





# 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

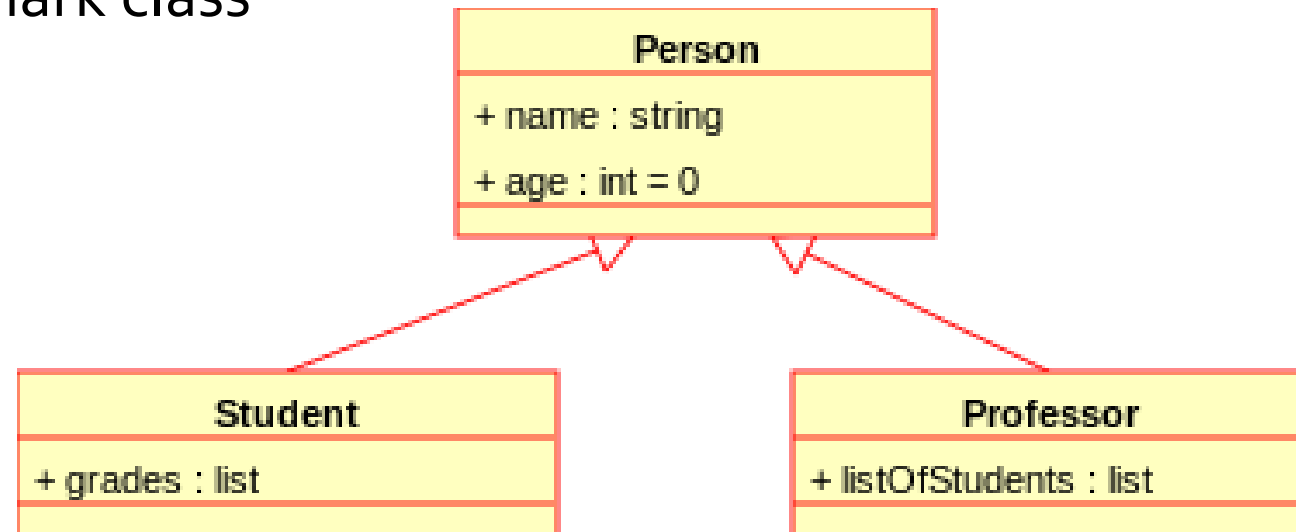


# Structural models

- 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](#):

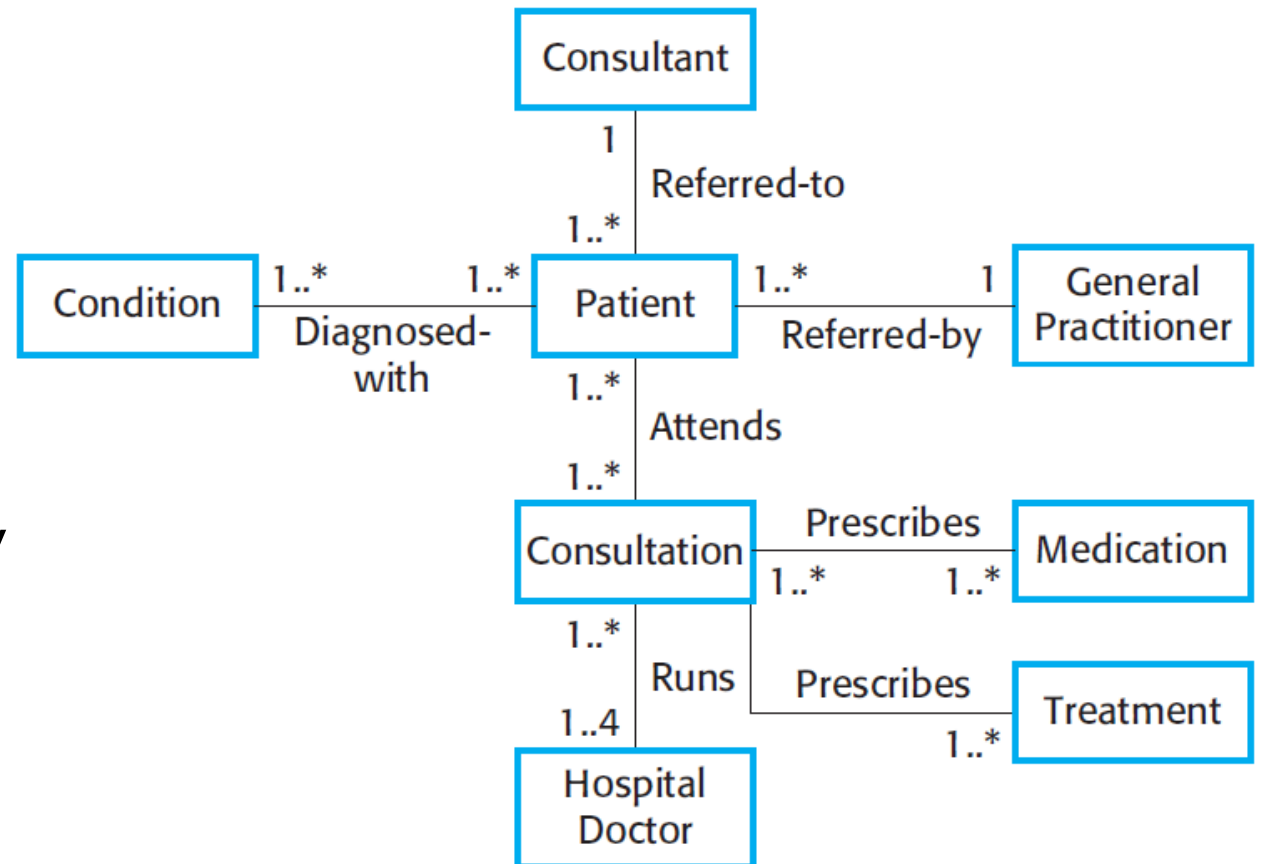


# Relationships

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

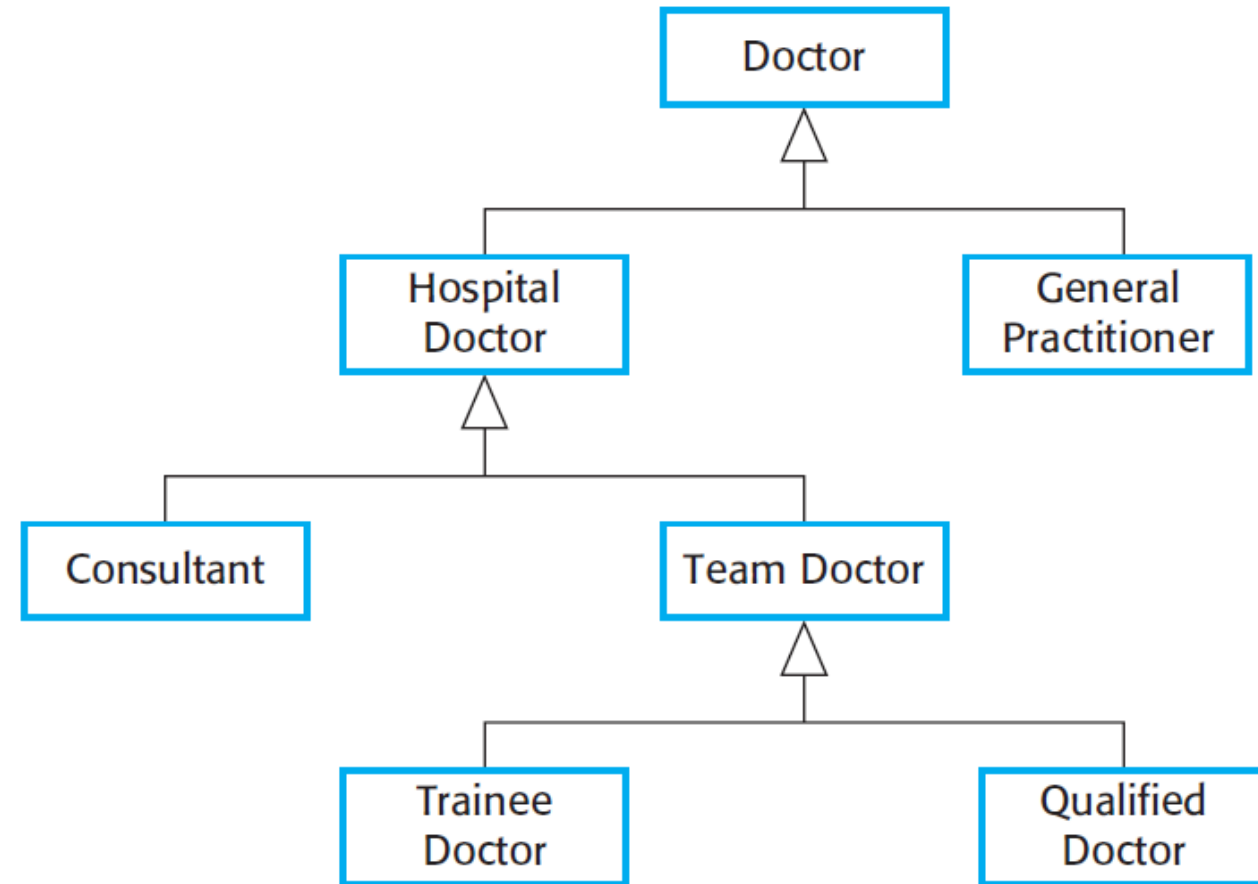


- Notations can be used to mark relationships as one to one, many to one, many to many, etc.
- These kinds of relationships are particularly important when designing a database



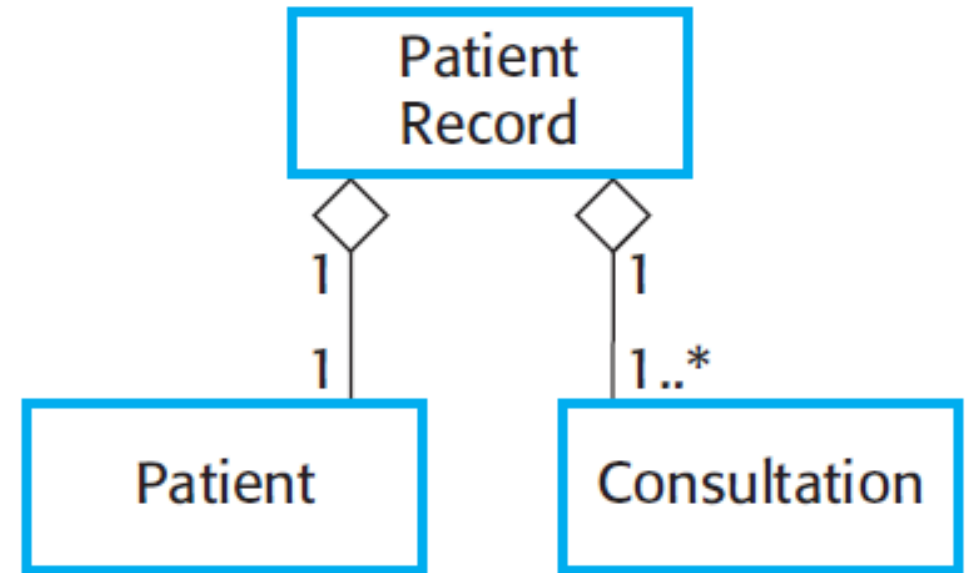
# Generalization

- 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



# Software Processes

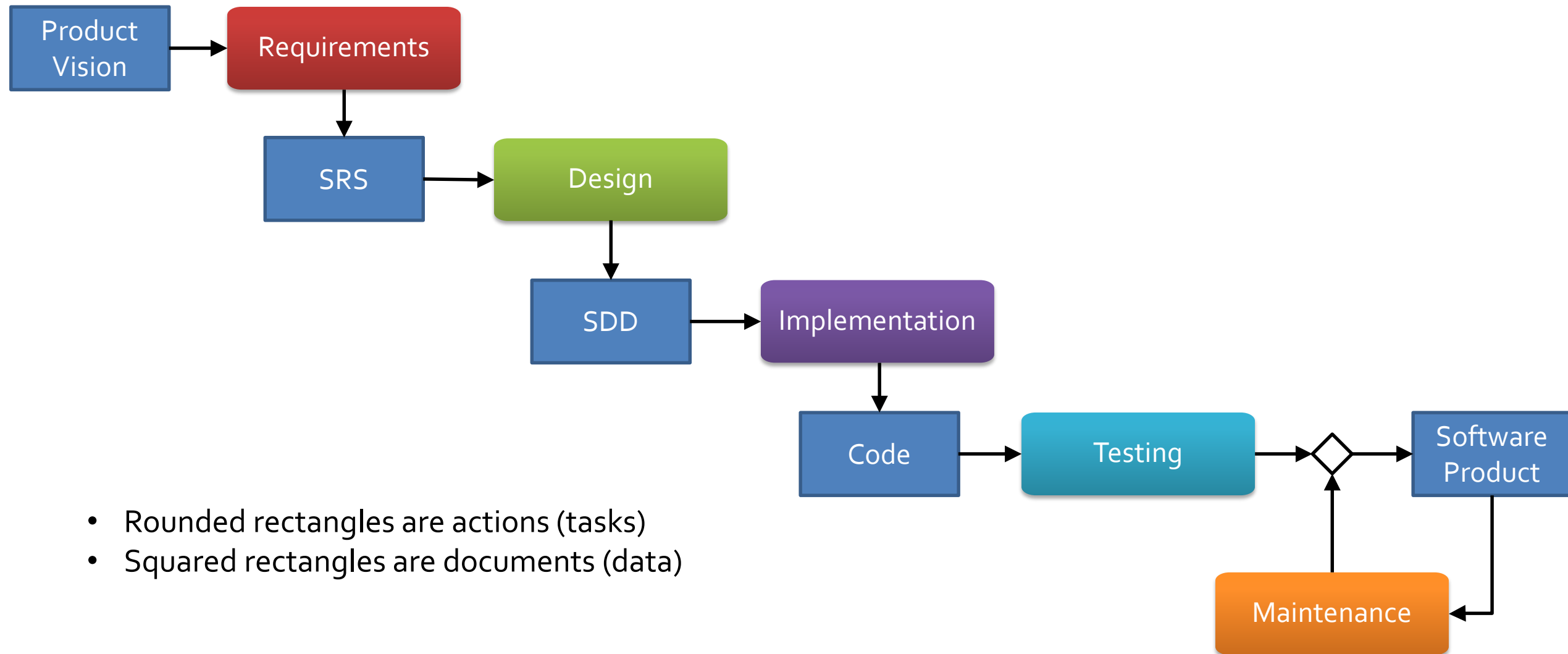
---

# Software processes

- A **software process** is a process used to make or support software
- A **software lifecycle process** shows the steps from product inception to retirement of the product
- A **model** is an entity used to represent another entity (the **target**)
- A **software process model** is a model for a software process
  - Usually a 2D diagram like a UML diagram



# Waterfall lifecycle model



# More on waterfall

1. Developers get a product vision
  2. From it and interaction with stakeholders, they create a software requirements specification (SRS)
  3. From the SRS, they create a software design document (SDD)
  4. Using the SDD, they implement the code
  5. Then they test the software product
  6. When the software is in use, problems are found, leading to maintenance and a new release
- The name "waterfall" is because each action flows to the next
    - Like a series of waterfalls
  - In principle, developers never return to an earlier action
  - In practice, earlier actions must always be reexamined because you never get it perfect the first time
  - Even so, the goal is to be as thorough as possible the first time

# 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

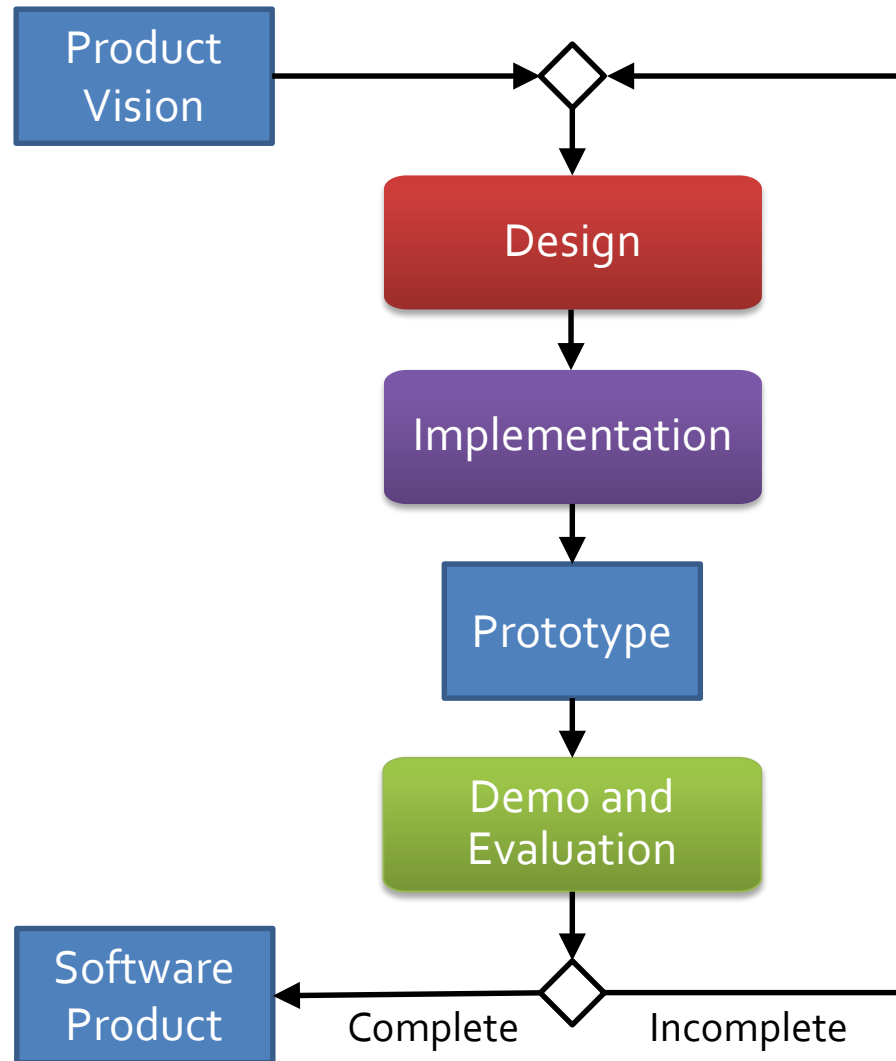
# Prototyping

- 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

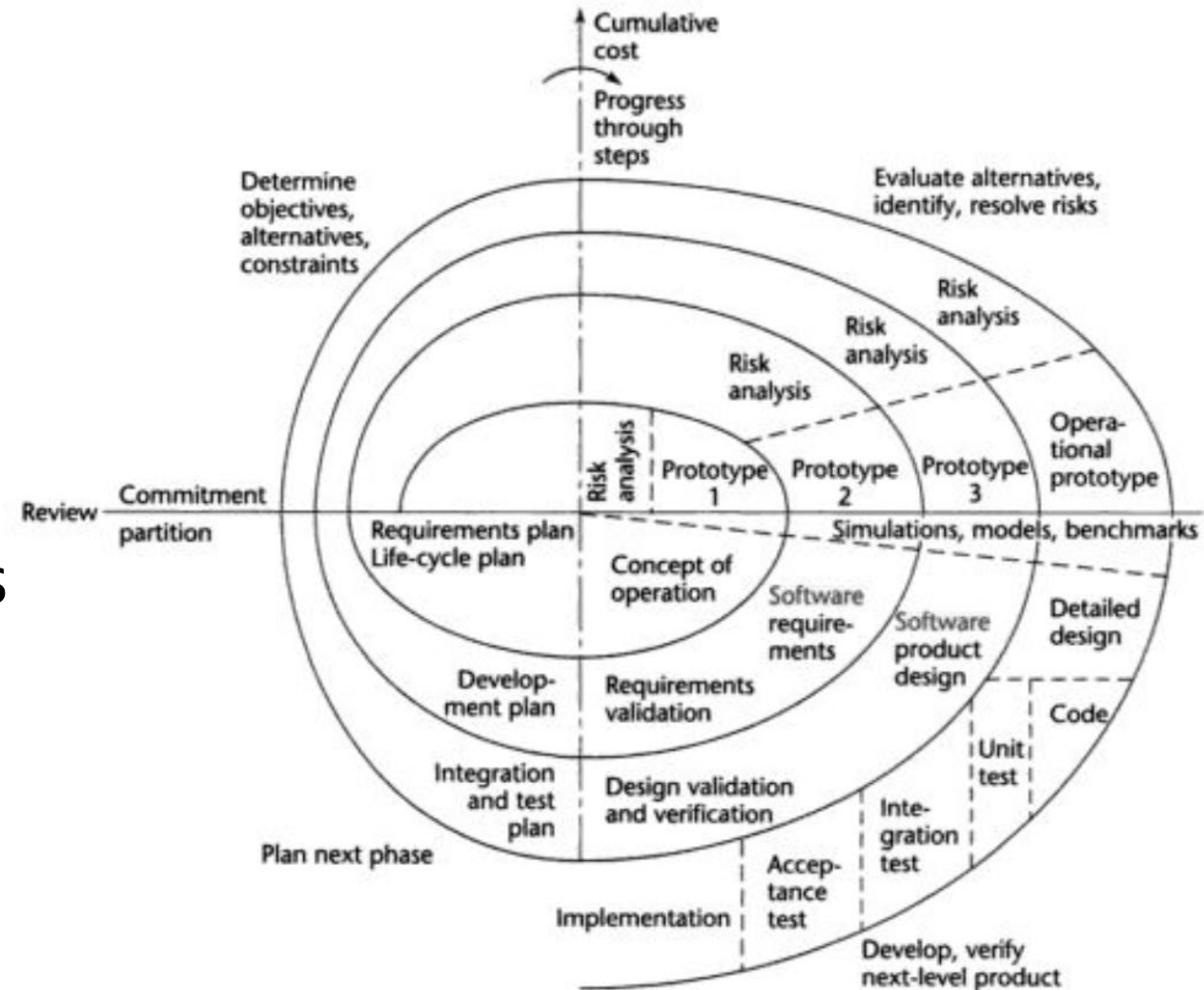
- Without the planning of a process like waterfall
  - It's hard to predict a reasonable deadline for the final product
  - It's hard to predict the budget
- Product design might be bad since the product evolved without following a plan
  - The biggest problem here is maintainability: How can new features 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

# Spiral model

- The spiral model is built around risk management
- Multiple cycles are used
- Each cycle starts by looking at goals
- Then evaluate different approaches to the goals in terms of risk
- The model on the right shows how the spiral model can be applied to waterfall



# 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 spiral process 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

# Rational Unified Process

- The **Rational Unified Process (RUP)** is a process that is both iterative and incremental
- Pure RUP is now rarely used, but it was a step in the evolution of modern agile methods
- RUP creates products in increments called **releases** made during a **cycle**
  - Each release is a working product
- Each cycle has four phases: inception, elaboration, construction, and transition
- Each phase has iterations divided into five workflows: requirements, analysis, design, implementation, and test

# 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

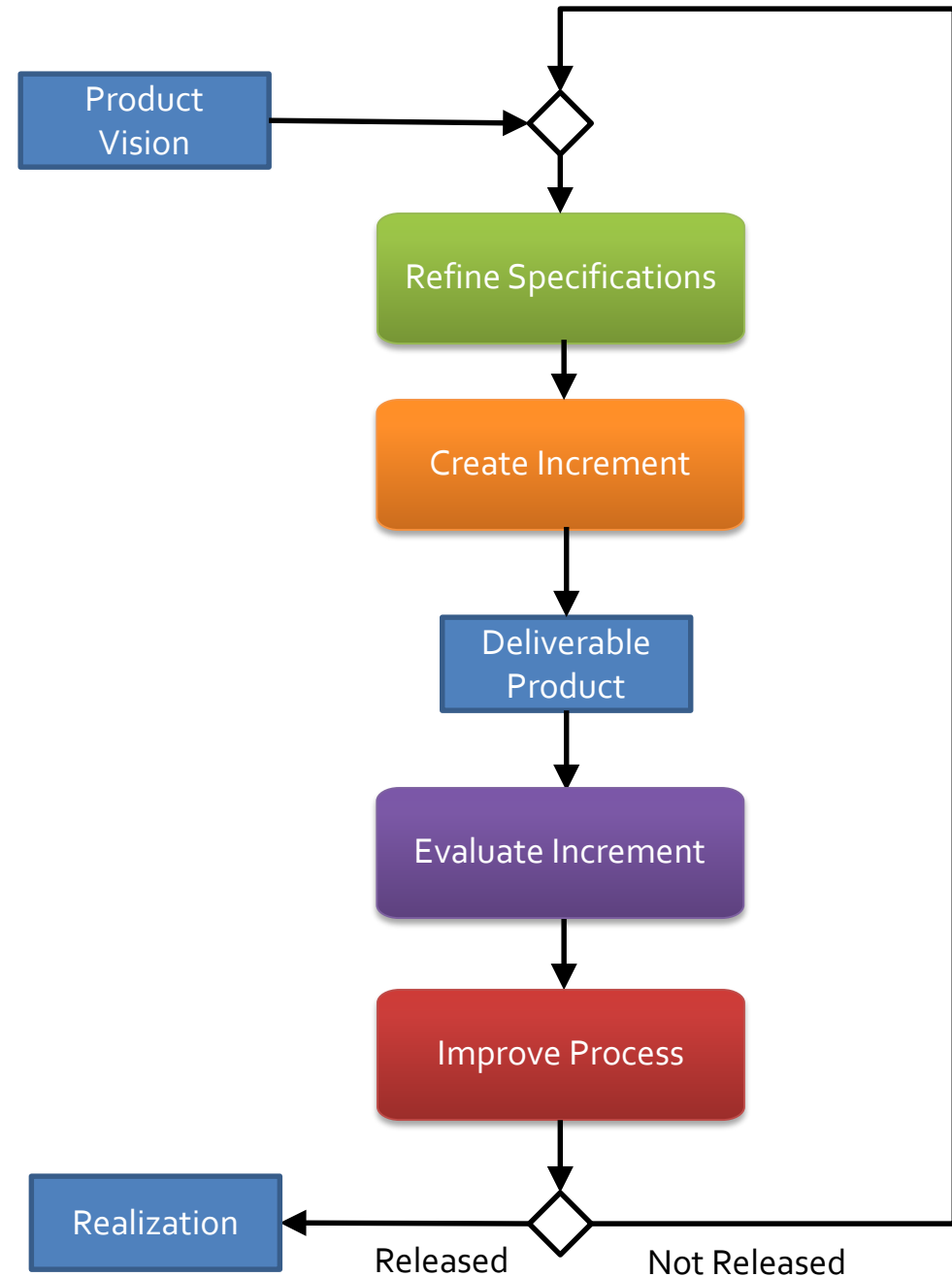
*That is, while there is value in the items on the right, we value the items on the left more.*

# 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

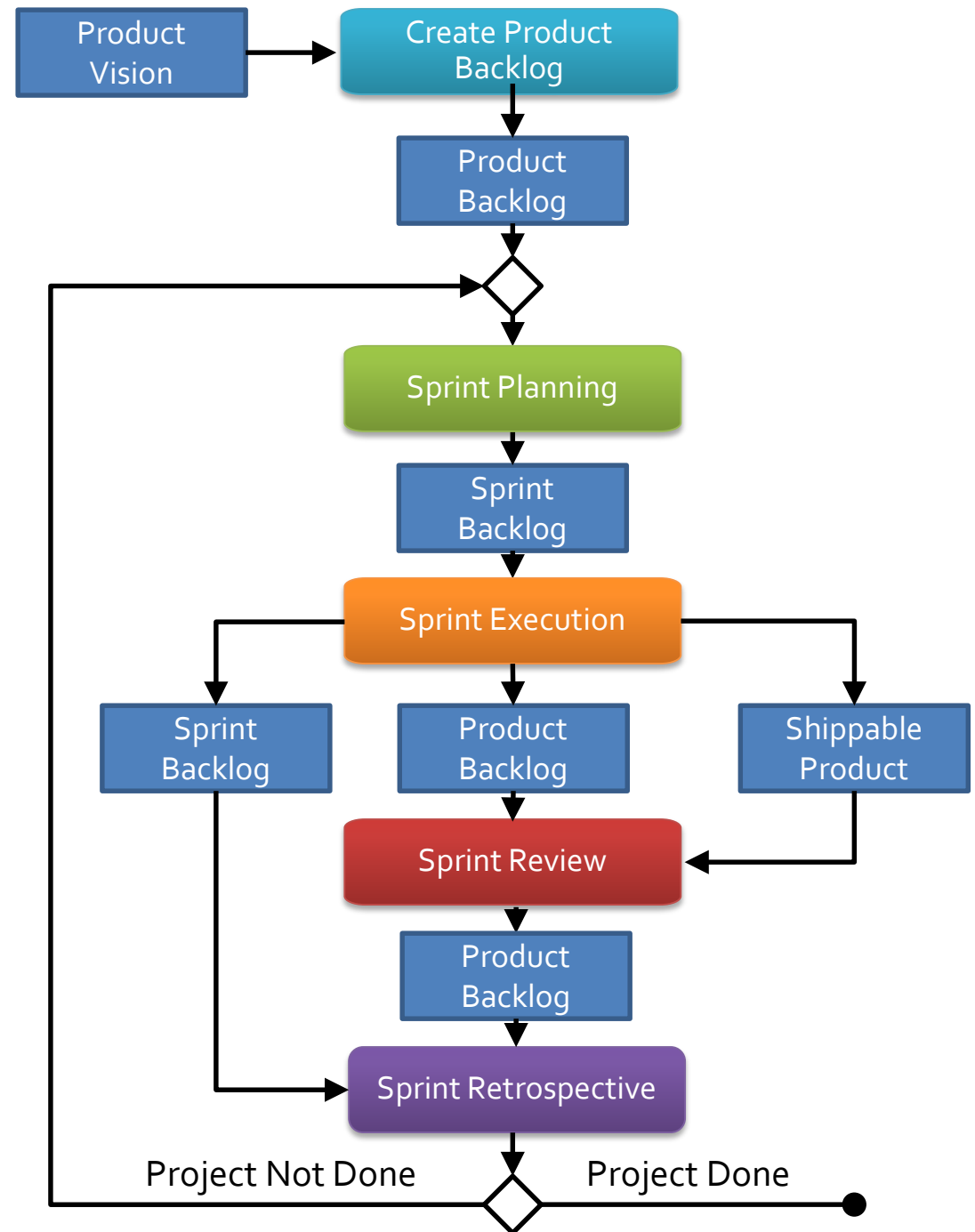


# Scrum

---

# 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

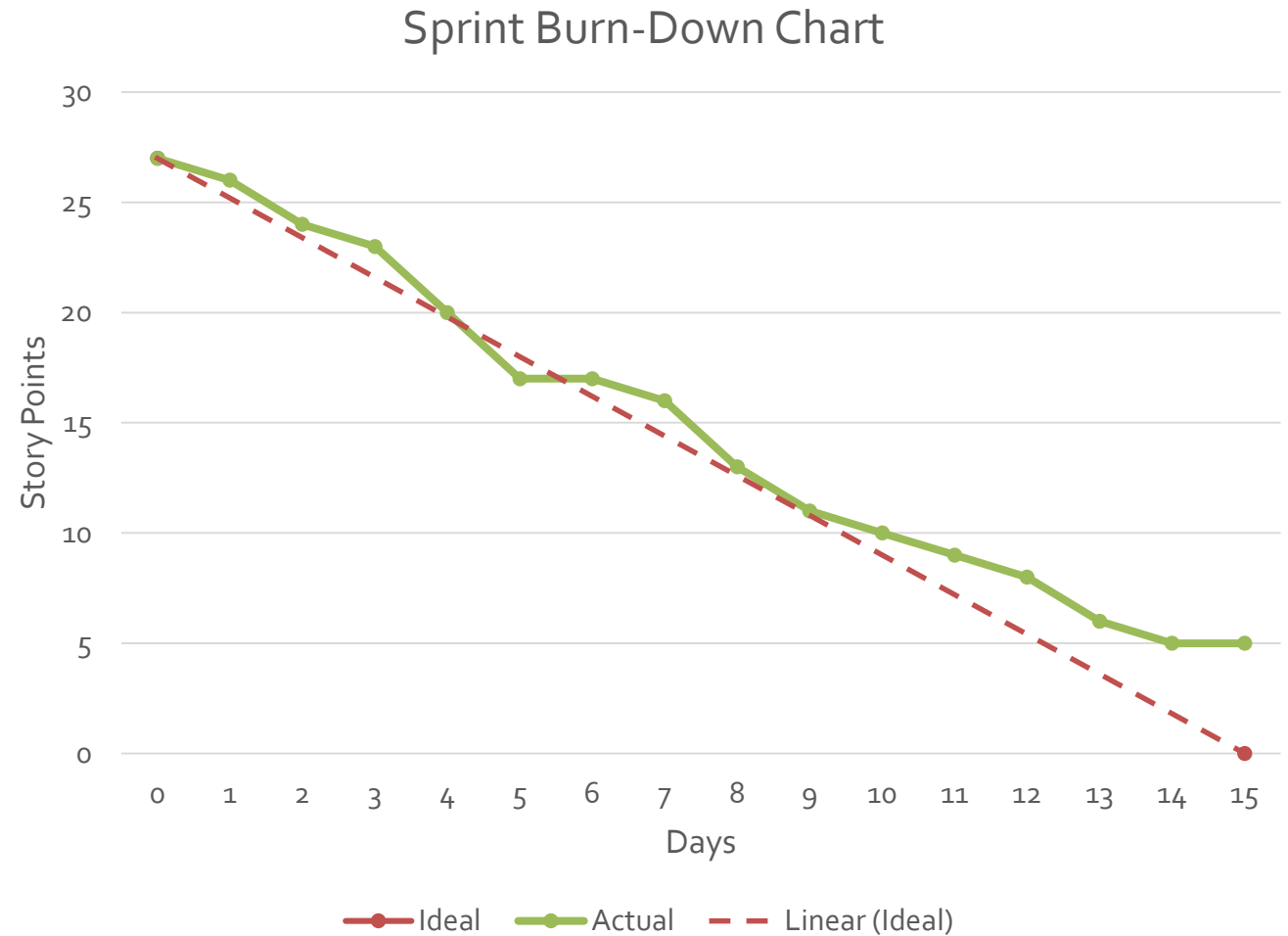
- **Velocity** is the amount of work done per sprint
- After a sprint, story points or ideal hours can be added up to see how much got done
- Past velocities can be used as a guide for how many story points can get done when planning the next sprint
- Ideally, tracking this information will help get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

# Burn charts

- **Burn charts** can be used to track progress on a project
  - **Burn-down charts** show how much work is left and are preferred for sprints
  - **Burn-up charts** show how much work has been done
- A burn-down chart shows the total effort in story points on the y axis and time on the x axis
- A straight line shows ideal progress while real progress is plotted against it



# Kanban boards

- To follow the progress of individual user stories, it's common to use **task boards**
- These boards are often called **kanban boards** because of the agile process Kanban that uses them
- Each row shows the progress of tasks related to the story on the left
- People use physical boards with sticky notes or electronic tools like Trello





# 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

# Quiz

---

# Upcoming

---

# Next time...

---

- Work day on Friday

# Reminders

- **Exam 1 is next Monday**
  - Review Chapters 1-3 and 5
- Finish the final version of Project 1 for Friday