Week 9 - Wednesday

COMP 3100

Last time

- What did we talk about last time?
- Refactoring
- TDD
- System testing
  - Alpha testing
  - Beta testing

Questions?

Deployment, Maintenance, and Support

Deployment, maintenance, and support

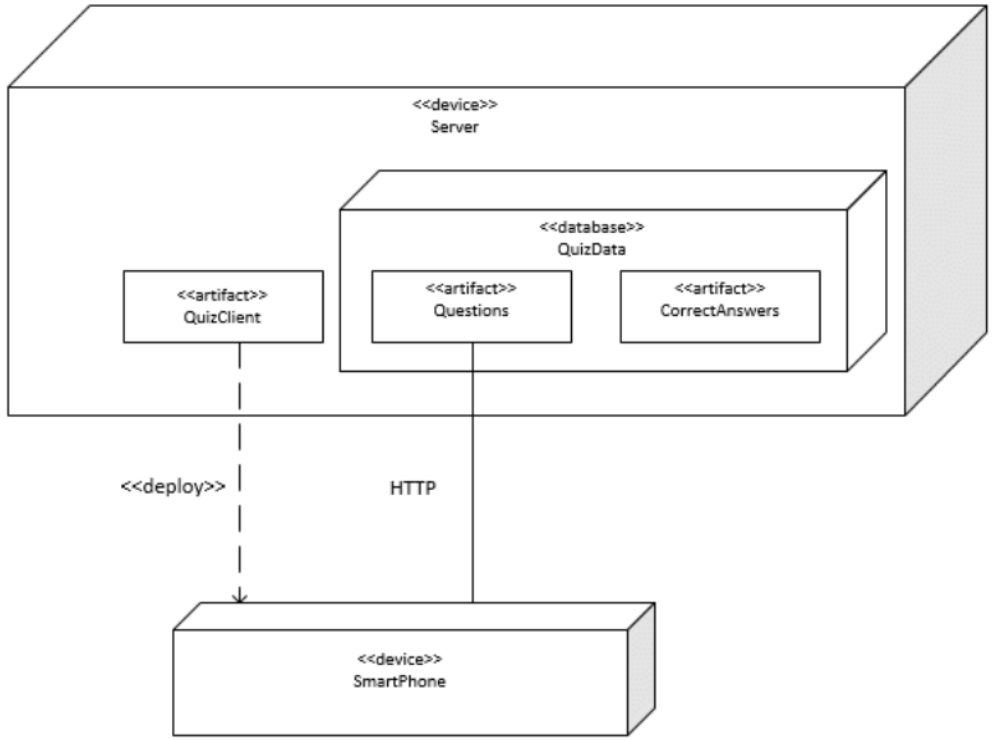
- The product has been designed, constructed, and tested…now what?
- Users will actually use the product in the **production environment**, the hardware and software systems where the product lives
- Making the product available in the production environment is called **deployment**
- Help that the developer (or their associates) provide to the user is called **support**
- Changes to the software after deployment are called **maintenance**

Physical architecture

- **Physical architecture** is how the program lives in a file system and executes on processors
  - As opposed to logical architecture, what we considered before
- Physical architecture can be structured by where it's installed, where it's executed, and where the data it uses is stored
- The following four categorizations are common:
  - **Personal:** Software is installed and executed on a user device, where the data is
  - **Shared:** Software is installed on a shared device and temporarily loaded on the user device where it is executed on user data
  - **Mainframe:** Software is installed and executed on a shared device accessible from a user device (terminal) using data stored on the shared device
  - **Cloud:** Software is installed on a shared device and temporarily loaded on the user device where it is executed using data stored on the shared device

Modeling physical architecture

- Like logical architecture, physical architectures can be modeled using UML
- UML deployment diagrams contain artifacts and nodes
  - **Artifacts** are physical components like files
    - Represented as rectangles with the stereotype «artifact» or an icon
  - **Nodes** are physical devices or execution environments like an operating system
    - Represented as boxes with the stereotype «device», «execution environment», or some other description
  - Communication between nodes is shown with a solid line
  - The deployment relationship is show by putting the artifact in the node box, listing the artifacts in the node box, or using a dashed arrow with the stereotype «deploy»



Deployment

- Deployment has the following steps
  - **Release:** Assembling the artifacts into a distributable package (like a zip file or an installer tool)
  - **Install:** Bringing the distributable package to the production environment and putting the artifacts in the right nodes
  - **Activate:** Start the executable artifacts
- Ideally, the installation and activation appear to be atomic
  - They happen as if they are a single activity
  - They can be rolled back to the state before the installation

Distribution channels

- Distribution has changed over time
- Once upon a time, someone with significant technical skill was needed to install software by hand
- Later, executable **installers** could be bought on disk or downloaded from the Internet
- **Stores** are now a common way to automatically install and update software
  - Examples: Apple Store, Windows Store, Steam
- **Package managers** are used for open source software
  - Examples: apt, rpm, dpkg, yum
- **Containers** like Docker are also used to provide software in a customized execution environment, ready to use

Maintenance

- Maintenance is a change to software after it's been deployed
  - **Corrective maintenance:** Changes that fix faults after they have given rise to failures
  - **Preventative maintenance:** Changes that correct faults before they give rise to failures (or to improve other characteristics like portability)
  - **Adaptive maintenance:** Changes that keep the product usable in a changing environment
  - **Perfective maintenance:** Changes that satisfy additional functional or non-functional requirements
- Since products are constantly changing in agile, it's not always clear what's maintenance and what's just another cycle of development

Cost of maintenance

- Maintenance is expensive
  - Some studies suggest that maintenance is responsible for 80% of the total effort surrounding a software product
  - This is exactly why Microsoft pushes OS versions off the supported list as soon as it can
- Maintenance is expensive for many reasons:
  - It goes on for a long time, maybe decades
  - Software is poorly written to begin with, and maintaining only gets harder if new features are added
  - Software structure deteriorates over time as changes are made
  - In traditional processes, maintenance can take a long time and still end up with a bad result
- Agile processes overcome some of these issues by making maintenance a natural continuation of development

Support

- Support are the activities between the user and a developer (or representatives of the developer) to help the user's experience
- Support is often put into two categories
  - **Professional support:** The person providing support is employed by or paid by the developer
  - **Community support:** The person providing support is another user or expert not employed by the developer
- Even professional support is usually not provided by the developers themselves
  - Support teams usually have lower skills and are paid less
  - Developers might not have the **ahem** interpersonal skills to deal with frustrated users
  - Support teams often have better knowledge of the application domain (the thing the product is being used for)

Support communication

- Support can come through synchronous communication channels like phone or chat
- Or through asynchronous channels like e-mail or forums
- Asynchronous channels are usually cheaper but lower quality
- Like with bugs, support teams can use issue tracking systems to track and prioritize support issues
- Support can be free or based on a fee
  - Sometimes different levels of support are provided depending on the fee
- Like maintenance, support can be a large part of the cost of a product and is sometimes neglected

Review

Exam format

- The exam will have:
  - Short answer questions
  - At least one diagram
  - Probably a JUnit test case or two to write
  - Probably some matching
  - One or two essay questions

- Look at the **point values** to determine where to invest your time

Software Quality Assurance

Eight dimensions of software quality

- **Functional suitability**
  - How much the product satisfies user needs
- **Performance efficiency**
  - Processing time and resources used
- **Compatibility**
  - How well the product can co-exist and interoperate with other products
- **Usability**
  - How easy the product is to learn and use
- **Reliability**
  - The extent to which the product does certain functions under given conditions and recovers from interruptions
- **Security**
  - Confidentiality, integrity, authenticity, non-repudiation, and accountability
- **Maintainability**
  - How easy it is to modify, adapt, and reuse the product
- **Portability**
  - How easy it is to make the product work in a different computing environment

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

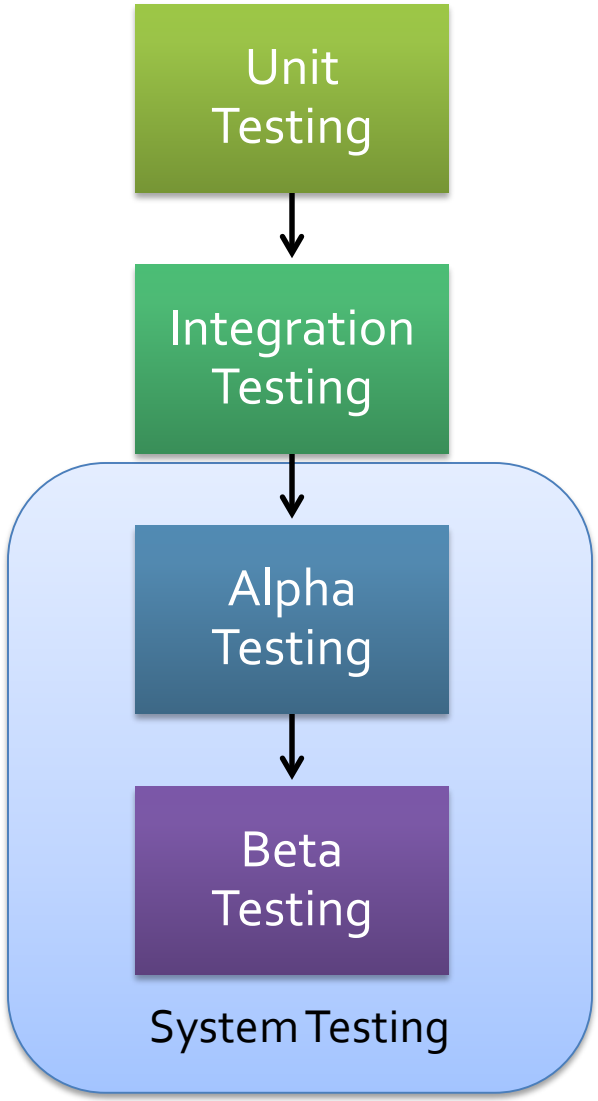
- There's a formal name for just looking at your code for errors: a **desk check**
- A **walkthrough** is when you explain your code to someone else
- An **inspection** is a more formal process with trained inspectors
- Inspection roles:
  - **Moderator** schedules and runs the meeting and distributes 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

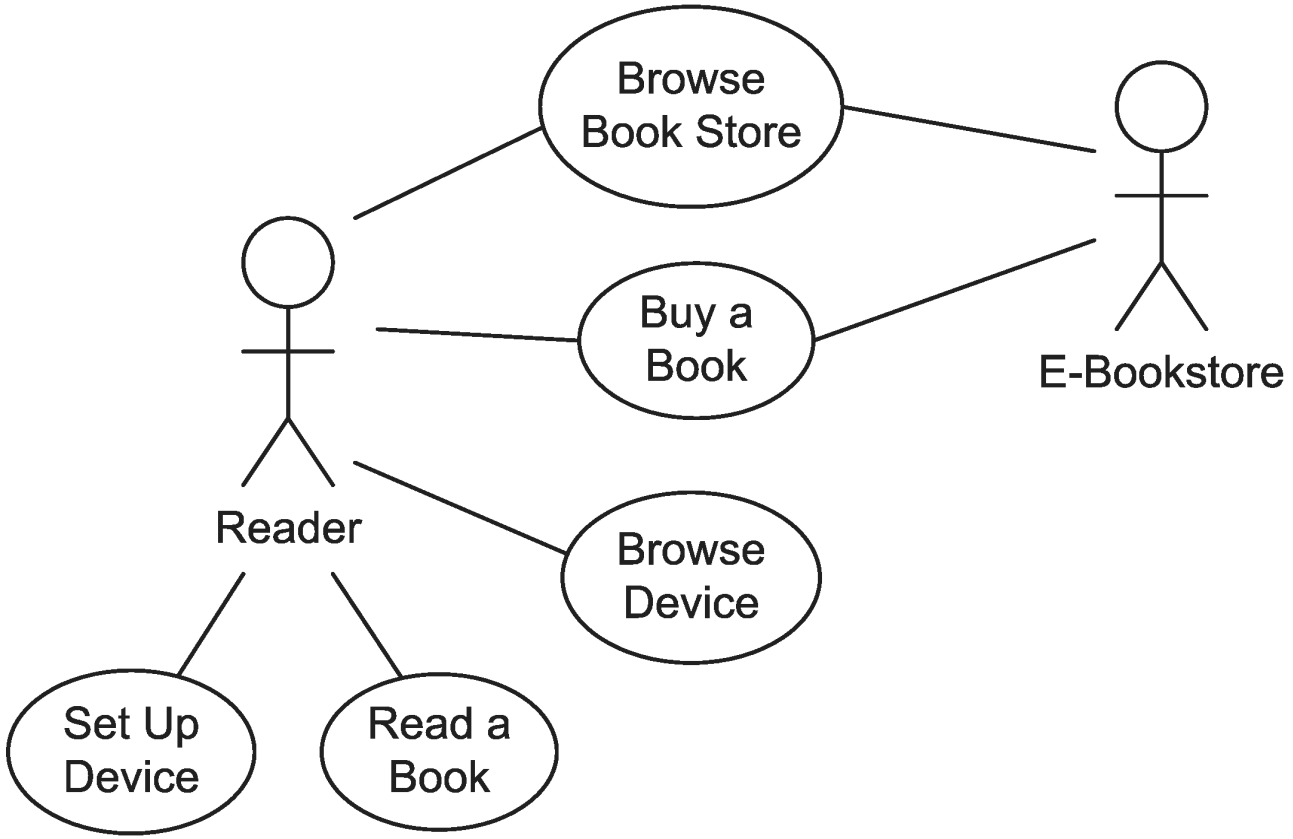
- **Effectiveness:** User can access all the features they need
- **Efficiency:** Users can achieve their goals quickly
- **Safety:** Users and computers aren't harmed
- **Learnability:** Users become proficient quickly
- **Memorability:** Users regain proficiency quickly after time away from the product
- **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

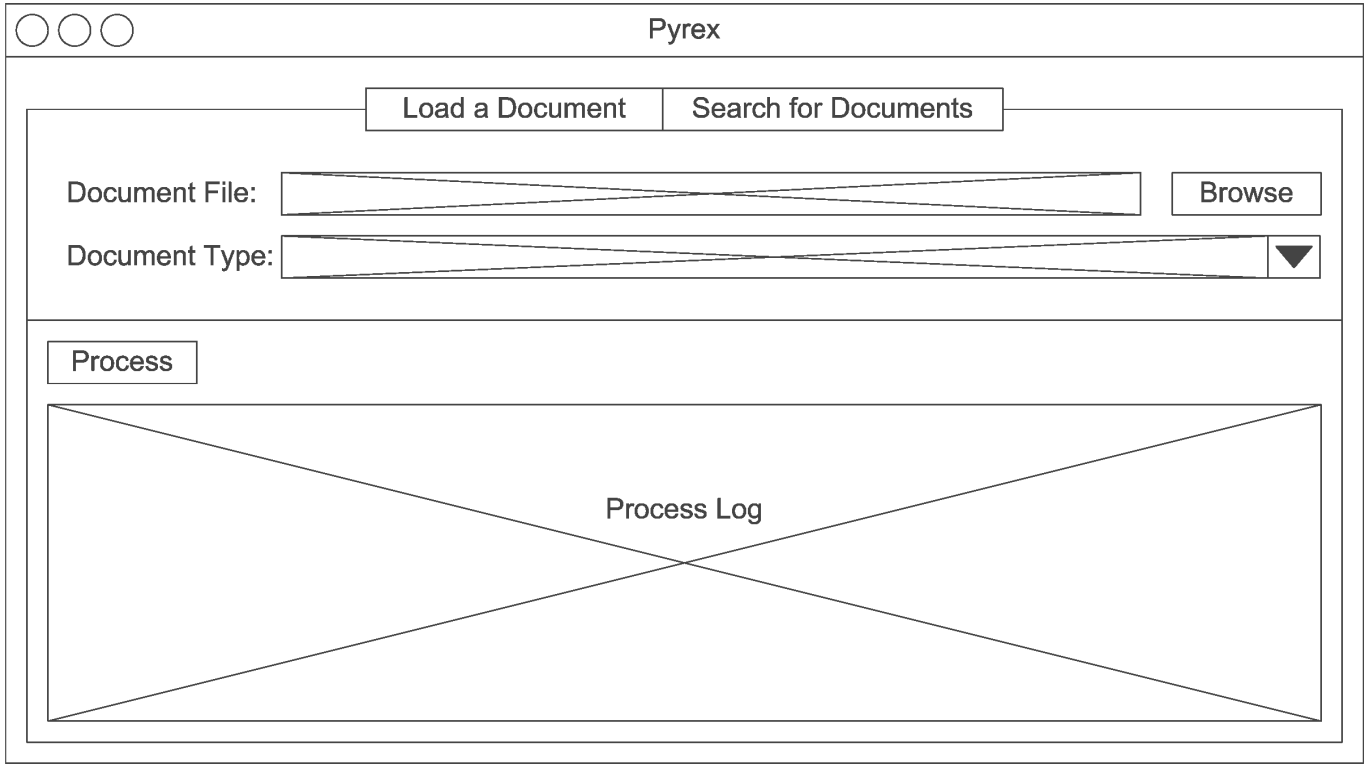
Use case diagrams

- 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

Preventing design defects

- A number of approaches can be used to avoid design defects
- **Design principles:** Using a list of good principles helps you make good choices
- **Design notations:** Using good notations (often UML diagrams) helps designs be complete and consistent
- **Design processes:** Using established processes for designs helps avoid mistakes
- **Design patterns:** Using patterns, models designed to be imitated, reuses solutions that have worked in the past (and make design easier)

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

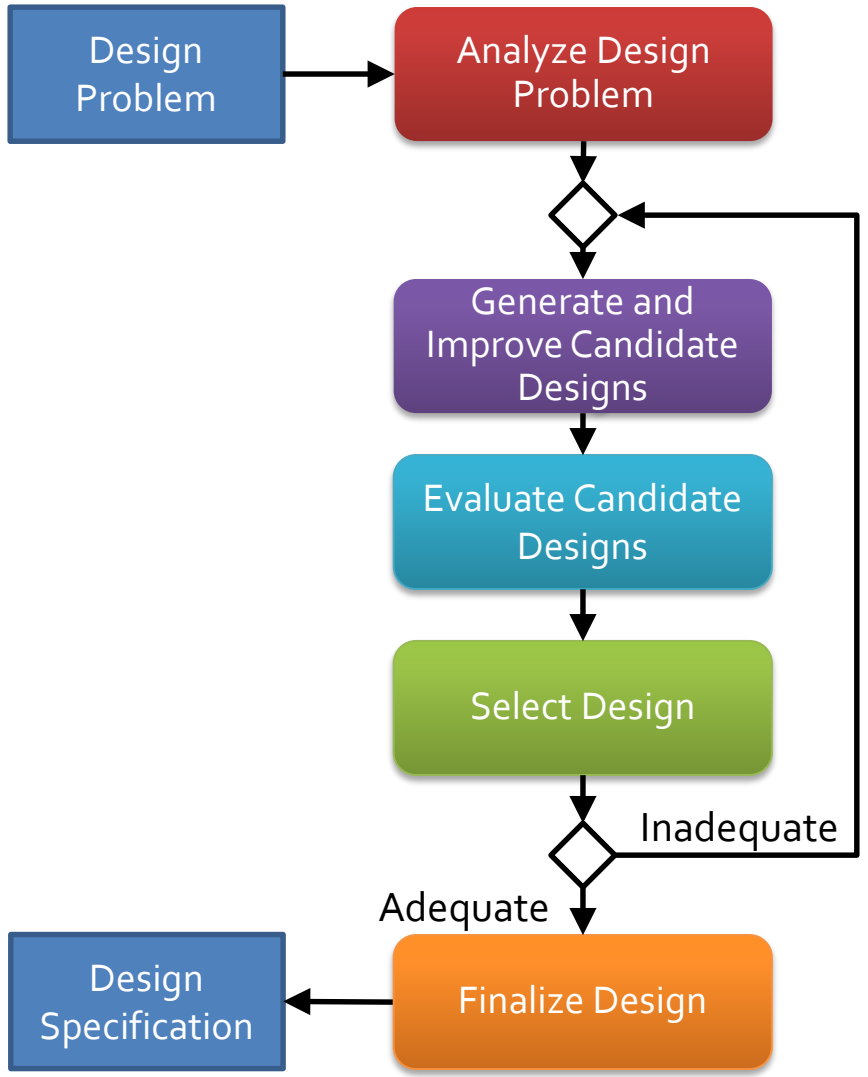
- **Module coupling** is the amount of connectivity between two modules
- Modules can be coupled in the following ways:
  - One class is an ancestor of another class
  - One class has a member whose type is another class
  - One class has an operation (method) parameter whose type is another class
  - 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

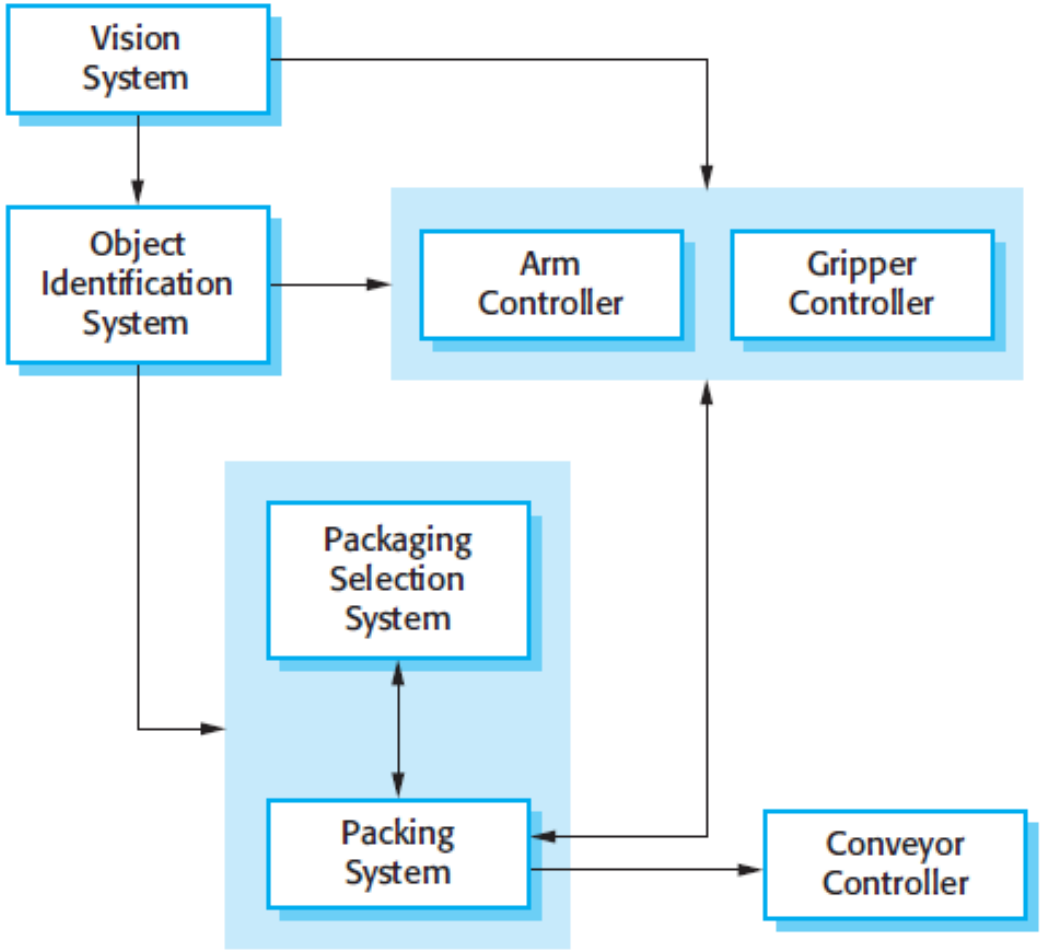
Design process

- 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



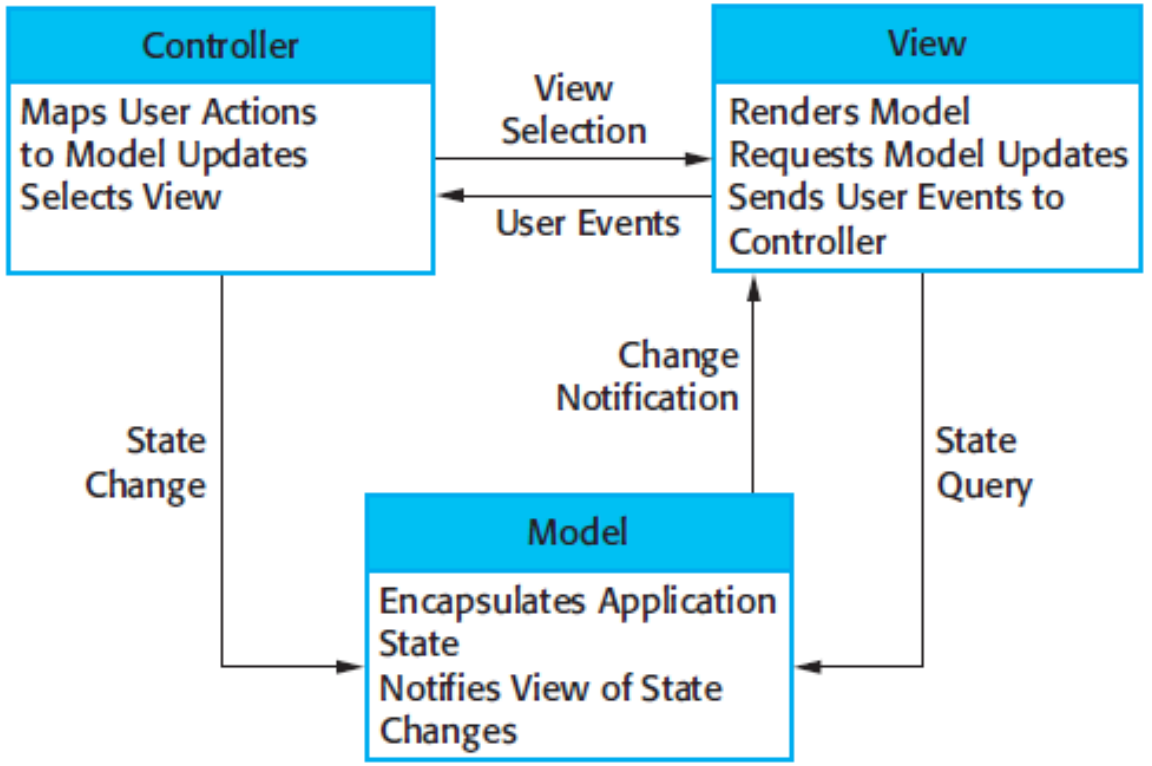
Architectural design

- **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



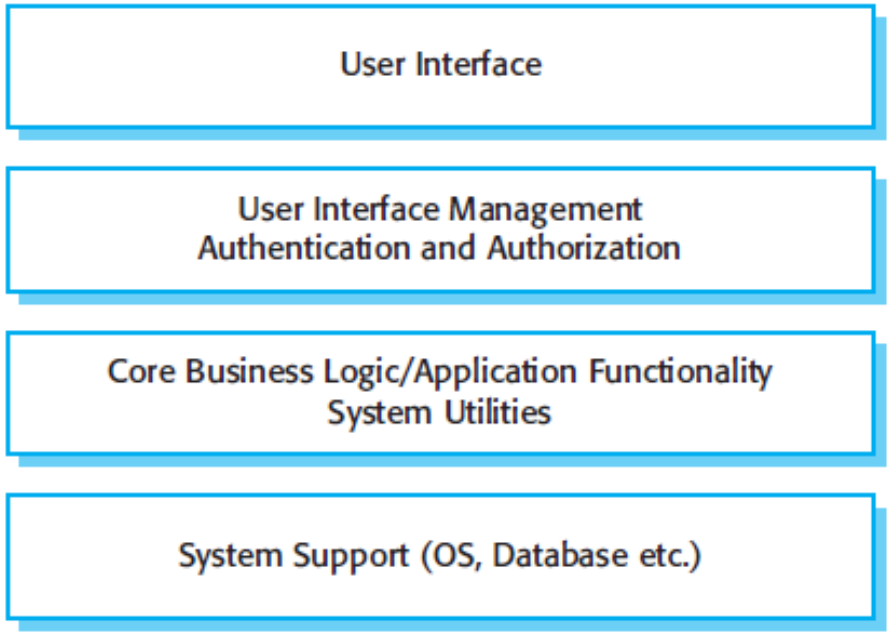
Model-View-Controller

- 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



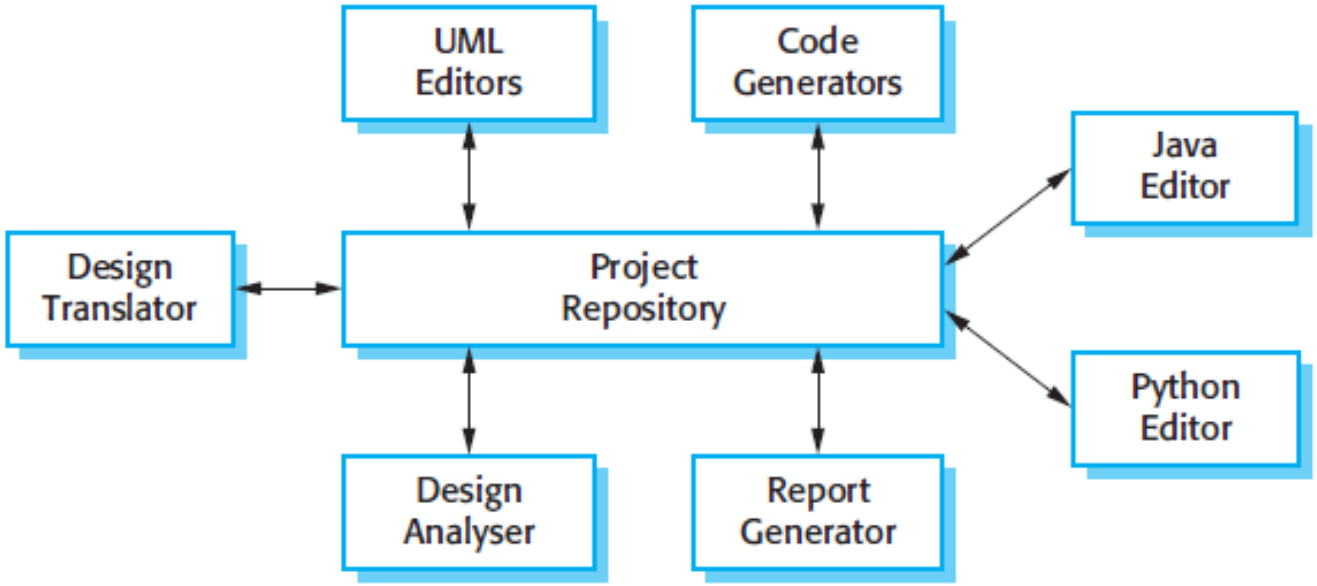
Layered style

- 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



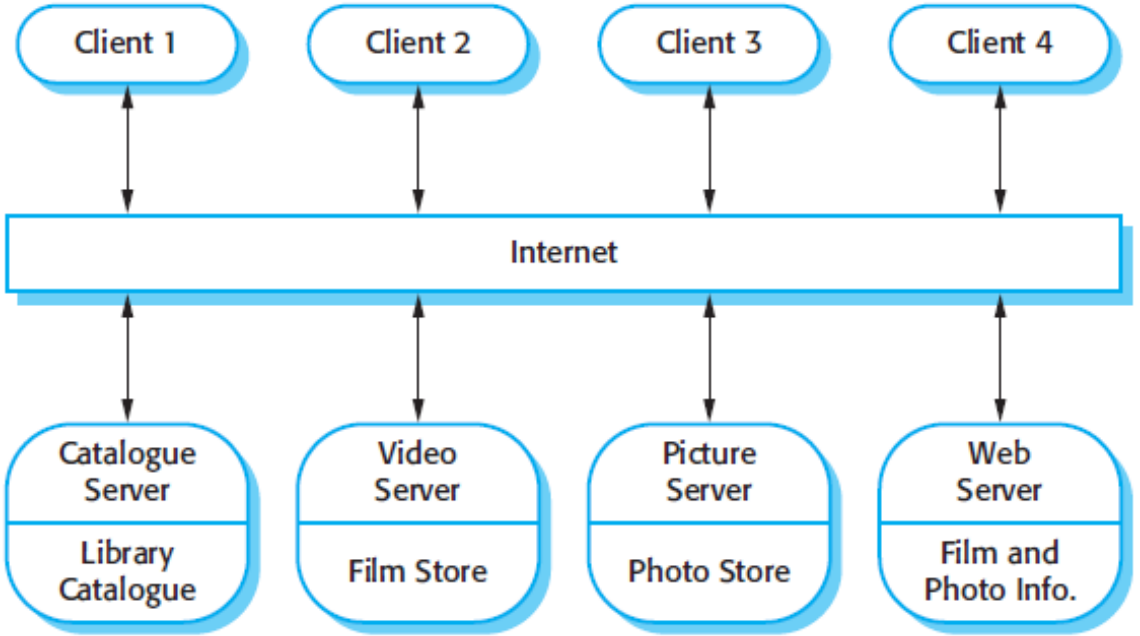
Repository style

- 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



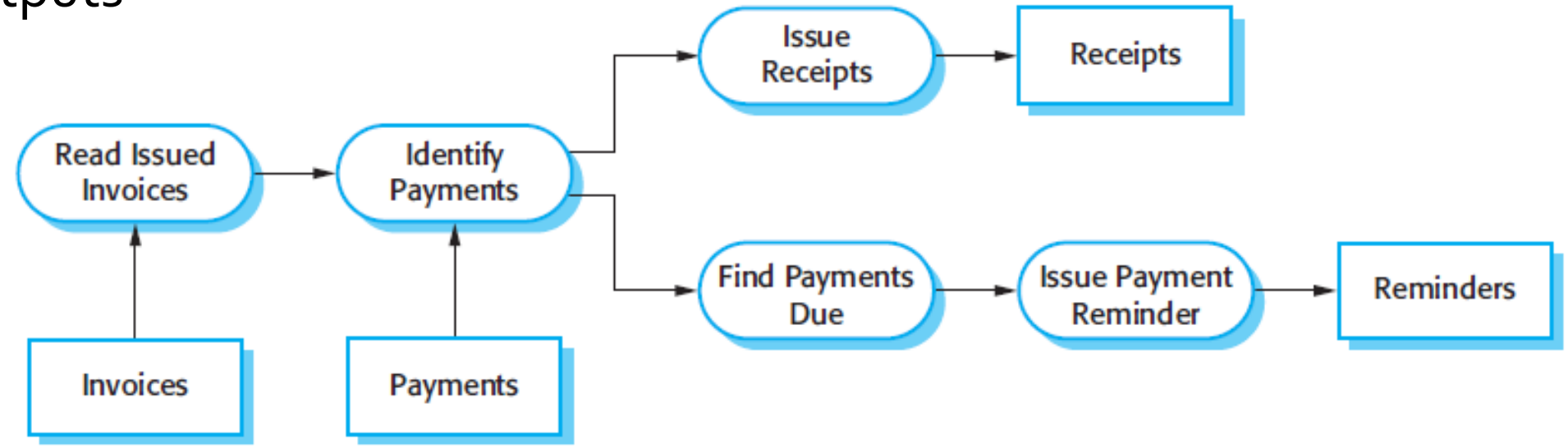
Client-server architecture

- 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



Project Scheduling

Project scheduling

- Project scheduling is organizing the work
  - Into separate tasks
  - When the tasks will be done
  - Who will do them
- Both waterfall and agile approaches benefit from scheduling
  - For waterfall, all tasks in the project are scheduled
  - For agile, there might be an overall schedule for when major phases of the project will be completed
- Tasks should last at least a week but not more than two months
  - A task taking more than two months should be broken into subtasks
- It's helpful to have visualizations of these tasks

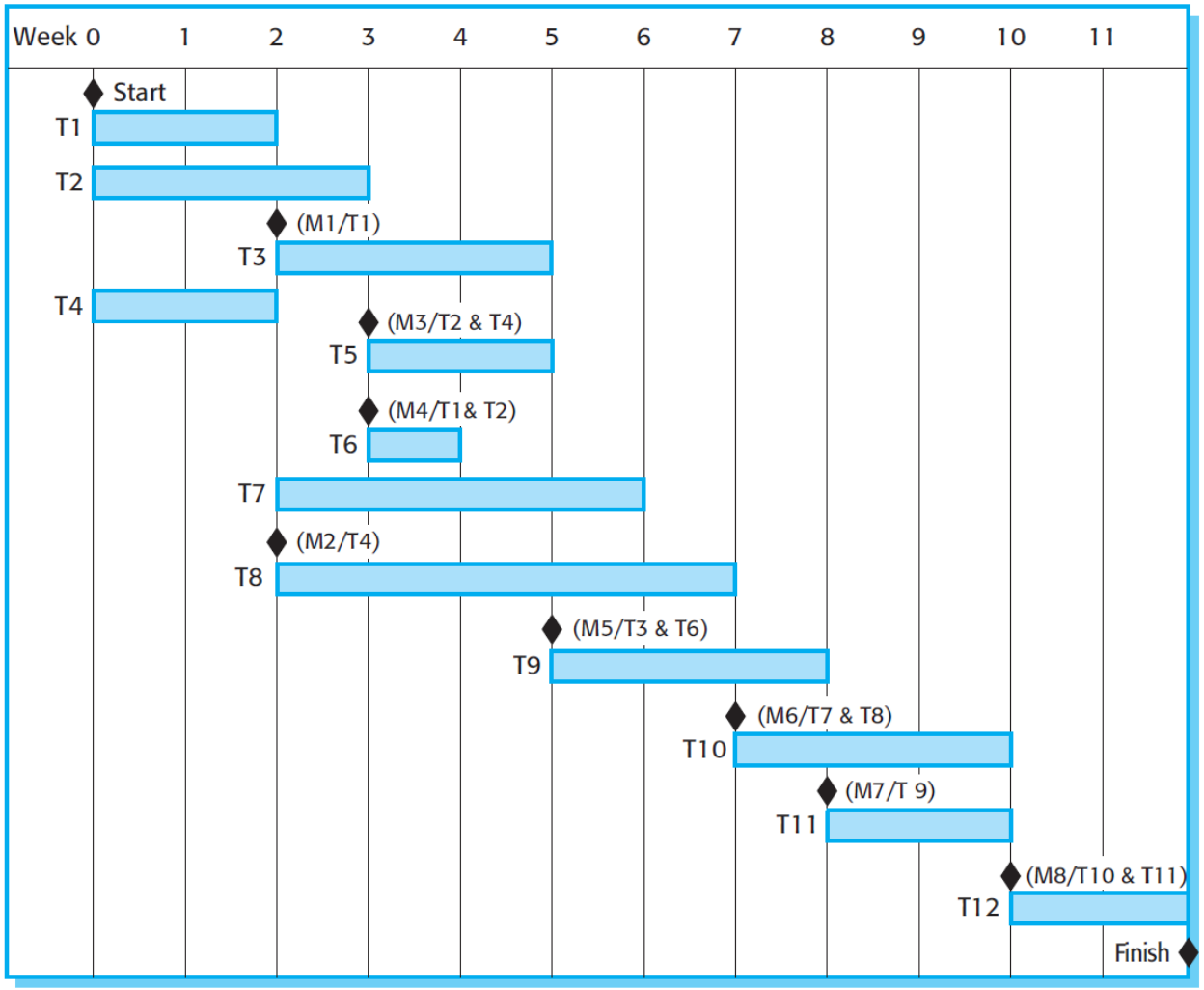
Example of tasks

- This table shows all the task information, but it's hard to visualize
- M is used to label milestones

Task	Effort (person-days)	Duration (days)	Dependencies
T1	15	10	
T2	8	15	
T3	20	15	T1 (M1)
T4	5	10	
T5	5	10	T2, T4 (M3)
T6	10	5	T1, T2 (M4)
T7	25	20	T1 (M1)
T8	75	25	T4 (M2)
T9	10	15	T3, T6 (M5)
T10	20	15	T7, T8 (M6)
T11	10	10	T9 (M7)
T12	20	10	T10, T11 (M8)

Gantt charts

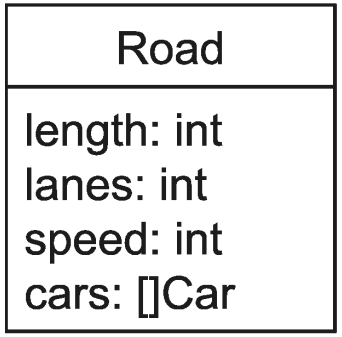
- Gantt charts show the same information, but in a much clearer way
  - Bars shows the length of each task
  - Dependencies are shown by the starting point of each task
- Recall that you made a Gantt chart for Project 2
- For it, you needed to break down your product into tasks and figure out which tasks are dependent on which

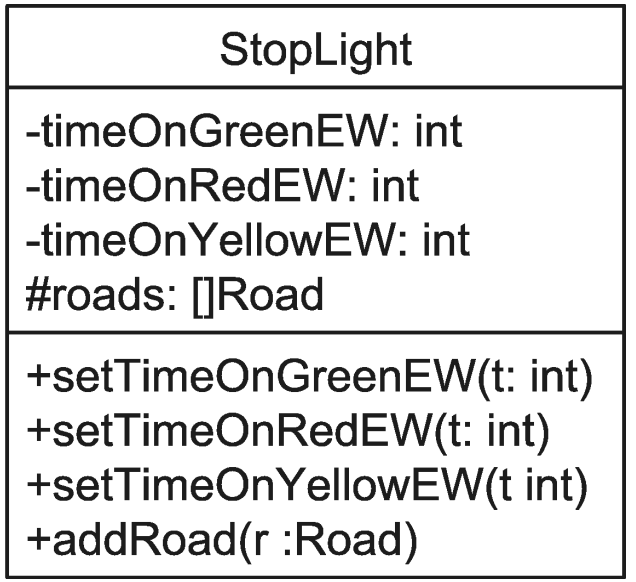


Detailed Design

More depth on class diagrams

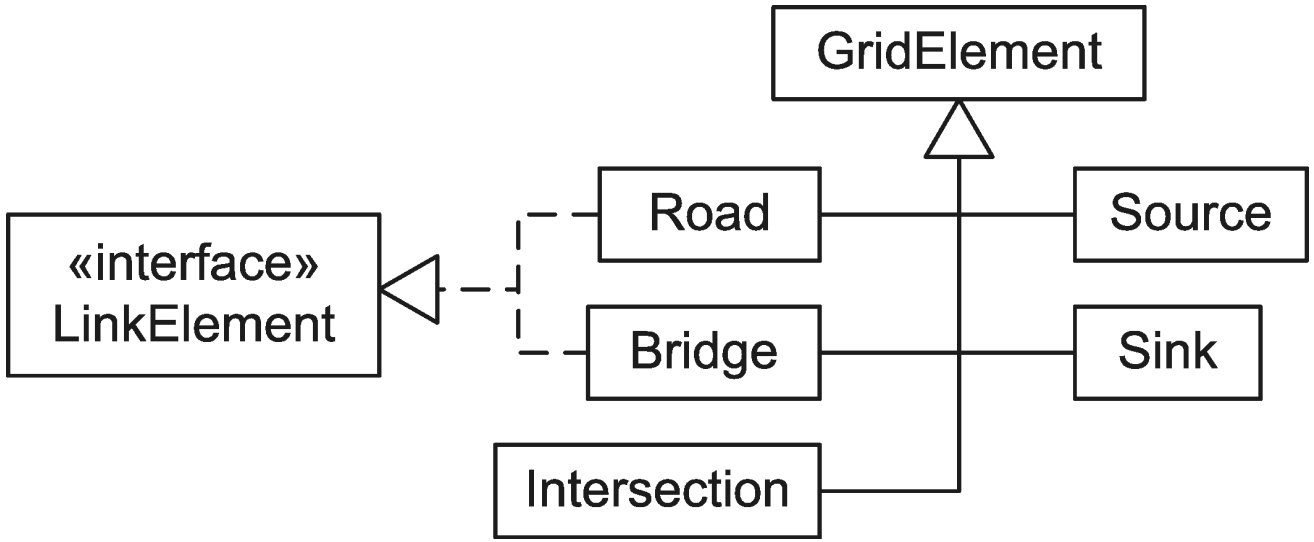
- **Class diagrams** are made up of **class symbols** (rectangles)
- These class symbols contain one or more **compartments**
- The top compartment has the class name
- A second, optional compartment often contains attributes (called member variables in Java classes)
  - Often followed by a colon with 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





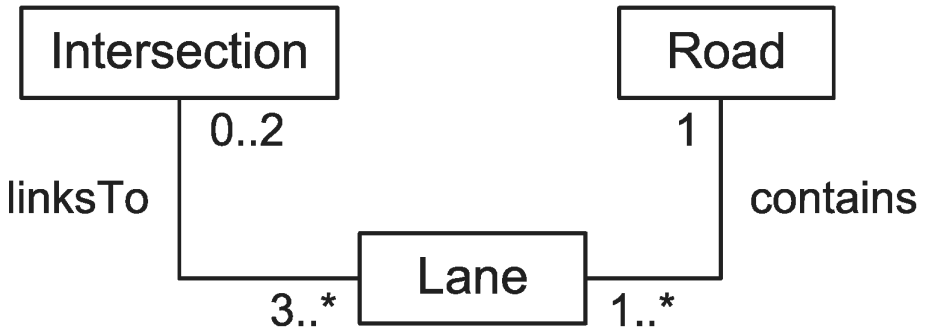
Inheritance and interfaces in class diagrams

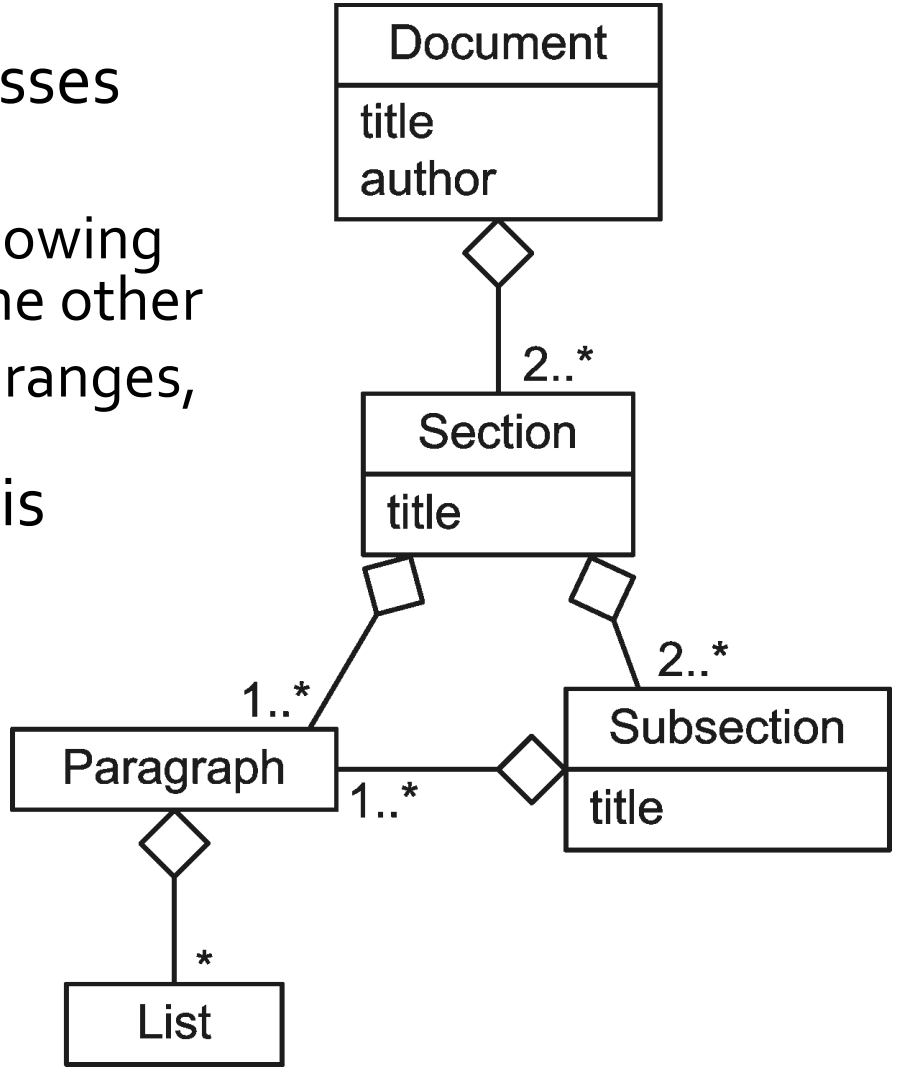
- Inheritance is shown with the **generalization** connector
  - A solid line from the child class to a solid triangle connected to the parent class
  - Confusingly, this means that children classes point at their parent classes
- Interfaces look like classes but are marked with **«interface»** above 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



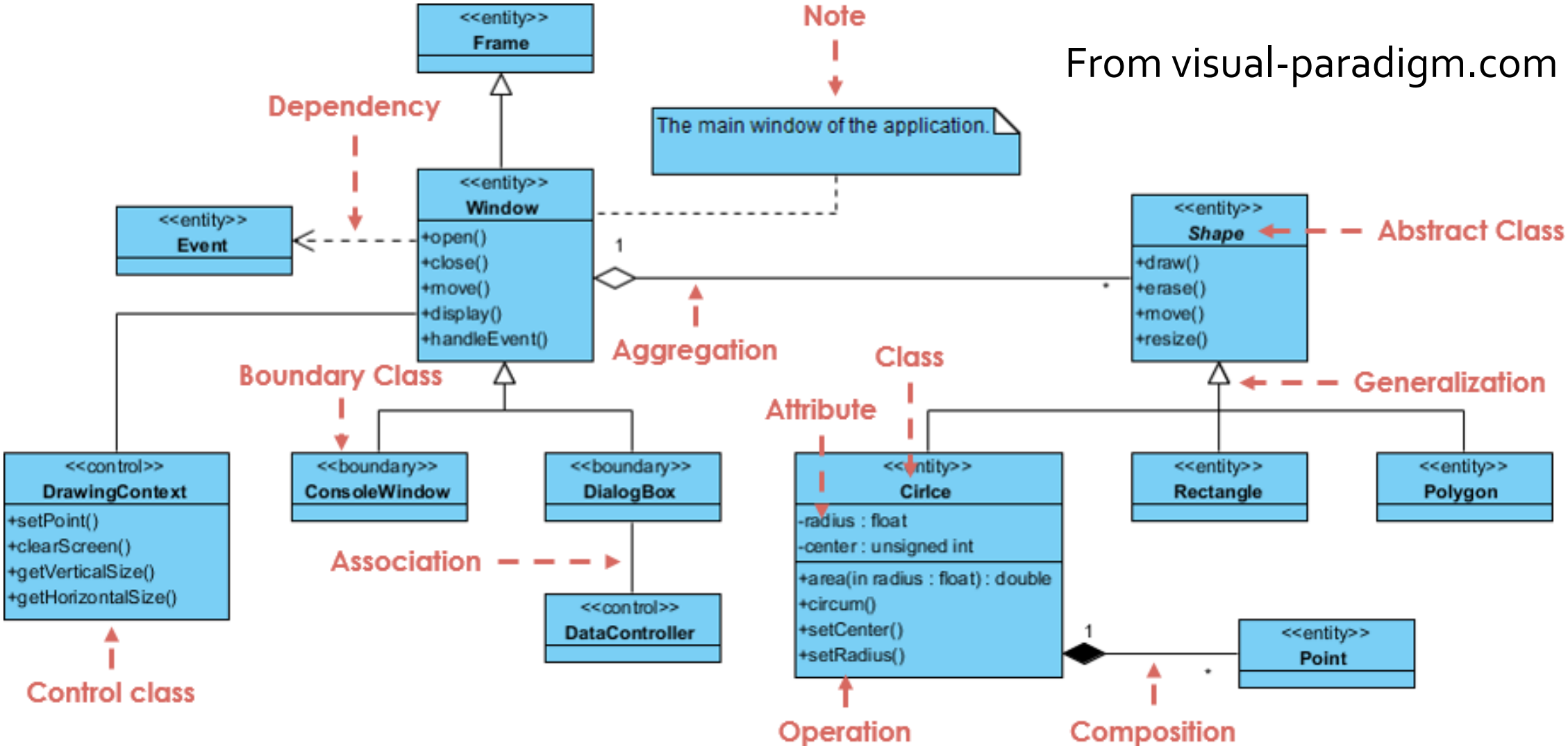
Other associations

- **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



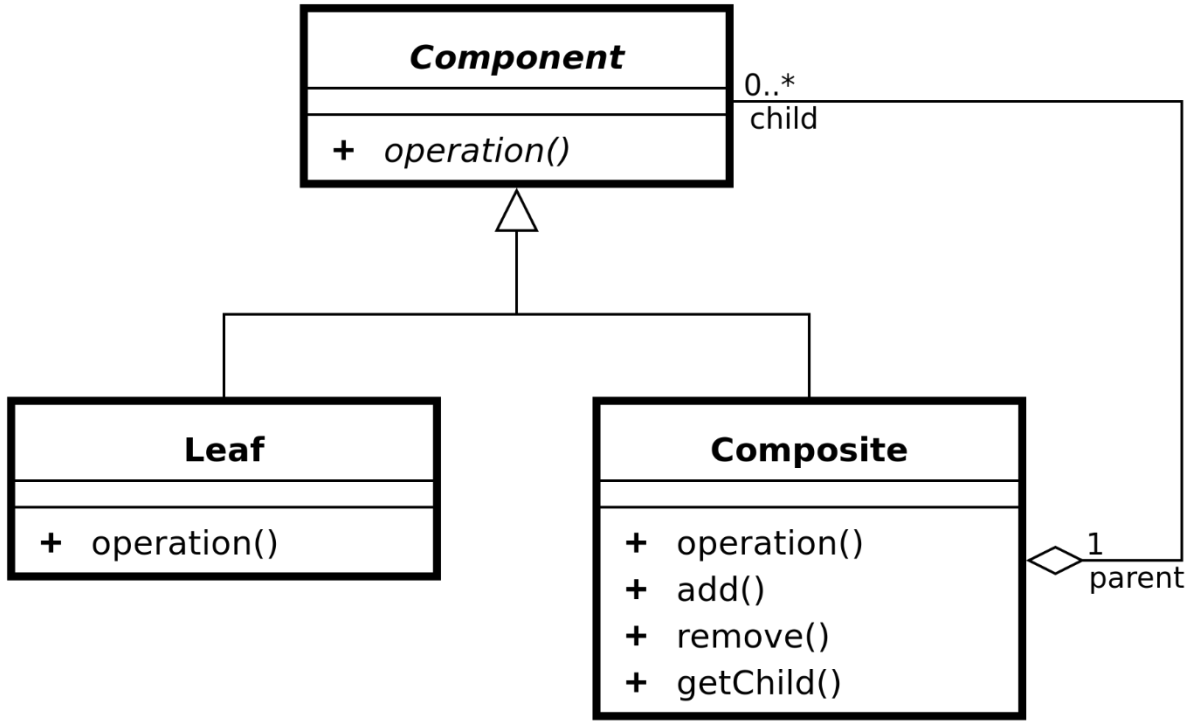
Design Patterns

Design patterns

- **Software design patterns** are ways of designing objects that have been used successfully in the past
  - Think of them as rough blueprints or guidelines
- Design patterns have four essential elements:
  - A meaningful name
  - A description of the problem area that explains when the pattern may be applied
  - A solution description of the parts of the design, their relationships, and their responsibilities
  - A statement of the consequences of using the design pattern
- Patterns are more abstract than code

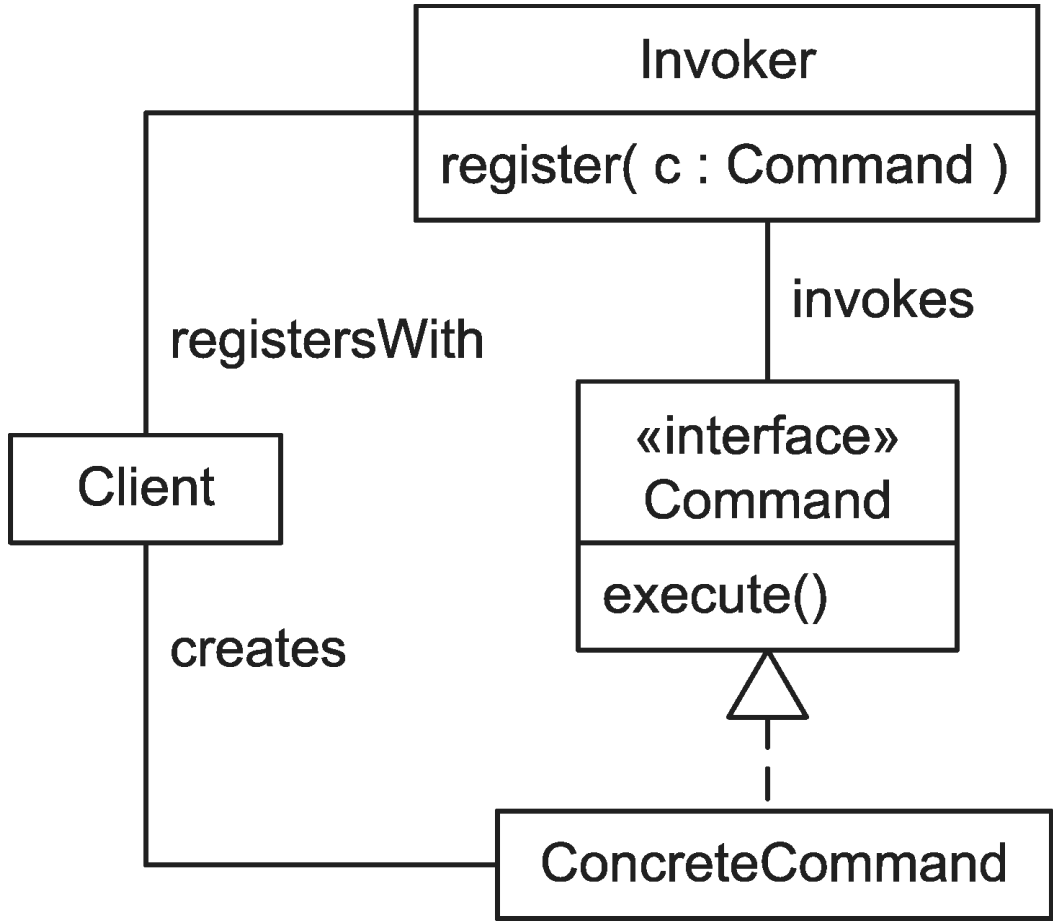
Composite pattern

- The **composite pattern** is useful for part-whole hierarchies of objects
- A group of objects somewhere in the hierarchy can be treated like a single object
- The Swing library uses the composite pattern for its graphical components
- Problems the composite pattern solves:
  - Representing a part-whole hierarchy so that clients can treat parts and wholes the same
  - Representing a part-whole hierarchy as a tree



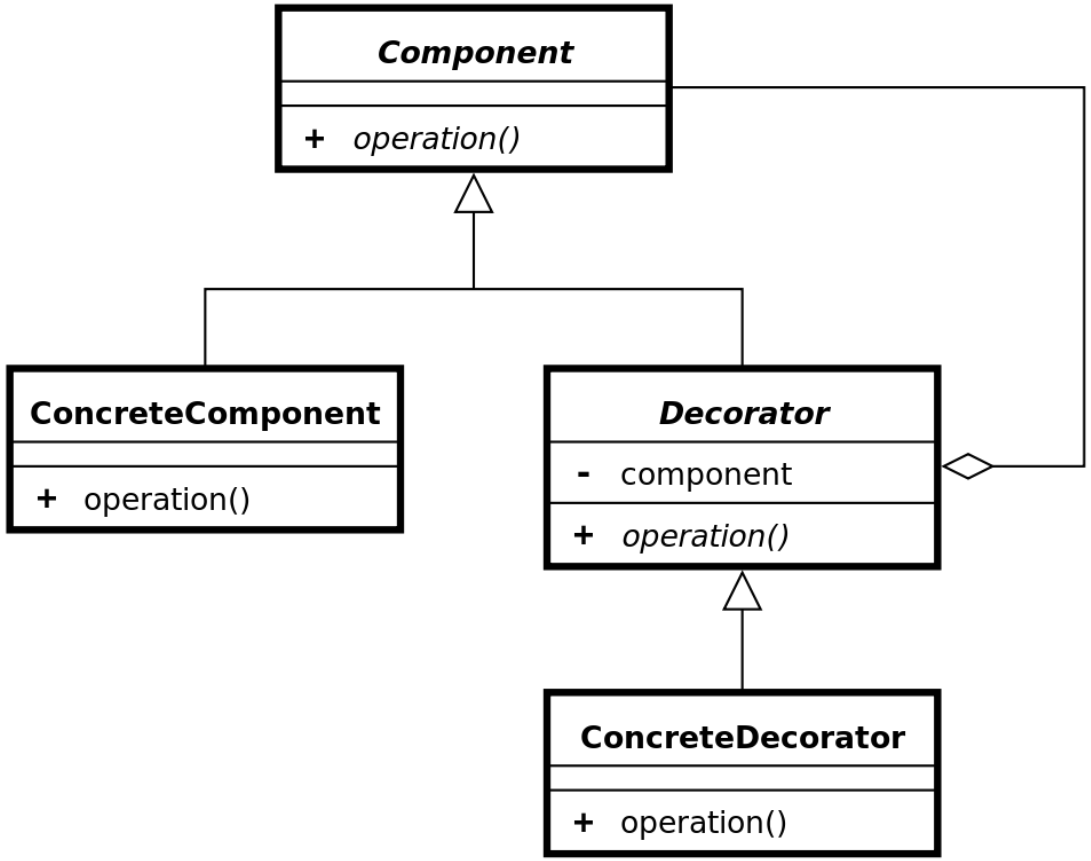
Command pattern

- The **command pattern** is useful for encapsulating an action in an object
- The action is independent from the objects that used it and can be stored for later
- The Swing library uses the command pattern for events
- Problems the command pattern solves:
  - Decoupling the requester from a request



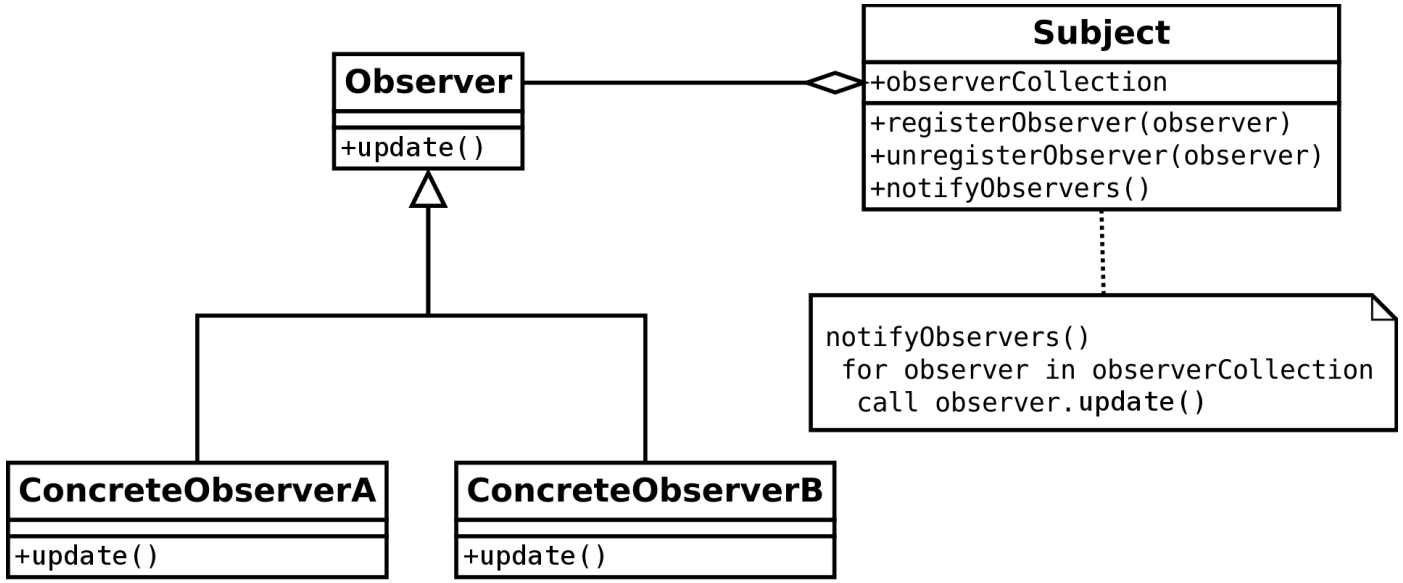
Decorator pattern

- The **decorator pattern** provides a way to add responsibilities to an object dynamically at run-time
- It is commonly used to customize the appearance of GUI elements
- The Swing library uses the decorator pattern to customize borders
- Problems the decorator pattern solves:
  - Adding responsibilities to an object dynamically at run-time
  - Providing a flexible alternative to inheritance for extending functionality



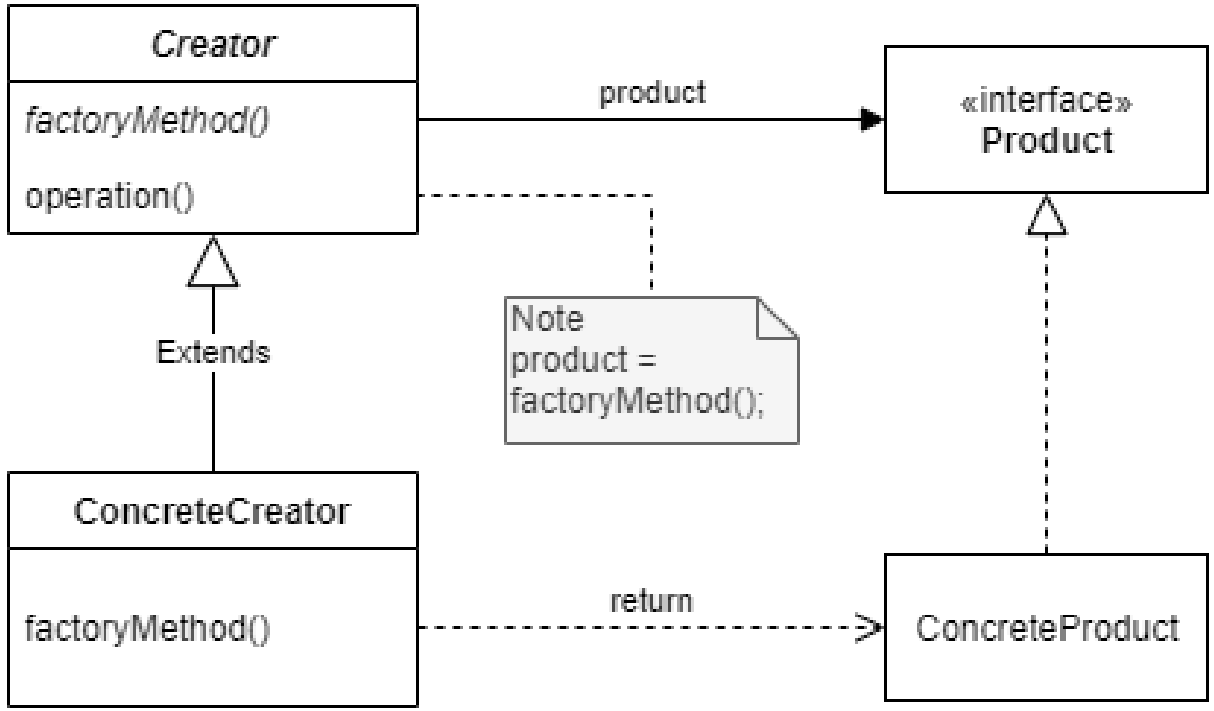
Observer pattern

- The **observer pattern** is useful for a one-to-many dependency where one object changing can update many other objects
- An observer pattern defines Subject and Observer objects
- When a subject changes state, registered observers are updated automatically
- Problems the observer pattern solves:
  - Making a one-to-many dependency between objects without tightly coupling the objects
  - Updating an arbitrarily large number of other objects automatically when one object changes state



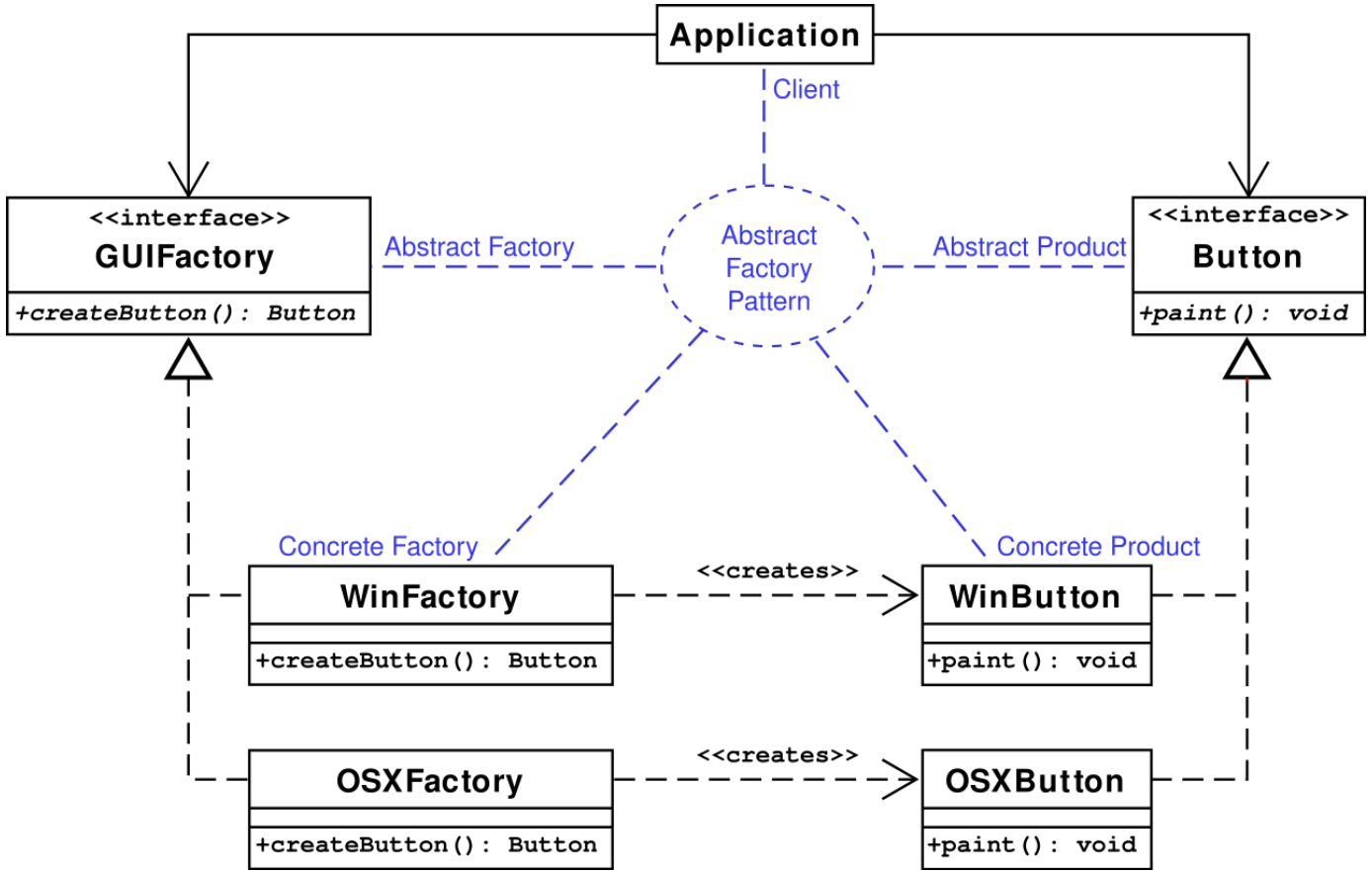
Factory method pattern

- The **factory method design pattern** allows a method to be overridden so that a child class can determine what kind of object to create
- A factory method is defined that is used to create objects
- Problems the factory method pattern solves:
  - Allowing subclasses to define which class to instantiate



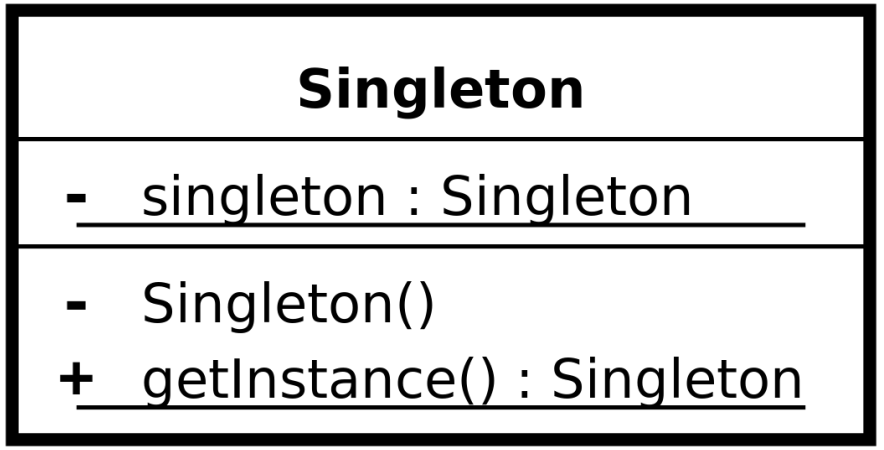
Abstract factory pattern

- The **abstract factory pattern** is similar except that it uses some object as a factory instead of overriding a method
- Problems the abstract factory pattern solves:
  - Making a class be independent of the objects it requires
  - Making a family of related objects



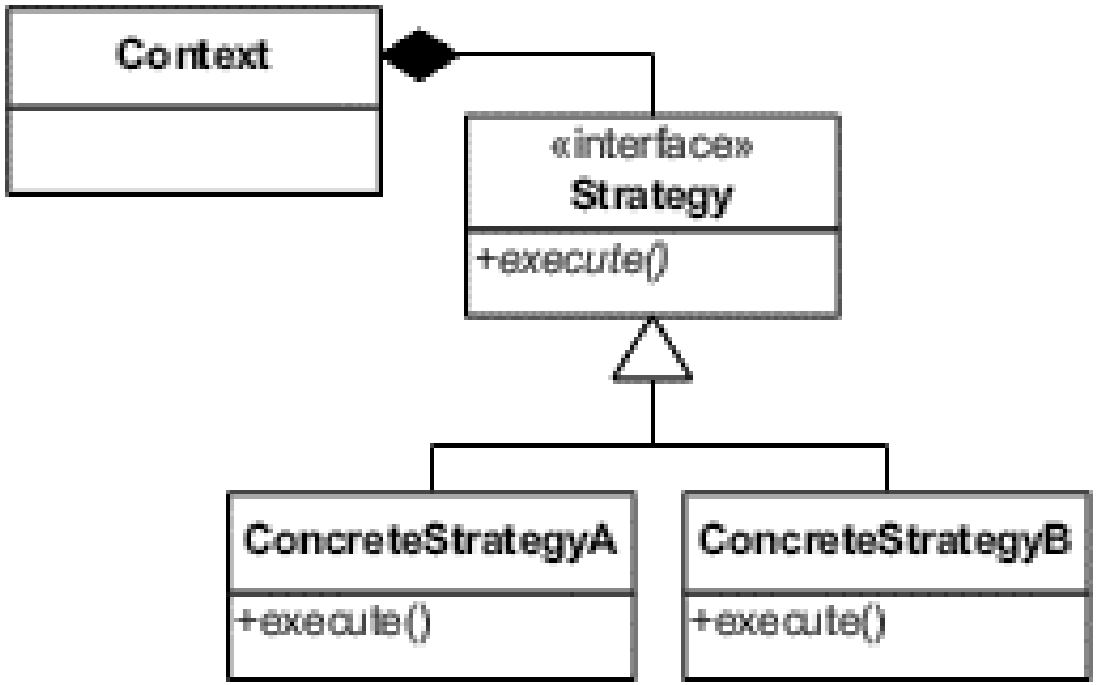
Singleton pattern

- Sometimes it's useful to have only a single instance of a class
- The **singleton pattern** makes it so that it's possible to make only one object of a class and makes it easy to access
- Problems the singleton pattern solves:
  - Ensuring that there's only one instance of a class
  - Making the instance of a class easy to get



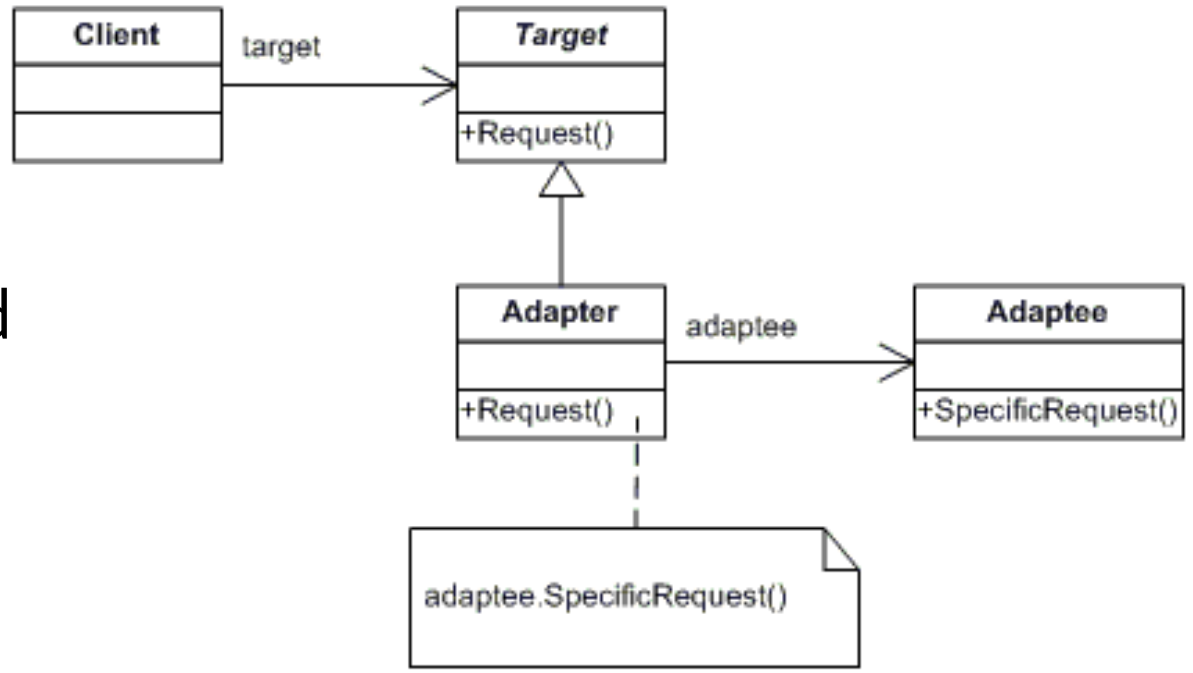
Strategy pattern

- The **strategy pattern** allows an algorithm to be selected at run-time
- In Java, that algorithm is usually encapsulated in the method of an object
- Problems the strategy pattern solves:
  - Configuring a class with an algorithm at run-time
  - Selecting or exchanging an algorithm at run-time



Adapter pattern

- Sometimes you have an object that doesn't generate the right kind of output
- The **adapter pattern** allows you to turn the output from something that gives one kind of output into the kind you need
- Problems the adapter pattern solves:
  - Reusing a class that doesn't have an interface the client requires
  - Allowing classes with incompatible interfaces to work together



Construction Techniques

Bought and customized systems

- It's not always necessary to build a system from scratch
- A **bought and customized** system is one with several bought subsystems that have been customized and integrated into a product that satisfies requirements
- These systems come in a number of overlapping categories:
  - **Commercial off-the-shelf (COTS) systems** are generic products (like SAP, SalesForce, or Blackboard) that need significant customization for a particular client
  - **Component-based systems** are constructed from individual objects that use standard interfaces, like Java Beans and .NET
  - **Service-oriented systems** are like component-based systems except that the connection between components is over the network, and the services are provided by servers

Pros and cons of bought and customized systems

- Pros:
  - Widely used components are usually reliable
  - Good documentation and standards exist for using such components
  - Constructing these systems is usually faster, and costs are easier to predict
- Cons:
  - Increased dependency on outside organizations and their support
  - Lowered flexibility
  - Software engineers have less creative control, potentially reducing job satisfaction (boohoo)

Idioms

- **Idioms** in programming languages are common ways to express ideas
- Example Java idioms:
  - Use **`for`** loops when you want to repeat a specific number of times
  - Use **`while`** loops when you don't know how much you're going to repeat
  - Use a three-line swap to exchange values
- It's a good idea to read code in a language you don't know well to figure out the idioms that people use
- Some people use idioms from languages they know better that can be either inefficient or confusing if they're not used in a different language
- **Syntactic sugar** is a kind of formalized idiom
  - An easy-to-use grammatical structure is converted to a harder-to-read one behind the scenes
  - Example: enhanced **`for`** loops in Java

Programming style

- Each language has stylistic considerations for how to write readable code
  - Many workplaces and open source projects publish style guidelines
- **Naming conventions** cover how to name variables, methods, classes, files, packages, etc.
  - Spelling matters
  - Capitalization is often a matter of convention
  - Being consistent makes everything clearer

Naming

- Most languages encourage either **snake case** or **camel case**
  - Snake case breaks up words with underscores: **`nuclear_silo_radius`**
  - Camel case breaks up words with capitalization: **`nuclearSiloRadius`**
  - Snake case is common in C and Python
  - Camel case is common in Java and C#
  - Very few programming languages allow spaces in variable names
- I prefer variables to be explicit so that it's clear what we're talking about even if we start reading in the middle of unfamiliar code
  - Java tends toward the explicit rather than the abbreviated
- A few other Java naming conventions:
  - Packages are all lowercase
  - Local variables, member variables, and methods start with lowercase letters
  - Classes, enums, and interfaces start with uppercase letters
  - Constants are written in snake case with ALL CAPS

Older naming conventions

- Most languages do not have meaningful limitations on variable name length now, but they used to
- Older C code in particular often leaves out vowels to save space
- Hungarian notations are naming conventions that describe the types of variables with prefixes:
  - **`wParam`** (word-sized parameter)
  - **`pfData`** (pointer to a floating-point value of data)
  - **`lpszName`** (long pointer to a zero-terminated string)
- Hungarian notations can also be used to specify scopes:
  - **`g_nGoats`** (global integer for number of goats)
  - **`m_nBoats`** (member variable integer for number of boats)
- These conventions have largely been given up, since IDEs provide tools for keeping track of types and scopes
  - Also, languages likes Java and C# have much stronger type-safety than C and C++, giving compiler errors for misusing types

Layout conventions

- Many languages (with the notable exception of Python) ignore whitespace
- Thus, we have a choice about how to layout our code
- In C-family, curly brace languages, it's common to put the opening brace of an **`if`** statement, method, or loop either on the same line as the header (K&R style) or on the next line (Allman style)
  - K&R is more common for Java, but Allman is more common for C#
- Some people also have strong feelings that indentation should be tabs while others prefer spaces
- A common convention is that lines of code should not exceed 80 characters

K&R style

```
if (raining) {
    System.out.println("I'm wet!");
}
```

Allman style

```
if (raining)
{
    System.out.println("I'm wet!");
}
```

Commenting

- Almost every language allows for comments
- Code that is so easy to understand that it needs no comments is called **self-documenting code**
  - Ideally, all code is self-documenting, but this goal is rarely reached
- Perhaps the other end of the spectrum is **literate programming**, which explains everything in English mixed in with the code, taking the perspective that code is for humans to understand and only incidentally for computers to execute
- Commenting should explain confusing code, especially unusual algorithms

Good commenting

- **Do** use comments to describe the intent of a complicated piece of code
- **Do** use comments to explain the rationale behind a decision so that people can understand in the future
  - Why this way?
  - Why not that other way?
- **Do** use comments to reference relevant outside documents
  - Explanation of an algorithm
  - API documentation page
  - Design document with UML diagrams

Questionable commenting

- **Don't** use comments to repeat the code
- Be careful about using comments for to-do items and future work
  - Especially if it means you don't do the right thing now
- It is possible to over-comment, so consider whether the supplemental information is useful

Bad comments that repeat the code

```
// Increase i by 1
++i;

// Include sales[i] in the total
total = total + sales[i];
```

Data organization

- Programs often include data, but how should it be organized?
- Data structures store the data in the program, but the data also needs to be stored between program runs or sent to someone else to use
  - Internal data vs. external data
- Common data organization approaches
  - Markup languages
  - Databases

Version control

- We already know the value of a **version control system (VCS)**
- Some details:
  - A VCS stores **items** (usually files)
  - A **version** is the set of items after one or more modifications
  - A **revision** is a version stored in a VCS
  - A **baseline** is the first revision
  - Storage for revisions is called a **repository**
  - Storing a version in the repository is called **checking in** or **committing**
  - Retrieving a version from the repository is called **checking out** or **updating**
  - A checked-out version of an item is a **working copy**

VCS choices

- How do we deal with two or more different people working on the same file and trying to commit them to the same repository?
  - **File locking:** When a files are checked out for modification, they are locked, meaning that no one else can check them out for modification
  - **Concurrent modification and merge:** If someone tries to commit a file based on an older version of the file, the commit fails, forcing the person to merge the newer repository file with the file they're working on
- Before you start modifying a file, it's wise to pull down the latest changes first
- A centralized VCS has one central repository
- A distributed VCS has many repositories that are peers

Quality Assurance in Construction

Static analysis and dynamic analysis

- **Static analysis** is looking at code without running it
  - Code reviews
  - Syntax checking
  - Style checking
  - Usage checking
  - Model checking
  - Data flow analysis
  - Symbolic evaluation
- **Dynamic analysis** is running code to test it
  - Unit testing
  - Debugging
  - Performance optimization and tuning
- Both static and dynamic analysis are valuable and have different strengths
  - Static analysis doesn't require a fully working program
  - Dynamic analysis can give real data about things like performance

Code reviews

- Desk checking is one form of code review
  - Looking over the code
  - Executing it by hand (actually computing values)
- Formal inspections (discussed earlier) are another
- Formal review guidelines
  - Don't read more than 200 lines of code per hour when preparing alone
  - Don't cover more than 150 lines of code when doing a team inspection
  - Use a checklist
- Examples from a Java inspection checklist
  - All variables and constants are named in accord with naming conventions
  - There are no variables or attributes with confusingly similar names
  - Every variable and attribute has the correct data type
  - Every method returns the correct value at every return point
  - All methods and attributes have appropriate access modifiers (**`private`**, **`protected`**, or **`public`**)
  - No nested **`if`** statements should be converted into a **`switch`** statement
  - All exceptions are handled appropriately

Syntax and style checking

- Syntax checking is now mostly done by editors and IDEs
- Be careful about the errors and warnings IDEs and compilers given
  - As computers, they can only guess about why the syntax is wrong
- Language-specific style guides are required on most projects
- Automated style checkers also exist
  - In addition to formatting, they can check semantic issues like variables that are declared and not used
  - Some features like this are included in modern compilers as warnings

Usage checking and idiom checking

- For broader semantic issues, usage and idiom checkers (which can be combined with a style checker) look for:
  - Suspicious or error-prone constructs
  - Non-portable constructs
  - Memory allocation inconsistencies
  - Language-specific issues
    - Loops that never execute
    - Loops that never terminate
    - Using types together that are legal but unusual

Formal methods

- **Formal methods** use mathematical models to do static analysis
- **Model checking** uses analysis to determine if a program meets requirements, usually if certain preconditions are met, it's guaranteed that certain postconditions will be met
- **Data flow analysis** represents a program as a graph and uses that knowledge to calculate the possible values at various points in the graph
  - Modern languages like Java use data flow analysis to complain, for example, that a variable might not have been initialized
- **Symbolic evaluation** traces through the execution of a program with symbolic values instead of concrete values

Unit Testing

Unit testing

- Testing is an important form of dynamic analysis
- **Unit testing** is testing individual units or sub-programs (classes or methods in Java) in isolation
- A **test case** has one value for every input and an expected value for every output
- A **false negative** happens when there's a problem with your code but you don't write a test that catches it
  - This almost always happens, since it's very hard to test everything
- A **false positive** happens when your code is fine but your test is bad
  - For example, you did the math wrong when coming up with your expected answer

Developing test cases

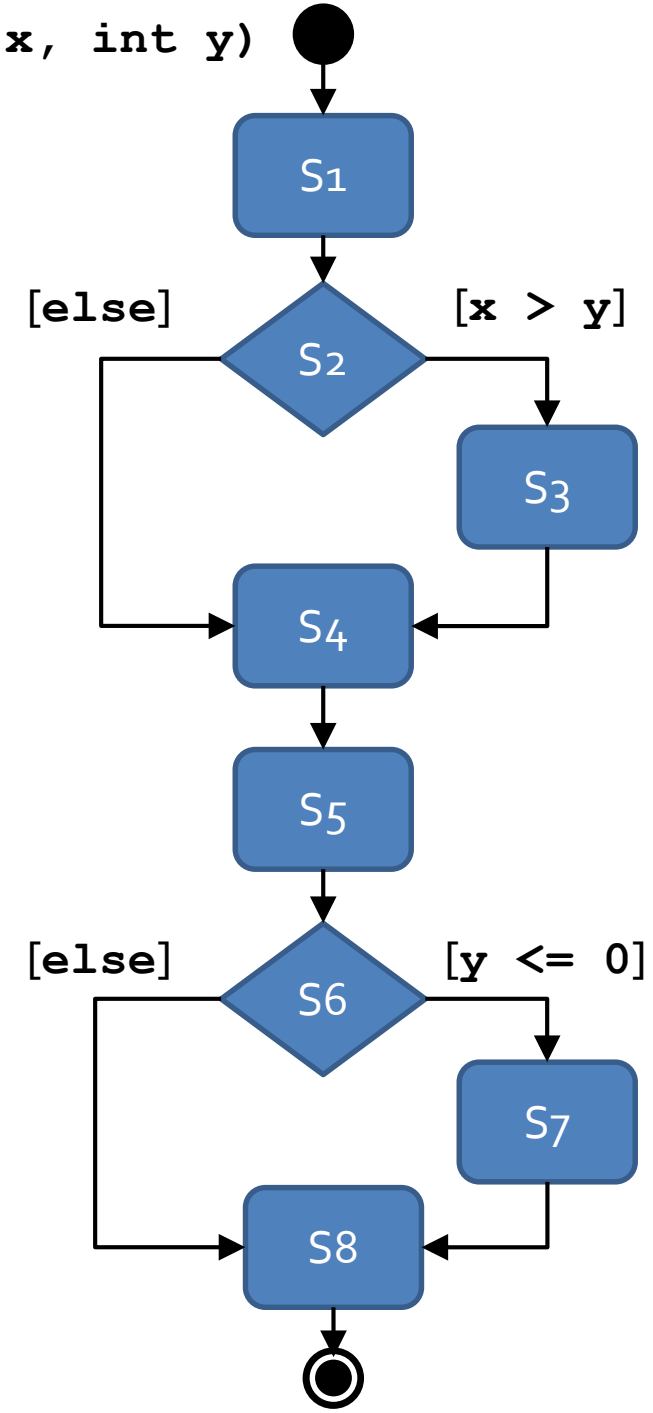
- Picking good test cases is an art form
- **Black box testing** is a strategy that assumes no knowledge of what happens inside the system
  - Only what the input and matching output should be are known
  - Black box testing is easily done by someone who had nothing to do with developing the code
  - Black box testing isn't affected by assumptions about how an algorithm should work
- **Clear box** (or white box or open box) **testing** uses knowledge of the system to generate good tests
- Both kinds of testing are needed to be thorough

Code coverage

- Clear box testing is built around the idea of **coverage**, which is *how much* of the unit is tested
- Coverage can be explore with a **control-flow graph (CFG)** that shows the possible paths execution could take in a program
  - An **action node** in a CFG is straight-line code with one entry point and one exit point
  - A **decision node** in a CFG is code like an **`if`** statement or a loop with multiple exit points
  - Arrows show the flow of execution through nodes

Example CFG

```
int calculate(int x, int y)
{
      int a, b;
      a = 1;                  // S1
      if (x > y)              // S2
      {
            a = 2;            // S3
      }
      x++;                    // S4
      b = y * a;              // S5
      if (y <= 0)             // S6
      {
            b++;              // S7
      }
      return b;               // S8
}
```



Kinds of coverage

- We say a statement is **exercised** by a test or a suite of tests if it gets executed
- **Statement coverage** is the percentage of statements exercised by a set of tests
  - Example: (`x = 1, y = 2`) exercises everything except S3 and S7 in the previous CFG, giving a statement coverage of 75%
- **Branch coverage** is the percentage of branch directions taken by a set of tests
  - Example: (`x = 1, y = 2`) covers the else edge from S2 and the else edge from S6, giving a branch coverage of 50%
- **Path coverage** is the percentage of all execution paths that have been taken
  - Example: (`x = 1, y = 2`) takes only one of the four paths from S1 to S8, giving a path coverage of 25%
- More coverage is better
- It will usually take many tests to get good coverage

Boundary value analysis

- **Boundary value analysis** uses values near the edges of legal limits
  - If input must be within a range, create tests just below, at, and just above the endpoints of the range
  - If output must be in a certain range, try to pick inputs that generate values around the minimum and maximum of that range
- Example: Boundary values for a method that's supposed to accept passwords if they're between 6 and 12 characters inclusive

Input	Length	Case	Valid
"goats"	5	Mininum – 1	False
"wombat"	6	Minimum	True
"wombats"	7	Minimum + 1	True
"abracadabra"	11	Maximum – 1	True
"hippopotamus"	12	Maximum	True
"administrator"	13	Maximum + 1	False

Other heuristics

- A number of other heuristics are commonly used because they often find errors
- For single input parameters
  - 0 (because people forget about 0 or because of division by 0)
  - Very large and very small numbers (because of underflow and overflow)
  - Character or string versions of numbers (which makes sense in a language like Python or JavaScript but not in Java where type checkers would prevent such thins)
- For multiple input parameters
  - Equal values for the parameters
  - Different relative values (*x* larger than *y*, then *x* smaller than *y*)
- For arrays and collections
  - Very small and very large arrays and collections
  - Arrays or collections of length 0 and 1
  - Arrays or collections that are unsorted, ascending, and descending
  - Arrays or collections with duplicated values and with no duplicated values

Regression testing

- Something's wrong with your program, so you change your code, what happens?

	No New Fault Introduced	New Fault Introduced
Fault Corrected	Good	Bad
Fault Not Corrected	Bad	Very Bad

- Data suggests that
  - 30% of software changes result in one of the three bad outcomes
  - On average, bad outcomes occur about 10% of the time
  - Faults introduced during bug fixes are harder to find and remove than others
- One safeguard is **regression testing**, running *all* tests after any software change
  - Any time you find a bug, add the test you used to find the bug into your test suite

Unit testing tools

- Nowadays, running large test suites can be automated
- Tools such as JUnit and other testing tools allow us to:
  - Write clearly marked tests with special set-up and clean-up code if needed
  - Run the tests, sometimes with randomized values or in randomized orders
  - Record which tests pass and fail
  - Show coverage information to see which lines of code the tests covered

JUnit

- JUnit is a popular framework for automating the unit testing of Java code
- JUnit is built into Eclipse and many other IDEs
- It is possible to run JUnit from the command line after downloading appropriate libraries
- JUnit is one of many xUnit frameworks designed to automate unit testing for many languages
- You are required to make JUnit tests for Project 3
- JUnit 5 is the latest version of JUnit, and there are small differences from previous versions

JUnit classes

- For each set of tests, create a class
- Code that must be done ahead of every test has the **`@BeforeEach`** annotation
- Each method that does a test has the **`@Test`** annotation

```
import org.junit.jupiter.api.*;
public class Testing {

        private String creature;

        @BeforeEach
        public void setUp() {
                creature = "Wombat";
        }

        @Test
        public void testWombat() {
                Assertions.assertEquals("Wombat", creature, "Wombat failure");
        }
}
```

Assertions

- An assertion is something that **must** be true in a program
- Java (4 and higher) has assertions built in
- You can put the following in code somewhere:

```
String word = "phlegmatic";
assert word.length() < 5 : "Word is too long!";
```

- If the condition before the colon is true, everything is fine
- If the condition is false, an **`AssertionError`** will be thrown with the message after the colon
- Caveat: The JVM normally runs with assertions turned off, for performance reasons
- You have to run it with assertions on for assertion errors to happen
- You **should** run the JVM with assertions on for testing purposes

Assertions in JUnit tests

- When you run a test, you expect to get a certain output
- You should assert that this output is what it should be
- JUnit 5 has a class called **`Assertions`** that has a number of static methods used to assert that different things are what they should be
  - Running JUnit takes care of turning assertions on
- The most common is **`assertEquals()`**, which takes the expected value, the actual value, and a message to report if they aren't equal:
  - **`assertEquals(int expected, int actual, String message)`**
  - **`assertEquals(char expected, char actual, String message)`**
  - **`assertEquals(double expected, double actual, double delta, String message)`**
  - **`assertEquals(Object expected, Object actual, String message)`**
- Another useful method in **`Assertions`**:
  - **`assertTrue(boolean condition, String message)`**

Assertion example

- We know that the **`substring()`** method on **`String`** objects works, but what if we wanted to test it?

```
import org.junit.jupiter.api.*;

public class StringTest {

	@Test
	public void testSubstring() {
		String string = "dysfunctional";
		String substring = string.substring(3,6);
		Assertions.assertEquals("fun", substring, "Substring failure!");
	}
}
```

Sometimes failing is winning

- What if a method is **supposed** to throw an exception under certain conditions?
- It should be considered a failure **not** to throw an exception
- The **`Assertions`** class also has a **`fail()`** method that should never be called

```
import org.junit.jupiter.api.*;

public class FailTest {
	@Test
	public void testBadString() {
		String string = "armpit";
		try {
			int number = Integer.parseInt(string);
			Assertions.fail("An exception should have been thrown!");
		}
		catch(NumberFormatException e) {}
	}
}
```

Debugging

Debugging

- **Debugging** is using trigger conditions to identify and correct faults
- Steps of debugging
  1. **Stabilize:** Understand the symptom and trigger condition so that the failure can be reproduced
  2. **Localize:** Locate the fault
     - Examine sections of code that are likely to be influenced by the trigger
     - Hypothesize what the fault is
     - Instrument sections of code (with print statements or conditional breaks)
     - Execute the code, monitoring the instrumentation
     - Prove or disprove the hypothesis
  3. **Correct:** Fix the fault
  4. **Verify:** Test the fix and run regression tests
  5. **Globalize:** Look for similar defects in the rest of the system and fix them

Debug code

- **Debug code** is temporary output and input used to monitor what's going on in the code
- Instead of printing out just numbers, add context information so that the debug statements are clear
- Debug code is quick and dirty, useful when setting break points and tracing execution with a debugger might be too much work to catch a small issue
- There are logging tools that can print logging data at various levels
  - Normally, nothing prints out
  - Running the program in logging mode prints out important data
  - Running the program in verbose mode prints out everything it can
- Debug output can go to **`stdout`** or **`stderr`**
  - **`System.err`** (instead of **`System.out`**) prints to stderr in Java

Debuggers

- IntelliJ, Eclipse, Visual Studio, gdb and most fully-featured IDEs provide debugging tools
- Typical debugging features:
  - Setting **breakpoints** that will pause execution of the program when reached
    - Breakpoints can often be **conditional**, pausing only if certain conditions are met
  - Executing lines of code one by one, **stepping over** method calls or **stepping into** them and **stepping out** when you're done executing its code
  - Setting **watches** that display the current state of variables and members
- If you don't use your debugger, you're choosing to play the game with one hand tied behind your back

Refactoring

- **Refactoring** means changing working code into working code
- It can be done to improve the structure, the presentation, or the performance
- You should refactor when:
  - There's duplication in your code
  - Your code is unclear
  - Your code smells:
    - Comments duplicate code
    - Classes only hold data (instead of operating on it)
    - Information isn't hidden
    - Classes are tightly coupled
    - Classes have low cohesion
    - Classes are too large
    - Classes are too small
    - Methods are too long
    - **`switch`** statements are used instead of good object-orientation

Common refactoring actions

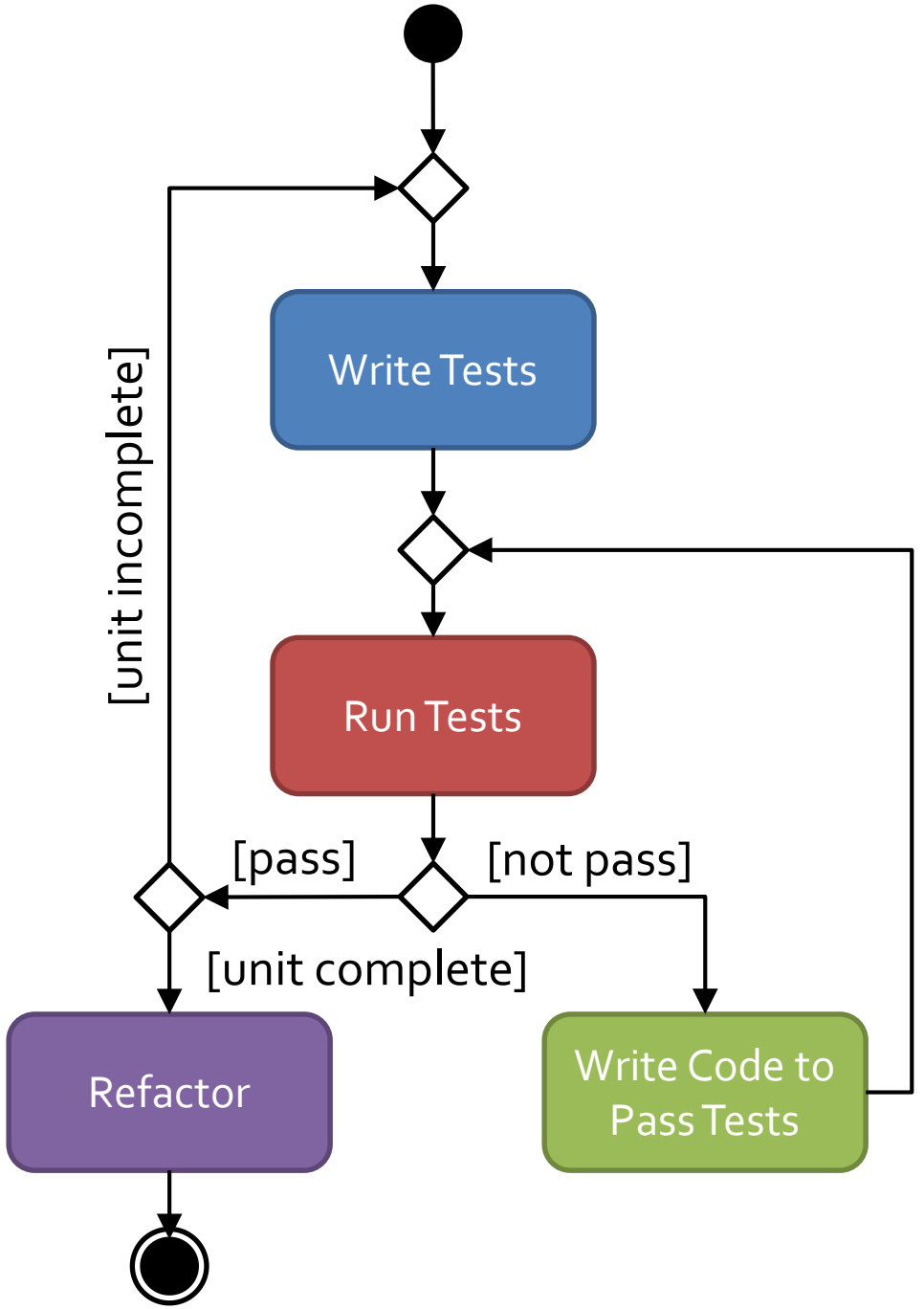
- Renaming a variable or method
- Adding an explanatory variable
  - If an expression is too long, storing a partial computation into a named variable can help it be understood
- Inline temporary variable
  - If a temporary variable is useless, just use the full expression (the opposite of the previous)
- Break a method into two methods
- Combine two short methods into a single one
- Replace a conditional with polymorphism
  - Instead of an if or a switch, behavior changes because different objects have overridden methods with different behavior
- Move methods from child classes to parent classes

Test driven development

- **Test driven development** (TDD) is a style of development where testing is an integral part of coding
- The key idea of TDD is that you write tests for the code **before** you write the code
  - Thus, the tests aren't distorted by writing the code
- TDD is used for Extreme Programming, but it can be used for any approach, agile or plan-driven

Principles of TDD

- You have to have a testing framework
- Tests are written before code
- Tests and code are written incrementally
  - Write tests for some functionality, then write code to pass them
- Code is *only* written to pass tests
  - "Doing the simplest thing that could possibly work"
- Refactoring is expected
  - Writing code only to pass tests might end up with funky design



Benefits of TDD

- By making the test first, you really understand what you're trying to implement
- Your testing has better code coverage, testing every segment of code at least once
- Regression testing happens naturally
- Debugging should be easier since you know where the problem likely is (the new code added)
- The tests are a form of documentation, showing what the code should and shouldn't do

System Testing

System testing

- **System testing** is testing of the whole product
  - Both unit testing and integration testing of individual classes and larger components should have been done by now
  - Testing both functional and non-functional requirements
- System testing is necessary because:
  - There could still be faults in the components
  - Some things can't be fully tested without all the pieces together
- **Alpha testing** is the first stage of system testing
  - Developers test behavior similar to what real users would do
- **Beta testing** has real users testing the product

Details of system testing

- Alpha testing and the two phases of beta testing are similar, but there are some details that are different, summarized in this table

	Alpha Testing	Beta Testing	
		Acceptance Testing	Installation Testing
Personnel	Testers	Users	Users
Environment	Controlled	Controlled	Uncontrolled
Purpose	Validation (Indirect) and Verification	Validation (Direct)	Verification
Recording	Extensive Logging	Limited Logging	Limited Logging

Functional alpha testing

- Functional alpha testing is based on the requirements listed in the product specification
- To isolate failures, basic functionality is tested before more complex functionality
- **Operational profiles** give information about how often different use cases come up and the typical order of use cases
  - Using these profiles, testers can make tests that simulate typical usage

Non-functional alpha testing

- Some non-functional requirements are development requirements
  - Cost of the product
  - Time the product takes to be made
- Development requirements generally can't be tested, but there are many kinds of non-functional execution requirements that are testable
- Common non-functional execution tests:
  - **Timing tests** time the amount of time needed to perform a function, sometimes using **benchmarks**, standard timing tests
  - **Reliability tests** try to determine the probability that a product will fail within a time interval: mean time to failure
  - **Availability tests** try to determine that probability that a product will be available within a time interval: percent up time
  - **Stress tests** try to determine **robustness** (operating under a wide range of conditions) and **safety** (minimizing the damage from a failure)
  - **Configuration tests** check the product on different hardware and software platforms

User interface tests

- Some user interface tests straddle the line between functional and non-functional
- Tests that check the user interface are called **usability tests** or **human factors tests**
- **Internationalization or localization tests** are a kind of usability test that check translations and other cultural information like currencies and the formatting of numbers, times, and dates
- **Accessibility tests** check whether the user interface works for all people, even with significant disabilities
  - There are guidelines for the kinds of disabilities that need support (low visual acuity or color blindness)
  - Testing often involves measuring the time needed to perform tasks

Beta testing

- Beta testing uses external testers, usually users from the population who will use your product
- These users have the duty to record and report failures
- Acceptance testing is a kind of beta testing done by clients to validate that the product meets their needs
  - Done in a controlled environment, like the one alpha testing was done in
- Installation testing is a kind of beta testing using real users in uncontrolled environments
  - Instead of validation, the goal is to verify that the product works properly in a (more) real environment
  - Installation testing can be inefficient, since the users often do not give the most detailed feedback

Upcoming

Next time…

- **Exam 2 on Wednesday!**
- Work day on Friday

Reminders

- Work on Project 3
- **Study for Exam 2**
  - **In class on Wednesday!**