Week 15 - Wednesday

COMP 3100

Last time

- What did we talk about last time?
- Review up to Exam 1 and halfway to Exam 2

Questions?

Final exam format

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

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

Review

Software Engineering Design

Simplicity

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

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

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

Small modules

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

Information hiding

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

Minimize module coupling

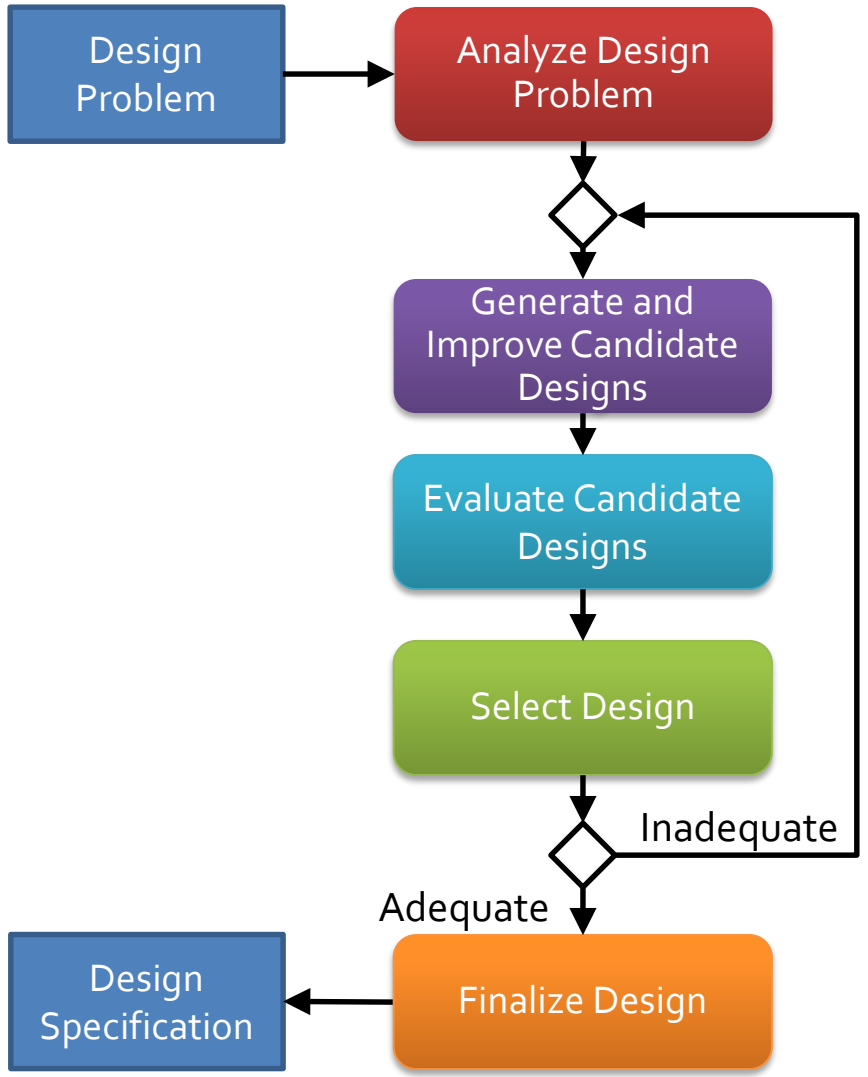
- **Module coupling** is the amount of connectivity between two modules
- Modules can be coupled in the following ways:
  - One class is an ancestor of another class
  - One class has a member whose type is another class
  - One class has an operation (method) parameter whose type is another class
 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

Maximize module cohesion

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

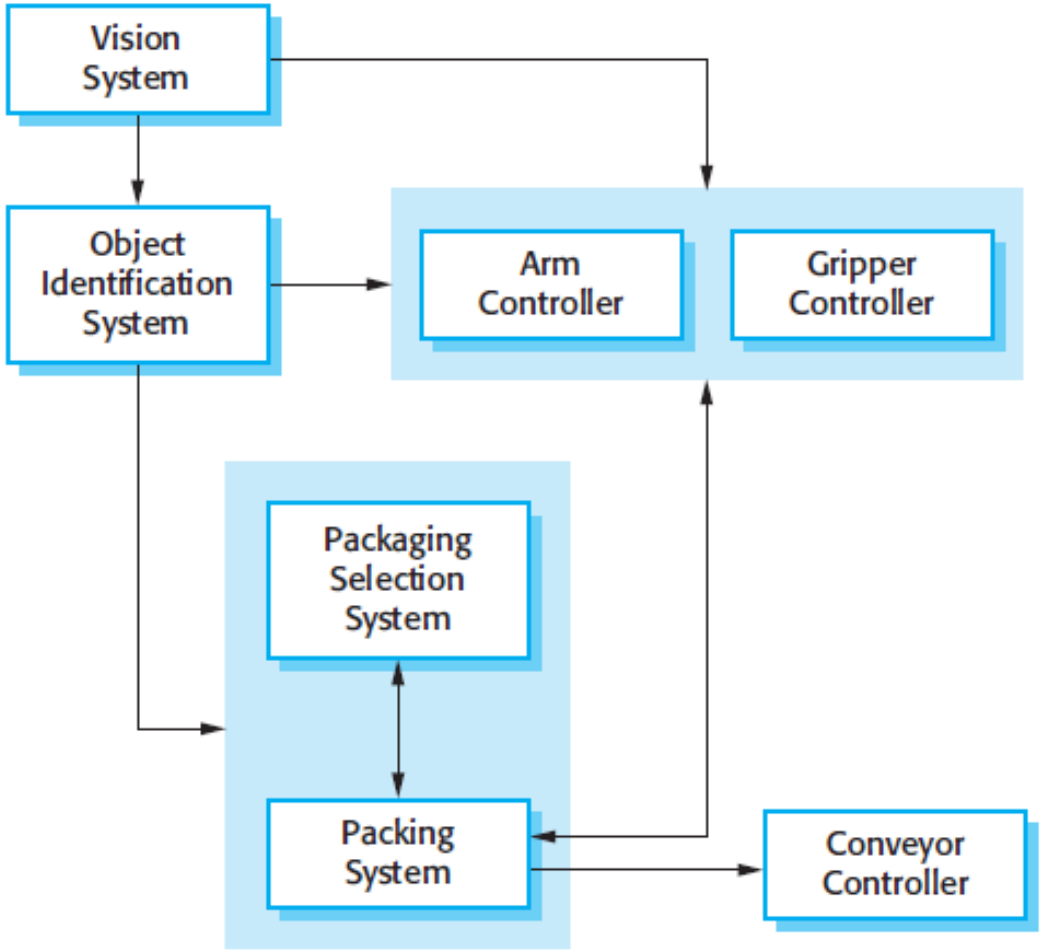
Design process

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



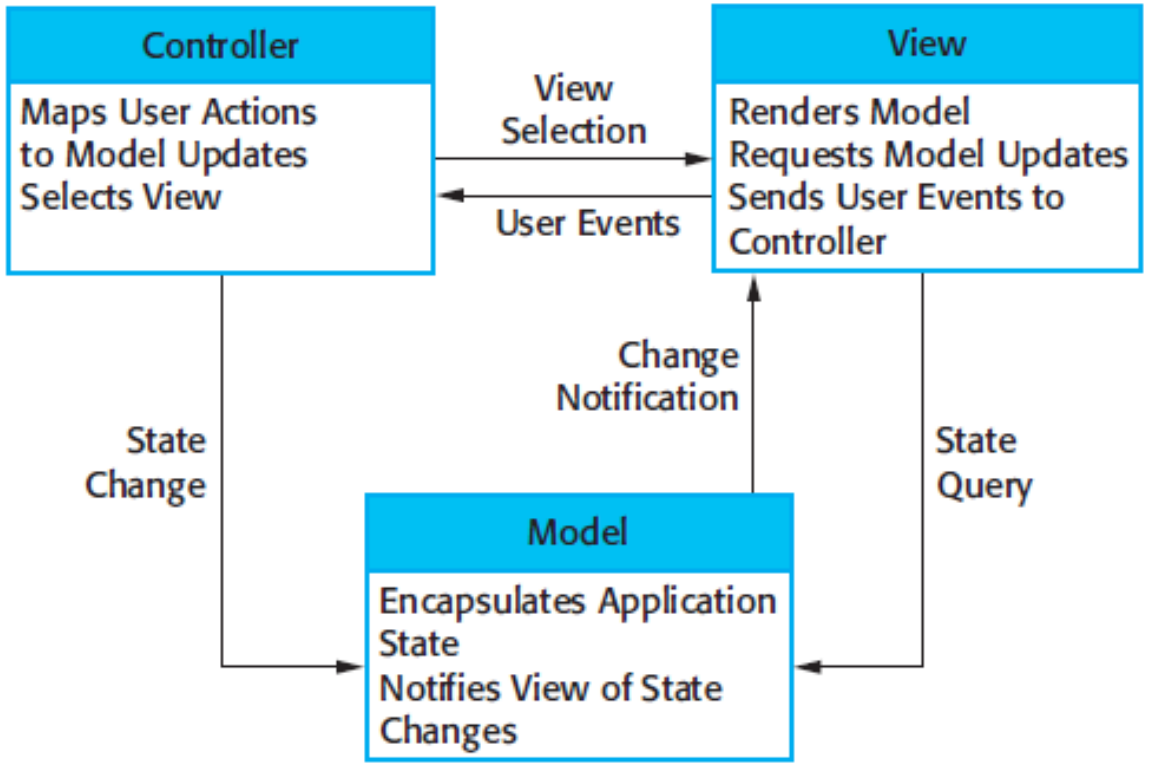
Architectural design

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



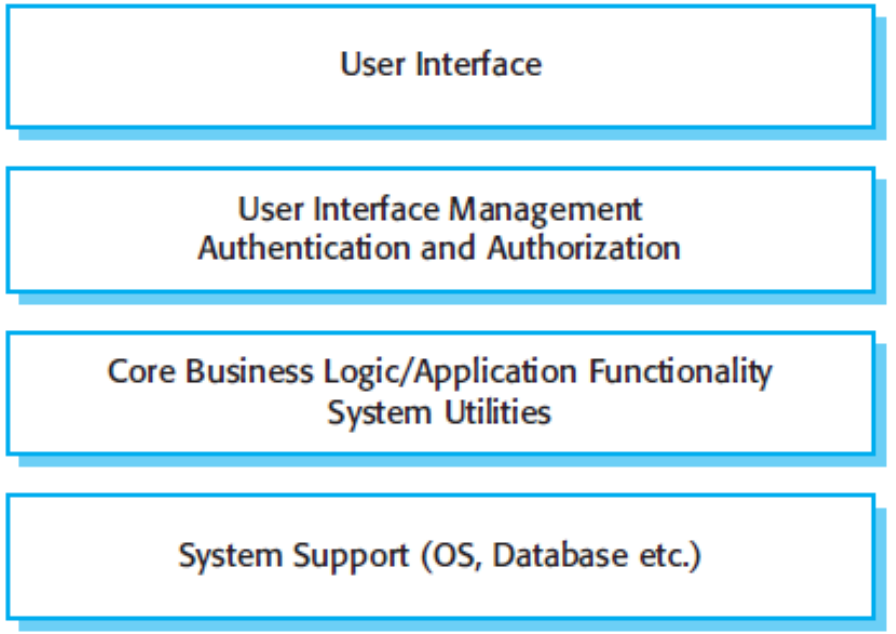
Model-View-Controller

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



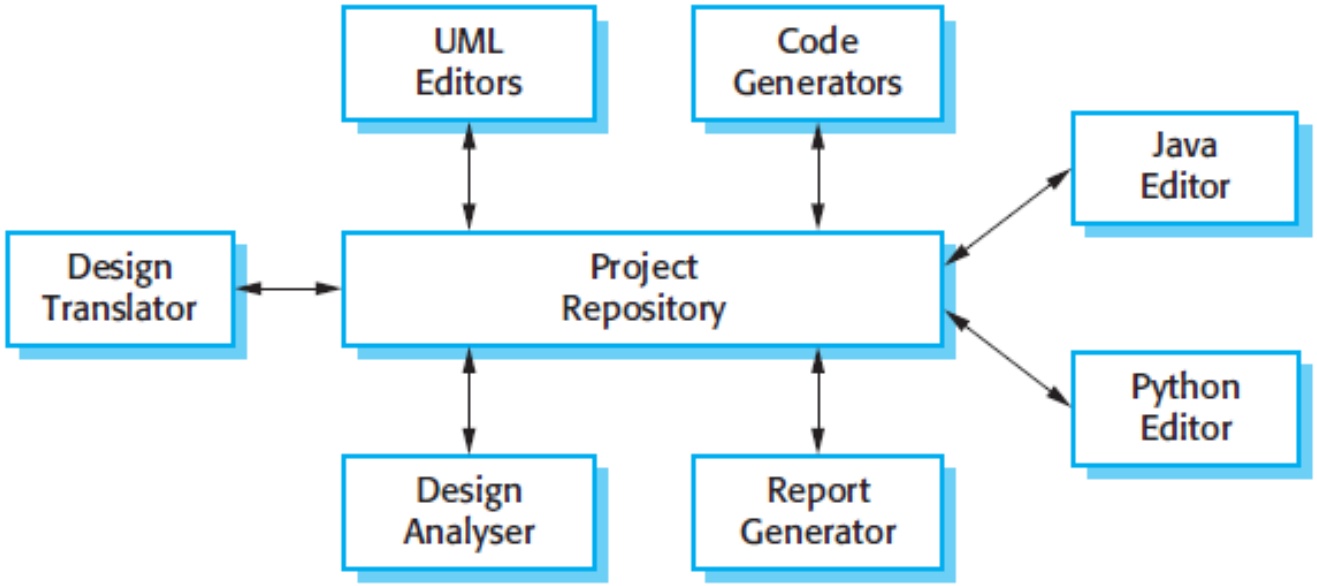
Layered style

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



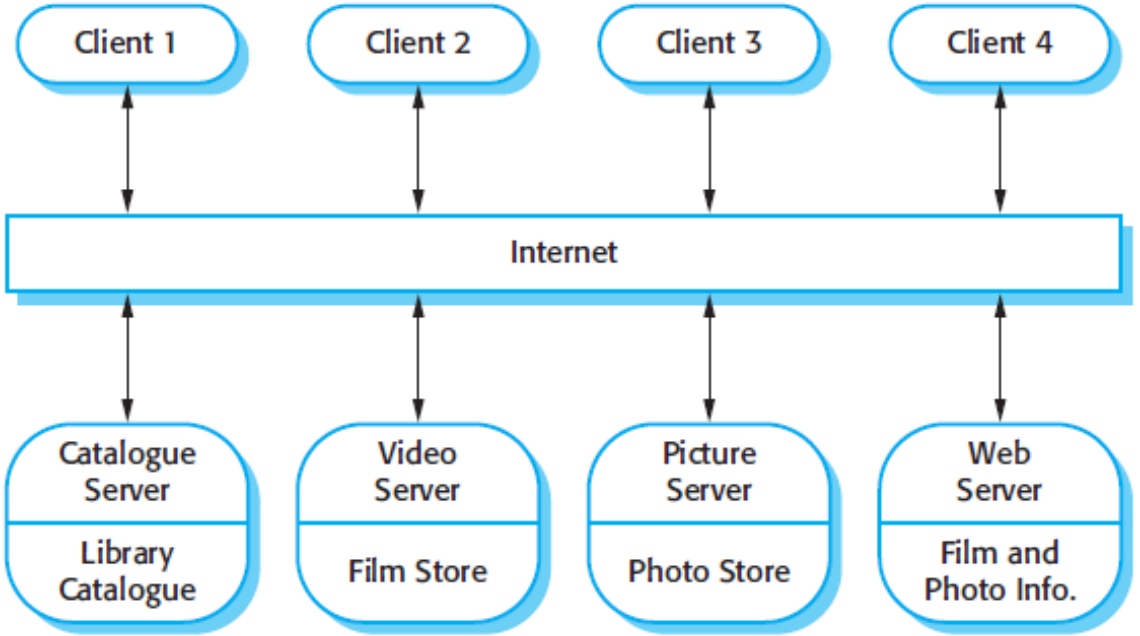
Repository style

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



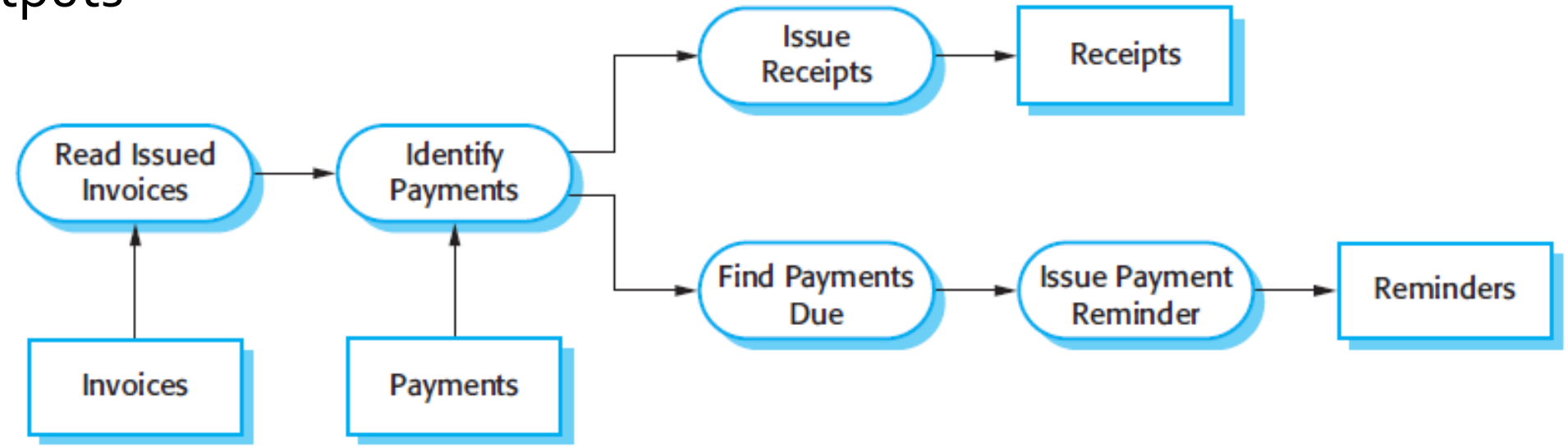
Client-server architecture

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



Pipe and filter style

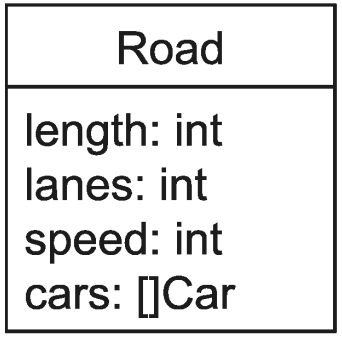
- In the pipe and filter style, data is passed from one component to the next
- Each component transforms input into output
- Good: easy to understand, matches business applications, and allows for component reuse
- Bad: each component has to agree on formatting with its inputs and outputs

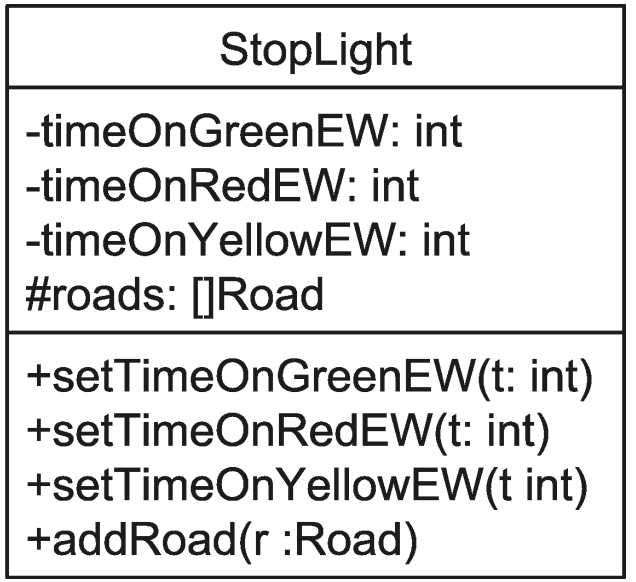


Detailed Design

More depth on class diagrams

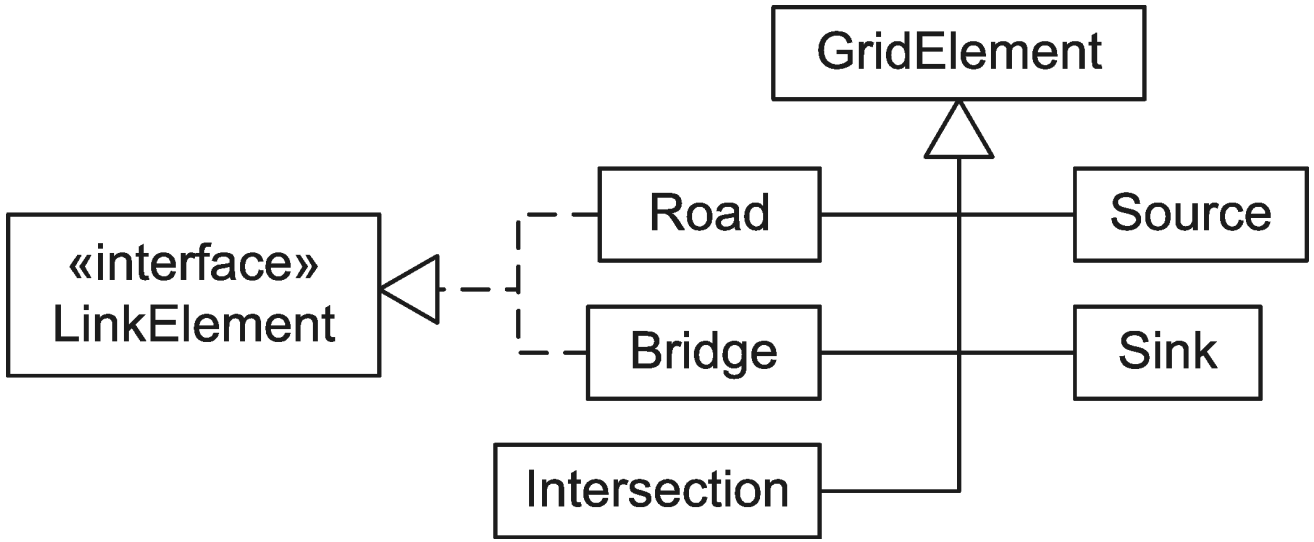
- **Class diagrams** are made up of **class symbols** (rectangles)
- These class symbols contain one or more **compartments**
- The top compartment has the class name
- A second, optional compartment often contains attributes (called member variables in Java classes)
  - Often followed by a colon with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





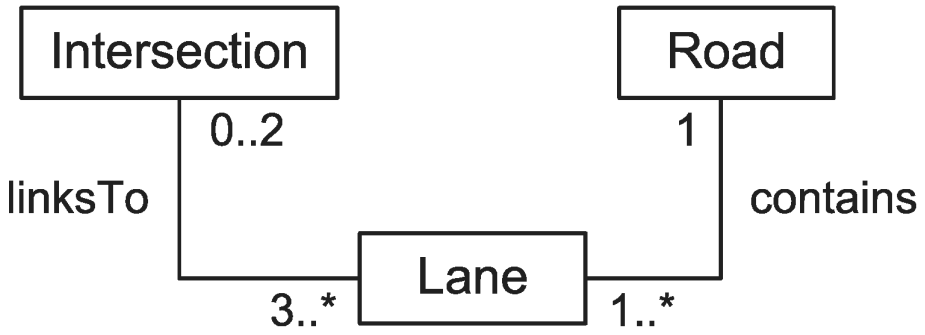
Inheritance and interfaces in class diagrams

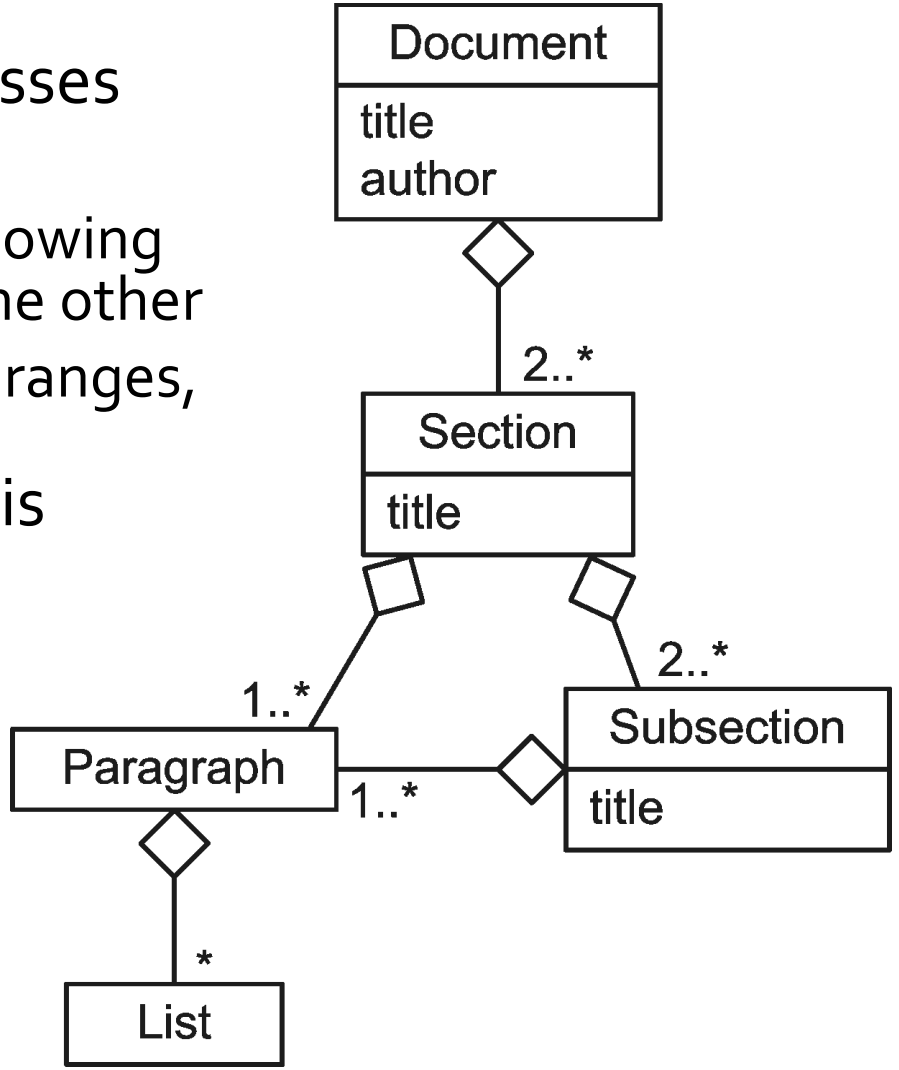
- Inheritance is shown with the **generalization** connector
  - A solid line from the child class to a solid triangle connected to the parent class
  - Confusingly, this means that children classes point at their parent classes
- Interfaces look like classes but are marked with **«interface»** above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



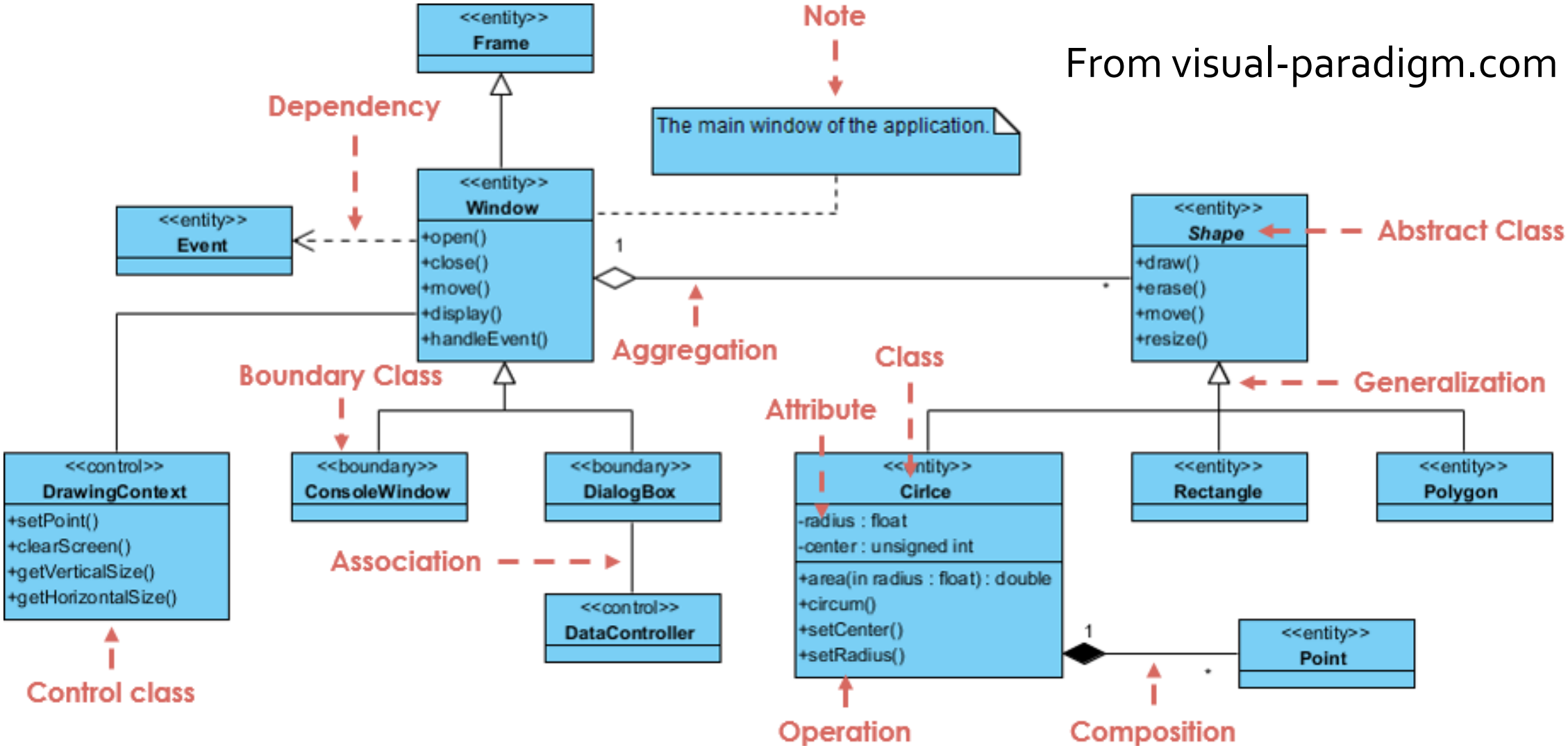
Other associations

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





Complex example



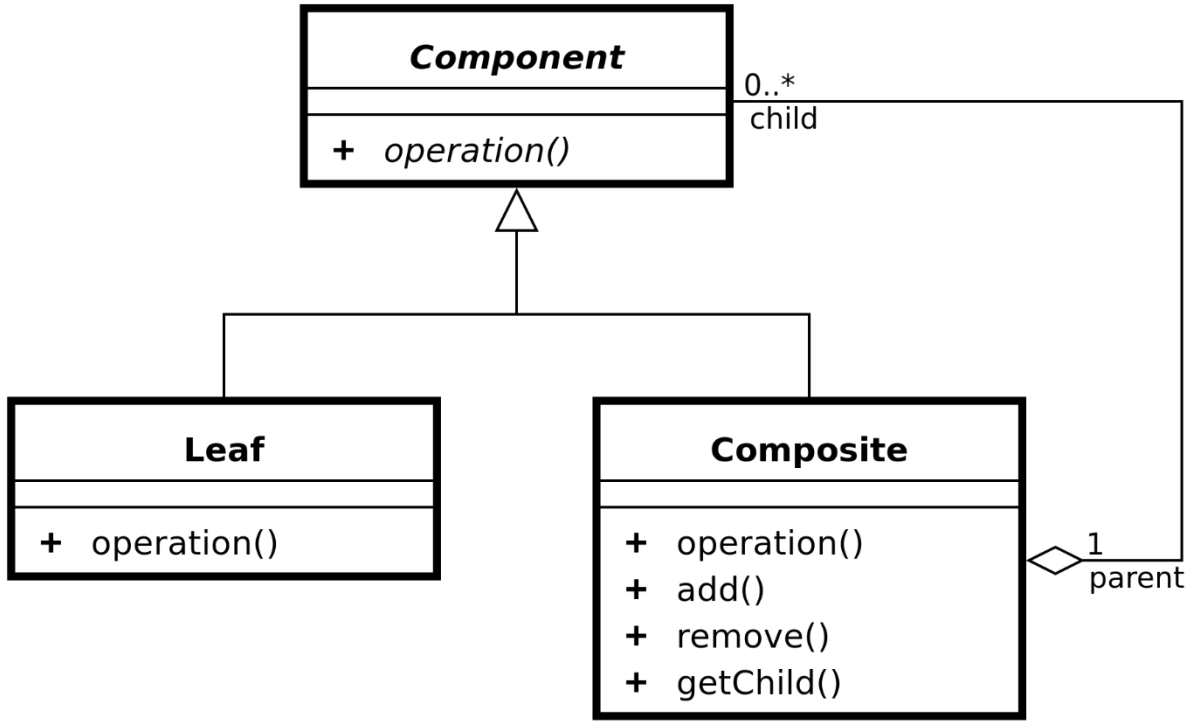
Design Patterns

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

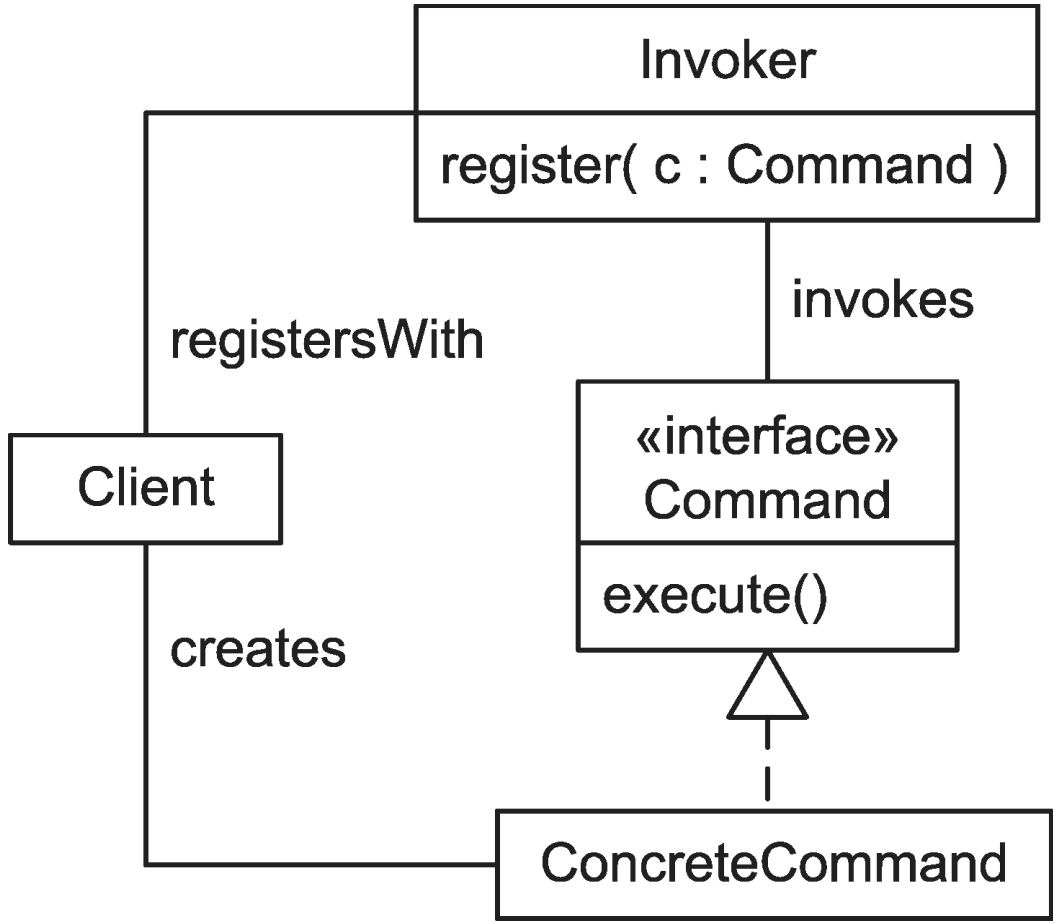
Composite pattern

- 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



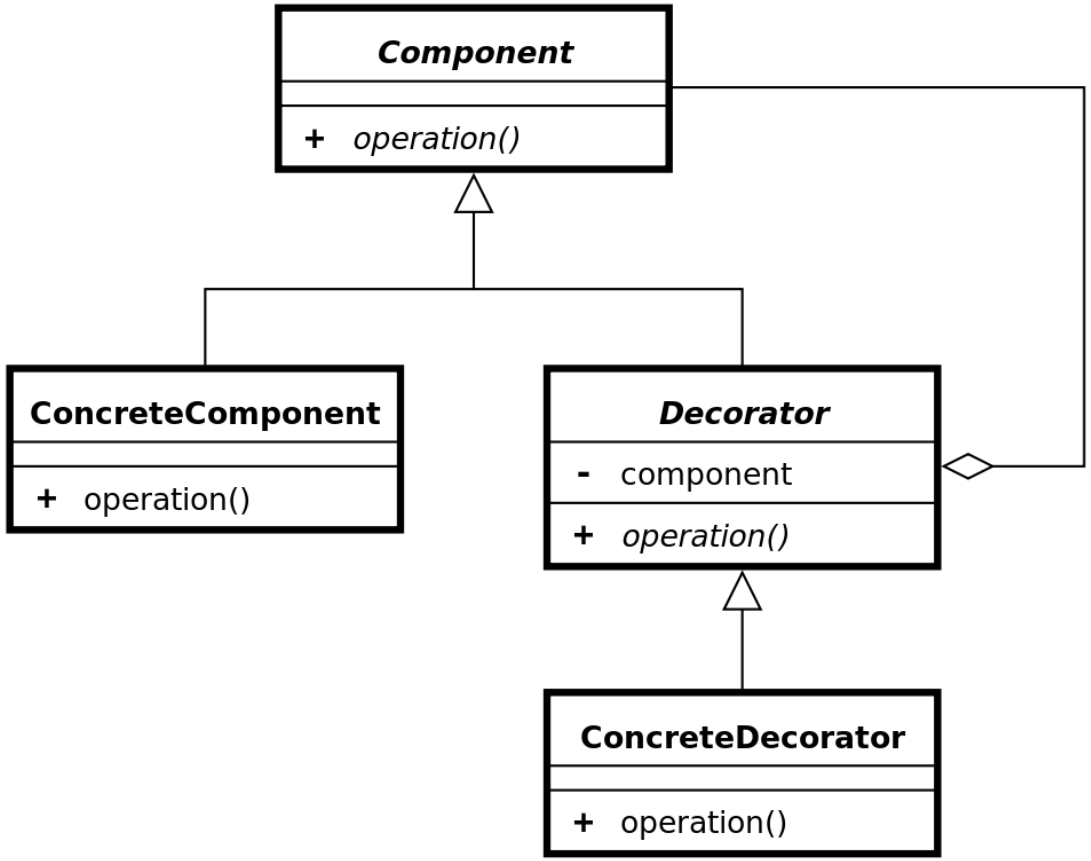
Command pattern

- 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



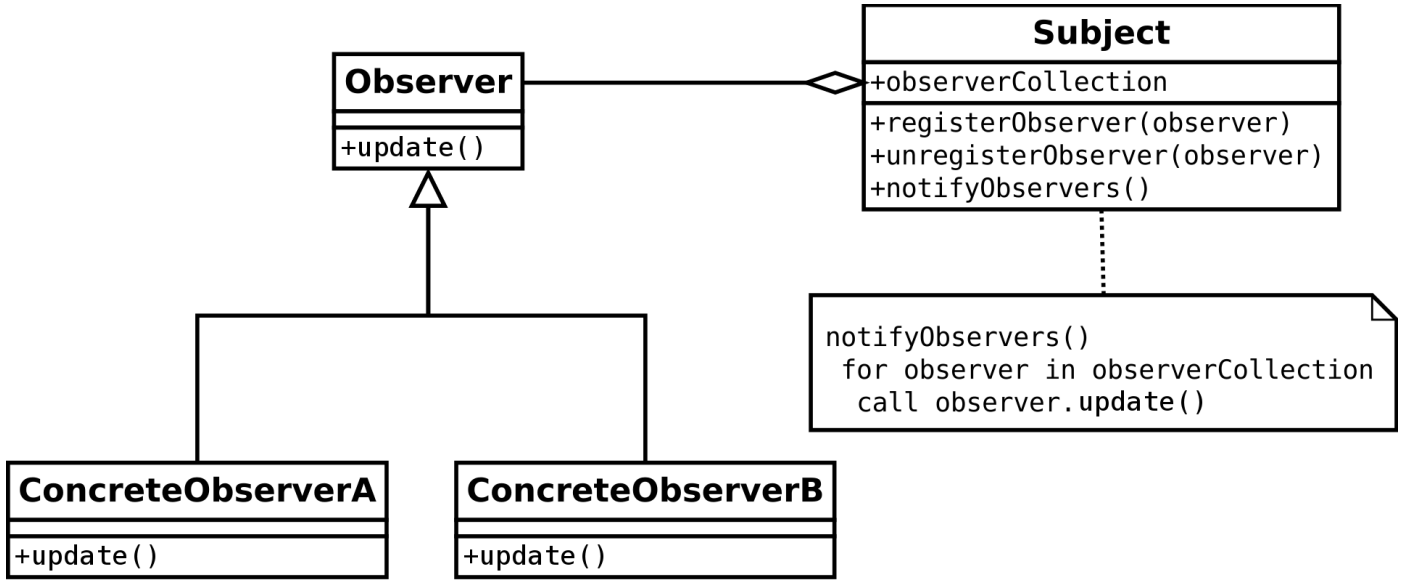
Decorator pattern

- 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



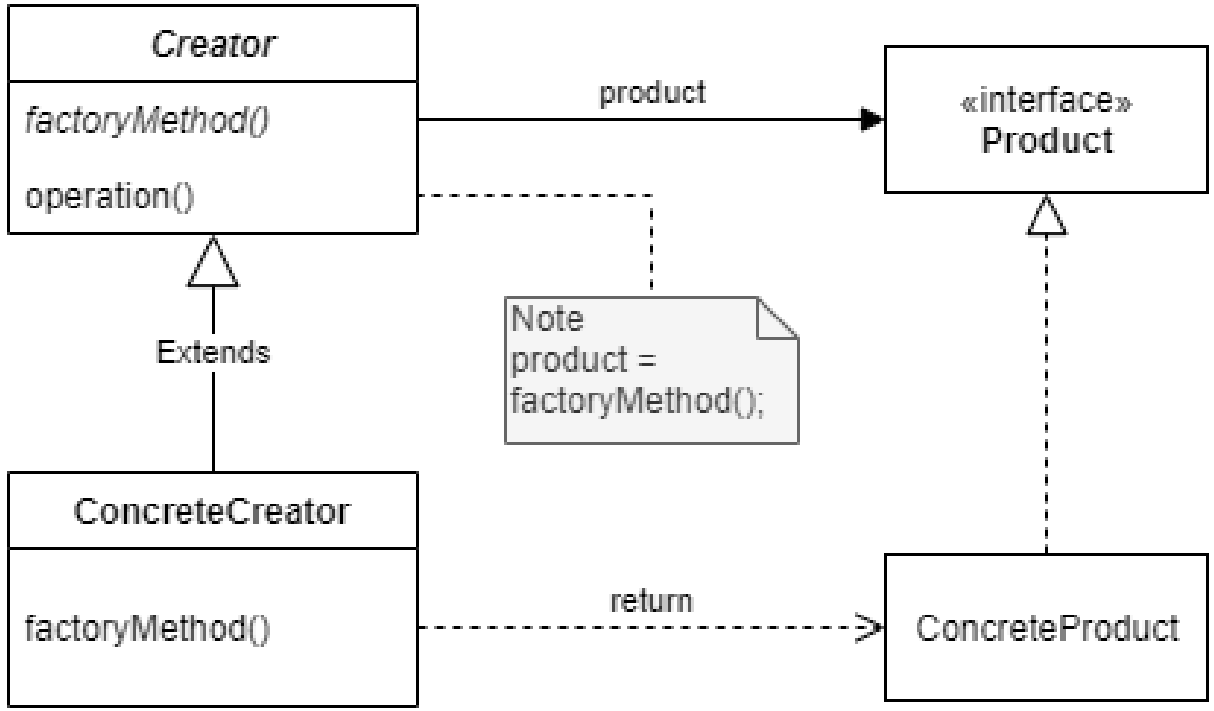
Observer pattern

- 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



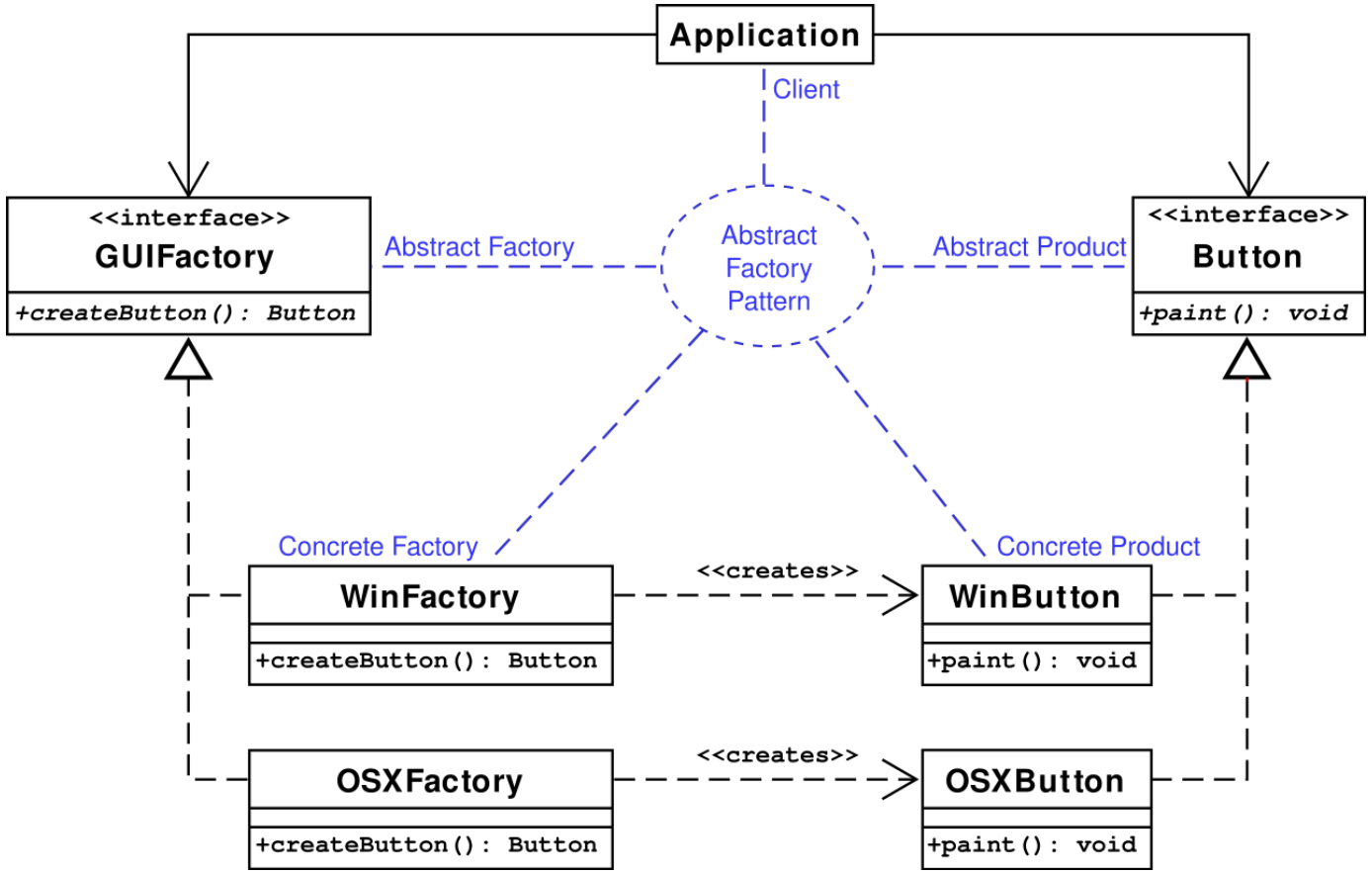
Factory method pattern

- 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



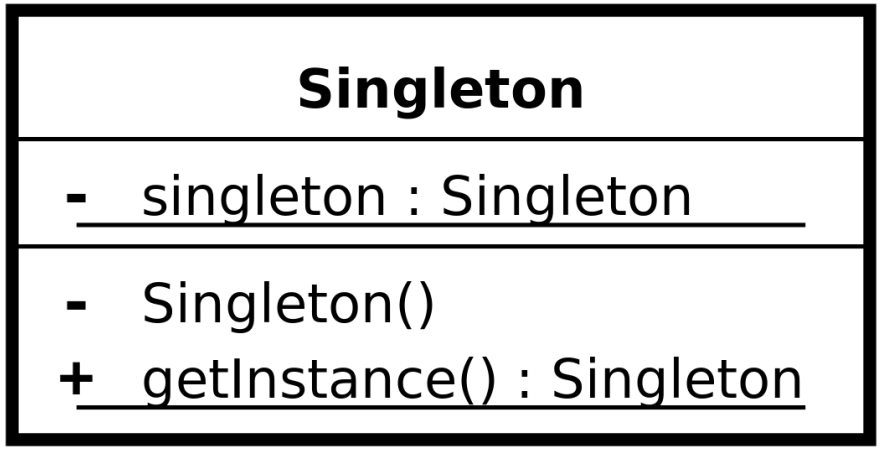
Abstract factory pattern

- 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



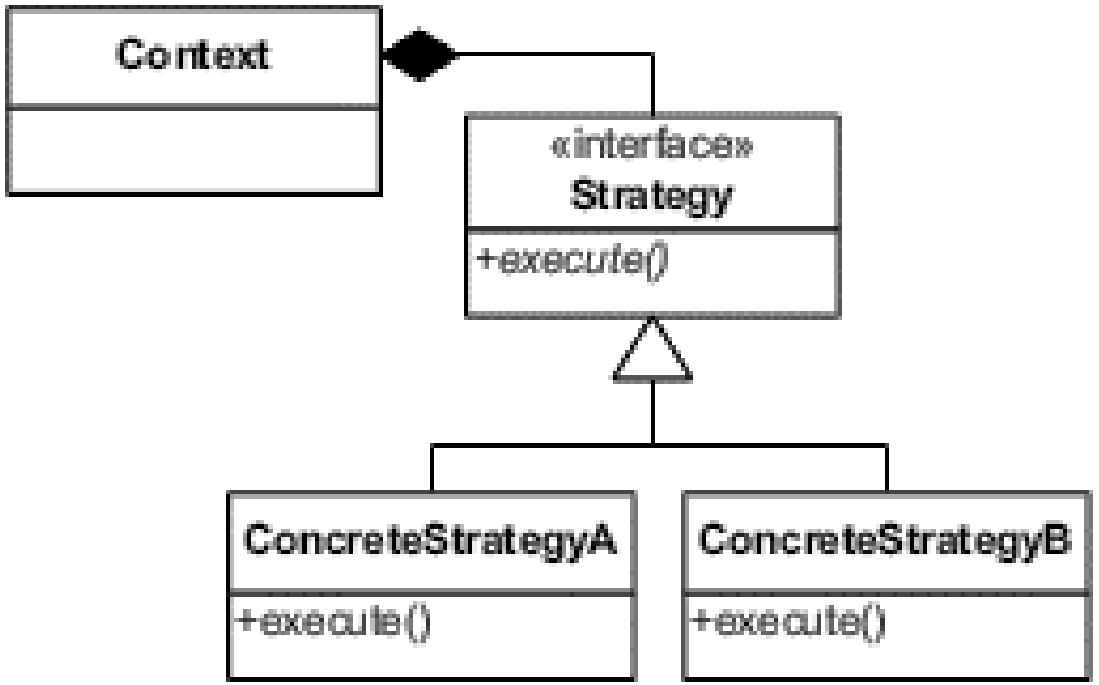
Singleton pattern

- 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



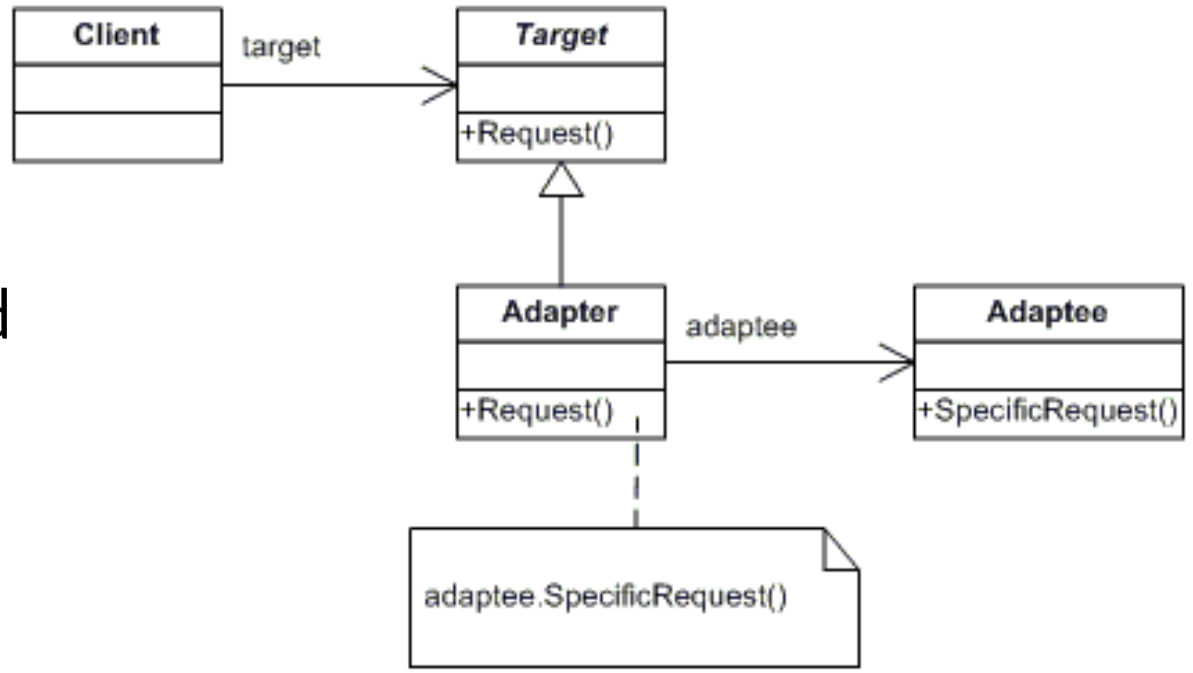
Strategy pattern

- 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

- A **bought and customized** system is one with several bought subsystems that have been customized and integrated into a product that satisfies requirements
- 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

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!");
}
```

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

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

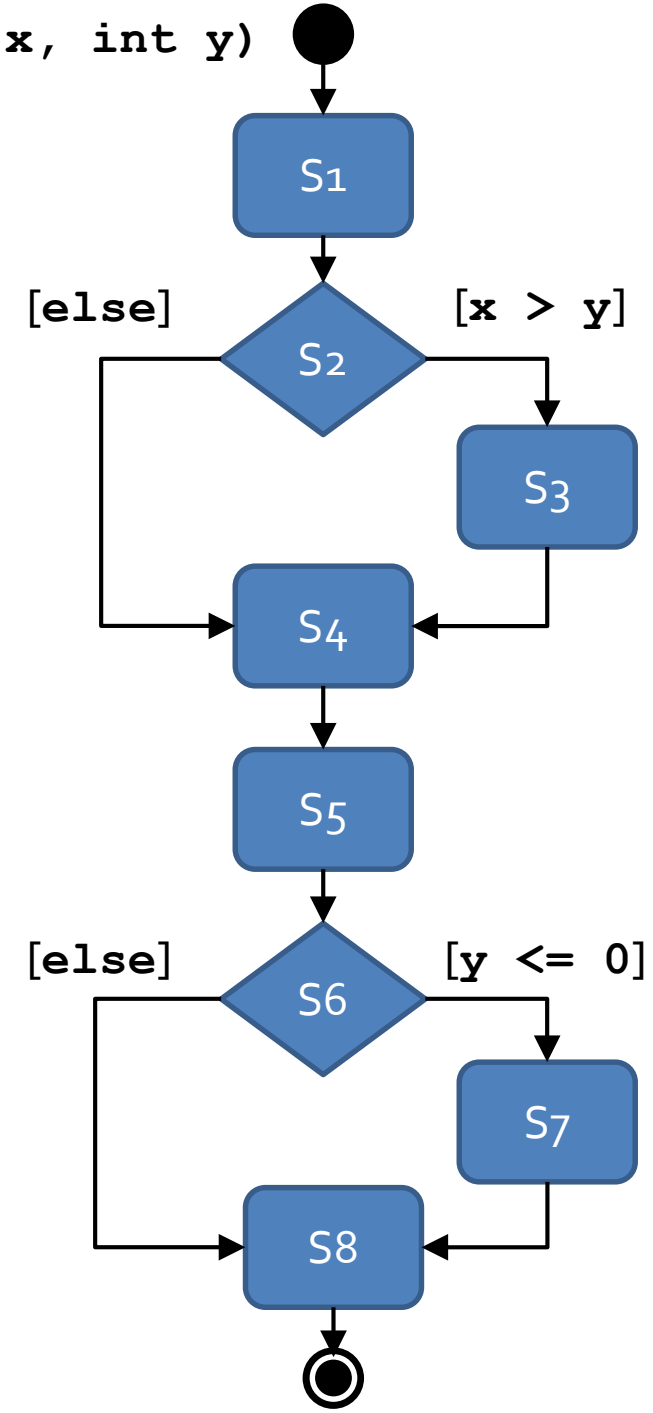
- Picking good test cases is an art form
- **Black box testing** is a strategy that assumes no knowledge of what happens inside the system
  - Only what the input and matching output should be are known
  - Black box testing is easily done by someone who had nothing to do with developing the code
  - 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	Minimum – 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

JUnit

- JUnit is a popular framework for automating the unit testing of Java code
- JUnit is built into IntelliJ 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 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

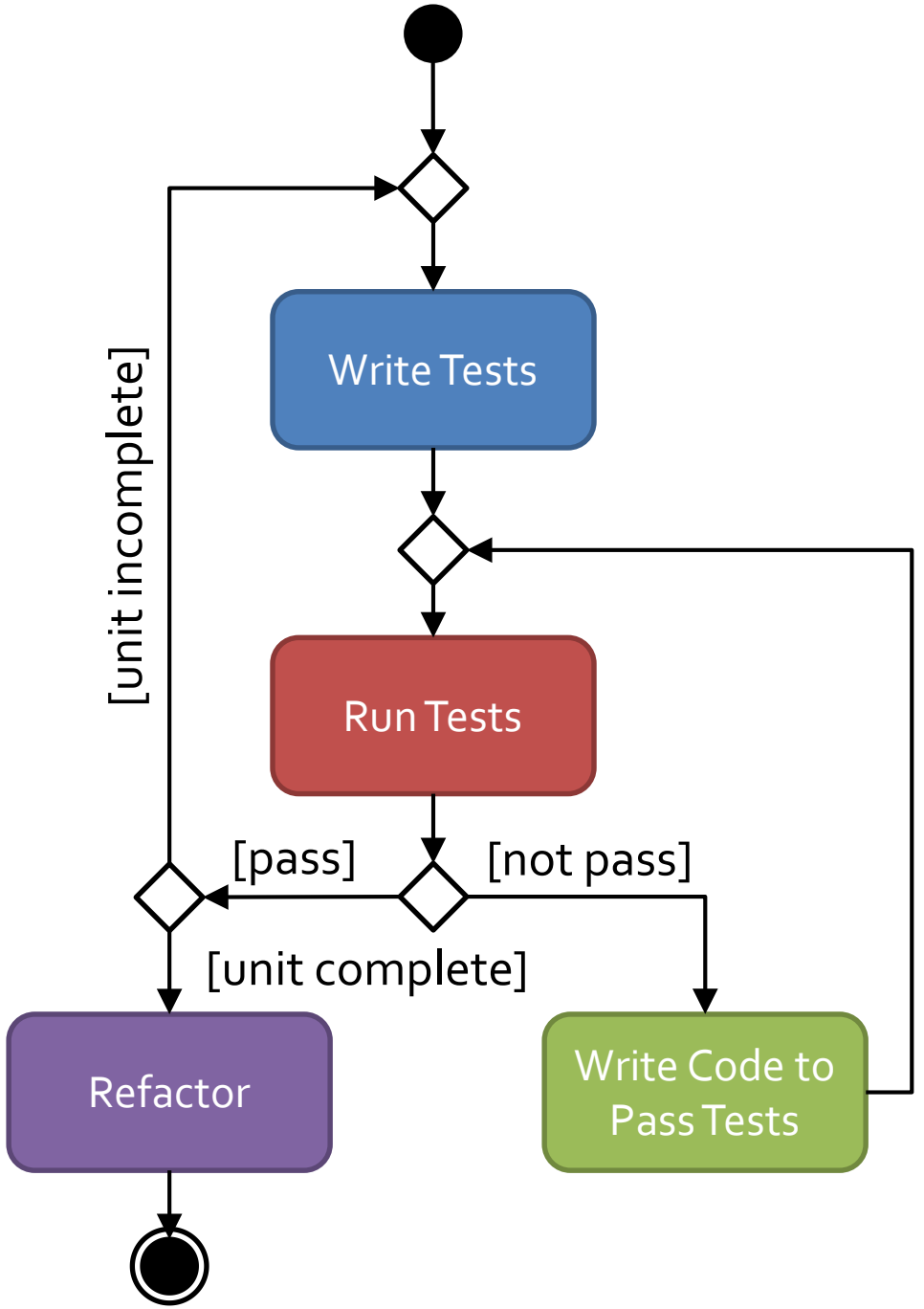
- Renaming a variable or method
- Adding an explanatory variable
  - If an expression is too long, storing a partial computation into a named variable can help it be understood
- Inline temporary variable
  - If a temporary variable is useless, just use the full expression (the opposite of the previous)
- 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

Deployment, Maintenance, and Support

Deployment, maintenance, and support

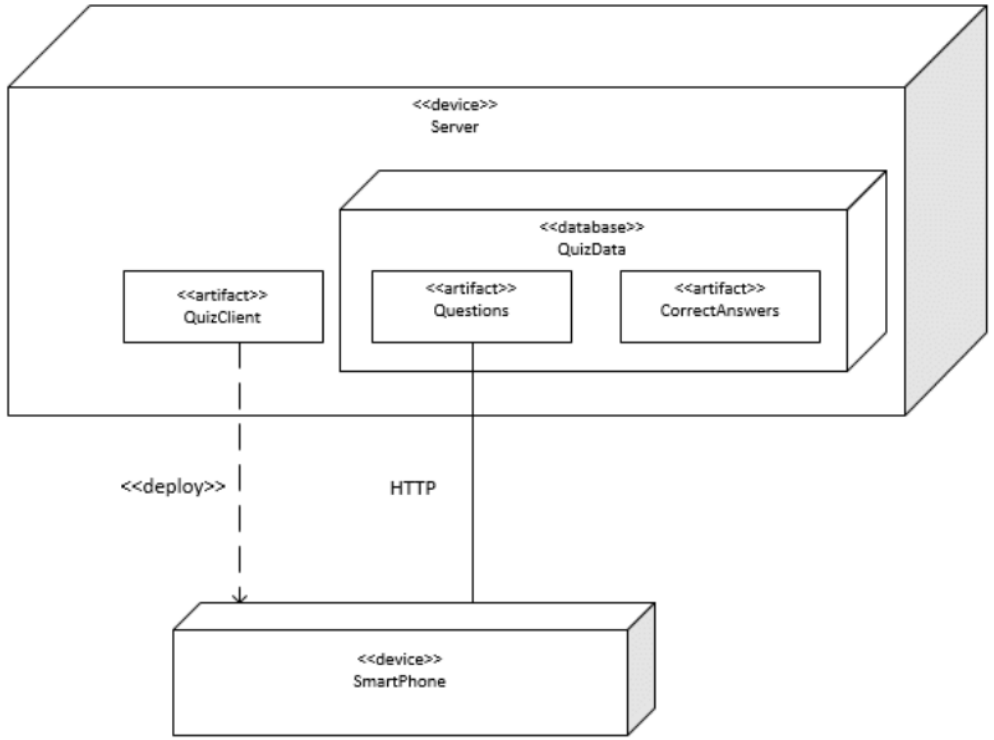
- The product has been designed, constructed, and tested…now what?
- Users will actually use the product in the **production environment**, the hardware and software systems where the product lives
- Making the product available in the production environment is called **deployment**
- Help that the developer (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 is 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)

Task Identification and Effort Estimation

Task identification and organization

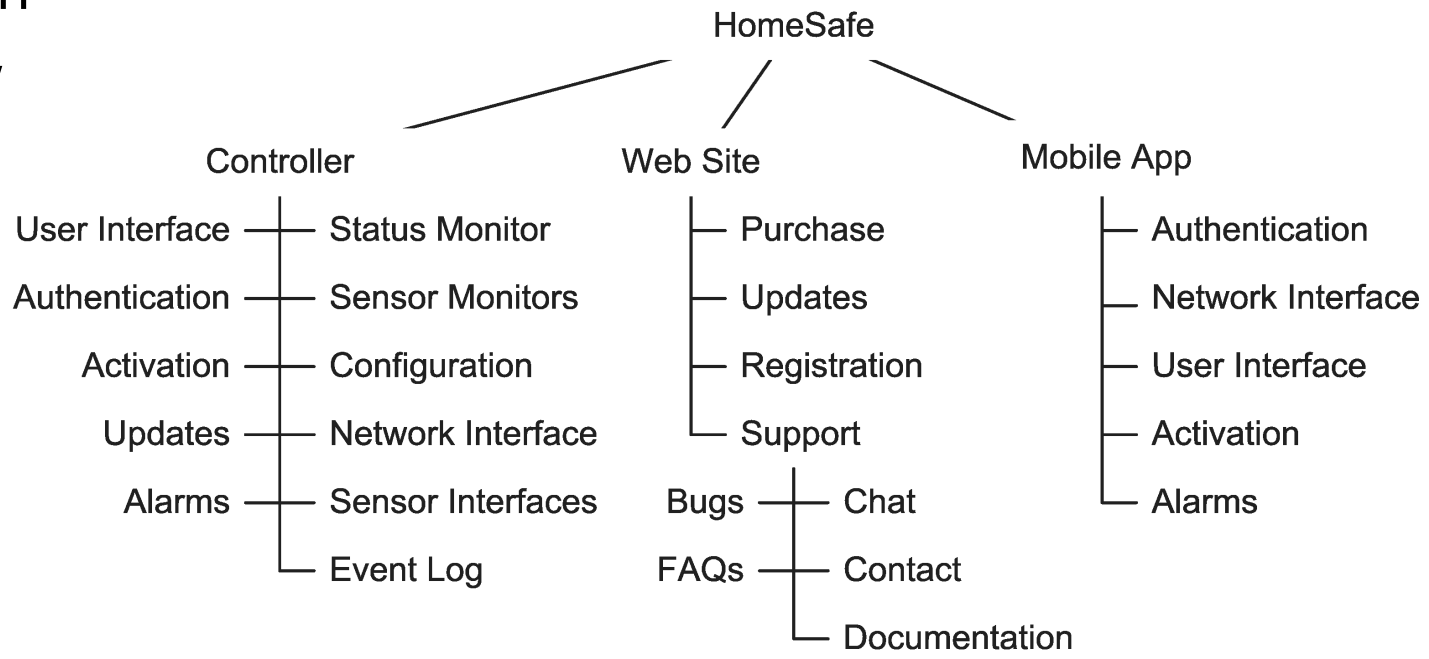
- High level tasks are pretty easy to identify
  - "Add networking support"
- But that level of detail isn't very useful
- Tasks are either:
  - Non-decomposable, also called **actions**
  - Decomposable, also called **activities** or **processes**
- The right level of detail is called a **work package**
  - A work package is a task that is small enough and detailed enough to estimate

Work breakdown structure

- A **work breakdown structure (WBS)** can be used to map out tasks at the right level of abstraction
  - The book prefers hierarchy diagrams to represent a WBS, since they balance the readability of trees with the space efficiency of hierarchical lists
- Nodes in a WBS are work to be done
- The root of a WBS is the project name
- The first level is all the deliverables for a project
- Each level below represents more and more detailed work
- Leaf nodes are work packages

WBS example

- The hierarchy diagram to the right shows a WBS for a home security product
- Note that different strategies can be used to decompose the work, especially at different levels:
  - Project deliverables
  - Product features or services
  - Project phases
  - Organizational units
  - Physical product decomposition
  - Logical product decomposition
  - Geographical location of team members



WBS heuristics

- But how do you know if you've done a good job breaking things down?
- **One hundred percent rule:** Nodes descended from a parent represent 100% of the work of the parent
  - Nothing's left out
  - No work is from outside the project
- **Mutually exclusive siblings:** No sibling nodes have overlapping work
- **8 / 80 rule:** Work packages (the leaves) take between 8 and 80 person-hours of effort
  - Work in that range (one day to two weeks) can be estimated reasonably well
- Get your project team and stakeholders together and make your WBS on a whiteboard

Effort estimation in traditional processes

- In traditional processes, effort estimation can be done in a few ways:
  - **Analogy:** Is your project like another project? It should take about the same effort
    - Problem: Only works if your project is very similar to another project
  - **WBS to effort:** Estimate the effort for each work package in a WBS and add them up
    - Problem: It's really hard to estimate effort accurately
  - **Size to effort:** Estimate the size of the final software product and use some math to predict how much work it will take to make the product
    - Problem: Oh, so many problems, which we'll discuss

Measuring size

- **Functional measures of size** have to do with how much functionality the program provides
  - Number of pages on a website
  - Number of reports in a database
  - Number of windows in a GUI
- **Non-functional measures of size** are based on the program's structure
  - Lines of code
  - Number of classes
- Non-functional measures are easy to measure after development but hard to predict ahead of time

Lines of code

- **Lines of code (LOC)** is a count of the lines of code needed for a project
  - LOC is the most popular non-functional measure of size
- Some people prefer **source lines of code (SLOC)**, ignoring whitespace (and perhaps comments)
  - It's even possible to weight some lines
  - LOC is only meaningful in context, since some programming languages tend to take more LOC to get the same job done
- Estimating LOC is done by breaking the product design into smaller and smaller components until the size of each component can be estimated
- Accuracy is hard to achieve early on, since there isn't even a design yet

"Measuring programming progress by lines of code is like measuring aircraft building progress by weight."

-Bill Gates

Function points

- Alternatively, a functional measure of size is possible called **function points**
- Function points are calculated by looking at five different types of components, organized into two categories:
- **Processes or Transactions**
  - **External Inputs (EI)**: Processes that provide data that will be used or stored by the product
  - **External Queries (EQ)**: Processes that retrieve stored data
  - **External Outputs (EO)**: Processes that provide derived information to a user (performing calculations)
- **Data Storage**
  - **Internal Logical Files (ILF)**: Groupings of data maintained by the product
  - **External Interface Files (EIF)**: Groupings of data external to the product but used by the product

Effort estimation

- All these estimates of size give us some arbitrary number, but how much effort is needed?
- **Algorithmic cost models** try to turn size estimates into a measure of effort called the person-month
  - The amount of effort a normal developer does in one month
  - Each person month has about 22 person-days
  - Effort covers all work from requirements, design, coding, testing documentation, collecting data, management, and so on

Simple models

- Maybe work grows linearly with function points
- Two different studies tried to model this to estimate effort $E = \alpha + \beta F$
- They found the following:

Study	α	β
Albrecht and Gafney	-91.4	0.255
Kemerer	-37.0	0.960

- These results are frustrating
  - The first one suggests that each function point adds ¼ person-month of work
  - The second suggests each function point adds about 1 person-month of work
- They were looking at different organizations and different accounting of function points, so estimates might work well only within an organization that is consistent about such things

Exponential models

- Alternatively, some researchers have looked at exponential models relating thousands of lines of source code (KLOC) to total effort using the following equation, where *L* is KLOC:
  - $E = \alpha \cdot L^{\beta}$
- Results found the following values of α and β:

Study	α	β
Watson and Felix	5.20	0.91
Basili and Freburger	1.38	0.93
Boehm	3.20	1.05

- Note here that $\beta < 1$ means economies of scale (time per line of code decreases at the project grows) while $\beta > 1$ means the opposite

Effort estimation in Scrum

- Everything we said before was about waterfall estimates
- Scrum skips size estimates and goes straight for effort estimates
- As you know, units of effort in Scrum are called **story points** (or sometimes task points)
  - Story points are relative units
  - They're based on some of the smallest tasks, using them as a baseline of 1 story point
  - Everything is estimated relative to those
- Story points aren't used for epics since they're too big and abstract
- As PBIs get refined, their effort estimate gets refined too
- By the time they're sprintable, they need a relatively accurate story point estimate
- This means that there are good estimates for sprintable stories but no estimates for how much work the whole project will take

Detailed estimation in Scrum

- What if members of the team disagree on the story points needed for several stories?
- Agreement is needed for the sake of fairness and to plan how much work can actually get done in a sprint
- **Planning poker** is a way to bring the team to consensus about the relative difficulty of user stories
- Its goal is accuracy (ranking the stories by true difficulty) rather than precision (getting true estimates of how long things will take)
  - It's really hard to get true estimates, but it's good to know which stories take more work

Planning poker

- First, the team decides what numbers to use as estimates
- The numbers are usually sequences that grow exponentially, written on cards
  - Modified Fibonacci: 1, 2, 3, 5, 8 ,13, 20, 40, 100
  - Powers of two: 1, 2, 4, 8, 16, 32, 64
  - This means that large stories won't be estimated precisely, but that's okay
- Planning poker has rounds
  - Each round estimates the effort for one PBI
  - Each team member throws in one card to show her effort estimation
  - If all cards match, the value is the estimate
  - If they don't match, the team discusses their estimates, focusing on the highest and lowest estimators
  - Repeat the round until consensus is reached
- It usually only takes a couple of rounds to reach consensus
- Estimates are usually pretty good because of discussion

Financial and Economic Planning

Finances matter

- There are three software planning activities where finances are particularly important
- Buy or lease decision
  - Should software or hardware be bought or leased?
  - Will it cost more or less to use open-source software over time?
- Make or buy decision
  - Should software be written from scratch or bought?
- Go or no-go decision
  - Should a project (especially a software development project) be done or not?
  - If the project is already running but there are issues, should it be continued?

Financial conventions

- Assets are good things
- Liabilities are bad things
- Even so, for reasons best known to business people, debt is sometimes listed as an asset
  - Take an accounting course if you want to go deep into this
- We will look at money from some actor's perspective
  - Assets like deposits and revenues will be positive
  - Liabilities like costs and rental payments will be negative

Time value of money

- Money seems simple, but a lot of the madness business people talk about boils down to a question:
  - Would you rather be given $100 today or $110 a year from now?
- A similar question is:
  - Should you spend $1,000 now to prevent a security hole that is likely to cost your business $10,000 over the next 10 years?
- The answer to both of these questions is:
  - **It depends.**
- You **cannot** answer these questions without considering *what else* you would be doing with the money over the time periods in question
- Money doesn't simply sit around: it can be invested with the chance to grow over time

Simple interest

- What money is doing over time is earning **interest**
- Interest is built around a few variables:
  - The present value *P* of the money
  - The number of periods *n* during which the money will be earning interest
  - The periodic interest rate *r*, which is the percentage of the present value the money will earn each period
- With simple interest, the interest $I = P \cdot n \cdot r$
- Future value $F_n = P + I$
- Example:
  - $P = \$1{,}000, n = 5, r = 0.70\%$
  - $I = \$1{,}000 \cdot 5 \cdot 0.007 = \$35$
  - $F_n = \$1{,}035$

Compound interest

- Our example with $1,000 at 0.7% interest for 5 years was simple interest
  - Which no one actually uses
- **Compound interest** means that when you earn interest in one period, you get to earn interest on that interest in the next period
- If we compounded every year, our example would become:
  - $F_n = \$1{,}000 \cdot 1.007 \cdot 1.007 \cdot 1.007 \cdot 1.007 \cdot 1.007 = \$1{,}035.49$
  - We got $0.49 more!
  - These interest rates are why savings accounts suck right now
- Compound interest: $F_n = P \cdot (1 + r)^n$
- Imagine you could earn 5% a year for 10 years
  - Simple interest: $F_n = \$1{,}000 + \$1{,}000 \cdot 10 \cdot 0.05 = \$1{,}500$
  - Compound interest: $F_n = \$1{,}000 \cdot (1.05)^{10} = \$1{,}628.89$

Discounted present value

- What if one of your options was to get $1,000 at some point in the future or to get some other sum right now?
- Working backwards from $1,000 with a given interest rate, what is the present value of that money?
  - $F_n = P \cdot (1 + r)^n$
  - $P = \frac{F_n}{(1+r)^n}$
- Calculating the present value from a future value is called **discounting**, which finds the **discounted present value**
- If someone promises $1,000 after 6 years with an interest rate of 8% compounded annually, the discounted present value is
  - $P = \frac{\$1{,}000}{(1.08)^6} \approx \frac{\$1{,}000}{1.586874} = \$630.17$
- Thus, if you can get more than $630.17 right now, it's better to do that
- If you can't, it's better to take $1,000 in the future

Example: buy or lease

- Consider a server you need for a 3-year project
- You have two options:
  - Buy the server for $-4,000
  - Lease it for four payments: $-1,000 now, $-1,000 in a year, $-1,100 in two years, and $-1,150 in three years at the end of the project
- Naïve math says that $-4,000 is better than $-1,000 + $-1,000 + $-1,100 + $-1,150 = $-4,250
- However, we can apply the discounted present value to those later payments (because we could have been investing that money)
- As you can see in the table to the right, the discounted present value of $-3,943.52 is better than $-4,000

n	F_n	$(1 + r)^n$	$F_n / (1 + r)^n$
0	-1000.00	1.00	-1000.00
1	-1000.00	1.05	-952.38
2	-1100.00	1.1025	-997.73
3	-1150.00	1.157625	-993.41
			-3943.52

Internal rate of return

- What if you knew how much someone would pay you today and how much you could get paid in the future and needed to compute the rate of return needed to make them match?
  - This helps you look for another way to spend your money with a better interest rate
  - Or it helps you understand the rate of return that a project provides
- It's simple algebra, solve for *r*:
  - $F_n = P \cdot (1 + r)^n$
  - $\frac{F_n}{P} = (1 + r)^n$
  - $r = \left(\frac{F_n}{P}\right)^{\frac{1}{n}} - 1$
- If you have multiple stages of costs and revenues, you'll need to do a binary search on *r* values:
  1. Start with a minimum bound for r and a maximum bound for r
  2. Guess the rate halfway between them
  3. Run through the math on the previous slide to see what the net is
  4. If it's too high, go back to Step 1 with the minimum and the midpoint as your range
  5. If it's too low, go back to Step 1 with the midpoint and the maximum as your range
  6. When the minimum and the maximum are close enough together (like 0.001%), you have a good estimate

Uncertainty

Uncertainty

- The previous examples have been **deterministic** (no uncertainty)
- But real life often needs **stochastic** models (which include uncertainty)
- To handle uncertainty, we need probability
- A possible outcome is called a **sample point**
- The set of all possible outcomes is called the **sample space**
- Each sample point has a nonnegative probability p_i where $p_1 + p_2 + \cdots + p_m = 1$
- In other words, the probability of all sample points together must sum to 1
- **Note:** For those of you who know more probability, sample points are events with only a single, mutually exclusive outcome

Expected value

- Assume each sample point has a value (like the money associated with that outcome)
- The **expected value** is the value of each sample point multiplied by its probability
  - It's the average value of everything, weighted by the probability that it happens
- Example:
  - You're playing roulette, always betting on black
  - An American roulette wheel has 38 outcomes: 18 are red, 18 are black, and two are neither (0 and 00)
  - If you bet $1 on black:
    - You have an 18/38 chance of winning $1
    - You have a 20/38 chance of losing $1
  - Expected value = $\$1 \cdot \frac{18}{38} - \$1 \cdot \frac{20}{38} \approx -\$0.05$
  - Thus, you'll win some and lose some, but on average, you'll lose about $0.05 each time they spin the wheel

Simple example with uncertainty

- Your company needs to install some free software
  - There's a 20% chance that the installation will be effortless and cost about $100 of worker time
  - There's an 80% chance that the installation will be a huge pain and cost about $8,000 of worker time
- Expected cost of the installation is:

$$0.20 \cdot \$100 + 0.80 \cdot \$8{,}000 = \$20 + \$6{,}400 = \$6{,}420$$

Scheduling

People

- Effort *E* is given in person-days or person months
- Thus, time *T* could be given by the following equation where *N* is the number of people:

$$T = E / N$$

- Unfortunately, this ideal equation is unlikely to work out for a couple of reasons

Details of tasks

- Some tasks are easy to split up, and others are not
- If it takes 5 minutes to pump up a bicycle tire, you can't do it 100 times faster by using 100 people instead of one
- Small tasks can usually only be done by a single person at a time
- Larger tasks generally obey the $T = E / N$ rule, but there are diminishing returns for large values of N

Task dependencies

- Looking at the time to do individual tasks isn't enough
- Consider tasks A, B, and C with the following amounts of effort:
  - A: 5 person-months
  - B: 3 person-months
  - C: 4 person-months
- If we have three employees, one can work on each task
- If the tasks are independent, the project will take 5 months to do, and tasks B and C can even run late without delaying the total project
- What if C requires B to be done and B requires A to be done?
  - If any task is delayed, the whole project will be delayed
  - We have to share work on each task

Personnel capabilities

- Some developers are better than others
  - This messes with the overall $T = E / N$ rule
- Some developers have specialized in certain areas
  - A tester might be great at testing but not so good at development
  - Only one person on the team might have experience with GUIs
- As a consequence, it might not be possible to have multiple people working on a given task, and one person might be needed for two different tasks
- Agile methodologies supposedly improve these issues by trying to make everyone work on everything and grow their skills

Simplifying assumptions

- We assume that we have a good estimate of the relationship between effort and time
- We assume small tasks assigned to one person
- We assume a dependency between two tasks if only one person has the skills needed to do both
  - This allows us to look at the problem of specific skillsets as the more general problem of dependencies
- With these assumptions, we can organize our tasks by duration and dependency

Dependency example

- The following example shows 14 tasks
  - The time for each task is given
  - The prerequisite tasks that must be done first are listed too
  - Tasks are numbered so that higher number tasks are dependent on lower number tasks
- This way of formatting the information doesn't make it easy to figure out several things we want to know:
  - How long the whole project will take
  - **Critical tasks**, the ones that determine the minimum time for the project
  - **Slack** (or **float**), the amount of time non-critical tasks can slip without delaying the project

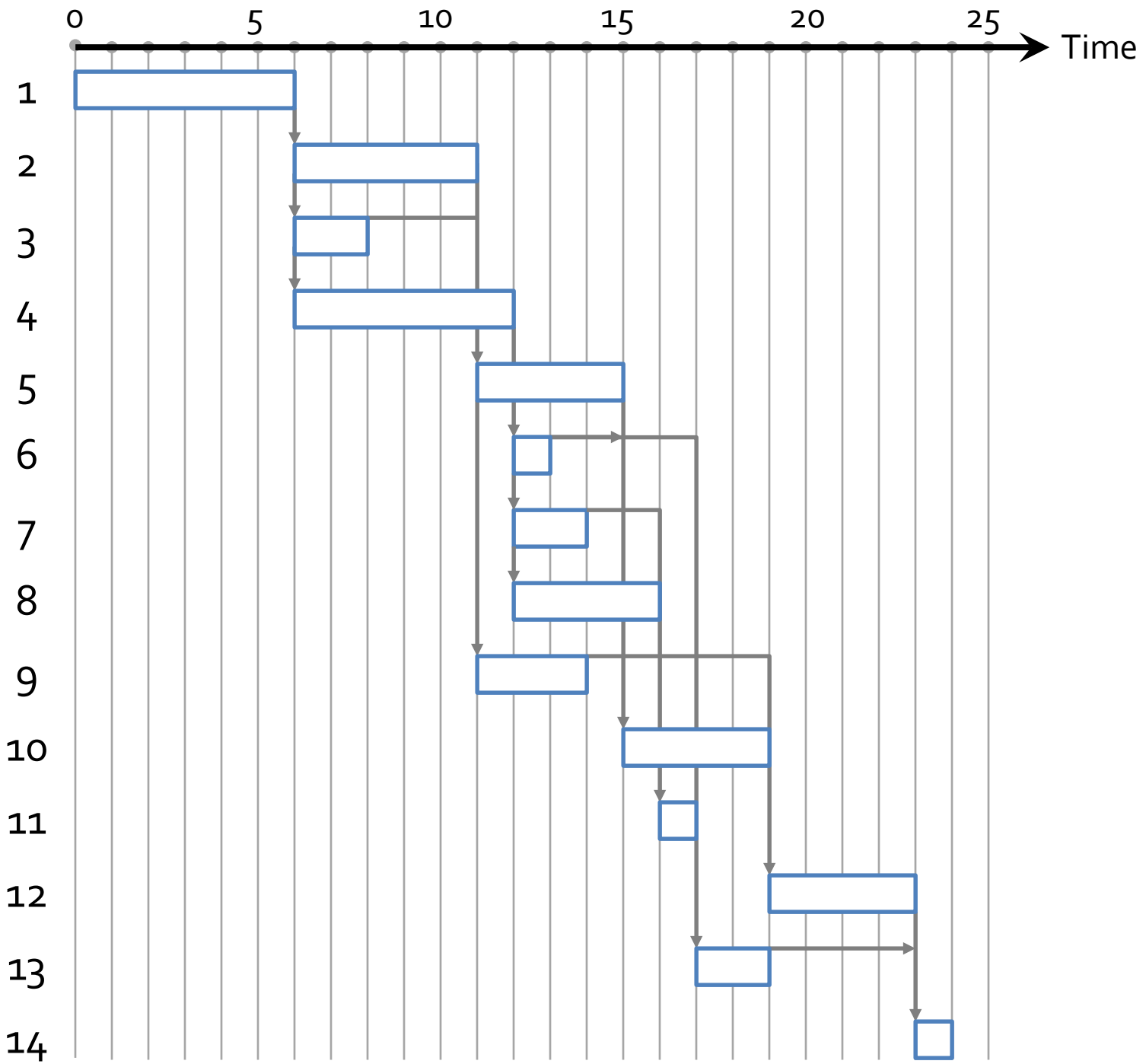
Task Number	Duration (Days)	Prerequisite Tasks
1	6	-
2	5	1
3	2	1
4	6	1
5	4	2, 3
6	1	4
7	2	4
8	4	4
9	3	2
10	4	5, 6
11	1	7, 8
12	4	9, 10
13	2	6, 11
14	1	12, 13

Gantt charts

- Gantt charts let us find total time, critical tasks, and float times
  - Tasks are represented as rectangles with length proportional to duration
  - Dependencies between tasks are arrows
  - Time increases from left to right
  - We put the task starts as early as possible, immediately after their last prerequisite finishes

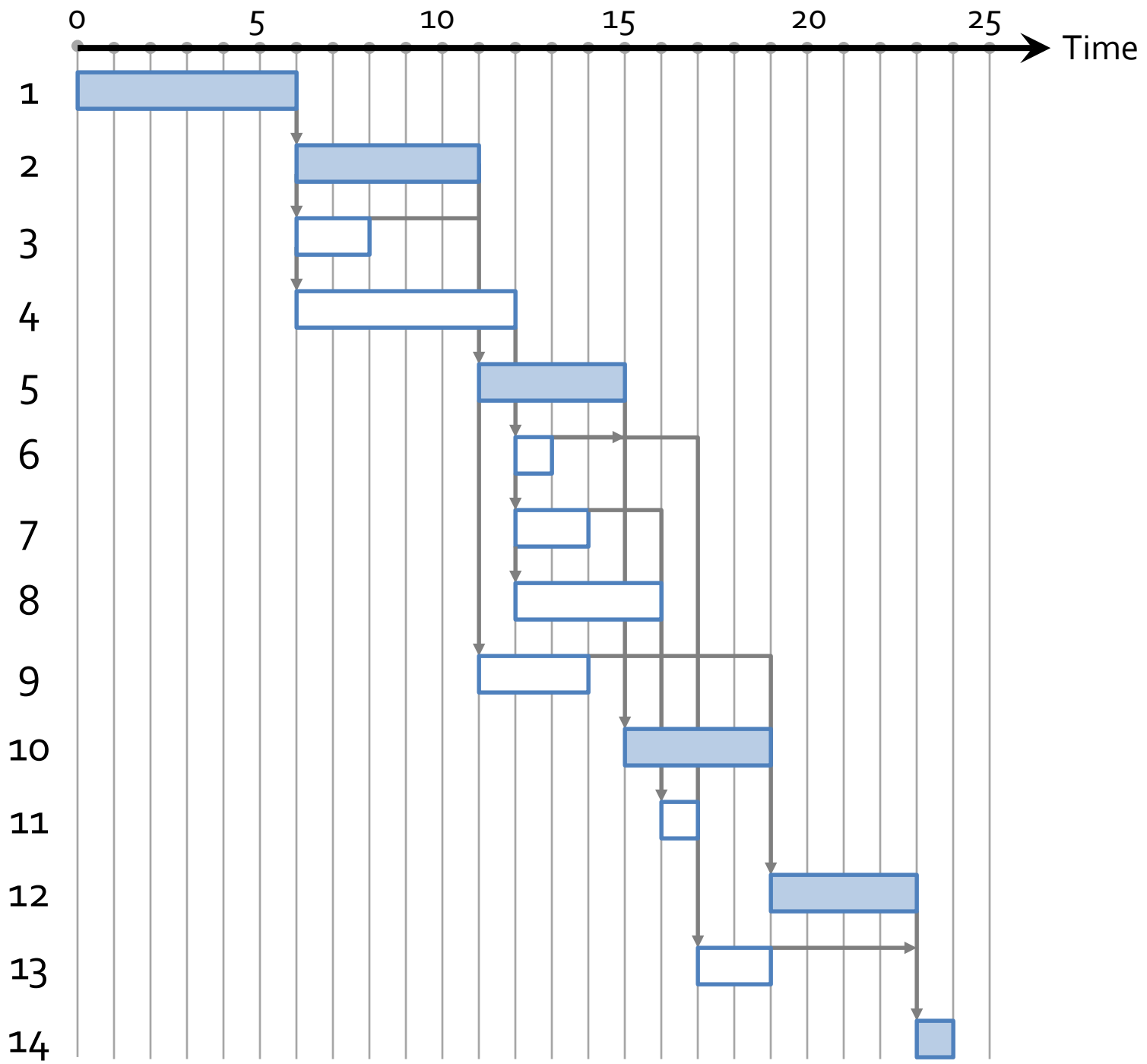
Full Gantt chart

- Here is the full Gantt chart
- People don't always draw the arrows, but we're doing so here to be explicit
- Looking carefully at the chart, it's clear that the project will take 24 days



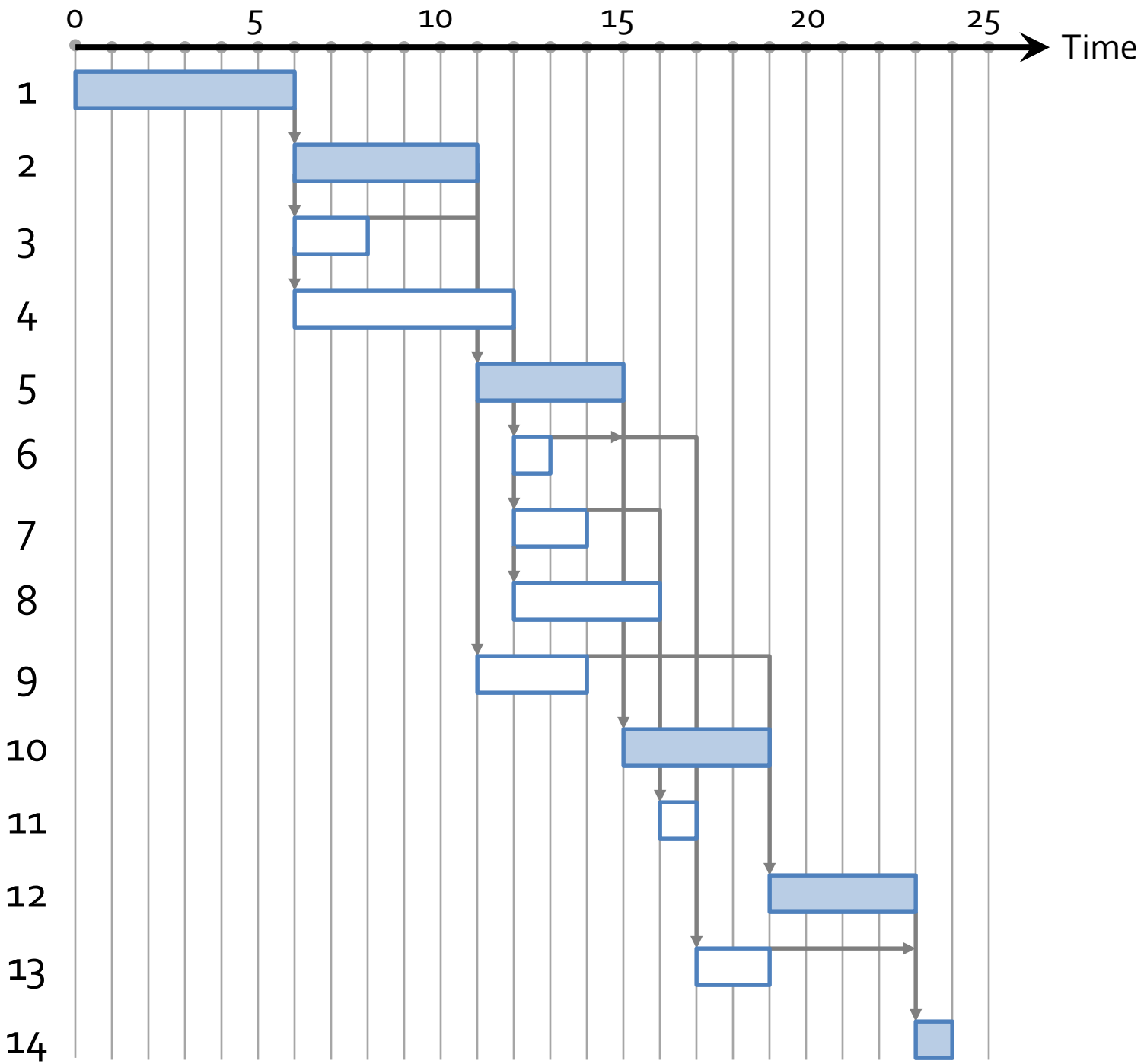
Critical tasks

- We can find the critical tasks by working backward from the task(s) with the latest finish time
- Whichever of its predecessors have the latest end time are also critical
- If any of these are delayed, the whole project will be delayed



Slack time

- Non-critical tasks have slack, an amount of time they can slip by and still not delay the project
- Horizontal arrows show slack times:
  - Task 3: 3 units
  - Task 6: 2 units
  - Task 7: 6 units
    - Tricky! It's 6 because that's what's needed before a critical task will be delayed
  - Task 9: 5 units
  - Task 13: 4 units



Critical path methods

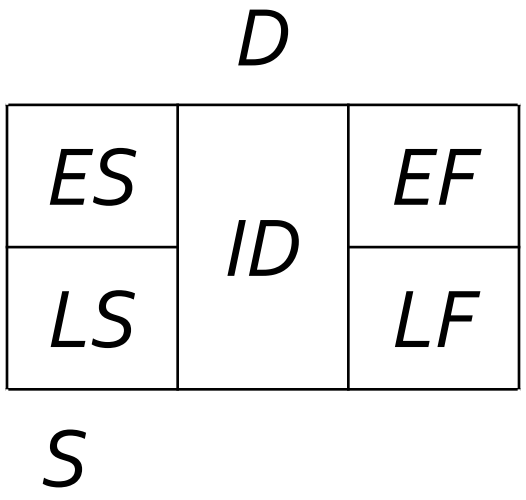
- Computer scientists love to use computer science for everything, even project management problems
- In addition to Gantt charts, similar information can be represented using graphs
  - Then, graph theory tools can be applied to the information
- These approaches are called critical path methods (CPM) because they focus on making the critical path as short as possible

More on critical path

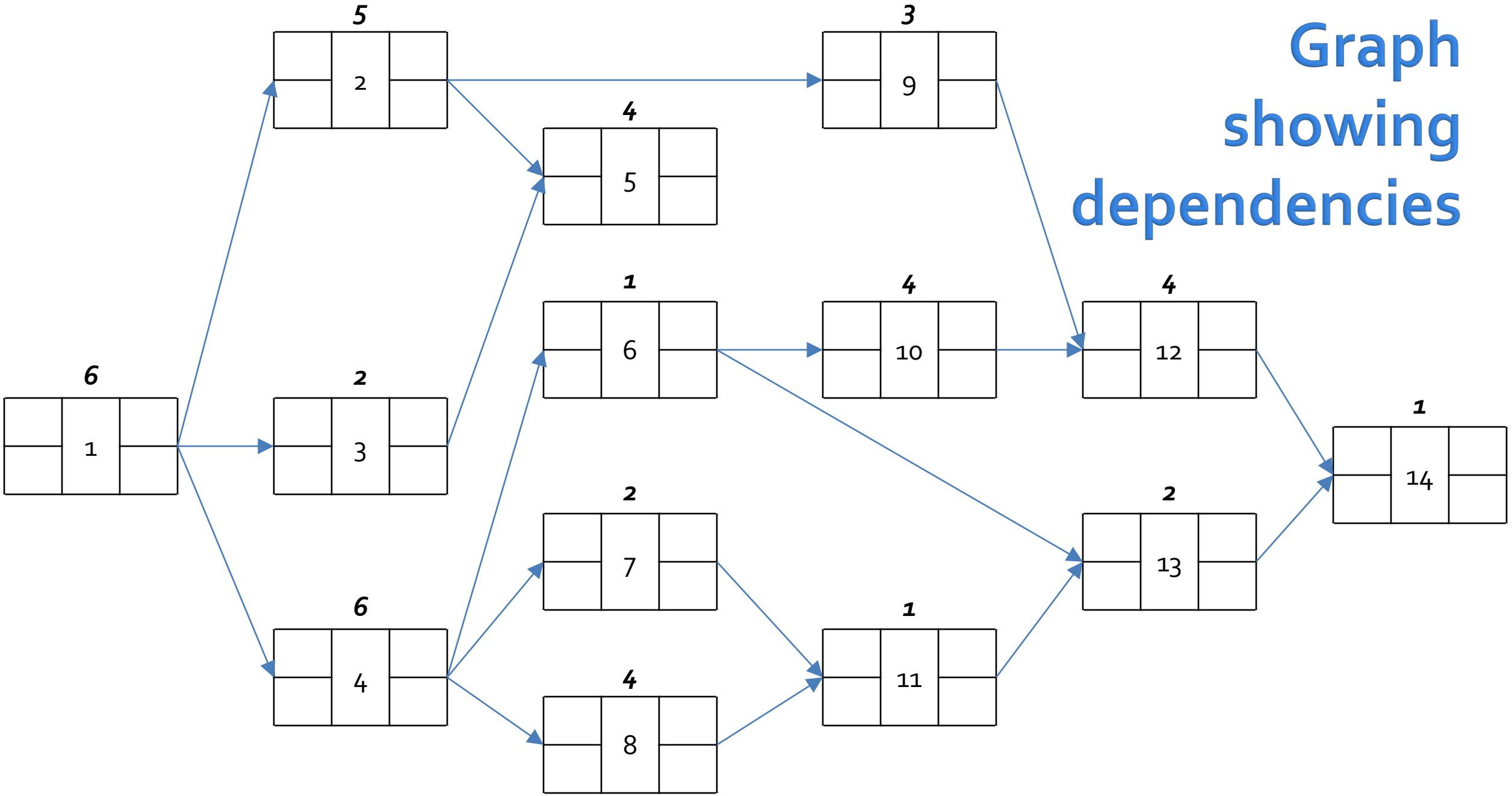
- An important idea that critical path methods add to the mix is a tradeoff between time and cost
- Each task has:
  - A **normal time** that the task would take
  - A **crash time** which is the fastest a task could possibly be done by spending more resources
  - A (usually linear) relationship between putting resources in and getting the task done quicker
- By using linear programming, a technique for finding optimal solutions to linear systems of equations, the cheapest way to finish a project by a given deadline could be determined
  - Maybe rushing Task 7 is worth the extra money but rushing Task 10 isn't

Nodes in a CPM graph

- The CPM we will talk about has nodes containing seven pieces of information, written in a peculiar way
  - *ID*: Task identifier
  - *D*: Task duration
  - *ES*: Earliest start time
  - *EF*: Earliest finish time
  - *LS*: Latest start time
  - *LF*: Latest finish time
  - *S*: Slack



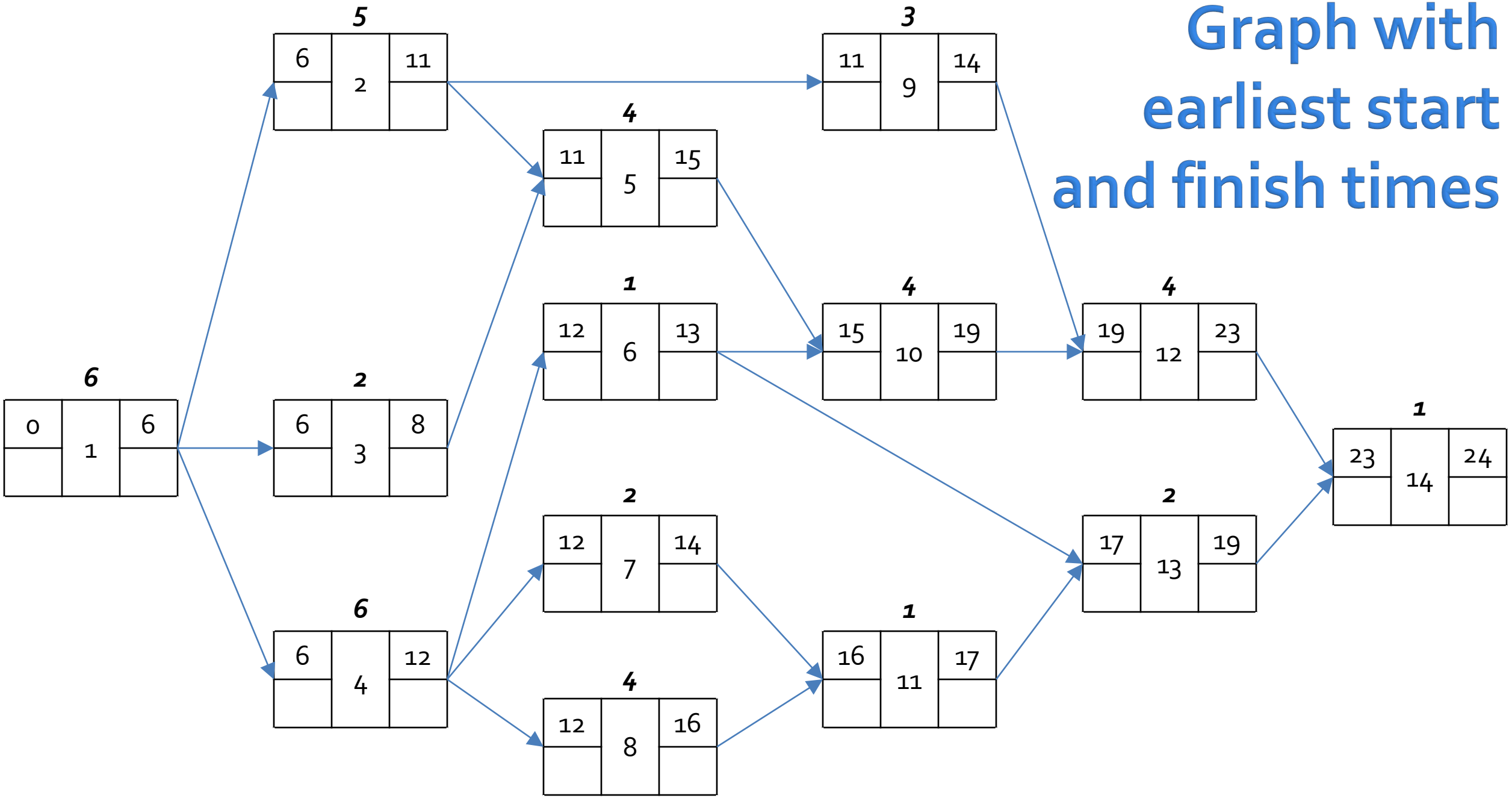
Graph showing dependencies



Earliest start and finish times

- Every task with no prerequisite has an *ES* of o
- For a task with prerequisites, its *ES* is the maximum *EF* of all of its prerequisites
- For each task, $EF = ES + D$
- Using these relationships, we can fill in the *ES* and *EF* for each task, starting from those with no prerequisites and working through the rest of the graph

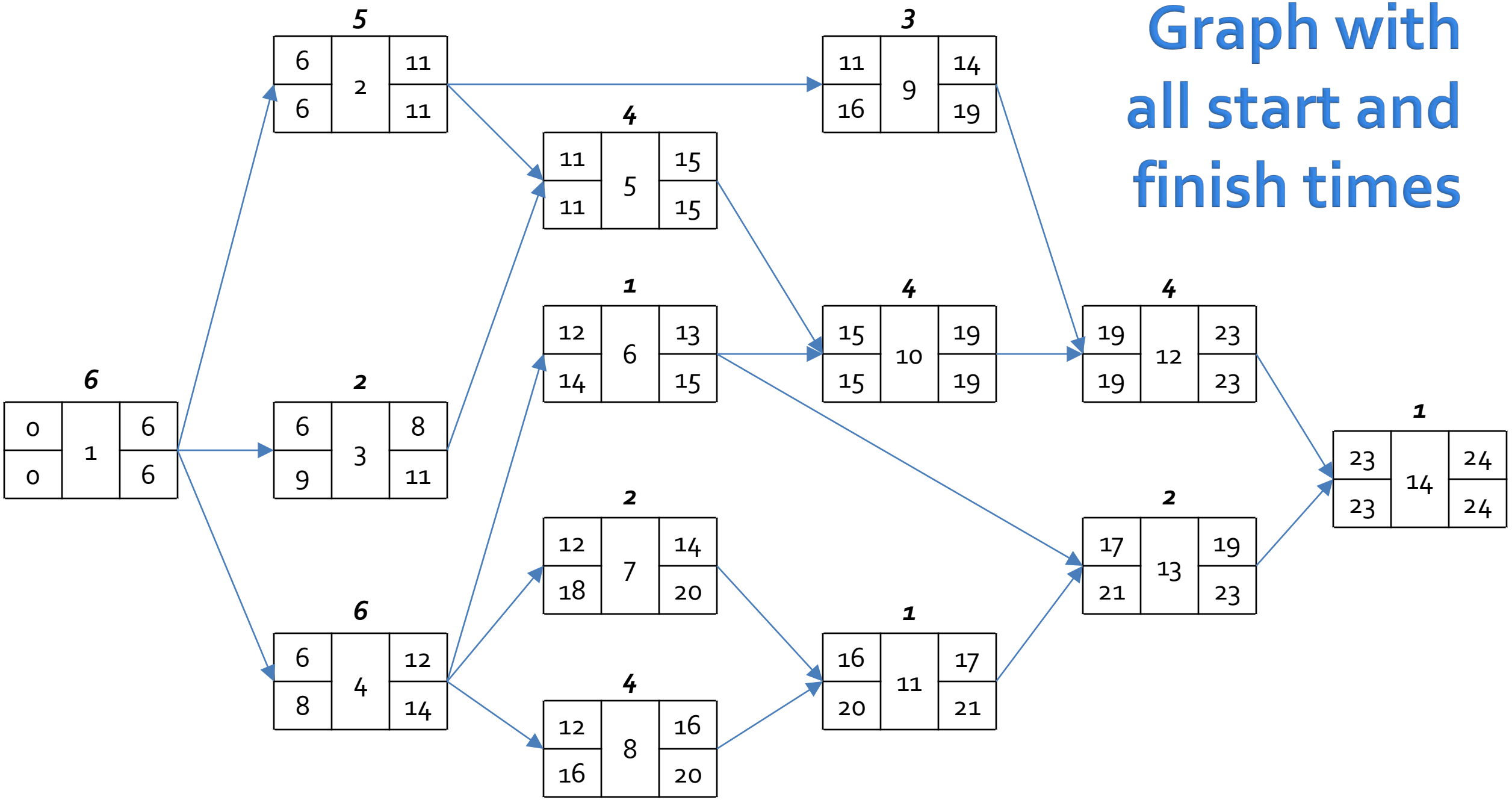
Graph with earliest start and finish times



Latest start and finish times

- Every task that isn't the prerequisite for anything has an *LF* = *EF*
- For a task that is the prerequisite for something, its *LF* is the minimum *LS* of the tasks it's a prerequisite for
- For each task, *LS* = *LF* - *D*
- Using these relationships, we can fill in the *LF* and *LS* for each task, starting from those that aren't the prerequisites for anything and working through the rest of the graph

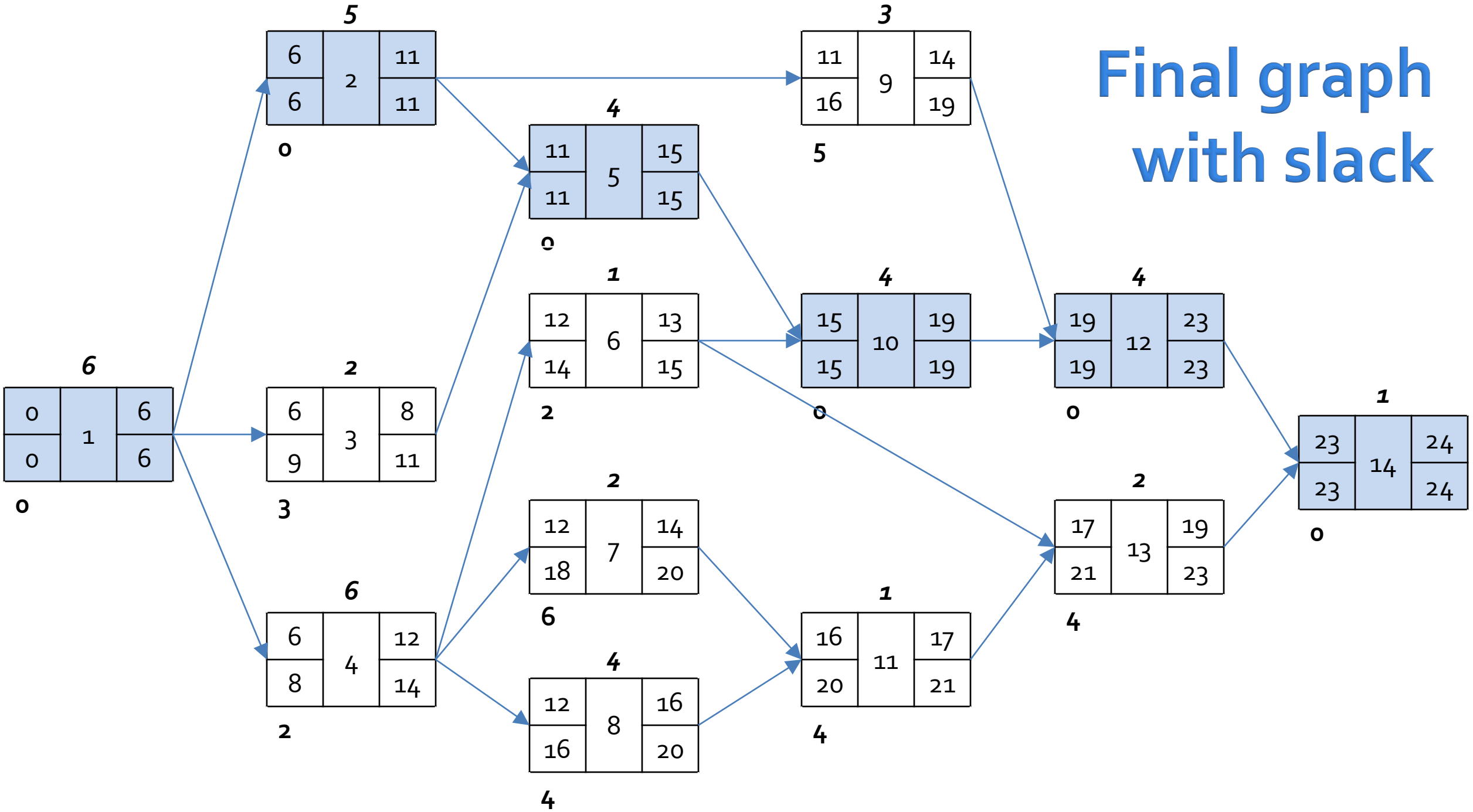
Graph with all start and finish times



Slack

- For each task, the slack time $S = LF - EF$
- We can run through the graph and mark that as well
- Nodes with no slack are on the critical path

Final graph with slack



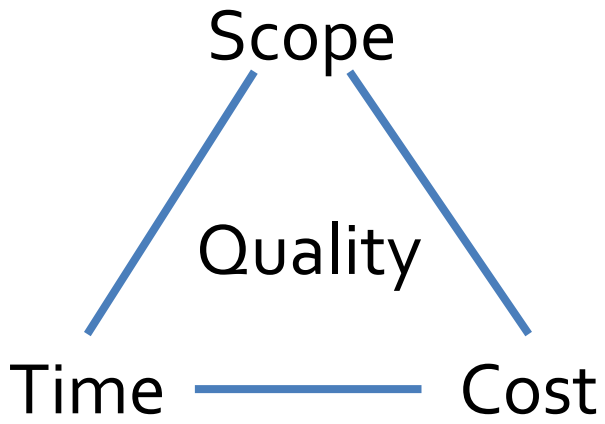
Execution and Control

Control in traditional projects

- Managers have to track progress
  - If a task is done early (sometimes it happens!), the project is ahead of schedule
  - If a task is done late, the project is behind schedule
  - Similarly, cost might be above or below expectations
- If progress doesn't meet expectations, actions must be taken:
  - Reduce the features but stay on schedule
  - Add more resources (people) but stay on schedule
  - Delay the delivery date, which probably also increases costs
  - Reduce costs by shrinking the scope, firing people or shortening the delivery date or both

Back to the 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

Brooks' Law

- Fred Brooks is a Turing Award-winning computer scientist who wrote *The Mythical Man-Month*, a book about software engineering
- **Brooks' Law:** "Adding programmers to a late project makes it later."
  - New people have to be trained on the project by existing employees
  - This effect gets worse with projects that are close to done since there's more to learn
- There are also lower bounds on how fast a project can get done no matter how many people you throw at it
  - Some tasks can only be done effectively by a single person
- Presumably, there's an ideal team size for a given project, but we do not (yet) know how to estimate it

Consulting with stakeholders

- Stakeholders get mad when bad news is sprung on them at the last minute
  - Product will be delayed
  - Product won't do what it's expected to do
  - Product will cost more than expected
- When problems arise, managers should consult with stakeholders to see how they want to proceed
- People prefer having input into the response to a bad situation

Earned value management

- If a job is expected to take 100 hours, and you've worked for 50, are you halfway done?
  - Probably not!
- **Earned value management** (**EVM**) (or **earned value analysis** (**EVA**)) tries to solve this problem
  - **Progress** is how much of the overall project is complete
  - **Health** is a comparison of how much you thought you'd get done with how much you did get done
  - Value can be measured in person-days or in dollars

More on EVM

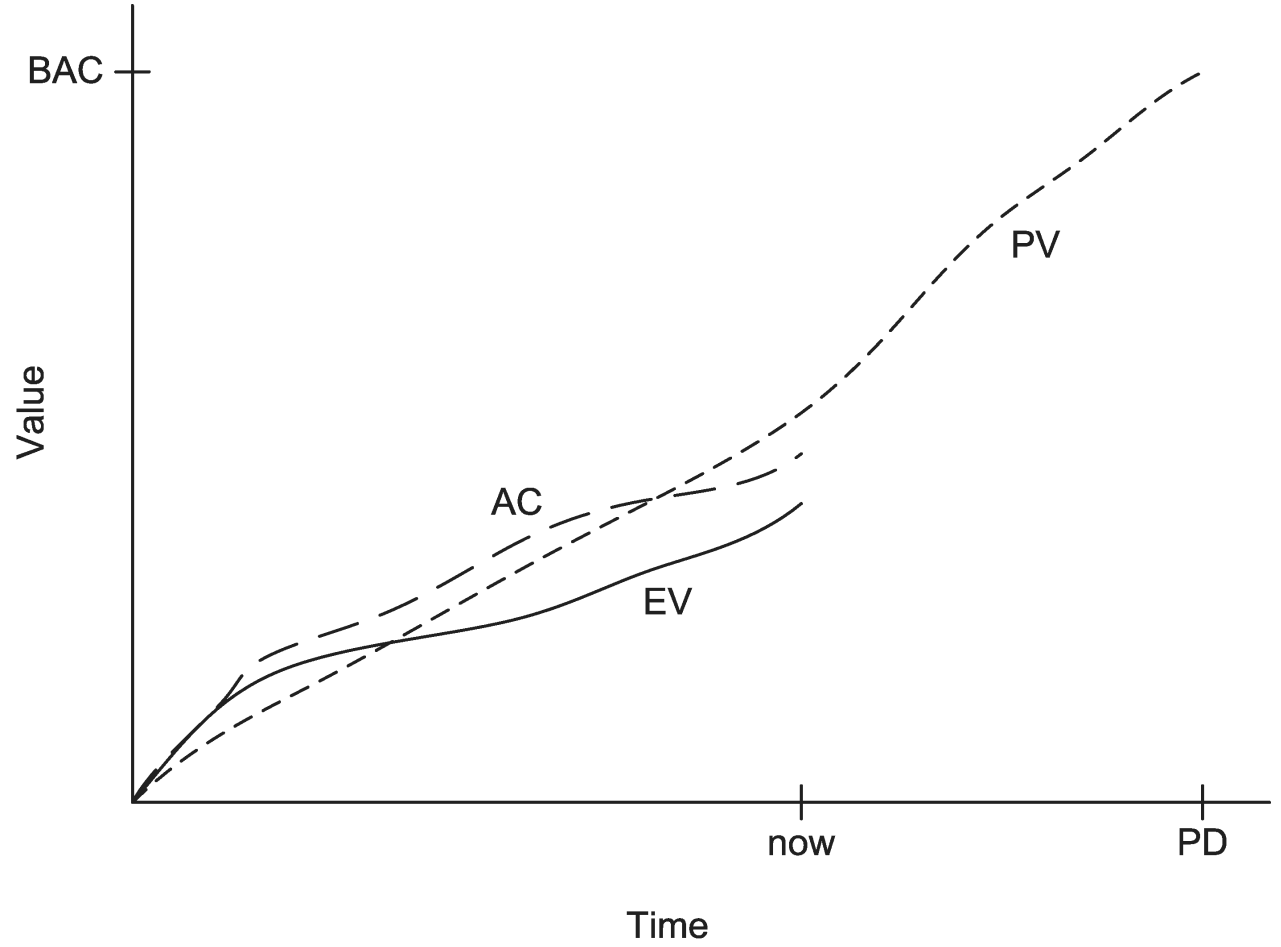
- We have talked about ways to decompose a project into tasks and how to estimate the effort or cost of each task (even if that's still a hard problem)
  - **Planned value** (**PV**) is the estimated cost of a given task
- Using a Gantt chart or CPM, you can make a schedule for all of your tasks
  - **Planned duration** (**PD**) is the estimated time for the entire project
- With the PV for every task and a schedule, you can graph the growth of PV over time
- This line ends at the **PD**, giving the **Budget At Completion** (**BAC**), the estimated cost of the whole project

Earned value and actual cost

- We can map out PV before the project is really underway
- While the project is going, we're interested in two more values for each point in time:
  - **Earned value** (**EV**): The planned value of the tasks that are done
  - **Actual cost** (**AC**): The effort or money spent on getting those tasks done
- It seems strange that we can have *three* different values for each point in time, but what we plan to do differs from what we get done, and what we get done doesn't always cost what we think it will

Earned value management with PV, EV, and AC

- The graph below shows an example of relationships between PV, EV, and AC



Quantifying project problems

- Where the EV is relative to the PV shows how much has been gotten done relative to what was planned
  - If the EV line is below the PV, the project is behind
  - If the EV line is above the PV, the project is ahead
- Where the AC is relative to the EV shows how much value has been expended relative to how much value was completed
  - If the AC is above the EV, the project is over budget
  - If the AC is below the EV, the project is under budget

More on quantifying project problems

- The **Schedule Performance Index** (**SPI**) is EV/PV
  - When SPI = 1, the project is right on schedule
  - When SPI < 1, the project is behind schedule
  - When SPI > 1, the project is ahead of schedule
  - Example: Here, the SPI = 360/400 = 0.9, meaning behind schedule
- The **Cost Performance Index** (**CPI**) is EV/AC
  - When CPI = 1, the project is right on budget
  - When CPI < 1, the project is over budget
  - When CPI > 1, the project is under budget
  - Example: Here, the CPI = 360/378.94 = 0.95, meaning over budget

Even more on quantifying project problems

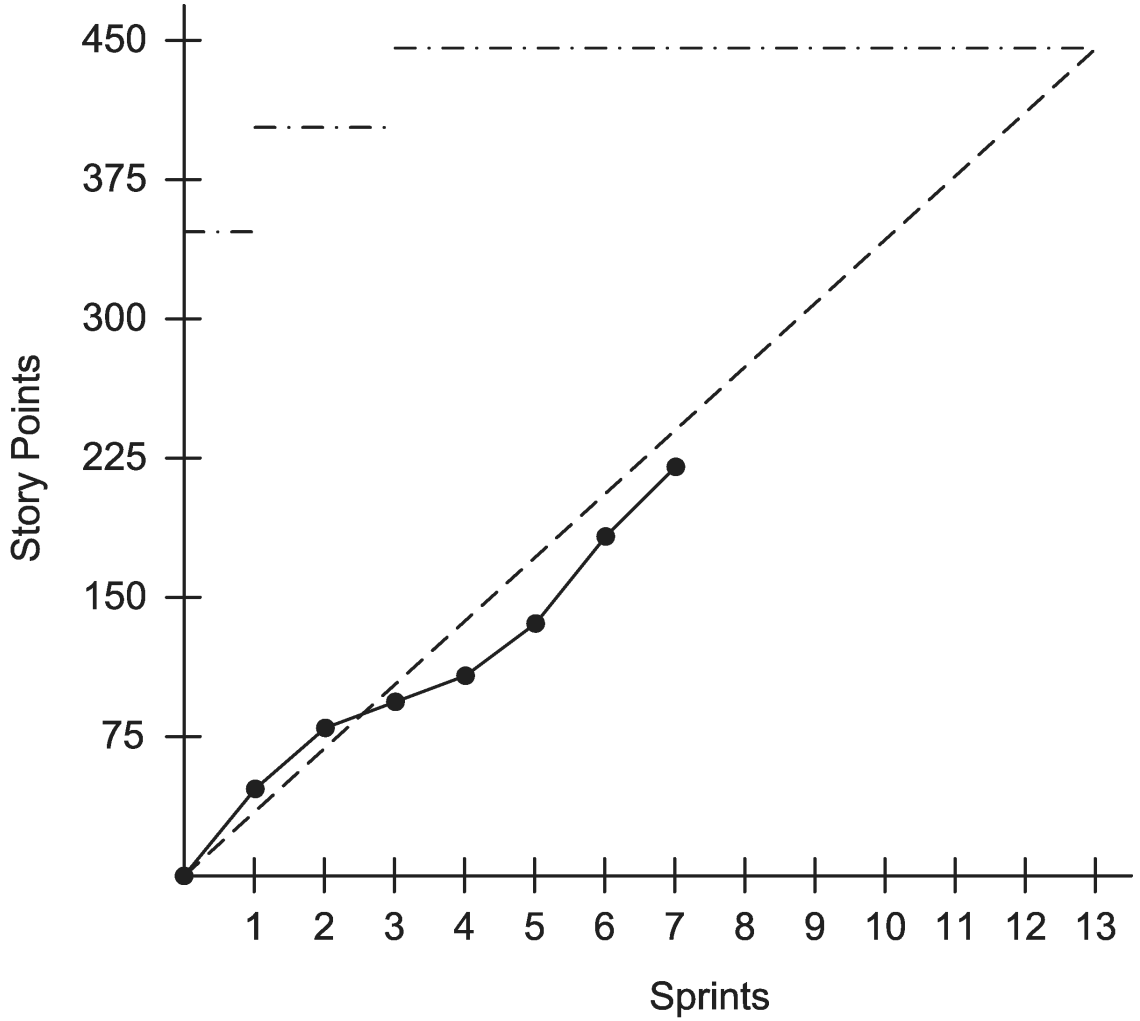
- Using these numbers, we can predict how well the project is doing
- **Forecast Project Duration** (**FPD**) is PD/SPI = PD/(EV/PV)
  - The FPD estimates the true project duration based on how far or ahead the project is at a given time
  - Example: In this case, assuming a planned duration of 30 months, FPD = 30/0.9 = 33.33 months
- **Estimate At Completion** (**EAC**) is BAC/CPI = BAC/(EV/AC)
  - The EAC estimates the true project cost based on how much it has cost to get tasks done so far
  - Example: In this case, assuming a budget at completion of 1140 (whatevers), EAC = 1140/0.95 = 1200
- Using these estimates and EVM charts, managers can make decisions about what to do when things are going wrong

Control in Scrum

- Everything revolves around sprints in Scrum
- We measure progress against the amount of work to be done using burn charts
- **Burn charts** have a vertical axis of work and a horizontal axis of time
  - **Burn up charts** show work completed and are more often used to track progress on full projects or releases
  - **Burn down charts** show work remaining and are more often used to track progress within a sprint

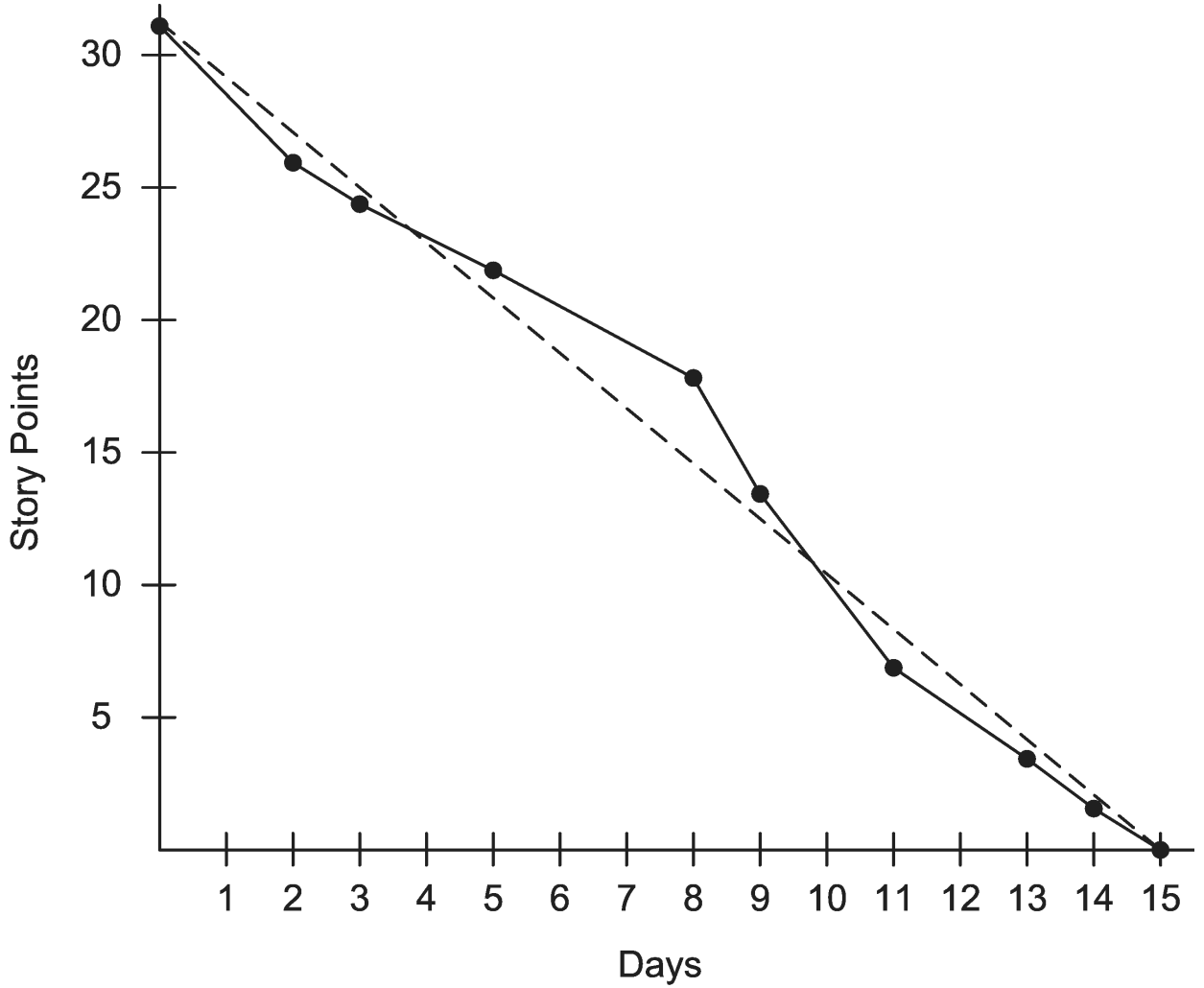
Burn up chart example

- This burn up chart shows progress over a whole project
  - 13 sprints are planned
- Solid black lines show real progress against planned progress
- The horizontal lines on the left show estimates of total story points needed for the project
  - They were revised up from about 350 to 450 over time
- Although velocity is improving, we'll probably have to add sprints or reduce features



Burn down chart example

- This burn down chart shows progress made during a three-week sprint
- Each dot shows the completion of a PBI
- As before, the solid line shows real progress against the dotted line of planned progress
- If all the story points had been finished early, the team could work with the PO to add more to the sprint backlog
- If some PBIs had not been completed, they go back to the product backlog



Task boards

- **Task boards** are boards showing the current status of all tasks for a sprint
- Tasks are often shown in rows corresponding to a PBI
  - Columns might be: To Do, In Progress, In Testing, Done, and similar
- Your Trello boards are approximations of the task boards people use in real development
- Tasks boards are not as analytical as burn charts, but they can help in other ways:
  - Too many things in the To Do column late in a sprint means that PBIs aren't all going to get done
  - Something stuck in the In Progress column for a while means that more developers should help with it
  - If it looks like some PBIs are impossible to finish this sprint, the team can focus on tasks to get PBIs done that are possible

Quiz

Upcoming

Next time…

- **Project 4 is due on Friday!**
  - You have until 11:59 p.m. to turn in all your code
  - But you have to demo your project online in class
- **Assignment 4 is also due Friday!**

Reminders

- **Fill out course evaluations!**
- **Finish Project 4**
- **Don't forget Assignment 4**
- **Final Exam:**
  - Next Friday, 12/13/2024
  - 2:45 - 4:45 p.m.