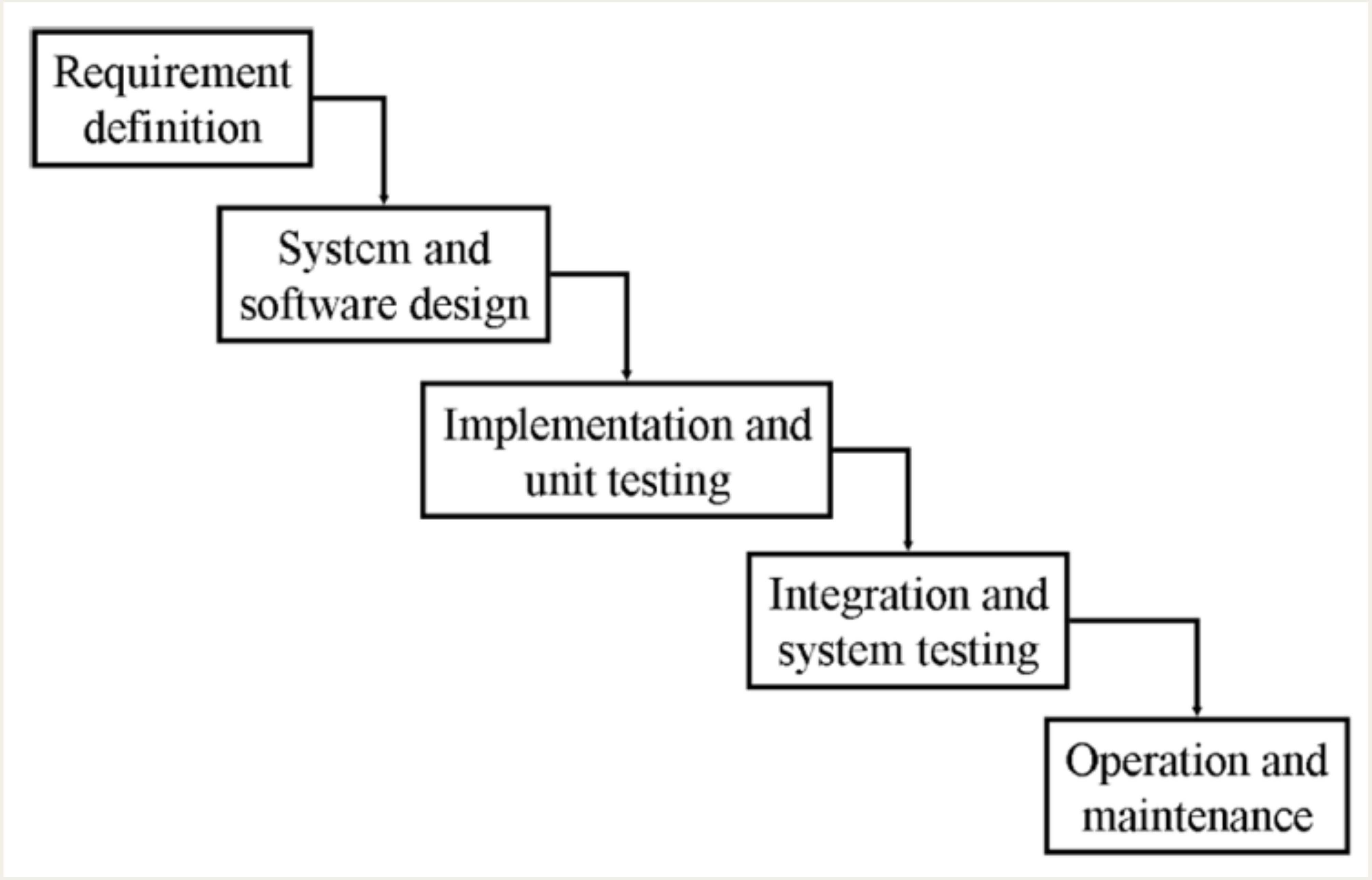
continuing from last time….

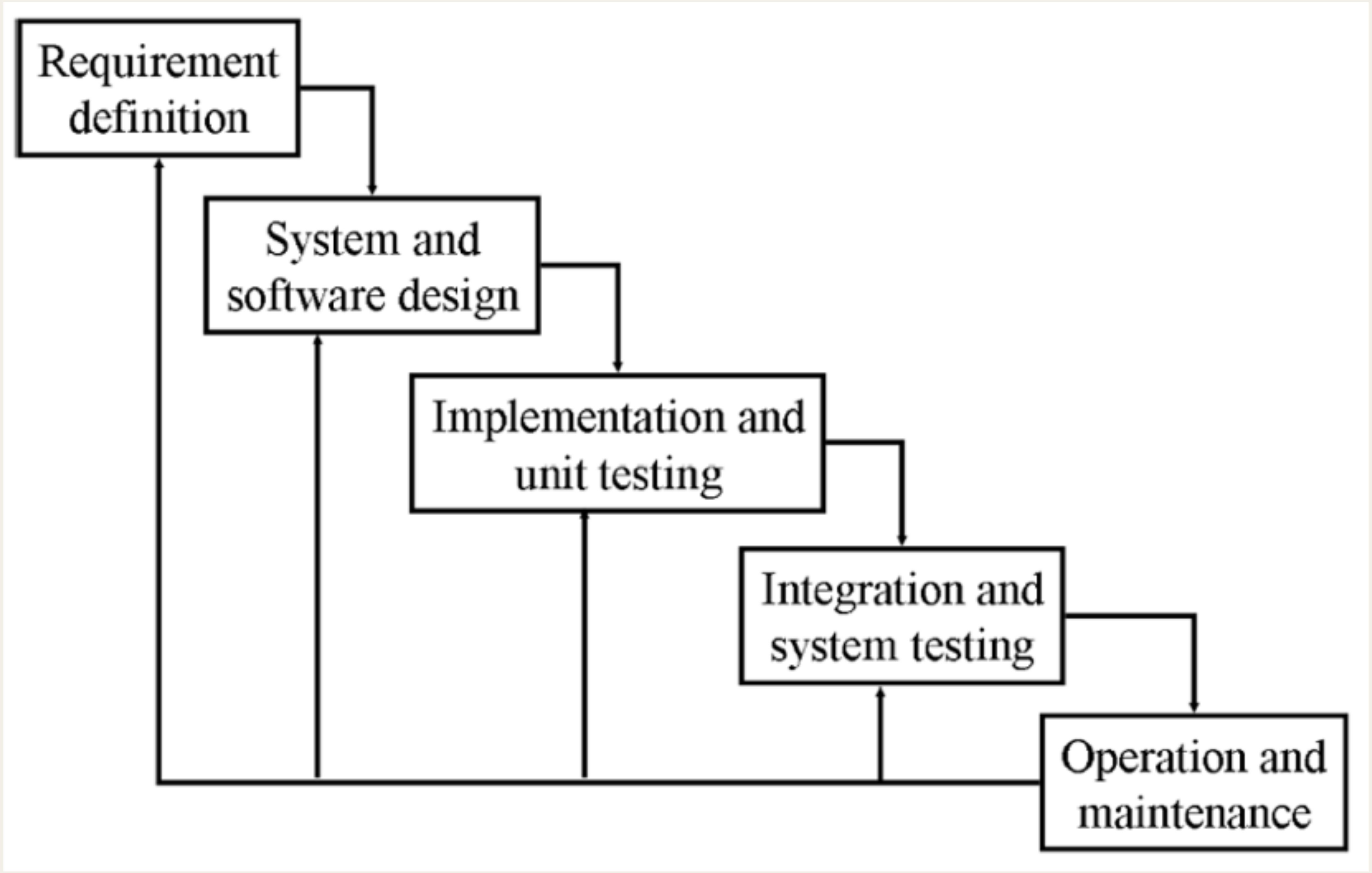
Software Process Models

- Waterfall Model
  - Breaks down a project based on activities: requirements analysis, design, coding, and testing.
  - A 1-year project might have a 2-month analysis phase, 4-month design phase, 3-month coding phase, and 3-month testing phase.
- Incremental / Iterative Development Model
  - Breaks down a project by subsets of functionality.
  - A 1-year project might be broken down into 3-month iterations. In the first, you take a quarter of the requirements and do the complete software life cycle for that 1st quarter: analysis, design, code, and test. Then, you have a complete system that works for a quarter of the needed functionality. In the next iteration, add the 2nd quarter of the functionality, etc..

The Waterfall Model


Requirement definition
System and software design
Implementation and unit testing
Integration and system testing
Operation and maintenance

The Waterfall Model with Feedback


Requirement definition
System and software design
Implementation and unit testing
Integration and system testing
Operation and maintenance

Limitations of the Waterfall Method

- It’s very difficult to tell if the project is really on track.
- The difficulty of accommodating change once the process is underway.
- You seldom know the requirements that early.
- Difficult to evaluate the risk of the project and/or identify the most risky aspects of the project.
- Testing and integration are the hardest activities to estimate, so it’s difficult to support these at the end of the process.

More on the Waterfall Method

- The waterfall model is still the most widely used deliverable-based model.
- Best applied to:
  - projects where the requirements are very well known, e.g. the team has experience in the particular domain.
  - projects that are low risk.

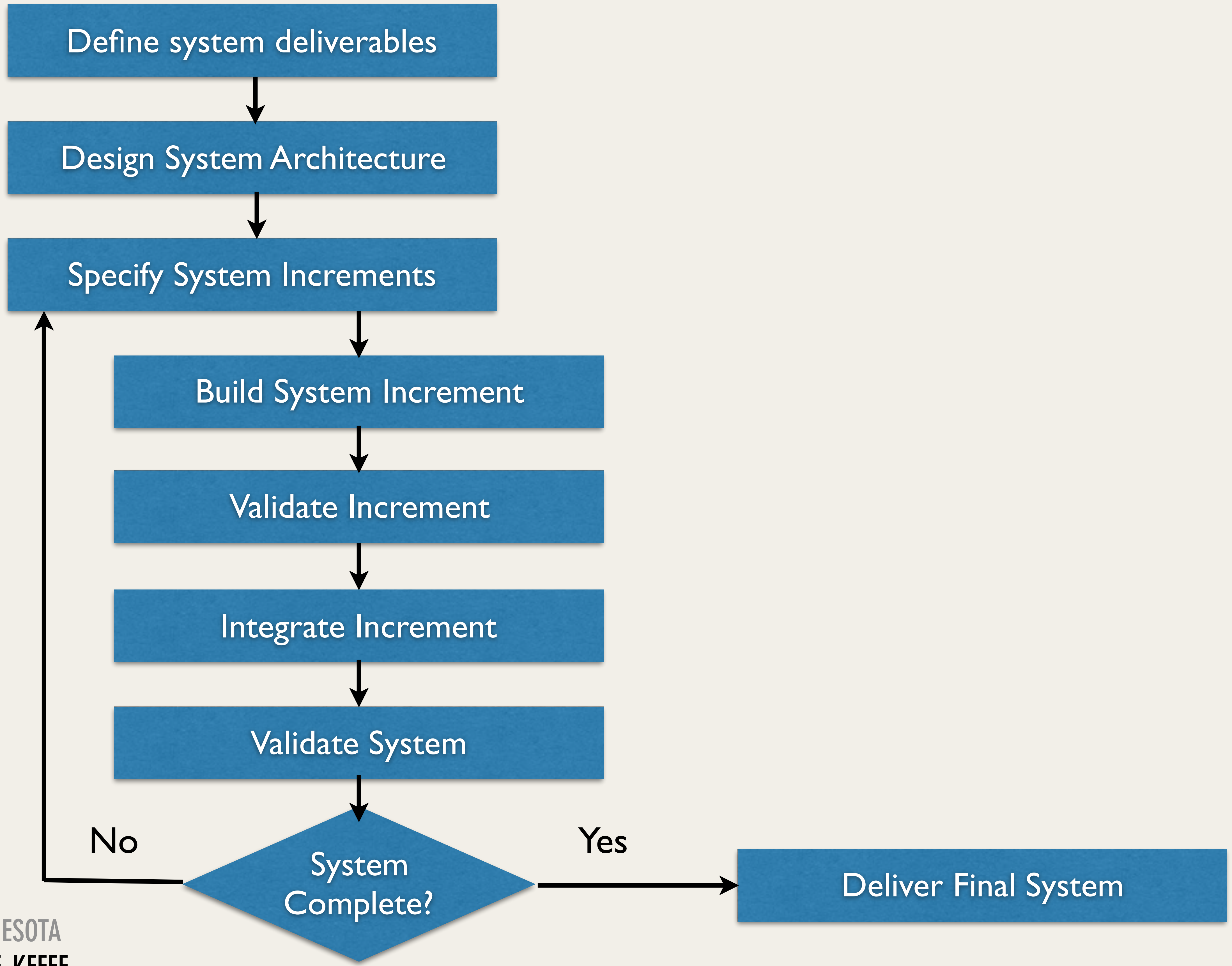
Iterative Development

- Is this iterative development (Fowler)?
  - “We are doing one analysis iteration followed by two design iterations...”
  - “The first iteration includes some bugs in the code that will be cleaned up in the second iteration.”
- No!

Iterative Development

- System is developed and delivered in increments after establishing an overall architecture.
- Users may experiment with delivered increments while others are being developed.
- These serve as a form of prototype system.

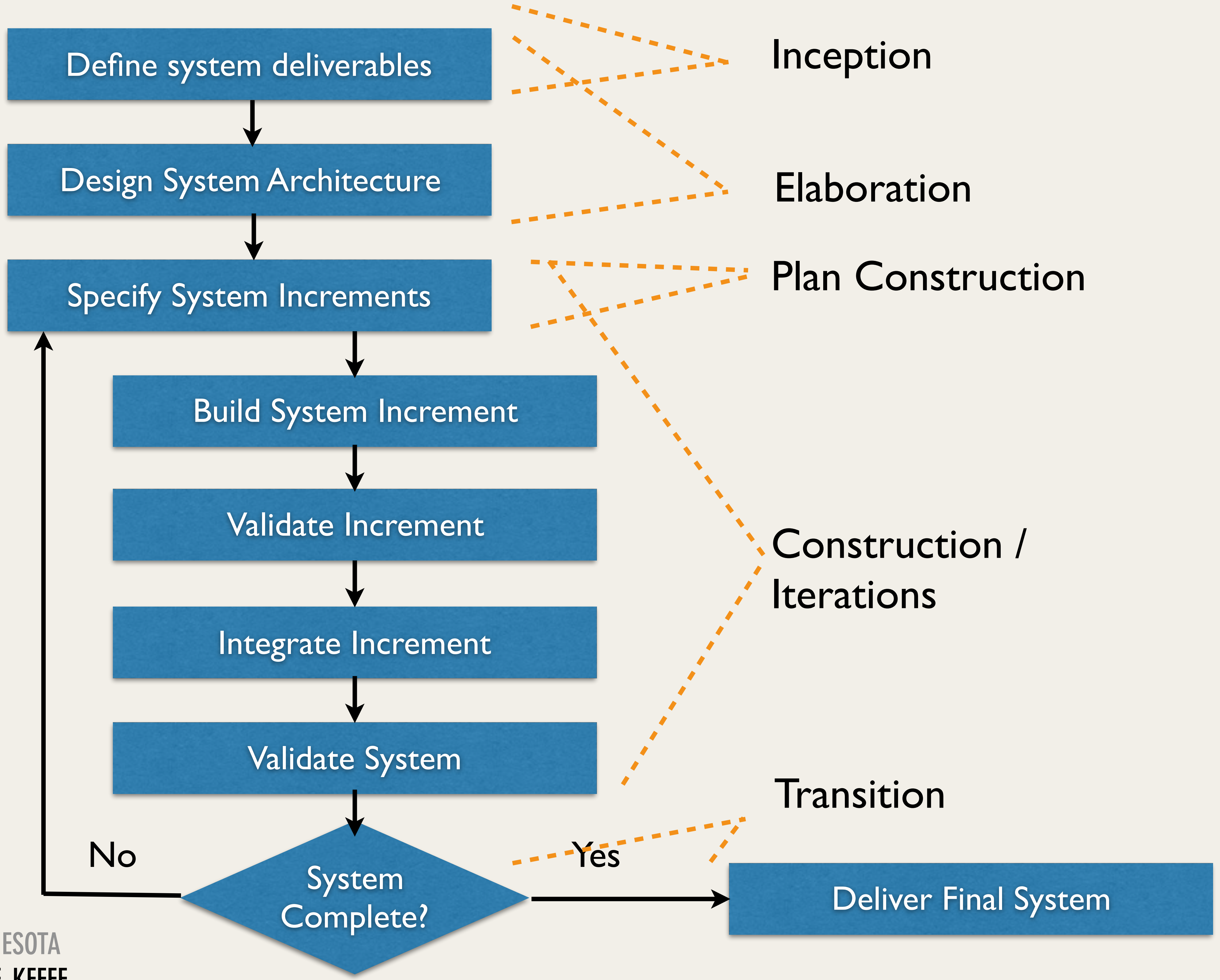
Iterative Development



Iterative Development -- Process Overview

- **Inception**
  - creation of the basic idea we want to implement, could be via discussion, could be a full fledged feasibility study.
  - outcome: project scope and business case.
- **Elaboration**
  - What is it you are going to build?
  - How are you going to build it?
  - What technology are you going to use?
  - Risk assessment
- **Plan Construction Iterations**
  - categorize use cases, “I must have this function”, “This is important, but I can live without it”, etc.
  - make time estimates and allocate the use cases to iterations
- **Construction**
  - each iteration is a “mini” project, a single iteration can follow the waterfall model: analyze, design, code, test, integrate.
- **Transition**: the phase between the beta release and the final product
  - performance evaluation and optimization
  - no new functionality, fix bugs.

Iterative Development



After Each Iteration:

- Code should be of near production quality.
- Should not have comments like, “This iteration’s code is buggy, but we’ll clean it up in the next iteration.”
- Many iterations are shown to the customer to get feedback -- called releases.

Iterative vs. Waterfall

- Waterfall skeptics point out it leaves two difficult and hard to predict activities to the end: system integration and system testing.
- The iterative model spreads these out across the entire development process.
- If each iteration is near production quality, then integration and testing will have been done properly.

Time Boxing

- Commonly used in iterative development.
- Fixes the amount of time allowed for each iteration.
- If planned features can’t be included, push them to another iteration.
- Forced to choose between slipping functionality and slipping release date.

Rework in Iterative Development

- Integrating the latest increment into the system often involves changing existing code.
- Isn’t this wasteful?
- Some of this work can be reduced by:
  - Automated regression testing.
  - Refactoring tools -- semi-automatic tools can often help with changes that improve readability/organization while preserving the interface to a program.
  - Continuous integration -- nightly builds.

Our Course

- We’re following an iterative model. (Sort of.)

Design Patterns: The Template Method Pattern

CSCI-3081: Program Design and Development

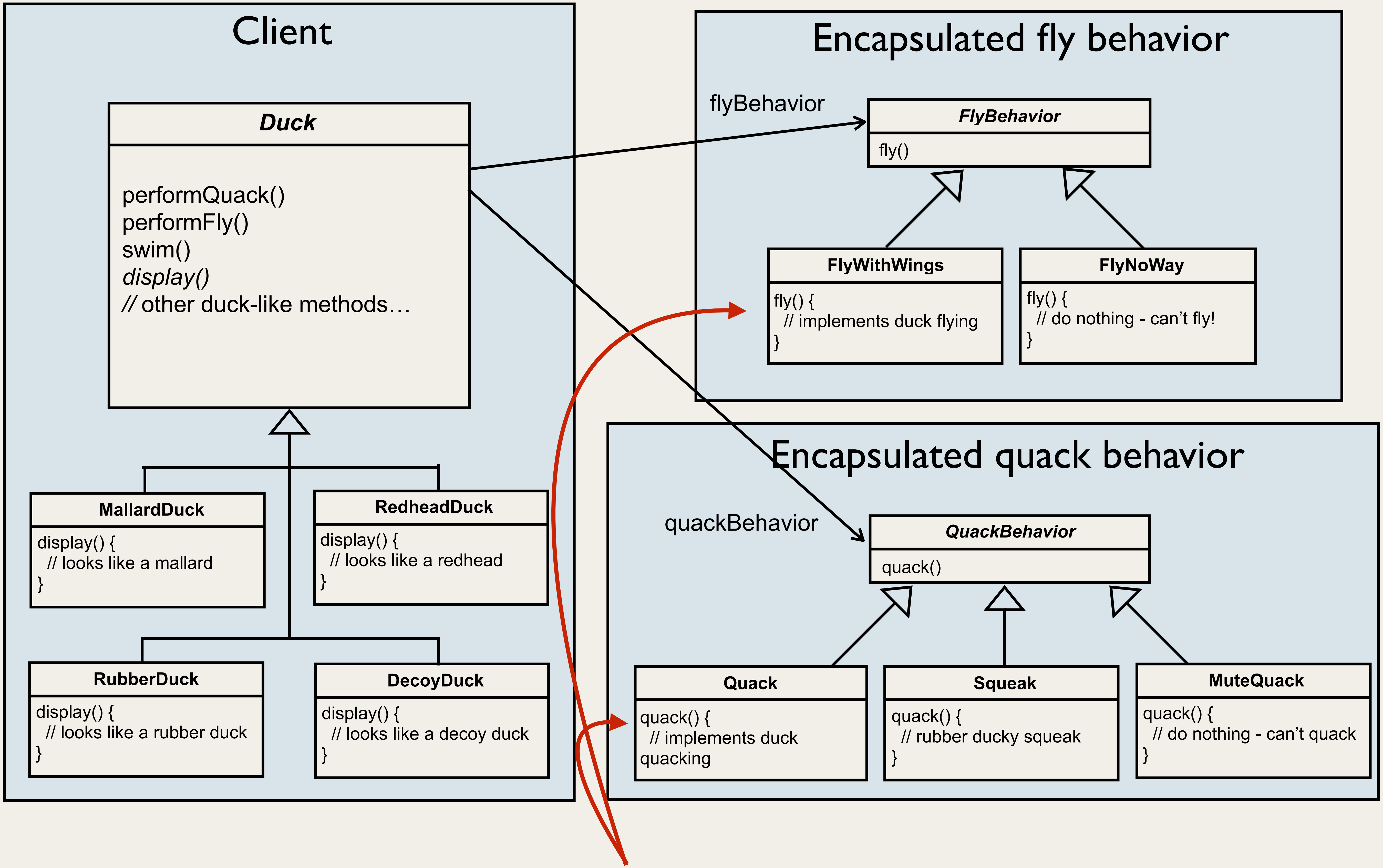
(Slides adapted from Head First Design Patterns)

Today

- Let’s return to the topic of good software design and add some Design Patterns to our toolbox.
- This builds upon the two design patterns we have studied so far…

Review #1: The Strategy Pattern

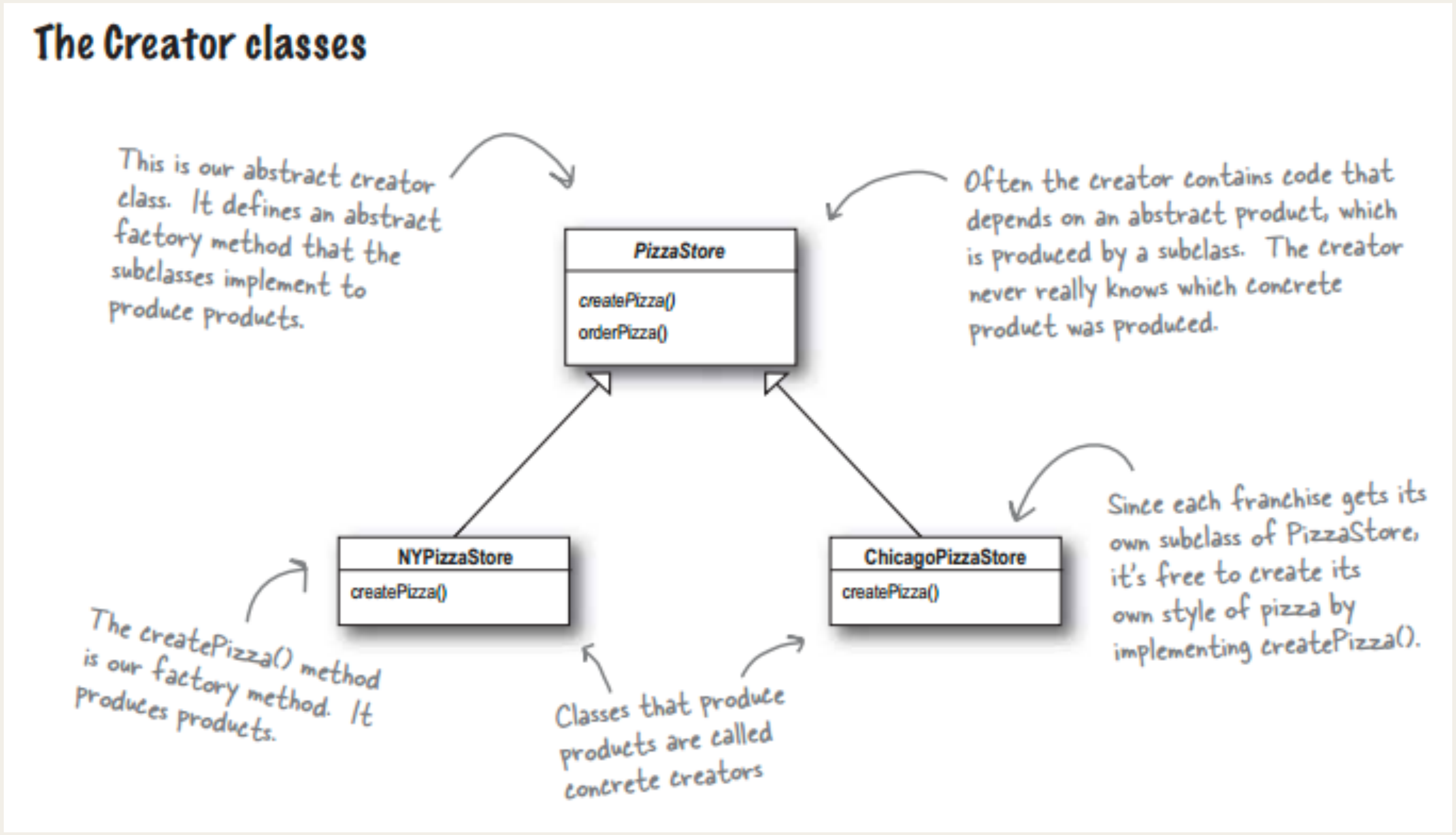
- **The Strategy Pattern** defines a family of algorithms, encapsulates each one, and makes them interchangeable.



Think of these as a family of interchangeable algorithms! Note the "has a" relationship - Duck has a FlyBehavior and Duck has a QuackBehavior. This uses composition rather than inheritance to swap in different algorithms.

Review #2: The Factory Method Pattern

- **The Factory Method Pattern** defines an interface for creating an object, but lets subclasses decide which class to instantiate.


The Creator classes
This is our abstract creator class. It defines an abstract factory method that the subclasses implement to produce products.
PizzaStore
createPizza()
orderPizza()
Often the creator contains code that depends on an abstract product, which is produced by a subclass. The creator never really knows which concrete product was produced.
NYPizzaStore
createPizza()
ChicagoPizzaStore
createPizza()
Since each franchise gets its own subclass of PizzaStore, it's free to create its own style of pizza by implementing createPizza().
The createPizza() method is our factory method. It produces products.
Classes that produce products are called concrete creators

Today: A New Design Pattern

A Very Hypothetical Design Problem

Oops, I wouldn't want to give anything away!

- Let's say you have an ~~applyToCanvas()~~ algorithm that will be repeated frequently in your program.
- ~~Each of your tools needs to use its mask to apply itself to the canvas~~, and many of the steps in the ~~applyToCanvas()~~ algorithm are the same for each object that calls the algorithm, so it would be great if we can reuse these.
- However, there are a couple of instances where some variation in the ~~applyToCanvas()~~ algorithm is needed.

A slightly different example:

Coffee Recipe

1. Boil some water
2. Brew coffee in boiling water
3. Pour coffee in cup
4. Add sugar and milk

Tea Recipe

1. Boil some water
2. Brew tea in boiling water
3. Pour tea in cup
4. Add lemon

A slightly different example:

We have some duplication here.
Let's move some functionality to a superclass.
Can you redesign using abstraction?
Draw a UML Class Diagram.

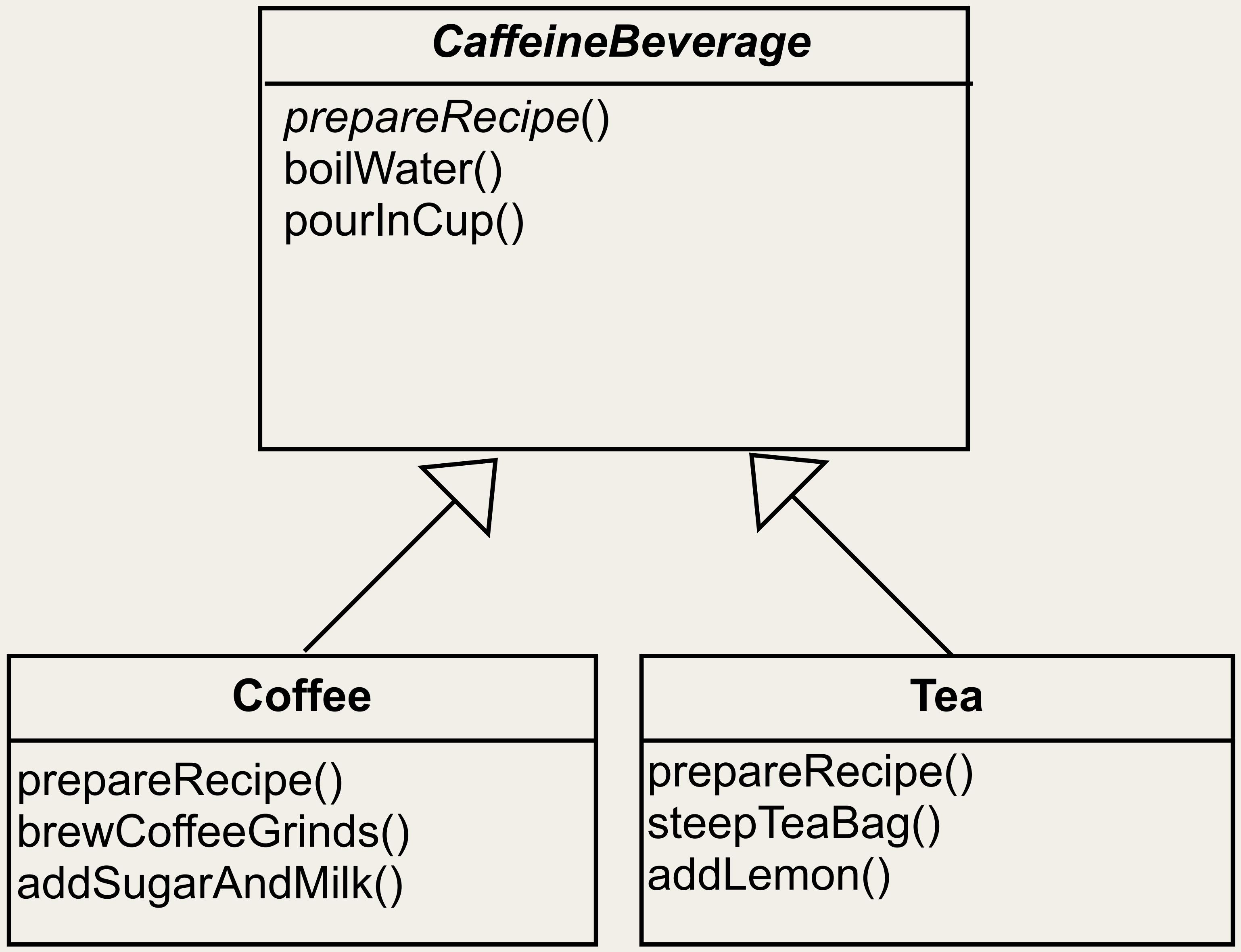
Coffee Recipe

```
class Coffee {
public:
  void prepareRecipe() {
    boilWater();
    brewCoffeeGrinds();
    pourInCup();
    addSugarAndMilk();
  }

  // more Coffee functions here…
};
```

Tea Recipe

```
class Tea {
public:
  void prepareRecipe() {
    boilWater();
    steepTeaBag();
    pourInCup();
    addLemon();
  }

  // more Tea functions here…
};
```

Good abstraction at the class level



Let's take it one step further…

What else is duplicated here? The Algorithm.

Coffee Recipe	Tea Recipe
1. Boil some water	1. Boil some water
2. Brew coffee in boiling water	2. Brew tea in boiling water
3. Pour coffee in cup	3. Pour tea in cup
4. Add sugar and milk	4. Add lemon

Write an abstract version of these algorithms that can go in the CaffeineBeverage superclass.

Coffee Recipe

```
class Coffee {
public:
  void prepareRecipe() {
    boilWater();
    brewCoffeeGrinds();
    pourInCup();
    addSugarAndMilk();
  }

  // more Coffee functions here…
};
```

Tea Recipe

```
class Tea {
public:
  void prepareRecipe() {
    boilWater();
    steepTeaBag();
    pourInCup();
    addLemon();
  }

  // more Tea functions here…
};
```

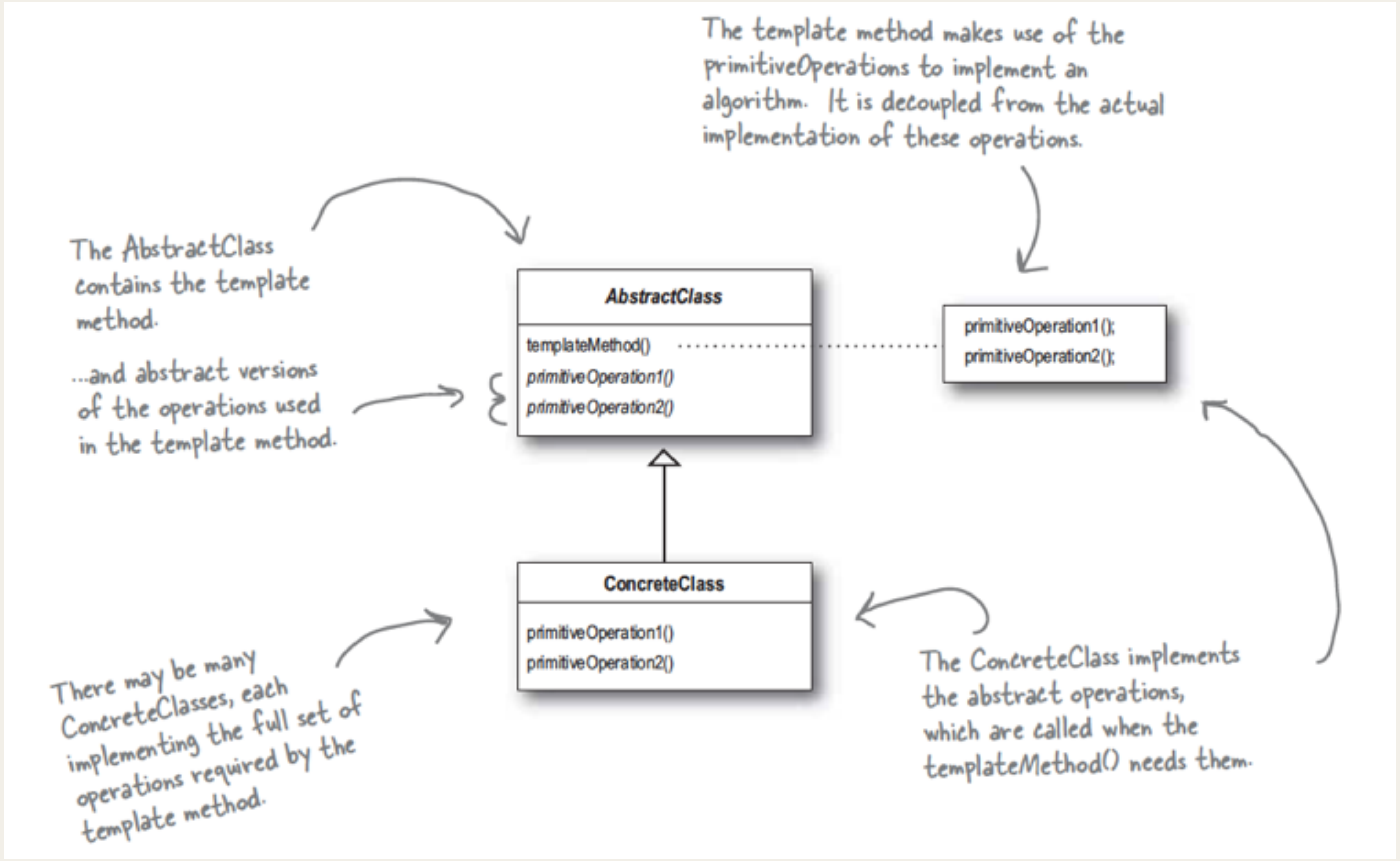
Meet the Template Method Design Pattern

```
class CaffeineBeverage {
public:
  void prepareRecipe() {

    boilWater();

    brew();

    pourInCup();

    addCondiments();
  }

  virtual void brew() = 0;

  virtual void addCondiments() = 0;

  void boilWater() {
    // implementation…
  }

  void pourInCup() {
    // implementation…
  }
};
```

- prepareRecipe() is a template method. Why?
- Each step of the algorithm is represented as a method.
- Some methods are handled by this class.
- And other methods are handled by the subclass.

Template Method Pattern

- **The Template Method Pattern** defines the skeleton of an algorithm in a method, deferring some steps to subclasses.

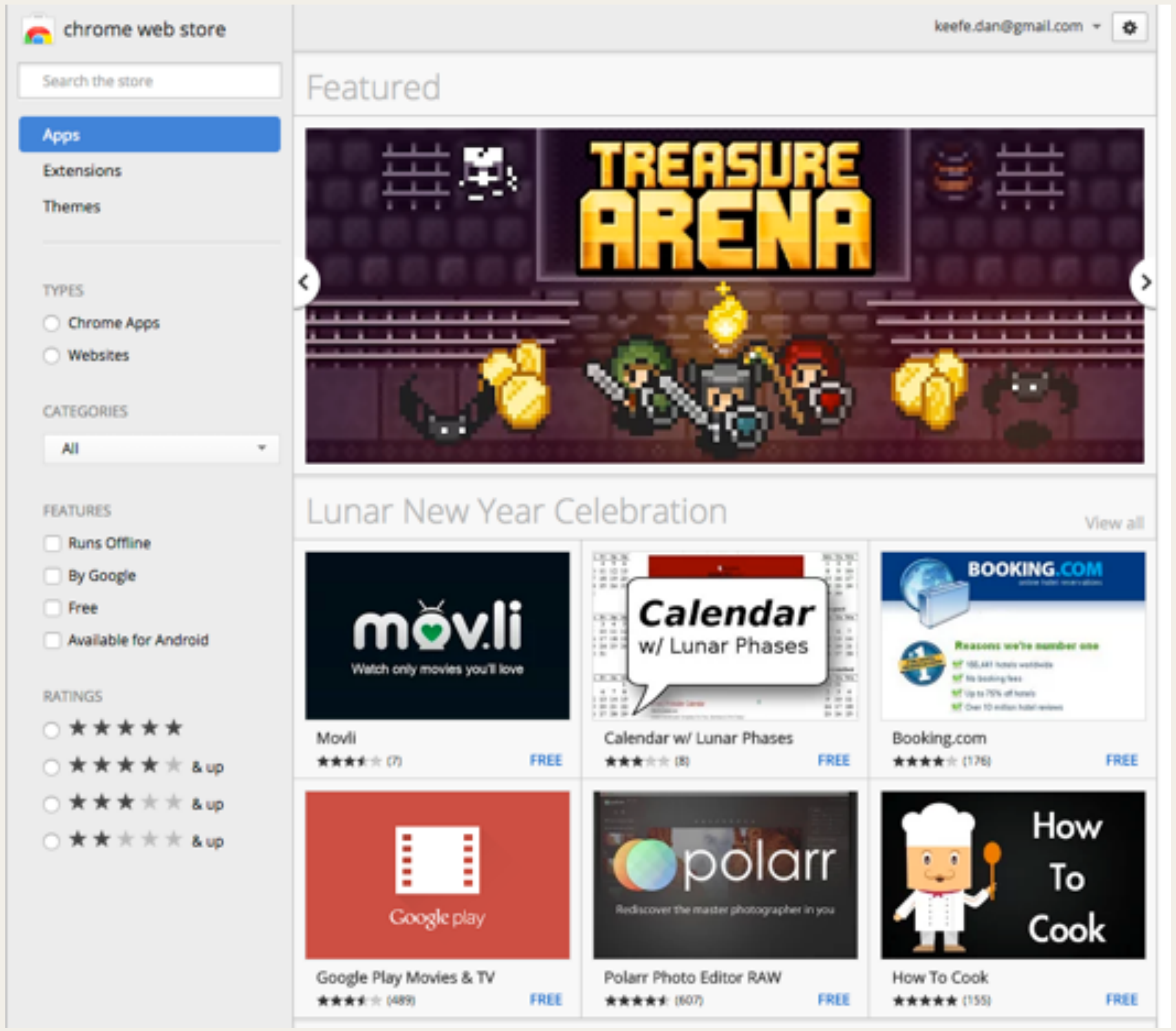

The template method makes use of the primitiveOperations to implement an algorithm. It is decoupled from the actual implementation of these operations.
The AbstractClass contains the template method.
...and abstract versions of the operations used in the template method.
AbstractClass
templateMethod()
primitiveOperation1()
primitiveOperation2()
primitiveOperation1();
primitiveOperation2();
ConcreteClass
primitiveOperation1()
primitiveOperation2()
There may be many ConcreteClasses, each implementing the full set of operations required by the template method.
The ConcreteClass implements the abstract operations, which are called when the templateMethod() needs them.

Where does this get used?

Ever sort anything?

- Let’s say we’re using mergeSort.
- We can write out a generic mergeSort algorithm that will work regardless of what type of object we are sorting (integers, strings, Ducks, Beverages).
- But, we’ll have to put a placeholder in for functionality that will change depending on the type of object.
- At some point in mergeSort() we need to compare two objects to see which one is greater, but “greater” will be interpreted differently depending on whether we are sorting integers or Ducks, so write this as a generic compareTo() method and make sure that Duck implements compareTo().

Other Examples



- Apps
- Applets
- Plugins
- …

Your Project Support Code

```
class BaseGfxApp {
public:

  // A template method for rendering one frame of computer
  // graphics, called repeatedly by the application's
  // main loop once each frame just after checking for
  // user input.
  void renderOneFrame() {
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    display();
    glutSwapBuffers();
  }

  // Intentionally empty so that subclasses have a hook
  // to display whatever they want
  virtual void display() {}

  // …
};
```

A Few Extensions and Notes

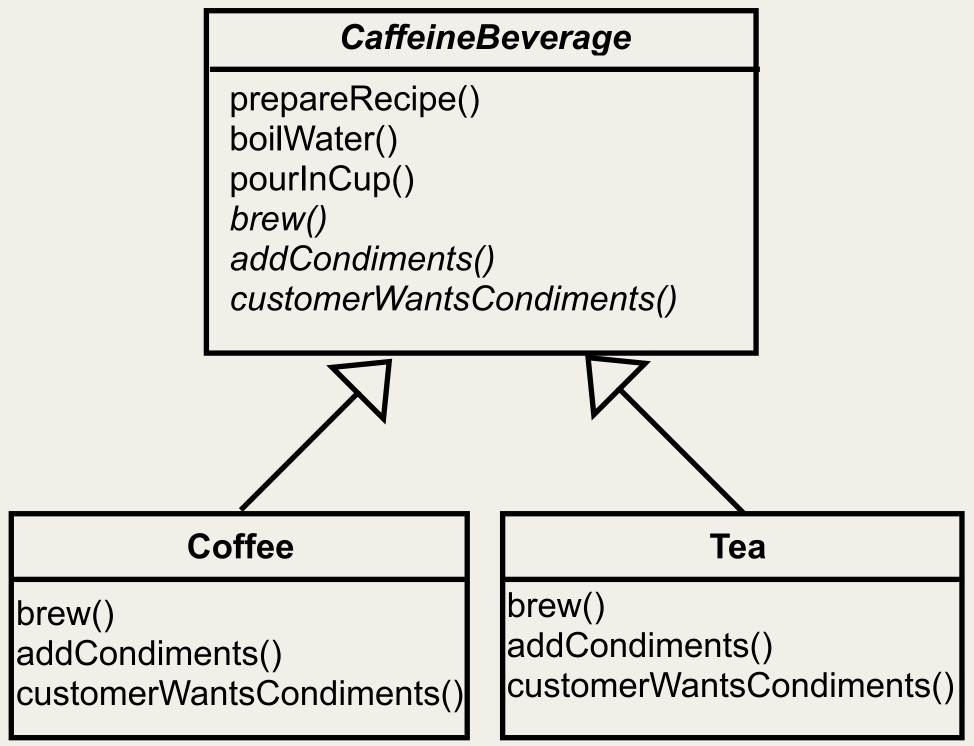
Hooks

- Methods with (at least mostly) empty default implementations.
- display() in the previous example is a hook with an empty implementation.
- This example has a default implementation but it is extremely simple.
- Subclasses don't have to override hooks, but they provide an opportunity to override if desired.

```
class CaffeineBeverage {
public:
  void prepareRecipe() {
    boilWater();
    brew();
    pourInCup();
    if (customerWantsCondiments()) {
      addCondiments();
    }
  }

  virtual void brew() = 0;

  virtual void addCondiments() = 0;

  void boilWater() {
    // implementation…
  }

  void pourInCup() {
    // implementation…
  }

  virtual bool customerWantsCondiments() {
    return true;
  }

};
```

hook

The superclass should be in charge.



- CaffeineBeverage has control over the whole algorithm.
- The subclasses simply provide implementation details as needed.
- As if CaffineBeverage said to Coffee and Tea, "Don't call us, we'll call you."

Project Questions?