1-10-2018

Software Engineering Lecture Notes (Student Version)

Kyle Burke
Plymouth State University, paithanq@gmail.com

CS 4140: Software Engineering*

Lecture Notes - Student Version†

Kyle Burke

January 10, 2018

"Weeks of programming can save you hours of planning." -unknown

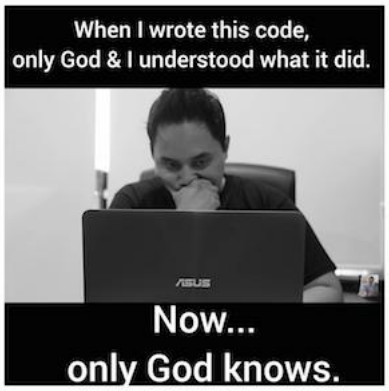
Abstract

Lecture notes for an upper-level undergraduate software engineering course, with a strong focus on software design. Students taking this course should have already completed a data structures course. These notes are designed to be used with Dale Skrien's text *Object Oriented Design using Java* [3].

Contents

*Kyle would always like to hear about how useful his notes are. If you have any comments about these, please email him at `paithanq@gmail.com`.

†Created with lectureNotes.sty, which is available at: `http://turing.plymouth.edu/~kgb1013/lectureNotesLatexStyle.php` (or, GitHub: `https://github.com/paithan/LaTeX-LectureNotes`). Many or most of the answers to questions are hidden so that some of class will still be a challenge for students.



[1]

When I Don't Comment My Code

[1]Found on reddit: https://www.reddit.com/r/ProgrammerHumor/comments/66xzzc/when_i_dont_comment_my_code/.

Preface

Plan and Goals

These lecture notes are designed to be used for a course in conjunction with Dale Skrien's book, *Object Oriented Design using Java* [3]. With this influence, the course spends most of its time covering elements of software design.

Acknowledgements

I must thank all of my students who have taken this course[2]. They have influenced the order of topics I cover and will continue to have an impact for as long as I teach this course. I have learned more about good design from teaching these students than I did while taking courses. I am so lucky to have many students who are quick to tell me the changes I should make and what worked well for them. This feedback is always helpful. I specifically want to thank:

- Dang and Amanda, the first two students who somehow survived a course that I made far too hard. Thanks and sorry!
- Will, who introduced me to the Command Pattern.
- Ryan and Bob, who didn't even take the course from me, but who convinced me to introduce MVC early.
- ... I'll add more as I remember them!

I learned to care so greatly about good design from Dale Skrien[3]. He showed us glimpses of design patterns in his Data Structures course, just enough to prime us for the "experimental" software design course he was creating. Everyone who could signed up for Dale's new course—offered first in 2002—and it did not disappoint. There was so much to learn, and I didn't get to flex that muscle during much of grad school. Luckily, the torch continued burning, and I got to teach an upper-level software course right after graduating in 2009. Dale's lessons lasted for seven dormant years, but armed with his excellent (and then new) text, I jumped right in as though no time had passed. Thank you, Dale!

I just discovered the `tikz-uml`[4] LaTeXpackage and am beginning to add diagrams using that instead of LibreOffice. Thanks to Nicolas Kielbasiewicz and other developers of this cool package!

Thanks also to Christin Wixson for helping me make all the necessary changes to include this (and my other lecture notes) in Plymouth State's institutional repository[5].

Under Construction

There's still so much to do. I'm working hard to convert over my hand-written notes into this form. Here's a list of some of the things I have left to do:

- Convert the diagrams to TikZ-UML.

[2] Wittenberg CS 253 and CS 353 in addition to Plymouth State CS 4140.
[3] `http://www.cs.colby.edu/djskrien/`
[4] `http://perso.ensta-paristech.fr/~kielbasi/tikzuml/index.php?lang=en`
[5] `http://digitalcommons.plymouth.edu`

- Add tons of missing chapters/sections/etc.
- Add references to Dale's book sections from each section in here.
- Add a little tutorial on using TikZ-UML in case students want to use that for their class diagrams. (Who am I kidding? They won't do that...)
- Add references to A2A for the Parallel/Concurrent OODP sections.

0 Introduction

In this course, you will:

- Write lots of code.
- Highly consider good software design.

0.0 Project Teams

First step: choose programming teams. Considerations:

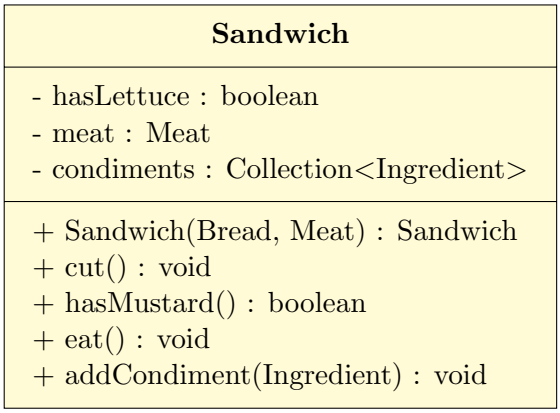
- Team members need to put equal effort into working on the projects.
- Team sizes: 2-3 people. Preference towards 2.
- Everyone must be on a team.
- Pair/Triple-Programming is most efficient. Find a team with a similar weekly schedule to yours.
- Large experience/skill gaps are dangerous. It's best to find teammates at your "level".

We need to choose teams immediately because you'll probably want to begin work on your project *today*.

⟨ Let students discuss with each other to determine teams. (Leave the room?) ⟩

0.1 UML: Class Diagrams

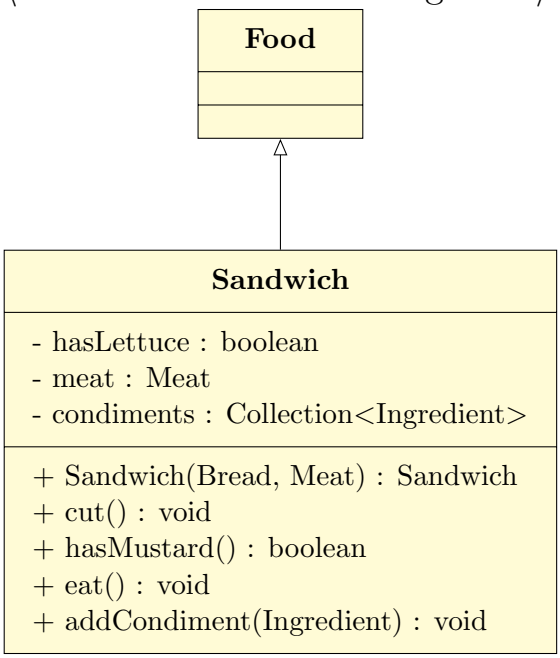
Often, easier to communicate and organize ideas in diagrams rather than code. UML (Unified Modelling Language) describes often-used diagrams. One part of UML is class diagrams. You will bring class diagrams to me each meeting.



Q: Would `Food` be a sub- or superclass of `Sandwich`?

A:

⟨ Include Food in the diagram: ⟩



Q: How are inheritance connections drawn?

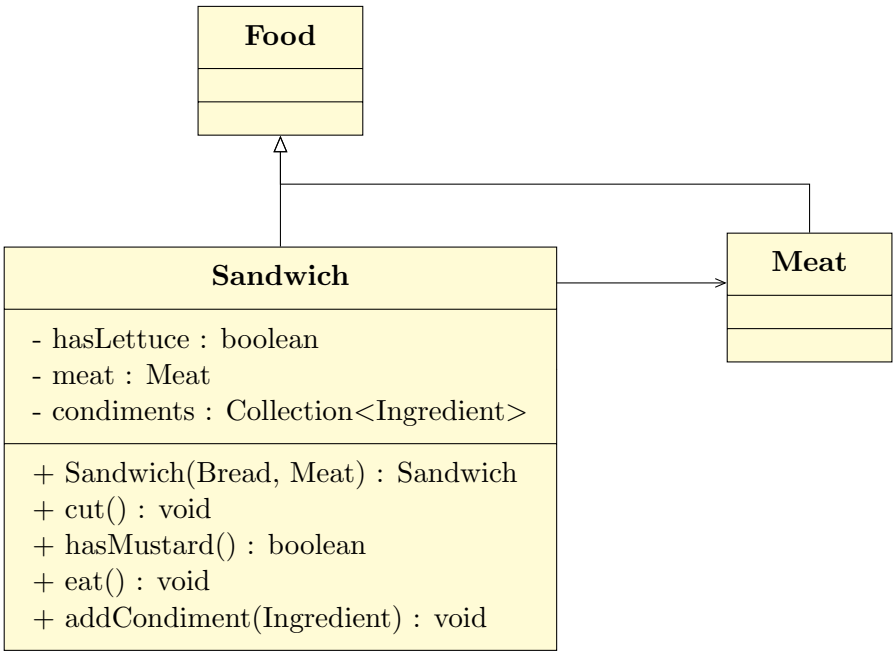
A: Big empty arrow pointing to the bottom of the superclass coming out of the top of the subclass. Implemented interfaces are the same, except that the edge is a dashed line.

Q: Which other types of connections come out of the tops and bottoms of class boxes?

A: None!

Q: What if we want to include Meat. How can we fit Meat into the diagram?

A:



Q:	Is it okay to combine the subclassing arrows for Food and Meat like in the drawing?
A:	Yes! In fact, you should only have one arrow coming in to the bottom of any class!

Q:	What does the arrow from Sandwich to Meat mean?
A:	It means that Sandwich has a field of type Meat.

Q:	What's up with the filled diamond?
A:	"Strong aggregation". The filled diamond means that Sandwich "owns" the field. Usually that means that Sandwich created it or it will be garbage collected because Sandwich disappears. It's definitely a subtle point.

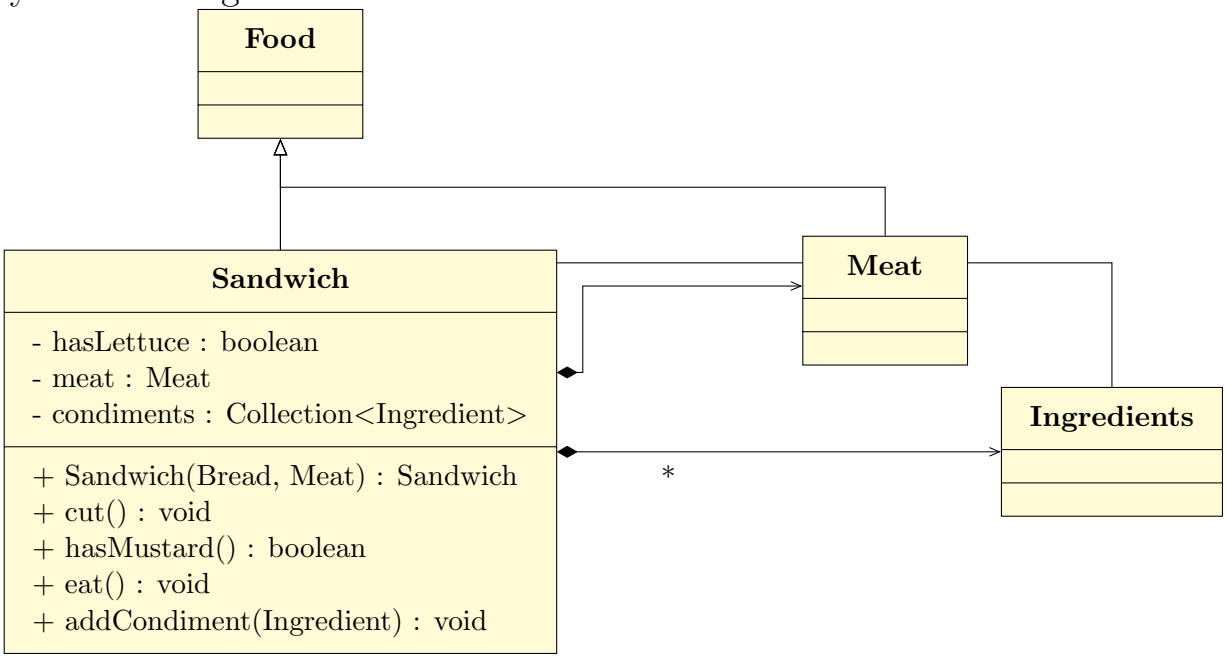
Q:	Wait... what's the other option?
A:	The diamond could be open. "Non-strong" aggregation. In this case, the object doesn't own it. I *do not* expect you to get all the diamonds right on your figures. In fact, you don't have to draw diamond in your figures.

Q: What is important about the positioning of the base and tip of the arrow?

A:

- Base: I like to put it at the field itself.
- Tip: I like to direct it to the head of the class (top box).

We won't worry about drawing the fields and/or methods of Meat at this point. If it's a class that you've designed, then you want to list all these things. If it's a built-in class (e.g. `JPanel`) then you only need to single box.



Q: What if there are multiple things?

A:

Q: Which other class should we add to the diagram that will use an asterisk?

A:

0.2 Bad Design: Repeated Code

Q: Let's say I'm new to a software project, and I'm looking at the code... and I see a line that could be causing an error. I fix it right then (probably not a good idea without talking to someone). What might I be worried about?

A:

Q: What if we're upgrading the code for a program, and we had a slew of integer variables that should only have positive values. We often had while loops that would decrement the variables, then afterwards:

```
if (x < 1) x = 1;
```

Now we want to allow zero to be a legal value. What is the biggest problem we have to face?

A:

Q: What's a common issue in these two situations?

A:

Q: What kind of other duplication do we need to be worried about throughout?

A:

Q: What if I am able to replace each of the tests on `x` with a method (function) call instead? What would be the benefit there?

A:

⟨ Check out this implementation for a Binary Tree: `http://turing.plymouth.edu/~kgb1013/DB/4140/binaryTreeExample/v0/BinaryIntTree.java` ⟩

Q: Does this code compile?

A:

Q: Does the unit test run?

A:

Q: Why not?

A:

Q: What do we have to do to fix it?

A:

⟨ Here's a version with the tests in place: `http://turing.plymouth.edu/~kgb1013/DB/4140/binaryTreeExample/v1/BinaryIntTree.java` ⟩

Q: What's a potential problem going forward with this code?

A:

This is not an immediately easy problem to solve. Notice: in each case, if the child is not null, I just call the appropriate method on that child. I'd like to call that method independent of whether the child is null.

Q: What happens if I call the method on a null object in Java?

A:

Q: How else could we represent not having a child? (Hard!)

A:

⟨ Check out the code here: `http://turing.plymouth.edu/~kgb1013/DB/4140/binaryTreeExample/v2/BinaryIntTree.java` ⟩

`null` is actually considered to be a mistake. Here's the quote from Tony Hoare, who was a developer of ALGOL back in the 1960's:

> "I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years."[6]

This is an example of a programming pattern that uses Polymorphism: the Null Object Pattern.

TODO: Should I introduce the Null Object Pattern in full here? (a la `http://www.cs.oberlin.edu/~jwalker/nullObjPattern/`) I could talk about them here and move all that stuff up...

[6]Source: `https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare`

⟨ Go over the syllabus! ⟩

0.3 JavaDoc

This is covered in the class book in Appendix B.

⟨ Describe JavaDoc comments: class headers, instance variable (constant) headers, public method headers, tags. ⟩

Q: Why do I want you to JavaDoc things for this class?

A:

0.4 Model-View-Controller Heuristic

⟨ Talk about old students helping me fix this course by suggesting topics. ⟩

One of these suggested topics is the Model-View-Controller paradigm. The idea to is identify and separate out these three parts:

- Model: The actual data for something (X).
- View: Handles the way X is displayed.
- Controller: Handles changes/input to the system.

The more you can separate these things, the better.

Q: However, there is one automatic view that I include in every model. What is that?

A:

MVC is a good way to start dividing up the different parts for a project. Some development tools (e.g. Spring framework) have you start with this.

- If they're using the game project you assigned them: "Note, I divided up the classes using MVC."
- TODO: add more cases here as/if you create more projects.

0.4.1 Example: Pokédex

In the Pokémon mythos, it is very useful to have your Pokédex while out hunting. It's like a little PDA that tells you about the Pokémon you encounter.

⟨ Draw a sketch of a Pokédex, like a game boy. ⟩

Q: If Dexter (common Pokédex name) is currently on a screen showing general information about Jigglypuffs, which part of MVC should be responsible for the information? (I.e., #39, Normal Type, "When its huge eyes light up, it sings a mysteriously soothing melody that lulls its enemies to sleep.")

A:

Q: What might we call this class?

A:

Q: What are some fields for this class?

A:

Q: What is responsible for presenting this data?

A:

Q: Why do we want the View and the Model to be disjoint? Why not use the same object?

A:

Q: What is a better term for classes being disjoint?

A:

Q: Some adversaries to MVC argue that every model should have a default view. Why?

A:

Q: Why might different views be good for Pokédex software?

A:

Q: Which MVC part will get involved if I push the Right-arrow, because I want to go to pokemon #40?

A:

Q: Fill in the following sequence of events from the Controller's perspective:

1. Press Right-Arrow
2. Controller "hears" the right-arrow-push
3. ?
4. ?
5. Controller tells the view to ...?

A:

Q: What are some of the fields for the Controller going to be in this instance?

A:

Remember that we want to decouple the different parts of MVC as much as possible. (It's very hard.)

Q: Which classes need to know about which other classes?

A:

Q: Is this how it always is? Hint: consider the project code.

A:

Q: Why doesn't the Controller need to know about the View?

A:

We'll see more ways of how to separate the different parts.

0.5 Second Intro

Three Problematic Expectations. You can't expect...

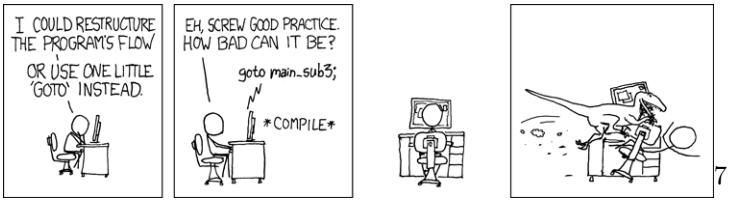
Past ...code to have been designed well.

Present ...to make the best choices the first time.

Future ...how your code will be used in the future. (Example: unsafe methods in one buried class.)

I expect you to overcome these with good design!

1 Basic Object-Oriented Programming



[7]

1.0 Elegance

Code Elegance! "Shivers of joy" vs "Shudders of revulsion."

Elegance is: (page 5) (list all out first)

- Usability: Is it easy for the client to use?
- Completeness: Does it satisfy all the client's needs?
- Robustness: will it deal with unusual situations gracefully and avoid crashing?
- Efficiency: will it perform the necessary computations with reasonable resources?
- Scalability: will it still perform when the problems grow by a lot?
- Readability: is it easy for other programmers to read the code?
- Reusability: can the code be reused in other settings?
- Simplicity: is the code unnecessarily complex?
- Maintainability: can defects be fixed easily without adding new defects?
- Extensibility: can new features easily be added/unwanted features easily be removed?

[7] XKCD comic 292: `https://xkcd.com/292/`

Q: Which are parts of Object-Oriented Design?

A:

Q: Which others are important for this class?

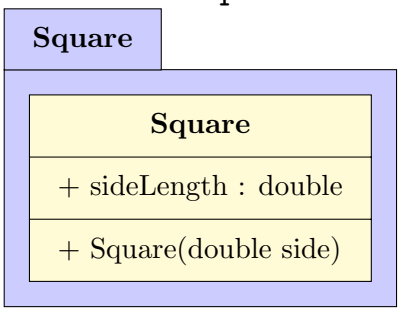
A:

Q: Improving which ones helps remove technical debt?

A:

1.1 Public vs Private Variables

Consider this `Square` class:



Q: Can I create an "illegal" square?

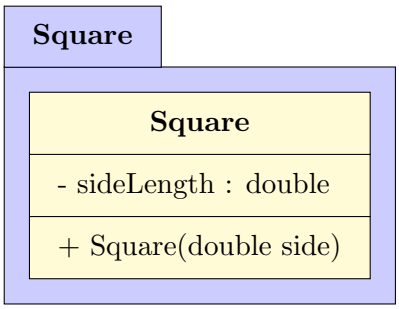
A:

Q: How?

A:

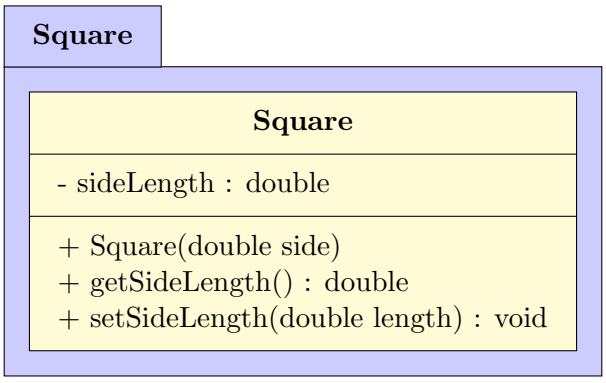
Q: What should we do instead of making them public?

A:



Q: What should I do if I want to access or change the value?

A:



Q: How could these setters prevent the problem with setting negative side lengths?

A:

```
public class Square {
   private double sideLength;

   public void setSideLength(double length) {
       if (length > 0) {
           this.sideLength = length;
       }
   }
...
}
```

Q: Is there a chance something unexpected could happen here?

A:

Q: What's a better solution?

A:

Q: Which kind of exception is most relevant here?

A:

Let's rewrite it using an exception!

```
public void setSideLength(double length) {
   if (length > 0) {
       this.sideLength = length;
   } else {
       throw new IllegalArgumentException(
"Square.setSideLength called with a non-positive
length!");
   }
}
```

Q: Where else might we need to guard against negative values?

A:

Q: How should we write the constructor?

A:

Q: How could we make this better?

A:

2 Observer Pattern

⟨ Draw a gameboy-based Pokédex with Control Pad, and buttons B and A. ⟩

Q: Recall our Pokédex example: What happened when we pressed right on the control pad?

A:

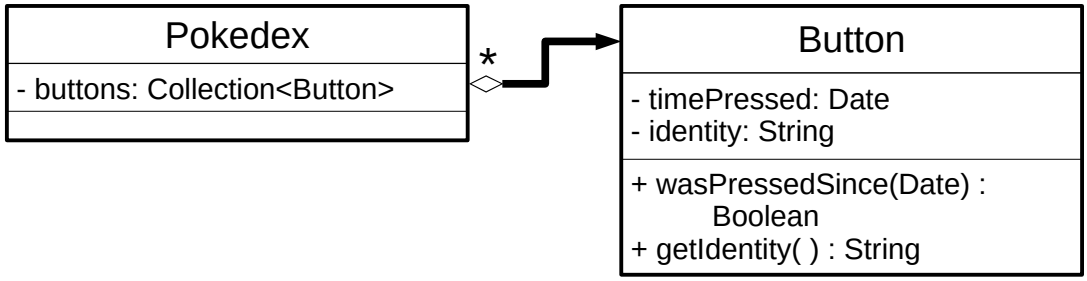
2.0 Code to Fix

Q: Consider the following buttons. What do they each do? (Assume we're currently looking at a single Pokémon.)

- Control Pad ←
- Control Pad ↑
- Control Pad ↓
- A
- B

A:

⟨ Draw this diagram:



⟩

Q: How does it work to determine whether a button was pressed?

A:

TODO: add some code in here?

2.1 Object-Oriented Design Pattern Basics

Each OO design pattern has some properties, including *participants* and *roles*.

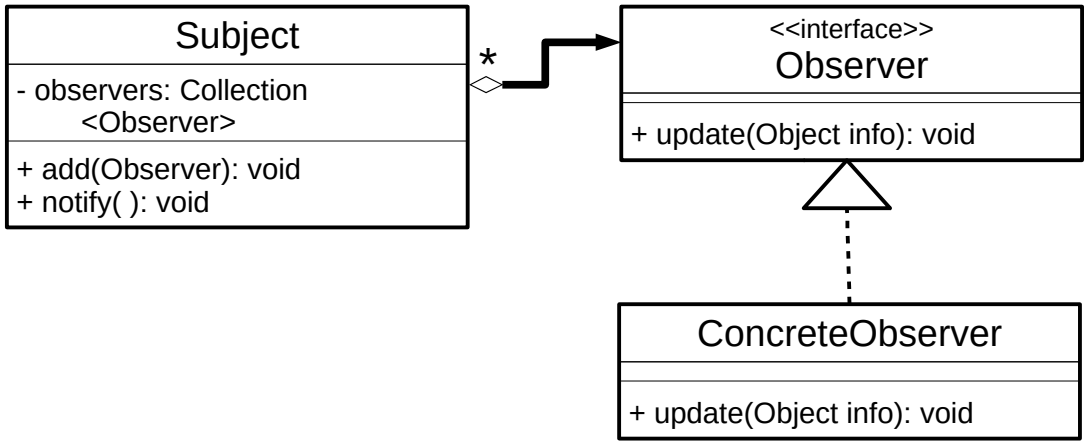
Q:	What are the participants in a design pattern?
A:	

Q:	What are the roles in a design pattern?
A:	

TODO: add more here?

2.2 Solution: Observer Pattern

The observer pattern is (another) object-oriented Design Pattern.
⟨ Draw diagram:



⟩

Q: What are the participants?

A:

Q: What field does the Subject need to have?

A:

Q: What is the role of the Subject? (What does it do?)

A:

Q: The Observer's role is to define the interface for concrete observers. What is the role of the Concrete Observer?

A:

Q: Why is it good to have the interface?

A:

Q: What will the `notify` method of the Subject look like? (We can add nearly all of that code.)

A:

Q: In Java, you might not have to implement the Observer interface yourself. Why not?

A:

Q: What's the Object that's sent as the parameter to Java's `ActionListener`?

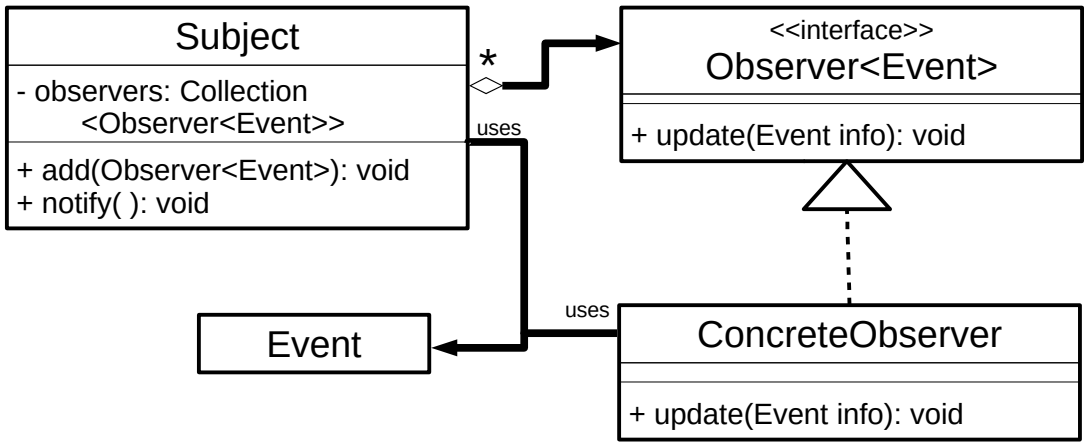
A:

For many of the events, you can get the object source of the event. (E.g. `ActionEvent.getSource()`)[9]

Q: How can we use generics in the Observer Pattern to make it more reusable?

A:

⟨ Draw this diagram:



⟩

[9]For a full list of Built-in Java Listeners, see `http://docs.oracle.com/javase/tutorial/uiswing/events/api.html`

2.3 Applying the Observer Pattern

Q: How do we apply the pattern to our Pokédex problem? For each Participant, which real class fills it?

A:

Q: Draw out the class diagram.

A:

3 When to Inherit?

We need to consider the question: When should you use inheritance? When should class B extend/implement/subclass class A?

⟨ Draw class diagram: B extends A with a question mark next to the extending arrow. ⟩

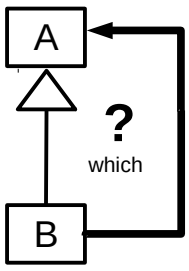
Q: Is it easy to misuse?

A:

Q: What is the alternative?

A:

⟨ Draw this figure:



⟩

There are four common reasons developers consider having B subclass A:

- Code Reuse
- "Is-A" perspective
- Matching (Public) Interfaces
- Polymorphism

Not all of them are good reasons.

Q: What do each of these mean? When is this a justification for using inheritance?

A:

3.0 Code Reuse

Justification reminder: "Lots of the fields in `A` and/or bodies of methods in `A` are what I want in `B`. I should reuse that code by having `B` inherit from `A`."

Code Reuse is good!

Q: Which mammal lays eggs and has a bill like a duck?

A:

Q: Should we have `Platypus` extend the `Duck` class so we can reuse the `layEgg` and `getBill` methods?

A:

Q: What if `Duck` also had the `molt` method? What would have to do in the `Platypus` class?

A:

Q: What would the body of `Platypus.molt` look like?

A:

Guideline: Platypus Rule[10]: A class inheriting methods should not nullify them or do something completely different or unexpected.

Q: Should `Platypus` extend `Duck`?

A:

[10] As named by Amelia Rowland, '18.

3.1 "Is-A" Perspective

Q: What was the justification we gave for the "is-a" perspective?

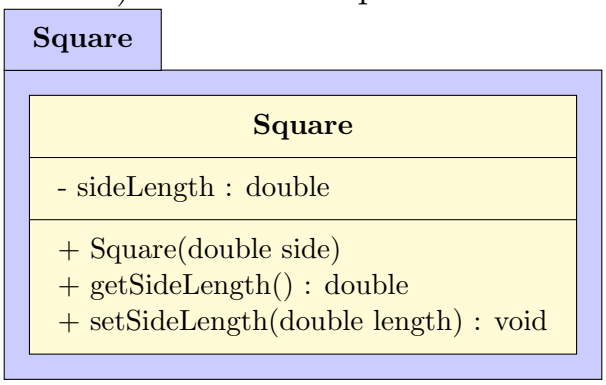
A:

Q: What kind of mammal **is a** platypus?

A:

Then maybe platypus should inherit from the monotrene class? This still might not always be the case...

Let's consider a "classic" example (Classic because it's about 20 years old.) Recall our Square class.



Let's add a Rectangle class now!

Q: We've been tasked to fix some code for Rectangles and Squares. what do we know about all Squares?

A:

Q: Let's inherit! What will our picture look like?

A:

Q: What fields should our Rectangle have? What public methods will it have?

A:

Ehh, screw all of that. I just used public values instead:


```
public class Rectangle {
    public double width;
    public double height;

    public Rectangle(double width, double height)
{
        if (width > 0 && height > 0) {
            this.width = width;
            this.height = height;
        } else {
            //exception!
        }
    }
}
```

Q: What's the first thing we need to do to rewrite the `Square` class to subclass `Rectangle`?

A:

Q: Is there any data duplication?

A:

Q: How should we fix this?

A:

Q: Do we need to fix `Square.setSideLength`?

A: Yes!

Q: Fix it!

A:

Q: Do we need to fix `Square`'s constructor?

A: Nope!

Q: Can the user create illegal Squares?

A:

Q: How?

A:

Q: How can we fix this?

A:

Q: What is the difference?

A:

Let's try private first and see if we can keep it that way! (This is the best strategy.)

Q: Do we still have a problem between Square and Rectangle?

A:

Q: Is this code duplication?

A:

Q: What can we do about it?

A:

Q: What are we going to want in `Rectangle` now?

A:

Q: What should the setters look like?

A:

Q: Should we change the `Rectangle` constructor?

A:

This last step is called *Refactoring*, which we will do a lot this semester.

Q: Can I make the instance variables private? (If they're not already.)

A:

Q: Okay, so now how do we implement `Square.setSideLength`?

A:

Q: Can we create illegal Squares?

A:

Q: How can we fix this?

A:

Q: What will the new `Square.setWidth` be?

A:

Q: Does that work?

A:

Q: How can we fix this?

A:

Q: Does this code work?

A:

Q: Can I create illegal Squares?

A:

Q: Did we have to override any methods in `Square`?

A:

Q: Can unexpected things happen?

A:

```
List<Rectangle> boxes = monkey.getBoxes();
//set all the boxes to have width 5.0; don't
change height
for (Rectangle box :  boxes) {
    box.setWidth(5.0);
}
```

Q: What could go wrong here?

A:

This is very dangerous!

Guideline: *Principle of Least Surprise* - The user should not be surprised when an object of type A has unknown subtype B.

There are even more guidelines to help dissuade subclassing based solely on the *Is-A* perspective:

Q: Sounds like we shouldn't often have B subclass A... What if both A and B are immutable?

A:

Guideline: Class B, identical to A but with extra restrictions

on its state, should not be a subclass of A unless they are both immutable.

Notice that this fits the Math model. Everything in Math is immutable, thus every Square is also a Rectangle without problem! (I don't think Euclid was considering modifier methods when writing Elements.)

It can be very easy to add lots of classes, but sometimes they aren't totally appropriate.

Q: What benefit does having a Square class have?

A:

Guideline: Remove classes with little or no unique behavior.

3.2 Similar Public Interfaces

Q: What was the justification we gave for inheritance based on matching public interfaces?

A:

Q: Are there any reasons this might lead us astray?

A:

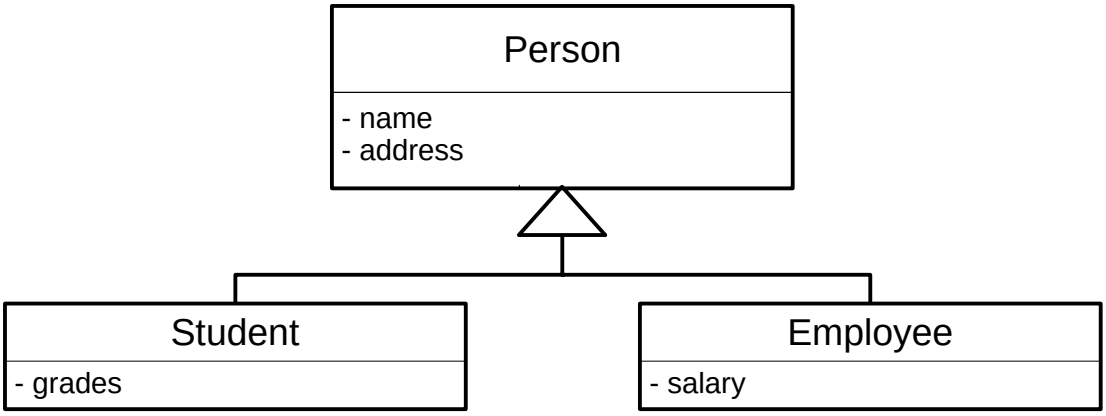
Guideline: *Liskov Substitution Principle (LSP)* - It is acceptable for `B` to subclass `A` only if for each method in both `A` and `B`'s interfaces, `B`'s method:

- Accepts as input all the values that `A`'s method accepts (and possibly more), and
- Does everything with those values that `A`'s method does (and possibly more).

Q: Why is this?

A:

⟨ Draw this figure:



⟩

Q: Plymouth State is creating a new system to keep track of people on campus. Is there any problem with this design?

A:

Q: What happens if we have a student who is also an employee?

A:

Q: What else could go wrong here?

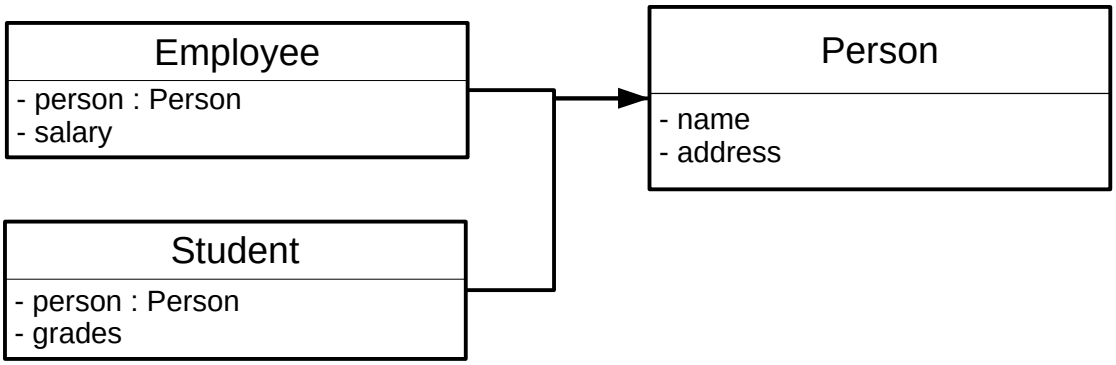
A:

Guideline: These are roles, and temporary roles should generally not be subclasses.

Q: What should we do instead of inheritance?

A: Composition. Have `Student` and `Employee` have `Person` fields.

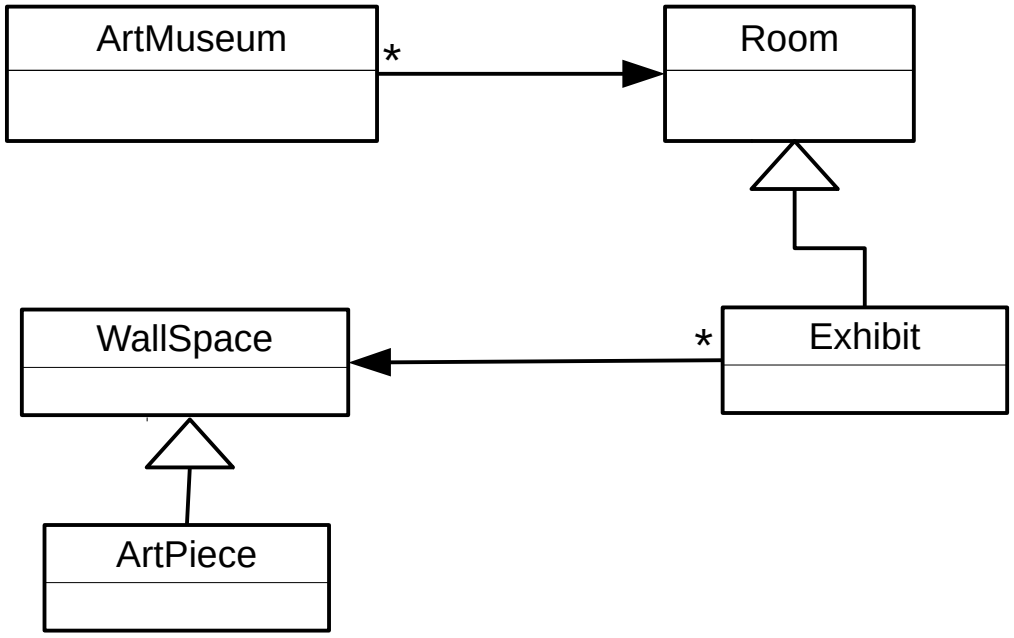
⟨ Draw this figure:



⟩

Let's pretend the following is the design that has been created for an Art Museum.

⟨ Draw this figure:



⟩

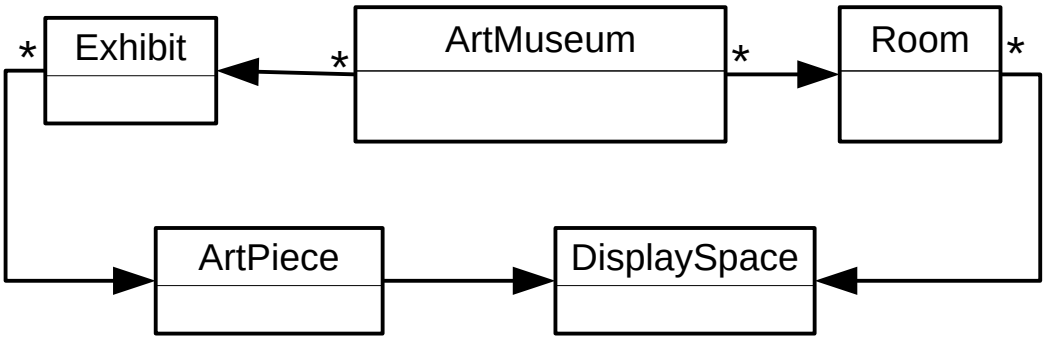
Q: What do you see that violates some of our guidelines? Hints: (reveal these one at a time)

- Are there any temporary roles here?
- Can an exhibit span multiple rooms?
- Can an art piece be in both the East Asian exhibit and the Turbulent Water exhibit?
- Does all art exist on walls?
- Should an art piece know about its location or the other way around?

A:

Q: Draw a new, more appropriate diagram.

Here's a version we often come up with:



3.3 Polymorphism

Q: Do we need inheritance to use polymorphism?

A:

3.4 Summary

Let's summarize what we've learned in this chapter. Which of the following are good reasons to use inheritance vs referencing?

Q:

- **Code Reuse**
- **"Is-A"**
- **Matching Public Interfaces**
- **Polymorphism**

A:

Q: Bonus point: Which has better efficiency: Inheritance or Referencing? (Don't ever make a programming decision based on this!)

A:

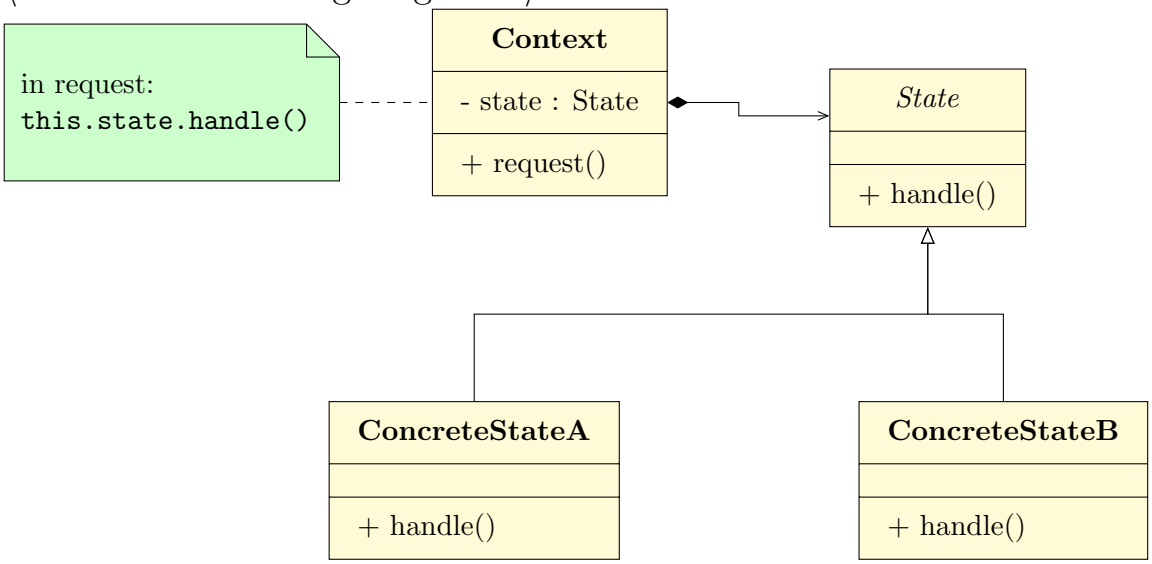
4 State Pattern

In our crusade to remove ugly conditionals, we enlist the aid of the State Pattern, a more general version of the Null Object Pattern.

4.0 State Pattern Basics

The State Pattern is the simple paragon of polymorphism.

⟨ Draw the following diagram: ⟩



Q: What are the Participants?

A:

Q: What are the roles?

A:

- Context: Object that has a changeable state.
- State: Abstract class for the different states.
- Each ConcreteStateX: One of the many states to implement the `handle` method.

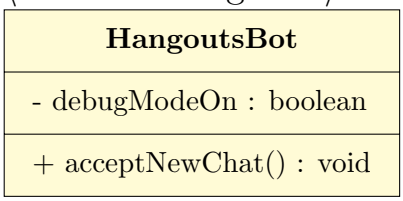
Q: Can we extend this pattern by adding another method to each State class?

A:

4.1 Example: Debugging Print Modes

When coding my Google Hangouts bot, I spend a lot of time adding and removing debugging print statements. In the `acceptNewChat` method in my HangoutsBot class, I decide to add a boolean flag field that tells me whether I want the debugging statements.

⟨ Draw the figure. ⟩




```
if (this.debugModeOn) {
    System.out.println("x=" + x);
}
```

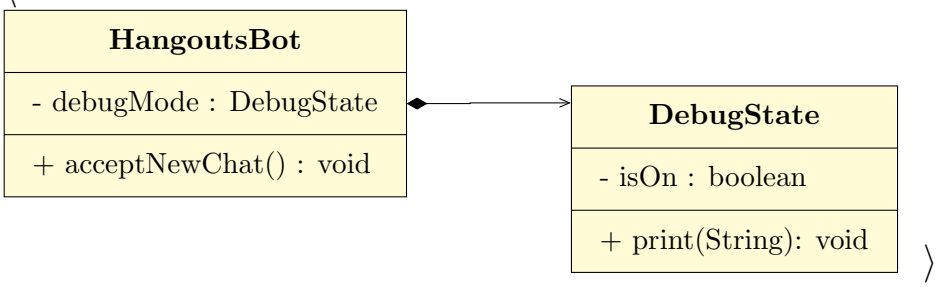

Q: Problem here?

A:

Q: How might be make this more elegant?

A:

⟨ Draw this:



⟩

Q: How would I implement `print`?

A:

Q: How would I replace the code in the `acceptNewChat` method?

A:

Q: Have we fixed most of the problem?

A:

Q: Have we used the State Pattern?

A:

Q: Our current state actually has two separate states. What are they?

A:

Q: So what should my states be?

A:

Q: What will that diagram look like?

A:

Q: Which classes here fill in for each of the State Pattern participants?

A:

Q: Now how do we implement `print` in the two classes?

A:

Q: Doesn't this violate something? We're nullifying a method! Which principle does this violate?

A:

Q: Could the situation get more complicated? If so, how?

A:

Q: How do we handle this?

A:

Q: What will *that* diagram look like?

A:

Q: What are the benefits of using the State Pattern?

A:

Q: What's so cool about being able to extend by adding new State classes?

A: Can add new states even if you don't have access to the source code for the Context class. Awesome!

4.2 Example: Pokédex - Caught and Unseen Pokémon

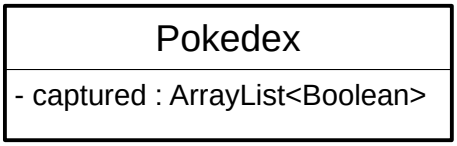
Q: I'm out walking in the woods, hunting for Pokémon, when I get a message from my younger sibling: "Hey! Have you seen a Bulbasaur yet? It's Pokémon number 1!" I pull out my Pokédex to see what it has to say about this species. If I've never seen one, what does it tell me?

A:

Q: I'm wondering how to explain to my younger sib that I'm a failure, but then I see a Bulbasaur! I throw a Pokéball at it and catch it! What does my Pokédex say now?

A:

Time to implement this in our `Pokedex` class:



This might be the code for the method that returns this view:

```
public JPanel viewPokemon(Pokemon pokemon) {
    int number = pokemon.getNumber();
    boolean caught = this.captured.get[number];
    String name;
    if (caught) {
        name = pokemon.getName();
    } else {
        name = "???";
    }
    Figure figure; //either a blank image or a spinnable
3D model.
    if (caught) {
        model = pokemon.getModel();
    } else {
        model = this.getBlankImage();
    }
    float height;
    ...
}
```

Q: What's wrong with this?

A:

Q: There's a simple fix to just have one conditional. What's that?

A:

Q: Why is this not a perfect fix?

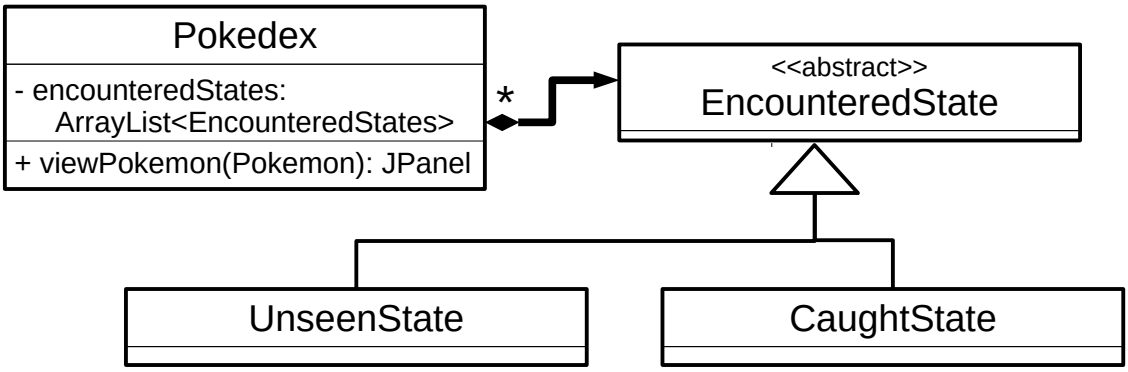
A:

Q: Let's employ the State Pattern. What are my participants going to be?

A:

Q: Okay, now draw the new Class Diagram.

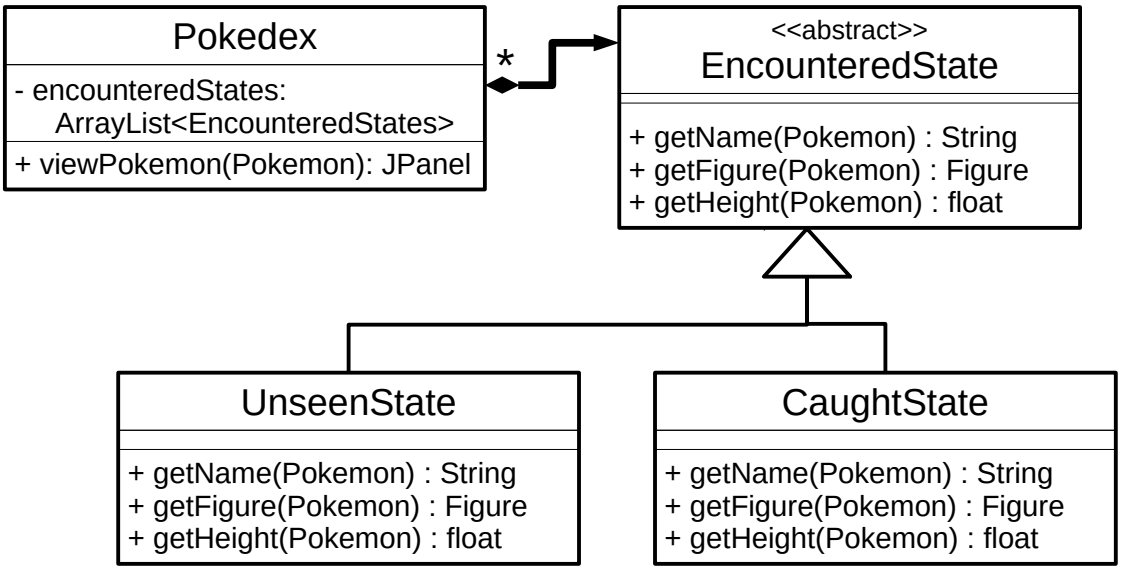
A:



Q: What are some of the methods we want in the `EncounteredState`s?

A:

Q: Okay, let's update the class diagram again!



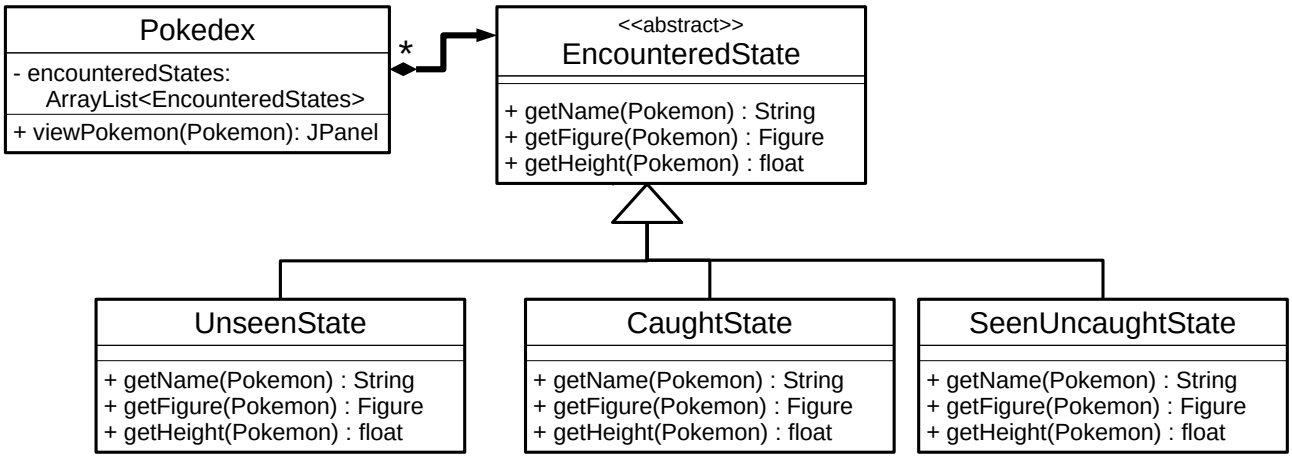
Q: How could we rewrite the `viewPokemon` method now?

A:

Q: Have we covered all the different states?

A:

Let's add a new state: `SeenUncaughtState`
⟨ Update the diagram!



⟩

Q: How much code do we have to write in the `Pokedex` class to implement this change?

A:

Q: Why is this so awesome?

A:

4.3 Example: Monopoly

Q: When rolling dice in Monopoly, what does getting "doubles" allow you to do?

A:

Consider the case where there isn't a maximum number of turns you can stay in jail. (Normally, you must be released after three turns.)

⟨ Draw Class diagram for `Monopoly Player`. Attributes:

- `- currentLocation: MonopolySpace`
- `- inJail: boolean`
- `- doublesSoFar: int`

And methods:

- `+ moveDistance(numberOfSpaces: int): void`
- `+ moveToSpace(space: MonopolySpace): void`
- `+ isInJail(): boolean`

⟩

Q: What can we make more elegant using the State Pattern?

A:

Q: Which State Pattern participant role will the MonopolyPlayer take?

A:

Q: Which fields should we be able to replace?

A:

Q: What shall we call this new state?

A:

⟨ Draw DoublesAndJailState:

- abstract!
- methods:
  - `+ processDiceRoll(die0:int, die1:int, player:MonopolyPlayer): void`
  - `+ isInJail(): boolean`

⟩

Q: In Groups: come up with a Class Diagram of your solution. Hint: my solution has no fields.

A:

Q: What does the `isInJail` method do in each?

A:

Q: What would the `processDiceRoll` method look like in `InJail`? Hint: I added a `setDoublesAndJailState` to the player class.

A:

Q: Write `processDiceRoll` for the other three!

A:

TODO: add more here. Find my notes; we did lots!

5 Singleton Pattern

TODO: add a class diagrams to the end of this section!

Sometimes you only want to have a single instance of an object. The *Singleton Pattern* is a means to enforce this.

Q: Consider a room-scheduling object, `RoomScheduler`, which keeps track of when rooms on campus are scheduled. It has methods such as:

- `isFree(Room, Date start, Date end)`
- `scheduleRoom(Room, Date start, Date end)`

What is the problem with having multiple instances of RoomScheduler?

A:

5.0 Use Reluctantly!

Q: In order to implement the Singleton pattern, we need a single "point of entry" to prevent people from creating new instances of our object. What's already a huge red flag about this?

A:

Often, inexperienced programmers will think it's okay to introduce a Singleton because they need to access some variable from anywhere. *Don't do this!* Using an OO Design Pattern does not automatically mean you're using good design!

5.1 The Singleton Participant

Q: Goal: have a class where the constructor can't be invoked by outside code. What's wrong with just removing the constructor?

A:

Q: What can we do to make the constructor inaccessible by outside classes?

A:

Common for other code to access the instance by doing something like the following:

```
RoomScheduler scheduler = RoomScheduler.instance;
scheduler.isFree(...);
```

Q: What does that mean must be in our code?

A:

Q: Where then are we going to call the constructor?

A:

Q: Shivers of Joy or Shudders of Revulsion?

A:

Example: Here's the code so far for a general Singleton:

```
public class Singleton {
    public static Singleton instance = new
Singleton();

    private Singleton() {
        //constructor code goes here
    }
}
```

Q: When can the constructor be executed?

A:

Q: Does that solve the problem this pattern sets out to solve?

A:

Q: Great! Is it well-designed?

A:

Q: Reasons for shudders?

A:

5.2 Removing the Public "Constant"

Q: Let's try to solve these design issues! How can we make the field private?

A:

In the `RoomScheduler` case, here's how we would get and use our singleton.

```
RoomScheduler scheduler = RoomScheduler.getInstance();
scheduler.isFree(...);
```


Q: How can we implement the `getInstance` method? (In general or specifically for `RoomScheduler`.)

A:

There is a bit of an efficiency problem here. Sometimes a Singleton can be a very heavy class, because it's doing lots of concurrency-management stuff. The constructor may take a lot of time to execute. We may not want the Singleton constructor to execute immediately upon execution of the code. This is especially a problem if we're testing the code; each time we test we have to first wait for the constructor to execute!

5.3 Delaying the Construction

Q: How can we use a Just-in-Time approach and not execute the constructor until we first access the instance?

A:

Q: Does this solve the problem?

A:

Q: Does it solve it well?

A:

Q: Why not? What design "red flags" do we have?

A:

Q: Are there any practical problems remaining?

A:

Q: How could we have multiple threads doing lots of unnecessary work? (And possibly cause big problems.)

A:

5.4 Solving the Race Condition

Q: We want Java to enforce that only one thread can execute the first branch of `getInstance` at a time. Can we do this?

A:

Q: We can enforce that only one thread can invoke `getInstance` at a time. How does that work? Hint: it's a Java keyword.

A:

Q: Sweet! No problems, right? Hint: think parallel efficiency.

A:

5.5 Java's Solution to the Bottleneck

Q: Java has a special way to solve this problem: Lazy Class Loading. What does that mean?

A:

Q: Does this work for inner classes?

A:

Q: How can we use it to solve our problem?

A:

```
public class Singleton {
    private static class SingletonHolder {
        public static Singleton instance = new
Singleton();
    }
    private Singleton() {
        //constructor code goes here
    }
    public static getInstance() {
        return SingletonHolder.instance;
    }
}
```

This solution lazy loading of inner classes was proposed by Bill Pugh[11] as a new implementation of Java. See `http://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples#bill-pugh-singleton` for more information.

5.6 Example: `LinkedIntList`

TODO: Tree or Linked List? I should probably do a LinkedList here and have the Tree as an exercise.

[11] `http://www.cs.umd.edu/~pugh/`

5.7 When to use the Singleton Pattern

5.8 When to Use?

Q: Which of the following are good reasons to use the Singleton Pattern? Which are terrible reasons?

1. Want to be able to access fields/methods from anywhere.
2. Want to enforce there is only one copy of an object.
3. Want a class that doesn't require any state (no fields).
4. Want to force client code to be unable to call constructors.

A:

Some slam dunks with Singleton:

- Something that can only have one instance.
- States from State Pattern with no fields. (You can save on memory by only having one instance of them.)

If you are ever in doubt, you should probably not be using the Singleton Pattern.

6 Elegant Methods

Q: Recall the *Query-Command Separation Principle*. What does that say?

6.0 Public vs. Private Methods

An *invariant* is a statement about the state of an object that must be true. Examples:

- The size of a stack is equal to the number of stack frames.
- A balanced tree should have all leaves with depth equal to the height of the tree or one less.
- A max heap should have the greatest element at the root.

Q: If we had a `Rodent` class, what could an invariant of an object in that class?

A:

One good way to debug is to write private methods that check the invariants. Then write a `satisfiesInvariants` method to check all of them. The following example could be from a sorted Tree Set class:

```
private boolean satisfiesInvariants() {
    return (this.isBalanced() &&
this.satisfiesOrder() && this.hasNoDuplicates());
}
```


Q: What is the difference between public and private methods?

Q: When *should* you make a method private vs. public?

Consider the following methods inside the `Rodent` class:

- Public: `mouse()`
- Private:`devilMouse()` and `angelMouse()`

⟨ Write the code for `mouse` on the board:

```
public void mouse() {
    (Leave space here)
    this.devilMouse();
    (Leave space here)
    this.angelMouse();
    (leave space here)
} ⟩
```

⟨ Let's add some calls to satisfiesInvariants in here:

```
public void mouse() {
    System.out.println(this.satisfiesInvariants());
    this.devilMouse();
    System.out.println(this.satisfiesInvariants());
    this.angelMouse();
    System.out.println(this.satisfiesInvariants());
} ⟩
```

Q: For which of those statements is it okay to print `false`?

A:

Q: Why?

A:

Let's restate that as a guideline:

Guideline: Invariants need to be satisfied between any public method calls.

TODO: more here? Check with written notes.

6.1 Method Documentation

Q: What is the difference between *internal documentation* and *external documentation*?

A:

Q: How is this specifically relevant to compiled languages?

A:

Q: What's the problem? Hint: similar to code and data duplication...

A:

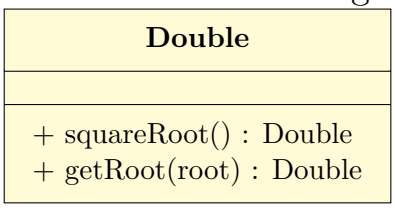
TODO: add more in here from the written notes
(And in the Javadoc stuff that starts on page 16.)

6.2 Pre/Post Conditions

Q: Documentation should always state the pre- and post-conditions for a method. What are pre-conditions?

Q: What are post-conditions?

Consider the following implementation of Double:



Q: What are the pre- and post-conditions of `squareRoot`?

A:

Q: What are the pre- and post-conditions of `getRoot`?

A:

Q: Recall the Liskov Substitution Principle (Section 3.2): "It is okay for B to subclass A only if for each method in both A and B: B's method takes all the inputs that A's takes (and possibly more) and does everything that A's does (and possibly more)."
Does it seem like we can rephrase this using pre- and post-conditions?

A:

Q: Let's use $<$ to mean "less strict than". Fill in the boxes using either $\leq$ or $\geq$:
It is okay for B to subclass A if, for all methods M in both A and B (M_A and M_B):

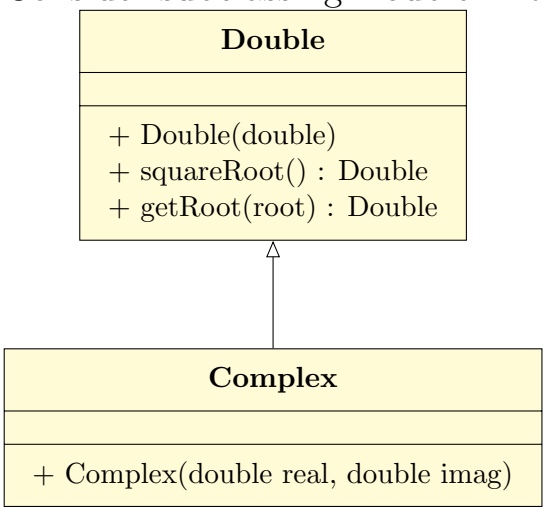
- preconditions(M_A) $\square$ preconditions(M_B), and
- postconditions(M_A) $\square$ postconditions(M_B)

A:

Let's do an example! Go back to our square root example. Let's assume there's a `Double.getRoot(int degree)` method. This could be used in the following way:

```
Double double = new Double(125.0);
double cubedRoot = double.getRoot(3);
System.out.println(cubedRoot); //should be 5.0
```

Consider subclassing Double in the following way:



Q: Is there any benefit to this?

A: Yes, because we can now return complex numbers if we take the square root of an imaginary number. Example:

```
Complex c = new Complex(-1, 0);
```

Q: What will the pre and post conditions be?

A:

6.3 Overriding Clone

Q: Let's do an example. What's the `Cloneable` interface? (Which method does it enforce?)

A:

Q: In intro programming, we learned about two types of clones. What were they?

A:

Q: Which type of clone is this (in the Triangle class, where each field is a Point object.)? (Note: you should always start with a call to the super class's clone method, as we will see.)

```
public Object clone() {
    Triangle copy = (Triangle)
this.super.clone();
    copy.p0 = this.p0;
    copy.p1 = this.p1;
    copy.p2 = this.p2;
}
```

A:

⟨ Draw this code, then the resulting object diagram:

```
Point origin = new Point(0,0);
Point right = new Point(3,0);
Point up = new Point(0, 4);
Triangle rightTriangle = new Triangle(origin, right, up);
Triangle rightClone = (Triangle) rightTriangle.clone(); ⟩
```

Q: Why is the shallow clone dangerous?

A:

Q: There's another problem here. What if we clone in the following code:

```
Collection<Triangle> triangles =
landscape.getTriangulation();
Collection<Triangle> copies = new
ArrayList<Triangle>();
for (Triangle triangle :  triangles) {
    Triangle copy = (Triangle) triangle.clone();
    copies.add(copy);
}
```

What could be a problem here? Hint: what's the type of triangle?

A:

Q: What do we need to do to get `clone` to return a `ColorTriangle`?

A:

Q: Okay, let's fix the shallow problem first. How can we rewrite the `clone` method to deepen itd?

A:

7 Elegant Classes

Yet another big section of information I could give you before you start work on even your first project...

Q: When you are first given a new Project, should you start off by implementing the project?

A:

7.0 Pre-Implementation Steps

7.0.0 Read the Project Spec(ification).

7.0.1 Ask Questions!

Usually the spec will not be fully clear. Now is a great time to clarify anything you're not certain about. (In true Agile Development, this should be easy because you have access to the Customer Representative[12] at all time. Questions can be asked at any point in the process!)

[12]See Section 12.2 for more information about the role of the Customer Representative

7.0.2 Come up with Use Cases.

Q: What is a *use case*?

A:

For example, when I took this course in 2002, we had to create a program for playing sounds. Here's an example use case for an early version of the program:

- User starts running the program. A window appears with a large empty box. Next to the empty box are two columns of buttons with one simple image on each. One has a cat, another a car. The button with a dog is selected already. Below the empty box is a button with the Play symbol on it.
- The user clicks in the empty box. A rectangle with the dog image appears at the mouse click. (We'll refer to this box as the Sound Panel from here on.)
- The user clicks the play button and a red line appears at the left-hand side of the Sound Panel, and moves from left-to-right. When it reaches the Dog-rectangle, it continues to move while a barking noise is heard. The sound ends exactly as the red line reached the right-hand side of the dog-box. The red line continues to the right until it reaches the end of the Sound Panel and disappears. The play button remains disabled while the red line is moving.
- The user clicks on the car button on the button panel. It becomes selected and the dog button becomes unselected.
- The user clicks on the sound panel again and a rectangle of different width than the dog rectangle with the cat image appears where the mouse was clicked.
- The user clicks again and another cat rectangle appears.

- The user clicks the play button and the red line moves across the sound panel again. It plays the sound of each rectangle while it's touching that box. The height of the box in the sound panel does not change the sound produced by each box.

7.1 Extract Nouns and Verbs

7.2 CRC (Class-Responsibilities-Collaborators) Cards

TODO: way more to put here!
2016: skipped to Cohesion

7.3 Cohesion

Guideline: Cohesion: Every class should be responsible for doing one thing and doing it well.

Q: Does this mean that every class should have one method?

A:

Q: What are some the benefits of following this guideline?

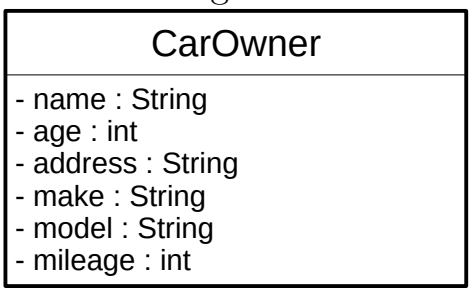
A:

Q: What are some good examples of classes?

A:

Often times, you'll see a name that looks like this: `BigFiletOMacFish`. Red flag!

Q: What's wrong with this class?



A:

Q: What's a better solution?

A:

Q: How can we measure cohesion?

A:

7.4 Responsibilities

Let's do more guidelines:
Guideline: Different *kinds* of responsibilities should be delegated to different classes.

7.5 Immutable Classes

Skipped in 2017!

7.6 Coding to Interfaces

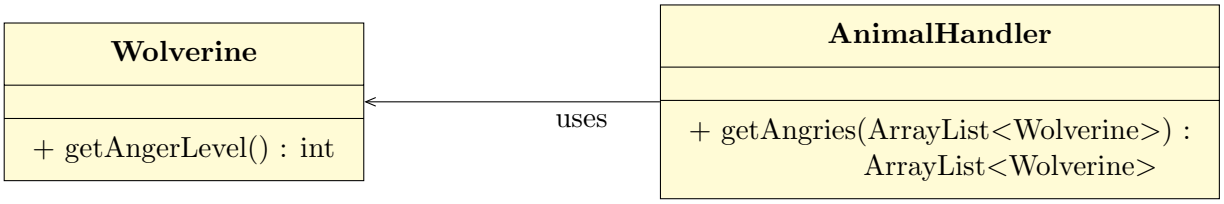
Guideline: Code to interfaces, not classes.

Q: What does this mean?

A: Whenever possible, write code that references other types as abstractly as possible. This means two things:

- Use Interfaces instead of classes.
- If you can't use an interface, go as high up the directory tree as possible.

Consider the following classes:



... and this implementation of `getAngryOnes`:

```
public ArrayList<Wolverine> getAngries(ArrayList<Wolverine>
wolverines) {
    ArrayList<Wolverine> angries = new ArrayList<Wolverine>();
    for (Wolverine :  wolverines) {
        if (wolverine.getAngerLevel() > 10) {
            angries.add(wolverine);
        }
    }
    return angries;
}
```

Q: Let's think about changing things to follow the spirit of "Coding to Interfaces". Could we generalize the parameter?

A:

Q: Great! How general could we make it?

A: Iterable<Wolverine>

Q: Why can we do that?

A:

Q: How can I change the return type?

A:

Q: When should you do that?

A:

Q: How does that help?

A:

Q: When should we leave the return type as an ArrayList?

A: If there's something specific about an ArrayList that is needed by the invoking code, then we should leave it alone.

Q: When would be the time to go "halfway"? E.g. to List?

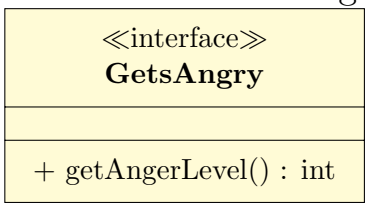
A: Maybe there's something important about having the return type be a list. Maybe it's ordered or needs to be indexable.

Q: Okay, one more generalization. Many members of the weasel family are known for being ornery (e.g. wolverine, badger, and honey badger). What if all subtypes of Weasel have the `getAngerLevel` method? How can we generalize further?

A:

Q: What would be even better?

A: If we have a GetsAngry interface:

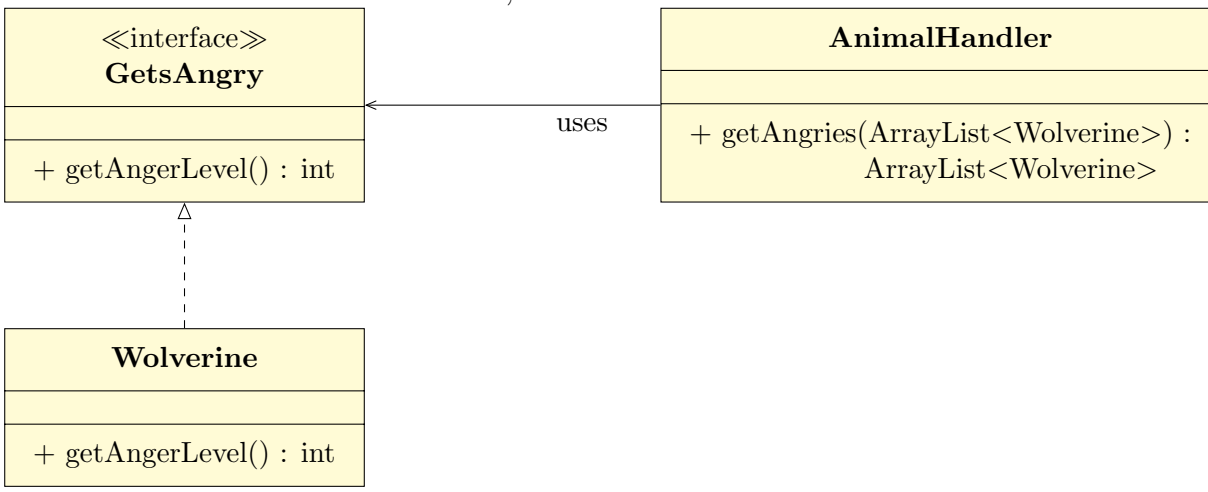


Q: What's the benefit of coding to interfaces?

A: You gain lots and lots of Freedom![13]

7.7 Coupling

If we are able to make the third change to our Wolverine and AnimalHandler classes from the last section, then we have a new class diagram:



Q: What does this new diagram highlight?

A: It's clear just how independent AnimalHandler is from Wolverine.

Q: Sometimes you will find classes that are very dependent on one another. Or you may find two different parts of a class diagram that have lots of interconnections between each other. Is this good or bad?

A: Bad! You want to minimize the interconnections between different parts of the program, either single classes or groups of classes.

Q: What are some other terms for this separating?

A:

- Minimizing *coupling*
- Keeping classes orthogonal (might have to explain this term)
- Minimizing interdependence between classes.

Q: How can I rephrase this if I think of a class diagram as a graph?

A:

Q: What can become nearly impossible if you don't do this?

A:

TODO: more to add here...
when you talk about tightly coupled, here's a funny picture:

[14]

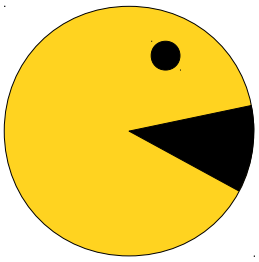
8 OODP: Composite Pattern

Sometimes we want to treat collections and individual objects the same way.

8.0 Pac Man

Consider the following: I'm making a poster about the history of video games. I'm making the poster in LibreOffice[15] and I start by making a few objects. I make the following three single objects:

⟨ Draw this:



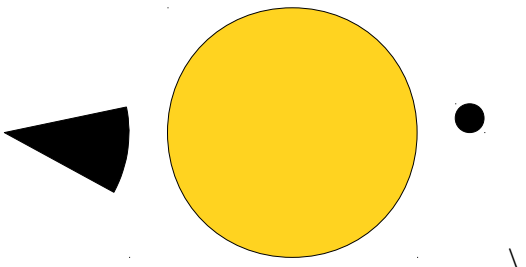
[16] ⟩

[14] Source: `https://twitter.com/jezenthomas/status/576376992167276544`
[15] `http://www.libreoffice.org/`
[16] Yes, I did make this in Libre Office.

Q: Which classic character can I make from these?

A:

⟨ Draw the combination:



⟩

Q: I'm probably going to be using this combination a bunch. Copy-pasting, resizing, etc. What should I do with both of these? (Hint: it's a LibreOffice thing and probably also something you can do in other graphics programs.)

A:

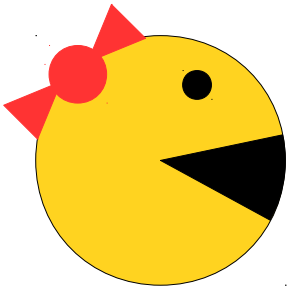
Q: Sweet! Who else might be an important character to include on my poster? Hint: similar in drawing to Pac-Man.

A:

Q: How do I "draw" Ms. Pac-Man?

A:

⟨ Draw Ms. Pac-Man:



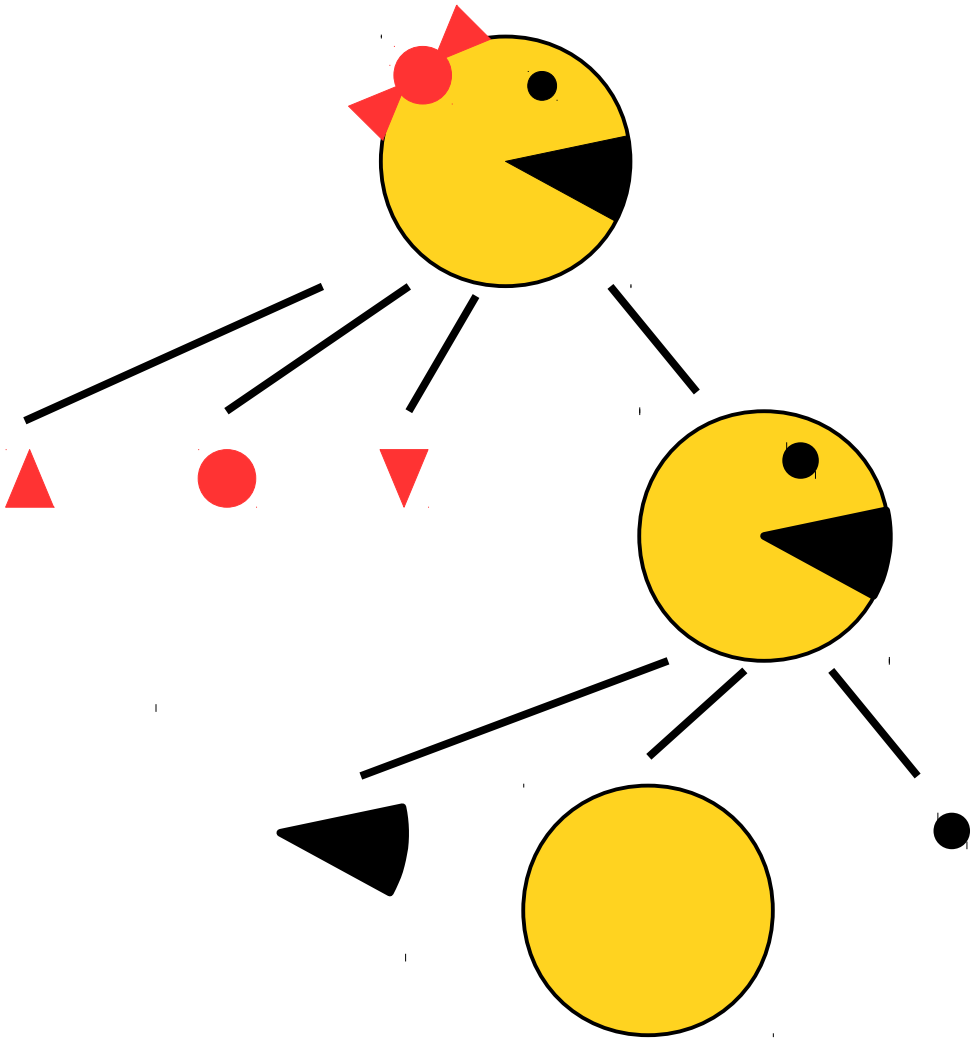
⟩

Q:	How do I draw a hairbow?
A:	

Q:	Now what do we do?
A:	

Q:	Which kind of data structure is the grouping like?
A:	

⟨ Draw the tree:



⟩

Q:	Which figures are the leaves?
A:	

Q:	Which are the internal nodes?
A:	

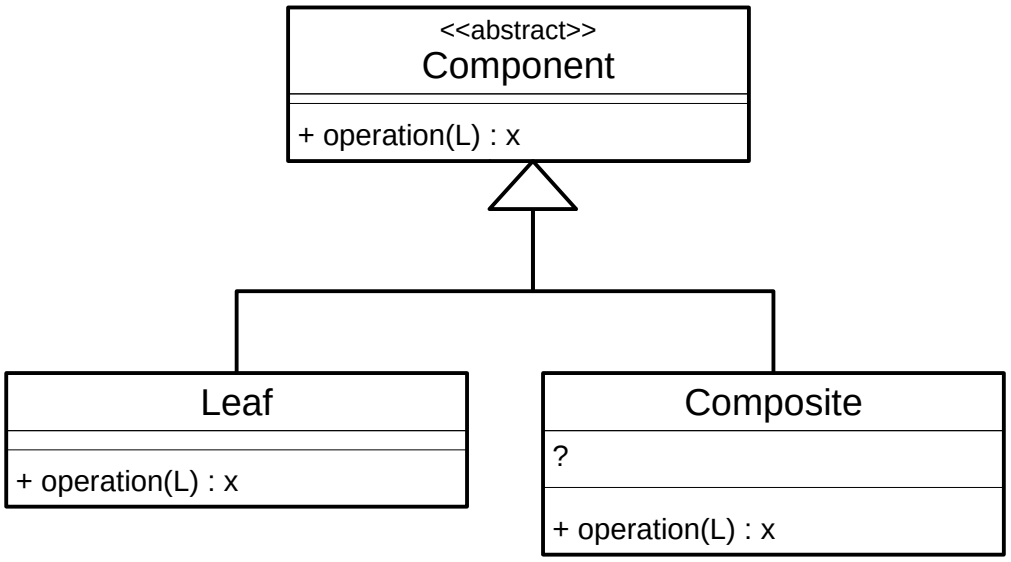
Q: Often we have a collection of objects that we want to act as a single object. How can we do this?

A:

8.1 Composite Pattern Specifics

The idea here is that we want our object to be either a leaf or a composite object.

⟨ Draw the initial figure: (note the space left blank for Composite fields!)



⟩

Q: Then I could give the `Composite` class a `Collection<Leaf>` objects. Does that solve the problem?

A:

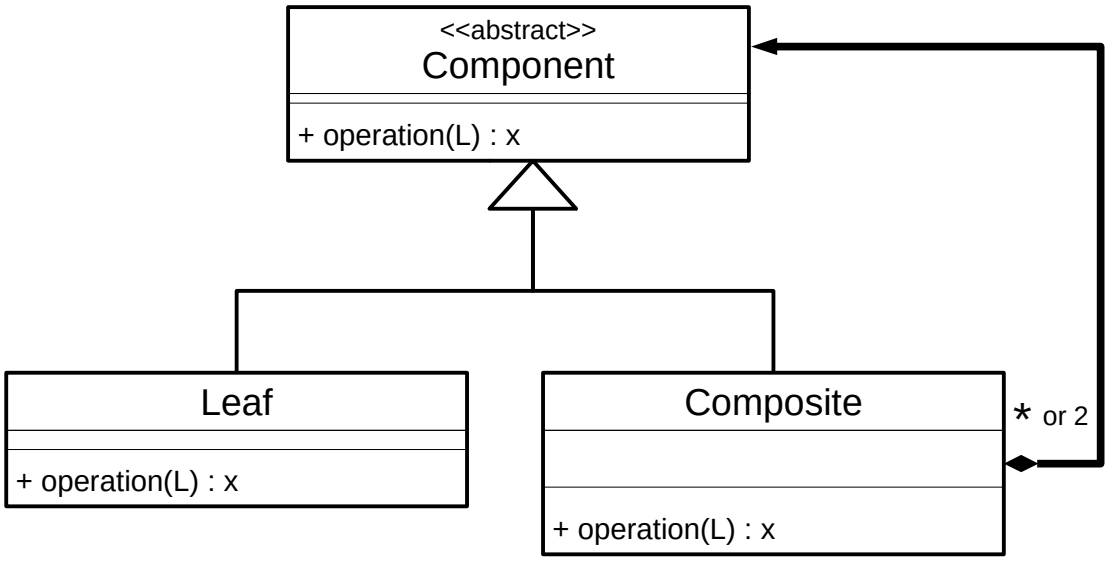
Q: What do we need to fix this?

A:

Q: What should the `Composite` have instead? Hint: to get trees of any level?

A:

⟨ Draw the figure:



⟩

Multiple ways to implement `Composite`:

- Constant number of "children" (usually 2) E.g. `left : Component`, `right : Component`.
- Variable number of children: `children : Collection<Component>`

Q: When user code interacts with classes in the Composite Pattern, which classes does it think it's using?

A:

⟨ Draw `Client`, which *uses* the `Component` class. TODO: make a figure for the whole thing. ⟩

Q: Why is this good?

A:

Q: How do we add/remove from a `Composite`?

A:

Q: What's the problem with adding methods to the `Composite` class?

A:

Q: How could we get those methods in the `Component` interface?

A:

Q: What's the problem here?

A:

Q: What is a problem with letting the Client use the `Composite` constructor?

A:

Q: What about this separate object idea?

A:

8.2 Composite Nim

⟨ Describe the game of Nim. ⟩

Q: A Nim position is really just a combination of single rows. How could we use the Composite Pattern here? (What will the three main participants be?)

A:

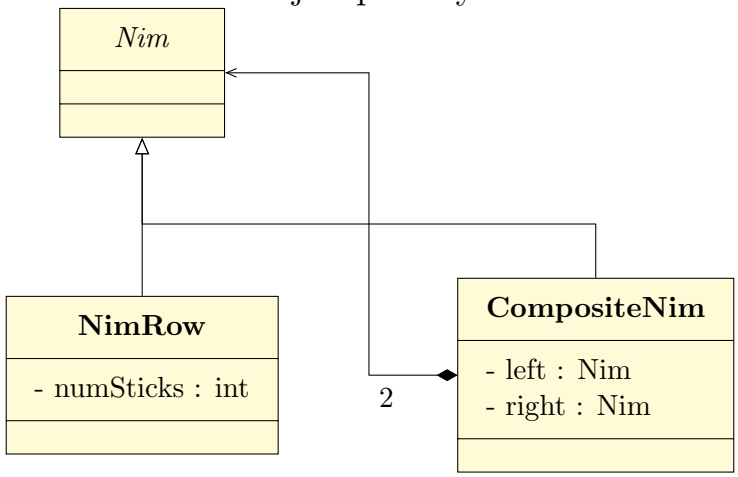
Q: What fields will CompositeNim have?

A:

Q: What about the fields for NimRow?

A:

Here's the class diagram for what we've got so far. Note: can't get the attribute to jump out yet.



Q: Let's add a constructor for the NimRow class.

A:

Q: Now add a constructor for CompositeNim

A:

Q: Let's add `toString() : String` to the `Nim` interface. Where am I going to add code?

A: NimRow and Composite Nim?

Q: Implement `toString` in NimRow.

A:

Q: Now implement `toString` in CompositeNim.

A:

Q: Let's add `hasMoves() : boolean` to the `Nim` interface. Where do we need to add code?

A:

- abstract stub in Nim:
  `public abstract boolean hasMoves();`
- implementation in NimRow and CompositeNim

Q: Where will the JavaDoc go?

A:

Q: What are we adding to NimRow?

A:

Q: What's the code for CompositeNim?

A:

Q: Next let's write `getMoves() : Collection<Nim>`. Where are we going to put code?

A:

Q: Implement `getMoves` in NimRow.

A:

Q: Okay, now implement `getMoves` in CompositeNim. (You may have to show an example of all the moves from, say, two piles: 3, 4.)

A:

Q: Can we now change `hasMoves` to reduce the number of implemented methods?

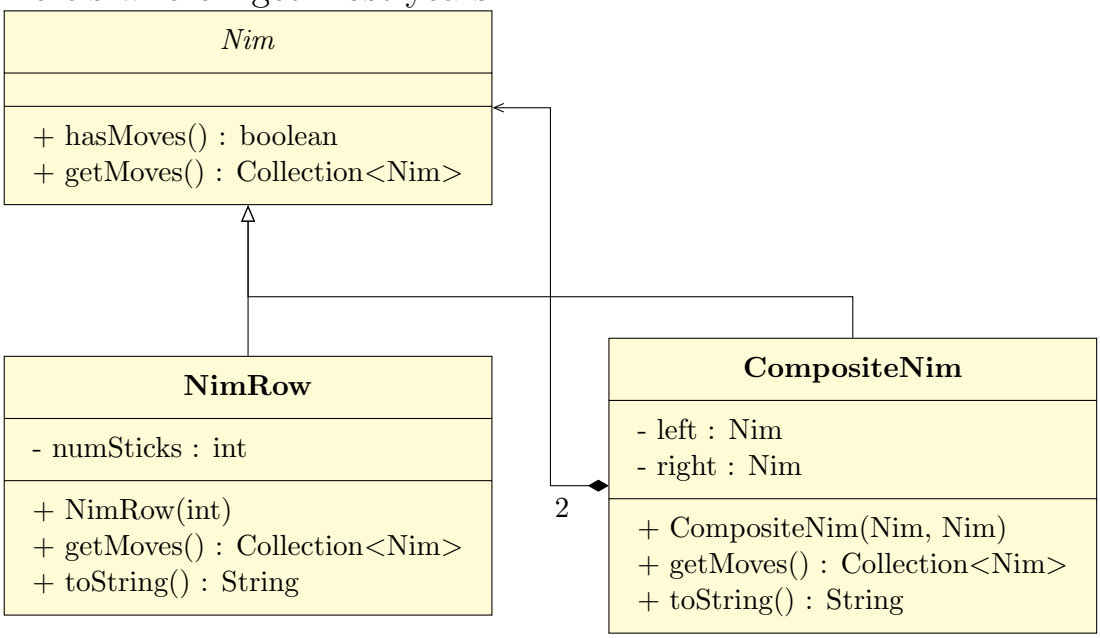
A: Yes! Move `hasMoves()` up to Nim!

Q: What will the implemented version of that look like in Nim?

A:

Note: This is as far as I got in 2017. We talked about equals, but didn't get there.

Here's where I get most years.



Q: We'd probably like a `hasMove(Nim) : boolean` method, but it will be much easier if we do which other method first?

A:

Q: Let's see if we can agree on when two Nim games should be equal. Are 3, 5, 7 and 7, 3, 5 equivalent?

A: Well, yes, but it's a bit odd. A player might be surprised if they moved from 3, 6, 7 to 7, 3, 5. It really depends on what we're doing here. For our purposes, let's say they're not equivalent. (i.e. order matters)

Q: How many different ways can we represent the game 3, 5, 7 with our code?

A:

Q: Let's add `equals(Object) : boolean` to the `Nim` interface. Plan: abstract `equals(Nim)` in `Nim`, implemented `equals(Object) : boolean` in `Nim`, then implement `equals(Nim)` in subclasses, which comes next...

A:

Q: Let's add `equals(Nim) : boolean` to the `NimRow` class. Hint: also include a `equals(NimRow): boolean` method here.

Q: Now let's add `equals` to the `CompositeNim` class. Does the order of the rows in a Nim board really change the game at all?

A:

Oh yikes! That's a problem. To handle this, let's write a `protected toMap()` method that returns a `Map<Integer, Integer>`. Then we can just test the equality of two maps.

⟨ Describe Maps if someone is not familiar with them. ⟩

Q: Okay, let's write `toMap`. First the `NimRow` version!

A:

Q: Okay, now the `CompositeNim` version of `toMap`.

A:

Q: Okay, now let's write `equals` for the `CompositeNim` class.

A:

Q: Does anyone see something we can do to clean things up?

A:

Q: Now let's add `hasMove(Nim) : boolean`. Can we implement this in the abstract class?

A:

Q: What does the code for that look like? Hint: Can you write it in one line?

A:

⟨ Describe algorithm for determining whether a Nim position has a winning move. ⟩

Q: Let's write a `hasWinningMove()` method. Is there a method we might want to write first?

A:

Q: Okay, let's write the `getNimSum`. First in the `NimRow` class.

A:

Q: Okay, now for the `CompositeNim` class.

A:

Q: Okay, now where should we implement `hasWinningMove()`? What's the code for that?

A:

9 OODP: Factory Method Pattern

Sometimes we either can't or don't want to have the user call a constructor to create an object. In that case, it might be time to use the *Factory Method Pattern*.

9.0 Example: Nim

Continuing from the Nim example introduced in Section 8.2 we have overlooked one vital piece: we can't create Nim objects! Issues:

- Can't initialize `Nim` objects: it's abstract.
- User shouldn't interact with the `Leaf` or `Composite` classes in the Composite Pattern, so can't call the constructor there.

Q: Why shouldn't the user interact with those classes?

A:

Q: How do we get around not using the constructor for the subclasses directly?

A:

Q: Which class should `createNim` be implemented in?

A:

Q: To start off, let's have this method always return the game: 3, 5, 7. How are we going to use it? (Write a line of code for a Client class.)

A:

Q: So is our method going to be static or not?

A:

Q: Implement the method!

A:

Q: Remember how we learned about coding to interfaces. Let's practice that now! How can we change the declared types of the variables in that method?

A:

Q: It's *slightly* inelegant to use the `CompositeNim` and `NimRow` constructors. Why is that?

A:

Q: How could we fix this problem?

A:

Q: Okay, what does our class diagram look like now?

A:

Q: It looks like NimFactory is very tightly coupled with the three Nim game classes. How do we fix that?

A:

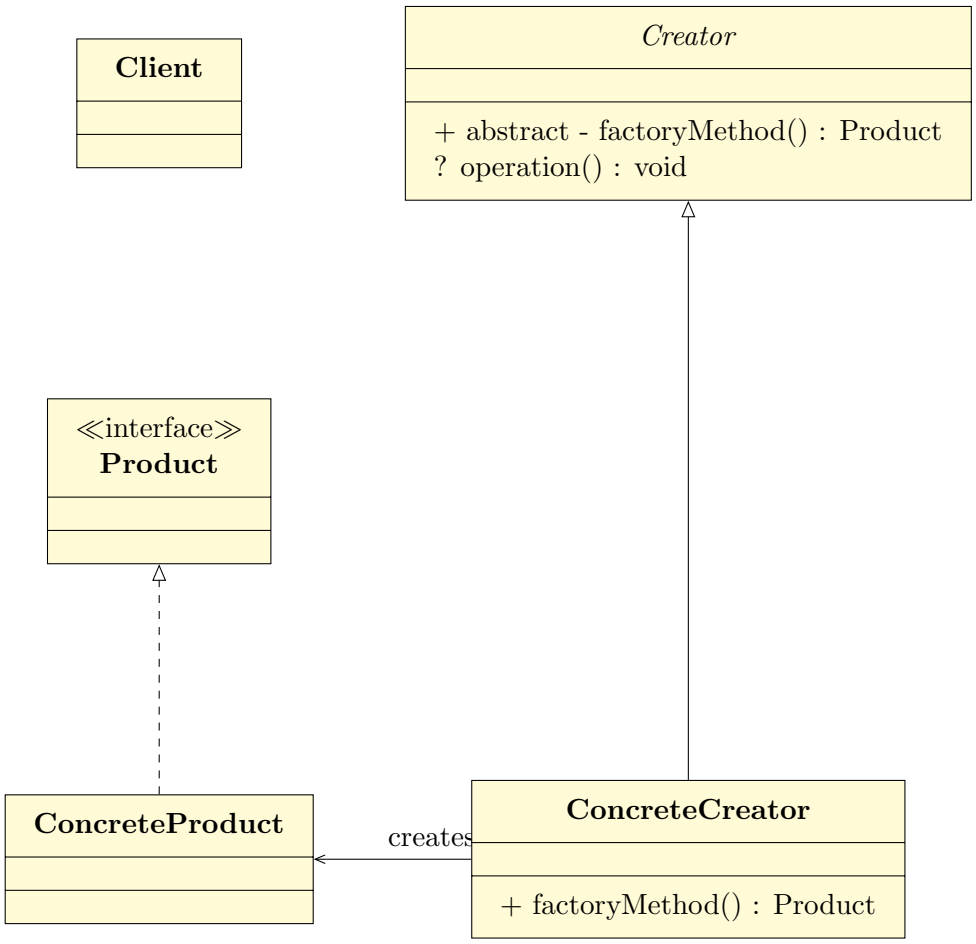
Q: What do I do if I want to create something other than the 3-5-7 board? Let's create random boards!

A:

9.1 Factory Method Pattern Basics

Q: What does the Factory Method class diagram look like?

A:



9.2 Factory Method with State Pattern

TODO: drawing of RandomNimFactory subclassing NimFactory.

Q: Why is this better?

A:

TODO: more here! Three factories: `StandardNimFactory`, `SpecifiedNimFactory` and `RandomNimFactory`.

10 OODP: Command Pattern

Often we want to treat executable code as a referenceable/passable/storeable value. Example: OS schedules jobs. Solution: keep a priority queue of jobs to execute, possible if each job is storeable.

Q: In some programming languages, you can do this with functions. Called: "first-order functions". What are some languages you know that act like this?

A:

Luckily a solution exists even for languages like Java without first-order functions: the *Command Pattern*.

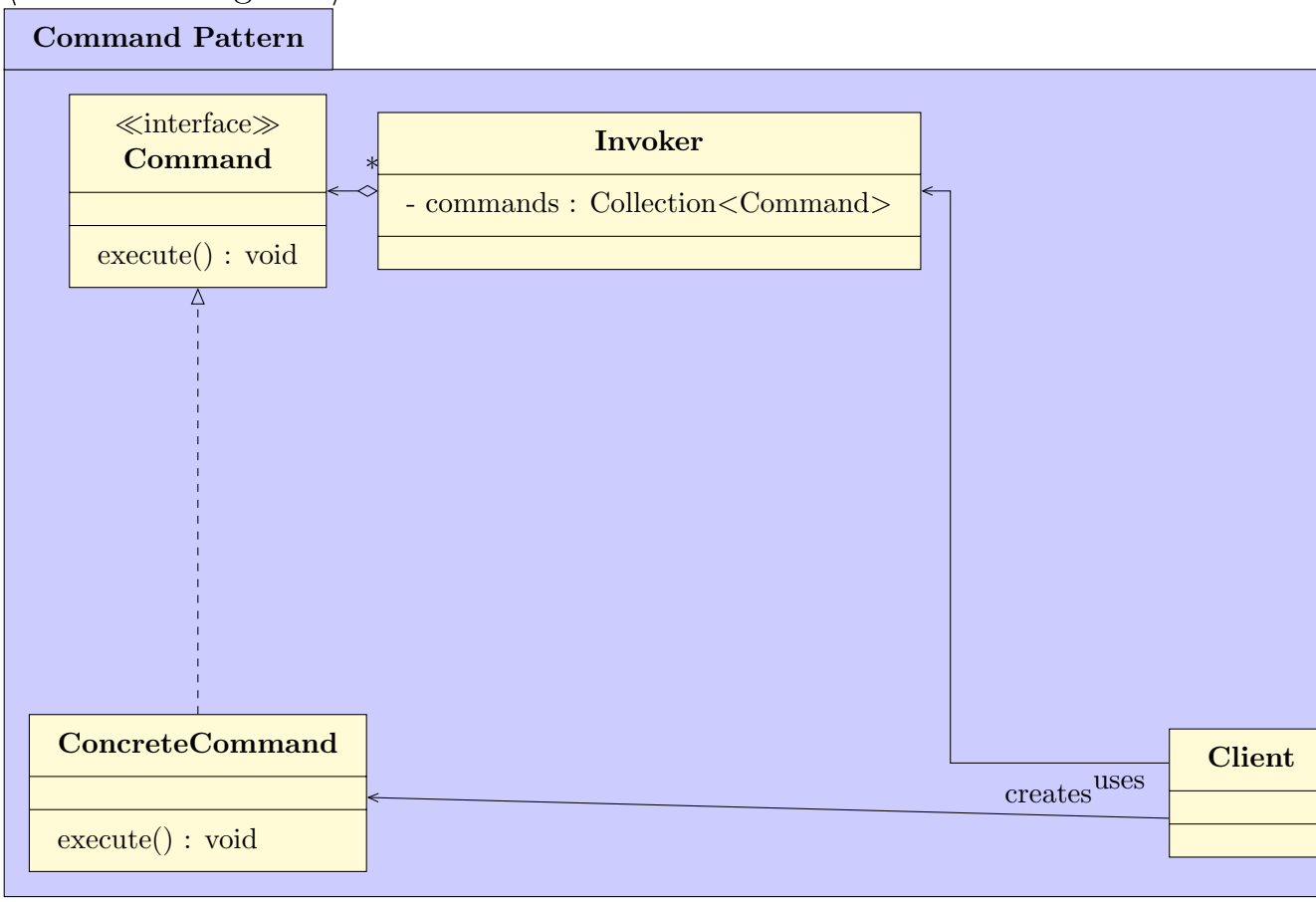
10.0 Command Pattern Specifics

Four main participants:

- `Command`: interface with an `execute() : void` method.
- `ConcreteCommand`: implementes the Command interface, encapsulating the action.

- `Invoker`: The class that invokes the `execute()` method of the `Command` objects.
- `Client`: The class that creates the `ConcreteCommand` objects and sends them to the `Invoker`.

⟨ Draw this figure: ⟩



Q: What is the advantage of separating the `Invoker` and `Client`?

A:

Q: Can the Invoker store commands?

A:

Q: What are some other benefits?

A:

Q: Does Java have a similar class to `Command`?

A:

Q: What is especially awesome about the `Runnable` class.

A:

10.1 Web Browsing

I would like a `WebBrowser` class with:

- `+goBack():void`
- `+goForward():void`
- `+goToPage(String url):void` (Called when a new URL is typed into the address bar or a link is clicked.)

We'll assume there is another method that does the actual loading of a page: `loadPage(String url):void` Assume this one is already implemented.

Let's use the Command Pattern to implement the other three methods. Plan:

- use a Command for moving to a new page. (`WebBrowseCommand`)
- store all the page movements in back and forward stacks of commands.

Q: Which role will the `WebBrowser` class fulfill?

A:

Q: Who will store the stacks of `WebBrowseCommand`s?

A:

I called my Invoker: `UrlManager` and gave it similar methods to my Client:

- `+goBack():void`
- `+goForward():void`
- `+goToNewPage(WebBrowseCommand):void`

Q: Which role remains to be filled?

A:

I called mine `GoToPageCommand`. It has two fields:

- `url:String`
- `browser:WebBrowser`

Q: What does the `GoToPageCommand` constructor look like? Hint: it's boring.

A:

Q: What about the `execute()` method?

A:

Q: What will the fields of the `WebBrowser` look like?

A:

Q: Let's try to implement the `UrlManager`. What fields do we need here? Hint: we need three:

- Stack of commands
- Another stack of commands
- A single command.

What are their roles?

A:

Q: Let's implement the `goToNewPage(WebBrowseCommand)` method!

A:

Q: Let's implement the `goBack` method next!

A:

Q: Let's implement `goForward`.

A:

Q: What about the methods in `WebBrowser`? What should the `goBack` and `goForward` methods do?

A:

Q: What should that `goToPage` method do?

A:

Q: Implement it!

A:

Q: We would like other code to not call the `WebBrowser.loadPage()` method. Can we make it private?

A:

Q: Isn't there another solution?

A:

An inner (non-static) class solves a lot of problems:

- Code in the inner class has access to all members of the outer class.
- Then `loadPage` can be private!
- Increases encapsulation of the `WebBrowser` and it's components.
- If the inner class is private, then we increase information hiding!

11 Waterfall Software Development

Waterfall development is a linear model for creating software. Each step is succeeded by the next, so that each step should be fully completed before moving on to the next.

11.0 Steps

1. Requirements: A complete product requirements document should be produced that details the product expectations.
2. Design: The structure of the code should be designed before any implementation is done.
3. Implementation: The software is actually coded.
4. Verification: The correctness of the software is measured so that it's certain that all the requirements have been met.
5. Maintenance: The software is changed to remove bugs and to implement new features, etc.

I'm not going to go into detail about these steps, mostly because I think you all have learned a bit about this in a previous class.

11.1 Benefits

Design is important! Design before you code! Cowboy coding doesn't really have a place here.

11.2 Limitations

In practice, these steps don't actually happen one after another.

TODO: add more to this chapter

11.3 Other Philosophies

There are many other software philosophies.[17] We're going to talk next about the philosophy of Agile Software development.

[17] There's a long list at `https://en.wikipedia.org/wiki/List_of_software_development_philosophies`.

12 Agile Software Development

Agile Software Development consists of a bunch of software development methods where change is expected. Change happens, so the project must be able to evolve without waiting for permission.

12.0 The Agile Manifesto

Agile Manifesto:

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

- **Individuals and interactions** over Processes and tools
- **Working software** over Comprehensive documentation
- **Customer collaboration** over Contract negotiation
- **Responding to change** over Following a plan

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

Q: What do you think each of the four points means?

A:

12 Major Principles:

- Customer Satisfaction by rapid delivery of useful software.
- Welcome changing requirements, even late in development.
- Working software delivered frequently (weeks rather than months)
- Close, daily cooperation between business people and devs
- Projects built around motivated individuals, who should be trusted.
- Co-location allows best communication.
- Working software is measurement of success.
- Sustainable Development to maintain a constant pace.
- Continuous attention to technical excellence and good design.

- Simplicity—the art of maximizing the amount of work not done—is essential
- Self-organizing teams
- Regular adaptation to changing circumstances

Q: Where is Agile Development appropriate?

A:

Q: In Agile Dev, is it better to predict problems or adapt to them?

A:

An adaptive team may have trouble reporting what they're going to do next week, but will be able to report about which features they are planning for next month.

Q: How much should an agile team document their code?

A:

Q: What's wrong with too much documentation?

A:

Q: What's wrong with too little documentation?

A:

12.1 Agile Iterations

Q: How long should an individual task be?

A:

Q: Each group of tasks organized into an *iteration*. What happens in each iteration?

A:

Q: What does this sequence remind you of?

A:

Each iteration is kind of like a mini-waterfall. Kind of.

Q: How long should each iteration last?

A:

Q: What happens at the end of the iteration?

A:

Q: Why bring the clients in between each pair of iterations?

A:

Consider the functions of the team discussed earlier: planning, requirements analysis, design, coding, unit testing, and acceptance testing.

Q: Which of the functions can only happen once?

A:

Q: When are planning and requirements analysis repeated?

A:

Q: When does unit testing happen?

A:

Q: Why is automated testing vital for an Agile team?

A:

Q: How often should an agile team meet to discuss progress?

A:

Q: What does each team member (or subgroup) report at the meeting?

A:

Q: What happens if you (a team member) think you can solve another team member's roadblock?

A:

12.2 Customer Representative

Q: Big idea: who is the unusual member of the agile team?

A:

Q: What are their responsibilities?

A:

Q: What happens if the customer representative is filled by a member of the software development team (instead of the stakeholders)?

A:

Q: What should be done if there cannot be a customer representative on the team?

A:

12.3 Information Radiator

An *information radiator* is a large physical display set up to display the current status of the project.

Q: Who should be able to see the information radiator?

A:

Q: Why passerby?

A:

Q: What media should the information radiator use?

A:

Q: What is a build light indicator?

A:

Often uses three lights:

- Pass
- Fail
- Being re-tested

Q: Who uses the BLI?

A:

12.4 Code Quality in an Agile Team

Q: We spent lots of time covering good code design. How important is that for an agile team?

A:

Q: How is good design often recognized?

A:

Q: What else helps drive quality code?

A:

Q: What happens when code quality is not maintained?

A:

12.5 Team Experience

Q: A customer requests a project be completed with Agile engineering with 10 software developers. You have 20 people to choose from: 10 senior devs and 10 junior devs. Who do you put on the project?

A:

Q: Why?

A:

Q: Why don't new people learn as they go?

A:

Q: What does this tell you about getting into Agile development?

A:

Q: What does this tell you about Agile development from a business perspective?

A:

Q: I've seen college described the following way: Enough Sleep, Good Grades, Social Life; Choose two. How could we use the "choose two" mentality in the Iron Triangle?

A:

Q: If you are choosing to use Agile Development, which two are you choosing?

A:

Q: In what ways is it expensive?

A:

13 OODP: Visitor Pattern

13.1 Motivation: Collectibles from the 1980's

Let's say we have a binary tree describing a My Little Pony toy collection.[19] The `MyLittlePony` class has a `toString` method so the client's tree can nicely print the collection.

Q: Old My Little Pony toys can be sold for money, and their recent trade values are updated daily on a fan website. How might our clients want us to improve the `MyLittlePony` class?

A:

One solution to this problem is to include a `toStringWithPrice()` method in the `MyLittlePony` class.

Q: Why should we add this new method instead of change the `toString` method to print out the price?

A:

Let's complicate the matter further:

- Another client has a collection of G.I. Joe toys, which we have recorded in an `ArrayList<GIJoe>`.
- A client also has a collection of Cabbage Patch Kids, itemized in a `Graph<CabbagePatchKid>`.
- Online, updating-value databases exist for both!

[19] Note: I came up with this example before the Friendship is Magic TV series was created. It predates any bronyhood I may or may not have adopted.

Q: What do these two clients probably want?

A:

Q: What is probably true of the `CabbagePatchKid`, `MyLittlePony`, and `GIJoe` classes?

A:

Q: Since we're such good developers, we also find a way to determine the eBay trade value and volume for each of the three 80's toys. What will our clients want once we tell them?

A:

Q: How many specialized `toString`-ish methods do we have?

A:

Q: In general, how many will we have if we have n toy types and m different methods required?

A:

Q: Where do we probably have some repeated code?

A:

Q: What should we do to fix this?

A:

This is what code that used that might look like:

```
MyLittlePony posey = new MyLittlePony(...);
EbayVolumePrinter printer = new
EbayVolumePrinter();
printer.print(posey);
```

Q: What's a problem with this?

A:

Q: Wait... how is that? What does the `EbayVolumePrinter.print(EightiesToy toy)` method have to do?

A:

```
public String print(EightiesToy toy) {
    if (toy instanceOf MyLittlePony) {
        MyLittlePony pony = (MyLittlePony) toy;
        return this.printPony(pony);
    }
    ...
}
```

Q: Also, now how many total methods do we have?

A:

Q: How many in general for n toys and m printers?

A:

Let's do something that looks a bit like a step backwards. Let's add a method to the `MyLittlePony` class:

```
public String toStringWithEbayVolume() {
    EbayVolumePrinter printer = new
EbayVolumePrinter();
    return printer.printMyLittlePony(this);
}
```

Q: Now how many total public methods do we have for these specialized `toString`-type methods? (For our specific 3 toy types.)

A:

Q: And in general? (n toys, m printers)

A:

Q: Is there some way we can condense all of the methods `toStringWithEbayVolume`, `toStringWithEbayPrice`, and `toStringWithPrice`?

A:

Q: Let's call that method `specialToString`. Is it going to take any parameters?

A:

Q: What is the parameter it's going to take?

A:


```
public String specialToString(Printer printer) {
    return printer.printMyLittlePony(this);
}
```

Q: What's the signature for the `printMyLittlePony` method?

A:

Q: Can I change this to `public String print(MyLittlePony pony)`?

Q: Wait... then is it different from `public String print(GIJoe joe)`?

A: Yup! Different Signatures!

Q: Wow! What's `specialToString` going to look like now?

A:

Q: Anything specific to Ponies in there?

A: Nope!

Q: Then where can we move it?

A: Nowhere.

Q: Why can't we move it up to `EightiesToy` class?

A: Because inside each of the subclasses, the type of `this` is specific. There isn't any `print(EightiesToy)` method in the `Printer` classes.

Q: How many total methods do we have now?

A:

Q: Can we do any better?

A: Not really

Q: What's the benefit of this reorganization, then?

A: The Printers are well-separated from the Toys

Q: Which can we add without having to modify anything else, Toys or Printers?

A:

13.2 Visitor Pattern Participants

We can generalize this plan for any objects by using the Visitor Pattern! Participants:

- `ConcreteElement`: Object to perform some operation on.
- `Element`: interface for different `ConcreteElement` classes.
- `ConcreteVisitor`: Object that will perform the action on the visited object.
- `Visitor`: Interface for the different visitors.
- `Client`: Uses the `ConcreteVisitors` to perform operations on elements.

Q: Which of the above roles do we want each of our example classes to fit in?

A:

The Printers are our Visitors, which do different things based on both:

- Visitor type

- Element type

Both dynamically! The Visitor Pattern lets us do this without sacrificing polymorphism, using a double-dispatch technique: we're going to use dynamic method invocation twice by making two method calls.

First invocation: choose the element type; the second one dynamically chooses the element.

Q: The pattern's two interfaces, `Visitor` and `Element`, are a bit different from what we've built so far as our abstract classes, `Printer` and `EightiesToys`. What can we do to make our visiting more flexible?

A:

Using the generic type: our `Visitor<T>` means that the `visit` methods have return type `T`.

Q: On the `Element<T>` side, we're going to need to *accept* the visitor, like the 80's toys accepted the printer. What is the signature of that `accept` method?

A:

Q: We can implement that accept method in the abstract superclass! What does that method look like?

A:

Q: What's the problem with implementing this in the abstract class?

A:

Q: What should we do instead?

A:

Q: What's super weird about this?

A:

Q: And that makes Kyle sad because...?

A:

Q: What's another benefit of this?

A:

Q: What is going to be in the abstract `Visitor`?

A:

Q: To add a new visitor, where do we have to add code?

A:

Q: Why is this helpful?

A:

⟨ Draw the class diagram! ⟩ TODO: add the diagram here

13.3 Implemented Example

Let's implement the Visitor pattern with our classes from before: `EightiesToy`, `Printer`, and their subclasses.

Q: What will the code look like for `EightiesToy`?

A:

Q: Okay, let's implement one of the subclasses: `MyLittlePony`. Ignore all other methods aside from the `accept` method.

A:

Q: Do the same for `GIJoe` and `CabbagePatchKid`.

Q: Let's implement the `EightiesToyVisitor` superclass. Do it!

A:

Q: Let's implement the `EbayVolumePrinter` class! (We don't need to include the bodies of any methods here...)

A:

13.4 Downsides

Q: What are some limitations of this pattern?

A:

14 Parallel OODP: Producer-Consumer

Q: Remember old newspaper printers with the big machines? If you're printing up today's newspapers, it can take a while for them all to finish. What should I do after the first bundles of papers come off the machines?

A: Start distributing them!

14.0 Motivation and the Big Problem

Q: In this metaphor, the machine is the Producer, creating the newspapers, and my army of young "Newsies"[20] (hooray no child labor laws) is my Consumer. What's the big idea here?

A:

Q: What are some examples of this (both real-world and computing-land)?

A:

Real World:

- Grocery Stores put milk into the fridge cabinets, customers buy them.
- I produce lecture notes as fast as I can and then teach them during my classes.
- Students are assigned homework in their different classes and turn it in when due.

Computing:

- Computer Jobs are created by applications and the user and are run on the CPU when the scheduler decides.
- Routers receive packets in one channel and send them out on another.
-

Q: There's an easy solution here: just put a queue as a "buffer" between the Producer and Consumer. Boom, code up a Queue using an ArrayList and we're done! Right? What's the problem here?

A:

Q: What do we want the Producer to do if they try to add something to a full buffer?

A:

Q: What do we want the Consumer to do if they try to remove something from an empty buffer?

A:

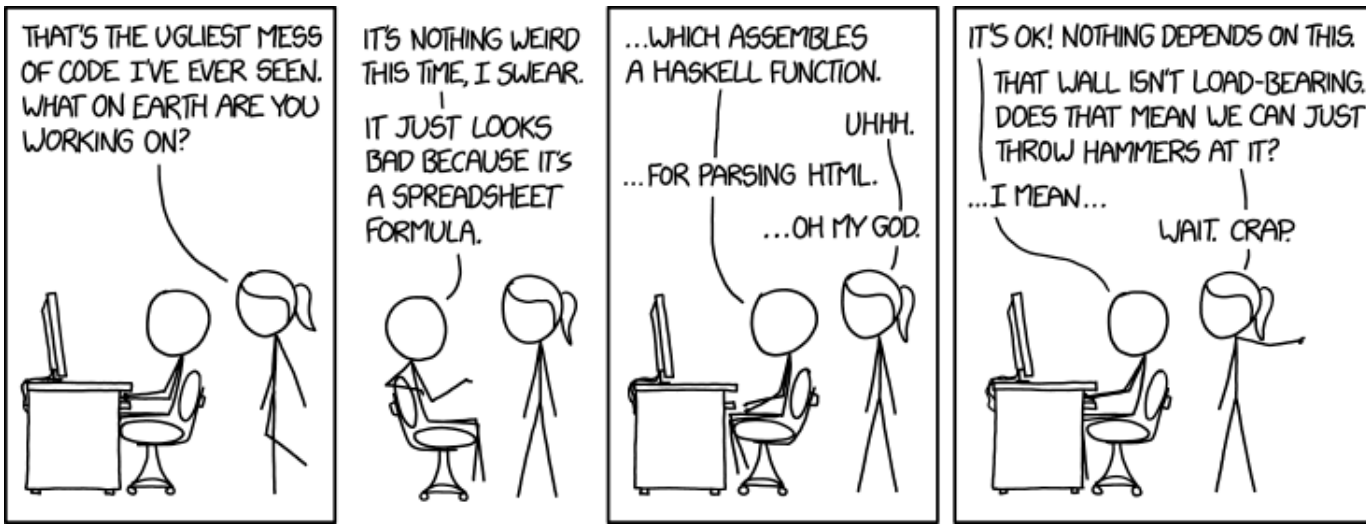
14.1 Common First Code

Here's what the (non-OO) code commonly looks like:

```
public void producer(Queue buffer) {
    while (true) {
        Object produced = this.generate();
        if (buffer.isFull()) {
            producerSleep(); //blocks 'til producerAwaken()
call
        }
        buffer.add(produced);
        if (buffer.size() == 1) {
```

```
                consumerAwaken();
            }
        }
    }

    public void consumer(Queue buffer) {
        while(true) {
            if (buffer.isEmpty()) {
                consumerSleep(); //blocks until consumerAwaken()
            }
            boolean wasFull = buffer.isFull();
            Object produced = buffer.remove();
            if (wasFull) {
                producerAwaken();
            }
            this.consume(produced);
        }
    }
```



[21] Comic source: https://xkcd.com/1926/

Q: What are some issues with this design? (There are lots.)

A:

- Not very OO. The producer and consumer seem like separate roles and should each have their own class. Then there's a lot to do there...
- Looks like generics could be useful in there since we are declaring things to be of type Object.
- These methods are not *thread safe*! That means that if you run them, a *race condition* can occur: unexpected results can happen due to the timing of instructions in separate threads. In this case, there is the potential for a deadlock!

14.2 Sleeping with Semaphores

Q: Let's first address an implementation issue: How do we implement the awaken and sleep methods?

A: Semaphores!

Q: What is the interface for a Semaphore?

A:

- Constructor: `Semaphore(numTokens : int)`
- `p()`: asks to "borrow" a token. If none is available it blocks until one becomes available.
- `v()`: returns a token. Sometimes it's okay for code that never borrowed a token to return one.

2223

Q: How many semaphores will we need?

A:

Q: What will we call them?

A: How about:

- `producerAwake` (blocks when the producer needs to sleep) and
- `consumerAwake` (block when the consumer needs to sleep)

[22] `p` is short for the Dutch *proberen* (to try out); `v` is short for *verhohen* (to increase)

[23] Java's Semaphore implementation (`https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html`) uses `acquire()` and `release()` as `p()` and `v()`, respectively.

Q: Can we just make the following replacements?

- `producerSleep()` → `this.producerAwake.p()`
- `producerAwaken()` → `this.producerAwake.v()`
- `consumerSleep()` → `this.consumerAwake.p()`
- `consumerAwaken()` → `this.consumerAwake.v()`

A: Not really. Remember that the problem happened because we skipped over some of the awaken calls.

Q: What can we do instead?

A: We'll set it up so that the calls are not inside conditionals. We'll just call them every single time through the loops!

Q:

There are two things to think about to understand how this works:

- How many tokens is each Semaphore going to start with? (Hint: not 1) and
- Where exactly are the calls to `p` and `v` going to go?

Let's tackle the second question first. Let's rewrite the `producer` loop.

A:

```
public void producer(Queue buffer) {
    while(true) {
        Object produced =
this.generate();
        //block until there's space in
the buffer
        this.producerAwake.acquire();
        buffer.add(produced);
        //indicate that there's another
element in the buffer
        this.consumerAwake.release();
    }
}
```


Q: How about the consumer loop?

A:

Q: Okay, how many tokens is the `producerAwake` Semaphore going to have initially?

A: The size of the buffer. So:

```
this.producerAwake = new
Semaphore(buffer.size());
```

Q: What about the `consumerAwake` Semaphore?

A:

14.3 Instilling OO principles

This fixes the race condition, but there are other problems related to the design that do make it easier to run this code. We will definitely be making other changes with the Semaphores also.

Q: What do I need to do to get the current code running? What would the constructor (let's just put everything in there for now) have to look like to get everything running?

A:

Q:

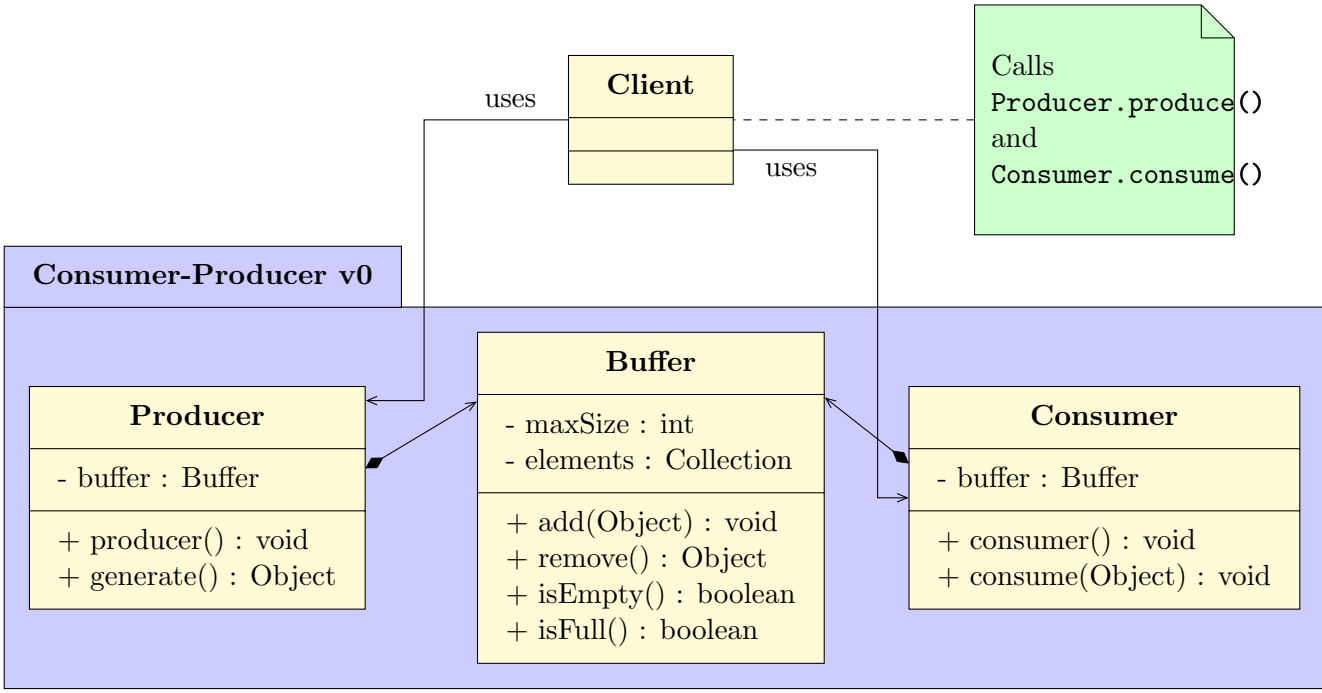
Q: Okay, let's think ahead a bunch and see if we can figure out what the four participants of our final design pattern are.

A:

- **Producer**: Class that creates objects that need to be consumed.
- **Consumer**: Class that consumes created objects.
- **Buffer**: Data Structure that holds produced objects that are waiting to be consumed.
- **Client**: Creates the Producer, Consumer, and Buffer and launches the Producer and Consumer.

Q: Which of the three main classes will have references to the others?

A: Producer and Consumer will have references to the Buffer.



14.4 Improvements

Q: What's the first change we notice? Hint: think about the data structures class.

A:

Q: What about the Producer class. What is going on in the `producer` method?

A:

Q: Should that be running in its own thread?

A:

Q: So what should we do with it?

A:

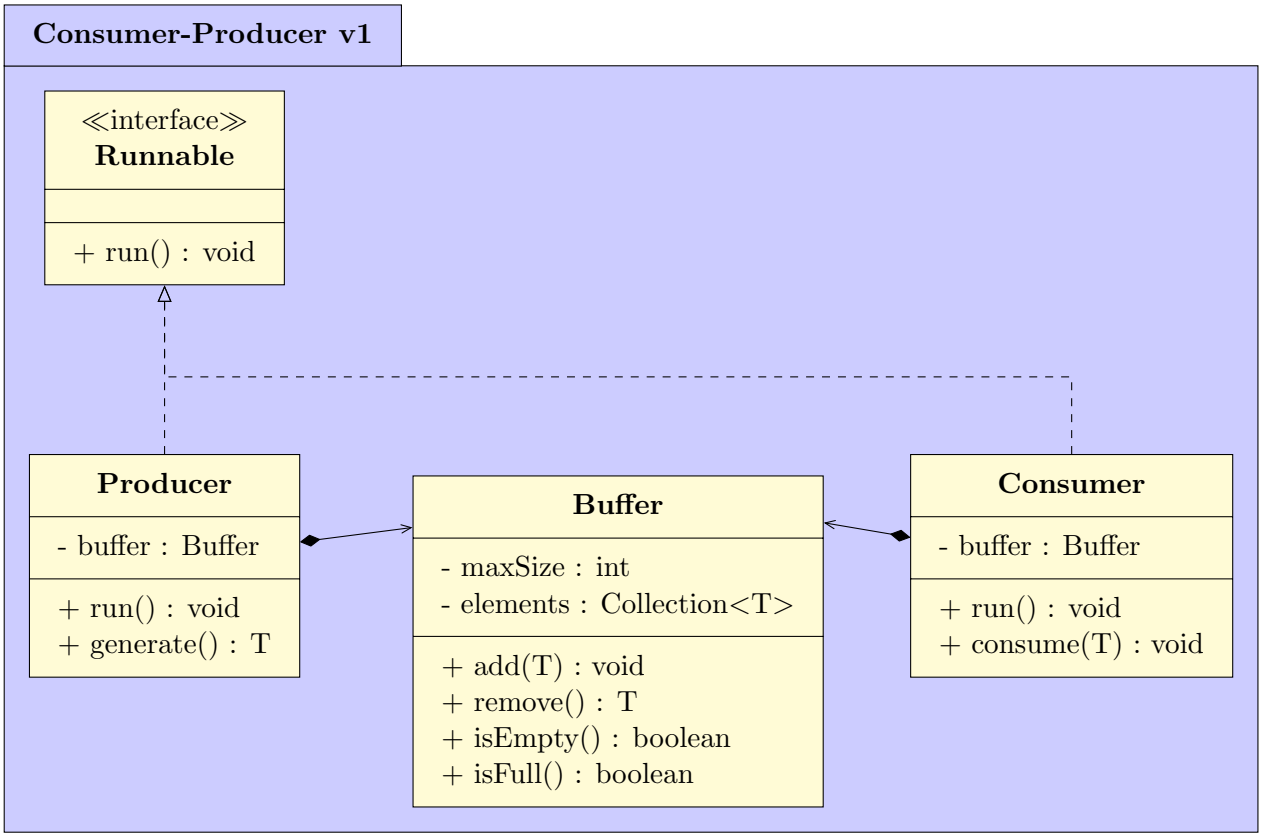
Q: What about Consumer's `consumer` method?

A:

Q: What might the code in the Client look like that launches everything? (Hint: I used Strings in my example.)

A:

New Class Diagram! (Note: I've removed the Client from here on out.)



5

14.5 Shifting Responsibilities

Q:	Okay, now where are we going to put the semaphores to keep everything thread safe? If we had to choose just one object to handle the concurrency, where does that responsibility fit most cleanly?
A:	Use the buffer!

Q: Let's give the buffer a semaphore field. How can we use this to solve our problem?

A:

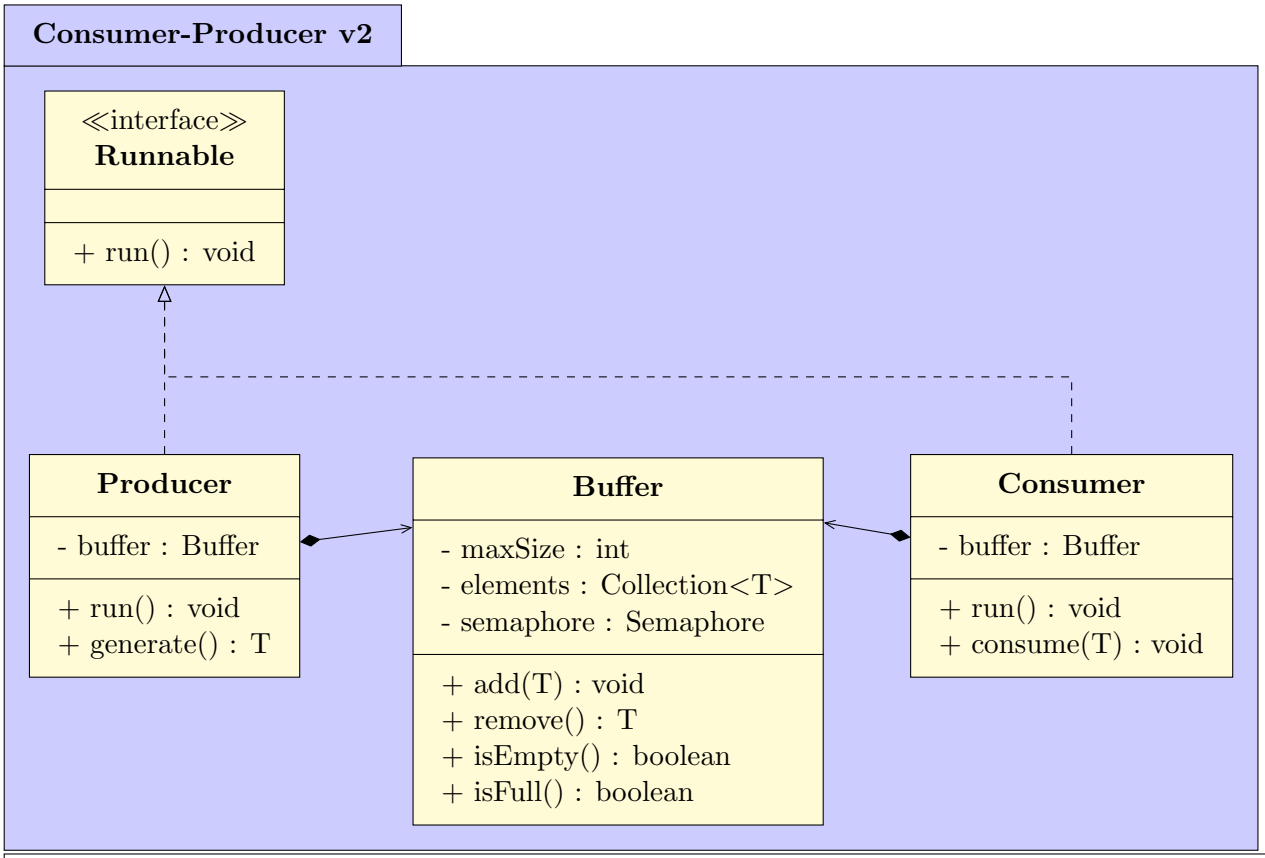
Q: Let's implement the add method for the Buffer.

A:

Q: Okay, now implement the remove.

A:

Here's the updated figure



Q: Let's fix the Producer's `run` method. Hint: significantly easier now!

A:

```
public void run() {
    while (true) {
        T produced = this.generate();
        this.buffer.add(produced);
    }
}
```


Q: Now what about Consumer's?

A:

Q: How can I simplify this in Java?

A:

24

I found out about these classes from the Java Revisited blog [2].

[24] API: `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/LinkedBlockingQueue.html`

Q: What will our Client code look like, then? (Use the String example again.)

A:

15 Parallel OODP: Master-Worker

[25]

Master-Worker is a classic parallel design pattern[1], though the specifics vary a bit. It is also known as the Master-Slave pattern and the SPMD (Single Program; Multiple Data) pattern.

15.0 Motivation

Q: What should we do if we have one big computation to solve and lots of available hardware threads?

A: Split the problem up into little pieces. If we can't divide up the problem into smaller concurrent problems, we can't use this pattern.

Q: If we divide it up into m pieces and have p threads, is it a problem if $m >> p$?

A: Not really.

[25] Thanks a ton to Dale Skrien for helping me work out the details of this!

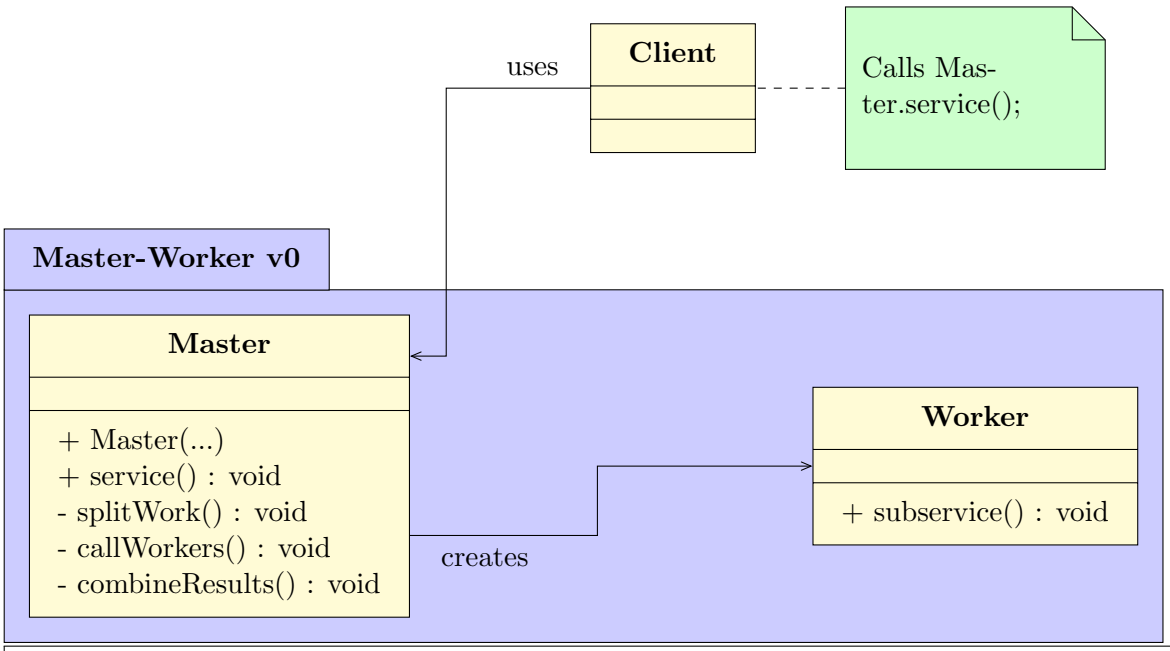
Q:	What are we going to do?
A:	Give each thread a piece to solve. When they're done, give it another one.

Q:	What are we hiding under the rug here?
A:	The actual part to break-down the problem into little pieces.

Q:	And what happens when all the little pieces are done?
A:	We have to put them back together.

15.1 Initial OO Master-Worker

The original object-version of this pattern was given in [1]. It has the following parts:



Q: What's the purpose of the three private methods in Master?

A: To do the three things we talked about:

splitWork : divides the work up into small pieces. (Presumably this sets up a data structure with all the data for the subcomputations.

callWorkers : assigns the data pieces to the Workers, then calls their `subservice` method, until all the data is processed.)

combineResults : takes the results of each subcomputation and builds the final data result.

15.2 First Improvements

Q: What's a problem with `Master.service`?

A: It doesn't return anything.

Q: Okay, let's make it fruitful. What should it return?

A:

Q: Even better?

A:

Q: How else should we change service?

A: Give it a parameter.

Q: And what type should that be?

A:

Q: Can we write the body of `Master.service()`?

A: Yes!

Q: Write it!

A:
```
public Out service(In input) {
    this.splitWork(input);
    this.callWorkers();
    return this.combineResults();
}
```

Q: What is `splitWork` going to do if it's void?

A: It will have to assign the results of the splitting to a field! Okay, let's add a field: `workerInputs`

Q: How is `combineResults` going to work without taking any parameters?

A:

Q: Since we know the necessary methods in Master and Worker, how can we take the first step to improving these patterns?

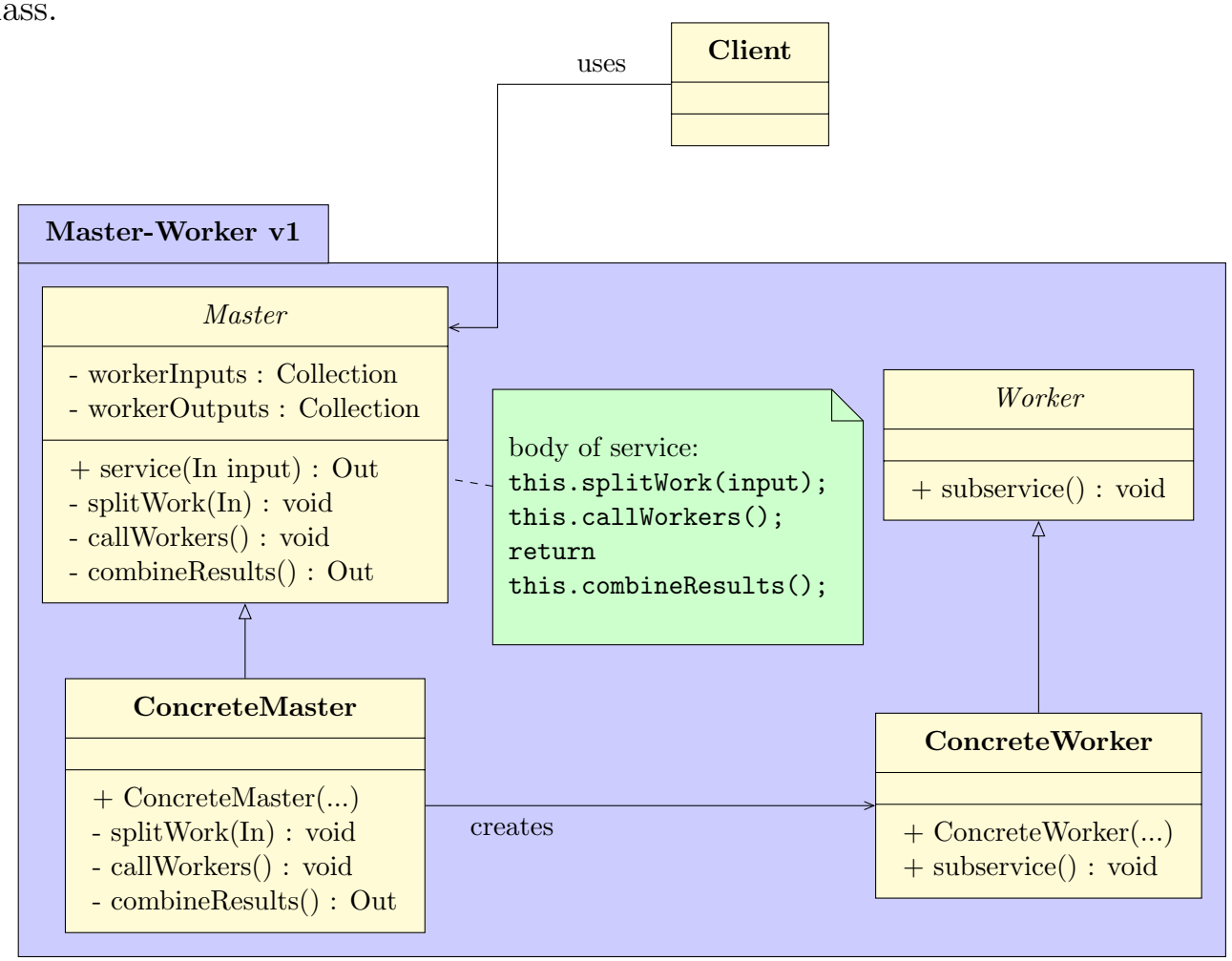
A: Use abstract classes.

Q: Can we implement any of the methods in the abstract classes?

A:

Here are the improvements we've made so far:

TODO: generic types are not working in the title of an abstract class.



15.3 Using the Command Pattern

Q: Which part of this is reminiscent of the Command Pattern (Section 10)?

A: Worker

Q: What class should we replace Worker with (in Java)?

A: Runnable

[26]

Q: What are the added benefits of using Runnable?

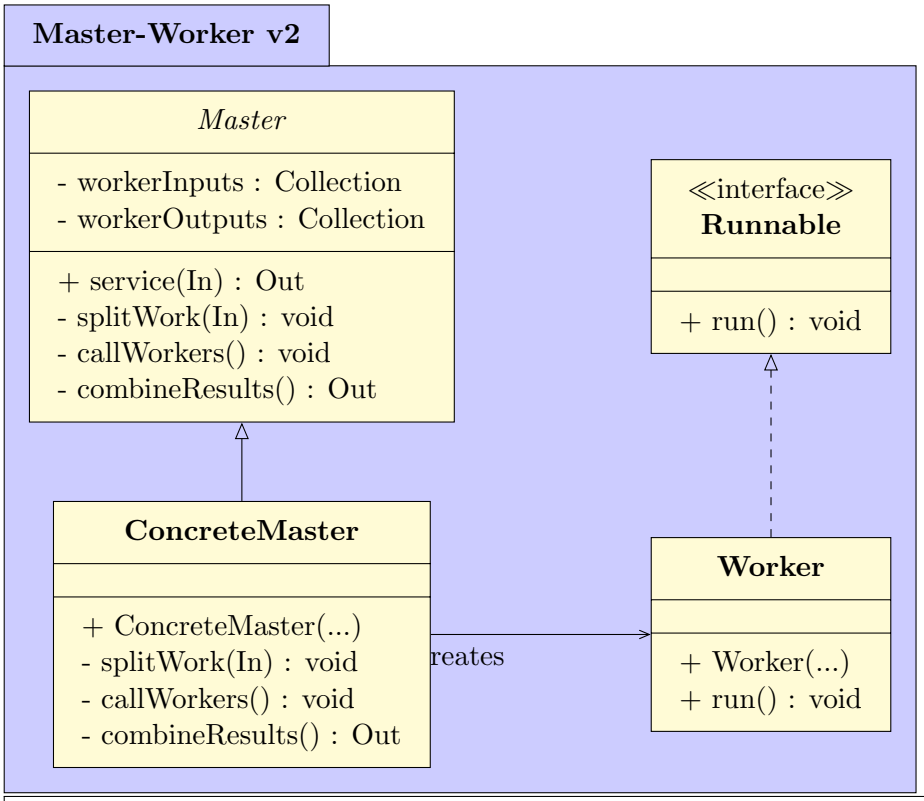
A:

Q: What changes do we have to make to replace Worker with Runnable?

A:

We'll also change ConcreteWorker to just be Worker. Here's the most recent version of our design. (Notice, I'm leaving the Client out from this point on.)

[26] `https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html`



Q: What does the code look like to create and dispatch a Worker?

A:
```
Worker worker = new Worker(...);
new Thread(worker).start();
```

15.4 Double-down on Command

Q: How different are each of the Workers going to be?

A: Not very. Going to run the exact same code...
but on different input.

Q: What design pattern could we use here?

A:

Q: Again??

A:

Q: Who will own this command?

A:

Q: What should ConcreteMaster.`callWorkers` look like?

A:

```
private void callWorkers() {
    for (Object datum :
dataForWorkers) {
        Worker worker = new
Worker(this.command, datum);
        new Thread(worker).start();
    }
}
```


Q: The Object variable is a bit awkward. How can we clean this up using generics?

A: Include Generics for input/output for the workers. Let's use WIn and WOut.

Q: Okay, let's fix ConcreteMaster.`callWorkers()`.

A:

Q: Should the Worker's command field be Runnable, Callable, or something else?

A:

27

[27] Callable is the version of Runnable that returns a value: `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Callable.html`

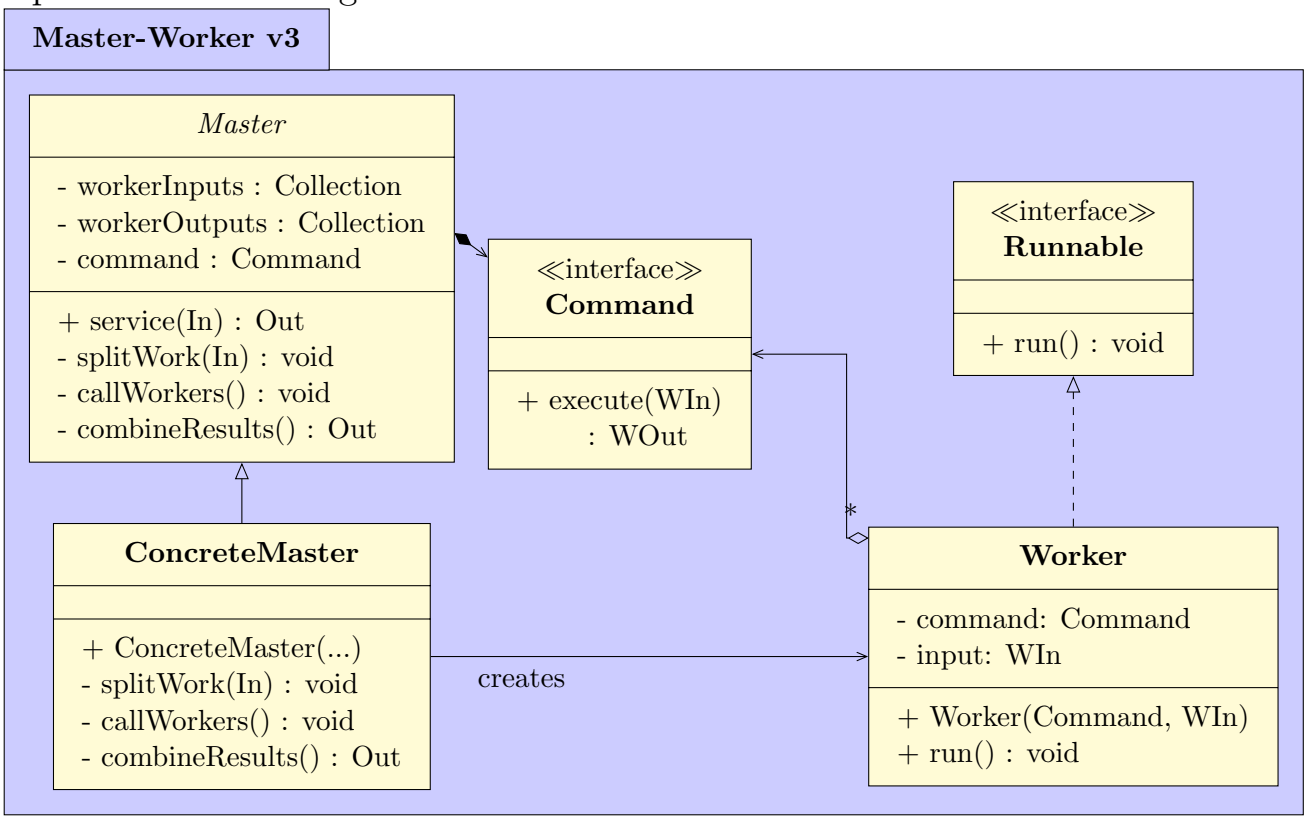
Q: Returning is nice. Why can't the Worker extend Callable instead of Runnable?

A:

Q: It's really important for the Command to either have no state or be immutable. Why is that?

A: The same Command object is used for all the worker commands. If any fields are changed during the `execute` method, then we can have some race conditions!

Update the class diagram:



15.5 Giving Back

Q: If the worker's run method is void, how will we hand data back to the ConcreteMaster?

A: Include a method in ConcreteMaster that accepts the finished work.

Q: How will that method put the work back in the proper place?

A: It will have to know the location in the data structure to return the information. E.g.:

```
public void returnData(WOut datum, int
index)
```

Q: What are some problems with this?

A:

Q: Can we do better?

A: Yes!

Q: How?

A:

Q: What other methods does this capsule object need?

A:

Q: What is the problem with these? Specifically, isFull?

A: Polling.

Q: What's the solution to polling?

A:

Q: Okay, so what methods will we add to the Capsule class?

A:

Q: Which methods can we remove then?

A:

Q: What's the signature of this class going to be?

A:

Q: Let's implement some of these! First off: the constructor.

A:

```
public Capsule() {
    this.isFull = false;
    this.value = null;
    this.listeners = new
LinkedList<ActionListener>();
}
```


Q: Next, fill()

A:
```
public void fill(T value) {
    this.value = value;
    this.isFull = true;
    this.notify();
}
```

Q: Finally, getValue()

A:
```
public T getValue() {
    if (this.isFull) {
        return this.value;
    } else {
        throw new
RuntimeException("Tried to get the
value from an empty capsule!");
    }
}
```

Q: What about the add(Observer) method?

A: Just like the one shown in the Observer pattern. For Java, we can use the ActionListener interface, so it'll be `addActionListener(ActionListener)`:
```
public void addActionListener(ActionListener
listener) {
    this.listeners.add(lisetener);
}
```


Q: So what about the notify method?

A:

```
public void notify() {
    for (ActionListener listener :
this.listeners) {
        listener.actionPerformed(new
ActionEvent(this, 0, ""));
    }
}
```

TODO: add another iteration of the class diagram here

Q: I think we can finally implement the Worker class! First up: constructor, which is extremely boring.

A:

```
public Worker(Command command, WIn
input, Capsule capsule) {
    this.command = command;
    this.input = input;
    this.capsule = capsule;
}
```

Q: Can we write the Worker.`run` method yet?

A:

Q: What could be a problem with the data structures for the worker inputs and outputs?

A:

Q: Do they need to be the same?

A:

Q: What should we do here?

A:

Q: Okay, what do we need to add to our capsule class?

A:

Q: How can this improve the Worker class's constructor?

A:

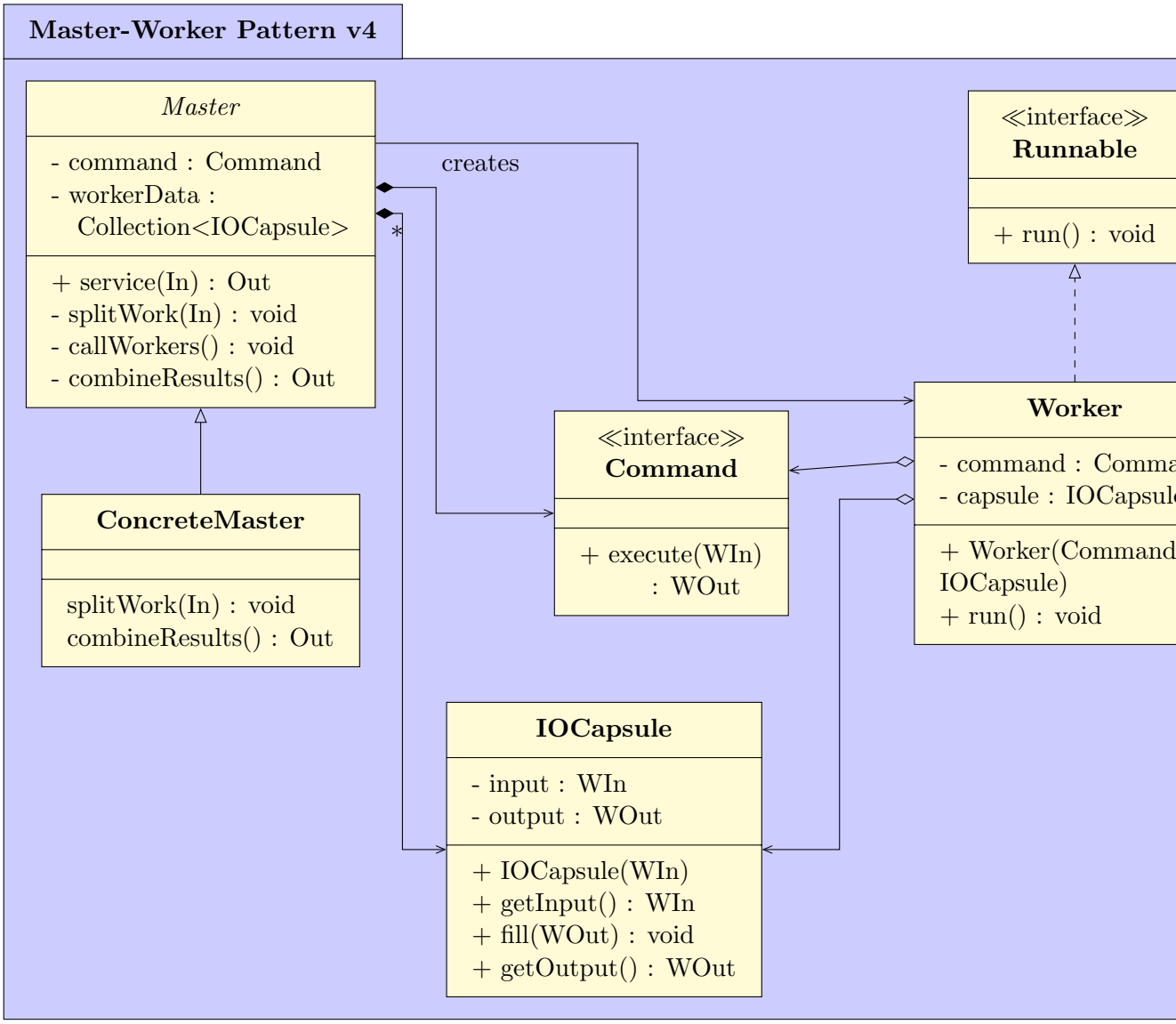
Q: So what method do we need to update?

A:

Q: Update it!

A:

⟨ Make sure the class diagram matches this: ⟩



15.6 Waiting for Workers

Q: Last big issue: There's a weird part in our old `service` method. What's wrong with this?

A: We need to wait for the workers to finish before calling `combineResults`. Right now it doesn't wait!

Q: How do we get the code to wait? (Either at the beginning of `combineResults` or at the end of `callWorkers`)

A:

Q: What is a Semaphore?

A:

Q: What is the interface for a Semaphore?

A:

- Constructor: `Semaphore(numTokens : int)`
- `p()`: asks to "borrow" a token. If none is available it blocks until one becomes available.
- `v()`: returns a token. Sometimes it's okay for code that never borrowed a token to return one.

[28][29]

Q: Which class will have a Semaphore field?

A: Master

Q: When will it be created?

A:

[28] `p` is short for the Dutch *proberen* (to try out); `v` is short for *verhohen* (to increase)

[29] Java's Semaphore implementation (`https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html`) uses `acquire()` and `release()` as `p()` and `v()`, respectively.

Q: Let's implement the following strategy:

- Start with a semaphore with lots of tokens.
- Each worker requests a token before it starts work, then returns it when it's done.
- Before we call `combineResults`, we need to request tokens again so that doesn't go until all workers have finished.

How many tokens should we include initially?

A:

Q: How many tokens do we need to request at that end part?

A:

Q: Is there a benefit to putting all of this synchronization code into `callWorkers`?

A:

Q: Okay, so we want all of the workers to take a token when they launch, and return it when they're done. Where should we put another call to `p()`?

A:

Q: Okay, when should the other calls to `p()` and `v()` happen?

A:

Q: Okay, where do I put that call to `p`?

A:

Q: What about the call to `v`?

A:

Q: What does that mean about the workers?

A:

Q: Let's update the `callWorkers` method!

A:

Q: Why does there have to be a separate loop at the end? Why can't I just put the call to `acquire` in at the last line of the loop?

A:

Q: Let's update Worker.`run`?

A:

Q: We can actually make our code run faster by removing some calls to `acquire` (and modifying the semaphore constructor). How can we do that?

A:

Q: What does the updated method look like now?

A:

Q: There are two ways to remove that last loop and replace it with a single call to `acquire`. What are those ways? (Hint: I had to look at the Semaphore API to make sure both were legal.)

A:

Q: What's the code for the negative tokens version? (Hint: how many negative tokens do you need to initialize with?)

A:

Q: Do we still need to use the Observer Pattern?

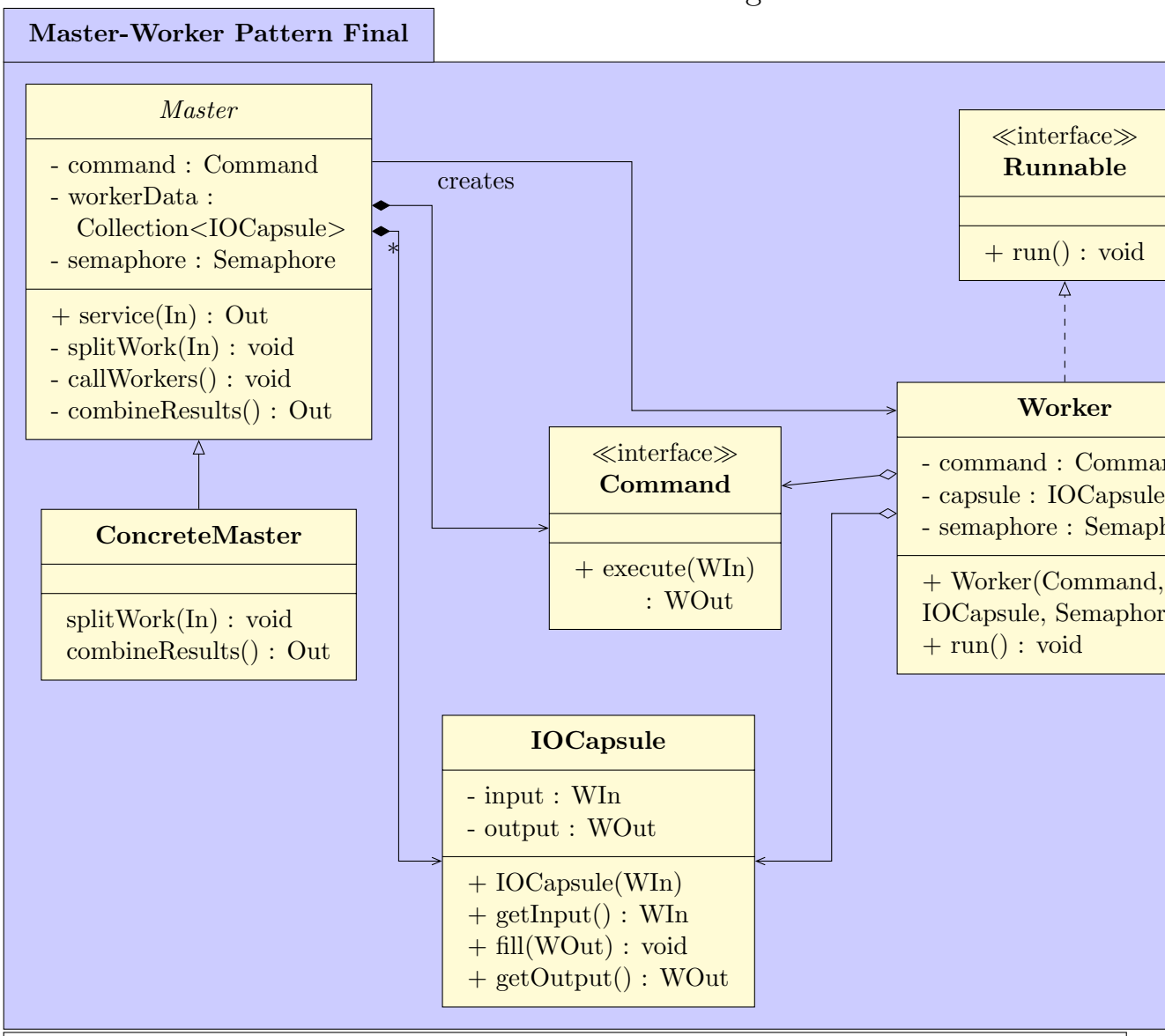
A:

Q: Okay, so what was the thing about the binary tree to recombine?

A: There's a bit of a bottleneck in terms of launching all the threads: it all uses a single loop. What if instead, we had a binary tree where the leaves had the data for the workers. Then we could launch them in parallel from the root in $\Theta(\log(n))$ steps. We'd have to have the recombining done in that much time too, though. Definitely a good plan, but that's for another time...

15.7 Master-Worker: Summary

Sweet! We did a lot of work! Here's the final class diagram:



Q: What are the benefits of this pattern?

A:

- Everything runs in parallel.
- No possibilities of data structure errors.
- User only has to implement a little bit.

Q: What exactly does the user have to implement?

A:

16 Parallel OODP: Pipeline Pattern

Requirements: need to have already seen the BlockingQueue Java stuff from the Producer/Consumer pattern.

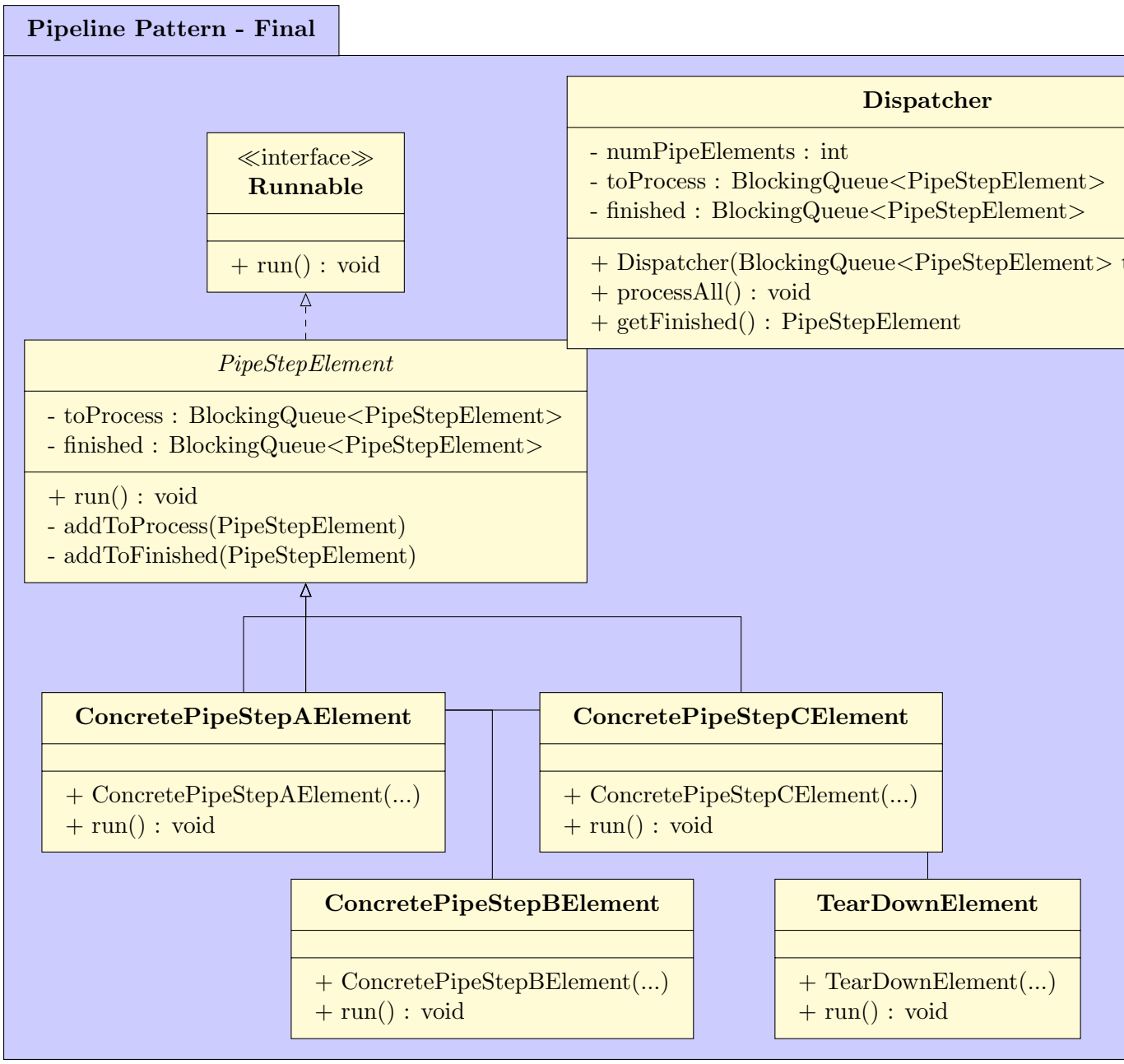
TODO: flesh out this section a bunch more. I don't remember what I did, but it went really well. I started with the state pattern.

16.0 Pipeline Pattern based on State Pattern Paper

I built this all off of the paper I read many years ago... an earlier version of this one by MacDonald, Szafron, and Schaffer: "Rethinking the Pipeline as Object–Oriented States with Transformations" (`http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.150.4556&rep=rep1&type=pdf`)

Talk about this stuff. Make some drawings on the board. We don't want an actual pipeline, just the stages in a queue that get run. Note: we have to make sure that Order Does Not Matter. (Otherwise, this doesn't really work.)

16.1 Final Code and Diagram



TODO: add the Client to the picture. The Client creates the toProcess queue, puts all the initial pipe steps in there, then calls processAll on the Dispatcher, then calls getFinished to get them all out.

Here's the code for the processAll method in the Dispatcher class:

```
public void processAll() {
  while (this.numPipeElements > 0) {
```



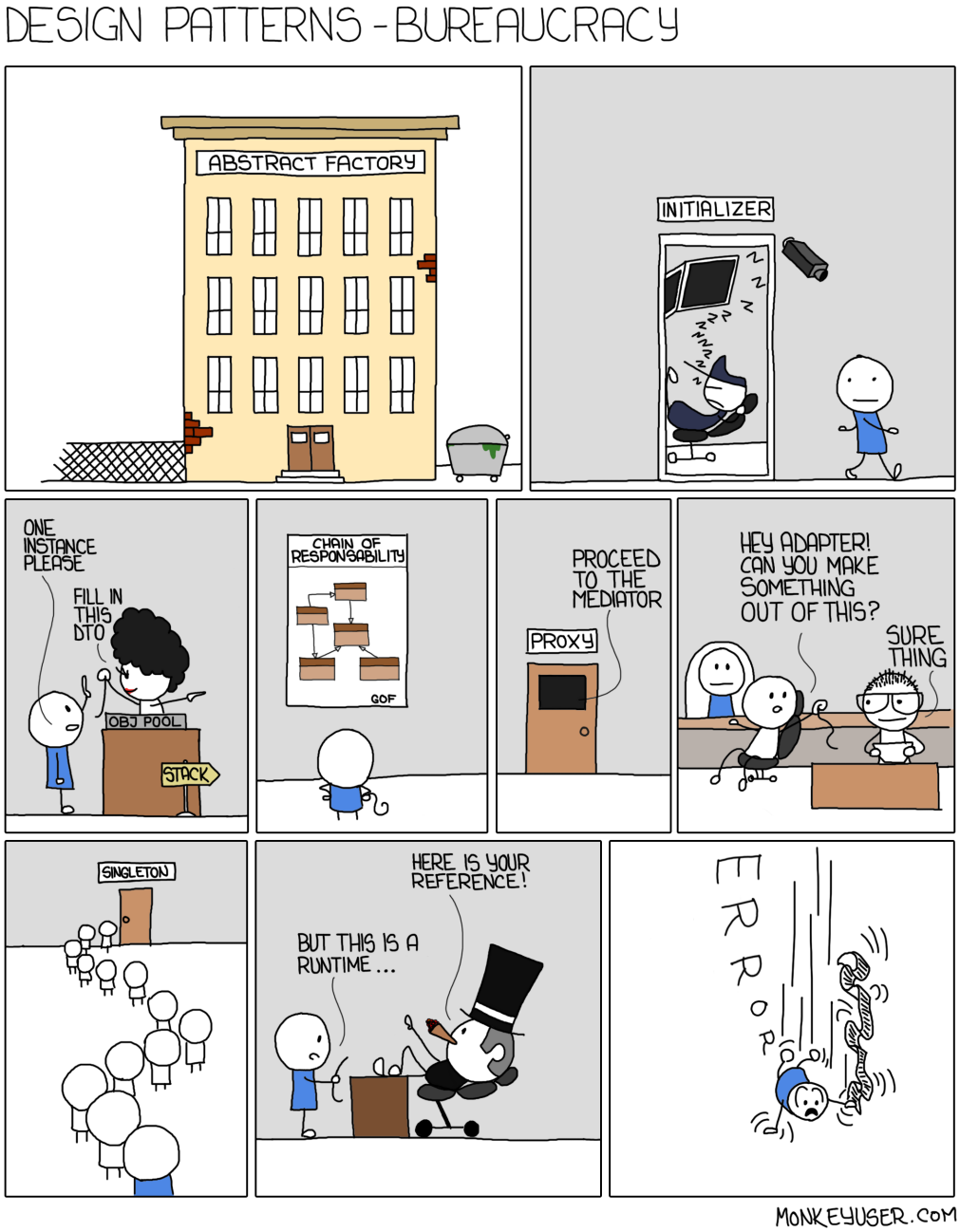
```
            PipeStepElement element = this.toProcess.take();
            new Thread(element).start();
        }
    }
```

Here's the code for getFinished, also in the Dispatcher class:

```
    public synchronized PipeStepElement getFinished() {
      PipeStepElement finished = this.finished.take();
      this.numPipeElements --;
      if (this.numPipeElements <= 0) {
          this.toProcess.put(new TearDownElement());
      }
      return finished;
    }
```

17 Evolution of Design Patterns

17.0 Downsides of Design Patterns



30

There are some arguments against Design Patterns. One is: it creates too much repeated code. If I implement a complicated pattern in 12 places, then there is going to be lots of repeated code.

[30]Monkey User comic "Design Patterns Bureaucracy, from `http://www.monkeyuser.com/2017/design-patterns-bureaucracy/`

Q: What's the solution?

A:

17.1 OODP and PL

New languages and/or language features often pop up to simplify or remove common code patterns.

TODO: talk about how this works with non-OO patterns. E.g. evolution of Java for-each loops.

Q: Which patterns have already been integrated into Java?

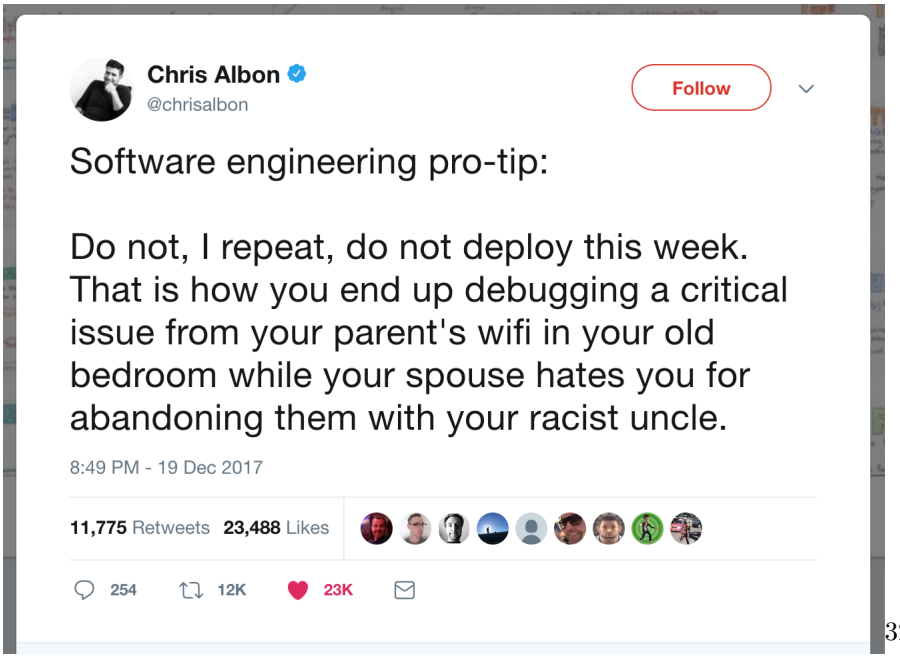
A:

The Master-Worker pattern, covered in **??**, is completely unnecessary in some languages, e.g. Chapel[31].

A Little Holiday Software Joke

I usually arrive here—the end of the notes—at the end of the fall semester. The midst of the holiday season. The following tweet is excellent advice for budding software developers.

[31] `http://chapel.cray.com`



[32]

Appendices

A Java Programming with Objects

⟨ Do Basic OOD pages 8 - 15 ⟩

Q: What if we have the `getName` method implemented in both Primate and Monkey. Which of those two does the following snippet call?

```
Primate primeape = new Monkey(‘‘Primeape", 057);
primeape.getName();
```

A:

[32] Source: `https://twitter.com/chrisalbon/status/943342608742604801`

Q: What is this called?

A:

A.0 Downcasting

Q: Let's return to the `primeape` example. What if we later want to get `primeape`'s `Bananas`?

A:

Q: What will that code look like?

A:

Q: What could be the result if we're not careful?

A:

Q: Workaround?

A:

Two red flags just jumped up:

- `instanceof`: Means you're probably not using Polymorphism when you should. Leads to repeated, ugly conditionals!
- "Workaround" Yuck! That means we're not being elegant.

A.0.1 Java Generics

Sometimes Programming Language changes can aid elegance. Prior to Generics (1.4, say) a Java snippet might look like:

```
ArrayList stolenBananas = monkey.getBananas();
Object element;
Banana banana;
for (int i = 0; i < stolenBananas.size(); i++) {
    element = stolenBananas.get(i);
    banana = (Banana) element;
    monkey.peelAndEat(banana);
}
```

Q: Would Java complain about any of this at compile time?

A:

Q: Why is this a problem?

A:

Q: How do generics solve this problem?

A:

Q: Assume now that `getBananas` returns an `ArrayList<Banana>`. How can we rewrite the code?

A:

Q: Which of our elegance criteria does this improve?

A:

Q: Is this new stuff worse in any way?

A:

That programming pattern became very common. Even before for-each loops existed, iterators were devised to abstract away the need for reliance on linear-shaped data structures.

```
ArrayList<Banana> stolenBananas =
monkey.getBananas();
for (Iterator<Banana> bananaIterator =
stolenBananas.iterator(); iterator.hasNext(); )
{
    //leave a space here
    monkey.peelAndEat(iterator.getNext());
}
```

Q: Why is this an improvement?

A:

Q: Any downsides?

A:

Q: What do we have to change to make this extremely reusable?

A:

Q: How does this change make our code more Extensible/Maintainable?

A:

Q: Let's rewrite the snippet using a for-each loop!

A:

Q: How is this an improvement?

A:

Q: Is there any more room for improvement here?

A:

Q: What do you think that signature is?

A:

Q: But... Monkeys can also eat Oranges! Should we have two methods?

A:

Q: How can we combine them?

A:

The code might look like this:

```
public void peelAndEat(Fruit fruit) {
    fruit.peel();
    this.eat(fruit);
}
```

Great! Now we're using lots of inheritance.
⟨ Draw a little class diagram: Banana and Orange are subclasses of Fruit! ... and so is Watermelon. ⟩

Q: Any problems?

A:

Q: What might we want to do?

A:

A.1 Summary

Q: What are some basic Heuristics (rules of thumb) we've learned so far? (Some might be from previous courses too!)

A:

References

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented Software Architecture: a System of Patterns, Volume 1.* Number v. 1. John Wiley and Sons, 1996.

[2] J. Paul. Producer consumer design pattern with blocking queue example in java, 2012. `http://javarevisited.blogspot.com/2012/02/producer-consumer-design-pattern-with.html`.

[3] Dale Skrien. *Object-Oriented Design Using Java (1. ed.).* McGraw-Hill Education, 2008.

