

Object-Oriented Paradigm

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 4

Acknowledgement & Materials Copyright

- I'd like to start by acknowledging Dr. Ken Anderson
- Ken is a Professor and the Chair of the Department of Computer Science
- Ken taught OOAD on several occasions, and has graciously allowed me to use his copyrighted material for this instance of the class
- Although I will modify the materials to update and personalize this class, the original materials this class is based on are all copyrighted © Kenneth M. Anderson; the materials are used with his consent; and this use in no way challenges his copyright

Learning Objectives

- Introduce some references used for class
- Discuss the class experience with OO development
- Introduce the object-oriented paradigm
 - Contrast it with functional decomposition
 - Discuss important concepts of object-oriented programming
 - Discuss the difference between abstraction and encapsulation
 - This is VERY important
- Address the problem of requirements and the need to deal with change

Paradigm

- Philosophical/theoretical framework for theories, laws, generalizations
 - <https://www.merriam-webster.com/dictionary/paradigm>
- Patterns of thought and sets of supporting information

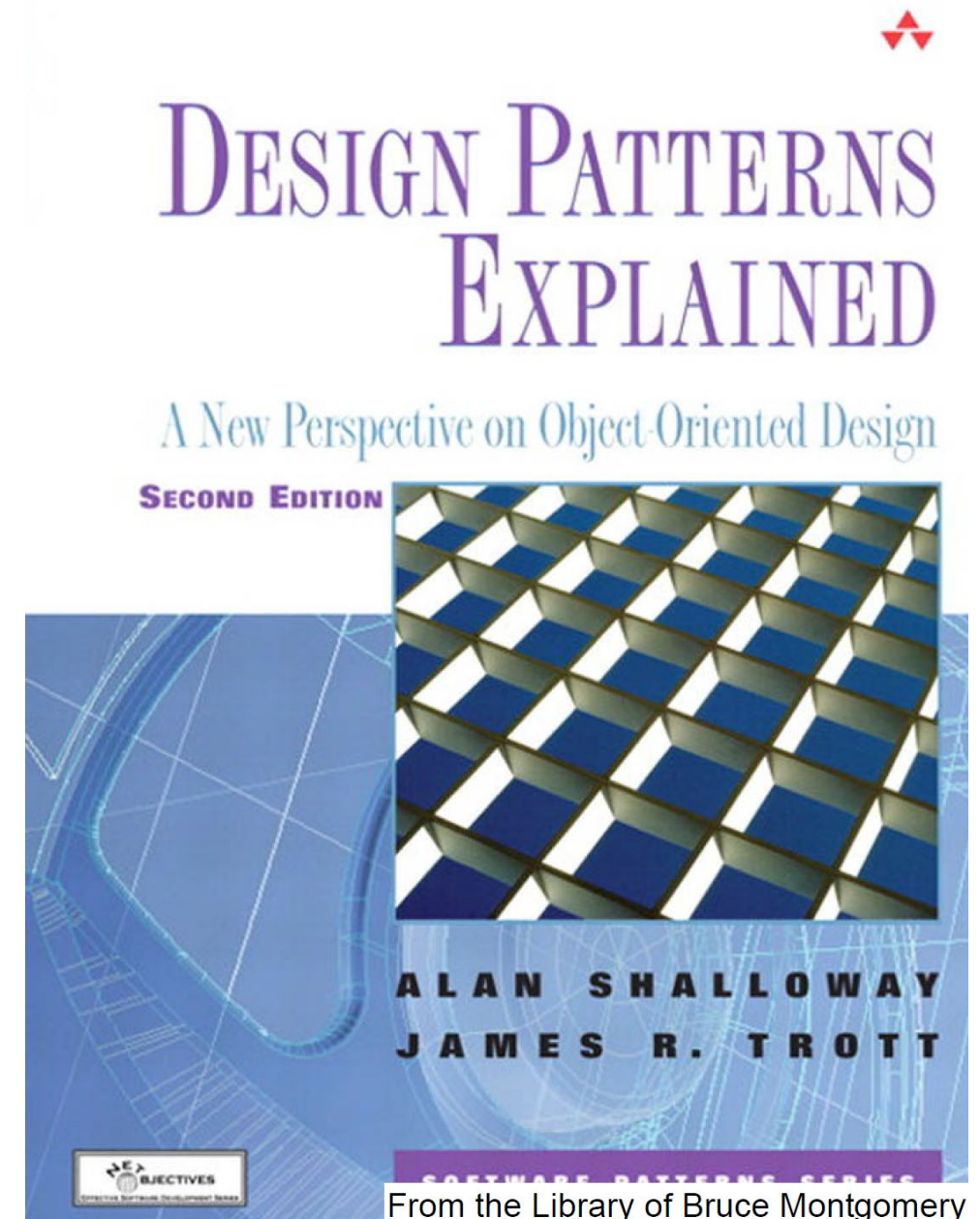
OO Resources

- Chunk of my home book shelf for UML, Patterns, and OO topics
- Design Patterns by Gamma, Helm, Johnson, and Vlissides is the famous “Gang of Four” book
- Gamma and Helm met at OOPSLA 1990 considering an architecture handbook, Johnson and Vlissides joined for the initial book published in 1994



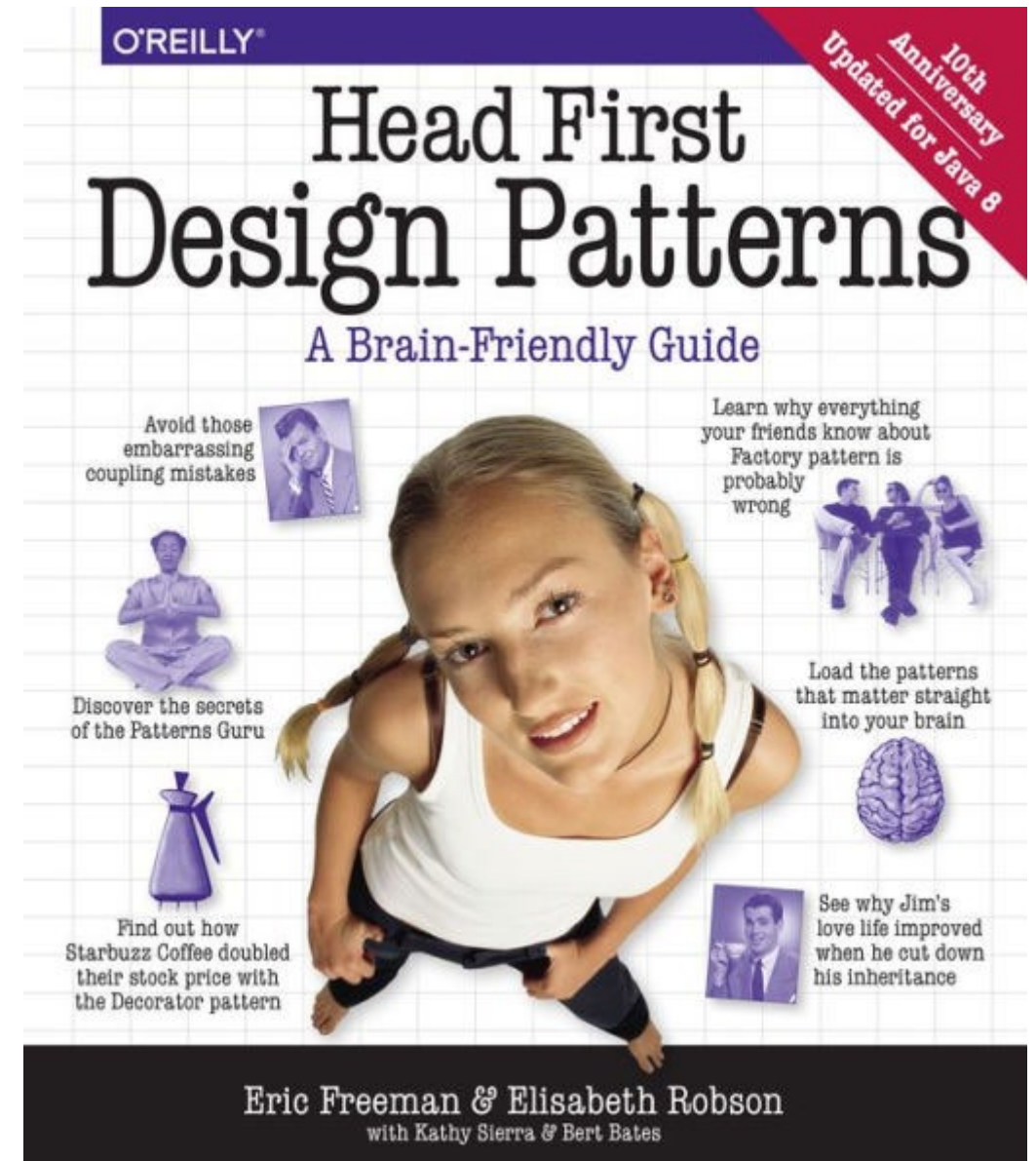
Prior Textbook

- Design Patterns Explained
 - A New Perspective on Object-Oriented Design, Second Edition
- Alan Shalloway and James R. Trott
- Addison Wesley, © 2005
- This was the textbook for some time
- It's a good book, and I will pull some examples from it, but we spent more time talking about examples from Head First Design Patterns, so I went there instead
- If I reference this, I'll call it "Shalloway/Trott"



Our Textbook

- Textbook:
 - Head First Design Patterns
 - By Bates, Sierra, Freeman, & Robson
 - O'Reilly Media
 - 10th Anniversary Edition - 2014
(updated for Java 8)
- I will be visiting most of the content in this book, maybe not in order
- Great examples, clear ties to OO principles
- Updated in 2014 to Java 8 examples (Java 8 is in LTS to March 2025) but the content didn't change much
- “Cool” poster also available out there
- I'll call this “Head First”



What can you do? (Show of hands)

- How many people are comfortable starting from scratch and creating:
 - a script?
 - a desktop application?
 - a web service?
 - a mobile application?
 - a system of systems? (i.e. desktop plus web service)
 - a database-backed application?
 - a cloud-based application?
 - IoT app on an SBC?
 - an embedded device?

Discussion

- When you create a program from scratch:
 - do you use OO techniques?
 - OO design heuristics?
 - design patterns?
- If not, what style of software design do you use?
 - What styles of software design are you aware of?

Design Methods

- Ways of solving problems
 - **Structured Design/Programming** (a.k.a. **functional decomposition**)
 - “Think in terms of steps”
 - Example languages – C, Pascal, FORTRAN
 - **Object-Oriented Design/Programming**
 - “Think in terms of objects that do things”
 - Example languages – C++, C#, Java, Python, Objective-C
 - **Functional Programming** (a.k.a. **declarative programming**)
 - “Think in terms of functions and their composition”
 - Example languages – Mathematica, Lisp, Erlang, F#

Simple Problem: Display Shapes

- **Functional decomposition: break problem into small steps**
 - Connect to database
 - Locate and retrieve shapes
 - Sort the shapes (perhaps by z-order; draw background shapes first)
 - Loop through list and display each shape
 - Identify shape (circle, triangle, square?)
 - Get location of shape
 - Call function to display the shape at the given location

Functional Decomposition

- **Decompose big problems into the functional steps required to solve it**
 - For a very big problem, simply break it down to smaller problems
 - then decompose smaller problems into functional steps
- Goal is to **slice up the problems** until they are at a **level of granularity** that is **easy to solve in a couple of steps**
 - Then arrange the steps into an order that solves all of the identified subproblems and, presto, the big problem is solved along the way
- Extremely natural approach to problem solving; we do this almost without thinking about it

Functional Decomposition: Problems

- There are two main problems with this approach to design
 - It creates designs **centered around a “main program”**
 - This program is in control and knows all of the details about what needs to be done and all of the details about the program’s data structures
 - It creates designs that **do not respond well to change requests**
 - These programs are **not well modularized** and so a change request often requires modification of the main program; a minor change in a data structure, for example, might cause impacts throughout the entire main program

Why do these problems exist?

- These problems occur with the functional decomposition approach because the resulting software exhibits
 - poor use of **abstraction**
 - poor **encapsulation** (a.k.a. **information hiding**)
 - poor **modularity**
- If you have poor abstractions and you want to add another one, it's often not clear how to do it (easily)
- If you have poor encapsulation and poor modularity, changes tend to percolate through the code since nothing **prevents dependencies from forming throughout the code**

Why should we care?

- Head First: One constant is software development: Change
- Shalloway/Trott book:
 - **“Many bugs originate with changes to the code”**
- and
 - **“Things change. They always do. And nothing you can do will stop change [from occurring to your software system].”**
- We need to ensure that we do not get overcome by change requests; that we create designs that are resilient to change;
- Indeed, we want software designs that are “designed” to accommodate change in a straightforward manner
- That is what OO A&D provides!

Abstraction & Encapsulation

- Simple Definitions
 - **Abstraction** refers to the **set of concepts that some entity provides you in order for you to achieve a task or solve a problem**
 - Simple Example: the public methods of Java's String class
 - **Encapsulation** refers to a **set of language-level mechanisms or design techniques that hide implementation details** of a class, module, or subsystem from other classes, modules, and subsystems
 - Simple Example: In most OO programming languages, marking an instance variable "private" ensures that other classes cannot access the value of that variable directly
 - They need to make use of a method in order to retrieve or update that particular internal variable

Design...

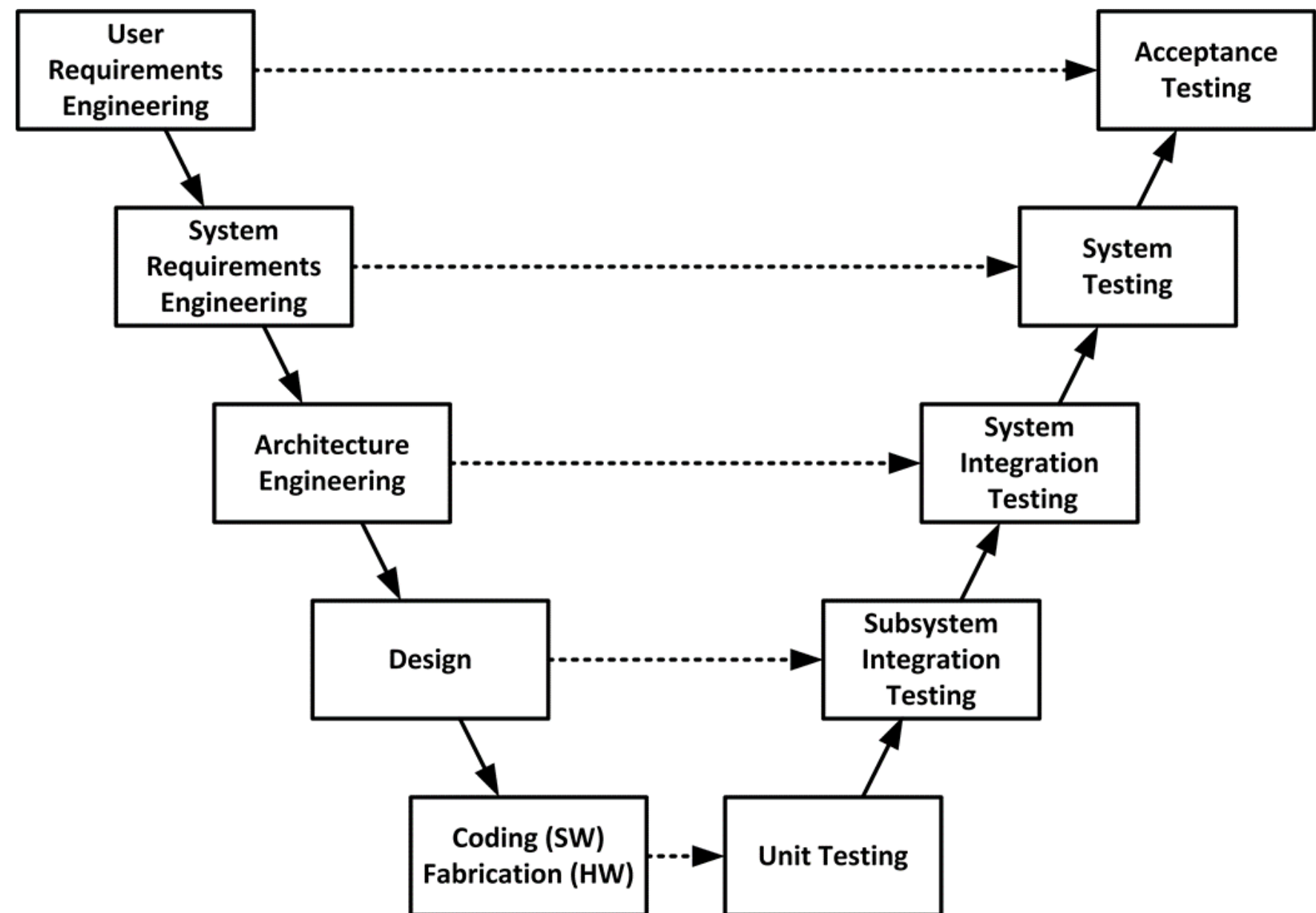
- Let's transition back to design methods
 - Discussion of Analysis and Requirements
 - Additional Problems with Functional Decomposition
 - Cohesion and Coupling
 - The OO Approach

Analysis and Requirements

- Analysis is the phase of software development that occurs
 - before design when starting from scratch
 - that occurs first when responding to a change request during the maintenance of an existing system
- Its primary goal is to answer the following question
 - **What is the problem that needs to be solved?**
- Design is the phase that comes after analysis and its goal is:
 - **How am I going to solve the problem?**
- Requirements for a software system are initially generated during the analysis phase of software development and are, typically:
 - **Functional**
 - “the system should allow its users to sort records by priority”
 - **Non-functional**
 - “the system should support 10,000 simultaneous users”
 - **Constraints**
 - “the system will comply with regulation XYZ at all times”

Levels of Requirements (“The V”)

- Progressive elaboration...
- Deeper levels of design requirements for more discrete elements of the application
- Best Practices for Requirements
 - **Specific** and distinct requirements
 - **Complete** and well thought out
 - **Consistent and prioritized** based on the objectives
 - **Able to be verified** during testing
- A road of good intentions...



https://insights.sei.cmu.edu/sei_blog/2013/11/using-v-models-for-testing.html

The Problem of Requirements (I)

- The problem? Experienced developers will tell you that
 - **It can be hard to elicit requirements**
 - **Requirements are incomplete and do not tell the whole story**
 - **Requirements are often wrong**
 - factually wrong or become obsolete
 - **Requirements and users are misleading**
- In addition, users may be non-technical and may not understand the range of options that could solve their problem
 - their ill-informed suggestions may artificially constrain the space of solutions

The Problem of Requirements (II)

- The other problem with requirements is
 - “requirements **always** change”
- They change because
 - a user’s needs (or the product) may change over time
 - as they learn more about a new problem domain, a **developer’s ability** to **generate better solutions** to the original problem (or the current problem if it has evolved) **will increase**
 - **the system’s environment changes**
 - new hardware, new external pressures, new techniques

The Problem of Requirements (III)

- Many developers **view changing requirements as a bad thing**
 - and few design their systems to be resilient in the face of change
- Luckily, **this view is changing**
 - **agile software methods tell developers to welcome change**
 - they recommend a set of techniques, technologies and practices for developers to follow to remove the fear of change
 - OO analysis, design and programming techniques **provide you with powerful tools to handle change** to software systems in a straightforward manner

The Problem of Requirements (IV)

- However, this does not mean that we stop writing requirements
 - They are incredibly useful despite these problems
 - **The lesson here is that we need to improve the way we design our systems and design our code such to allow change to be managed**
- Agile methods make use of “user stories”; other life cycle methods make use of requirements documents or use cases (scenarios that describe desired functional characteristics of the system)
 - Once we have these things, **and the understanding of the problem domain that they convey**, we then have to design our system to address the requirements while leaving room for the requirements to change

The Problem with Functional Decomposition

- A common problem with code developed with functional decomposition
 - **weak cohesion** and **tight coupling**
 - translation: “it does too many things and has too many dependencies”
- Example
 - `void process_records(records: record_list) {`
 - `// sort records, update values in records, print records, archive records and log each operation as it is performed ...`
 - `}`

Cohesion

Code Complete 2 by Steve McConnell;
Microsoft Press, 2004

- Cohesion refers to “how closely the operations in a routine are related”
 - A simplification is to say “*we want this method to do just one thing*” or “*we want this module to deal with just one thing*”
- We want our code to exhibit **strong cohesion** (a.k.a. highly cohesive)
 - methods: the method performs one operation
 - classes: the class achieves a fine-grain design or implementation goal
 - packages: the package achieves a medium-grain design goal
 - subsystems: this subsystem achieves a coarse-grain design goal
 - system: the system achieves all design goals and meets its requirements

Coupling

Code Complete 2 by Steve McConnell;
Microsoft Press, 2004

- Coupling refers to “the strength of a connection between two routines”
 - It is a complement to cohesion
 - weak cohesion implies strong coupling
 - strong cohesion implies loose coupling
- With strong or tight coupling, a single change in one method or data structure will cause **ripple effects**, that is, additional changes in other parts of the system – often unwanted side effects from change
- We want systems with parts that are **highly cohesive** and **loosely coupled**

Transitioning to the OO Paradigm

- Rather than having a main program do everything
 - populate your system with **objects that can do things for themselves**
- Scenario: You are an instructor at a conference. Your session is over and now conference attendees need to go to their next session
 - With functional decomposition, you would develop a program to solve this problem that would have you the instructor do everything
 - Get the roster, loop through each attendee, look up their next session, find its location, generate a route, and, finally, tell the attendee how to get to their next class
 - You would do everything, attendees would do (almost) nothing

Transitioning to the OO Paradigm

- **Would you do this in real life?**
- And the answer is (hopefully) NO!
- What would you do instead?
 - You would assume that everyone has a conference program, knows where they need to be next, and will get there on their own
 - All you would do is end the session and head off to your next activity
 - At worst, you would have a list of the next sessions at the front of the class and you would tell everyone “use this info to locate your next session”

Compare / Contrast

- In the first scenario (functional decomposition),
 - you know everything, you are responsible for everything, if something changes you would be responsible for handling it
 - you give very explicit instructions to each entity in the system
- In the second scenario (object-oriented),
 - you expect the other entities to be self sufficient
 - you give very general instructions and
 - you expect the other entities to know how to apply those general instructions to their specific situation
 - it will be easier to add special cases by adding other independent entities
- OO shifts responsibility away from a central control program to the entities themselves

Foreshadowing

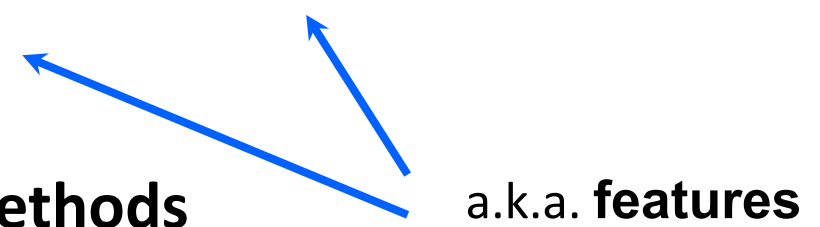
- The benefits we've been discussing are inherent in the OO approach to analysis, design and implementation that we will be learning this entire semester
 - self-sufficient entities $\boxed{\rightarrow}$ objects
 - "give general instructions to" $\boxed{\rightarrow}$ code to an interface
 - "expect entities to apply those general instructions to their specific situation" $\boxed{\rightarrow}$ polymorphism and subclasses
 - "add new attendees without impacting session leader" $\boxed{\rightarrow}$ code to an interface, polymorphism, subclasses
 - shift of responsibility $\boxed{\rightarrow}$ functionality distributed across network of objects

As an aside...

- OO main programs tend to be short
 - On the order of create an object and send a message to it

```
1 import wx
2
3 from ACE.GUI.Managers.RepositoryManager import RepositoryManager
4
5 class ACEApp(wx.App):
6
7     def OnInit(self):
8
9         bmp = wx.Image("images/ace_logo.png").ConvertToBitmap()
10        wx.SplashScreen(bmp, wx.SPLASH_CENTRE_ON_SCREEN | wx.SPLASH_TIMEOUT, 500, None, -1)
11        wx.SafeYield(None, True)
12        self.repoman = RepositoryManager()
13        return self.repoman.HandleAppStart(self)
14
15    def OnExit(self):
16        self.repoman.HandleAppQuit()
17
18 if __name__ == '__main__':
19     app = ACEApp(redirect=False) ← Create an object
20     app.MainLoop() ← Send a message to it
21
```

The Object-Oriented Paradigm

- OO Analysis & Design is centered around the concept of an **object**
 - It produces systems that are **networks of objects** *collaborating* to **fulfill the responsibilities** (requirements) of the system
 - Objects are conceptual units that combine both data and behavior
 - The **data** of an object is referred to by many names
 - **attributes**, properties, instance variables, etc.
 - The **behavior** of an object is defined by its **set of methods**
 - Methods and attributes are **members** of an object (or class)
 - Objects **inherently know what type they are.**
 - Its attributes **allows it to keep track of its state.**
 - Its methods **allow it to function properly.**
- 
- a.k.a. **features**

Object Responsibilities

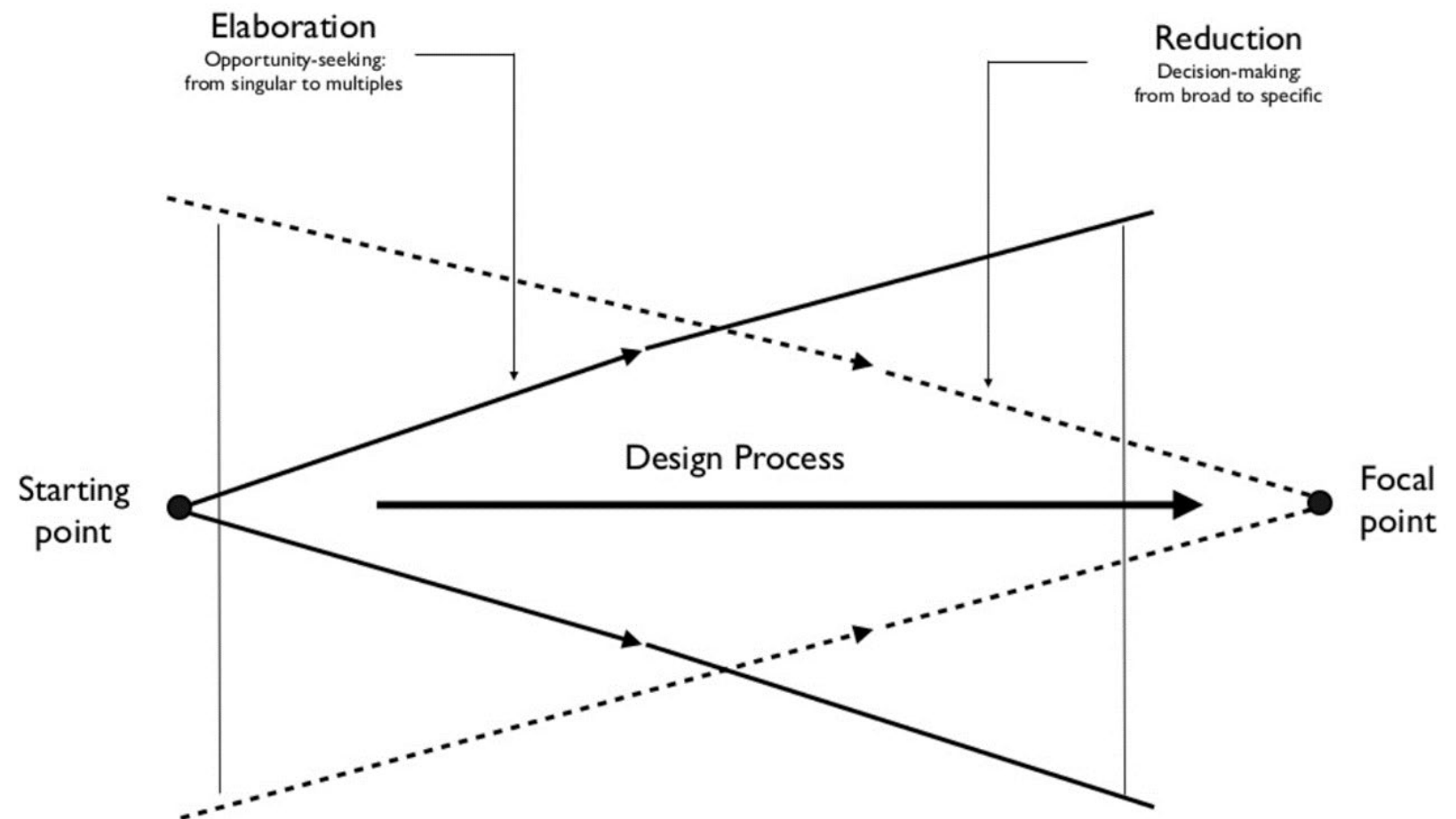
- In OO Analysis and Design, it is best to think of an object as **“something with responsibilities”**
 - As you perform analysis (What’s the problem?), you discover responsibilities that the system must fulfill
 - You will eventually find “homes” for these responsibilities in the objects you design for the system; indeed this process can help you “discover” objects needed for the system
 - The problem domain will also provide many candidate objects to include in the system
 - **This is an example of moving from a conceptual perspective to the specification and implementation perspectives**

Objects and Design Perspectives

- Conceptual — a set of responsibilities
- Specification — a set of methods
- Implementation — a set of code and data
- Unfortunately, OO A&D is often taught only at the implementation level
 - if previously you have used OO programming languages without doing analysis and design up front, then you've been operating only at the implementation level
 - “jumping into code”
 - as you will see, there are great benefits from starting with the other levels first

Benefits of Thinking Conceptually

- Broad thinking in the early phases of **Progressive Elaboration** on a design
- Avoidance of **Premature Optimization** away from possible design choices
- In Object terms – focusing on responsibilities of the system and its elements
- Envisioning this – Bill Buxton's Design Funnel
 - From Sketching User Experiences, Buxton, 2007, Morgan Kaufmann



Aside: Sketching (defined broadly) is a great way to explore many different ideas quickly...

Objects as Instances of a Class

- If you have two Student objects, **they each have their own data**
 - e.g. Student A has a different set of values for its attributes than Student B
- But they both have the **same set of methods**
 - This is true because methods are associated with a **class** that acts as a blueprint for creating new objects
 - We say “**Objects are instances of a class**”
- Classes define the complete behavior of their associated objects
 - what data elements and methods they have and how these features are accessed (whether they are public or private)

Classes, Subclasses, Superclasses

- The most important thing about a class is that it defines a type with a legal set of values
- Consider these four types
 - Complex Numbers \supset Real Numbers \supset Integers \supset Natural Numbers
- Complex numbers is a class that includes all numbers; real numbers are a subtype of complex numbers and integers are a subtype of reals, etc.
 - in each case, moving to a subtype reduces the set of legal values
- The same thing is true for classes; A class defines a type and subclasses can be defined that excludes some of the values from the superclass

Class Inheritance

- Classes can exhibit **inheritance relationships**
 - Behaviors and data associated with a superclass are **passed down** to instances of a subclass
 - The subclass can **add new behaviors and new data** that are **specific to it**; it can also **alter behaviors that are inherited from the superclass** to take into account its own specific situation – making it a **derived class**
- It is extremely desirable that **any property that is true of a superclass is true of a subclass; the reverse is not true**: it is okay for properties that are true of a subclass not to be true of values in the superclass
 - For instance, the property `isPositive()` is true for all natural numbers but is certainly not true of all integers

Superclass/Subclass Inheritance Relationships

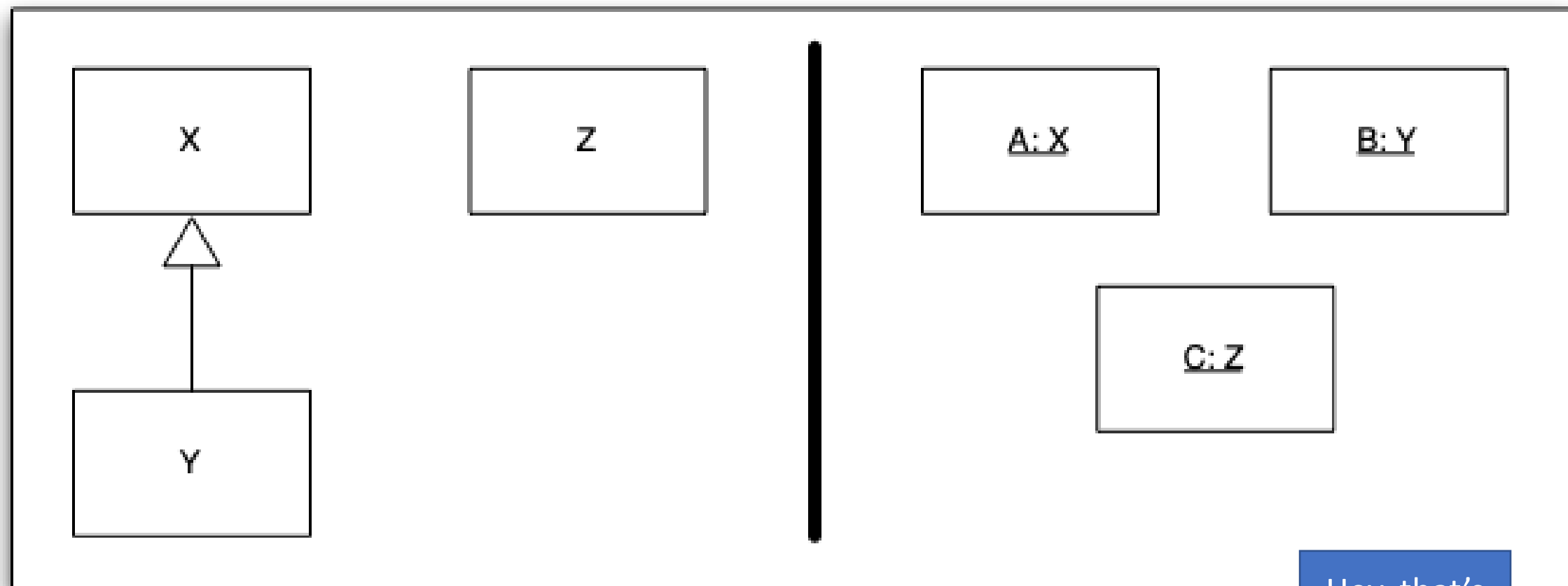
- Inheritance relationships are known as **is-a** relationships
 - Undergraduate **IS-A** Student (i.e. Undergraduate is-a subclass of Student)
- This phrase is meant to reinforce the concept that the subclass represents a more refined, more specific version of the superclass
- **If need be, we can treat the subclass as if it IS the superclass.**
 - It has all the same attributes and all the same methods as the superclass
- **so code that was built to process the superclass can equally apply to the subclass**

Class Encapsulation

- Classes can control the **accessibility of the features** of their objects
 - That is they can typically specify whether an attribute or method has an accessibility of public, protected, or private.
- This ability to **hide features of a class/module** is referred to as **encapsulation** or **information hiding**;
 - however, **encapsulation is a topic that is broader than just data hiding**, as we will discuss later in the semester

Accessibility, continued

- Object A is an instance of class X
- Object B is an instance of class Y which is a subclass of X;
- Object C is an instance of class Z which is unrelated to X and Y



Hey, that's
a UML
Model!

Accessibility, continued

- **Public** visibility of a feature of class X means that A, B and C can access that feature
- **Protected** visibility of a feature of class X means that A and B can access the feature but C cannot.
- **Private** visibility of a feature of class X means that only A can access the feature
- Note: these are the general definitions; different programming languages implement these accessibility modifiers in different ways
 - Consult the documentation for details specific to your favorite OO programming language

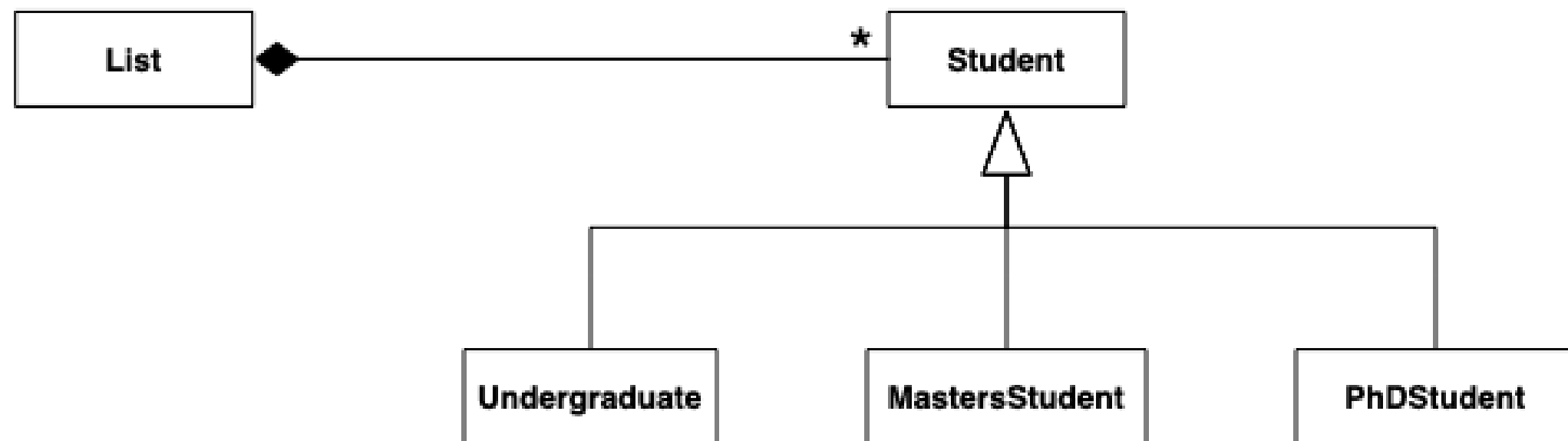
Class Constructors/Destructors

- Classes can control how their objects are created and destroyed
 - OO Programming languages will (typically) provide “special methods” known as **constructors** and **destructors** (a.k.a. **finalizers**) to handle these two phases in an object’s life cycle
 - Constructors are useful for ensuring that an object is properly initialized before any other object makes use of it
 - Destructors are useful for ensuring that an object has released all of the resources it consumed while it was active
 - Destructors can be tricky; in languages with garbage collection, an inactive object might hang around for a significant amount of time before the garbage collector gets around to reclaiming its space

One benefit of superclasses

- Treat **all** instances of a **superclass-subclass hierarchy** as if **they were all instances of the superclass** even if some are instances of subclasses
- Example
 - Suppose we have the classes, Undergraduate, MastersStudent and PhDStudent in a software system
 - Problem: We may have a need for acting on all instances of these three classes at once, for instance, storing them all in a collection, sorting by last name and displaying a roster of the entire university
 - Solution: Make all three of these classes a subclass of the class Student; You can then add all of the students to a single collection and treat them all the same

Example rendered in UML



Note: UML Notation will be discussed in further lectures

Another benefit of superclasses

- Not only can you group all instances of an object hierarchy into a single collection, but you can apply the same operations to all of them as well
 - In our example, any method defined in the superclass, Student, can be applied to all instances contained in our collection (the List of Students)
 - On the following slide:
 - Student has a method called saySomething() which is overridden by each subclass to say something different
 - Yet look how clean the code is...

```
1 import java.util.LinkedList;
2 import java.util.List;
3
4 public class Test {
5
6     public static void main(String[] args) {
7
8         List<Student> students = new LinkedList<Student>();
9
10        students.add(new Undergraduate("Bilbo Baggins"));
11        students.add(new MastersStudent("Aargorn"));
12        students.add(new PhDStudent("Gandalf the White"));
13
14        for (Student s: students) {
15            System.out.println(" " + s);
16        }
17
18        System.out.println();
19
20        for (Student s: students) {
21            s.saySomething();
22        }
23
24    }
25
26 }
27
```

The True Power: Clean Code!

- The most powerful code in the previous example was

```
for (Student s: students) {  
    s.saySomething();  
}
```

- Why?
 - You can add as many subclasses to the Student hierarchy as you want and this code **never has to change!**
 - **It doesn't even have to be recompiled (demo)**
 - Indeed, given the right techniques, a server running this code doesn't even need to be "brought down"; the new subclass can be dynamically loaded and this code will recognize instances of that subclass and do the right thing

Polymorphism

- The previous example demonstrated polymorphism
 - which literally means “**many forms**”
 - in OO A&D it means that we can treat **objects as if they were instances of an abstract class but get the behavior that is required for their specific subclass**
 - The “many forms” refers to the many different behaviors we get as we operate on a collection of objects that are instances of subclasses of a generic, abstract class
- In Shalloway/Trott, polymorphism is defined specifically as
 - “**Being able to refer to different derivations of a class in the same way, but getting the behavior appropriate to the derived class being referred to**”
- Awkward definition... maybe:
 - “**Ability of related objects to implement methods specialized to their type**”
- We will see many examples of polymorphism as we move forward in the semester and you will get a chance to try it out for yourself

Abstract Classes

- The classes that sit at the top of an object hierarchy are typically **abstract classes** while the classes that sit near the bottom of the hierarchy are called **concrete classes**
- Abstract classes
 - **define a set of generic behaviors** for a related set of subclasses;
 - act as **placeholders** for other classes defining method signatures that they must implement, defining method bodies for behaviors that should be the same across all subclasses, defining data that will be useful for all subclasses
 - In OO programming languages, abstract classes **cannot be instantiated**
 - instead you instantiate concrete classes but access them via the interface defined by the abstract class

Summary

- In this lecture, we have touched on a variety of OO concepts
 - Functional Decomposition vs. the OO Paradigm
 - Requirements and Change in Software Development
 - Design perspectives: Conceptual, Specification, Implementation
- Be comfortable with the OO concepts and terminology
 - Coupling and Cohesion
 - Classes and Objects
 - Constructors, Destructors
 - Abstract, Derived, and Concrete Classes, Sub- and Super-class
 - Instance, Instantiation
 - Member, Attributes, Methods
 - Encapsulation
 - Inheritance
 - Polymorphism

Homework, Teams, and Group Work

- **Group Work:** Start forming teams to work on homework/project assignments.
- When working in a teams, it is “ok” to assign parts of the assignments to a different member of the group to work on individually, but you should make sure that you come together as a group at least once to discuss all of the answers to ensure a high level of quality.
- Also, ensure that you understand the answer to each question because similar questions may arise on the midterm or final.
- All students should be on a team of two or three.

Project Late Policy – General

- Projects will have a set due date and time
- The general late policy for this semester
 - Submit in first 4 hours after due date/time – no penalty
 - Submit in first 44 hours after that (day 1, day 2) – 5% penalty
 - Submit in 48 hours after that (day 3, day 4) – 15% penalty
 - Cannot submit after that
- Late policy will be included in each assignment with any modifications
- I will generally avoid extensions – if I feel we need one I'll announce it

Next Steps

- Make sure you sign up for Piazza and Canvas notifications
- Get access to the Head First Patterns book
- Consider the tutorials/sources for Git, Java, Python if you need them
- Organize your 2-3 person team for projects/homework
- One more OO discussion for the fundamentals
- Coming soon: Git, Markdown, Github; Graduate Project info; Java & Python
- Then we'll move on to UML, TDD, and the start of the OO patterns reviews
- First homework assigned on Friday, schedule for classroom use also