

# Object Oriented Analysis and Design Basics

A review of many things you know and some things you don't know, but should!

# Background Questions

- Java is an Object Oriented (OO) Programming Language
  - What makes it OO?
  - What features are not OO?
- When you create a program from scratch, do you:
  - Use OO techniques?
  - Design Patterns?
  - OO Design Heuristics?
  - If your answer is no to any of the above three items, you will be able to say yes by quarter's end, young padawan

# Design Basics

- What is design?
- What comes before design?
- What comes after design?
  - Does this question make sense in software development?
- What would make the process of software design object-oriented?

# Design Methods (Chapter 1 of DPE)

- Ways of solving problems
  - Structured Design/Programming (a.k.a. functional decomposition)
    - “Think in terms of steps”
  - Functional Programming (a.k.a. declarative programming)
    - “Think in terms of functions and their composition”
  - Object-Oriented Design/Programming
    - “Think in terms of objects that do things”

# Simple Problem: Display Shape

- 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 the problems up 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

# With respect to change...

- A process-based approach to solving problems does not lead to program structures that can gracefully react to change
  - And change in software development often involves a variation on an existing theme
    - display new types of shapes
    - change the way shapes are rendered
    - add new functionality to the program such as being able to move the shapes after they have been displayed
  - In “main programs,” these types of changes typically cause complexity to increase and require that lots of files have to be recompiled

# Why do these problems exist?

- These problems occur with the functional decomposition approach because the resulting problems exhibit
  - 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 shields dependencies from forming throughout the code

# Why should we care?

- As the book says
  - “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!

# Analysis

- 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

- Requirements for a software system are initially generated during the analysis phase of software development and are, typically:
  - simple statements of desired functional capabilities
    - “the system should allow its users to sort records by priority”
  - statements of non-functional capabilities
    - “the system should support 10,000 simultaneous users”
  - statements of constraints that must be met
    - “the system will comply with regulation XYZ at all times”

# The Problem of Requirements (I)

- The problem? Experienced developers will tell you that
  - Requirements are incomplete and do not tell the whole story
  - Requirements are typically 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 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)

- Most 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 write our code such that change can be managed
- Agile methods make use of “user stories”; other life cycle methods make use of requirements documents or use cases (dressed-up 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 them while leaving room for them to change

# The Problem with Functional Decomposition

- The book highlights a problem with code developed with functional decomposition
  - such code has 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

- Cohesion refers to “how closely the operations in a routine are related” - from Code Complete by Steve McConnell
  - 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

- 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
- We want systems with parts that are highly cohesive and loosely coupled

# Ripple Effects

- Ripple effects cause us to spend a long time doing debugging and system understanding tasks
  - We make a change and unexpectedly something breaks
    - This is called an unwanted side effect
  - If we have tightly coupled code we discover that many parts of the system depended on the code that changed
    - It takes time to discover and understand those relationships
- Once understanding is achieved, it often takes very little time to actually fix the bug

# 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

- The book asks
  - 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 their 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,
  - 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,
  - 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

# Benefits of the second scenario

- The biggest benefit is that entities of the system have their own responsibilities
  - indeed this approach represents a shift of responsibility away from a central control program to the entities themselves
- Suppose we had attendees and student volunteers in our session and that volunteers needed to do something special in between sessions
  - First approach: the session leader needs to know about the special case and remember to tell volunteers to do it before going to the next session
  - Second approach: the session leader tells each person “Go to your next session”; volunteers will then automatically handle the special case without the session leader needing to know anything about it
    - We can add new types of attendees without impacting the leader

# Revisiting the Shape system

- Recall our “display shapes” program from earlier in the lecture
  - How would you rearrange it to follow this new approach?

# Foreshadowing

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

# Perspectives in Software Development

(UML Distilled, Martin, Fowler, Addison-Wesley, 1999)

- Conceptual — What are the main concepts of the problem domain and what are each of those concepts responsible for? “A conceptual model should be drawn with little or no regard for the software that might implement it.”
- Specification — What are the interfaces of the objects (derived from the concepts above) in the system. How are these objects used?
- Implementation — How do the objects fulfill their responsibilities?
- “Go to your next class” : a conceptual instruction that shields the requestor from how this command is actually carried out (implementation). The fact that we know that objects can respond to this command comes from the specification level.

# 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
- Objects inherently know what type they are. Its attributes allows it to keep track of its state. Its methods allow it to function properly.

# 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 the conceptual perspective to the specification and implementation perspectives

# Objects

- 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
    - as you will see, there are great benefits from starting with the other levels first

# Classes and Inheritance

- We are familiar with these topics through our use of Java
- Inheritance can be very powerful, but we'll find that we should favor composition over inheritance – this is a fundamental OO design principle

# Polymorphism

- 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

# Polymorphism (II)

- In the book, 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”
- As you can see, it is not an easy thing to define! But, it is very powerful and the “clean code” example should show why we as designers should strive to design OO hierarchies that allow us to write polymorphic code
  - There are other variations on polymorphism to learn, we will get to those in future lectures

# 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