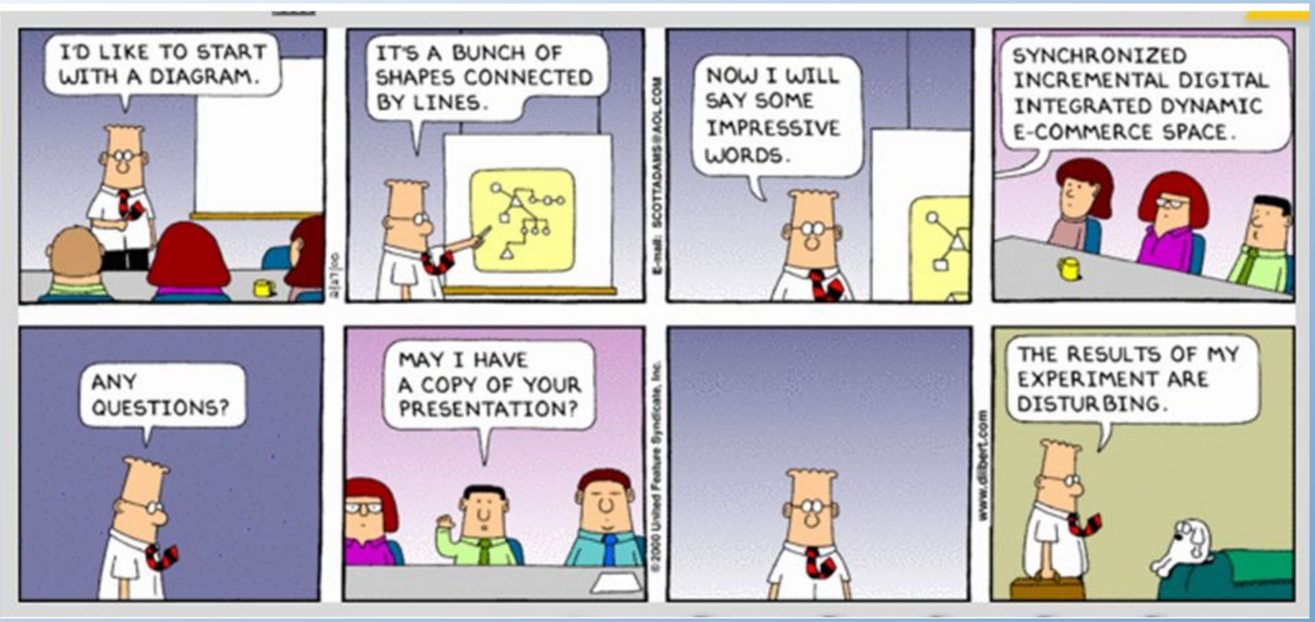
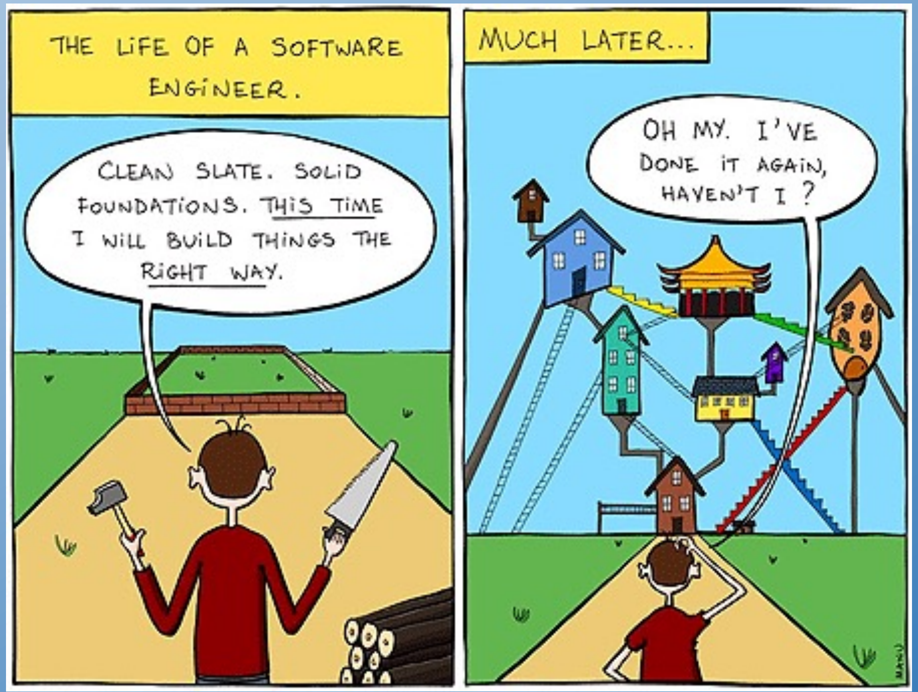




Detailed Design

CSCI2340: Software Engineering of Large Systems

Steven P. Reiss

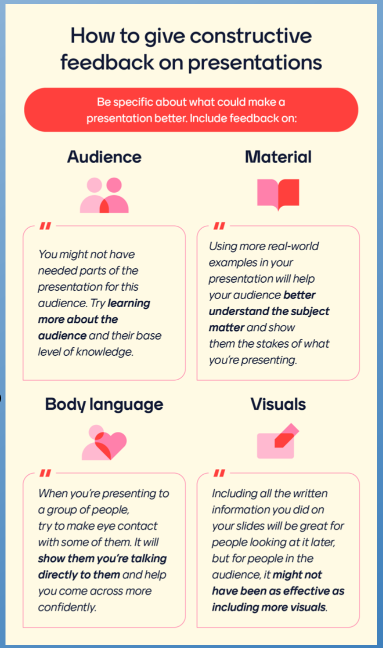

"Some of these ideas ... they sound like they ... have been pulled out of a hat."

Comments on Presentations

- Projects seem to have well organized project teams
- Projects seem to have a plan for proceeding toward the implementation
- Projects provided a nice overview of what they intend to do
  - Enjoyable presentations
  - Real specifications and requirements
- Several projects seem overly ambitious for a semester course
  - This is good – planning for the future
  - However, you need to be realistic about the course
    - Have a good definition of what you expect to get done this semester
  - You want to use LLMs
    - Don't really understand what the inputs/outputs are going to be
    - Prompt engineering and output interpretation
  - Integrating with VS Code or other IDE might be future plans
- Need to really understand what you are building
  - UI generation – react is for interactive pages; how to specify and define interaction
  - Accessibility -- both while creating (intro to talk) and after the fact (based on a URL)
  - Speech -- what edits are allowed and how are these supported

"OK, now that we all agree, let's all go back to our desks and discuss why this won't work."

Comments on Presentations



- General Problem: deciding on technologies to use before understanding the des
  - This commits the design rather than vice versa
  - Choice of language
    - Untyped (Python; JavaScript vs TypeScript)
  - Choice of database
    - This is an implementation decision, not a design decision (e.g., songs might be better stored in NoSQL)
    - Possibly specify what needs to be stored, not the schema
      - Definition of database schema is implementation – that is not part of the interface between components
      - Shouldn't need to be known by all members of the team
  - Choice of framework
    - This is generally an implementation decision, not a design decision
    - Front-end shouldn't care what is responding to their RESTful requests
- Need actual users to provide feedback on specifications, UI, …
  - Some projects did this nicely
  - Others planned to – these are needed before as well as after design

Comments on Presentations

- Need to identify and account for risks
  - No project listed the risks, but they are there (e.g., LLM usage)
- Need to define the interfaces
  - No one talked about the component interfaces, the set of RESTful messages, etc.
  - This is what should be defined in terms of high-level design, not implementation details
- Document as you write, not afterwards
- Need to think about how app could be self-supporting
  - How could you monetize it to support AWS, maintenance

- Remember that the goal of the project in the course
  - Is to learn how to write large, long-lived software
  - Not just to get something working this semester

- Project Meeting at the end of class
  - I'll be available for questions

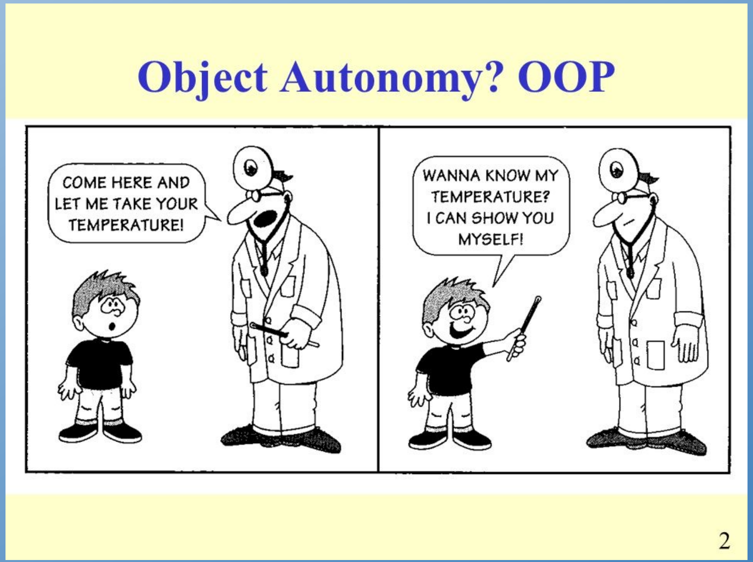
User Interface Homework

- What approach did you take to graphic note bubbles
  - Do you think your solution will work
  - How to minimize the interface
  - What ideas did you have

High-Level Design

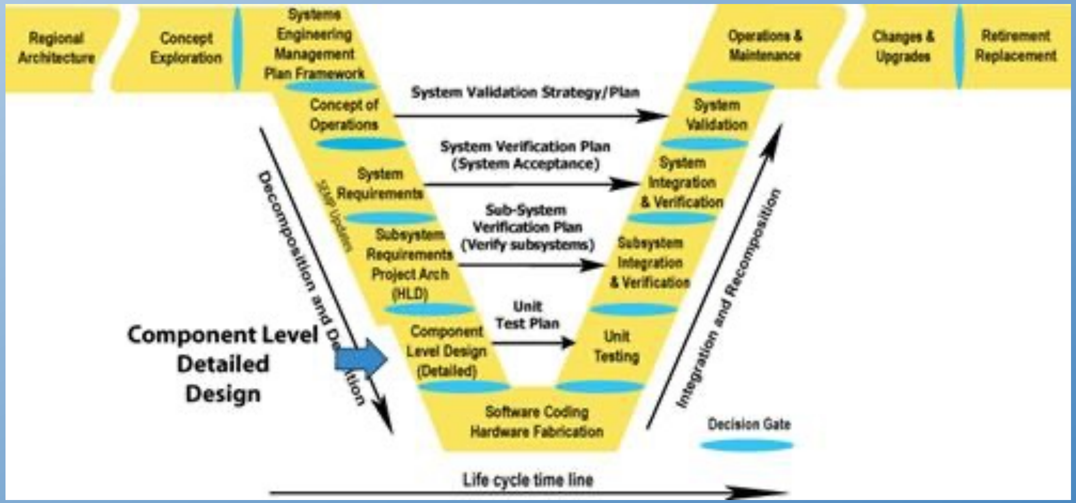
- Breaks the system into components
- Each component to be coded by individual (or a small team)
- Each component has well-defined interfaces
  - Note that these may change
- Next step is to design the components
  - **And** choose appropriate technologies
  - **Then** build the implementation
- You should have some experience at this level
  - From CS32, CS134 or equivalent
  - I want to highlight what I think you should have learned

Object-Oriented Design



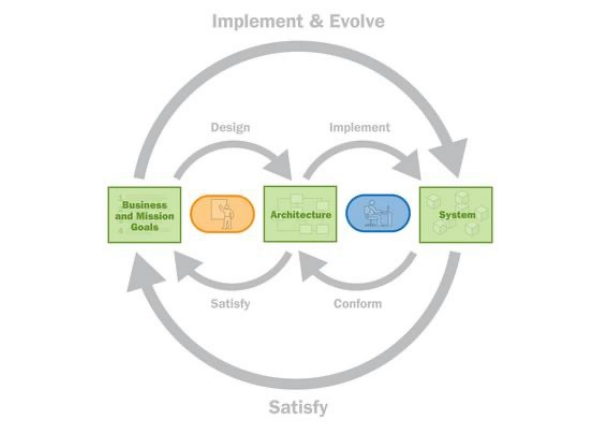
- Object-oriented design fits most applications
  - Objects provide information hiding
  - Objects are a natural way of representing things
  - Objects can be singletons or sets of items
- Objects are supported by most languages
  - Modules in some languages represent singleton objects
  - But mixing modules and objects can get confusing
- Objects are flexible
  - Naturally represent many aspects of programming
  - Not just physical entities

Detailed Design



- For a single high-level design component
  - Repeating high level design at a more detailed level
- Designing classes (modules, files)
  - Determine the set of top-level classes needed
  - What are the methods/functions and fields/local variables of those classes
  - Determine how these classes are organized (inheritance)
- Designing methods
  - What do the methods do
  - High-level specification (not code)
    - Signature (data types, return value, exceptions)
    - Statement as to what the method does (JavaDoc? Pseudo code?)
  - Don't be afraid of exceptions
- Designing private methods and fields comes later
  - Part of implementation, not design

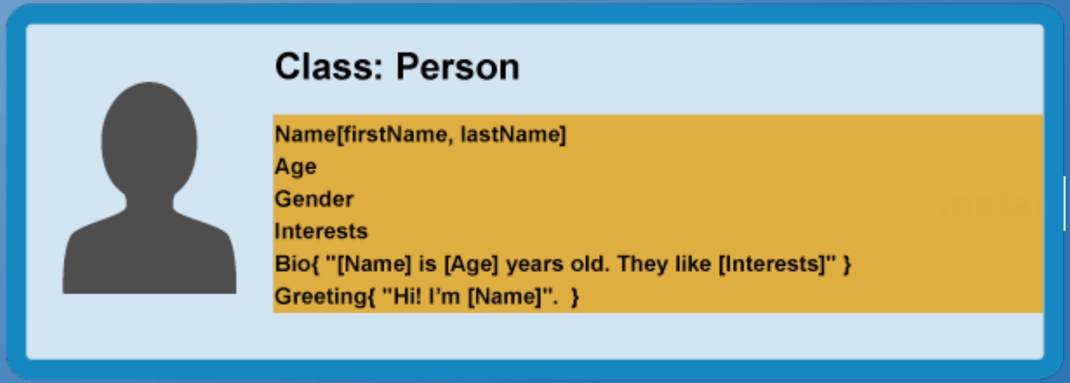
Goals of Detailed Design



- Compact, coherent implementation
  - Before you commit to code
- Changes, new features, etc. are contained in a single class
- Top-level classes aren't too big or too small
  - Inner or support classes should be small
  - File sizes are reasonable
- Number of files & classes is reasonable
- Methods have reasonable (small) number of parameters
- The interfaces between classes are **SIMPLE**
  - Simpler means easy to code, maintain, evolve
- You should feel comfortable coding it from specification

What do Classes Represent

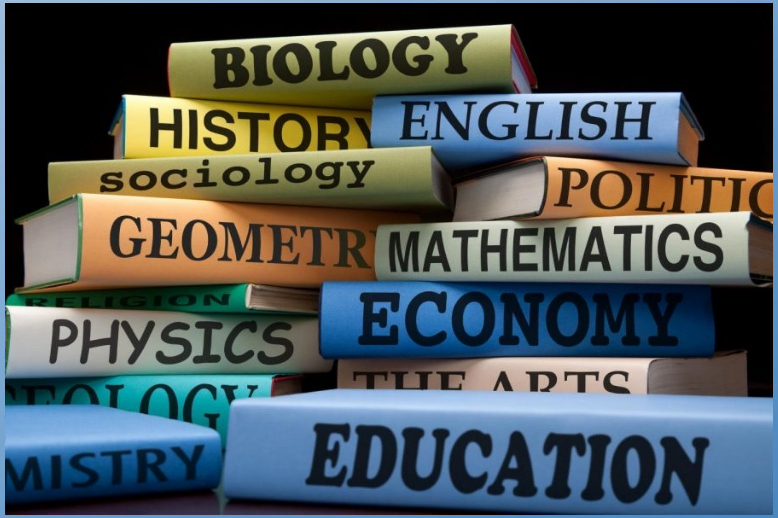
- Objects (physical or virtual) in the solution
  - Data with operations
  - Anthropomorphic
  - Example: switches, sensors,
- Algorithms (functional classes)
  - Example: safety controller (switches, signals)
- Reactive Elements (callback classes)
  - Function pointers, completions
  - Sets of these
- Control (Thread/Runnable)
- Controllers
  - Phases of a run
- Data types (e.g., lists and maps)
  - Complex combinations of these



Choosing a Set of Classes

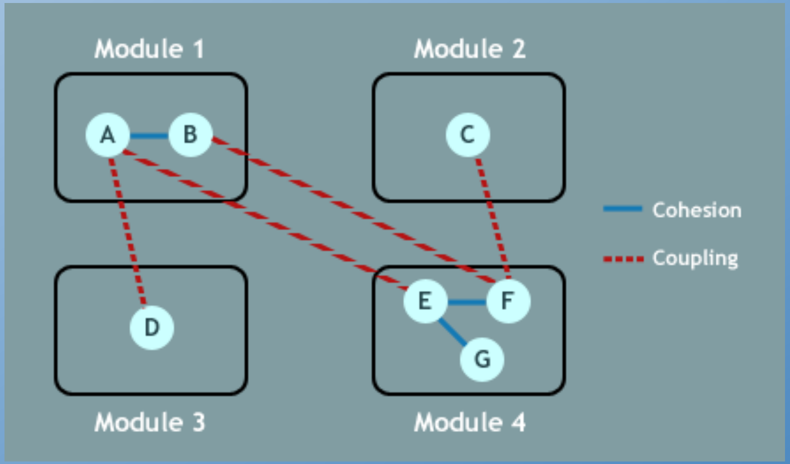
- You should have done this in the top-level design
  - Set of interfaces or façades for the design
  - Here the components are packages/modules
    - If very complex, use multiple or nested
    - Additional packages developed for supporting or common code
- You need to do it again for each package or module
  - Start with classes representing the top-level design components
    - Façades – probably on one class for this package
    - Interfaces – a public class for each interface this package implements
  - Then add whatever is needed to support these

Choosing Classes



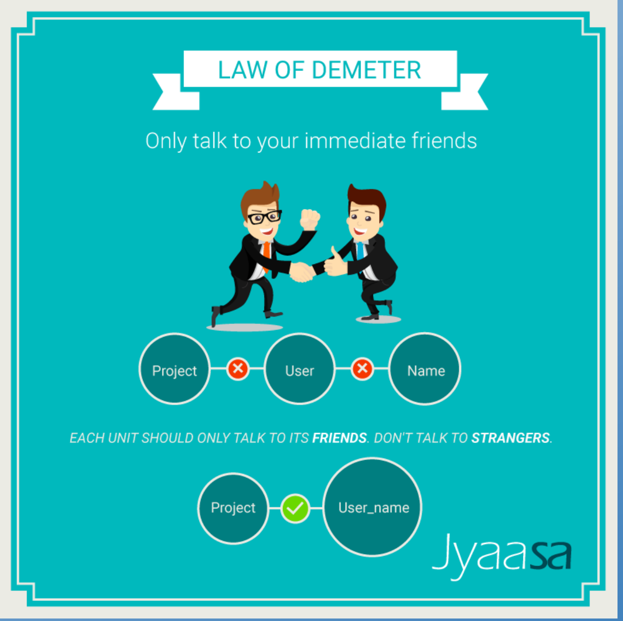
- Goal: set of coherent classes
- Start with the set of all possible classes
- Organize this
  - Cluster classes that are similar (e.g., hierarchies)
  - Find representatives of clusters (or create)
  - Find dominant classes (this controls or owns that)
  - Find redundant classes
  - Find common functionality
  - Can use UML again, interfaces, paper, …
- Choose a subset of these
  - That cover the original set
  - That is “good” : what does this mean?

Coupling and Cohesion



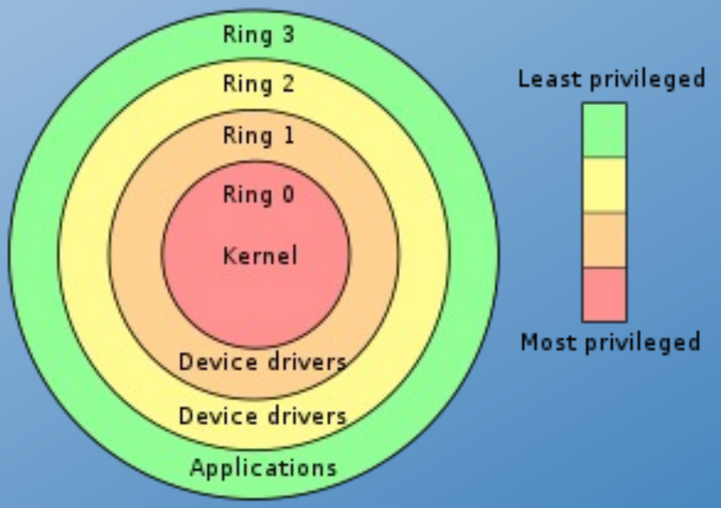
- Coupling
  - How much one class needs to understand or use another
  - Generally, communication should be 1-way, not 2-way
  - Avoid implementation dependencies
  - Want to minimize coupling
- Cohesion
  - How unitarian (sole-purpose) one class is
  - Should be able to describe a class with a simple phrase
  - Want to maximize cohesion

Law of Demeter



- Principle of least knowledge
- A unit should have limited knowledge of others
  - Only units "closely" related to the current unit
  - These are the unit's friends
  - Should only have a small circle of friends
- Each unit should only talk to its friends, not to strangers
  - Only talk to immediate friends
  - Very limited communication to things outside the package/module
  - Limited communication to other classes even in the package
- Principle of least privilege
  - Restrict and annotate who can access what

Principle of Least Privilege



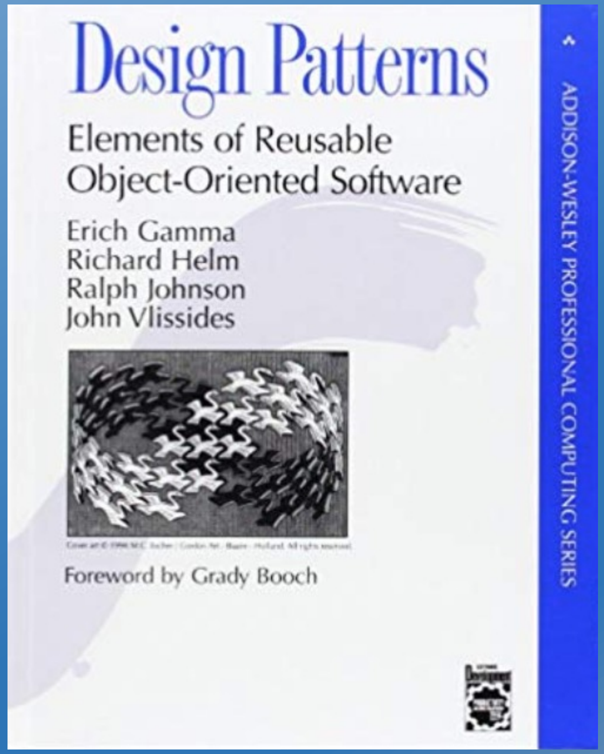
- Fields should always be private
  - Except for constants defined in an interface
  - Possibly protected for use in subclasses, but this is discouraged
    - You need to look at the superclass when fixing the subclass
  - Implementations should depend on another class's fields – they are low-level details
- Methods should only be public where necessary
  - Implementing an interface, part of a façade – defined in high level design
- Methods should only be package-protected where necessary
  - When needed by other classes of the package
  - But the package-protected set of methods for a class should be small (its local interface)
- Methods should only be protected where necessary
  - When needed by subclasses, no-one else (don't use as package-protected too)
- Methods should be private by default
- Inner classes should be private
  - And static where possible
- Pure constants (strings, numbers) should be avoided in code
  - These tend to change, and you need to find them and keep them consistent
  - Strings might change with internationalization

Design Patterns

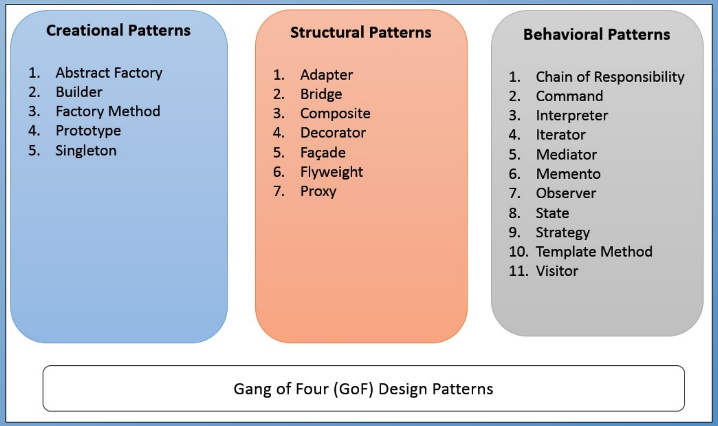
- Early on we noted design is the application of patterns
- We talked about architectural patterns
  - And noted that patterns exists at all levels
  - It is your job as a software engineer to know lots of patterns
  - That is what makes a good designer
- What is an object-oriented design pattern
  - Set of classes and methods to serve a particular purpose
- Description of a design pattern
  - Purpose
  - When it should be used
  - When it should **not** be used
  - The actual classes and methods
  - Alternative implementations

Design Patterns

- Are useful
  - Handle common situations in a standard way
  - Provide a common vocabulary for understanding design
  - Provide a starting point for doing design
- Can be overused
  - Or underused
- Gang of four (GoF) book
  - 20 some common patterns (sequential)
  - You should know these by name
    - Some are more common than others
    - Your design vocabulary



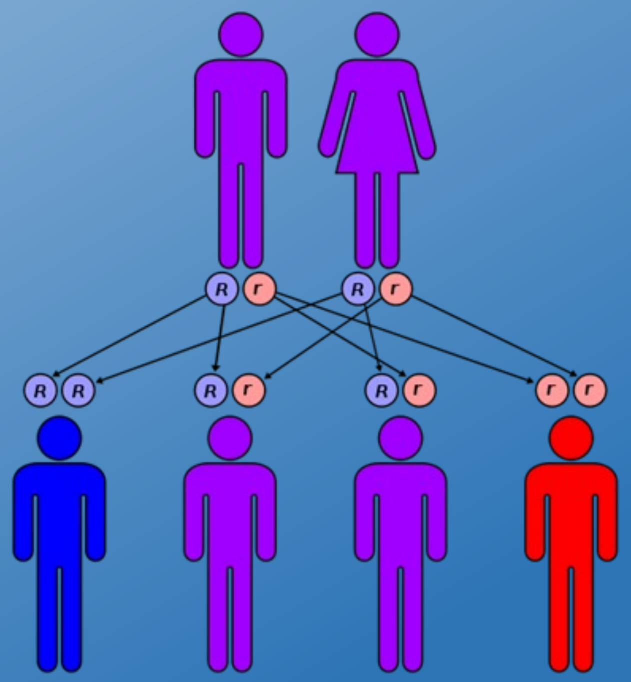
Categories of Design Patterns



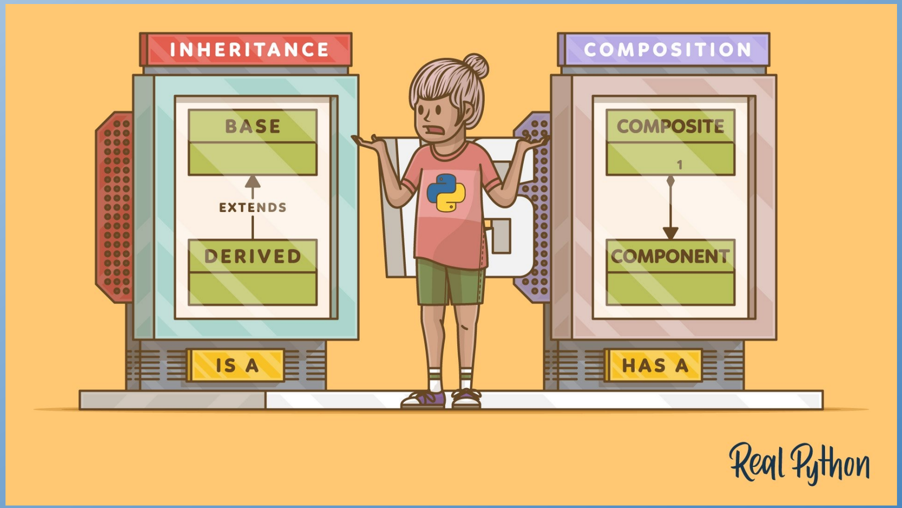
- Factory patterns
  - Builder, Abstract Factory, Flyweight, Singleton, Factory Method, Prototype
- Delegating responsibility patterns
  - Adaptor, Bridge, Decorator, Façade, Proxy
- Control patterns
  - Composite, Interpreter, Command, Iterator, Strategy, Template, Visitor
- Algorithmic patterns
  - Mediator, Memento, Observer

Inheritance

- What object-orientation is all about
  - Not really, sort of a side issue
    - OO is about abstraction and information hiding
  - Inheritance offers OOP lots of functionality
- Forms of inheritance
  - Interface inheritance
  - Class inheritance
  - Multiple inheritance
  - Prototype inheritance (Self, early JavaScript)
  - Mix-ins
- Uses of inheritance
  - The different functionalities they provide



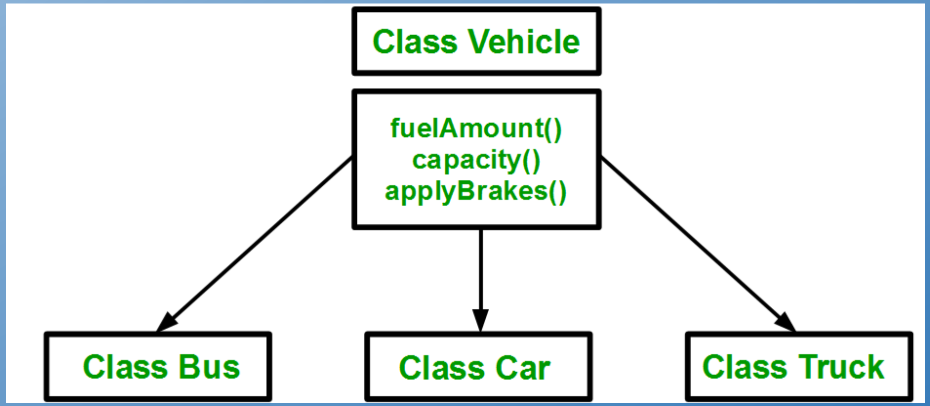
Natural Inheritance

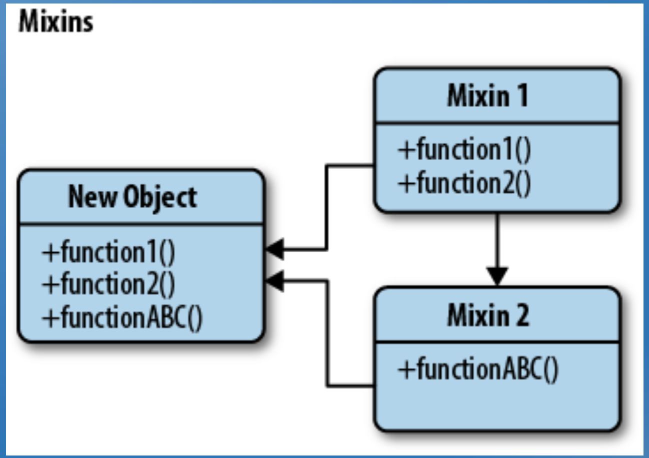


- Representing a hierarchy of object types
  - Obvious example (animal/mammal/rodent/…)
  - Not that frequently used
- Most natural hierarchies are shallow
  - AST nodes, symbol types, device types
- Intermediate classes should be abstract
  - Used to group lower-level classes, not used as objects
  - Used to define or specify particular functionalities
- All external references should be to the root class
  - Or abstract classes (internally)
  - Don’t not want to expose the implementation or hierarchy
    - Because it is going to change and is an implementation detail
    - Others shouldn’t be dependent on it
  - Generally, don’t want to expose intermediate classes
  - Generally, don’t want to expose leaf classes either

Inheritance for Shared Functionality

- Providing shared functionality
  - Common methods go into super class
    - Which should be abstract
    - Without a public constructor
    - Only used to represent any of its implementors
  - Subclasses are directly visible
    - Can be created, etc.
  - Example: CATRE saved objects, BudaBubble
- Provide Additional functionality
  - Mix-Ins are designed for this
    - Multiple inheritance can provide this
    - Java: interfaces with default methods
  - How these work can be confusing and messy
    - Conflicting names, DAG-like inheritance



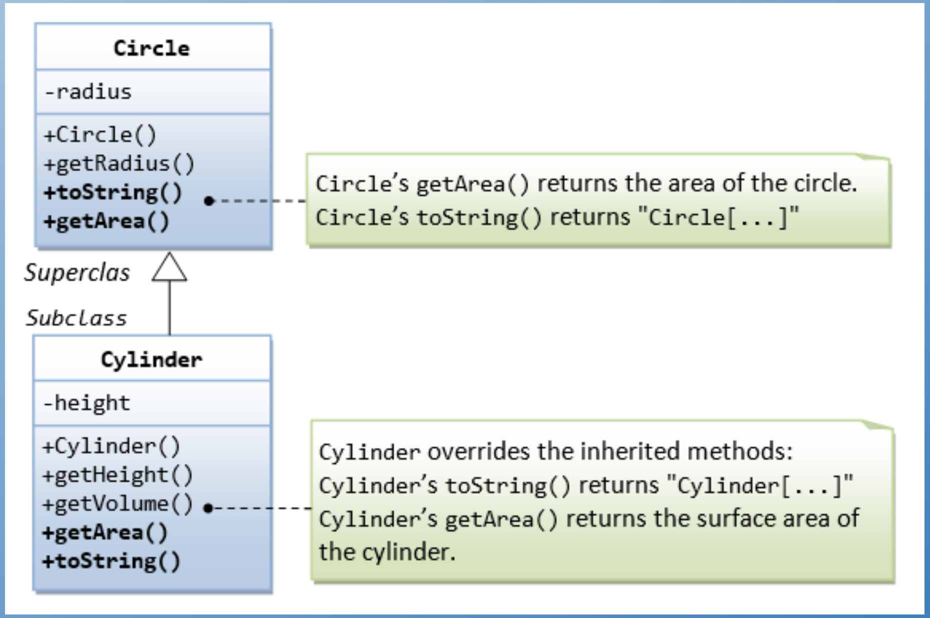


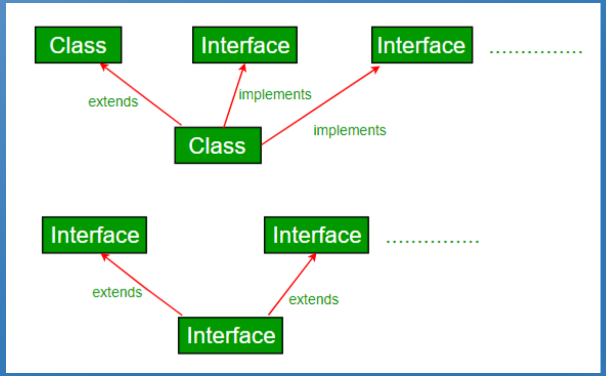
Other Uses of Inheritance

- Providing common definitions
  - Constants interface
    - Fields defining global constants
    - Enumerations
    - Can also define inner interfaces and classes
  - Implementing this provides access to names, not functionality
- Providing annotations
  - Indicate a class has certain properties
  - Serializable, Cloneable
  - Code Bubbles: Zoomable

Other Uses of Inheritance

- Modifying the behavior of a class
  - Inheriting from a Swing class
    - Set properties in the constructor
    - Modifying the paint method
- Providing alternative implementations
  - Interface or abstract class as the root
  - Subclasses implement the actual algorithm
  - Jcomp (compiler)
    - Internal implementation to read byte codes using ASM
    - Jcode implementation that tracks all the information in a project
- Interface inheritance
  - Supporting high-level design interfaces
  - Defining abstract functionality
  - Requires factory classes or static methods to create the actual objects
  - This is the only safe instance of multiple inheritance
    - Assuming there are no default methods
  - ActionHandler, Runable, CatreSavable



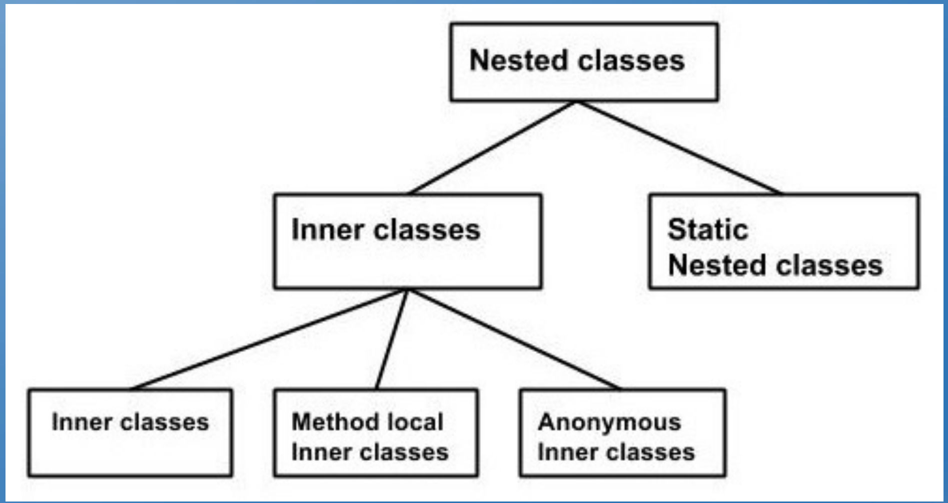


Other Uses of Inheritance

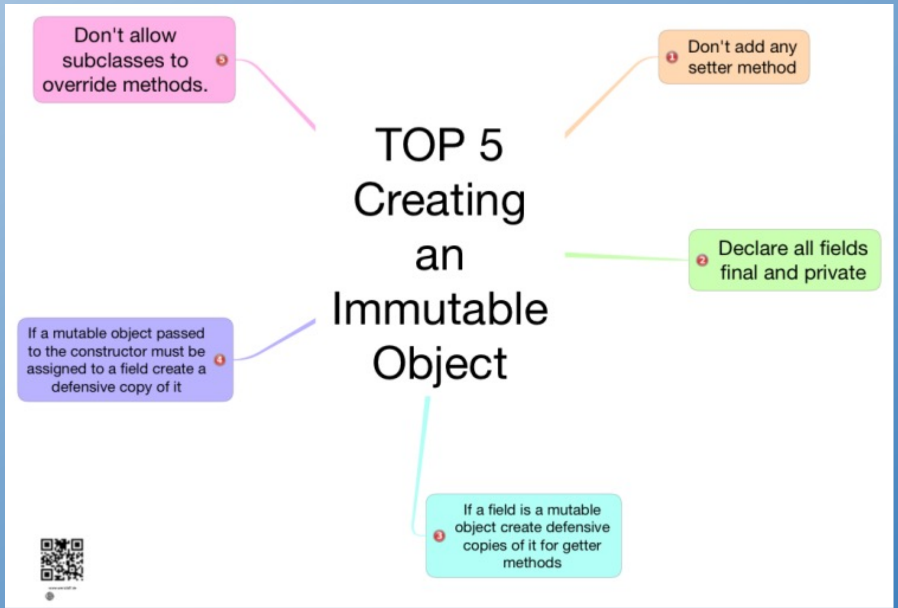
- Defining callbacks
  - Server defines a callback interface
    - Use default methods to allow simple use (if > 1)
    - Or provide an implementation class with empty callbacks that can be inherited from
  - Client defines an implementation of the callback
    - Registers it with the server
  - Server invokes the callback when an event occurs
  - Akin to callback functions, but more general
  - MouseListener, ActionListener, …
- Behavior inheritance
  - Defining the default behavior (Thread, AbstractAction)
  - Adding behaviors to a class or interface (JcompExtendedFile)
  - Identifying the availability of a behavior (Scalable in Bubbles)

Inner (Nested) Classes

- Any class used only by a single class should be a private inner class if possible
  - When allowed by the language (Java)
  - Or a private class defined inside a module (JavaScript, Dart)
  - Can also use private class defined in the same file (C++)
  - Unless it is too complex (> k lines, m methods, one page) – then it becomes its own outer class
- A local hierarchy can be implemented as inner classes
  - Outer class is the root of the hierarchy
  - Internal classes are private Inner classes and not exposed directly
  - Inner classes are static (why?)
  - Example: JcompSymbol => various types of symbols
- Inner classes should always be private (& static if possible)
  - Inner classes are implementation details
  - Never refer to inner classes of another class
    - Exception: Inner interfaces of a top-level interface are okay
  - Don't export inner classes
- Inner classes will often evolve to outer classes
  - When they get to large or complex



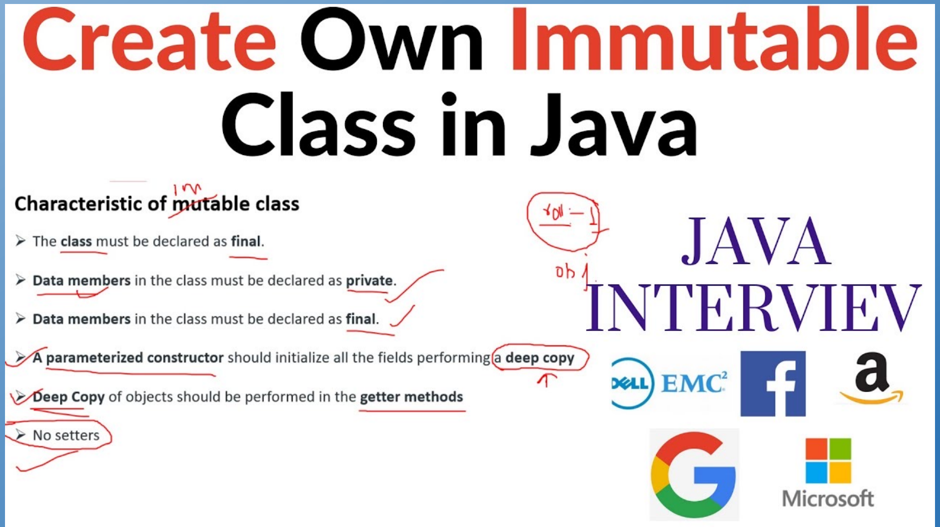
Immutable Objects



- Objects that are never changed once created
  - String in Java; Java Records
- Immutable objects might change internally
  - But the changes are not visible to anyone
  - And the changes are thread safe (or duplicable)
  - String in Java computes and stores its hash code the first time it is needed
- These are much simpler to reason about
  - And generally safe for concurrency
- Can be tricky to code
  - Should be final, everything done in the constructor, no internal fields for computations, …
- Can be more difficult to code with
  - Less efficient (lots of new objects)
  - Need to remember to use correctly (String.replace(…))
  - Must be careful to keep it immutable as the implementation evolves
    - Document immutability and concurrency

Unique Immutable Objects

- If you create many instances of an object
  - And they are essentially the same
  - Create only one instance of that object
- Have a factory method that checks if object exists
  - That calls a private constructor if it doesn't
  - Returns original if so
  - Returns a new instance if not
  - String.intern()
- Generally, implies object is considered immutable
- This has advantages
  - Saves memory
  - Can use == rather than .equals (fast comparison, hashing, …)
- Disadvantages
  - Need to define fast lookup
  - Factory method must be synchronized in a multithreaded environment
  - Can be tricky to code and use: must be immutable



PROJECT Status

- Make sure you know what you are building
- Should have ideas for your user interface (10/15)
- Should have broken project into components
  - Separate the various components
  - High-level design: façades and interfaces
  - Components assigned to individuals (or small teams)
- Should have interfaces for the components (part due)
- Ensure components are separated for implementation
  - For example, in Java, use different packages
  - In other languages, use different directories
- Individuals should start developing a set of top-level classes for each package
  - Or modules or components or files (depending on language)
- We will have project status reports in a few weeks

Further Reading

- https://w3sdesign.com/GoF_Design_Patterns_Reference0100.pdf

Project Meeting Time