CMPT 213: Object Oriented Design in Java

A Course Overview

Jeffrey Leung
Simon Fraser University

Spring 2020

Contents

1 Introduction

- Standards:
  - Make fields private when possible
- Commenting:
  - Comment purpose of a class
  - Name fields/methods/parameters so comments are unnecessary
- When possible, convert strings to non-string types internally for consistency
- **Clean code:** Code which is correct, easy to read/maintain, and conforms to a standard
- Software design:
  - 4 steps:
    - Requirements
    - Design and implementation
    - Verification
    - Evolution
  - Designing involves identifying classes, responsibilities, and relationships to create a diagram
  - Implementation process options:
    - **Skeleton code:** Beginning minimal parts/features of a system
    - **Component-wise:** Creating components one at a time
  - Methods of integrating code from multiple people:
    - **Continual integration:** Gradual system growth by constantly integrating changes
    - **Big Bang integration:** Building all parts separately without integrating until the end
- **Feature envy:** Characteristic of a class which relies heavily on another class
- Warning sign: Characteristic of a method which operates more strongly on another object than its own
- **Deprecation:** State where a public interface is no longer supported or recommended, and is slated to be removed in the future
- **try-catch:** Structure which watches for an exception and handles it
  - Only one exception can be live at a given time
  - **finally clause:** Optional clause after catch clauses which is executed regardless of the result
    - If exception is thrown, the finally clause is executed immediately afterwards
  - **try-with-resources:** Block which cleans up a resource when a try block exits
- Exception: Issue which may be fixable and is not out of the software's control
  - **Checked exception:** Exception which must be caught or listed in a throws clause
  - **Unchecked exception:** Exception which will automatically propagate and does not require catching
    - E.g. RuntimeException
    - Preferred as it does not require modification of methods between try/catch, which decouples code

2 Java

- **this:** Keyword which refers to the current surrounding object
- Almost everything is an object except for primitive types (e.g. int, float, double, boolean)
- Assignment and parameterization passes by:
  - Value for primitive types
  - Reference for objects
- Type conversion:
  - **Type promotion:** Implicit conversion from a smaller to larger type (e.g. int to double)
  - **Type demotion:** Explicit conversion from a larger to smaller type (e.g. double to int)
- String literals are:
  - Immutable
  - Of the String class
  - Compared with the equality operator using a reference
- Equality comparison compares by reference; .equals() compares by value
  - .equals() must be overridden
  - Using instanceof violates symmetry for derived classes
  - Using getClass() violates Liskov Substitution Principle (ability of derived classes to share the same behaviour)
- hashCode():
  - Used to sort objects into easily-searchable accessible buckets in a hash table
  - Must be overridden with getClass()
  - Multiples the result of each previous step's addition of fields by 31 as this value is prime, odd and optimized on some hardware
  - Shortcut for checking equality: If two objects have inequal hash codes, then they must not be equal
- Automatic garbage collection once no references to an object exist
- Collections store objects (not primitives)
- **Inner class:** Class within a file-level class
  - If static, can be accessed without referencing the outer class
  - Can receive a reference to final variables and methods in the outer class
    - Variables must be final (or not modified) because the method will be executed later, and a definitive value must be captured
- for−each loops can take unmodifiable collections to iterate through, preventing side effects
- Common interfaces:
  - Classes which are Iterable must implement the Iterator iterator () method
  - Classes which are Comparable must implement the int compareTo(Object obj) method

- **Plain Old Java Object (POJO):** Java class/object which is basic and only interacts with Java classes, without external component access
- **Spring (Boot):** Dependency injection framework for Java

3 Object-Oriented Design

- Effective because classes represent stable problem domain concepts
- Design involves identifying classes, responsibilities, and relationships
- **Object:** Unique software entity with state and behaviours
  - **State:** Information about an object
  - **Behaviour:** Methods and operations which view and/or modify state
  - **Class:** Type of a set of objects with the same behaviours and set of possible states
    - **Final class:** Class which cannot be inherited from
  - **Field/instance data:** Member variable stored by an object
  - **Method:** Member function of a class
    - **Final method:** Method which cannot be overridden
  - Statics:
    - **Static/class field:** Field for which only a single instance exists and is shared between all classes
    - **Static/class method:** Method which can be called on the class without a constructed object
  - Unique types of objects:
    - Agent: Object which performs a specific task
    - Users/roles, events, systems, interfaces
- Anonymous classes:
  - **Anonymous class:** Single-use implementation of an interface, defined only once within another body of code
  - **Anonymous object:** Instance of an unnamed class
- Relationships between objects:
  - **Dependency (uses):** Relationship between two classes where one uses the other (i.e. the first may need to change if the second changes)
    - **Coupling:** Relationship between two classes where one is a dependency of the other
      - The greater the coupling, the greater the changes required when one component is modified
      - Ideally minimized
  - **Aggregation (has a):** Relationship between two classes where one contains the other
  - **Inheritance (is a):** Relationship between two classes where one is a subclass of the other (see subsection 3.5)
- **Side effect:** Observable change to state caused by code execution
- **Idiom:** Common practice
- **Protected:** Access class which is available at package-level as well as derived classes, but not external code
  - Breaks encapsulation; exposes implementation details to derived classes
  - Difficult because no control over the extending clases

3.1 Public Interface

- **Interface:** Set of methods which an implementing class can choose to implement
- Object type cannot be dynamically changed

3.2 Principles

- **Encapsulation:** Characteristic of a module which only exposes some public interfaces, allowing some internal changes to be made without breaking outward interface
  - Modules manage their own state
  - Reduces scope of change (which encourages modification) and cognitive load
- **Immutability:** Characteristic of an entity which has no methods which change its visible state
  - A modification to an object must return a new object
  - An accessor should return a value or immutable reference
  - Avoids issues from partial setters and shared references
  - **Final:** Keyword which declares a field to have an immutable reference
- **Command-Query Separation:** Principle which states that methods should not be both mutator and accessor, to avoid unexpected side effects
  - **Command:** Mutator method
    - Should return null
  - **Query:** Accessor method
- **Responsibility heuristic:** Tendency to avoid exposing internal details only for external access

3.3 SOLID Principles

- **Single Responsibility Principle:** Each class only has one responsibility
- **Open-Closed Principle:** Classes must be open for extension but closed for modification
- **Liskov Substitution Principle:** Subtype objects must be interchangeable with base objects
- **Interface Segregation Principle:** Models should support multiple possible interfaces
- **Dependency Inversion Principle:** Code should depend on abstractions, not concrete classes

3.4 Polymorphism

- **Concrete type:** Exact instantiated type of an object
- **Interface:** Definition of a set of public methods to be implemented
  - A class implements an interface
  - Code to an interface, not a concrete type
  - Can inherit from another object
- **Polymorphism:** Characteristic of an object which can be one of multiple different concrete types
  - Possible due to composition
  - Allows late binding

- **Late binding:** Characteristic of a variable which is assigned a specific concrete type at runtime
  - Allows loose coupling

3.5 Inheritance

- **Inheritance:** Subclass created from and extending a superclass
  - Allows sharing members (properties and methods)
  - Allows polymorphism between two concrete classes
  - **Single inheritance:** A subclass may have at most 1 superclass
  - **Multiple inheritance:** A subclass may have multiple superclasses
  - **Liskov Substitution Principle (LSP):** Rule which states that class B can inherit from A if and only if, for each method in A, the corresponding method in B accepts the same parameters (or more) and executes the same operations (or more)
    - I.e. Client code using the base class must be able to use the derived class without modification
    - Is-a relationships must satisfy the Liskov Substitution Principle
  - Should be used for permanent relationships, not temporary assignments
  - Favor polymorphism over inheritance
- **Superclass/Base class:** Class from which another class inherits
  - Cannot be assigned to an instance of a subclass
  - Referred to by super
  - Cannot be final
- **Subclass/Derived class:** Class which inherits from another
  - Can access non-private members of the superclass
  - Cannot directly access private members of the superclass
  - Can be assigned to an instance of a superclass
- **Constructor chaining:** Call to a superclass constructor by a constructor in the subclass
  - Ensures base classes are initialized first
  - In Java, if no superclass constructor is called, then superclass default 0-argument constructor is automatically called
- Abstracts:
  - **Abstract method:** Un-implemented method which must be implemented by any concrete subclass
  - **Abstract class:** Class with abstract methods
    - Concrete derived classes must implement all abstract methods
  - Abstract classes vs. interfaces:
    - Both require implementation of a given set of methods
    - Abstract classes can have member variables and implemented methods; interfaces cannot
    - A class can implement multiple interfaces; a class can only have a single (abstract) superclass

3.6 Overriding

- The overriding method:
  - Must have identical signature (except for visibility and return type)
  - Visibility cannot be reduced
  - Cannot throw additional checked exceptions
  - Can return a subclass of the original return type
- Method to override:
  - Cannot be private, static, final
- **Shadow variable:** Subclass variable which overrides a variable of the superclass

4 Design Patterns

- **Lambda expression:** Short function implementation for an interface with only one method
- **Software design pattern:** Description of a common software design problem and a solution
- **Don't Repeat Yourself (DRY):**
  - **Pulling-up:** Elimination of duplicate code by placing the code in a called function
- **Strategy pattern:** Design method which modularizes an algorithm by encapsulating it into a class
  - E.g. FileFilter, Comparator
  - Uses composition and can be swapped during runtime
    - Compatible with lambda functions
  - Function accepts an interface as a parameter; the subclass passed into the function implements the algorithm
- **Template method:** Design pattern where a base class delegates primitive operations to derived classes inside a template method
  - Requires inheritance and cannot be modified after compilation
  - **Primitive operation:** Modular procedure overridden create different implementations of similar methods
  - Algorithm is in the abstract base class which calls abstract primitive operations, which are overridden by subclasses
- **Static factory method:** Static method which creates an instance of its object, configured as per the method name
- **Dependency injection (DI):** Architectural design pattern where classes are passed references to dependencies
  - Allows loose coupling by separating object instantiation/ownership from use
  - **Constructor injection:** Dependency injection through a class constructor
- **Iterator:** Interface which allows abstraction of iteration over a set of objects
  - Encapsulates implementation details
- **Observer pattern:** Design method where observer objects register for updates with a certain class
  - Upon update, the class notifies all observer objects which retrieve the updates
- **Model View Controller pattern (MVC):**
  - **Model:** Component(s) which handle data and application logic
  - **View:** Component(s) which display information to users
  - **Controller:** Component(s) which receive user interaction
  - **Clean design:** Separating logic and display
- **Facade pattern:** Design pattern which encapsulates structure of the model code into a common interface

5 Code Smells

- If a function is longer than 25 lines, separate it into multiple functions
- If there are many comments, replace them with variables or functions
- If expressions are complex/difficult to read, replace them with variables or functions

6 Interface Quality

- 4 characteristics of interface quality:
  - **Cohesion:** Characteristic of an entity which has all its interfaces based on the same abstraction
    - **Single Responsibility Principle:** A class should only have one responsibility and reason to change
    - Break up classes if violated
  - **Completion/convenience:** Characteristics of entity which has all relevant and necessary features
  - **Clarity:** Characteristic of an entity which is easy to understand
    - Intention-revealing member names
    - Meaningful and relevant abstractions
  - **Consistency:** Characteristic of an entity which is designed with consistent interactions
- Other evaluations of quality:
  - Whether constructors create fully-formed objects
  - Whether each idea has one name
  - Whether command-query separation is followed
  - Whether Iterable/Comparable are implemented when appropriate
  - Whether encapsulation is respected

7 Defensive Programming

- **Precondition:** Guarantee before a method is called
  - Enforced by client, or else the class may have unintended behaviour
- **Postcondition:** Guarantee after a method is called
  - Enforced by caller
- Implementation has a contract for the client to fulfill
  - **Design by Contract:** Client is expected to enforce contract
    - Less error checking
  - **Defensive Programming:** Paradigm where implementation checks for contractual violations and is responsible for maintaining correct state
    - Quicker error catching
- Use assert to maintain consistent internal state

8 Unified Modeling Language

- **Class Responsibility and Collaborator (CRC) Card:** Diagram showing the name, responsibilities, and dependencies of a class
- **Unified Modeling Language:** Diagram of classes and their relationships
- Each entity may have name, attributes, and/or operations
  - Operations:
    - Permissions are public (+) / protected (#) / private (-)
    - Example: + setLocation(Location) : void
  - Stereotype descriptor: Label above a UML entity
    - Use to declare an interface
- Relationships:
  - Has-a: Arrow with a solid line
  - Uses: Arrow with a dotted line
  - Is-a: Arrow with a solid line and closed head
  - Implements an interface: Arrow with a dotted line and closed head

9 REST APIs

- **Application Program Interface:** Protocol for communicating with a program component
  - **Create-Read-Update-Delete (CRUD):** Set of general operations to manage an entity
  - **Idempotent:** Operation which has the same response when executed multiple times
- **Hypertext Transfer Protocol (HTTP):**
  - Ports:
    - HTTP port: 80
    - HTTPS port: 443
- **Model View Controller API:** Web application architecture where server sends fully formed HTML pages
- **Representational State Transfer (REST):** Web application architecture standard where client and server send and receive chunks of data
  - Client sends a request to the server's endpoint
  - Improves performance by caching
  - **Uniform interface:** Interface which returns self-descriptive resources
  - **Stateless:** Interface where all information about each request is contained, and no connection state is maintained by the application
  - **Cacheable:** Implementation detail where as much information as possible is cached to minimize server load
- Request components:
  - **Method:** Action by an endpoint
    - POST, DELETE, GET, PUT
  - **Uniform Resource Locator (URL):** Endpoint to send the request to
  - **Path variable:** Data within a URL
  - **Query string:** Data as a non-hierarchical key-value pairs exposed in the URL, after a question mark
  - Only available for GET requests
  - **Header:** Data as a key-pair value
  - **Body:** Data as a block
- Response components:
  - **Response status code:**
  - E.g. 200, 201, 401, 403, 404, 500

9.1 Servers

- **Server:** Program in web development which exposes an API to consume
- **Controller:** Component in a server which manages the endpoints and parameters