CSCB07 Lecture Slides*

Vinesh Benny

Sunday 5th September, 2021

Contents

1 **Version Control** 1

2 **Introduction to Java** 4

3 **Object Oriented Programming (1)** 7

4 **Object Oriented Programming (2)** 10

5 **Software Testing** 13

6 **Design Patterns** 18

7 **SOLID Design** 20

8 **Software Development Life Cycle** 23

9 **I/O and Regular Expressions** 28

10 **Introduction to Android** 30

11 **Android – Storing Data** 32

12 **Android – Testing** 33

*Thanks to everyone in the CSCB07 Summer 2021 class.

1 Version Control

- Two flavours of Version Control
  - Centralized (B07 uses this)
  - Decentralized
- Centralised Version
  - Keep code in a centralized location (the "Repository")
  - Code in repository is the "Master Copy" (**Never** directly modify)
  - Instead make local copies of the repository on each computer you will be working on (working copy)
  - When major changes are made to local copy that you want to save, "commit" change to repo
  - Tools allow you to revert to a previous version of the source code (only when commits have occurred)
- Some Terminology
  - Repository/Repo
  - Working copy
  - Commit
  - Client program
  - Checkout
- Centralized systems include:
  - **S**ub**V**ersio**n** (**SVN**)
    - SVN is the successor to **C**oncurrent **V**ersions **S**ystem (**CVS**), and was built to help fix many issues in CVS
  - Git
  - ClearCase
  - Mercurial
  - Perforce
- SSH and SCP are **not** version control systems
  - **S**ecure **Sh**ell (SSH) is used to connect to a remote computer and work in a shell on that computer
  - **S**ecure **Co**p**y** (SCP) is used to:
    - Securely copy files from one computer to another
    - Transfer a copy of the files but does **not** version them
- Version Control – Managing Concurrency

  When two or more people want to edit the same file at the same time
  - Pessimistic concurrency
    - Only allow one writeable copy of each file
    - e.g. Microsoft Visual SourceSafe, Rational ClearCase
  - Optimistic concurrency
    - Allow writes, fix issues afterwards
    - Merging
      - SVN is either able to merge without help from the user, or
      - *Conflict*: SVN needs the user to resolve the conflict
    - e.g. Subversion, CVS, Perforce

- Optimistic Concurrency – Merging Options
  Select from: **(p)** postpone, **(df)** diff-full, **(e)** edit, **(mc)** mine-conflict, **(tc)** theirs-conflict and **(s)** show all options.

```
            (e) edit - changed merged file in an editor

       (df) diff-full - show all changes made to merged file4

        (r) resolved - accept merged version of file


  (dc) display-conflict - show all conflicts (ignoring merged version)

   (mc) mine-conflict - accept my version for all conflicts (same as above)

  (tc) theirs-conflict - accept their version for all conflicts (same as above)


      (mf) mine-full - accept my version of entire file (even non-conflicts)

      (tf) theirs-full - accept my version of entire file (same as above)


       (p) postpone - mark the conflict to be stored later

          (l) launch - launch external tool to resolve conflict

       (s) show all - show this list
```

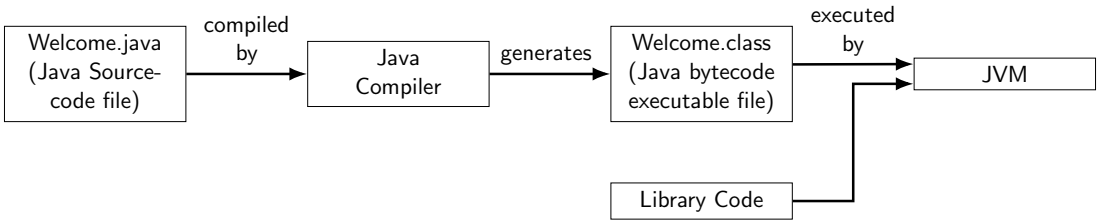
- Integrating the code – Reasons for merge conflicts
  - Communication
  - Experimental features being built
  - Complex code bases
  - More than one project on the go that impacts this code
  - Two features being built in same class by different developers
- Branching
  - Branches are divergent copies of development lines
  - These versions are used to build out complex features, or do experiments, without having an impact on the main code line
  - Strategies include:
    * No branching
    * Release branching
    * Feature branching
- Storage scheme
  - Storing every copy of every file generated over the course of a project is not practical
  - Version control systems store incremental differences in files/folder structures
  - These differences store enough information to re-construct previous versions, without storing every single copy ever made of the file
- What's Stored Where
  - Server side: out of scope
  - Local copy contains a special directory, **.svn**
    * It stores (locally) the information subversion needs to keep track of your files, version numbers, where the repository is, etc.

    * Needless to say, you should not mess with the contents of this directory. Let subversion do its job

- General rules
  - Update and commit frequently
  - Never break the main branch
  - Always comment clearly what changes are in a revision
  - Test all code before accepting merge
  - Communicate with your team!

2 Introduction to Java

- What is Java?
  - An object-oriented language invented by James Gosling in 1994 at Sun Microsystems
  - Write once, run anywhere (WORA)
  - Widely-used in industry
  - Used to develop software running on:
    - Desktop Computers
    - Servers
    - Mobile devices
- Java Programs
  1. Writing the source code using a text editor
  2. Translating the source code into Java bytecode using a compiler
     - Bytecode is similar to machine instructions but is architecture neutral and can run on any platform that has a Java Virtual Machine (JVM)
  3. Executing the bytecode
     - The JVM is an interpreter: it translates bytecode into the target machine language code one at a time rather than the whole program as a single unit
     - Each step is executed immediately after it is translated



- Integrated Development Environment
  - A system comprising several tools that facilitate software development and testing
  - Popular IDEs:
    - Eclipse
    - NetBeans
    - IntelliJ
- Data Types
  - Eight primitive types
    - byte, char, short, int, long, float, double, boolean
  - Objects
    - Defined using **classes**
    - Java provides wrapper classes to use primitive types as objects (e.g. Integer, Double, etc)
- Numeric Primitive Types

Name	*Range*	*Storage Size*
byte	-2^7 to $2^7 - 1$ $(-128$ to $127)$	8-bit signed
short	-2^{15} to $2^{15} - 1$ $(-32768$ to $32767)$	16-bit signed
int	-2^{31} to $2^{31} - 1$ $(-2147483648$ to $2147483647)$	32-bit signed
long	-2^{63} to $2^{63} - 1$	64-bit signed
	(i.e., -9223372036854775808 to 9223372036854775807)	
float	Negative range: $-3.4028235\text{E} + 38$ to $-1.4\text{E} - 45$	32-bit IEEE 754
	Positive range: $1.4\text{E} - 45$ to $3.4028235\text{E} + 38$	
double	Negative range: $-1.7976931348623137\text{E} + 38$ to $-4.9\text{E} - 324$	64-bit IEEE 754
	Positive range: $4.9\text{E} - 324$ to $1.7976931348623137\text{E} + 38$	

- Classes
  - A typical Java class includes the following:
    - Data fields to represent the state of an object
    - Methods to represent the behavior of an object. Each method has:
      - A return type (**void** if nothing is returned)
      - Zero or more arguments
    - Special type of methods, known as constructors, that perform initialization actions. A constructor:
      - Has no return type (not even **void**)
      - Has zero or more arguments
      - Should have the same name as the class
      - Is invoked using the **new** operator
  - Instantiation is creating an object (or an instance of a class)
- The *main* method
  - The main method is the entry point where the program begins execution
  - Should have the following form:

```
public static void main(String [] args) {
        //write your code here
}
```

- Default values
  - The default value of a data field is:
    - **null** for a reference type
    - **0** for a numeric type
    - **false** for a boolean type
    - **'\u0000'** for a char type
  - Java assigns no default value to a local variable inside a method
- Scope
  - The scope of fields and methods is the entire class
  - The scope of a local variable starts from its declaration until the end of the block that contains it
- Differences between Variables of Primitive Types and Reference Types
  - Every variable represents a memory location that holds a value
  - For a variable of a primitive type, the value is of the primitive type
  - For a variable of a reference type, the value is a reference to where an object is located (i.e. a pointer)
  - When you assign one variable to another:
    - For a variable of a primitive type, the real value of one variable is assigned to the other variable
    - For a variable of a reference type, the reference of one variable is assigned to the other variable.
- The *this* reference
  - The **this** *keyword* is the name of a reference that an object can use to refer to itself
  - It can be used to reference the object's instance members
- The *static* modifier
  - Static fields/methods can be accessed from a reference variable or from their class name
  - Non-static (or instance) fields/methods can only be accessed from a reference variable

- Arrays
  - An array is a data structure that represents a collection of the same types of data
  - Once an array is created, its size is fixed
    - e.g. **int [] A = new int[10];**
  - The size of an array A can be found using **A.length**
  - When an array is created, its elements are assigned the default value
  - Array elements could be initialized individually
    - e.g. **A[0] = 5;**
  - Array initializer (combines declaration, creation, and initialization)
    - e.g. **double[] myList = 1.9, 2.9, 3.4, 3.5;**
- Two-dimensional Arrays
  - The syntax for declaring a two-dimensional array is:
    - **elementType [][] arrayRefVar;** (e.g. **int[][] matrix;**)
  - For a two-dimensional array A, **A.length** returns the number of rows
  - Two-dimensional array examples:

	[0]	[1]	[2]	[3]	[4]
[0]	0	0	0	0	0
[1]	0	0	0	0	0
[2]	0	0	0	0	0
[3]	0	0	0	0	0
[4]	0	0	0	0	0

```
matrix = new int[5][5];
```

	[0]	[1]	[2]	[3]	[4]
[0]	0	0	0	0	0
[1]	0	0	0	0	0
[2]	0	7	0	0	0
[3]	0	0	0	0	0
[4]	0	0	0	0	0

```
matrix[2][1] = 7;
```

	[0]	[1]	[2]
[0]	1	2	3
[1]	4	5	6
[2]	7	8	9
[3]	10	11	12

```
int[][] array = {
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9},
  {10, 11, 12}
};
```

3 Object Oriented Programming (1)

- Object-Oriented Thinking
  - Procedural paradigm
    - Focuses on designing methods
    - Data and operations on the data are separate
  - Object-oriented paradigm
    - Couples methods and data together into objects
    - Organizes programs in a way that mirrors the real world
    - A program can be viewed as a collection of cooperating objects
    - Makes programs easier to develop and maintain
    - Improves software reusability
- Inheritance
  - Powerful feature for reusing software
  - Helps avoid redundancy
  - Different objects might have common properties and behaviors
    - e.g. Person, Employee
  - Inheritance allows developers to
    - Define a general class (or superclass). E.g. Person
    - Extend the general class to a specialized class (or subclass). e.g. Employee
  - In Java, the keyword extends is used to indicate inheritance
- Casting objects and the **instanceof** operator
  - It is always possible to cast an instance of a subclass to a variable of a superclass (known as upcasting)
    - e.g. **Person p = new Employee();**
  - When casting an instance of a superclass to a variable of its subclass (known as *downcasting*), explicit casting must be used
    - e.g. **Person p = new Employee(); Employee e = (Employee)p;**
    - If the superclass object is not an instance of the subclass, a runtime error occurs
    - It is a good practice to ensure that the object is an instance of another object before attempting a casting. This can be accomplished by using the **instanceof** operator
  - Casting an object reference does not create a new object
- Overloading and Overriding
  - Overloading
    - Defining methods having the same name but different signatures
      - Signature: method name + types of its formal parameters
    - Overloading methods can make programs clearer and more readable
  - Overriding
    - Defining a method in the subclass using the same signature and the same return type as in its superclass
    - The **@Override** annotation helps avoid mistakes
    - A static method *cannot* be overridden (it can be invoked using the syntax SuperClassName.staticMethodName)

- The **super** keyword
  - Refers to the superclass
  - Can be used to invoke a superclass constructor
    * Syntax: **super**() or **super**(parameters)
    * Must be the first statement of the subclass constructor
    * A constructor may invoke an overloaded constructor or its superclass constructor. If neither is invoked explicitly, the compiler automatically puts super() as the first statement in the constructor
    * If a class is designed to be extended, it is better to provide a no-argument constructor to avoid programming errors
  - Can be used to invoke a superclass method
    * Syntax: **super.methodName**(parameters)
    * Useful in the case of overridden methods
- The **Object** class
  - Every Java class has **Object** as superclass
  - It has methods that are usually overwritten
    * **equals**
    * **hashCode**
    * **toString**
  - **equals** method
    * Header: **boolean equals(Object obj)**
    * The implementation provided by the **Object** class checks whether two reference variables point to the same object
      · Does not check "logical equality"
  - **hashCode** method
    * Header: **int hashCode()**
    * The implementation provided by the **Object** class returns the memory address of the object
    * The **hashCode** method should be overridden in every class that overrides **equals**
      · Equal objects must have equal hash codes
    * A good hashCode method tends to produce unequal hash codes for unequal objects
  - **toString** method
    * Header: **String toString()**
    * The **toString** method is automatically invoked when an object is passed to **println** and the string concatenation operator
    * Class **Object** provides an implementation of the **toString** method that returns a string consisting of the class name followed by an "at" sign (@) and the unsigned hexadecimal representation of the hash code
    * **toString** is usually overridden so that it returns a descriptive string representation of the object
- Polymorphism
  - Every instance of a subclass is also an instance of its superclass, but not vice versa
  - Polymorphism: An object of a subclass can be used wherever its superclass object is used

- Example

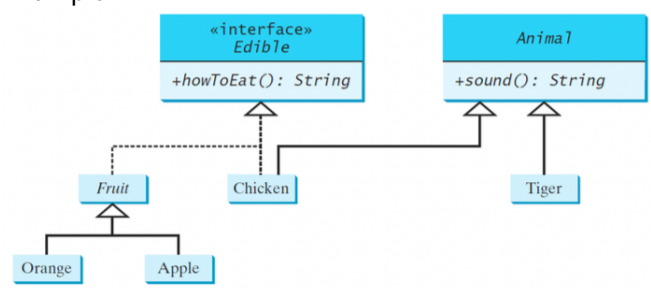
```
public class Demo {
        public static void main(String [] args) {
                m(new Point(1,2));
        }

        public static void m(Object x) {
                System.out.println(x);
        }
}
```

- Dynamic Binding
  - A method can be implemented in several classes along the inheritance chain
  - The JVM dynamically binds the implementation of the method at runtime, decided by the actual type of the variable
    * `Object x = new Point(1,2); // declared type: Object, actual type: Point`
  - Dynamic binding works as follows:
    * Suppose an object x is an instance of classes $C_1, C_2, \ldots, C_{n-1}$, and C_n, where C_1 is a subclass of C_2, C_2 is a subclass of $C_3, \ldots$, and C_{n-1} is a subclass of C_n,
    * If x invokes a method p, the JVM searches for the implementation of the method p in $C_1, C_2, \ldots, C_{n-1}$, and C_n, in this order, until it is found. Once an implementation is found, the search stops and the first-found implementation is invoked

- Encapsulation
  - The access control mechanism in Java facilitates encapsulation
  - There are four possible access levels for members, listed in order of increasing accessibility:
    1. **private** – The member is accessible only from the top-level class where it is declared
    2. **package-private** – The member is accessible from any class in the package where it is declared (default access)
    3. **protected** – The member is accessible from subclasses of the class where it is declared and from any class in the package where it is declared
    4. **public** – the member is accessible from anywhere
  - Rule of thumb: *make each member as inaccessible as possible*

4 Object Oriented Programming (2)

- Abstract Classes
  - Cannot be instantiated using the **new** operator
  - Usually contain abstract methods that are implemented in concrete subclasses
    - e.g. computeArea() in GeometricObject
  - Abstract classes and abstract methods are denoted using the **abstract** modifier in the header
  - A class that contains abstract methods must be defined as abstract
  - If a subclass of an abstract superclass does not implement all the abstract methods, the subclass must be defined as abstract
- Interfaces
  - An interface can be used to define common behaviour for classes (including unrelated classes)
  - Contains only constants and abstract methods
  - Interfaces are denoted using the **interface** modifier in the header
  - Example




```java
abstract class Fruit implements Edible {
  // Data fields, constructors, and methods omitted here
}

class Apple extends Fruit {
  @Override
  public String howToEat() {
    return "Apple: Make apple cider";
  }
}

class Orange extends Fruit {
  @Override
  public String howToEat() {
    return "Orange: Make orange juice";
  }
}
```

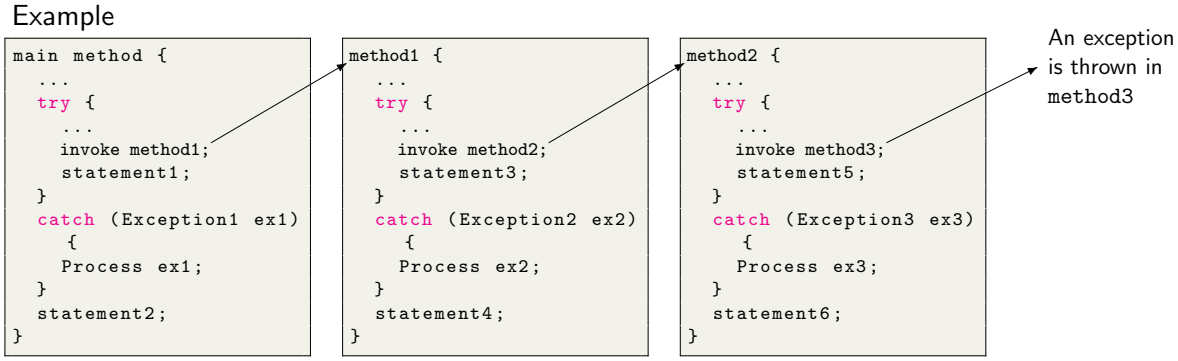
- Generics
  - Enable type parameterization
    - Generic interfaces
    - Generic classes
    - Generic methods
  - Example: **ArrayList** class
    - ArrayList<Integer> A = new ArrayList<Integer>();
    - ArrayList<String> B = new ArrayList<String>();
  - Generic types must be reference types
  - Enable error detection at compile time
  - The **Comparable** interface
    - Defines the **compareTo** method for comparing objects
    - Defined as follows:

```java
public interface Comparable<T> {
        public int compareTo(T t);
}
```

    - The **compareTo** method determines the order of the calling object with **t** and returns a negative integer, zero, or a positive integer if the calling object is less than, equal to, or greater than **t**
    - Many classes implement Comparable (e.g. **String**, **Integer**)
  - The **ArrayList** class
    - Arrays can be used to store lists of objects. However, once an array is created, its size is fixed

    - Java provides the generic class **ArrayList** whose size is variable
    - Imported using: **import java.util.ArrayList;**
    - Commonly used methods (**ArrayList<E>**)
        - **boolean add(E e)**
        - **E get(int index)**
        - **int size()**
        - **boolean contains(Object o)**
        - **int indexOf(Object o)**
    - An **ArrayList** could be traversed using a for-each loop
  - The **HashSet** class
    - Generic class that can be used to store elements without duplicates
        - No two elements e1 and e2 can be in the set such that e1.equals(e2) is true
    - Imported using: **import java.util.HashSet;**
    - Objects added to the hash set should override **equals** and **hashCode** properly
    - Commonly used methods (**HashSet<E>**)
        - **boolean add(E e)**
        - **int size()**
        - **boolean contains(Object o)**
    - A **HashSet** could be traversed using a for-each loop
  - The **LinkedHashSet** class
    - Elements of a **HashSet** are not necessarily stored in the same order they were added
    - **LinkedHashSet** is a subclass of **HashSet** with a linked-list implementation that supports an ordering of the elements in the set
    - Imported using: **import java.util.LinkedHashSet;**

- Exceptions
  - Example



  - Java has a **finally** clause that can be used to execute some code regardless of whether an exception occurs or is caught. For example:

```
try {
        //statements;
}
catch Exception ex) {
        //handling ex; }
finally {
        //final statements;
}
```

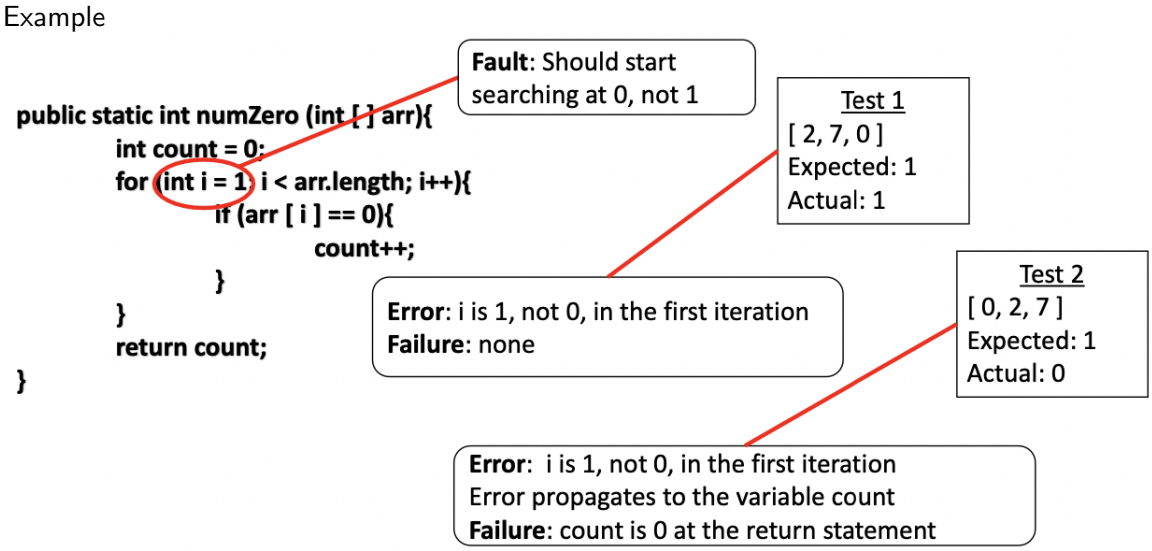
Object Oriented Programming - Design Guidelines

- Methods Common to All Objects
  - Always override hashCode when you override equals
  - Always override toString
  - ConsiderimplementingComparable
- Classes and Interfaces
  - Minimize accessibility
  - Favor composition over inheritance
  - Prefer interfaces to abstract classes
  - Prefer lists to arrays
- Methods
  - Check parameters for validity
  - Return empty arrays or collections, not **null**
  - Document your API properly
- Exceptions
  - Use exceptions only for exceptional conditions
  - Use checked exceptions for recoverable conditions and runtime exceptions for programming errors
  - Do not ignore exceptions
- General Programming (Naming Conventions)
  - Package names should be hierarchical with the components separated by periods. Components should consist of lowercase alphabetic characters and, rarely, digits. e.g. **javax.swing.plaf.metal**
  - Class and interface names should consist of one or more words, with the first letter of each word capitalized
  - Method and field names follow the same typographical conventions as class and interface names, except that the first letter of a method or field name should be lowercase, for example, **ensureCapacity**
  - The names of constant fields should consist of one or more uppercase words separated by the underscore character, for example, **NEGATIVE_INFINITY**
  - Local variable names have similar typographical naming conventions to member names, except that abbreviations are permitted, as are individual characters and short sequences of characters whose meaning depends on the context in which the local variable occurs, for example, **i**, **xref**

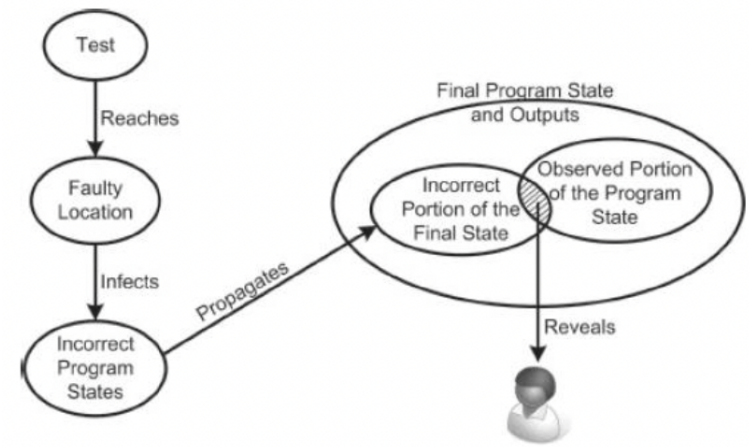
5 Software Testing

- What is Software Testing?
  - Running a program in order to find faults
    - Examining the code without execution is <u>not</u> testing
  - The main practical approach to validate/verify software
    - Formal methods that aim at proving the correctness of a program are not scalable
  - "Program testing can be used to show the presence of bugs, but never to show their absence!" – Edsger W. Dijkstra
- Testing Levels
  - Acceptance testing
    - Test whether the software is acceptable to the user
  - System testing
    - Test the overall functionality of the system
  - Integration testing
    - Test how modules interact with each other
  - Module testing
    - A module is a collection of related units the are assembled in a file, package, or class
    - Test modules in isolation including how the components interact with each other
    - <u>Responsibility of the programmer</u>
  - Unit testing
    - Test units (methods individually)
    - <u>Responsibility of the programmer</u>
- Black-Box and White-Box Testing
  - Black-Box Testing
    - Test are derived from external descriptions of the software
  - White-Box Testing
    - Test are derived from source code internals of the software
    - More expensive to apply
- Why is Software Testing Hard?
  - Exhaustive testing is infeasible
    - e.g. Exhaustively testing a method with two integer parameters would require $\sim 10^{19}$ tests
  - Random/statistical testing is not effective
- Why Do We Test Software?
  - Software is everywhere
    - Communication, transportation, healthcare, finance, education, etc.
  - Software failures could have severe consequences
    - A 2002 NIST report estimated that defective software costs the U.S. economy $59.5 billion per year and that improvements in testing could reduce this cost by about a third
    - In certain areas such as healthcare and transportation, software failures could cost lives

- Infamous Software Failures
  - Northeast blackout of 2003
    - Caused by a failure of the alarm system
    - Affected 40 million people in USA and 10 million people in Canada
    - Contributed to at least 11 deaths
    - Cost around $6 billion
  - Ariane 5 explosion (1996)
    - Unhandled floating point conversion exception
    - Estimated loss: $370 million
  - NASA's Mars lander (1999)
    - Crashed due to an integration fault
    - Estimated loss: $165 million
  - Boeing 737 Max
    - Crashed due to overly aggressive software flight overrides
  - Boeing A220
    - Engines failed after software update allowed excessive vibrations
  - Toyota brakes failure
    - Dozens dead
    - Thousands of crashes
  - Therac-25 radiation therapy machine
    - Three patients were killed
- Fault/Error/Failure
  - **Software Fault**: A static defect in the software
  - **Software Error**: An incorrect internal state that is the manifestation of some fault
  - **Software Failure**: External, incorrect behavior with respect to the requirements or another description of the expected behaviour
  - The term **bug** is often used informally to refer to all three of fault, error, and failure
    - The first computer bug was an actual bug!
  - Example

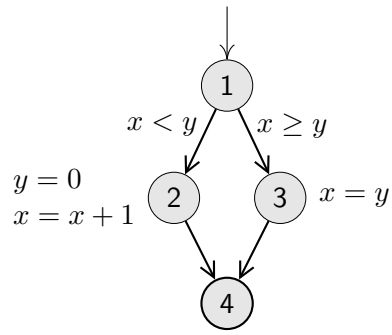


- The RIPR model
  - Four conditions are needed for a failure to be observed
    1. **Reachability**: a test must reach the location in the program that contains the fault
    2. **Infection**: After the faulty location is executed, the state of the program must be incorrect
    3. **Propagation**: The infected state must propagate through the rest of the execution and cause some output or final state of the program to be incorrect
    4. **Revealability**: The tester must observe part of the incorrect portion of the final program state



- Criteria-based Test Design
  - **Coverage Criterion**: A rule or collection of rules that impose test requirements on a test set
    - e.g. For each statement in the code, there should be at least one test case that covers it
  - Coverage criteria give us structured, practical ways to search the input space. Satisfying a coverage criterion gives a tester some amount of confidence in two crucial goals:
    1. We have looked in many corners of the input space, and
    2. Our tests have a fairly low amount of overlap
  - Criteria subsumption
    - C_1 subsumes C_2 if and only if every test set that satisfies C_1 satisfies C_2
- Graph Coverage
  - The software is modeled as a graph where nodes and edges could represent:
    - Methods and calls
    - Statements and branches
    - Etc.
  - Coverage criteria are defined based on the graph. For example:
    - Cover every node
    - Cover every edge
    - Cover every path
    - Etc.
  - Example (Control Flow Graph)

```
if (x < y)
{
  y = 0;
  x = x + 1;
}
else
{
  x = y;
}
```



- Logic Coverage
  - Involves the boolean expressions of the code
  - Coverage criteria include:
    - Predicate coverage
    - Clause coverage
    - Combinational coverage
    - Etc.
  - Example

```
if(((a>b) || c) && (x<y))
...
else
...
```

  - Predicate coverage
    - The test set should make each predicate evaluate to true and false
    - e.g. $((a > b) \,||\, c) \,\&\&\, (x < y) = \{\text{True}, \text{False}\}$
  - Clause coverage
    - The test set should make each clause evaluate to true and false
    - e.g. $(a > b) = \{\text{True}, \text{False}\}, c = \{\text{True}, \text{False}\}, (x < y) = \{\text{True}, \text{False}\}$
- Active clause coverage
  - Clause coverage has a weakness
    - The values do not always make a difference
  - A clause c_i in predicate p, called the major clause, determines p *if and only if* the values of the remaining minor clauses c_j are such that changing c_i changes the value of p
  - Two requirements for each c_i: c_i evaluates to true and c_i evaluates to false
  - This is a form of MCDC, which is required by the FAA for safety critical software
- Inactive clause coverage
  - Ensures that "major" clauses do not affect the predicates
  - Four requirements for each c_i
    1. c_i evaluates to true with p true
    2. c_i evaluates to false with p true
    3. c_i evaluates to true with p false
    4. c_i evaluates to false with p false
  - Example

    * Testing the control software for a shutdown system in a reactor where the specification states that the status of a particular valve (**open** vs. **closed**) is relevant to the reset operation in **Normal** mode, but not in **Override** mode

- Test Oracles
  - A *test oracle* is an encoding of the expected results of a given test
    * e.g. JUnit assertion
  - Must strike a balance between checking too much (unnecessary cost) and checking too little (perhaps not revealing failures)
  - What should be checked?
    * The output state is everything that is produced by the software under test, including outputs to the screen, file, databases, messages, and signals
    * Each test should have a goal and testers should check the output(s) that are mainly related to that goal
      · At the unit testing level, checking the return values of the methods and returned parameter values are almost always enough
      · At the system level, it is usually sufficient to check the directly visible output such as to the screen
  - How to determine what the correct results are
    * Specification-Based direct verification of outputs
      · e.g. "a **sort** program should produce a permutation of its input in increasing order "
      · Specifications are hard to write
    * Redundant computations
      · Refer to another trustworthy implementation of the program
      · Usually used for regression testing
    * Consistency checks
      · Check whether certain properties hold (e.g. a value representing probability should neither be negative nor larger than one)

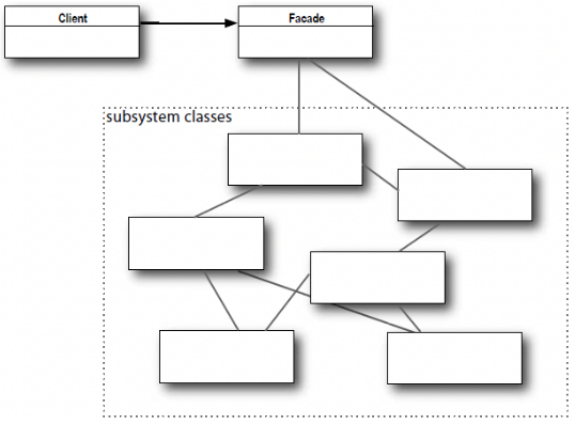
6 Design Patterns

- What are Design Patterns
  - Descriptions of communicating objects and classes that are customized to **solve** a general design **problem** in a particular **context**
  - Gamma et al. described 23 design patterns divided into three categories:
    1. Creational patterns
    2. Structural patterns
    3. Behavioral patterns
- Creational Patterns
  - Concern the process of object creation
  - Six creational patterns
    1. Factory Method
    2. Abstract Factory
    3. Singleton
    4. Prototype
    5. Builder
    6. Object Pool
- Structural Patterns
  - Deal with the composition of classes or objects
  - Seven structural patterns
    1. Adapter
    2. Bridge
    3. Composite
    4. Decorator
    5. Facade
    6. Flyweight
    7. Proxy
- Behavioral Patterns
  - Characterize the ways in which classes or objects interact and distribute responsibility
  - Ten Behavioral patterns
    1. Chain of Responsibility
    2. Command
    3. Interpreter
    4. Iterator
    5. Mediator
    6. Memento
    7. Observer
    8. State
    9. Strategy
    10. Template
- Singleton (Creational)
  - Intent: Ensure a class has only one instance, and provide a global point of access to it

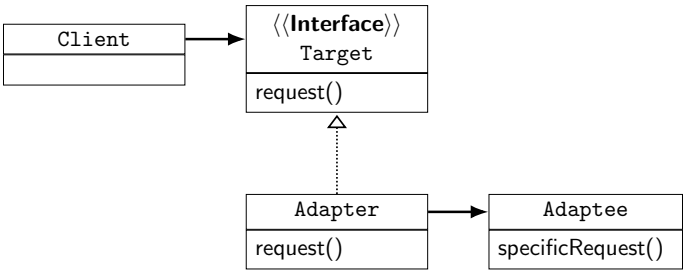
Singleton
−instance: Singleton
−Singleton() +getInstance(): Singleton ...

(− means private, + means public)

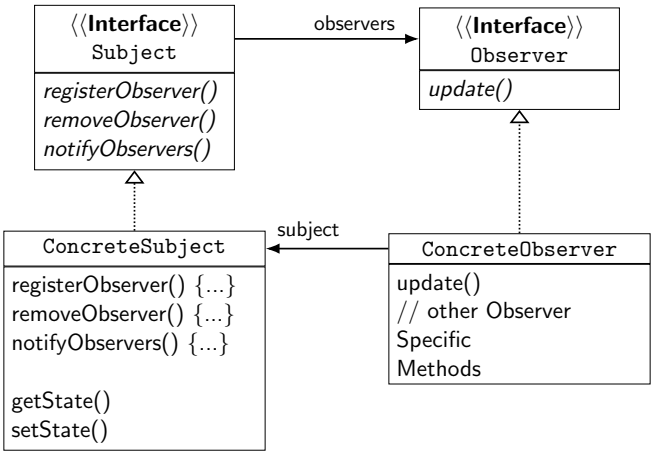
- Facade (Structural)
  - Intent: Hide complexities and provide a unified interface to a set of interfaces in a subsystem



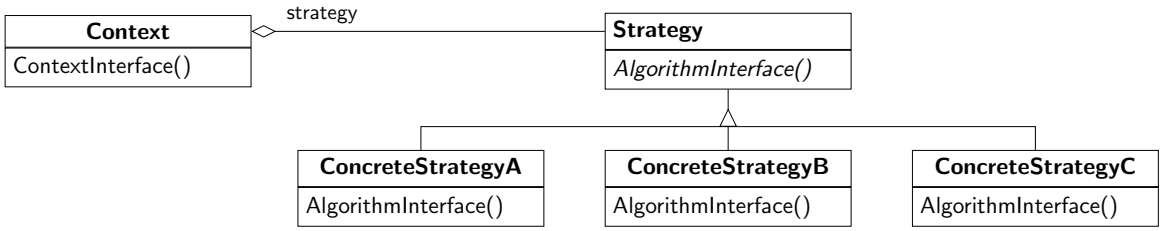
- Adapter (Structural)
  - Intent: Let classes work together that couldn't otherwise because of incompatible interfaces



- Observer (Behavioral)
  - Intent: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

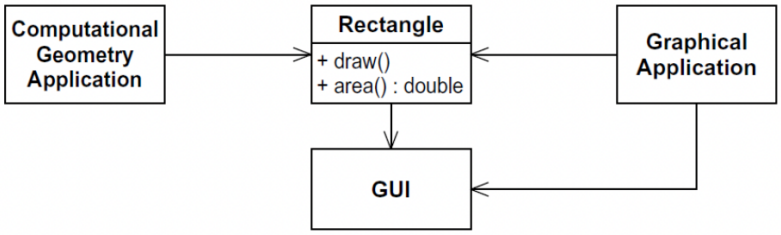


- Strategy (Behavioral)
  - Intent: Define a family of algorithms, encapsulate each one, and make them interchangeable

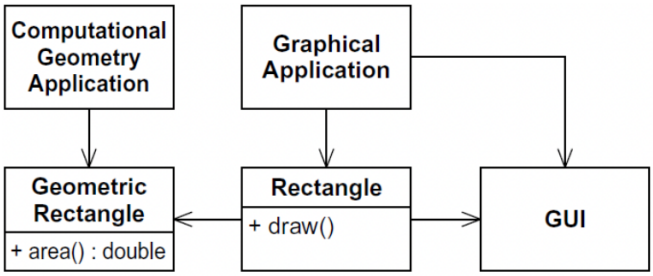


7 SOLID Design

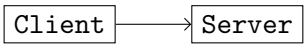
- What is SOLID?
  - **S**ingle Responsibility Principle
  - **O**pen/Closed Principle
  - **L**iskov Substitution Principle
  - **I**nterface Segregation Principle
  - **D**ependency Inversion Principle
- Single Responsibility Principle (SRP)
  A class should have only one reason to change
  - If you can think of more than one motive for changing a class, then that class has more than one responsibility
  - If a class has more than one responsibility, then the responsibilities become coupled
  - Violating the SRP



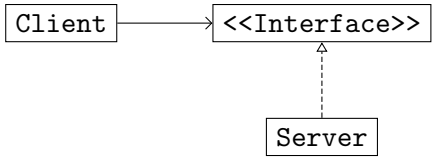
  - Conforming to the SRP



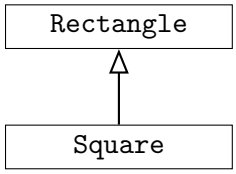
- The Open/Closed Principle (OCP)
  Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.
  - When a single change to a program results in a cascade of changes to dependent modules, the design smells of rigidity.
    * If the Open/Closed principle is applied well, then further changes of that kind are achieved by adding new code, not by changing old code that already works.
  - In Java, it is possible to create abstractions that are fixed and yet represent an unbounded group of possible behaviors
    * The abstractions are abstract base classes, and the unbounded group of possible behaviors is represented by all the possible derivative classes.
  - Violating the OCP
    * Both classes are concrete
    * The **Client** uses the **Server** class



- Conforming to the OCP



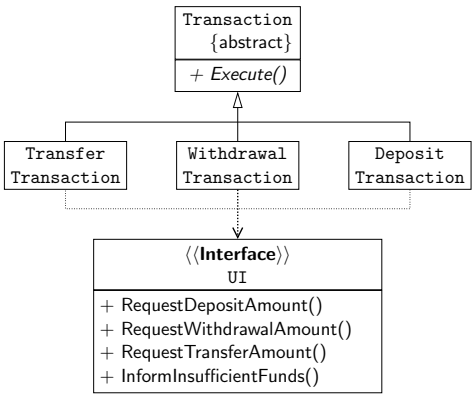
- The Liskov Substitution Principle (LSP)
  Subtypes must be substitutable for their base types.
  - Formally: Let $\Phi(x)$ be a property provable about objects x of type T. Then $\Phi(y)$ should be true for objects y of type S where S is a subtype of T.
  - Counter-example: "If it looks like a duck, quacks like a duck, but needs batteries – you probably have the wrong abstraction"
  - Violating the LSP
    Issues:
    * Inheriting **height** and **width**
    * Overriding **setHeight** and **setWidth**
    * Conflicting assumptions. For example:



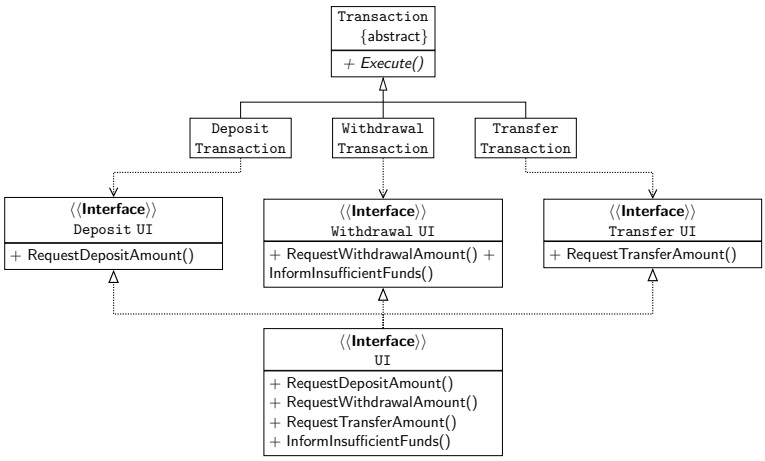

```
void testRectangleArea(Rectangle r){
        r.setWidth(5);
        r.setHeight(4);
        assertEquals(r.computeArea(), 20);
}
```

  - Implication: A model, viewed in isolation, cannot be meaningfully validated.
    * The validity of a model can only be expressed in terms of its clients.
    * One must view the design in terms of the reasonable assumptions made by the users of that design.

- The Interface Segregation Principle (ISP)
  Clients should not be forced to depend on methods that they do not use.
  - This principle deals with classes whose interfaces are not cohesive. That is, the interfaces of the class can be broken up into groups of methods where each group serves a different set of clients.
  - When clients are forced to depend on methods that they don't use, then those clients are subject to changes to those methods.
  - Violating the ISP



  - Conforming to the ISP

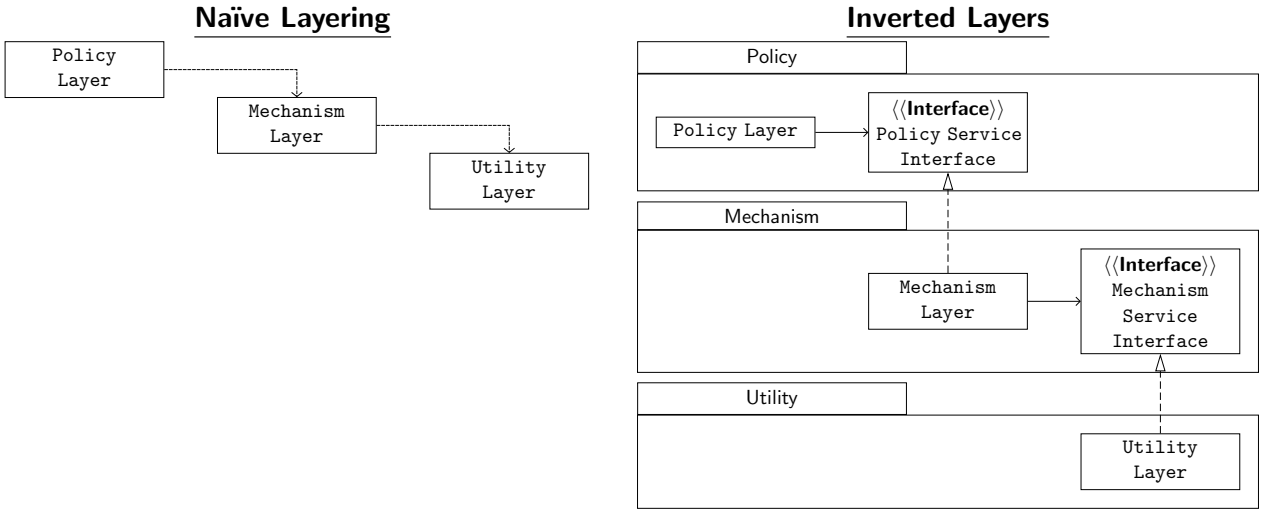


- The Dependency-Inversion Principle (DIP)

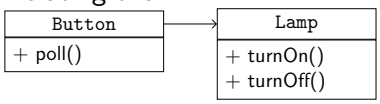
  A. High-level modules should not depend on low-level modules. Both should depend on abstractions.

  B. Abstractions should not depend on details. Details should depend on abstractions.

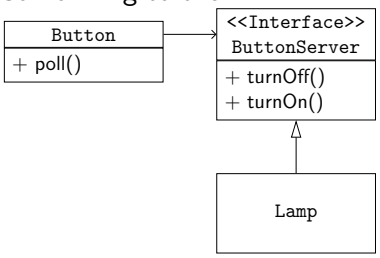
  - The modules that contain the high-level business rules should take precedence over, and be independent of, the modules that contain the implementation details.
  - When high-level modules depend on low-level modules, it becomes very difficult to reuse those high-level modules in different contexts.



  - Violating the DIP



  - Conforming to the DIP

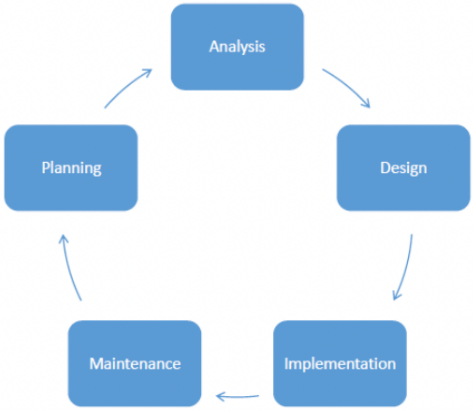


- Design Smells

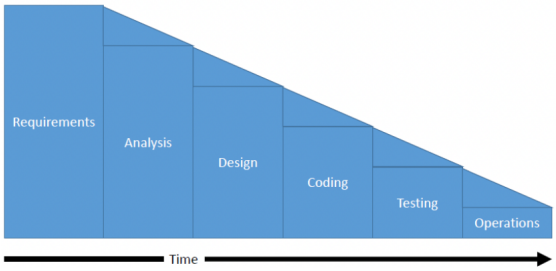
  - Symptoms of poor design
  - Often caused by the violation of one or more of the design principles
    * For example, the smell of *Rigidity* is often a result of insufficient attention to OCP.
  - These symptoms include:
    1. Rigidity – The design is hard to change.
    2. Fragility – The design is easy to break.
    3. Immobility – The design is hard to reuse.
    4. Viscosity – It is hard to do the right thing.
    5. Needless Complexity – Overdesign.
    6. Needless Repetition – Mouse abuse.
    7. Opacity – Disorganized expression.

8 Software Development Life Cycle

- Software Development Life Cycle (SDLC)
  - **Planning** – develop a plan for creating the concept or evolution of the concept
  - **Analysis** – analyze the needs of those using the system. Create detailed requirements
  - **Design** – Translate the detailed requirements into detailed design work
  - **Implementation** – Complete the work of developing and testing the system
  - **Maintenance** – Complete any required maintenance to keep the system running



- Different SDLC implementations
  - Rigid timeline / budget (Waterfall)
  - Rick Adverse (Spiral)
  - Quality Deliverables / Less management (Agile)
- Waterfall
  - A sequential (non-iterative) model
  - Involves a large amount of upfront work, in an attempt to reduce the amount of work done in later phases of the project



- Spiral
  - Risk-driven model
  - More time is spent on a given phase based on the amount of risk that phase poses for the project

- Agile
  - Issues with Waterfall
    - Inappropriate when requirements change frequently
    - Time gets squeezed the further into the process you get
  - Agile Methodologies
    - Extreme Programming (XP)

    * Scrum
    * Test-driven Develpoment (TDD)
    * Feature-driven Development (FDD)
    * Etc.

- 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*
  Reponding 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."

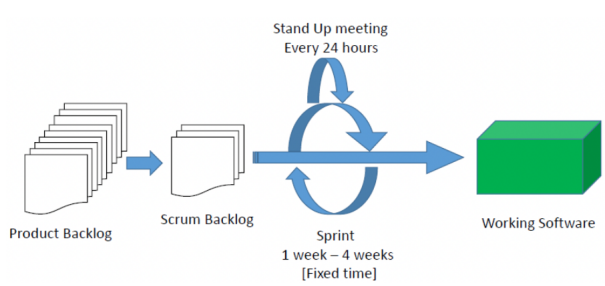
- Agile vs. Waterfall

	Agile	**Waterfall**
Iterative?	Yes	No
Late changes?	Yes	No / $$$
Fixed timeline?	No*	Yes
Fixed cost?	No*	Yes*
Volume of meetings	Consistent	Heavy up front, reduced middle, heavy end
Release frequency	Every sprint	Once per project
Business Involvement	Heavy throughout	Heavy early, and at very end
Cost to fix mistakes	Low	High

- eXtreme Programming (XP)
  - One of the most rigorous forms of Agile
  - Involves building a series of feedback loops, which are used to help guide when change can occur and allow for changes to be quickly integrated into the plan for development
  - Built on the idea that you can reduce the cost of developing software, and build better software, by having goals
  - XP requires that everything that can be unit tested is unit tested, that everyone works in pairs, and that these pairs change frequently

- Code Review
  - Although pair programming has gone out of vogue along with XP, it is important to note a practice that has become common place that was born from this idea – Code Review.
  - A code review is a session in which you **must sit down with another developer** from the team and walk them through your implementation line-by-line in order to get advice and feedback.
  - This process has been shown to lead to better code, through finding bugs earlier, and an increased amount of collaboration on difficult concepts.

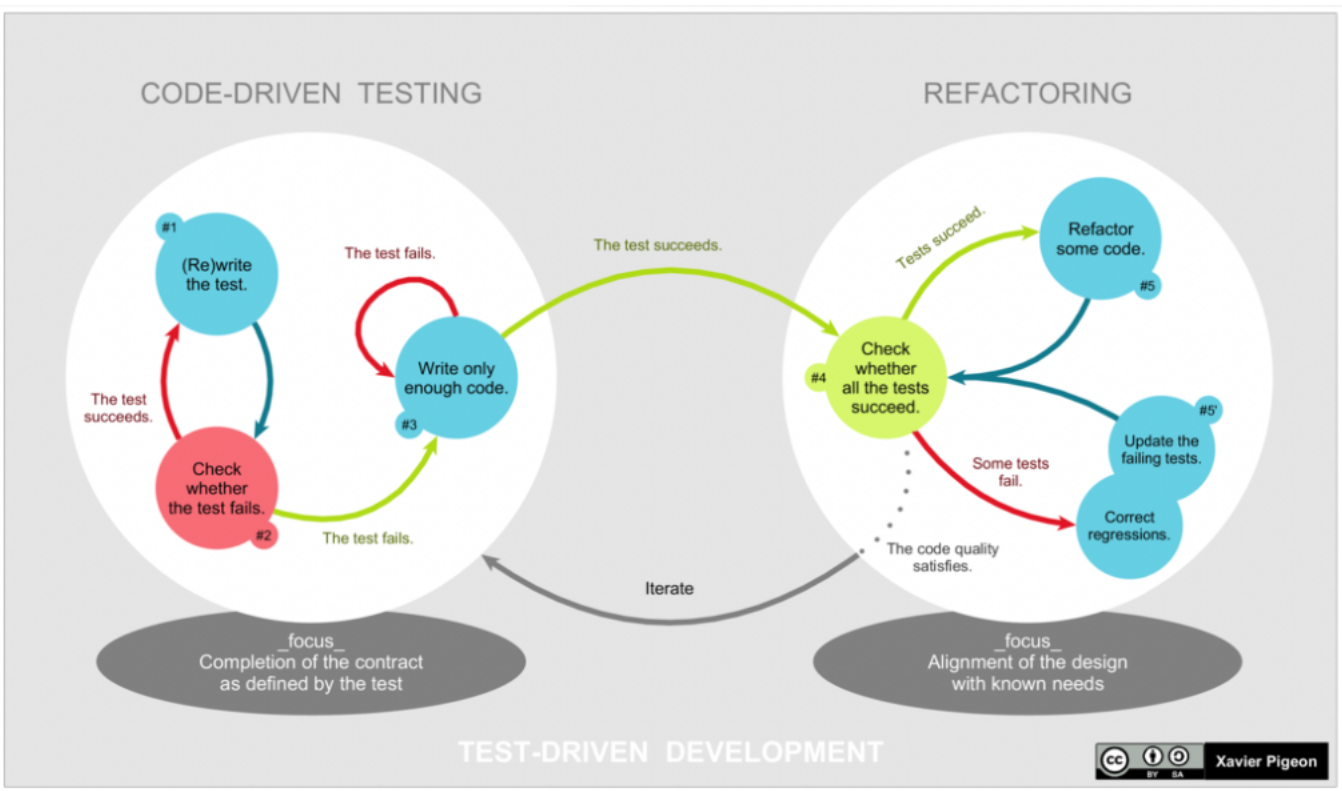
- Scrum
  - Scrum is currently one of the most widely used methodologies of software development



- Scrum - Roles
  - Product Owner
    * Responsible for delivering requirements and accepting demos
    * Involved in planning session
  - Scrum Master
    * Responsible for removing impediments
  - Team members
    * No one has a fixed role other than the scrum master and product owner
    * Everyone takes on tasks, and completes them based on what they are most comfortable with
- Scrum - Sprint
  - The sprint is a fixed time to deliver a working set of features, that are reviewed in a demonstration to the product owner
  - Tasks in Scrum are broken into "User Stories"
  - In a sprint, a team agrees at the beginning to take on a certain number of user stories
  - Sprints are usually between 1 and 4 weeks in length
  - At the end of each sprint, teams hold a "retrospective" which is a meeting where the past sprint is discussed, and chances for improvement for the next sprint are raised
- Scrum - User Stories
  - User stories are similar to requirements. They are written in the following format:
    As a $\{ACTOR/OBJECT\}$ *I want to* $\{ACTION\}$ *so that* $\{RESULT\}$
- Scrum - Planning Poker
  - In scrum, we do not assign time to tasks, but assign arbitrary points. This is a form of estimation that helps gauge how much work something will take to complete.
  - Planning poker takes a set of pre-determined numbers (usually: 1, 2, 3, 5, 8, etc.) and gets you to estimate how much work something will be relative to a known task.
  - After discussing the story at hand , everyone selects a card. Then, the cards are turned over simultaneously. Usually time is given for those who had the lowest and highest numbers to state their case
  - The process is repeated until everyone ends up at the same number.
- Scrum - Planning Session
  - Planning sessions happen at the start of each sprint.
  - They usually take a few hours. During this time, the team decides how much work it will take on, and discusses any major technical challenges they expect to face.
  - Usually, Product Owners are available for at least a portion of this meeting, to help with prioritization. They are only there to assist in this regard, and not to dictate what the team will complete.
- Scrum - The Standup Meeting
  - Happens EVERY SINGLE day that you are working
  - The goal is to make sure people are doing alright
  - Shouldn't be longer than 15 minutes
  - Answer three questions:
    1. What did I finish since the last standup?

2. What am I going to finish by the next standup?
3. What is stopping me / what impediments am I facing?

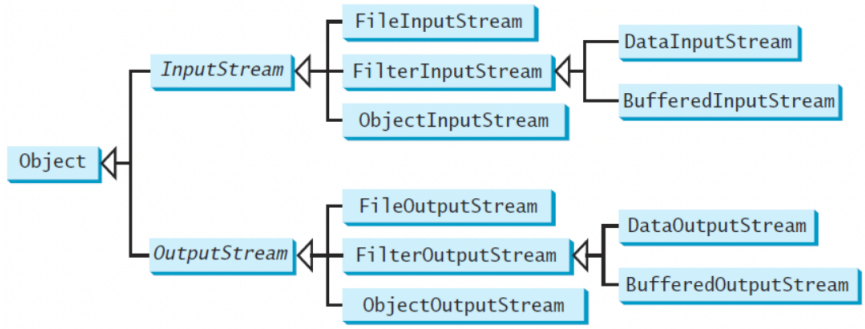
- Scrum - Working Agreement
  - A series of statements that everyone on the team agrees to about how the team will work
  - Things in working agreements may include:
    * The standup will occur at 1:00 pm every day, and last 15 minutes
    * We will not speak during the standup, unless it is our turn to speak
    * Our meetings will take place in the lobby of the IC building
    * All code must be peer-reviewed
    * We will submit all code 24 hours prior to the due date
- Scrum - Definition of Done
  - A formal agreement of when work is considered complete
  - For example, a story can be marked as done when:
    * It has been fully unit tested
    * It successfully integrated with the rest of the code
    * It has been peer reviewed
    * It is fully commented
    * Etc.
  - It is important that team comes to an agreement on this definition before they start work.
- Test Driven Development (TDD)
  - TDD is a way to develop software that revolves around writing test cases.
  - The basic concept is to write the unit tests needed to be passed for a feature to be considered working. You then code to the unit tests –writing the minimum amount for the tests to succeed.
  - Once working, you review and refactor. Then move on to the next set of tests.



- Feature Driven Development (FDD)
  - Based on the idea of building a focused model for the project, and the iterating on the features needed.
  - Splits development into 5 major pieces:
    1. Develop overall model
    2. Build feature list
    3. Plan by feature
    4. Design by feature
    5. Build by feature

9 I/O and Regular Expressions

- Input and Output (I/O)
  - Input sources include:
    - Keyboard
    - File
    - Network
  - Output destinations include:
    - Console
    - File
    - Network
- Input and Output Streams
  - Java handles inputs and outputs using streams



- Standard I/O
  - **System.in**
    - Object of type **InputStream**
    - Typically refers to the keyboard
    - Reading data could be done using the **Scanner** class. Its methods include:
      - **String next()**
      - **int nextInt()**
      - **String nextLine()**
      - **double nextDouble()**
  - **System.out**
    - Object of type **PrintStream**
    - Typically refers to the console
- The **File** class
  - Contains methods for obtaining the properties of a file/directory and for renaming and deleting a file/directory
  - Files could be specified using absolute or relative names
  - Constructing a **File** instance does not create a file on the machine
  - Methods include:
    - **boolean createNewFile()**
    - **boolean delete()**
    - **boolean exists()**
    - **boolean isDirectory()**
    - **File [] listFiles()**
- File I/O
  - Reading could be done using the **Scanner** class
    - e.g. **Scanner input = new Scanner(new File(filename));**
  - Writing could be done using the **FileWrite** class
    - e.g. **FileWriter output = new FileWriter(filename, append);**

- Regular Expressions
  - A regular expression (abbreviated regex) is a string that describes a pattern for matching a set of strings.
  - Regular expressions provide a simple and effective way to validate user input
    - e.g. phone numbers
  - Java supports regular expressions using the **java.util.regex** package
  - The **Pattern** class can be used to define the pattern
    - The **compile** method takes a string representing the regular expression as an argument and compiles it into a pattern
  - The **Matcher** class can be used to search for the pattern. Its methods include:
    - **boolean find()**
    - **boolean matches()**
  - Example

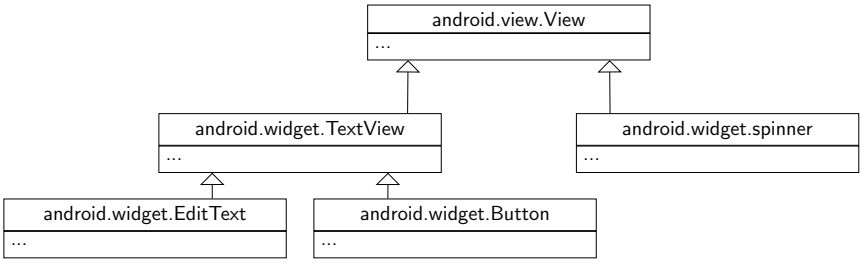
```
Pattern pattern = Pattern.compile("H.*d");
Matcher matcher = pattern.matcher("Hello World");
System.out.println(matcher.matches());
```

- Commonly Used Regular Expressions

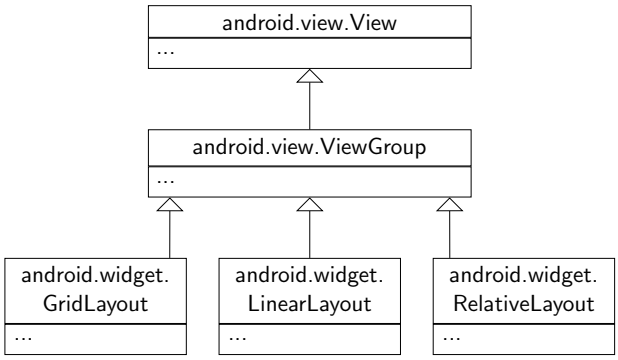
Regular Expression	*Matches*	*Example*
`.`	any single character	`Java` matches `J..a`
`(ab|cd)`	ab or cd	`ten` matches `t(en|im)`
`[abc]`	a, b, or c	`Java` matches `Ja[uvwx]a`
`[^abc]`	any character except a, b, or c	`Java` matched `Ja[^ars]a`
`[a-z]`	a through z	`Java` matches `[A-M]av[a-d]`
`[^a-z]`	any character except a through z	`Java` matches `J]av[^b-d]`
`[a-e[m-p]]`	a through e or m through p	`Java` matches `[A-G[I-M]]av[a-d]`
`[a-e&&[c-p]]`	intersection of a-e with c-p	`Java` matches `[A-P&&[I-M]]av[a-d]`
`\d`	a digit, same as `[0-9]`	`Java2` matches "`Java[\\d]`"
`\D`	a non-digit	`$Java` matches "`[\\D][\\D]ava`"
`\w`	a word character	`Java1` matches "`[\\w]ava[\\w]`"
`\W`	a non-word character	`$Java` matches "`[\\W][\\w]ava`"
`\s`	a whitespace character	"`Java 2`" matches "`Java\\s2`"
`\S`	a non-whitespace character	`Java` matches "`[\\S]ava`"
p`*`	zero or more occurrences of pattern *p*	`aaaabb` matches "`a*bb`" `ababab` matches "`(ab)*`"
p`+`	one or more occurrences of pattern *p*	`a` matches "`a+b*`" `able` matches "`(ab)+.*`"
p`?`	zero or one occurrence of pattern *p*	`Java` matches "`J?Java`" `Java` matches "`J?ava`"
p`{`*n*`}`	exactly n occurrences of pattern *p*	`Java` matches "`J{1}.*`" `Java` does not match "`.{2}`"
p`{`*n*`,}`	at least n occurrences of pattern *p*	`aaaa` matches "`a{1,}`" `a` does not match "`a{2,}`"
p`{`*n*`, `*m*`}`	between n and m occurrences (inclusive)	`aaaa` matches "`a{1,9}`" `abb` does not match "`a{2,9}bb`"

10 Introduction to Android

- Android
  - Android is a platform comprising of three components
    - An operating system
    - A framework for developing applications
    - Devices that run the Android operating system and the applications created for it
  - Android SDK
    - A collection of libraries and tools that are needed for developing Android applications
  - Android Studio
    - IDE for Android application development
- Android App Basics
  - An Android app is a collection of screens, and each screen is comprised of a layout and an activity
    - Layout: describes the appearance of a screen (written in XML)
    - Activity: responsible for managing user interaction with the screen (written in java)
  - Folder structure:
    - Manifest file
    - Resource files
    - Java file
    - Gradle scripts
- The Manifest file
  - It defines the structure and metadata of an application, its components, and its requirements
  - Stored in the root of its project hierarchy as an XML file
- Resources and resource IDs
  - Resources are maintained in sub-directories of the `app/res` directory
    - `res/layout`
    - `res/values`
    - Etc.
  - A resource can be accessed in the code using its resource ID (e.g. `R.layout.activity_main`)
    - Android uses `R.java` to keep track of the resources used within the app
- View
  - Most GUI components are instances of the **View** class or one of its subclasses
    - e.g. Button, EditText, ImageView, etc.



- View Group
  - A special type of view that can contain other views
  - A layout is a type of view group



- Common GUI compnents
  - TextView
  - EditText
  - Button
  - Switch
  - Spinner
  - Toast
- Intents
  - An intent is an object that can be used to bind activities together at runtime
    * If one activity wants to start a second activity, it does it by sending an intent to Android. Android will start the second activity and pass it the intent
  - Data can be passed between activities using intent extras
    * e.g. `intent.putExtra("message", value);`

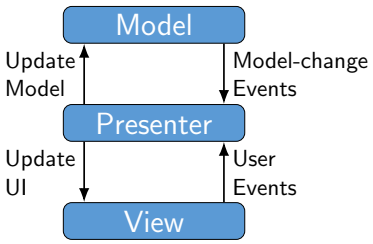
11 Android – Storing Data

- Data storage options
  - File system
  - Shared preferences
  - Databases
    - e.g. SQLite, Firebase Realtime Database
- File system
  - Android's file system consists of six main partitions
    - `/boot` `/recovery` `/cache`
    - `/system` `/data` `/misc`
  - Reading/writing data to a file on internal storage can be done using
    - `openFileInput()` `openFileOutput()`
- Shared preferences
  - Suitable for simple data that could be stored as key/value pairs
  - A **SharedPreferences** object refers to a file containing key/value pairs and provides methods to read and write them
  - Creating/accessing shared preference files can be done using:
    - `getPreferences()` `getSharedPreferences()`
- SQLite
  - Relational database
  - Zero-configuration
  - Widely used
  - Serverless
  - File-based
- Firebase Realtime Database
  - Cloud-hosted
  - Employs data synchronization
    - Every time data changes, all connected clients automatically receive updates
  - NoSQL
    - Data is stored as JSON
  - The Firebase SDK provides many classes and methods to store and sync data. E.g.
    - `DatabaseReference` `ValueEventListener`
    - `DataSnapshot`
- JSON
  - **J**ava**S**cript **O**bject **N**otation
  - Language-independent
  - Supported by many programming languages
  - Uses readable text to represent data in the form of key/value pairs
  - Example

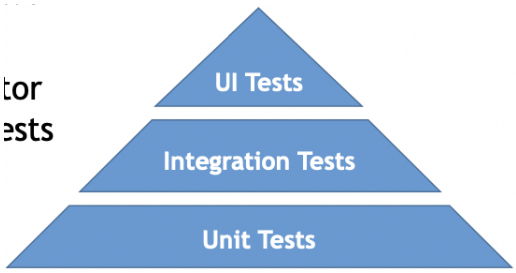
```
{
        "name": "Alex",
        "age": 25,
        "address": {
                "country": "Canada",
                "city": "Toronto"
        }
}
```

12 Android – Testing

- Model-View-Presenter
  - An architectural design pattern that results in code that is easier to test
  - It consists of three components:
    1. Model (Data)
    2. View (UI)
    3. Presenter (Business logic)



- Local and Instrumented Tests
  - Local unit tests
    - Run on the machine's local JVM
    - Do not depend on the Android framework
  - Instrumented tests
    - Run on an actual device or an emulator
    - Usually used for integration and UI tests



- Commonly used tools
  - JUnit
    - Writing unit tests
  - Mockito
    - Creating dummy (mock) objects to facilitate testing a component in isolation
  - Roboelectric
    - Running tests that involve the Android framework without an emulator or a device
  - Espresso
    - Writing UI tests
- Mock Objects
  - A mock is software component that is used to replace the "real" component during testing
  - Mock objects could be used to:
    - Represent components that have not yet been implemented
    - Speed up testing
    - Reduce the cost
    - Avoid unrecoverable actions
    - Etc.
- Mockito
  - A mocking framework for Java
  - Features include:
    - Creating mocks
    - Stubbing
    - Verifying behavior