CPSC 210 Summary Notes

Markus de Medeiros

April 27, 2020

Abstract

This is a summary of the basic content of CPSC 210. All credit is owed to the instructors of the 2019WT2 session of CPSC 210 at the University of British Columbia, or the respective owners. This document does not cover the practical skills of this class (such as debugging or working in intellij) and should not be taken as wholeheartedly comprehensive.

1 Basics

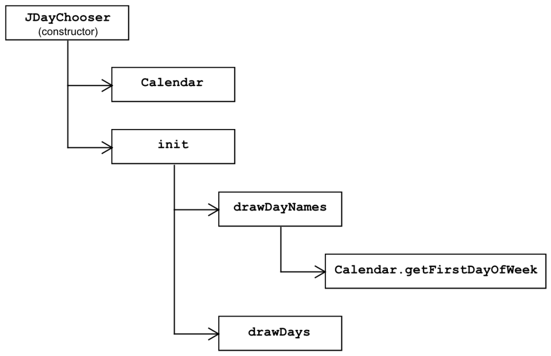
1.1 Program Structure

- Basic Java files are called **classes**.
- Folders, in Java, are called **packages**.
- The runnable class has a **main** method.
- Classes see other classes with **import statements**.

We can represent this diagramatically, but we will probably use UML diagrams instead.

1.2 Methods, Calls, Call Graphs

- A **call graph** shows us how a method is run step by step:



Each method in a call graph must appear only once (recursion becomes loops).

1.3 Classes, Objects, and Variables

Design Principle: Object Orientation

Java is built around **classes** which encode the data and behavior of a single concept. Creating an **instance** of a class gives the instance it's own copy of all of the classes data and methods.

- A **variable** is a symbol, with a **type**, which has a holder for an object of that type.
- We assign a value to a type to out the value into it's holder.
- Primitive types (int, bool, ...) exist in the holder. Objects are different, a variable is given a reference to an object (not the object itself).
- If a non primitive variable a is assigned to another non primitive variable b, then a is defined to be a reference to whatever b refers to at the time of instantiation. If b changes to point to a different object, a will not update to point to that new object (arrows do not chain, they copy).
- An object is **active** if it has been instantiated.
- Within a class, global variables are called **fields**.
- The **static** keyword denotes a method to live outside of instances, and can be called without reference to any particular object.

1.4 Data Flow

- Paramaters of methods are local variables. This means that a primitive paramater will copy and not mutate, but non primitive objects (which are copied as references) will indeed mutate.
- A similar principle applies to **return values**, which also return references to objects.
- A constructor is a function in a class which returns a new object of that type.
- The keyword **this** is used to refer to the instance of the object. It is like a variable in an object with a reference to itself.

1.5 Excecution of a Method

- Methods excecute their lines of code sequentially, and can be changed by control structures.
- We use a **flowchart** to encode the sequence of operations a method preforms. A cheat sheet (taken from edX) is included in the root folder of this document.
- In summary of flowcharts: **code block** is a rectangle, and **if** statement is a diamond (outgoing edges labelled T/F), a **switch** or **else if/else** block are linked diamonds, a **while** (or **do/while**) loop is a diamond linked to itself, a **for each** loop is a loop labelled *collection hasNext()?* and a **for** loop is like a while loop with additional code before the loop check.

2 Abstraction

2.1 Specification and Using a Data Abstraction

Design Principle: Abstraction

Abstraction is the principle of intentionally hiding inessential details of something beihind an interface or container, so that the container can be treated exactly as the object it represents without concern for the details of it's implementation. **Statment abstraction** is applying labels to data or basic methods (like machiene code). **Functional abstraction** is applying a label to a function which may be implemented elsewhere. **Data abstraction** is applying a label to a piece of data which may include methods and fields.

It is helpful for long term reusability and code sustainability to keep the public interface focused on the towards of the object itself. The public interface of an object should be designed to be stable over time.

- The **visible interface** (public in Java) of an abstraction are the things that are designed to be interacted with, the **invisible implementation** (private in Java) are the things that are designed to be hidden.
- It is good practice for essential fields (foe example width or height) to be private, and to create **getter** and **setter** interfaces to change them instead.

Design Principle: Designing an Abstraction

There are four steps to black box design:

1. **Specify** what the abstraction provides.
2. Consider the **use** of the data abstraction.
3. **Test** the abstraction based on these uses (correct and incorrect).
4. **Implement** the abstraction to pass the tests.

- In the specification phase, we write three clauses for our methods. The **requires** clause outlines conditions for correct operation, the **modifies** clause outlines what data the method changes, and the **effects** clause outlines what the method is designed to do.
- The modifies clause must keep in mind that non-primitive objects are passed by reference, so can have side effects on those objects. Additionally, **this** means that the object may mutate itself.
- Empty clauses need not be written.

2.2 Testing a Data Abstraction

Design Principle: Black Box Testing

Black box testing is designing tests strictly on the abstraction level. It serves as a way to encode the specification in code, and to ensure that the abstraction (public interface) is remaining functional as designed over long periods of development.

- Good testing has **branch coverage** (no cases missed) and **boundary checking**.

Design Principle: Testability

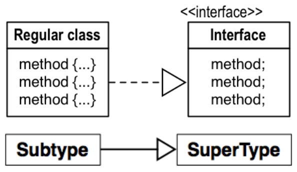
An abstraction is **testable** if it includes methods to determine if a the other methods operated succesfully. An abstraction with poor testability would have methods which require details of the internal implementation to be tested.

2.3 Implementing a Data Abstraction

- Implementatation of an abstraction is the process of writing the code (or stubs) for the interface, running tests, and fixing code for failing tests. If a test suite is comprehensive, any solution satisfying all tests is valid.
- If all the fields are private, this process even allows for the types of fields to be modified and maintain a valid solution. This is helpful for **collections** in Java, which have different types for different purposes which may arise (more on collections below).

2.4 Types and Interfaces

- **Inheritance** is a way to encode similarities between types, and also to avoid code duplication.
- **Interfaces** are special classes which can have a public interface but no implementation (so cannot be instantiated). **Abstract Classes** have a public interface and an an implementation but cannot be instantiated.
- Both of these serve to abstract similarities between types through inheritance: for example if type A and B both have a method c, we can avoid duplication by having A and B **implement** an abstract class or interface C which has details for c.
- If a type A is a special version of another type B with additional or modified behavior, we would use the inheritance A **extends** B. A child class can only extend one parent class, but a parent class can have many children.
- In UML, a class is a named box with a list of important operations (in the public interface) and a tag *abstract* or *interface* if it applies.
- In UML implements is represented by a dotted line with a hollow arrowhead at the supertype, and extends is represented by a solid line with a hollow arrowhead at the supertype.



- Suptypes are always substitutable for a supertype.
- Classes are instantiated with **apparent** and **actual** types. An object can be of any actual type which is a subtype of the apparent type, and must only use the functionality of the apparent type.

```
ApparentType name = new ActualType(paramaters)
```

- Concrete types must have all abstract methods implemented.
- Paramaters of methods check for the apparent type of an object (at compile time). Thus, methods can take several types (a concept called **subtype polymorphism**).

2.5 Multiple Interfaces

- A class can extend multiple interfaces. If two interfaces have methods with the same name (and paramaters) there can only be one implementation for both. This is a potential design issue, if two methods with the same name in different interfaces have different intended behaviors.
- In some implementations the apparent type of an object can be **casted** to the actual type if substitution is violated, but in general this is a symptom of poor type hierarchy design.

2.6 Extends and Overriding

- Extended subclasses inherit public and protected fields and methods of their parent classes.
- **Method Dispatch** is the process of searching for an implementation for a mehthod. If the actual type of an object does not have an implementation, Java seaches up the type hierarchy for an implementation.
- A subclass can implement a supertype method and it will **override** the implementation during method dispatch.
- In a subclass the **super** keyboard accesses supertype methods. This is particularly helpful for subclasses which augment superclass behavior, using methods of the same name.

2.7 Abstract classes and Overloading Methods

- The **abstract** keyword before methods in abstract classes allow a method with no default implementation but that must be implemented in concrete subclasses.
- A method is **overloaded** if there are two methods with the same name but different paramater lists in scope. Java will view these as seperate methods, but it is often helpful to have overloaded methods for allow some flexibility in paramater types.
- Abstract classes must be tested with abstract or overridden test classes.
- Fields should not be redeclared in subclasses.

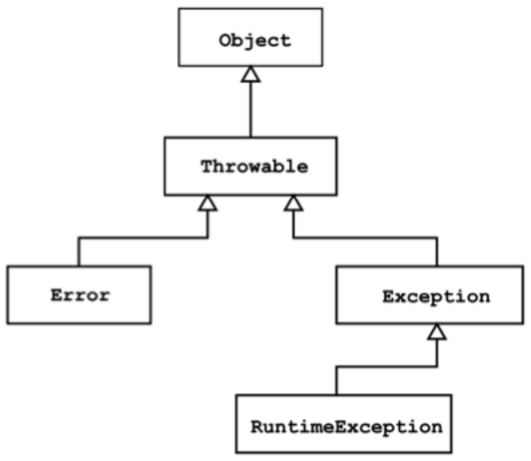
3 Construction

3.1 Throwing and Catching Exceptions

Design Principle: Robust Code

Code is **robust** if it preforms well over a large domain of inputs. In Java, Robust code will have small *requires* clauses, and will be able to handle unexpected errors that occur even when the *requires* clause is met.

- In Java, **exceptions** are objects which represent unhandled conditions in code.
- When a functions **throws** an exception, it passes it the exception up the call stack.
- We can write new exceptions by extending Exception classes



and throw a new exception with

```
throw new ExceptionType(paramaters)
```

It can be helpful to override the constructor with a string describing the error.

- Exceptions that may throw an exception of type E without catching it are denoted with the **throws** E keyword in the method signature.
- Exceptions can be caught with **try/catch** blocks. Every **catch** block catches exception by exception type.. A **finally** clause can be used to run code regardless of if an exception is thrown, but before the exception is passed up the stack trace.
- It is possible, and sometimes helpful, for catch blocks to throw new exceptions themselves. However each try/catch block will only catch one exception, even if that exception throws a new exception which is catchable below in the block.
- If main throws or does not catch an exception, the program will crash.

3.2 Exception Hierarchy/Unchecked Exceptions/Assertions

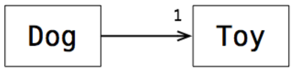
- Duplicate catch blocks can be abstracted away by having exceptions extend other exceptions. By substitution, Java catch blocks will also catch all subtypes of exceptions. It also follows that the order of catch blocks is important.
- **Runtime (Unchecked) Exceptions** do not need to be caught. An unchecked exception can be caught with a try/catch block, and if an unchecked exception is thrown in a try/catch/finally block the finally clause will be run and the exception will be passed up the call stack.
- Runtime exceptions are used for errors that rarely, but sometimes do, happen.
- A **class invariant** is a statement (written at the top of the class) which must be maintained by all functions.
- To ensure class invariants are maintained, we use **assert** statements to check class invariants at points in our code, and these statements can be turned off by the compiler so they do not slow down the final program.

3.3 Testing Exceptions

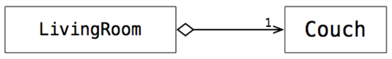
- To test if exceptions are thrown, we use try/catch blocks in our test classes. If exceptions should be thrown but aren't, or if an exception is thrown but isn't, the **fail(message)** method can be used to indicate an incorrectly exception.

3.4 Extracting Hierarchy and Associations

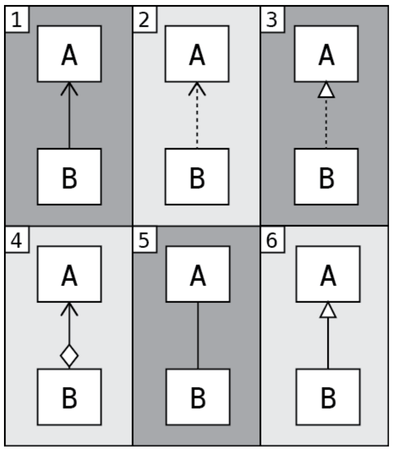
- We represent object oriented design in three ways: with a **class hierarchy**, **associations**, and **sequence diagrams**.
- We have seen class hierarchies in section 2.4, these arrows can be consistently added to association diagrams.
- **Association diagrams** describe the fields each class has.
- Every field is represented by an arrow from the container class to the type class with an open arrowhead labelled by the cardinality (natural number, $0..n$ or $1..n$ or $0..*$ for lists of variable size) of that field.



- An **aggregation relationship** represents situations when the field type is a part of or belong to the class, and are represented with a diamond on the tail of the arrow.



- If it is impossible to draw a dual arrowhead (such as when arrow tails are diamonds), the arrowheads are left off.
- The **depends upon** association, represented by a simple dashed arrow, represents when a class uses constants or has local variables of a given type but does not have that type as a field.
- In summary of the arrow types,



1. *B* is associated with *A*.
2. *B* depends on *A*.
3. *B* implements *A*.
4. *B* is an aggregation of *A*.
5. *B* and *A* have a bidirectional association.
6. *B* extends *A*.

3.5 Extracting Sequence Diagrams

- A **sequence diagram** represents method excecution.
- Each active object has a vertical **lifeline** labelled with the object name and type (superscript) with time increasing lower on the line.
- Calls to methods are represented by horizontal lines labelled with the method name with paramaters. A box on a lifeline represents a method on the call stack, in this way arrows create and end boxes. There is an implicit lifeline leftwards of the leftmost arrow representing time, and calls from or returns to the rest of the program.
- Calls to the same object (**self calls**) should create another box to the right of the object's box with a little round arrows going to and from it.
- Loops are represented by large rectangles enclosing loop behavior. The loop

```
for (X x : listofx) {method(x);}
```

  will be represented by a large rectangle labelled *loop listofx* in the top left corner, followed by the implicit *getNext* call to listofx which returns x, and then the sequence diagram for *method* using a lifeline for x.
- Similarly, the diagram for

```
if (statement) {method();}
```

  would be a rectangle labeled *if (statement)* and enclosing the sequence diagram for *method()*.
- Other methods for loops use labels like *loop [until listofx.hasNext() is false]* which is more similar to *while*-type loops. Abbreviations are often used in these diagrams as well, the purpose of these diagrams is clarity first and foremost.

3.6 Implementing a Hierarchy, Simple Fields and Optional Sequences

- When we implement associations as fields we may need to determine how exactly the associations are represented. For fields with cardinality > 1, we must decide to implement them individually (only possible if the cardinality is fixed) or as a collection.
- When implementing collections, the **Java Collections Framework** provides several options depeding on the collection's use case:
  - **List Interface**
    Positional and allowing duplicates
    - **ArrayList**: Items stored in an array
      - Fast: append/get(1)
      - Slow: insert/remove/contains/add or remove using iterator (n)
    - **LinkedList**: Items not stored in an array
      - Fast: append/add or remove using iterator (1)
      - Slow: insert/get/remove/contains (n)
  - **Set Interface**
    Non-positional and disallowing duplicates
    - **HashSet**: Items are not ordered (iterator returns items in no order)
      - Fast: add/remove/contains (1)
    - **TreeSet**: Items are ordered (iterator returns items in order)
      - Slow(-er than HashSet): add/remove/contains (log(n))

3.7 Overriding Equals and Hashcode

- A **map** is a structure of key-value pairs. Pairs are inserted using the **put** method, values with the **get(key)** method, and a collection of keys with the **keyset** among others.
- A concrete subtype of map is the **HashMap**.
- HashMap's methods, among other Java operations determine equality with the **.equals** method which by default is the same as the == method. The == method compares if two objects are literally the same thing, which is not what we want to compare in general. Thus we must **override the equals method** in our objects to specify how exactly we would like our objects compared.
- When we override the equals method, we must also override the HashCode method.

3.8 Implementing Bi-Directional Relationships and Sequence Diagrams

- Consider a bi-directional association of arbitrary cardinality between A and B. Every implementation of this has a similar form:
  1. A must have a collection of B as a field, and B must have a collection of A as a field.
  2. In the class A, the relationship looks like

```
class A {
    Collection<B> listOfB;
    public addB(B b){
        if (!listOfB.contains(b)) {
            listOfB.add(b);
            b.addA(this);
        }
    }
}
```

and vice-versa in B. The order is important to avoid infinite recursion.

- A bi-directional association of fixed cardinality would be similar, replacing list addition with setting the field value and the guard with field equality testing. Again, changing the order will give a recursion.
- Implementing sequence diagrams is also fairly straightforward, the method is essentially encoded in the arrow labels and control flow rectangles.

4 Design

4.1 Cohesion and Coupling

Design Principle: The Single Responsibility Principle

Each abstraction (class or method) should have a single purpose or represent a single concept. A class contain the minimal amount of information to complete this purpose effectively.

- If an abstraction has several purposes, it is lacking in **cohesion**. To improve the cohesion of the class, it's methods should be seperated into several classes of a single purpose and with only the data needed for those purposes.
- **Coupling** refers to how indimately two abstractions are related. Coupling is acceptable if it only makes reference to the public interface of a module. Poor coupling is associations between implementations, or creating an excessive public interface as an afterthought of the implementation and is not essential to the purpose of the class.
- Code duplication is another example of poor coupling.
- In general, errors in well coupled code will be detected by tests and the compiler while errors in poorly coupled code will not.
- One way to improve coupling is by applying concepts from functional programming. References to object attributes can sometimes be passed as functional paramaters, which can serve to remove unwarranted associations. Other ways to improve coupling could involve abstracting similar behaviors into a type hierarchy with a class representing the similarity in function between two modules.

4.2 Liskov Substitution Principle

Design Principle: The Liskov Substitution Principle (LSP)

A sub-abstraction can be substituted for it's super-abstraction if it does not break the expectations set by it's super-abstractions. Unsubstitutible behaviors include

- Narrowing of domain (strengthening of preconditions)
- Widening of range (broadening of postconditions)

- In Java the LSP can be violated if the REQUIRES clause of the subclass methods do not include some elements of the superclass, or if the EFFECTS clause of the subclass describes broader behavior than the superclass.
- We can fix violations of the LSP by abstracting the essential behavior as an interface or abstract class, or alternatively by changing an *extends* relationship to weaker *uses* association. Either one of these allows us to relieve some expectations placed on a particular class and satsify the LSP.
- Some tests which can help to determine if a substitution violates the LSP:
  - **Substitution test:** Can the subtype be substituted as the supertype in Java's type systemm?
  - **Concatentation test:** Does the subtype make sense as a noun preceding the supertype as an adjective?

  - **IS-A test:** Is the subtype a special version of the supertype? This intuition based test is the most comprehensive.

If the tests fail, it may be more prudent to represent this as a *knows about* or *has* type of association.

4.3 Refactoring

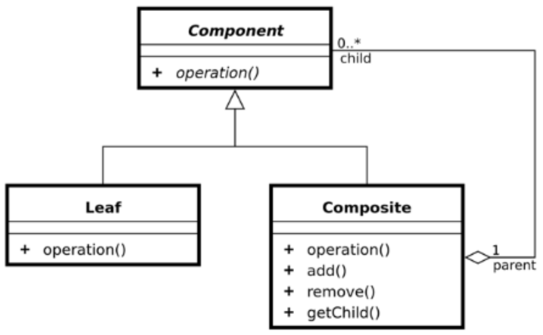
- **Refactoring** is the process of making structural changes (not behavioral changes) to make our code more principled.
- Refactoring is best done starting with exhaustive, passing tests. With a sufficient set of tests, we have complete domain to alter anything in our implementation without changing anything about the public interface.
- It is also to refactor methods for readability, as more code is written the purpose of old code will be less obvious and potentially even less relevant.
- It is important in larger scale projects, especially those with multiple contributors, that the public interface of abstractions be maintained and stable. Under our design principles, it is more important for sustainability to have well defined public interfaces at the outset than well designed private implementations because the former becomes hard or infeasible to change.

4.4 Composite Pattern/Singleton Pattern

Design Principle: Design Patterns

Design patterns are broadly applicible and highly principled solutions to common problems in computer programming. Learning design patterns is a straightforward way to making principled solutions easier or effortless to write.

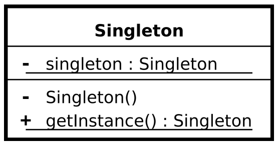
- The **Composite Pattern** is an object oriented solution to hierarchy.
- Elements of a hierarchy are represented as **composites** (nodes with children), **leaves** (nodes without children), and both are subtypes of a **component** which encodes the behavior of existing in the hierarchy.



- To write a composite pattern:
  1. Determine the composite (the object which contains other objects)
  2. Detmine the leaves (the object that does not contain other objects)
  3. Write them to extend component, a abstract class or interface with a temlate for the operation(s)

4. Equip the composite with a list of components and methods thereof
5. Implement the operation(s) recursively over the list of components

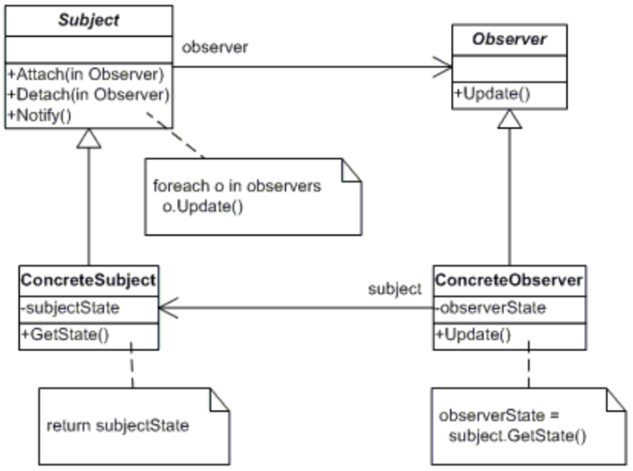
- A related design pattern is the **singleton**, which ensures a class has only one instanceand to provide it with a global reference.



- A singleton is achieved by equiping a class with a static, final reference to a singleton, a private constuctor and a static method accessing this reference. The only way to create a singleton is by the accessing method, and the accessing method gives a reference to exactly one singleton. One could also program behavior on the singleton creation (creation used pretty loosely here) or allow a finite number of singletons by removing the finality of the singleton and having more complicated behavior in the access method. The key to the singleton pattern is the private constructor so object creation is controlled.

4.5 Observer Pattern

- The **Observer Pattern** is an object oriented solution for one object watching another.



- The observer pattern takes place as follows:
  - ConcreteSubjects are set up to know about Observers (an *addObserver* method)
  - When the ConcreteSubject does somehting of note it makes a call to *notifyObservers*, which instructs all known observers to make a call to an abstract *update* method accordingly.
- In this model the subject has the responsibility to **push** the information to the observers. Alternatively, another model would push a notification to observers that a change has occured and that the observers should **pull** information from some public *getSubjectState* method instead.

4.6 Java's Observer

- Java implements an Observer by default. The Subject is replaced by an **Observable** class and the Observer is replaced by an **Observer** interface. Barring these changes, it has the same notation as the figure in subsection 4.5.

4.7 Basic Iterator

- The **Iterator Pattern** is an object oriented solution to iterating over a collection of sub-objects.
- In Java, **Iterator**$\langle E\rangle$ is an interface (with notable functions *hasNext()* and E *next()*) which represents something which can iterate over all of it's sub-objects of type E.
- In Java Collections, Colleciton implements Iterable which represents something which can has an Iterator. There are Iterator subtypes for all of the subtypes of Collection such as a ArrayListIterator or a TreeSetIterator, which are ensured to be there by implementing tht Iterable interface.
- Similarly if we would like to implement Iterable in a class A over a sub-object B (that is, have A implement Iterable$\langle B\rangle$ we must implement the *iterator()* method. In a simple case were we are already using a collection c, we can just return a premade iterator associated with that collection:

```
@Override
public Iterator<B> iterator() {
    return c.iterator();
}
```

- Iterable classes can use the for/each loop syntax.

4.8 Advanced Iterator

- If the class is not using a collection, to implement iterable you must create your own custom iterator.
- An instance of iterator is best implemented as an inner class (a class within another class which can see private data). This instance of iterator needs at a minimum to implement *hasNext()* and *next()*.
- One common situation for creating a custom iterator is when two collections need to be iterated over. It is possible to combine two default iterators in a custom iterator inner class.