COMP2511 OOP and Code Smells

Alperen Onur

November 2024

1 Inheritance

Rather than defining a whole new class, we can have a class extend an existing class. The subclass inherits the existing attributes and behaviors of the superclass. Inheritance is an "is-a" relationship. You can often times recognize whether to use inheritance if you notice a class having an "is-a" relationship with another. The converse is a "has-a" relationship where you create an instance of a class in another class.

Abstract classes are a way support a particular type of inheritance. These classes include abstract methods that the inheriting class must implement. The abstract class can not be instantiated but it's children can be. Though it is not supported in Java, some other languages support inheritance from multiple classes which can create issues since a behavior can be inherited in multiple different ways. Single inheritance is generally the best option in these cases.

Interfaces also support inheritance but all their methods are implicitly abstract. Variables can be declared in an interface but must be static and final (ie constants). A class can implement more than one interface.

2 Method Forwarding

There may be scenarios where you want to use functionality and methods that exist from another class but can't extend it or can only extend it's parent. In this case, we can create the object of functionality we desire in our current class and use it's methods freely. This is called method forwarding.

3 Polymorphism

Polymorphism is changing the behavior of a method at some point in an inheritance tree by defining a method with the same signature and return type as the super class. For this class, it'll use this new definition of the method rather than it's parents (essentially overriding the method). The object decides which method to apply to its self and this is the idea behind polymorphism. Method overloading is a similar concept however is applied when an object has a method of same name as its super class with different arguments.

4 Visibility Modifiers

- Public: visible to all classes.
- Private: visible to only to the current class.
- Protected: visible to the package and all subclasses.
- Default: Applied when no modifier. Visible to the package.

5 Domain Modeling

There are several different types of relationships we can define to show how classes interact.

- Dependency: A loose form of relationship. A class depends on some other class in some way.
- Association: A class uses another class but it's not clear the direction in which the dependency occurs. Both classes could use each other.
- Directed association: Association but a clear direction.
- Aggregation: A class contains another. The diamond end indicates the contained class.
- Composition: Like aggregation but the containing class MUST contain the contained class.

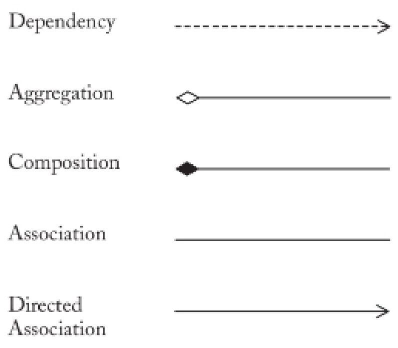


Figure 1: Relationships arrows

6 Design by Contract

Design by contract is a contrast to Defensive programming; address all edge cases of a programs input and potential behavior to ensure its continual function. An example of defensive programming would be handling a case where your program is supposed to read 3 inputs from STDIN but instead reads in 4. An error can be thrown in this case to ensure your program works correctly. Design by Contract says that we can assign clear responsibilities to software elements such that we can exclude these redundant checks and make the code simpler/easier to maintain.

To achieve a design by contract we'll need a Pre-condition, Post-condition and Invariant.

- Pre-Condition: What does the contract expect? The pre-condition must always be true prior to the section of code that is entered. If this isn't the case, the behavior becomes undefined and the intended work may not be carried out. Pre-conditions can be weakened in a subclass definition but never strengthened (as it would violate LSP).
- Post-Condition: What does the contract guarantee? A post condition must always be true after the execution of some section of code. It's an essential guarantee that if the pre-conditions are satisfied, the post-condition behavior will always be as expected. Post-conditions may be strengthened in an inherited class but never weakened.
- Invariant: What does the contract maintain? The invariant of some code constrains the state of variables stored in an object. Invariants are established during construction and maintained between calls of public methods. Each method of a class must make sure they maintain the invariant.

7 Exceptions

Exceptions describe a disruptive event in a section of code that disturbs the normal flow of a programs runtime. In OOP, there are checked an unchecked exceptions. Checked exceptions do not need to be caught whereas unchecked exceptions do.

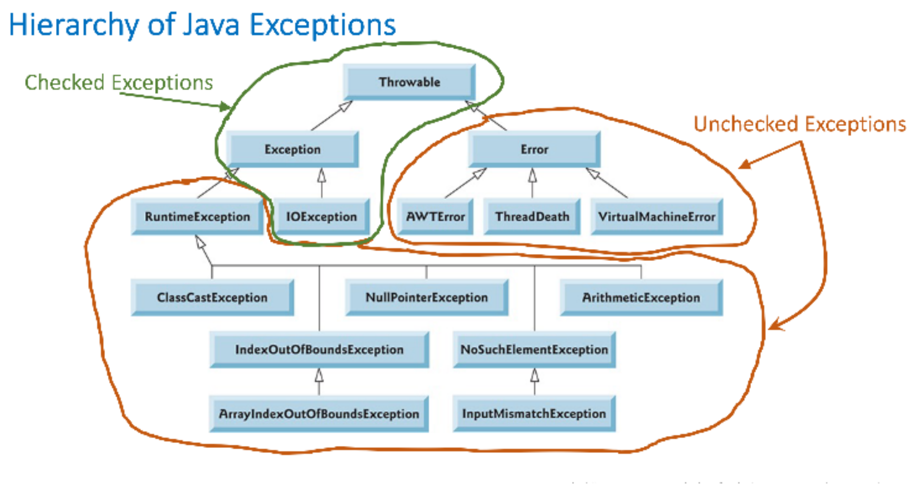
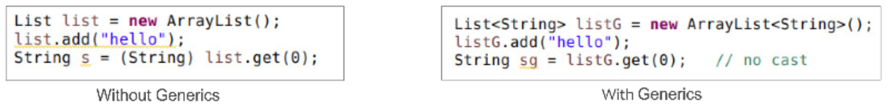


Figure 2: Exceptions Hierarchy

8 Generics

Generics provide a way for classes to be parameters when defining other classes, interfaces and method. This approach removed type casting and offers stronger checks at compile time.



```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

Without Generics

```
List<String> listG = new ArrayList<String>();
listG.add("hello");
String sg = listG.get(0);    // no cast
```

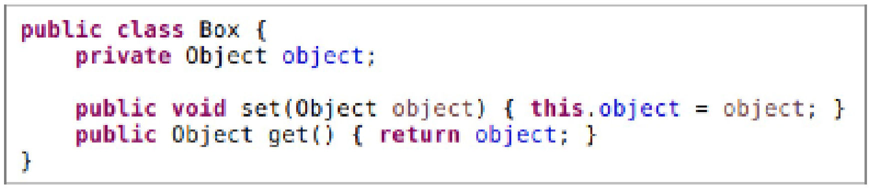
With Generics

Figure 3: Generics example

You can define generic type classes which is a generic class or interface that is parametrized over types (rather than specific types). There are several choices to parameter names:

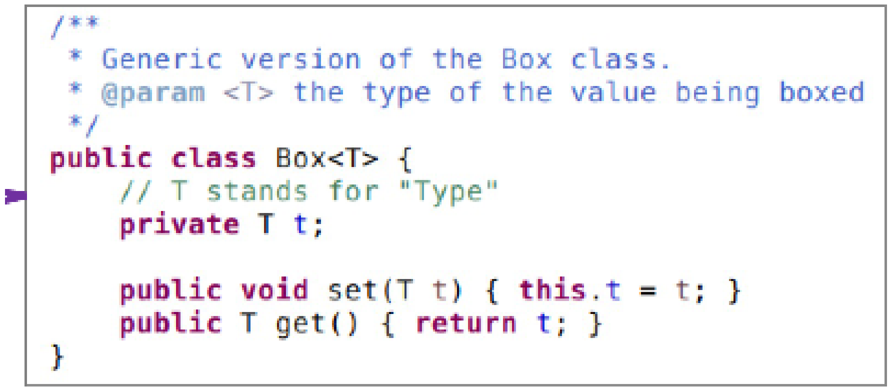
- E - Elements
- K - Key

- N - Number
- T - Type
- V - Value



```java
public class Box {
    private Object object;

    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}
```



```java
/**
 * Generic version of the Box class.
 * @param <T> the type of the value being boxed
 */
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

Figure 4: Generic type

9 Collection Interface

A collection is simply some group of objects with each object being an element. A collection interface is used to pass around collections when generality is desired. These interfaces can contains methods such as .isEmpty() or .contains(Object element) such that these basic operations can be performed on the collection.

10 Design Smells

A design smell is a symptom of poor design; key design principles have been violated. This usually leads to refactoring when changes need to be made or features added which is slowed down because of the poor code quality. Loose coupling (little interdependence) and high cohesion (the degree of which classes work together) of a design generally is what indicates good design. There are several design smells:

- Rigidity: Software being too difficult to change even in simple ways. Often an issue when a single change causes a cascade of changes to other classes.
- Fragility: Tendency for software to break in many places when a single change is made.
- Immobility: Design is hard to reuse. The effort required to export it to other systems requires too much effort.
- Viscosity: Changes are easier to implement through shortcuts rather than preserving the current design. This is usually an indicator when development is slower than it should be.
- Opacity: Tendency of a module being difficult to understand.
- Needless complexity: Contains features that are currently not useful or ahead of requirement.
- Needless repetition: Design contains structures that could be unified under a single abstraction.

11 SOLID

SOLID is just one design principal to achieve good design.

- Single responsibility principle: Classes should be responsible for a single function.
- Open closed principle: Classes should be open for extension but closed for modification.
- Liskov substitution principle: Classes in a program should be replaceable with instances of their superclass without altering the correctness of that program.
- Interface segregation principle: Many specific client interfaces are better than one general purpose interface.
- Dependency inversion principle: Depend upon abstractions and not concretions.

12 Principle of least knowledge - Law of Demeter

The Law of Demeter says that classes should interact and know about as few classes as possible. This helps us loosely couple our software so that changes in one area to not cascade to other areas. A method in an object should only invoke methods of:

- The object itself
- The object passed in as a parameter
- Objects instantiated within the method
- Any component objects
- And not those of objects returned by the method.

A violation of Law of Demeter is often indicated by long chaining of methods.

13 Liskov Substitution Principle

All subclasses must be substitutable for their base types. Sometimes LSP is hard to abide by, this usually indicates that inheritance is likely not the correct option for this class. You can consider delegating the classes function to another class or use composition instead.

14 Code Smells

Code smells are inconvenient bits of your code that can make readability worse or the software difficult to work with/understand. They often violate fundamental design principles and impact code quality. Common indicators of code smells are:

- Duplicated code
- Long method
- Large class
- Long parameter list
- Divergent change: When one class is commonly changed throughout development.
- Shotgun surgery: When lots of small changes need to be changed in different classes during development.
- Poor abstraction

- Classes have little cohesion
- Classes have too much coupling
- A subclass doesn't use a majority of inherited functionality
- "Data classes"
- Design is too general
- Design is too specific

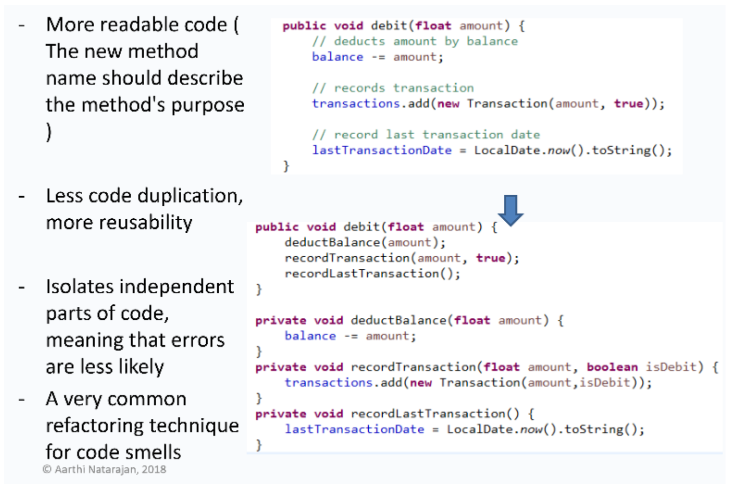
There are specific types of code smells that we'll cover:

- Bloaters: code in methods and classes that have grown in size and become too difficult to work with (long method, large class, long parameter list, data clumps).
- OO Abusers: Incorrect application of OO principles (switch statements, refused bequest).
- Change preventers: Code changes are difficult (divergent change, shotgun surgery).
- Dispensables: Code that is pointless and unnecessary (data classes, comments, lazy class, duplicate code)/
- Couplers: Excessive coupling between classes (feature envy, inappropriate intimacy, message chains)

15 Refactoring

Refactoring techniques help us remove code smells and sometimes design smells without changing its existing behavior. There are several refactoring techniques:

- Extract method: reduce the length of a method body. Aims to make code more readable and reduce code duplication. Isolating independent parts of code such that errors are less likely.



- More readable code (The new method name should describe the method's purpose)
- Less code duplication, more reusability
- Isolates independent parts of code, meaning that errors are less likely
- A very common refactoring technique for code smells

```
public void debit(float amount) {
    // deducts amount by balance
    balance -= amount;

    // records transaction
    transactions.add(new Transaction(amount, true));

    // record last transaction date
    lastTransactionDate = LocalDate.now().toString();
}
```

```
public void debit(float amount) {
    deductBalance(amount);
    recordTransaction(amount, true);
    recordLastTransaction();
}

private void deductBalance(float amount) {
    balance -= amount;
}
private void recordTransaction(float amount, boolean isDebit) {
    transactions.add(new Transaction(amount,isDebit));
}
private void recordLastTransaction() {
    lastTransactionDate = LocalDate.now().toString();
}
```

© Aarthi Natarajan, 2018

Figure 5: Extract method

- Introduce parameter object: Consolidate parameters into a separate class and pass that class into the method. This can remove smells such as long parameter list, data clumps, primitive obsession and long methods.

- Replace temp with query: Often, we place the result of an expression an a local variable for later use in the code. We can fix this by moving the entire expression to a separate method and return the result from it. Now we can just query this method instead of using a variable. This eliminates smells such as long methods and duplicate code.

- Extract class: Having all details in one class is bad OO design and breaks the single responsibility principle. We can fix this be separating the classes into their appropriate responsibility.

15.1 Code Smell: Long Method

Use the extract method to reduce the length of the method body. If local variables and parameters interfere with the extraction, use replace temp with query, introduce a parameter object or preserve whole object. A last resort is to move the entire method to a separate object.

15.2 Code Smell: Large Class

Violates single responsibility principle usually. Try applying the extract class or extract subclass techniques.

15.3 Code Smell: Long Parameter List

The method has too many parameters to remember which is bad for readability and maintenance. An easy solution is the place a query call inside the method by encapsulating the parameters into an object and giving that to the method.

15.4 Code Smell: Data Clumps

Different parts of your code containing identical groups of variables can be a sign of a data clump. A solution is to move the behavior to another class via the move method. If repeating data comprises the fields of classes you can instead extract this to be in their own class. If they are passed into parameters of methods you may introduce a parameter object.

15.5 Code Smell: Refused Bequest

A subclass only uses some of the methods and properties that it has inherited from the superclass. The unused methods may go unused or undefined. A technique to refactor here could be to replace inheritance with delegation instead. If inheritance is insisted though, try to push down methods and fields from the superclass to the subclass.

15.6 Code Smell: Duplicate Code

When code segments look too similar, there are many ways you could refactor. Extracting methods into a superclass might make sense but if the code is not the exact same you could use a Template Method or Substitute algorithm technique (select the best implementation of the code and use that).

15.7 Code Smell: Feature Envy

When a class is more interested in a class other than it self whether that be constant calling of its methods or instantiation, this is a sign of feature envy. To refactor this, move the method to the relevant class that needs it. If only part of the method accesses another class, you can use the extract method to move that specific part to the relevant class.

15.8 Code Smell: Divergent Change

Implementation of a feature causing one class to change in several ways is a sign of divergent change. Identify everything about the class that changes and apply the extract method to move these into their own class.

15.9 Code Smell: Shotgun Surgery

Trying to implement a small change that causes minor changes everywhere else is an example of shotgun surgery. Move those methods and fields into a single class.

15.10 Code Smell: Data Classes

Classes with simple getters and setters and no behavior beyond that. Move behavior related to the data to the data class.

15.11 Code Smell: Lazy classes

A class that doesn't do much work to justify their existence causing unneeded maintenance. Move the data from the lazy class to another class and remove the lazy class.

15.12 Code Smell: Switch Statements

From an OOP point of view, switch statements are bad design. Replace the switch statements with a polymorphic solution based on a strategy pattern.