Menti Q&A QR

Advanced OOP for Software Engineering

Blair Archibald

Goal

- Link OOP to Coupling/Cohesion
- Learn Advanced OOP Features that *control* component use
  - E.g. to improve coupling

Recap

- Challenges of SE: Scale + Change

Recap

- Challenges of SE: Scale + Change
- Handling scale: Abstract into components
  - Care about the “what” not the “how”

Recap

- Challenges of SE: Scale + Change
- Handling scale: Abstract into components
  - Care about the "what" not the "how"
- Handling Change: Make sure these components are well designed
  - Look at *relationships*
    - Within components: **Cohesion**
    - Between components: **Coupling**

Some Questions from The Labs

Why do we need to remove coupling?

- Too much coupling can indicate *poor* design
  - Sharing too much information
  - Tricky to use interactions (e.g. routine coupling)
- Essentially: code too much coupling (not enough cohesion) is **difficult to change**
  - Difficult meaning "propagates through large sections of your program"
  - Not a big issue in the smaller lab examples
  - *But* A massive issue if you have a 1m line code-base
- I'll try and show some examples in this lecture

Object Oriented Programming

- Objects let us build systems from highly cohesive components
- Objects: Data + Methods that act on that data
  - And some ways to extend/manipulate/subtype objects
- Philosophically:
  - Objects are encapsulated domain entities
  - Computation is *interaction* between domain entities

Remember: Coupling and Cohesion are also about interaction/relationships

Internal Object Cohesion

Objects are communication cohesive

Communication Cohesion: components operating on the *same data* are kept together

- The data are instance variables
- "Components" (here) are methods

Remember

- Objects aren't cohesive just because they are objects

Remember

有疑问

- Objects aren't cohesive just because they are objects
- You have to make sure there isn't:
  - Redundant data
  - Methods that don't need to know about particular data

Example: Redundant Data

Probably not a cohesive class:

```
public class Everything {
  private DataBase db;
  private WebsiteServer server;
  private BankAccount acc1, acc2l;
  private HTMLFormatter fmt;
  ...
}
```

Example: Methods

```
public class User {
  private DataBase db;
  ...

  public String getName();
  public void validateUser();
  ...

  // Assume only method using db
  public void syncDB();
}
```

- Hints that db should not be part of User
  - Pass to syncDB
  - Have a UserDatbaseController or similar class

Internal Coupling: Common Data

- Technically, methods of a class have *Common Data Coupling*
  - They access the same member variables (even if private)
- Low impact due to the high communication cohesion
  - Only need to change *internal* methods on update
  - Global variables need to change all class/method using them!

External Coupling: Interface

- (public) Methods give a well-defined interface
  - Return type, name, parameters
  - Implementation is hidden
- Can be difficult to change an API later once people are using it
  - Be careful what methods you expose
  - Really easy to change *internal* implementations though!

Other Forms of Choesion in OOP

- *Packages* collect classes that work together
  - Sometimes called in languages like C++
- General Purpose Packages as Layer Cohesion
  - e.g. Many classes will utilise (or inherit from) Collections/Swing GUI etc
- Methods might be ordered (temporal cohesion)
- Some scope for utility cohesion: Java.Math

OOP Features To Manage Coupling

- OOP is just data + methods
- Many "additional" OOP features are about *controlling* access
  - Visibility modifiers for functions/variables
  - Inheritance
  - Abstract classes/interfaces
  - Polymorphism
  - `final` keyword

Visibility Modifiers

- Explicitly mark access:
  - **private:** only this class can access
    - No external coupling allowed
  - **public:** anyone can access
    - Coupling allowed (APIs public since you need to couple to them!)
  - **protected** this class and children can access
    - Controlled coupling between a (hopefully!) cohesive set of *similar* objects
- In Java, no modifier $\implies$ package protected
  - Controlled coupling between (hopefully!) cohesive set of objects

Visibility Modifiers

Reduce coupling by locking things down as much as possible

- Content Coupling is avoided with *private variables*
- One of the easiest ways to stop a code-user doing something stupid
  - e.g. randomly updating a bank account balance
  - Lots of public data? Be suspicious
- There are very few times that public is what you want

Getters/Setters: Fundamental for Access Control

- Let's look at an example of Getters/Setters
- *Why*:
  - Tells us info about the class: *mutable/immutability of data*
  - **Getters/Setters** can control type conversions if needed
  - Additional functionality, e.g. access checks, can be added

Inheritance

- "is a" relationship between entities
  - `FordFocus` is a `Car`
    - Can do everything a car can (e.g. move method)
    - Maybe it has an extra `getFordSerialNumber()` method
  - `Html` is an `OutputFormatter`
  - `ArrayList` is a `List`
- **Should be** cohesive
  - Objects are so closely related it makes sense to share behaviour/data

Inheritance

- Introduces coupling between parent and any child classes
  - Any changes in parent are reflected in child
  - *Common Data/Common Methods*
- Can still control this via access modifiers
  - E.g. Don't let a child method affect parent data
- Be aware of this when you go to extend a class

Super Methods

- Sometimes we want to do the parent method and then a bit more
- Can't (and shouldn't!) copy the implementation
  - Copying implementations is bad for cohesion
  - Won't work anyway if data might is private
- "super" methods call a (direct) parent implementation

Super Methods in Constructors

- Super is often used in constructors
  - Ensures everything is initialised correctly
  - Can't force this to be called!
    - Need to be defensive

```
public LabelledPoint(int x, int y) {
  super(x,y)
  label = "";
}
```

Example of super in verification methods

- Parent knows how to verify it's data
- We need to add verification for new data

```
class Point {
  private int x, y;
  ...

  public boolean isValid() {
    return x >= 0 && y >= 0;
  }
}
```

```
class LabelledPoint extends Point{
  private String lbl;
  ...

  public boolean isValid() {
    return super.isValid() && l.length() >= 0;
  }
}
```

Inheritance Examples

Lets look at some examples

Abstract Classes

- You can *force* inheritance through *abstract classes*
  - E.g. when the class is conceptual only
- `abstract` keyword means:
  - `abstract class`: class must be inherited from before being constructed
  - `abstract method`: method must be implemented by subclasses

Abstract Classes

Doesn't make sense to have "places" as a concept, only specific places

```
abstract class Place {
  protected String name;
  public void toString() { return "Place name" + name; }
}
```

```
Place p = new Place();
```

```
BadPlace.java:8: error: Place is abstract; cannot be instantiated
        Place p = new Place()
```

Abstract Classes

To use we make specific (concrete) instances

```
// Note: Not abstract
class School extends Place {
  public School() {
    name = "School";
  }
}
```

```
School s = new School();
System.out.println(s.toString());
```

```
Place name: School
```

Example

- Forcing a name to be chosen for a place

Interfaces

- **Interfaces** are abstract classes with
  - no data
  - all methods marked abstract
- They define the API a child class must implement

```
public interface OutputFormatter {
  public void writeParagraph(String s);
}
```

Interfaces

- A class can implement multiple interfaces[1]

```
// We use implements, not extends, for interfaces
public class HTMLFormatter implements OutputFormatter {
  public void writeParagraph(String s) { ... }
}
```

[1] In Java you can't extend multiple *classes* (but can in other languages)

Polymorphism

- Object behaviour depends on the specific type
  - But we aren't allowed to know the specific type!
- Key way to reduce coupling:
  - Code against a super-class (often an *interface*)
  - Specific implementation is unknown
    - Which means we can *change it if needed*
- Can sometimes replace control coupling
  - No longer need an `if(someFlag)`
  - Pass an object that does the "flagged" behaviour

Polymorphism: Longer Example

```
// If functions were shared then abstract class would be better!
public interface Car { public String details(); }

public class FordFocus implements Car {
  public String details() { return "Ford Focus"; }}

public class HondaCivic implements Car {
  public String details() { return "Honda Civic"; }}

// Does the right thing, but doesn't rely on the type of Car
// We have *decoupled* the implementation using polymorphism!
public void printCatDetails(Car c) {
  System.out.println(c.details());
}
```

Polymorphism

Interact with the most general type you can

```
// Usually Bad: Leaks implementation details
public ArrayList<Integer> f(...) {}

// Much better: Just says what sort of thing it is
// (some form of ordered collection)
public List<Integer> f(...) {}
```

Too Many Interfaces?

- Some developers take this to the extreme
  - Define interfaces for *almost everything*
  - Just in case it changes in future
  - Even for things that currently have only one implementation
- Seems like a lot of extra typing/files/complexity
  - Is it worth it?
  - Who knows!

The Important Idea

- You need to always be thinking how elements of classes/systems *interact* and are *related*:
  - Does this data *need* to be here
  - Will these interactions make it difficult to change in future
- OOP has techniques to *help* with coupling/cohesion
  - But **you** need to put the work in to ensure it's actually well designed

Finality

- Child classes can *override* any public/protected function from a parent
- Private functions can't be overriden (since there is no access)
  - But they also can't be called by the child
- To fully control access we need something in the middle
  - A public/protected method that can't be overriden

Finality

- Child classes can *override* any public/protected function from a parent
- Private functions can't be overriden (since there is no access)
  - But they also can't be called by the child
- To fully control access we need something in the middle
  - A public/protected method that can't be overriden
- Functions marked `final` have this feature

Finality Example

```java
class Finality {
    public final String prettyPrint() { return "Final"; }
}

class SubFinal extends Finality {
    public String prettyPrint() { return "Override"; }
}
```

```
Finality1.java:10: error: prettyPrint() in SubFinal cannot override pretty
    public String prettyPrint() {
                  ^
  overridden method is final
```

More Final

Final can be applied elsewhere:

- Final *classes* cannot be inherited from
- Final *variables* cannot be reassigned
- It's all about controlling use/relations between components
  - That is, controlling coupling/cohesion

Summary

- Objects are Communication Cohesive
  - Methods that operate on the same data
- Language features for Cohesion/Coupling control
  - Visibility modifiers: public, private, protected
  - Inheritance: create cohesive “trees” of functionality
  - Abstract classes/interfaces: Coupling to API only
    - Specific implementations kept abstract
  - Polymorphism: Disallows coupling to concrete implementations
  - Finality: Control when overriding is allowed

Q & A From the Lecture

Q: What is the relationship between coupling and cohesion, does that mean we need to remove the coupling and increase the cohesion all the way

A: Coupling and Cohesion are inversely related such that programs with high cohesion usually have low coupling. One way to see this is to think about how much information moves over the system: if everything is cohesive then most information that changes together will be local to a component and unaffected by external components. If it's not, then you likely need information from across a larger number of places and so coupling increases.

Reminder: There's better and worse forms of coupling, but you can never remove *all* the coupling else you'd never write anything useful. E.g. you need to allow coupling to a classes public methods.

Q & A From the Lecture

Q: What is "common data" in Java code? How do we identify it

A: Common data occurs whenever multiple components (usually functions in OOP) access shared data. For example: methods in a class can share the data members of that class; so this is common data coupling. The main difference to other forms of data coupling is that this data is not passed via parameters.

There's an example on the next side to explain

Q & A From the Lecture; Cont.

```
public class User {
  // Common since both toString
  // and addYears have access to modify
  private String name = "";
  public int age = 1;

  public String toString() { return name + " " + age; }

  // i is *not common* since only addYears has access.
  public String addYears(int i) { age = age + i; }
}

public static void main(String args[]) {
  User u = new User();
  // Also common data since it directly accesses data within u
  // Getters/Setters turn common data coupling into interface coupling.
  u.age = 5;
}
```

Q & A From the Lecture

Q: Do you mean controlling the access to a class can reduce coupling

A: Yes! And this is one of the main reasons we have the access modifiers etc.

In particular it can transform "bad" coupling, e.g. common data, into better coupling like "interface coupling".

Q & A From the Lecture

Q: When should we use abstract classes vs interfaces

A: Use abstract classes when:

- You want to define shared functionality, e.g. default implementations of some methods
- You need to define *data* members in the class
- You don't need multiple-inheritance

Use Interfaces when:

- You only want to define abstract methods with no default implementation or data
- You need multiple-inheritance

Q & A From the Lecture

Q: Are there more examples of abstract classes

A: The common examples are things like:

- Defining an "Animal" class: https://www.w3schools.com/java/java_abstract.asp and forcing an override of the `sound` method
- Defining a "Shape" class: https://www.guru99.com/java-abstract-class-method.html and forcing an override of the `calculuateArea` method
- General "Product" class forcing an override of specifics

They occur frequently when we have a general interface and want to allow multiple concrete implementations

Q & A From the Lecture

Q: What is the difference between `Place p = new School()` and `School p = new School()`

A: The important bit is to think of the *type* of p, in `School p = new School()` we remember p is a School meaning it can do anything defined in **either** Place or School.

For `Place p = new School()` we ignore the fact we know the concrete version of the place, and instead treat p as a (abstract) Place only. This means we can only call things defined on Place. This avoids context coupling, since we no longer know the exact implementation details (the School) only general interface details (the Place). It also means we can transparently change it later, e.g. make it `Place p = new Hospital()` without changing any other code below it (so we have reduced coupling!)

Q & A

Q: Can we have an example of extreme use of interfaces

A: Some people like to always define a class interface using `interfaces` for example:

```
public interface HTMLFormatter { public String formatPara(String s); }

public class HTMLFormatterImpl implements HTMLFormatter { ... }
```

This means we can write a `HTMLFormatterImpl2` in future that is not coupled to `HTMLFormatterImpl`, e.g. it's not using inheritance and so it's quite flexible.

But, if this is the only implementation, and it's unlikely to change, then it seems like a lot of writing/overheads. There's no right answer to this, and every programmer/software company will have different standards around this!