Coupling and Cohesion

Gal Lalouche
Oath Inc.

Computer Science
Technion

Spring 2018
Coupling and Cohesion

There are many useful metrics in OOP, but we will focus on mainly two: Coupling and Cohesion

Cohesive
A module is cohesive to the degree that its elements belong together

Coupling
Two program modules are coupled to the degree that a change in one requires a change in the other

We want low coupling between modules, and high cohesion within them
Benefits of high cohesion

- Small code elements are easier to maintain and reason about
  - And easier to test too
- It’s easier to reuse smaller components than larger ones
- It’s easier to replace smaller components than larger ones
- But don’t go overboard, e.g., one method per class
  - OOP is about allowing us to combine things after all
What is “one” anyway?

“"A class should only complete a single task”, but what is a single task?
- Every non-trivial task is composed of smaller tasks
- Those small tasks have to be combined somewhere
  - For example, a compiler has many different steps (lexical analysis, semantic analysis, code generation, etc.)
  - But that doesn’t mean you shouldn’t have single compile method that compiles an entire project

- Cohesion is measured by the problem domain you are trying to solve
  - If you are getting bogged down by details, than you should delegate responsibilities
There are many smells indicating low cohesion

- **Disjoint field set.** A subset of methods using only a subset of fields indicating those fields and methods should be moved to their own class
- **Multiple unrelated changes.** A class should only have a single reason to change
  - Delegate responsibilities to dependencies
- **Asymmetry.** Code mixes up several layers of abstraction

Generally speaking, we improve cohesion either by **splitting** or by **delegating** (and splitting is better)
Disjoint field set

class EmailServices {
    private final EmailServer server;
    private final AddressBook addresses;

    public void sendEmail(Email e) {
        server.post(e.address(), e.content());
    }

    public String getAddressFromName(String name) {
        return addressBook.find(name.toLowerCase());
    }
}

Solved by splitting in two classes
Multiple unrelated changes

The below code will have to change if:

1. The communication protocol with the remote server changes
2. The data format changes

```java
class DataDownloader {
    private Json downloadJson() { ... }
    private Data parseData(Json json) { ... }

    public Data downloadData() {
        Json json = downloadJson();
        return parseData(json);
    }
}
```

Solved by extracting both parts to their own classes (improving testability in the process)
Asymmetrical code

```java
class DataDownloader {
    private final JsonDownloader jsonDownloader;
    private final DataParser dataParser;

    public Data downloadData() {
        Json json = jsonDownloader.download();
        PrintStream ps = new PrintStream("log.txt");
        ps.write(getCurrentTime() + " " + json.toString());
        ps.close();
        return dataParser.parse(json);
    }
}
```

Solved by extracting the logging portion to its own class
Types of cohesion

Types of cohesion (adapted from c2Wiki), from worst to best

1. **Coincidental.** Module elements are unrelated
   - Easiest to fix, too

2. **Logical.** Elements perform similar activities as selected from outside module
   - E.g., an IOUtils class, functools module in Python, or an exceptions package
   - When using a logically cohesive module, we usually just **pick and choose** the parts we care about
   - Easy enough to extract unrelated parts
   - Sometimes this is the easiest/cheapest way to bundle together (superficially) related activities

3. **Temporal.** Operations related only by general time performed
   - initialization(), or fatalErrorShutdown()
   - Harder to solve, since initialization has to be performed *somewhere*
   - But that does not mean that a single class should hold all unrelated initialization logic
   - Consider performing required procedures closer to where they are needed/possible
The below cohesions aren’t necessarily bad

4. **Procedural.** Elements involved in different but sequential activities

```java
// All of the below checks are unrelated to each other, but
// they all fall under the "register" procedure
public boolean registerUser(String userName, String passwd) {
    if (!isValidUserName(userName))
        return false;
    if (!isStrongEnoughpasswd(passwd))
        return false;
    Url url = getDatabaseUrl();
    Connection connection = getUserdatabaseConnection();
    if (!checkAvailableUsername(url, connection, uesrName))
        return false;
    putUser(connection, url, userName, passwd);
}
```

- The procedure just binds all required steps together
- Can increase cohesive by extracting the independent parts to their module
5 **Communicational.** Unrelated operations except need same data or input
   - E.g., StringUtils, itertools module in Python, or Data.List module in Haskell
   - In some languages this is implemented as **extension methods**
   - Not particularly more or less cohesive than procedural

6 **Informational.** performing a number of actions, each with its own entry point, with independent code for each action, all on the same data structure
   - In other words, a class
   - Differs from Communicational in that the data being operated on is the same for all operations
   - Not particularly more or less cohesive than sequential (some sites lists it as being less, some sites as more)
7 **Sequential.** Data is piped from one operation to another

- Differs from procedural cohesion in that operation **order matters**
- For example, a compile() method needs to do lexical parsing, followed by semantic parsing, followed by bytecode generation
- Some sources (e.g., c2wiki) list it as being less cohesive than informational

8 **Functional.** All elements contribute to a single, well-defined task

- E.g., Math.sin, file.readLines()
- Only applies to functions, but not all functions are functionally cohesive, e.g., handleException(e), or shutdown()
Coupling definition

Coupling

Two program elements are coupled to the degree that a change in one requires a change in the other.

- Coupling is **directional**
- Coupling is **quantitative**
- Some couplings are stronger than others
- Some couplings are worse than others
  - Coupling to **data** classes (POJOs) is *usually* not as bad as coupling to **service** classes
By and large, coupling is bad and we should minimize it

- Changes incurred by coupling are a **waste of time**
  - You’re not adding new features or fixing bugs
  - You’re not improving the design of your code
    - Though you might be improving the design of the classes you’re coupled to, that’s not relevant here

- The more modules are coupled, the harder they are change
  - The more susceptible things are to change, the less likely they are to change
  - The less likely things are to change, the more likely your design will deteriorate (entropy)
Types of coupling

Types of coupling (again, from c2Wiki)

1. Pathological/Content. Directly accessing (or, much worse, modifying) the data of another class
   - E.g., public fields
   - Not only are you coupling yourself, you can modify the code of other clients of the class!
   - They might not even be aware that such change are even possible!
   - Not common in Java, but all-too-common in dynamic languages

2. Global. Modules communicate via global data
   - Global variables are obviously bad, but in this type of coupling, the fact that all involved parties use the same global data is leaked
   - Therefore, it would be hard to change the global variable in the future since there many who depended on its behavior
   - Related: communication via data writing/reading, e.g., files or database modifications
3 Control. Communicating with a control flag

- In other words, the control flow of the receiving methods changes depending on the value of the flag (boolean, or enum)
- Which in turn means that method exposes its implementation details to the outside world
- Some sources put this as more coupled than global coupling
class ShoppingCartCoupled {
    // Cookies are an implementation detail
    public void addProduct(
        Product p, boolean saveToCookie) { ... }
}

// Also, leads to magic booleans
shoppingCart.addProduct(p, true); // What's true?
// Enums are universally better than Booleans, even if there
// are only two choices
shoppingCart.addProduct(p, Method.PERSISTENTLY);
// Of course, it's better to have different methods to
// reveal intent better
shoppingCart.addProductPersistenly(p); // Even better
// If you can, use polymorphism to remove the choice from
// the client code altogether
class PersistentkCart extends ShoppingCart {
    // Saves persistently, but clients don't know or care
    public void addProduct(Product p) { ... }
}
**Stamp.** Passing more data than is required to complete the task

- Parameters are harder to initialize
- Harder to change parts which aren’t used
  - And harder to change implementation to a similar interface which doesn’t include the unused data
- Can be easier than creating narrower interfaces, but not worth it in the long run
- Related: passing a more specific type than is needed, e.g., ArrayList vs. List vs. Collection vs. Iterable

**Why stamp?**

“The story goes that back in olden times some shops made rubber stamps for key data structures. You could stamp it onto a program spec document and perhaps check some boxes off for which one the module should use as input or output. It’s easier to pass the whole darned stamp to a subroutine than to make a new custom structure just for a single-use parameter. The COBOL copybook is a perfect stamp, too. I don’t know if there really were physical rubber stamps or if the whole story is apocryphal”
5. **Data.** Passing data as parameters
   - Obviously, not a bad kind of coupling, just not the best
   - Future implementations may require different, or fewer, parameters
   - Some passed is worse than others (next slides)

6. **Empty method.** We are only coupled to the method name, but pass no parameters

7. **No coupling.** We are not coupled to the module at all
Reducing coupling

There are several ways we can reduce coupling:

1. Depending on interfaces instead of classes
2. Encapsulation
3. Law of Demeter
4. Dependency injection
Depending on interfaces

It’s better to be coupled to interfaces than classes
- Interfaces are easier to implement than classes are to extend
  - No constructor
  - A class can implement multiple interfaces
  - Interfaces can’t be final (duh)
- Interfaces offer more control
  - No final methods (even on default methods)
  - No inherent state
  - A class can have a lot of functionality and only expose smaller parts of it to different clients via its implemented interfaces
- A class is specific; an interface is general
- Rule of thumb: interface for functionality, class for data
- But it might be best to avoid an interface with a single implementation
Preferring interfaces to classes is another example of PoLP

- Classes are more powerful since they add requirements (constructors) and state
- A fat interface is more powerful than a thin one

Reducing the coupling to concrete classes also increases the flexibility of your classes

- The more abstract and precise an interface is, the easier it is easier to replace it with a different implementation
- Since interfaces are easier to implement, it’s easier to replace one implementation with another
  - Or with a test double
Encapsulation

- Whenever you expose pieces of your implementation, you can assume clients will depend on them.
- And whatever clients depend on become much harder to change.
- But clients can’t depend on what they can’t see.
- Hiding details helps protect your clients.
  - Even if they don’t want it.
  - Convenience will always trump safety if you let it.
- Ideally your clients won’t even be aware of your class’s existence if you use interfaces.
  - Replace constructors with static factory methods.
AKA Tell, don’t ask

- A method \( M \) of an object \( O \) may only invoke the methods of the following kinds of objects:
  1. \( O \) itself
  2. \( M \)'s parameters
  3. any objects created/instantiated within \( M \)
  4. \( O \)'s direct component objects (that is: \( O \)'s fields)

- In particular, \( M \) should avoid invoking methods on results of invoked methods.

- A special case of encapsulation
class Shipment {
    private final List<Order> orders = new ArrayList<>();
    public void addOrder(Order o) {
        orders.add(o);
    }
    public void checkout() {
        for (Order order : orders) {
            // Violates LoD
            order.getCustomer().getAccount().withdraw(order.getAmount());
        }
    }
}
We apply LoD by only invoking methods on `Order`, not its dependencies.

```java
class Shipment {
    // ...
    public void checkout() {
        for (Order order : orders) {
            order.withdraw();
        }
    }
}
```

- The `withdraw` method has been “lifted” up to `Order`.
- In turn, the method will also probably be lifted to `Account` so `Order` could invoke it without violating LoD itself.
Law of Demeter pros

What do we gain by following LoD?

- Law of Demeter prevents us from reaching **deep** into our transitive dependencies
  - This creates couplings all along the dependency chain between all elements in the chain (one-way)
  - Not only is Shipment tied to Customer and Account, but Order is tied with Account as well
- Using mocks with code that violates the law is **very** cumbersome
  - You need to configure mockOrder to return a mockCustomer which itself needs to be configured to return a mockAccount!
What do we lose?

- The main cost is of course **API Bloat**
  - Even though, e.g., Order is no longer exposing an instance of Customer, it still has to expose its *methods*
  - Taken to the extreme, Order has to provide the functionality of all its *transitive closure*
    - But you probably don’t need all of it
- While reducing **coupling** we inadvertently reduced **cohesion**
Law of Demeter doesn’t always apply

- Generally speaking, data objects (or POJOs) are exempt from Law of Demeter
  - `person.dateOfBirth().year()` is fine
  - Even if in the future the data structure or internal representation changes, it’s usually possible to provide different view methods of the same data

- Some APIs use chaining explicitly, e.g., the Stream API in Java 8
  - They aren’t exposing dependencies, but are using transformative fluent APIs
    - We will talk about Fluent APIs later in the course
  - But such classes are still difficult to mock if one needs to

- Remember, LoD is about avoiding dependency exposure
Avoid invoking static methods
  - Including **new**!

Static coupling is the most explicit kind, since you’re coupled to a specific class name
  - And therefore lose any interface polymorphism

Instead, pass an instance to the invoking class at construction
  - Or use a framework like **guice**

You will learn more about dependency injection in the tutorials
Good, or not bad, couplings

Not all couplings are bad, or not as bad as others

- It’s fine to be coupled to standard library classes
  - And very stable classes in general
- You aren’t coupled to package-private classes
  - The same way you aren’t “coupled” to a private method
  - Private-package classes are an implementation detail
- Coupling to data classes isn’t as bad as coupling to service classes
The major problem is that increasing cohesion often leads to increased coupling, and reducing coupling can lead to reduced cohesion

- Delegation is the primary way to increase cohesion, but it also increases coupling, since you are now coupled to your delegation
  - But remember, not all couplings are as bad or as strong as others
- Doing more stuff internally, or exposing interfaces to that effect, reduces your couplings, but also your cohesion
- Balancing between the two is part of the challenge
SOLID is a common set of design principles, originated by Uncle Bob

1. Single responsibility
   - I.e., low cohesion

2. Open/Close

3. Liskov substitution

4. Interface segregation

5. Dependency inversion
Open/close principle

An entity should be **open to extension** but **close to modification**

- It should be possible to control the behavior of a class without changing its source code
  - Changing existing code is hard and error-prone
  - Instead, you should write flexible classes that can be customized by clients
- How do we achieve it?
  1. Polymorphism
  2. Dependency injection and parameterizing
  3. Inheritance
Open/close principle: Polymorphism

// If we want to shop for more than books, we have to modify
// this class's source code.
class BookShoppingCart {
    public void add(Book b) { ... }
}

// This class can accept any kind of product, including ones
// not written yet, and is thus more future-proof & robust
class ShoppingCart {
    public void add(Item i) { ... }
}
Open/close principle: Parameterization

```java
class ShoppingCart {
    // Saves to file, but can we make it save to database?
    public void save() {
    }
}

class CustomSavingShoppingCart {
    // Accepts a saver from an outside source
    public void save(Consumer<String> saver) {
    }
}
```
Open/close principle: Parameterization

Alternatively, we could have solved the previous problem using inheritance.

```java
class ShoppingCart {
    public void save() { ... }
}

class DatabaseSavingShoppingCart extends ShoppingCart {
    @Override public void save() { ... }
}

class CloudSavingShoppingCart extends ShoppingCart {
    @Override public void save() { ... }
}
```
Open/close principle: Parameterization

Alternatively, we could have solved the previous problem using inheritance

```java
class ShoppingCart {
    public void save() { ... }
}

class DatabaseSavingShoppingCart extends ShoppingCart {
    @Override public void save() { ... }
}

class CloudSavingShoppingCart extends ShoppingCart {
    @Override public void save() { ... }
}
```

However this solution is usually not as a robust
- Couples between the saving logic and the shopping cart logic
  - What if we want to reuse the saving logic for other classes?
- Doesn’t scale to multiple configurable properties

We’ll revisiting composition vs. inheritance when we talk about code reuse
Liskov substitution principle

Let $\phi(x)$ be a property provable about object $x$ of type $T$, then $\phi(y)$ should be true for object $y$ of type $S$, where $S$ is a subtype of $T$.

- The subtype cannot require more
  - AKA cannot strengthen pre-conditions
- The subtype cannot promise less
  - AKA cannot weaken post-conditions
- The subtype cannot throw new exceptions (but may throw subtypes of exceptions)
- The subtype may not modify the supertype’s contract
  - For example, make immutable fields mutable
  - Or allow items to be removed from an originally "add-only" interface
- Many properties (but not all) can be verified by the compiler
Suppose we are writing a Sims clone, but with robots! Should Robot extends Person?

- Would save a lot of code duplication, since a Robot can do many of the things a person can
  - walk, work, play, cook
- But some operation would be No-op
  - sleep, eat, bathroom
- Some operations would be down-right impossible
  - makeBabyWith???
- But as far as Liskov is concerned, the real gotcha is in the assumptions
  - A human won’t die if given food, water, and sleep, except for old age
  - A robot doesn’t need any of the these, but has to have its batteries charged
Can’t we fix it by overriding?

class Robot extends Person {
  // Charges the robot batteries, like a bed charges
  // a person! RobotBed extends Bed
  void sleep(RobotBed bed) { ... }
}

But the sleep method doesn’t override, it overloads!

Covariance only works for return values
parameters should be contravariant

For example, a robot might drink gasoline
as well as orange juice, but not instead of

In Java, as in most static languages, parameters are invariant
We can partially solve the above problems by having
Robot extends PersonLike, as well as Person extends PersonLike

Or, better yet, implements instead of extends

We can still get implementation using default methods
Can’t we fix it by overriding?

```java
class Robot extends Person {
    // Charges the robot batteries, like a bed charges
    // a person! RobotBed extends Bed
    void sleep(RobotBed bed) {
        ... 
    }
}
```

But the sleep method doesn’t override, it overloads!

- **Covariance** only works for return values
  - parameters should be **contravariant**
  - For example, a robot might drink gasoline **as well as** orange juice, but not **instead of**
  - In Java, as in most static languages, parameters are **invariant**

- We can partially solve the above problems by having Robot extends PersonLike, as well as Person extends PersonLike
  - Or, better yet, **implements** instead of **extends**
  - We can still get implementation using **default** methods
Liskov substitution principle (cont.)

LSP is an important principle for abstraction and decoupling

- If we adhere to principle, we can replace one implementation with another, more specific one
- Clients won’t be surprised by the behavior
- Clients can rely on more general supertypes than specific subtypes
- Rule of thumb: if A extends B, then, A is a B
  - “Subclass only if you can say with a straight face that every instance of the subclass is an instance of the superclass.” (JB)
Interface segregation principle

Clients should not be forced to depend on methods they do not use

- Avoid stamp coupling
- Prefer many small, precise interface to general large ones
**Dependency inversion principle**

1. Higher level modules should not depend on lower modules; both should depend on abstractions
2. Abstractions should not depend on details; details should depend upon abstractions

- Program to interfaces, not concretes
  - Increases code reusability
  - Reduces coupling
- But then what’s inverted?
Dependency inversion principle: example

Consider the following shopping cart that can save its state:

```java
package shoppingcart;

class ShoppingCart {
    private final SqlRepository repo = new SqlRepository();
    public void save() {
        repo.insert(serialized(this));
    }
    public void load() {
        setState(deserialized(repo.load()));
    }
}
```

The class depends on another class in another package for saving its data:

```java
package repository;

public class SqlRepository {
    public void insert(String data) {
    }
    public String load() {
    }
}
```
Of course, the first thing we should do is replace the concrete dependency with an interface one.

```java
package shoppingcart;

class ShoppingCart {
    private final Repository repo;
    public ShoppingCart(Repository repo) {
        this.repo = repo;
    }
    // Rest of code is the same
}
```

And in the repository package:

```java
package repository;

public interface Repository {
    public void insert(String data);
    public String load();
}

public class SqlRepository extends Repository {
    // ...}
```
Dependency inversion principle: example (cont.)

While this was a necessary step, we still didn’t fully adhere to the principle
○ Repository mirrors SqlRepository
  ○ If we decide to change methods in the concrete class, we will also have to change them in the interface
○ And therefore, we still have problems
  1. Our higher level module (package shoppingcart) still depends on a lower level one (package repository)
  2. Our abstraction (Repository), depends on details (SqlRepository), when it should be the other way around
○ We only did dependency injection, not inversion
The solution is, unsurprisingly, to **invert** the dependencies

- In this case, we do this by simply moving the Repository interface to the shoppingcart package
  - All other code remains the same!

- **What did we achieve?**
  - The shoppingcart module **owns** the Repository interface
  - Therefore, the SqlRepository must **conform** to *its* specifications, not the other way around

- Or, from the view point of the principle
  1. Our higher level module shoppingcart no longer depends on a lower level one, but both depends on abstractions (Repository)
  2. Our abstraction (Repository) no longer depends on details (SqlRepository), but our details depends on our abstraction

- **Cons?**
Dependency inversion principle: solution

The solution is, unsurprisingly, to **invert** the dependencies

- In this case, we do this by simply moving the Repository interface to the shoppingcart package
  - All other code remains the same!

- So what did we achieve?
  - The shoppingcart module **owns** the Repository interface
  - Therefore, the SqlRepository must **conform** to *its* specifications, not the other way around

- Or, from the view point of the principle
  1. Our higher level module shoppingcart no longer depends on a lower level one, but both depends on abstractions (Repository)
  2. Our abstraction (Repository) no longer depends on details (SqlRepository), but our details depends on our abstraction

- Cons?
  - Harder to reuse SqlRepository outside of shoppingcart
  - Usually solved by using **adapters**