Modules: Abstractions and Class metrics

Gal Lalouche
Oath Inc.

Computer Science
Technion

Spring 2018
Modules

We’re moving our focus from methods / single class to multiple classes

- The ideas we will discuss apply to more than just classes
  - Packages, modules, libraries, etc.
- Unless we are talking specifically about classes, we will usually use the more general term **modules**

There are two main reasons why we separate code into modules:

1. Abstraction
2. Composition
Abstraction

For our purposes, an abstraction is any construct that hides details under an interface

- **Functions** hide code behind function signatures
- **Classes** hide functions and related functionality data under a single name
Reasons for abstractions

What do we gain by abstraction?
Reasons for abstractions

What do we gain by abstraction?

1. **Naming**
   - Giving something a name lets us refer to it
   - Categorization aids documentation

2. **Code reuse**
   - More about this in the next lectures

3. **Simplicity**

4. **Flexibility**

5. **Inventor’s paradox**
We hide away details to make complexity bearable.
“The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise” — Djikstra

- Abstractions help us reason about our programs.
- Humans are particularly bad at understanding complicated systems.
  - 7 ± 2 rule
- It’s impossible to understand any non-trivial piece of code without abstraction.
  - How many assembly lines is your "Hello world!" program?
- By removing details, we can focus on essence.
- Software engineering is really just building higher-level abstractions on top of lower-level ones.
Abstractions are about trading control for simplicity

- We have the most fine-grained control when writing assembly language, but it’s hard to get anything done
  - We can’t control what we can’t access
  - But we don’t always need to

- Higher level code is usually less performant, but developer time is more expensive than CPU time
  - Features like garbage collection automate things for us, freeing our time to deal with other issues
  - But can make real time applications unusable due to lag
By hiding details from users of our abstraction, we give ourselves flexibility

- In OOP-jargon, we often call this encapsulation
- The less users know about our implementation, the easier it is to change; the more they know, the harder it is
  - Flexibility is a two-way street
  - Clients can also replace one abstraction with another as long as they are not depending on its details
- We can even replace the entire implementation of our abstraction with another one as long as we keep the same interface
  - For example, replace production classes with test doubles
**Inventor’s paradox**

“The more ambitious plan may have more chances of success” (George Pólya)

- Abstractions are more general, but also easier to reason about.
- As a result, solving the general case can sometimes be easier than solving the specific one.
  1. Proving properties
  2. Solving for $n$
  3. Reductions
  4. Polymorphism
The paradox originated in math

- Using formulas, which are general, to solve concrete problems
- Sometimes proving a more powerful theorem is easier than a weaker one
- It can be easier to prove properties of more abstract structures
  - E.g., groups vs. fields, or an arbitrary function vs. a specific one
- By hiding details, we can deal with less edge cases, and focus on the properties that do matter
- And of course, everything we can prove on a general case also applies to a specific one
Inventor’s paradox: solving for $n$

It’s almost always easier to solve a problem for a general $n$ instead of a concrete number for even small $n$s

- 25 different cases in your code? It’s easier to solve a general case, and then configure it to solve 25 cases
- Need a data-structure to contain 100 elements? It’s easier to use design one that can hold $n$ elements and set $n = 100$
  - Likewise for sorting, graph traversals, pretty much any algorithm
  - This is not (only) about code-reuse, but about simplicity
  - An ad-hoc solution might be more efficient, but would take a lot longer to develop
- `IntStream.range(2, n).filter(Prime::isPrime)` is shorter than hard-coding the list of the first $n$ primes
- `def gcd(a, b): return a if b == 0 else gcd(b, a % b)` is easier than programming the computation steps for solving specific $a$ and $b$
Reductions

- Reductions are very common in computer science
- We reuse general algorithms and data structures to solve concrete problems
  - We sort students by name using algorithms originally designed for sorting numbers
  - We use graph path algorithms to solve car routing problems
- Even programming: we use a general purpose language to solve our specific problem
Polymorphism

- Polymorphic code, whether via generics or interfaces
- On the one hand, the code is more general, since it works on more types
- On the other, by limiting the actions of what we can achieve, it is easier to reason about our code
- As a result, almost all data structures we use are generic
  - And before Java 5, we used Object containers
- For example, it’s easier to sort an array of Comparable<T> than it is to sort an array of some custom type with a complex comparison schema
  - We have two separate problems here: sorting, and specifying the schema
  - By forcing the two together, we make the sum bigger than its parts
  - Sometimes referred to as **super-linearity**
Abstraction complexity

- Obviously, the abstraction needs to be less complex than the thing it is abstracting.
- But the abstraction’s complexity should also *increase slower* than the implementation.
  - A useful analogy: if our implementation is some vague 2D shape, the abstraction is the perimeter of our shape.
  - The shape’s surface increases quadratically, but the perimeter also increases linearly.
Composition

Composition means taking multiple things and grouping them together

- Most compositions can recursively combine abstractions
  - A function may invoke other functions
  - Structs include other structs
  - An interface can accept interface arguments or return interface values
  - A class can have fields whose type is a class
  - Libraries use other libraries

- Without composition, abstraction is useless
Composition is at the heart of programming

- We take complicated problems and decompose them into simpler ones
- Still too complicated? Continue recursively
- Stop when you know how to solve the problem
- Compose the small elements back to bigger ones
Software design is all about trying to control complexity and fighting entropy.

- There is constant tension between analytical power and expressive power.
  - The more expressive power we have, the hard it is to analyse.
  - But on the other hand, the easier it is to solve the problem at hand.
- The more a system can do, the less we can predict what it will do.
- Abstracts are easier to compose and reuse than concretes.
  - Aside from the obvious type conformance, they make less assumptions.
  - A small language can be embedded in a larger one, but not the other way around.
  - Small abstractions can be composed into larger ones, but not the other way around.
- This is another example of the Principle of Least Power.
This relationship is discussed at length in a talk by Runar Bjarnason, titled *Constraints Liberate, Liberties Constrain*

- Constraining ourselves in one level frees us in another level, and vice versa
- More general abstractions have fewer characterizations; more concrete ones have more
- Since it’s easier to analyze and compose, it’s also easier to build larger abstractions on top of limited functionality
- Opting for a powerful solution now might make it harder to reuse or understand in the future
  - For example, regular expressions are easier to create and parse than context-free grammars, but are less powerful
  - But we can prove two regular languages are equivalent, and we can’t do the same for two context-free languages
Abstraction and precision

The more abstract our system, the more precisely we can analyse it

- Disregarding side effects and code that doesn’t finish, how many implementations exist for this **concrete** signature?

  \[
  \text{int} \ foo(\text{int} \ x);
  \]

- And how many implementations exist for this **general** signature (ignoring **null**)?

  \[
  <T> \ T \ foo(T \ t);
  \]

- And how many implementations exist for this **general** signature (ignoring **null**)?

  \[
  <T, S> \ S \ foo(T \ t);
  \]
Constraints and roads

Constraints are like roads, whereas liberties are like being able to travel wherever you want.

- Roads limit our ability to travel, since we can travel is predetermined paths.
  - They might force us to take a longer route than if we would have traveled in a straight line.
- But roads also help guide us to our destination.
- They make our travel safer, easier, and probably faster.
- Roads compose.
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 large ones
- It’s easier to replace smaller components than large 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

```java
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
Types of cohesion (cont.)

The below cohesions aren’t necessarily bad

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

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

### Informational

a module performs a number of actions, each with its own entry point, with independent code for each action, all performed 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)
**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

**Functional.** all elements contribute to a single, well-defined task
- E.g., Math.sin, file.readLine()
- Only applies to functions, but not all functions are functionally cohesive, e.g., handleException(e), or shutdown()
The major problem is that increasing cohesion often leads to increased coupling

- As mentioned, delegation is the primary way to increase cohesion
  - Splitting usually only applies in obvious cases and is not that common in even moderately well-designed code
- But now we are coupled to those dependencies!
- Balancing between the two is part of the challenge
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 units are coupled to your unit, the harder it is to 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 common in Java, but all-too-common in dynamic languages
   - 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!

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
Types of coupling (cont.)

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) { ... }
}
Types of coupling (cont.)

4 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”
Types of coupling (cont.)

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
To be continued