Mechanized Mining

https://courses.grainger.illinois.edu/cs126/fa2020/assignments/mineopoly-3-0/

Goals

- Finding your way in an unfamiliar codebase
- Writing modular code
- Testing code that is difficult to test
- Working with interfaces

Unfamiliar Codebase

Working with an existing codebase is the most likely scenario in which you will find yourself in industry. This is also true of working on a research project other than your own. This will be a good experience for you.

Modular Code

We are always working toward better design. Good designs typically result in modules that can be developed independently of one another. This is even true when you are the only developer working on a project.

Testing Private Methods - you will hear or read “Don’t do it” or “why would you want to do that?”

We can talk all day about why not to test private methods or should you directly test private methods.

Perhaps the question we are really asking is: “do you want your suite of unit tests to directly test private methods?” because OF COURSE we want to test our code. We want to test our private methods if for no other reason than we want personal assurance that they work EXACTLY as intended.
We definitely want to take an incremental approach to forming our solution and we want assurances along the way. This is working SMART.

You can:

● indirectly test them. (JUnit functionality, Black Box)
● “informally” test them (not tested with JUnit). Use “assert()” or write output to console, file, database, etc
● design an object that allows you to pass additional information out of your class. Use a getter for some status(es).
● You can declare them package private.

Every strategy has ramifications. You may end up:

● Creating a lot of output. This is messy and you will have to have a strategy to remove the code.
● Writing code that breaks the single responsibility principle in that it does something other than the most basic purpose.
● Writing code that you remove in the “final version”.
● Expose parts of your code in ways that violate encapsulation.
**Game Play Strategy**

Do not let this freak you out. Do not overthink it. A game play strategy may be a poor one. It only needs to meet the requirement that it allows automatic play of the game and “doesn’t crash”.

The strategy that you are expected to beat, the “random” strategy, is a poor one so almost any simple strategy SHOULD be able to beat it.

I will use an example from the card game “Uno”. The player must play a card that is of the same color or same number value as the one showing. The player may hold a card that allows them to override the current color and number value and then declare a new color.

**Strategy 1:**
Scan through your cards and play the first card in your hand that is playable.

**Strategy 2:**
Scan through your cards and play a number value card that matches the card showing of a color of which you have the most cards. You can do this by giving each of your cards a score that indicates the number of cards currently in your hand that could be played on subsequent turns assuming the color doesn't change.

**Strategy 3:**
Keep track of all cards you see played by any player including yourself. Make choices based on probabilities of other players being able to play.

- All of them qualify as strategies.
- None of them are probably going to be great strategies.
- None of these strategies are difficult to code.
CS126 - Design

So far, you have been learning design in a trial-by-fire scenario. You have had to use the requirements to satisfy assignments as direction for your design. It has forced you to think more about exactly how to write your implementations. Much of what you’ve probably been experiencing in succeeding at coding is satisfaction in getting functionality working. That makes you feel like a wizard. Design will be your new wizardry. Design is hard but can improve your ability to solve problems beyond your ability to understand and code complex algorithms.

Design is hard.

But there are principles you can follow. Last time we talked about “coupling” and that our goal was loose coupling. This is what we mean by design principles. Another related measure is “cohesion”.

Cohesion is the Object Oriented principle most closely associated with making sure that a class is designed with a single, well-focused purpose.

NumberGuesser

In NumberGuesser, the author moved from a design that had one method that does everything to using helper methods that also “do one thing well”. He also set up a way to decouple the UI from the “game engine”.

NumberGuesser still handles all of the UI work besides generating a number to guess and determining if the user’s guess was correct or not. We know this because it has one class that does everything.

While the UI has been given a way to pass results out of the game play, the functionality of the INTERACTION is still controlled by NumberGuesser5.
What would be the responsibility of NumberGuesser5 if it were completely decoupled from the UI?

What would be the responsibility of the UI?
What if the extent of NumberGuesser5’s responsibilities looked like this:

1. Creates the first number to guess on instantiation
2. Returns -1,0,1 depending on “Too Low”, “Correct”, “Too High”
3. Generates new number automatically when correctly guessed

UI:

1. accepts and validates user input (number or quit)
2. Sends correct type of guess to NumberGuesser5
3. Displays message to user based on return value

We could literally have two teams writing these two units independently. This is your goal.
Single Responsibility Principle:

From Wikipedia:

“The single responsibility principle is a computer programming principle that states that every module, class, or function\(^1\) should

- have responsibility over a single part of the functionality provided by the software, and
- that responsibility should be entirely encapsulated by the class, module or function.
- All its services should be narrowly aligned with that responsibility.

Robert C. Martin expresses the principle as, "A class should have only one reason to change,"\(^1\) although, because of confusion around the word "reason" he more recently stated "This principle is about people.(Actor)"\(^2\)

Perhaps more succinctly? (MJW):

Every module, class or function should do only one thing and do it well. “Do it well” would include doing all that it should do which means that it does not need anything added.
Cohesion and Coupling deal with the quality of an OO design. Generally, good OO design should be loosely coupled and highly cohesive. The aim of the design should be to make the application:

- easier to develop
- easier to maintain
- easier to add new features
- less Fragile.

Cohesion:

- Cohesion is used to indicate the degree to which a class has a single, well-focused purpose.
- Cohesion focuses on how single class is designed.
- Coupling concerns how classes interact with one another.

Benefits of Higher Cohesion:

- Highly cohesive classes are much easier to maintain and less frequently changed.
- Such classes are more usable than others as they are designed with a well-focused purpose.
Encapsulation:

From: THORBEN JANSSEN - NOVEMBER 30, 2017

Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It describes the idea of bundling data and methods that work on that data within one unit, e.g., a class in Java.

This concept is also often used to hide the internal representation, or state, of an object from the outside. This is called information hiding. The general idea of this mechanism is simple. If you have an attribute that is not visible from the outside of an object, and bundle it with methods that provide read or write access to it, then you can hide specific information and control access to the internal state of the object.
**Coupling:**

Coupling is the degree to which one class knows about another class.

Given two classes, A and B.

- If class A only interacts with class B through its interface then class A and class B are said to be loosely coupled.
- If class A also interacts through the non-interface parts of class B (A “knows too much about” B) then they are said to be tightly coupled.

If the developer changes the class B’s non-interface part (i.e non API stuff), when A and B are tightly coupled, A may break (depending on the change to B). If A and B are loosely coupled then then class A should not break.

“So its always a good OO design principle to use loose coupling between the classes i.e all interactions between the objects in OO system should use the APIs. An aspect of good class and API design is that classes should be well encapsulated.”

---

**Tight coupling has several dangerous consequences:**

- Particularly with strongly typed languages, such as C++, the compiler parses every class in the web to ensure type correctness. At best, this behavior may result in
make-dependencies that cause a recompilation of the complete system after each minor bug fix. At worst, the compiler or linker runs against some internal limitation and refuses to build the program at all.

- It is difficult to isolate a group of classes for component tests. Tracking down errors will be a hard and time-consuming job - harder than in any conventional project.

- The program is hard to maintain. Changes propagate all through the system and it is difficult to determine the responsibilities of a class. Therefore classes blow up with dozens of attributes and methods."
Levels or Degrees of Coupling:

Taken from (with edits by M. Woodley):

The levels of coupling below are ordered from high to low:

- **Content Coupling**: Content coupling, or pathological coupling, occurs when one module modifies or relies on the internal workings of another module.
- **Common Coupling**: Global coupling, or common coupling, occurs when two or more functions share global data. Any changes to them have a ripple effect.
- **External Coupling**: External coupling occurs when one or more modules share a format, interface or communication protocol, that is imposed upon them. This usually happens when modules are in direct communication with infrastructure layers, e.g., OS functions.
- **Control Coupling**: Control coupling occurs when one module controls the flow of another by passing control information, e.g., a control flag, a comparison function passed to a sort algorithm.
- **Stamp Coupling**: Stamp coupling, or data structure coupling, occurs when modules share a composite data structure and use only a part of it, possibly different parts. One example is of a print module that accepts an Entity, and retrieves its information to construct a message.
- **Data Coupling**: Data coupling occurs when methods share data, regularly through parameters. Data coupling is better than stamp coupling, because the module takes exactly what it needs, without the need of it knowing the structure of a particular data structure.
- **Message Coupling**: Message coupling is the lowest form of coupling, realized with decentralization and message passing.
Decoupling

Decoupling is the systematic coupling reduction between modules with the explicit intent of making them more independent, i.e., minimizing the value of $c$, as defined in the previous section. (refer to above link for metric “c”)

- Content coupling can be eliminated by following encapsulation.
- Common coupling can be resolved by introducing abstractions. Design patterns could prove useful towards achieving a good architecture.
- External coupling can be resolved by eliminating the knowledge of formats from the domain, and operating on concepts.
- Control coupling can be eliminated by using strategies or states.
- Stamp coupling can be eliminated by passing actual data.
- Data coupling can be eliminated by employing message passing.
- One very important principle to guide by in reducing coupling is the Law of Demeter, presented below.