Coupling and encapsulation

Coupling is the degree to which two or more modules are related to or depend on each other's implementations. In general we want *loose* coupling (or "less" coupling) between software modules. This is not unique to software; the idea is applicable to any complex systems with multiple interacting modules.

A non-software example

Consider the design of a Formula 1 car (or really, a regular car, but Formula 1 makes the example more exciting). It's got a huge number of parts designed and built by hundreds or thousands of engineers. For example, you have:

- Tyres
- Steering wheel
- Front/rear wings
- Engine
- Gearbox etc.
- Safety features on the chassis

Each of these modules is crucial for the car to continue functioning — the car can't turn at high speeds without a front or rear wing, it can't turn *at all* without a steering wheel, and it certainly can't go anywhere without an engine. So all of these parts need to work together harmoniously to give us all that on-track action.

Sometimes, these pieces need to be updated, e.g., if they stop working as well as we need them to.

For example:

- Your tyres are very old and need changing
- Your front wing has fallen off
- Electronics on your steering wheel aren't working any more.

If these modules were *tightly coupled* they would have to change together.

{: .callout } Put another way, if one module has to change, so do others that are tightly coupled with it, because they depend on each other's *current implementations*.

To put it in terms of the car, if the gear-changing controls on the steering wheel were tightly bound to the actual gearbox, Lewis Hamilton's race would have ended right there and then. One part needing changing would have rendered the entire system unusable.

The reason F1 pit teams are able to swap out an alarming number of parts of the car during a race is that these individual components make up mostly unrelated modules. That is, the modules are **loosely coupled** with each other.

When modules are *loosely coupled*, they are mostly independent. This does not mean they don't work together to enable the system as a whole to function; it just means that individual modules can be updated (or even swapped out entirely!) without other modules noticing, as long as they adhere to the same *interface*.

When I say *interface* above, I don't (necessarily) mean the `interface` construct in object-oriented languages like Java. I mean it more generally as "the surface at which two systems interact". In the F1 cars example, it's whatever nuts and bolts the front wing (or the new tyres, or the new steering wheel) is expected to fit into.

In software, the "interface" is made up of the public fields and functions that a module exposes to other modules. In Java, this means `public` variables and `public` methods in classes. What happens inside those public methods cannot and should not be relied upon by clients, as long as the function's effect is as intended.

Reducing coupling in software

There are many strategies for reducing coupling in software. The small-ish demos we look at now will be in Java, but these concepts (and most of what we talk about this quarter) are not limited to Java or object-oriented programming.

EJ15: Minimise the accessibility of classes and members[1]

Encapsulation (information hiding) — a module's internal information is hidden from the rest of the world. The idea behind this is simple; the less information a module exposes to the *other* modules, the less those other modules can rely on this internal information.

So for example, consider the `String` class in Java. The `String` class is an *abstraction*, that is, it's a simplification or a generalisation of the concrete underyling data, which is simply a byte array (`byte[]`). If each time we wanted to deal with variable length text in our programs, we had to build and reason about an array of individual characters, things would get messy quickly.

However, our code that uses `String`s rarely has to contend with the fact that there is an underlying array of characters. This information is hidden from all

[1] Items marked with `EJ` are principles from *Effective Java* by Joshua Bloch.

other classes.

We do this by making internal fields (data members, instance variables, attributes, etc.) *private* and inaccessible to the outside world. If classes need access to that data, we make `public` accessor methods available—the advantage is that this lets us control *who* accesses our data and *how*. Those `public` accessor methods form the "public interface" that we expose to other modules.

This leads into our next strategy.

EJ64: Favour interfaces over classes for parameter type

- E.g., `List<String>` instead of `ArrayList<String>` or `LinkedList<String>`
- Similarly `Map` instead of `HashMap` – you can replace `HashMap` with `TreeMap`, `ConcurrentHashMap`, or any other `Map` implementation as yet unwritten

Often used concurrently with this strategy is another strategy called *dependency injection*.

Dependency injection

- Instead of classes initialising their own dependencies, let the user of a class ("client") *inject* the dependency, i.e., as a parameter while initialising the object.
- This way, the class only depends on the *publicly exposed interface* of the dependency, and the client can choose *which specific implementation* will be used at runtime.

A toy example

```
public class Subject {
  private Topic topic = new Topic();
  public void startReading() {
    t.understand();
  }
}

public class Topic {
  public void understand() {
    System.out.println("Coupling");
  }
}
```

Introduce an interface

The `Subject` class would now not notice if the underlying `Topic` implementation changed.

```
public interface Topic {
    void understand();
}
```

```
public class Topic1 implements Topic {
    public void understand() {
        System.out.println("Got it");
    }
}

public class Topic2 implements Topic {
    public void understand() {
        System.out.println("Coupling");
    }
}

public class Subject {
    private Topic topic;

    public Subject(Topic topic) {
        this.topic = topic;
    }

    public void startReading() {
        this.topic.understand();
    }
}
```

A real-world example

Let's consider a software example in the real world. `ajv` is a JSON validator written in TypeScript. We won't be analysing the design of this entire package, but instead we'll take a look at a pull request and the discussion that led to it eventually being merged.

Let's take a look at the following issue and pull request that addresses the issue[2]. You're encouraged to work on this collaboratively with your classmates. Just make sure to make an individual submission on Canvas.

Key moments in the discussion

Issue is opened. Author suggests using the Re2 regex engine in the JSON parser because it guarantees worst-case linear time complexity, avoiding the possibility of "catastrophic backtracking" while parsing a regular expression. They want to give users of this JSON parsing library ("clients") the ability to use Re2 for all regexes for which it would work (not all regexes work with all engines).

[2]Thanks to Jamie Davis for pointing me to this example.

Initial PR is made. In their initial proposed solution (Pull request #1684), the developer bundles the Re2 engine as a dependency in the `ajv` module, and added an option to the parsing function that can be used to toggle use of the Re2 engine or the native Node.js engine.

- See the changes made in this initial commit for this PR The draft includes a new `useRe2` boolean option that clients can set as `true` if they want to use the Re2 regex engine.
- In the `usePattern` function in that commit, they check the `useRe2` option, and if that is `true`, they use a `new Re2(pattern)` object for parsing (necessitating the re2 dependency in this project).

{: .callout .ponder } What do you think of this proposed solution? Discuss with your classmates the pros and cons involved. Even better, can you think of alternatives?**

Project owner suggests a way to *not* require clients to have to include the Re2 dependency. See their comment. This involves creating an *interface* for a regex engine. So now the `usePattern` function doesn't know or care *how* a pattern is being parsed. The client makes this choice when they use the software. This was implemented in commit a0720f881b8331db1c8c38a805e24a71f5daacbb (see `lib/core.ts` and `usePattern` function in `lib/vocabularies/code.ts`)

{: .callout .ponder } What strategies that we talked about were implemented in this pull request to reduce coupling?
