Dependency graph

Gal Lalouche
Oath Inc.

Computer Science
Technion

Spring 2018
We can think of our modules and their dependencies as a directed graph $G(V, E)$ where:

- $V$ is the set of modules
  - Remember: a module can be a function, class, package, etc.
- An edge $m_1 \leftarrow m_2$ exists if module $m_1$ depends on module $m_2$

What can we say about this graph?
Simple graphs

A simple graph is better than a complicated one

We want to have this:

But not this:
We want to minimize the following properties

- Number of vertices
- Number of edges
- Cross-layer edges
- In-degrees
- Out-degrees
- Cycles
- Graph breadth
- Graph depth
- Graph connectivity
We generally want all properties to be as small as possible, but:

- Keeping cohesion high
  - We can dump all code to a single module result in a single-vertex graph
- Not one property is necessarily more important than another
- Improving one property will usually diminish another
- Sometimes diminishing a property without improving another is the right way to go
Let’s look at some graph properties and how they translate to software design

1. Degrees
2. Linear paths
3. Cycles
4. Layers
Out degrees

- A high out-degree could be a code smell, e.g., God object
- Could be fixed by abstracting several cohesive dependencies to a single module
- This is one of those rare cases where we can improve cohesion and coupling

Converting this:

![Initial Graph](image1)

To this:

![Refactored Graph](image2)
A large in-degree isn’t necessarily a code smell

- After all, it means we get high levels of code reuse, which is always good, right?
- But it *might* mean that the module in question does too much (i.e., low cohesion)
  - If that is the case, it should be broken down to multiple independent modules

Converting this:

![Original Diagram]

To this:

![Revised Diagram]
Bad graphs: cycles

A cycle in the graph is an obvious code smell

- Cycles mean co-dependency, which is a particularly nasty kind of anti-pattern
  - Have to refactor all cycle modules together
  - Source of bugs
  - AKA Shotgun Surgery
  - Especially problematic if two co-dependent classes are from unrelated modules
- Replace classes with interfaces or use mediators to fix cycles
Replacing concrete classes with interfaces

class Server {
    private final List<Player> players;
    // ...
    public void updatePlayers(Event e) {
        players.forEach.notify(e);
    }
}

class Player {
    private final Server s = new Server();
    public void notify(Event e) {
        // Update client with new event
    }
    public void makeMove(Event e) {
        s.updatePlayers(e)
    }
}
interface IServer {
    void updatePlayers(Event e);
}

interface IPlayer {
    void notify(Event e);
}

class Server implements IServer {
    private final List<IPlayer> players;
    @Override public void updatePlayers(Event e) {
        // Code unchanged
    }
}

class Player implements IPlayer {
    private final IServer s;
    @Override public void notify(Event e) {
        // Code unchanged
    }
}
Replacing concrete classes with interfaces (summary)

- Recall that a server and client can have a lot of functionality, not all of it relevant to their communication (PoLP)
- If required, it’s better for IPlayer and IServer to know each other than it is for the concrete classes

We went from this:

![Diagram 1](image1)

To this:

![Diagram 2](image2)
Another way of solving the previous cycle is through a mediators: common classes known by both parties

```java
interface Subscriber {
    public void accept(Message m);
}

public class PubSub {
    private final Multimap<Topic, Subscriber> subscribers = new Multimap<>();
    public void subscribe(Topic t, Subscriber s) {
        subscribers.add(t, s);
    }
    public void publish(Topic t, Event o) {
        subscribers.get(t).forEach(s -> s.accept(o));
    }
}
```
Using mediators

class Server implements Subscriber {
    private final PubSub ps = new PubSub();
    public Server(PubSub ps) {
        ps.subscribe(Topic.SERVER, this);
    }
    @Override public void accept(Event e) {
        ps.publish(Topic.PLAYER, e);
    }
}

class Player implements Subscriber {
    private final PubSub ps = new PubSub();
    public Player(PubSub ps) {
        ps.subscribe(Topic.PLAYER, this);
    }
    @Override public void accept(Event e) {
        // Update client with new event
    }
    public void makeMove(Event e) {
        ps.publish(Topic.SERVER, e);
    }
}
Mediators summary

- Client and server are completely oblivious of each other
- Easy to change behavior (e.g., peer-to-peer)

We went from this:

![Diagram of two circles connected bidirectionally]

To this:

![Diagram of three circles with arrows connecting them]

Cycles

Gal Lalouche
most non-trivial applications are built on **service layers**

We can find these layers in the graph by running BFS from the root node (e.g., the main function)

For example, UI layer above business layer above database layer
Back edges

Back edges are a code smell

- Low layer cohesion
- Weakly defined layer barriers
  - For example, a database that also does UI
Forward edges

Forward edges are also a code smell

- Leaky abstractions
- Law of Demeter is probably broken
Linear paths

- Too many classes makes the code harder to follow
- Loss of code locality
- Might indicate classes that aren’t carrying their weight
  - For example, a class that only wraps a single other class
- Not always bad, e.g., adapters and proxies
  - Lengthy paths are usually more problematic than short ones
- Fix by in-lining classes

Converting this:

```
+--------------------------+
|                          |
|                          |
| 1                         |
+--------------------------+
```

To this:

```
+--------------------------+
|                          |
|                          |
| 1  (wrapped by 2)        |
+--------------------------+
```

Gal Lalouche

Layers
Graph connectivity

Graph connectivity: the number of vertices/edges that need to be removed to make the graph not connected

- High connectivity between components makes it hard to replace one component with another
- We can reduce connectivity by using adapters and mediators
  - And limiting creation of said adapters to as few places in the code as possible

Converting this:

To this:
Using mediators, we can reduce vertex connectivity

Converting this:

To this:
We can define our vertices for any module level
- Functions, classes, packages, libraries, components
- Everything discussed still applies

We can define edges between components not if A depends on B, but if A passes B data
- Same underlying undirected graph
  - You can’t send or receive messages from another components without depending on it
- But the graph gives insight in to how data moves through our application
  - And again, we can apply the same minimization principles apply