Menti Q&A QR

Complexity and Challenges in Software

Blair Archibald

Overview

We will explore:

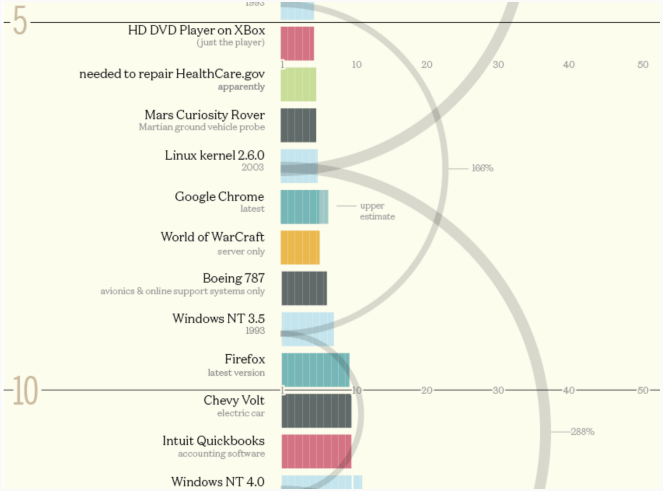
- Key challenges of developing software
- How abstraction helps us deal with complexity
- Difference between **code-owners** and **code-users**
- Key concepts of **coupling** and **cohesion**
  - Examples/classification of coupling/cohesion

Goal

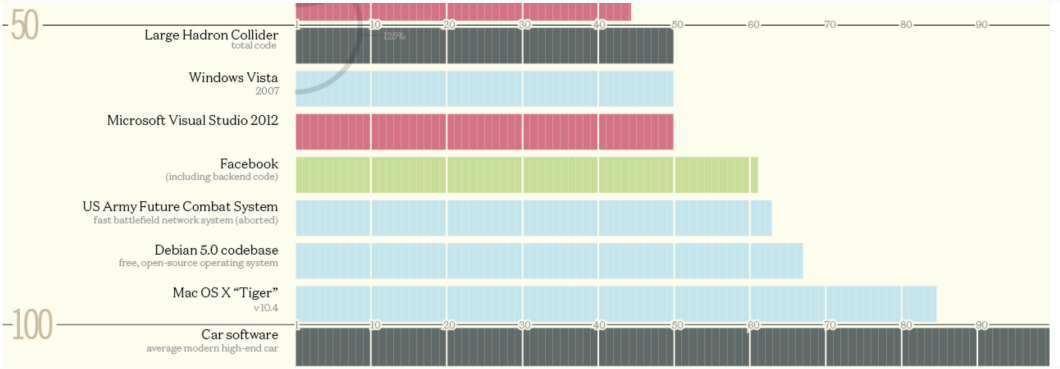
The **goal** of SE is to develop software solutions that **work** and **solve** some problem

- You might have the best design in the world
  - But if it doesn't work it is useless
    - In critical situtations can be worse than useless
  - But if it solves the wrong problem it is useless
    - Why the techniques of semester 1 are so important

Challenge 1: Scale



Challenge 1: Scale



From https://www.visualcapitalist.com/millions-lines-of-code/

Challenge 1: Scale

- No way any one person can fully understand all that code
  - Research suggests around 10,000 lines (of well designed code) **max**
- Necessitates:
  - **Techniques** to deal with complexity (this course!)
  - Working in teams (which add it's own complexities; Last semester)

Challenge 2: Change

- Unlike other disciplines software changes lots
  - To deal with hardware changes
  - Because the requirements weren't well scoped
  - Because everything needs emoji's now...

Image by Andrew Bell (Wikimedia)

Challenge 2: Change

- The **agile** techniques (semester 1) are **all about change**
  - How do we handle requirements changing? don't *rely* on gathering ahead of time
  - How do we detect issues that need change? constant communication (standups)
  - How do we track *changes* to functionality? user stories/sprint boards

Good Software Design must also allow for **change**

Challenges Recap

- Key challenges for SE are:
  - Scale
  - Change

We must **Design** for both

Abstraction as a Fundamental Technique to Handle Scale

- We need ways to reason about a system, without reasoning about every detail
  - Too many details otherwise
- Key technique is **Abstraction**
  - Replace *specific* code/chunks/modules/components/ideas with a *generalised* version
  - Reason on abstract version unless you need more information
- *Divide and Conquer* is a related technique
  - Divide general components into sub-components

Abstraction is Not Just Useful For Software

- Electronic Engineers *abstract* charge movement and call it *current*
- Electronic Engineers *abstract* transistors into *integrated circuits*
- Mathematicians *abstract* objects with (specific) binary operators into *groups*
- Car users care about speed-up/slow-down not how disk-brakes work internally
- ...

Working at Levels of Abstraction

- Engineers work at different levels of abstractions
- Don't need to understand all code, but still need to understand:
  - How to use components
  - How components fit together into a larger design
    - Sometimes a specialised position: *software architect*
- Components provide a well specified[1] application programming interface (API)
  - Details expected inputs/outputs/types/state changes etc

[1] Hopefully!

Developer Roles

Code-Owner

The person (or team) developing a particular component

Owners know all the details!

Code-User

The person (or team) using a particular component

Don't need to know everything, an abstract model of what happens is enough

Developer Roles

Code-Owner

The person (or team) developing a particular component

Owners know all the details!

Code-User

The person (or team) using a particular component

Don't need to know everything, an abstract model of what happens is enough

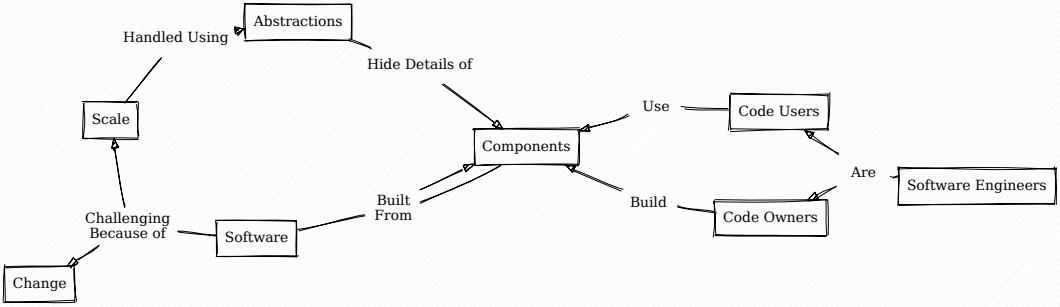
- In practice you are often both
  - Owner during initial development
  - User when developing other parts of the program
  - Owner when fixing bugs/improving documentation etc

Ownership

- Important to be aware of ownership
  - Company policy might stop you editing other code[2]
  - Might be owned by a third party, e.g. open source
  - Might not have editing rights, e.g. interfacing with commercial libraries
- A big reason companies minimise external dependencies
  - It's great there's a python library for everything; but who actually owns it?

[2] No matter how much you wish that int was a float please don't just change it!

Concept Map


Abstractions
Handled Using
Hide Details of
Scale
Components
Use
Code Users
Are
Software Engineers
Built From
Build
Code Owners
Challenging Because of
Software
Change

Coupling and Cohesion

- Q: How do we best split up a program into components?
  - What makes a good component?

Cohesion

"the situation when the members of a ~~group or society~~ **[component]** are united (Cambridge Dictionary)"

How components are built **internally**

Coupling and Cohesion

Cohesion

"the situation when the members of a ~~group or society~~ **[component]** are united (Cambridge Dictionary)"

How components are built **internally**

Coupling

"a device that joins two things together" (Cambridge Dictionary)

How components are **connected**

Coupling and Cohesion

Cohesion

"the situation when the members of a ~~group or society~~ **[component]** are united (Cambridge Dictionary)"

How components are built **internally**

Coupling

"a device that joins two things together" (Cambridge Dictionary)

How components are **connected**

Both are about **relationships** between components/elements

Key Take-away

We want high cohesion and low coupling

These are (roughly) opposites: Low coupling usually means High Cohesion

High Cohesion

- Internals of Components should be related
  - Helps future developers (probably future-you!) find things
  - Example: you don't expect the database connector to be inside the HTML rendering function
- Components should do one thing well
  - The push towards *microservices* based around this

Cohesion From The Creator of eXtreme Programming

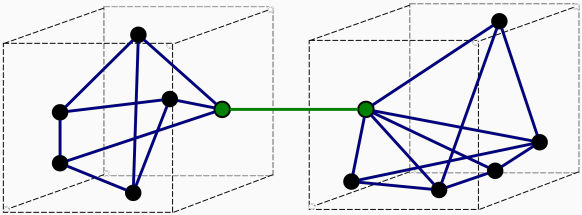
Change as a central theme again!

Lecture Q&A: menti.com/alo6c2asngoh

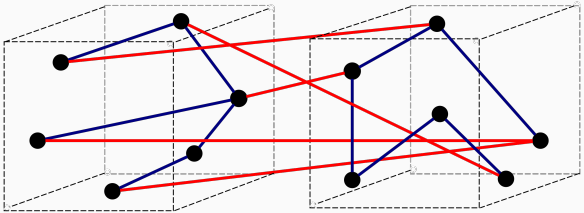
Low Coupling

- Internal details should not leak between components
  - If they do **change** becomes very difficult

Coupling and Cohesion Diagrammatically



a) Good (loose coupling, high cohesion)

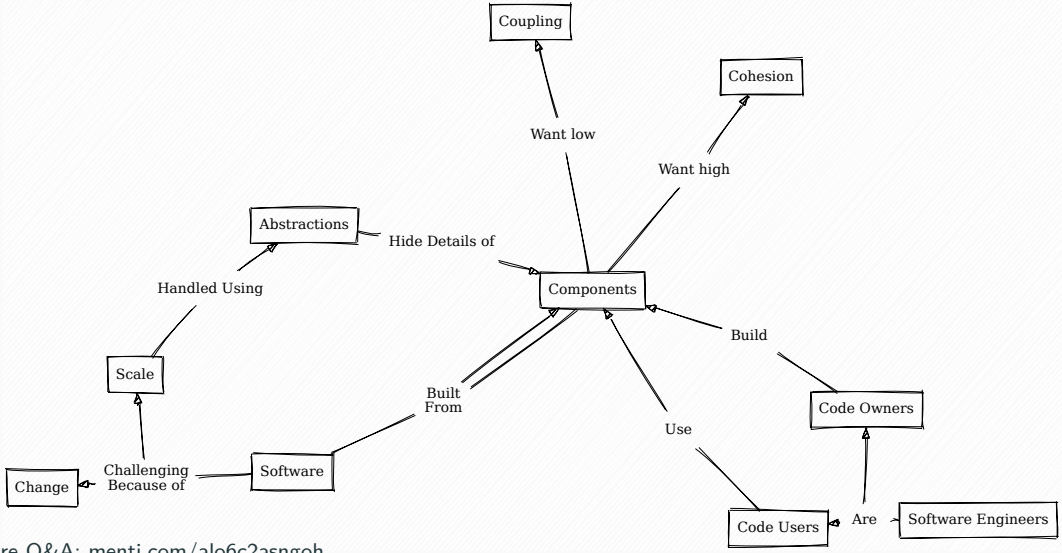


b) Bad (high coupling, low cohesion)

Lecture Q&A: menti.com/alo6c2asngoh

Public Domain Image (Wikimedia)

Recap


Coupling
Cohesion
Want low
Want high
Abstractions
Hide Details of
Components
Handled Using
Build
Scale
Built From
Code Owners
Use
Change
Challenging Because of
Software
Code Users
Are
Software Engineers

Software Cohesion

- Cohesion can be split into different *types*
  - Some are better than others
- What follows is a longish list of types
  - You might need to go over this a few times
  - Don't worry if you don't remember it all first time

Software Cohesion

- Cohesion can be split into different *types*
  - Some are better than others
- What follows is a longish list of types
  - You might need to go over this a few times
  - Don't worry if you don't remember it all first time
- Important bit is to **think about relationships between components**

Coincidental/Utility Cohesion (Usually Bad)

- Grouping of components without meaningful relationships
- Often a *misc* or *utils* module 杂项和工具块儿

```
List<Integer> list_of_int_pair(Integer x, Integer y) { ... }

List<Pair<Integer, Integer>>
cartesian_prod(List<Integer> x, List<Integer> y) { ... }

String prettyPrintList(List<Showable> s) { ... }
```

Logical Cohesion (Usually Bad)

- Grouping components that do similar logic
- Commonality can be quite superficial
- Examples:
  - All writes/reads to a database from a single place
  - All formatting functions for different datatypes

```
List<User> readUserDB() { ... }
List<Websites> readWebsiteContent() { ... }
```

Sequential/Procedural/Temporal Cohesion

- Groupings components that are used in-order/at the same time
- Related via *time*

```
void initialise() { ... }

int start_processing() { ... }
float compute_something(int) { ... }

void tear_down() { ... }
```

- *Might* have ordering/data movement constraints
  - Procedural: f(); g(); h()
  - Sequential: x = f(); y = g(x); z = h(x, y); ...

Communication/Informational Cohesion

- Components that operate on the *same data* are kept together
  - Should sound familiar from from object orientated programming!
  - The data should make coherent sense too: `int numUsers` and `bool ctrlKeyDown` don't seem very related!

```
class Point {
  private int x;
  private int y;

  ...
  void move(int offsetx, int offsety) { ... }
  bool isOrigin() { ... }
  String toString() { ... };
}
```

Functional Cohesion (Best)

- Grouping to solves a (well-defined) single problem
  - Binary search of a list: Takes a list and an element and says yes/no
  - Greatest common divisor
- Good for "solved" problems; much harder in the messy "real-world"
- Good when thinking at a higher-level of abstraction
  - The function of a web server is to serve web-pages and that (should be!) all

Recap

- Types of Cohesion
  - Coincidental/Utility Cohesion (Usually Bad)
  - Logical Cohesion (Usually Bad)
  - Sequential/Procedural/Temporal Cohesion
  - Communication/Informational Cohesion
  - Functional Cohesion (Best)

Reminder: We want **High Cohesion** when possible

Software Coupling

- Like Cohesion, can be split into different types
- Important bit is to **think about components interact**

Content Coupling (Usually bad)

- Component relies on the *internal* details of another
  - Assumes a specific algorithm, e.g. that some variable will exist after the algorithm runs
  - Assumes a data format
- Issue: if the internal details change, another component stops working!
- In general, internal details should be *hidden*
  - Remember: What the "encapsulation"/"abstraction"/"polymorphism" pillars of OOP are all about!

Common Data Coupling (Usually bad; read-only data can be okay)

- Dependency on shared data: **Global Variables**
- Issue: What if someone else changes/removes the global data you were depending on?
  - Very tricky to track changes and to debug
  - For modern systems makes *parallelism* very difficult
- We will see later in the course a better way (Singletons) to handle this

Control Coupling (Not always bad)

- Functionality of a module is changed from another
  - An additional parameter determines what a function should do
  - `void printValue(bool alsoPrintNewLine) { ... }`
- One component now needs to know control parameters for another
  - Okay if this is a small set, and all callers agree
  - Can make it harder to change a component since it needs to respect the "old" control interface

Stamp Coupling

- More information/features is passed between components than is needed
  - Gives too much power to the other component
  - For languages that copy parameters, can hurt performance

```
// Usually you don't want public here; just for the example!
class UserAccount { public String name; public int accBalance }

class View {
  public void displayName(UserAccount u) {
     // Why is the displayName function allowed to do this!
     // Does it need to know about balances?
     u.accBalance += 100000;
     System.out.println(u.name);
  }
}
```

Stamp Coupling

Better approach is to only give parameters that are needed

```
class UserAccount { public String name; public int accBalance }

class View {
  public void displayName(String name) {
     // Now we can't do this; It's an error!
     // u.accBalance += 100000;
     System.out.println(name);
  }
}
```

Data Coupling

- Two components interact through sharing data
- The more data you pass, the tighter the coupling is:
  - Passing lots of parameters to a method
  - Extra objects can reduce this, e.g. now you only couple to a single interface
    - But be careful of stamp coupling
- Not *common data coupling* as the data movement is local, e.g. via parameters

Routine/Temporal Coupling

- Methods/Components need to be called together to do something
  - `setup()`, `act()`, `teardown()`
- A *temporal* relationship exists between these
- Calling code must get the order right

Interface Coupling (Best Form)

- Interactions through a well defined API
  - Implementation details are hidden
  - Overall goal needs to be explicit: `CalculateTotal()` not `f()`

Recap

- Types of Coupling:
  - Content Coupling
  - Common Data Coupling
  - Control Coupling
  - Stamp Coupling
  - Data Coupling
  - Routine/Temporal Coupling
  - Interface

Reminder: We want **Low Coupling** when possible

Worked Example

```
1  void cleanup(boolean memfull, boolean user_req) {
2    if (user_req && IGNORE_USER_REQ) return;
3    log.write(LOG_LEVEL);
4    log.write("start cleanup");
5    collector.cleanup(memfull, mem.size, mem.start, mem.end, mem.id);
6    log.write(LOG_LEVEL);
7    log.write("end cleanup");
8  }
```

Worked Example

```
1  void cleanup(boolean memfull, boolean user_req) {
2    if (user_req && IGNORE_USER_REQ) return;
3    log.write(LOG_LEVEL);
4    log.write("start cleanup");
5    collector.cleanup(memfull, mem.size, mem.start, mem.end, mem.id);
6    log.write(LOG_LEVEL);
7    log.write("end cleanup");
8  }
```

- *Routine*: log writes are ordered
- *Common Data*: Who is setting `IGNORE_USER_REQ`, where is `mem`
- *Control*: `user_req` parameter controls behaviour
- *Content*: `mem.size` relies on `mem` having a (public) size attribute
- *Data*: Many elements of `mem` passed to collector

Improved Solution

```
1  void cleanup(boolean memfull, boolean user_req) {
2    if (user_req && IGNORE_USER_REQ) return;
3    log.write(LOG_LEVEL);
4    log.write("start cleanup");
5    collector.cleanup(memfull, mem);
6    log.write(LOG_LEVEL);
7    log.write("end cleanup");
8  }
```

- Let's start by replacing data and content coupling by passing `mem` directly
  - Shouldn't directly access members of objects (they should be private)
  - The collector can decide what attributes of mem it needs
    - No reason this method needs to know about them
- This requires a non-local change: collector needs a new interface

Improved Solution

```
1  void cleanup(boolean memfull, boolean user_req) {
2    if (user_req && IGNORE_USER_REQ) return;
3    log.write(LOG_LEVEL, "start cleanup");
4    collector.cleanup(memfull, mem);
5    log.write(LOG_LEVEL, "end cleanup");
6  }
```

- Remove routine/temporal coupling by adding a new log method
  - Takes the log_level and message together
  - User now **can't** forget to set this

Improved Solution

```
1  void cleanup(boolean memfull, boolean user_req) {
2    if (user_req && globals.should_ignore_user_req()) return;
3    log.write(LOG_LEVEL, "start cleanup");
4    collector.cleanup(memfull, mem);
5    log.write(LOG_LEVEL, "end cleanup");
6  }
```

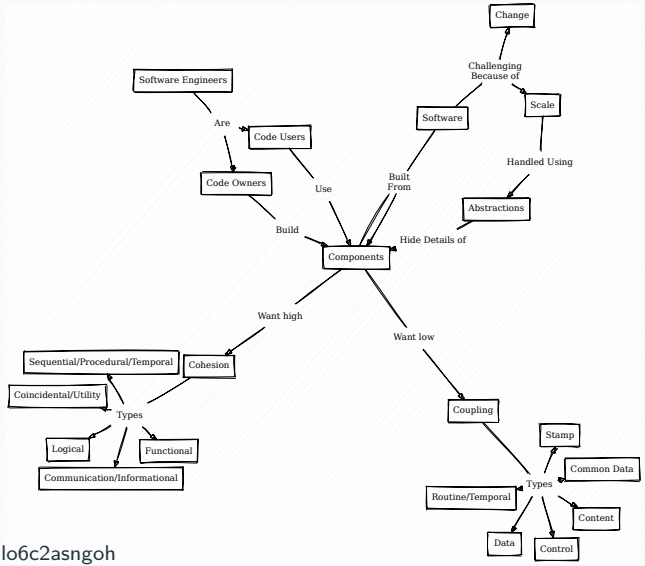
- Global object allows control over variable access
  - Better than global, publicly accessible variable
  - Singleton pattern (later in the course) allows for this
- Control coupling still present
  - Probably not a big issue in this case
    - Very simple control change
    - Not clear how you would use polymorphism etc

Improved Solution

```
1  void cleanup(boolean memfull, boolean user_req) {
2    if (user_req && IGNORE_USER_REQ) return;
3    log.write(LOG_LEVEL);
4    log.write("start cleanup");
5    collector.cleanup(memfull, mem.size, mem.start, mem.end, mem.id);
6    log.write(LOG_LEVEL);
7    log.write("end cleanup");
8  }
```

```
1  void cleanup(boolean memfull, boolean user_req) {
2    if (user_req && globals.should_ignore_user_req()) return;
3    log.write(LOG_LEVEL, "start cleanup");
4    collector.cleanup(memfull, mem);
5    log.write(LOG_LEVEL, "end cleanup");
6  }
```

Summary



Lecture Q&A: menti.com/alo6c2asngoh

Labs Next Week (25/1/23)

- Will be looking at programs with poor coupling/cohesion
  - Determine ways to fix it
- Please try to attend
  - Being about to read code, understand it, criticise the design, and suggest improvements would make a great exam question...

Links and Further Reading

- https://thevaluable.dev/cohesion-coupling-guide-examples/
- https://www.educative.io/answers/what-are-the-different-types-of-coupling
- https://en.wikipedia.org/wiki/Cohesion_(computer_science)
- https://en.wikipedia.org/wiki/Coupling_(computer_programming)