COMP1531

🎨 Design - Designing For Maintainability

Lecture 5.4

Author(s): Hayden Smith

(Download as PDF)

In This Lecture

- **Why? 🤔**
  - Maintainable software is resistant to breaking when inevitable changes occur over time
- **What? 📰**
  - Why care about maintainability?
  - How do we maintain?
  - Examples of increasing maintainability

⏮️ Recap: What Is Software Engineering?

Software Engineering is where we focus on building the right software for the right people, and making sure it's maintainable over time as it grows and as the people who work on it change.

⏮ Recap: What Is Software Engineering?

Software Engineering is where we focus on building the right software for the right people, and making sure it's maintainable over time as it grows and as the people who work on it change.

⏮ Recap: What Is Software Engineering?

Software Engineering is where we focus on building the right software for the right people, and making sure it's maintainable over time as it grows and as the people who work on it change.

So today we will discuss what **maintainable software** is.

🤔 Maintainability: Why Care?

Software in the real world changes over time. The less easily maintainable software is, the harder it is to adapt to these changes.

Well designed software is maintainable. Maintainable software resists the tendency to break as software changes or grows.

This topic is more critical than ever with the rapidly iterative nature of customer software.

🤔 Maintainability: Why Care?

Why does it change over time?

- Different people working on it.
- Requirements changing over time (e.g. new features).
- Needing to increase performance.
- Fixing bugs.

Code always grows, and never shrinks. You will never get on top of it.

You just need to be ready to adapt.

🤔 Maintainability: Why Care?

Maintainability is also more important than performance.

If code isn't maintainable, performance issues will appear through growth or through patching issues.

If code is easy to maintain, it will be easy to speed up because things tend to be modular.

Maintainability: But How?

OK, we get why maintainability matters. But how do we make software more maintainable?

 Maintainability: But How?

Generally there are a few key ways that improve software maintainability, including:

- **Testing**: By verifying the correctness of your code, it becomes easier to make changes to it without worrying about regressions in correctness. We've covered this.
- **System design**: Planning systems to make sense at a very high level / conceptual level (we cover this later).
- **Code design**: Thinking about code at a high level and low level of detail in terms of what makes things resilient to adapt to inevitable changes in the future.

 Maintainability: But How?

Generally there are a few key ways that improve software maintainability, including:

- **Testing**: By verifying the correctness of your code, it becomes easier to make changes to it without worrying about regressions in correctness. We've covered this.
- **System design**: Planning systems to make sense at a very high level / conceptual level (we cover this later).
- **Code design**: Thinking about code at a high level and low level of detail in terms of what makes things resilient to adapt to inevitable changes in the future.

We'll be talking today about the last one.

🧠 Code Design

Code Design is something that happens between writing tests and writing code.

It's when you know "what" a solution needs to do but need to plan "how" it does it.

Our tendency is not to write well designed code. We're often in a rush or people put pressure on us. Good code design takes a little more time and energy in the short term, but pays itself in the future.

🤔 6 Design Questions To Ask

1. Is there one source of truth for this?
2. Is this as simple as possible?
3. Is this over-designed or under-designed?
4. Are related modules kept close together?
5. Are unrelated modules kept far apart?
6. Am I speculating about how necessary this is?
7. Does this follow standard conventions?

1️⃣ Is There One Source Of Truth For This?

When you repeat yourself (e.g. same value defined in multiple places), a change in that value or capability requires changes in all locations.

It's very easy to forget to change it in all locations.

To avoid this, we follow a "don't repeat yourself" (DRY) method which focuses on reducing repetition in code. The same code/configuration should ideally not be written in multiple places.

"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system"

1️⃣ Is There One Source Of Truth For This?

How can we clean this up?

```ts
import { argv } from 'process';

if (argv.length !== 3) {
  process.exit(1);
}

const num = parseInt(argv[2], 10);

if (num === 2) {
  for (let i = 10; i < 20; i++) {
    const result = Math.pow(i, 2);
    console.log(`${i} ** 2 = ${result}`);
  }
} else if (num === 3) {
  for (let i = 10; i < 20; i++) {
    const result = i ** 3;
    console.log(`${i} ** 3 = ${result}`);
  }
} else {
  process.exit(1);
}
```

5.4_dry_dirty.ts

2️⃣ Is This As Simple As Possible?

Maintainable software is simple software. Use the simplest tools to solve a problem in the simplest way.

The more code we write, the more code we maintain and the more code we have to test.

Every line of code you don't write is bug free.

2️⃣ Is This As Simple As Possible?

Sometimes we refer to this approach as the "Keep it Simple, Stupid" (KISS) principle: the idea that a software system works best when things are kept simple. It is the belief that complexity and errors are correlated.

Clear code > Clever code.

No sane person likes clever code if it could be simpler.

2️⃣ Is This As Simple As Possible?

Example 1: Write a function to generate a random string with up to 50 characters that consist of lowercase and uppercase characters

2️⃣ Is This As Simple As Possible?

Example 2: Write a function that prints what day of the week it is today

2️⃣ Is This As Simple As Possible?

Example 3: Write a program that takes in command line arguments!

(Hint: Look at `Commander.js`)

3️⃣ Is This Over-Designed Or Under-Designed?

If we over-designed things, we have very complex abstractions to maintain for trivial changes.

If we under-designed things, we have to constantly restructure our systems to "grow" with scope increase.

3️⃣ Is This Over-Designed Or Under-Designed?

For example, sometimes repeating yourself twice (but only twice) is OK! Unconditional conforming to a principle is a bad idea, and can sometimes add complexity back in.

(Hayden notes: U = Credit, O = Cash Account)

3️⃣ Is This Over-Designed Or Under-Designed?

(In my opinion) most programmers go through three stages of growth:

- (Early) Under-design
- (Amateur) Over-design
- (Matured) Good design

4️⃣ Are Related Modules Kept Close Together?

5️⃣ Are Unrelated Modules Kept Far Apart?

Coupling is the degree of interdependence between software components.

We want related components to be tightly coupled, and unrelated components to be loosely coupled.

The more software components are connected, the more changes and alterations to one component may break another.

Excessive coupling can also lead to **spaghetti code**.

6️⃣ Am I Speculating About How Necessary This Is?

Top-down thinking says that when building capabilities, we should work from high levels of abstraction down to lower levels of abstraction.

This means that we avoid writing low level utiltiy functions that have the risk of never getting used.

Sometimes we call this the "You aren't gonna need it" approach (YAGNI) that says a programmer should not add functionality until it is certain it will eventually be used.

This helps remove unnecessary code, and therefore results in having less to maintain.

(Example: Cache)

6️⃣ Am I Speuclating About How Necessary This Is?

Question 1: Given two Latitude/Longitude coordinates, find out what time I would arrive at my destination if I left now. Assume I travel at the local country's highway speed

7️⃣ Does This Follow Standard Conventions?

The last question you should always ask yourself is "has this been solved before? And if so, has it been solved a certain way?"

For example, you wouldn't build a web server in the C programming language. The more popular an approach is, the more likely it is that other people will be able to maintain it because they are familiar with it.

This can be everything from style (e.g. uppercase constants) through to library choice (e.g. express server is popular).

7️⃣ Does This Follow Standard Conventions?

```
1 function dateNow() {
2   return new Date().toISOString();
3 }
4
5 console.log(dateNow());
```

5.4_benign.ts

7️⃣ Does This Follow Standard Conventions?

This is arguably a good reusable abstraction - but is it really easier to maintain? Why would you take away clarity for this?

```ts
function loop(count: number, callback: (num: number) => void) {
  for (let i = 0; i < count; i++) {
    callback(i);
  }
}

loop(5, console.log);
loop(7, console.log);
```

5.4_benign2.ts

 Refactoring

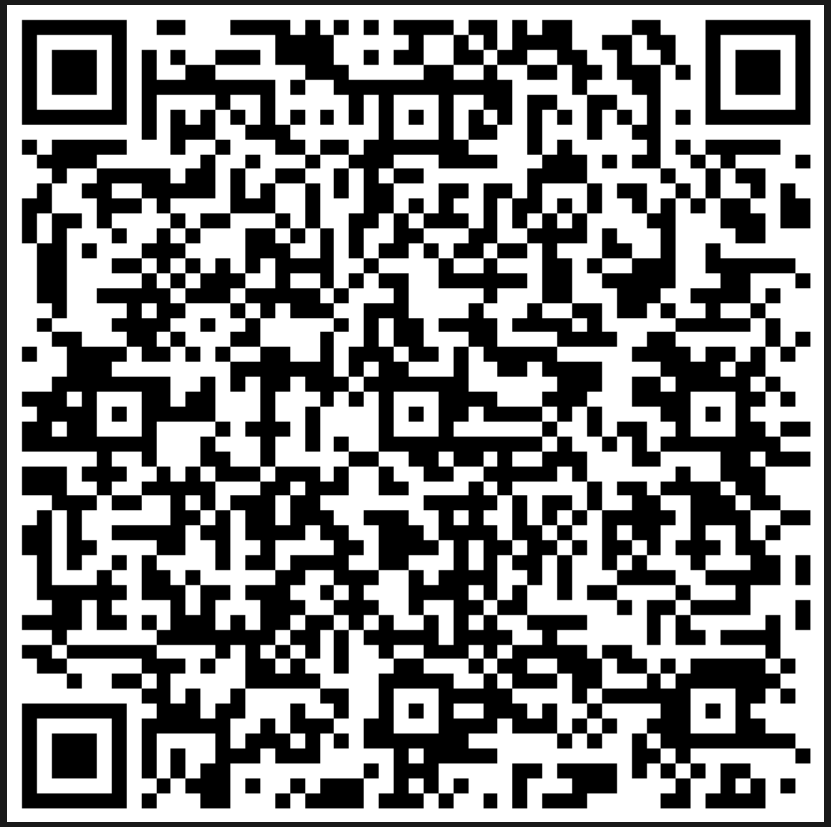
Conversations about code design don't finish just because your software is finished. No one writes perfect software the first time.

Refactoring is the process of restrucutring existing code without changing its external behaviour.

Typically this is to fix poor design in existing code to make things more maintainable.

Refactoring is strongly discouraged if you don't have a strong test suite. "Blind" refactoring may introduce more bugs than the cost of harder-to-maintain software.

🦻 Feedback



Or go to the form here.