Real-world Software Patterns

https://tinyurl.com/331-20sp-ethics-slides

Jared Moore

June 5, 2020

Software patterns

In this class, we teach you to recognize a pattern and apply its solution.

We ask you:

- to go from finding a bug to using a debugger,
- to go from implementing a method to reasoning about that method by writing tests for it,
- or to go from repeating yourself in the methods of various objects to defining inheritance.

Why do this?

There are many answers. Some include:

- because we say so,
- because it often increases computational efficiency,
- and because it often increases efficiency more generally.

But

- there's a limit to the number of helpful tests to write,
- inheritance doesn't always prevent repeated code,
- and what's efficient for one isn't always efficient for all.

So, today we'll discuss the 'why' of software engineering.

A software ethic

You have learned to be "ethical" software engineers.

- *(Where "ethics" is doing good according to the practices and culture of software engineering.)*

But what about when these practices of software engineering run into the real world?

Consider, for example:

- APIs which compromise individual privacy,
- software which enforces a gender binary,
- and user interfaces unusable by people with different abilities.

Looking for real-world patterns

What if we could recognize design patterns not just in software but also in how software interacts with the real world?

What about for:

- **accessibility** (e.g. sightedness, handedness, mobility, deafness, and many more),
- **diversity** (e.g. race, gender, sexuality, age, learning style, and many more),
- or **privacy** (e.g. of individual data, of trends, from certain groups, and many other facets)?

(*We might look for many other patterns besides.*)

Applying those patterns

Each of you have been assigned to a Zoom break-out room.

- While there, please unmute, turn on your video, and be vocal!

We'll consider three real-world software patterns (scenarios, really).

- First, I'll present a scenario to class in the main room.
- Then, groups will discuss the *"in group"* slides in break-outs.
- Then, groups will report back to the class in the main room.

APIs and privacy

```
// pre: Accepts a String, address, representing a written
//      address; attempts to disambiguate addresses based
//      on spelling, etc. (ex. "1600 Pennsylvannia Avenue"
//      and "1600 Pensylvania Ave" have the same return).
// post: Retuns that address in a standard Adress object
//       or null if bad or unrecognized input;
//       blocking function.
public static Address lookupAddress(String address);

public abstract class Address {
    public String getStreetName();
    public String getStreetNumber();
    public String getPostalCode();
    ...
}
```

APIs and privacy: *in groups*

As a group, 1) briefly discuss the questions below. Then, 2) write one other 'what if' question about `lookupAddress`*.*

Often we say implementation details are irrelevant, but what if...:

- `lookupAddress` saves all of the addresses it receives,
- `lookupAddress` stores timestamps and meta-data on queries,
- `lookupAddress` sends all of that information to a web-server,
- a user of `lookupAddress` is recognizable based on their misspellings (e.g. always writes "Pensylvania"),
- or `lookupAddress` is used for sensitive addresses (e.g. domestic violence survivors, government dissidents)?

APIs and privacy

Conclusion:

- When we pay attention to **privacy** we sometimes find issues with our good software engineering practices like **abstraction**.

Inheritance and diversity

```
public abstract class User {
    // lots of shared, non-abstract, methods
}

public class Man extends User  {
    public String toString(){
        return "I am a man!";
    }
}

public class Woman extends User {
    public String toString(){
        return "I am a woman!"
    }
}
```

Inheritance and diversity: *in groups*

As a group, 1) briefly discuss the questions below. Then, 2) discuss the questions on the next slide.

- What might be the motivation to use a software pattern in this way?
- What might an understanding of gender say about such a use of inheritance?

Inheritance and diversity: *in groups*

Does this 'fix' the problem?

```
public class User {
    public String toString(){
        return "I am a person!";
    }
}
```

What if:

- another part of the application requires a user to be either a `Man` or a `Woman`? *(e.g. a sorting function, a key in a database)*
- a user considers their gender mutable?
- gender identity is inferred by the program as opposed to provided to it?
- we allow gender identities besides `Man`, `Woman`, or none?

Inheritance and diversity

Conclusion:

- When we pay attention to **diversity** (here, gender) we sometimes find issues with our good software engineering practices like **inheritance**.

(For more check out [this poster] from an Allen School alum!)

Campus Paths and accessibility

Now let's think a little more open ended. Recall the Campus Paths assignment.

What might we say about Campus Paths regarding the pattern of (in)accessibility?

Campus Paths and accessibility: *in groups*

As a group, briefly discuss the questions below.

- Does Campus Paths have assumptions that favor some people over others?
  - *(If you're stuck: when might someone not prefer the shortest path? Why?)*
- What are those assumptions?
- Did you notice these assumptions as you completed your assignment?
- What could, should, or would you do about these assumptions? What should we do?

Campus Paths and accessibility

Conclusion:

- When we pay attention to **accessibility** we sometimes find issues with our good software engineering practices like a **generic** user.

$$\lim_{abstraction \to \infty} f(abstraction) = 0$$

Computer science is a function of abstraction:

- "give me your parameters and returns," we say "not your implementation details";
- "let's just assume that memory is infinite";
- "how about we treat all users as the same for now."

But sometimes abstraction doesn't work. Think about

- blocking method calls for i/o,
- memory intensive methods like deep recursion,
- or users who fit into some of the patterns mentioned above.

Evaluating the limit of abstraction

Evaluating the limitations of abstraction is hard.

- (*But such evaluation is the essence of comptuer science!*)

So what do we do—simplify the world so we can find a closed-form optimum for it? Do we ignore the difficulties posed by abstraction?

No, we critically design, test, and revise our abstractions and our design patterns to form new patterns and new abstractions.

“I object!”

“Okay, I get it,” you say, “but you haven’t told us what to do about these ‘real-world’ patterns; they aren’t **fixable**.”

- Don’t we deal with constraints all the time?
  - (*e.g. writing a good spec*)
- Even when there are no ‘fixes’ often acknowledging shortcomings is enough.
  - (*e.g. error messages*)

“I object!”

“Okay, I get it,” you say, “but recognizing these ‘real-world’ patterns isn’t my **job**; I’m just a software engineer.”

- Often these patterns occur *because of* software engineering and, as software engineers, we understand the software best.
- So, whose ‘job’ is it to recognize these patterns but our own?

Learning to see, then learning to fix

Ask yourself:

- Can you recognize not just software patterns but also patterns about the use of software in the world?
- Can you communicate your technical choices to decision-makers? (*When it is someone else's 'job' to decide.*)
- Can users, peers, colleagues, non-users, etc. communicate their needs to you?

If not, you can learn.

(*After all, you did learn how to recognize software patterns.*)

Learning to see, then learning to fix

Consider classes which approach this concept more explicitly:

- 440 on human-centered design
- 444 on data visualization and perception
- 481h on accessibility
- 484 on adversarial thinking
- 492 on computer ethics
- and many aspects of other courses besides, including courses in the Information School, HCDE, Design, etc.