Secure Software Design

Andey Robins

Spring 23 - Week 7

Design Patterns

Outline

- Lots of Different Design Patterns
- State Patterns
- “Preventing Security Bugs Through Software Design”

Design Patterns

Command Pattern

Encapsulate a request as an object, *thereby letting users parameterize clients with different requests, queue or log requests, and support undoable operations.*

In other words, wrap up a method call as an object.

```
interface Command {
    execute: () => ();
}
```

Add in Actions

```
class SendCommand implements Command {
    execute() {
        send();
    }
}

class ReadCommand implements Command {
    execute() {
        read();
    }
}
```

Refer to Commands

```
class Deaddrop {
    reader: Command
    sender: Command

    constructor(r: Command, s: Command) {
        this.reader = r;
        this.sender = s;
    }

    sendMessage = () => sender.execute();
    retrieveMessages = () => reader.execute();
}
```

Making Things Undo-able

```
interface Command {
    execute: () => ();
    undo: () => ();
}

class SendCommand implements Command {
    lastMessageUid: number

    execute() {
        this.lastMessageUid = send();
    }

    undo() {
        deleteMessage(this.lastMessageUid);
    }
}
```

Not all Commands Need to be Undo-able

```
interface Command {
    execute: () => ();
}

interface UndoableCommand extends Command {
    undo: () => ();
}

class SendCommand implements UndoableCommand {
    /// --- SNIP --- ///
}

class ReadCommand implements Command {
    /// --- SNIP --- ///
}
```

When to Use Commands

Use the command pattern when you want to easily:

- maintain some sort of navigable "history" of actions
- generalize actions to allow for customizable behavior

When not to Use Commands

The command pattern should be avoided when you have:

- a stateless action
- a general action where having multiple instances of behavior is unnecessary overhead

Flyweight Pattern

Use sharing to support large numbers of fine-grained objects efficiently.

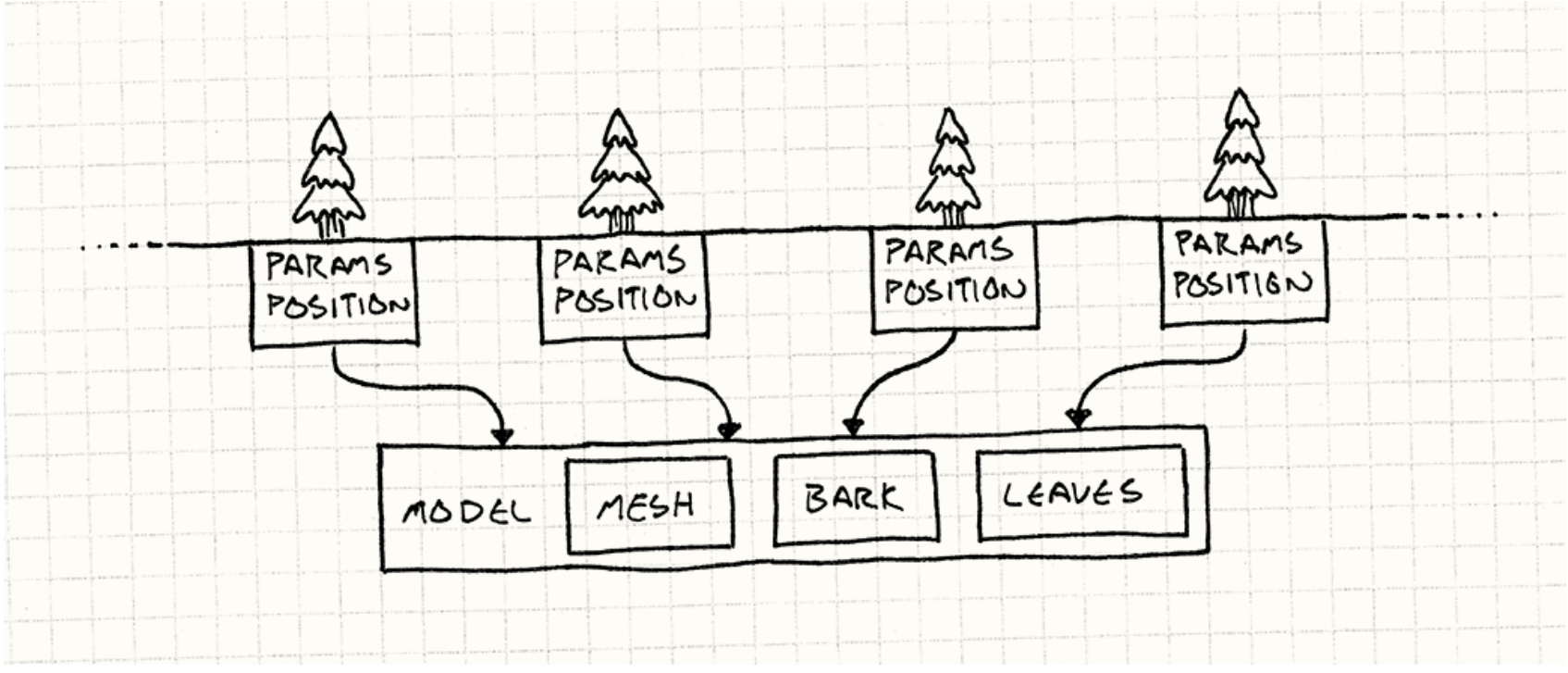

Figure 1: Using Flyweight to share Tree Assets

```
class TreeModel {
    mesh: Mesh
    bark: Texture
    leaves: Texture
}

class Tree {
    model: TreeModel
    position: [number, number]
    height: number
    tint: [number, number, number]
}
```

Usage of Flyweight

1. Identify the *intrinsic* or shared information
2. Identify the *extrinsic* or unique information
3. Create an object to hold both
4. Add a reference to the single intrinsic object in each extrinsic object

Usage of Flyweight

Use the Flyweight pattern when there are large volumes of shared information. Don't use it when there aren't.

Also, this fits very nicely into a "cache" pattern as well. On the first lookup of a piece of data, return the full thing, but then future instances can then use the previous lookup to improve performance.

Observer Pattern

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Read Receipts

```
const readMessages = async (user: string) => {
    /// --- SNIP AUTH & ERROR HANDLING --- ///

    getMessagesForUser(user).then((messages, senders) => {
      messages.forEach((message: string, i: number) => {
        console.log(message, "\n");
        // add in notification
        senders[i].notify(message);
      });
    });
}
```

Who are the Senders?

```
interface Sender {
    uid: number
    notify: (message: message) => ()
}

class DeaddropSender {
    uid: number

    constructor(uid: number) {
        this.uid = uid;
    }

    notify(message: string) {
        db.saveReadReceipt(message);
    }
}
```

When to Use Observers

Make use of the observer pattern when you want to decouple your logical modules and change to notification patterns.

Avoid the observer pattern in scenarios where its simpler to just call a method. This pattern makes a lot of sense in highly OO-languages, but can make less and less sense as we move to more functional languages or function-first designs.

Prototype Pattern

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

User Factory

```
class User {
    username: string
    password: string
    reader: Command
    sender: Command

    constructor(u: string, p: string, r: Command, s: Comma
        this.username = u;
        this.password = p;
        this.reader = r;
        this.sender = s;
    }

    /// --- CONTINUED --- ///
}
```

```
/// --- CONTINUED --- ///

changeUsername(u: string) {
    this.username = u;
}

changePassword(p: string) {
    this.password = p;
}

copy(): User {
    return new User(
        this.username,
        this.password,
        this.reader,
        this.sender
    )
}
```

```
class NewUserFactory {
    proto: User

    constructor(p: User) {
        this.proto = p;
    }

    newUser(name: string, pass: string): User {
        let newUser = this.proto.clone();
        newUser.changeUsername(name);
        newUser.changePassword(pass);
        return newUser;
    }
}
```

When to Use Prototypes

From Game Programming Patterns: *"I honestly can't say I've found a case where I felt the Prototype* design pattern *was the best answer."*

Instead, the paradigm can be useful.

The Paradigm of Prototypes

1. Allow overloading/instantiation
2. Fall back to a default lookup when things fail
3. Make use of data as a way to declare prototypes instead of code

Goblins

```
{
    "name": "goblin grunt",
    "health": 30,
    "resistances": ["cold", "poison"],
}
{
    "name": "goblin wizard",
    "prototype": "goblin grunt",
    "spells": ["fireball", "magic missiles"],
}
{
    "name": "goblin archer",
    "prototype": "goblin grunt",
    "weapons": ["crossbow"]
}
```

Singleton Pattern

State Pattern

Monads

Monads are the solution to state, because "state bad."

If you learned about Monads in functional programming, forget what you know.

"*A monad is a monoid in the category of endofunctors.*"

"A monad is a value with additional information encoded in the type."

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

```
enum Result<&str, &str> {
    Ok(&str)
    Err(&str)
}

Ok("user: andey");
Err("no user found");
```

Either Monad

An encoding of one thing or the other in the type of the value.

In other words, the `Result<T, E>`. Either a `<T>` or an `<E>`

```
enum Option<T> {
    None,
    Some(T),
}
```

```
enum Option<&str> {
    None,
    Some(&str)
}

Some("andey");
None;
```

Maybe Monad

An encoding of something or nothing within a single type.

In other words: `Option<T>`.

Monads

"A monad is a value with additional information encoded in the type."

We use them, because it allows us to have logic which doesn't need to handle errors, we can instead encode whether an error happens through the type (i.e. a Monad).

User Authentication with Monads

```
fn get_user_session_token() -> Option<Vec<u8>> {
    let user: Option<String> = get_user();
    let authentication: Option<String> =
            authenticate_user(user);
    let session: Option<Vec<u8>> =
            create_session(authentication);
    session
}
```

Consume with:

```
match get_user_session_token() {
    Some(t) => t,
    None => panic!("Unable to get session token")
}
```

Questions

Friday “Guest” Lecture

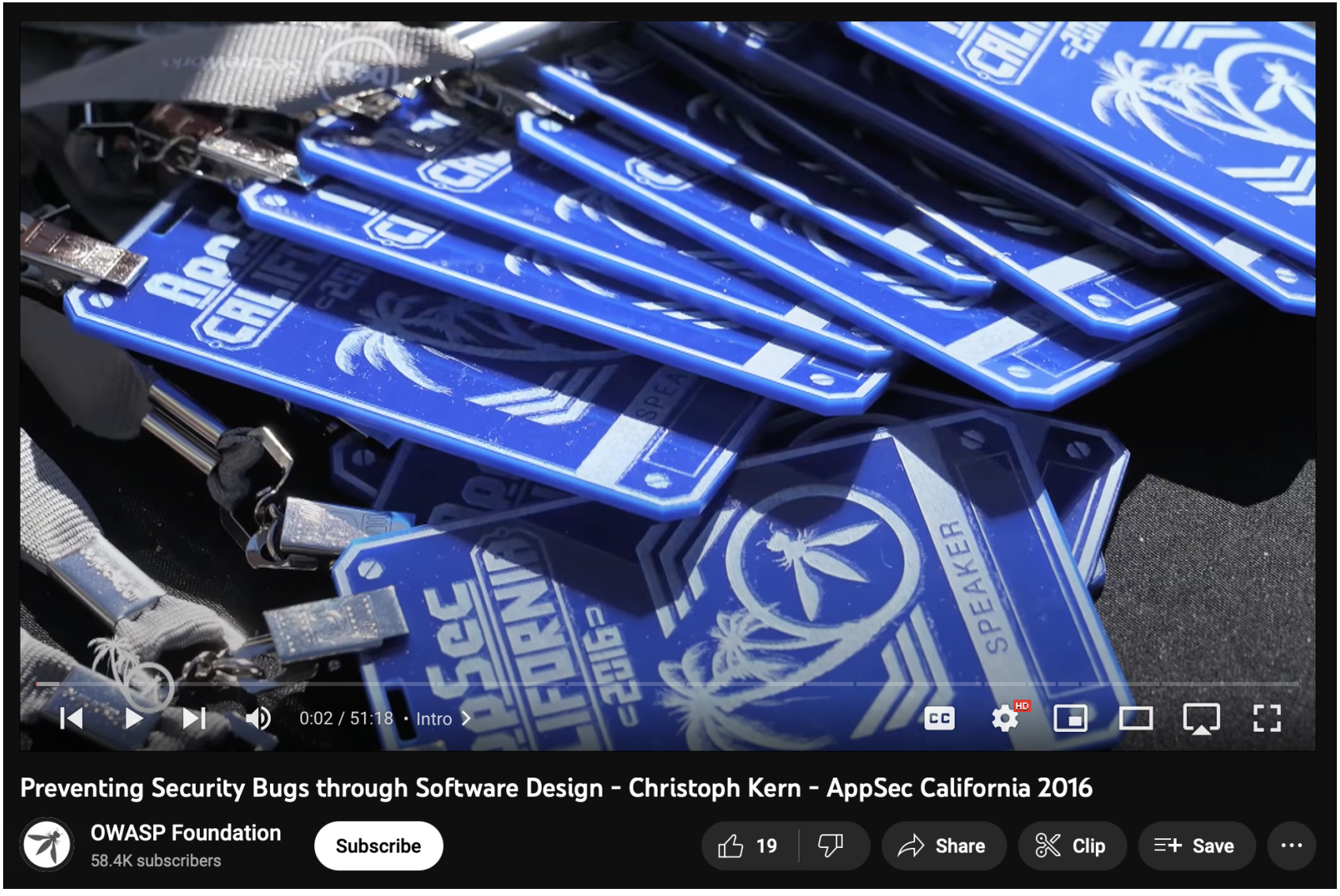

Figure 2: Preventing Security Bugs Through Software Design

Next Time

- Clean Code