Secure Software Design

Andey Robins

Spring 23 - Week 7

Design Patterns

Outline

- Design Patterns
- Design Paradigms
- “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.

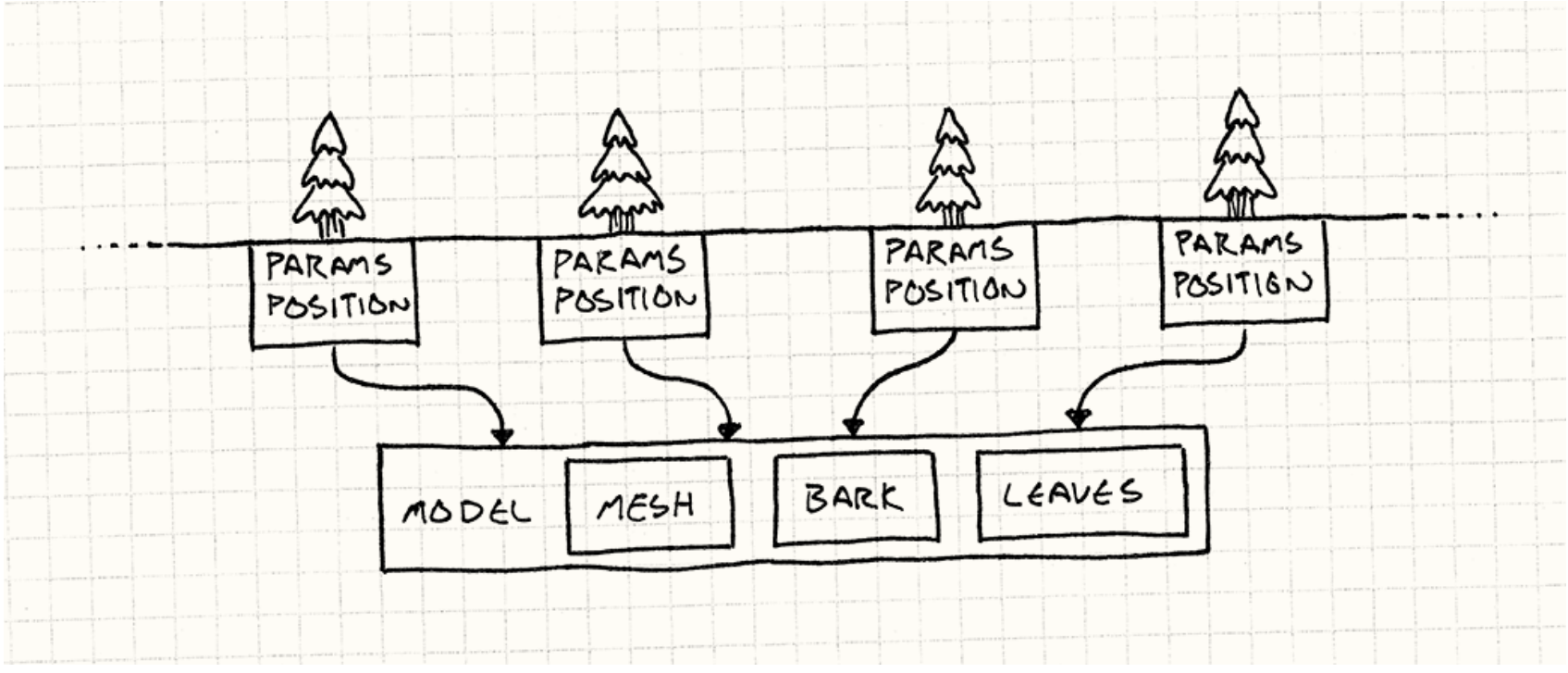


Figure 1: Using Flyweight to share Tree Assets

```
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

Ensure a class has one instance, and provide a global point of access to it.

"There are times when a class cannot perform correctly if there is more than one instance of it. The common case is when the class interacts with an external system that maintains its own global state"

Like a file system or a Database

DB Connection

```
type Database struct {
    Db *sql.DB
}

var instance *Database
var once sync.Once

func Connect() *Database {
    once.Do(func() {
        /// --- initialize and connect --- ///
    })

    return instance
}
```

Once Code

```
mustCreateDb := false
if _, err := os.Stat("dd.db"); err != nil {
    mustCreateDb = true
}

database, err := sql.Open("sqlite3", "dd.db")
if err != nil {
    log.Fatalf("Error connecting to database: %v", err)
}

/// --- CONT --- ///
```

```
/// --- CONT --- ///
if mustCreateDb {
    if command, err := os.ReadFile("init.sql");
        err == nil {
        _, err := database.Exec(string(command))
        if err != nil {
            log.Fatalf(
                "Error initializing database: %v", err
            )
        }
    } else {
        log.Fatalf("Error loading sqlite initial schema")
    }
}

/// NOTE THIS LINE
instance = &Database{Db: database}
```

```
var instance *Database


func Connect() *Database {
    once.Do(func() {
        /// --- initialize and connect --- ///
    })

    return instance // this is what makes it singleton
}
```

Benefits of Singleton

- The instance isn't created if nobody uses it
- It's initialized at runtime
- **JIT interface**
- It can be used as a polymorphic object

Problems of Singleton

- It's a global variable/system/state
- It encourages coupling
- It's not concurrency friendly

State Pattern

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class

```
interface State {
    entry: () => (),
    tick: (input: object): any => (),
    exit: () => (),
}
```

```
class IdleState implements State {
    entry() {
        this.setGraphics(IDLE);
    }

    tick(input: string) {
        if (
            input === "left" ||
            input === "right"
        ) {
            return new RunState();
        }
        return undefined;
    }

    exit() {}
}
```

```
class RunState {
    entry() {
        this.setGraphics(RUNNING);
    }

    tick(input: string) {
        if (input === "left") {
            this.x -= 5;
        } else if (input === "right") {
            this.x += 5;
        } else {
            return new IdleState();
        }
        return undefined;
    }

    exit() {
        this.playSound(SLIDE_STOP);
    }
}
```

The State Machine

```
class Player {
    activeState: State

    constructor() {
        this.activeState = new IdleState();
    }

    tick(input: object) {
        let nextOpt = this.activeState.tick(input);
        if (nextOpt) {
            this.activeState.exit();
            this.activeState = nextOpt;
            this.activeState.entry();
        }
    }
}
```

Limitations of State and State Machines

1. State machines are only so expressive
2. State counts explode quickly
3. There's no concept of "previous state" baked in

Possible Solutions

1. Concurrent states
2. Hierarchical states
3. PDA
4. Don't use state?

Design Paradigms

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")
}
```

Inheritance vs Composition

Inheritance is all about the "is a" relationship.

Composotion is all about the "has a" relationship.

A Jeep *is a* car, but it *has an* engine, wheels, and radio.

Inheritance

```
interface Car {
    mpg: number,
    fuel: number,
    drive: () => (),
    gasUp: () => (),
}
```

Jeep

```
class Jeep implements Car {
    mpg: number;
    fuel: number;

    constructor() {
        this.mpg = 14;
        gasUp();
    }

    drive() {
        move(this.mpg * 0.1);
        fuel -= 0.1;
    }

    gasUp() {
        fuel = 18.0;
    }
}
```

Tesla

```
class Jeep implements Car {
    mpg: number; // ????????

    /// --- SNIP --- ///
}
```

Composition

```
interface Motor {
    charge: number
    getPower: () => number,
}

interface Engine {
    fuel: number,
    getPower: () => number,
}
```

Jeep

```
class Jeep {
    fuel: number;
    engine: Engine;

    drive() {
        x, y += engine.getPower();
        fuel -= 0.1;
    }
}
```

Tesla

```
class Tesla implements Vehicle {
    charge: number;
    motor: Motor

    drive_electrically() {
        x, y += motor.getPower();
        charge -= 0.2;
    }
}
```

How do we drive?

We coul always add an `isElectric` method.

```
cars.forEach((car) => {
    if (car.isElectric) {
        car.drive_electrically();
    } else {
        car.drive();
    }
})
```

But what about nuclear powered cars?

Both Inheritance and Composition

```
interface Vehicle {
    move: (() => number) => (),
}

interface Motor {
    charge: number
    getPower: () => number,
}

interface Engine {
    fuel: number,
    getPower: () => number,
}
```

Jeep

```
class Jeep implements Vehicle {
    fuel: number;
    engine: Engine;

    move() {
        x, y += engine.getPower();
        fuel -= 0.1;
    }
}
```

Tesla

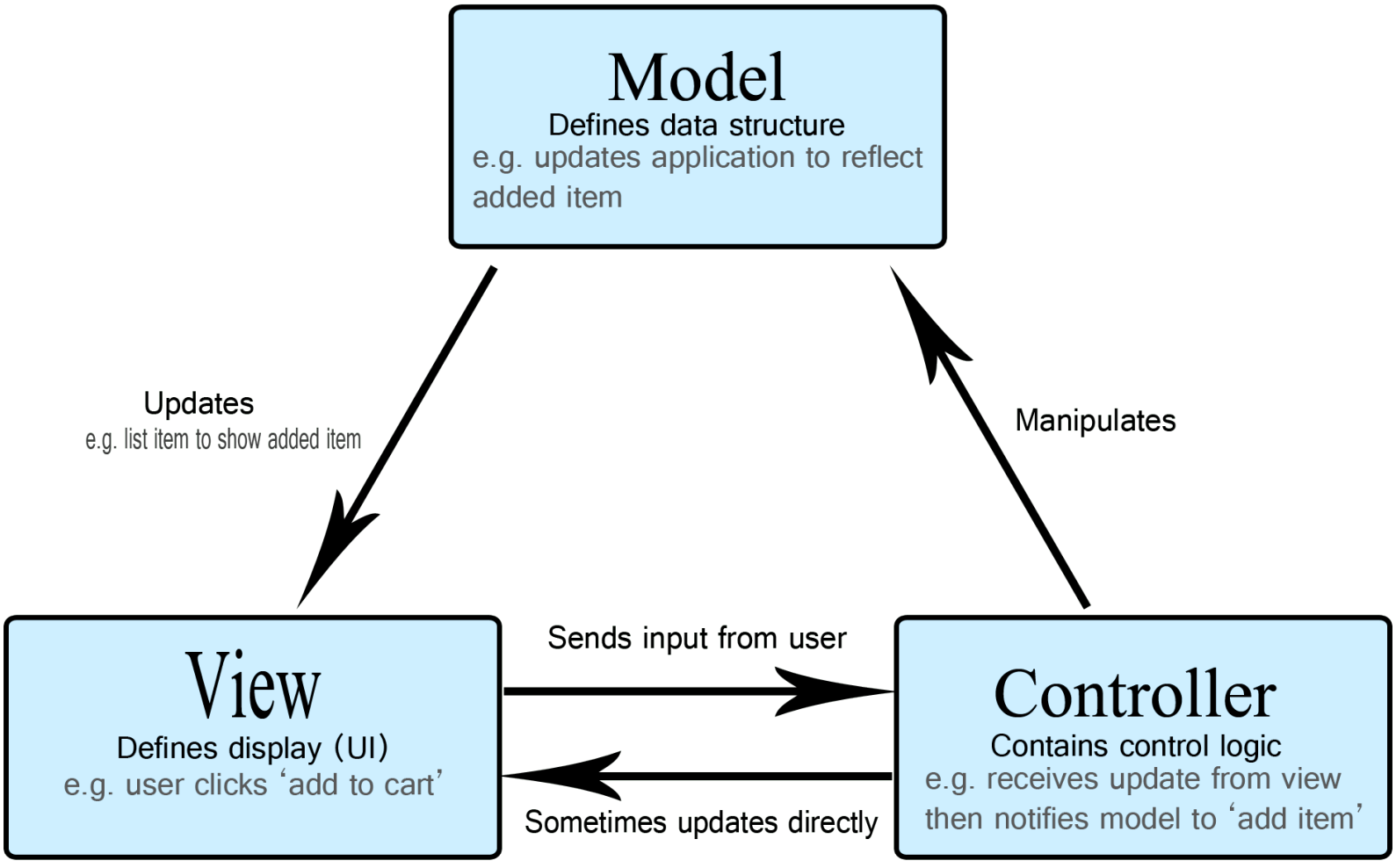
```
class Tesla implements Vehicle {
    charge: number;
    motor: Motor

    move() {
        x, y += motor.getPower();
        charge -= 0.2;
    }
}
```

Takeaways

1. Composition allows for more granular/specific behavior
2. Inheritance can lead to "unused" fields and methods
3. Composition can lead to difficulty "combining" different types of things
4. The best option is probably somewhere in the middle if OOP makes sense

MVC


Model
Defines data structure
e.g. updates application to reflect added item
Updates
e.g. list item to show added item
Manipulates
View
Defines display (UI)
e.g. user clicks 'add to cart'
Sends input from user
Sometimes updates directly
Controller
Contains control logic
e.g. receives update from view then notifies model to 'add item'

Model-View-Controller Principles

1. Model = data
2. View = visuals
3. Controller = logic

(i.e. Frontend, Backend, DB)

Model

The model defines what data our application contains. When the state of the data changes, it will notify the view (subscriber pattern?).

View

The view defines how data is displayed and how input will be accepted from the user. When it receives input, it notifies the controller.

Controller

The controller contains the logic which operates on the input. It then updates or modifies the model.

ECS

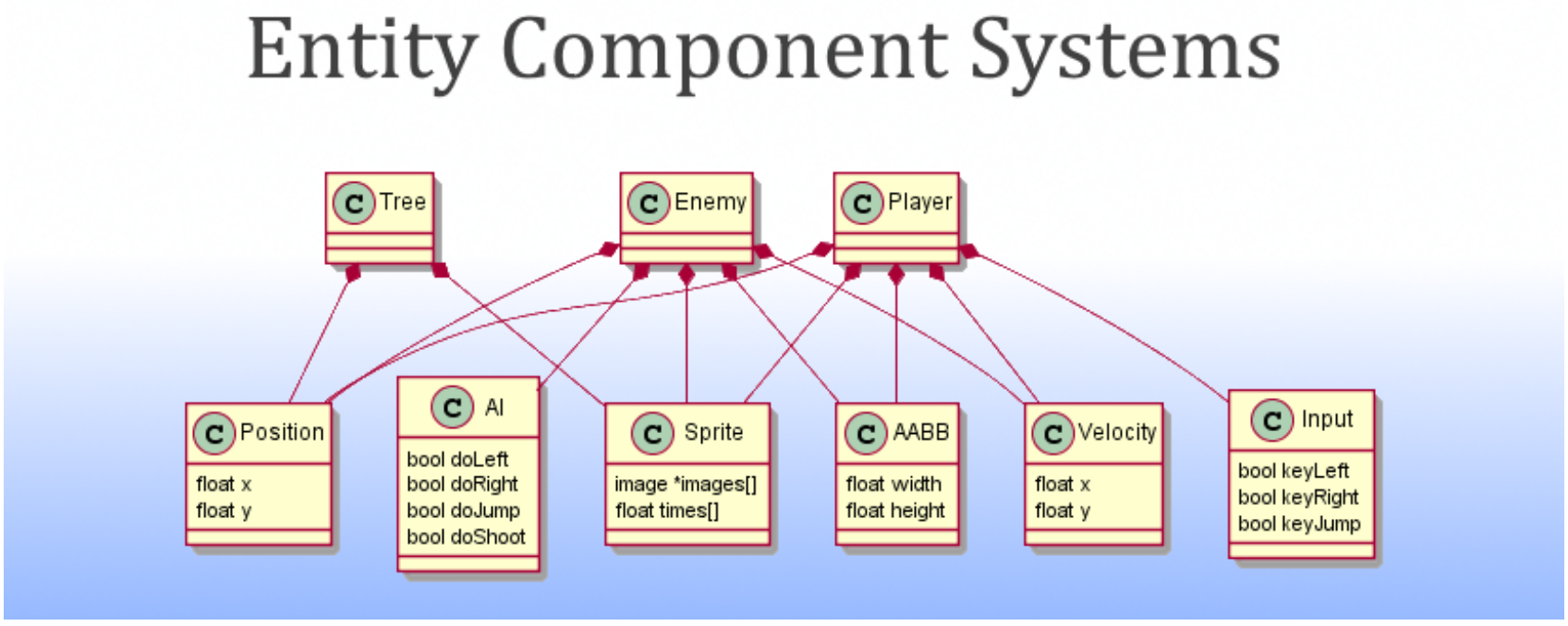


Figure 3: ECS Composition Patterns

Entity-Component-System Principles

1. Entity = uid
2. Component = data
3. System = behavior

Entity

Entities are just unique identifiers. These identifiers can then be correlated with various components (or more accurately components can be assigned to entities).

Component

Components are datatypes. They can have additional data or only be a “tag.”

System

Systems are behavior defined for entities with specific components.

Questions

Friday “Guest” Lecture

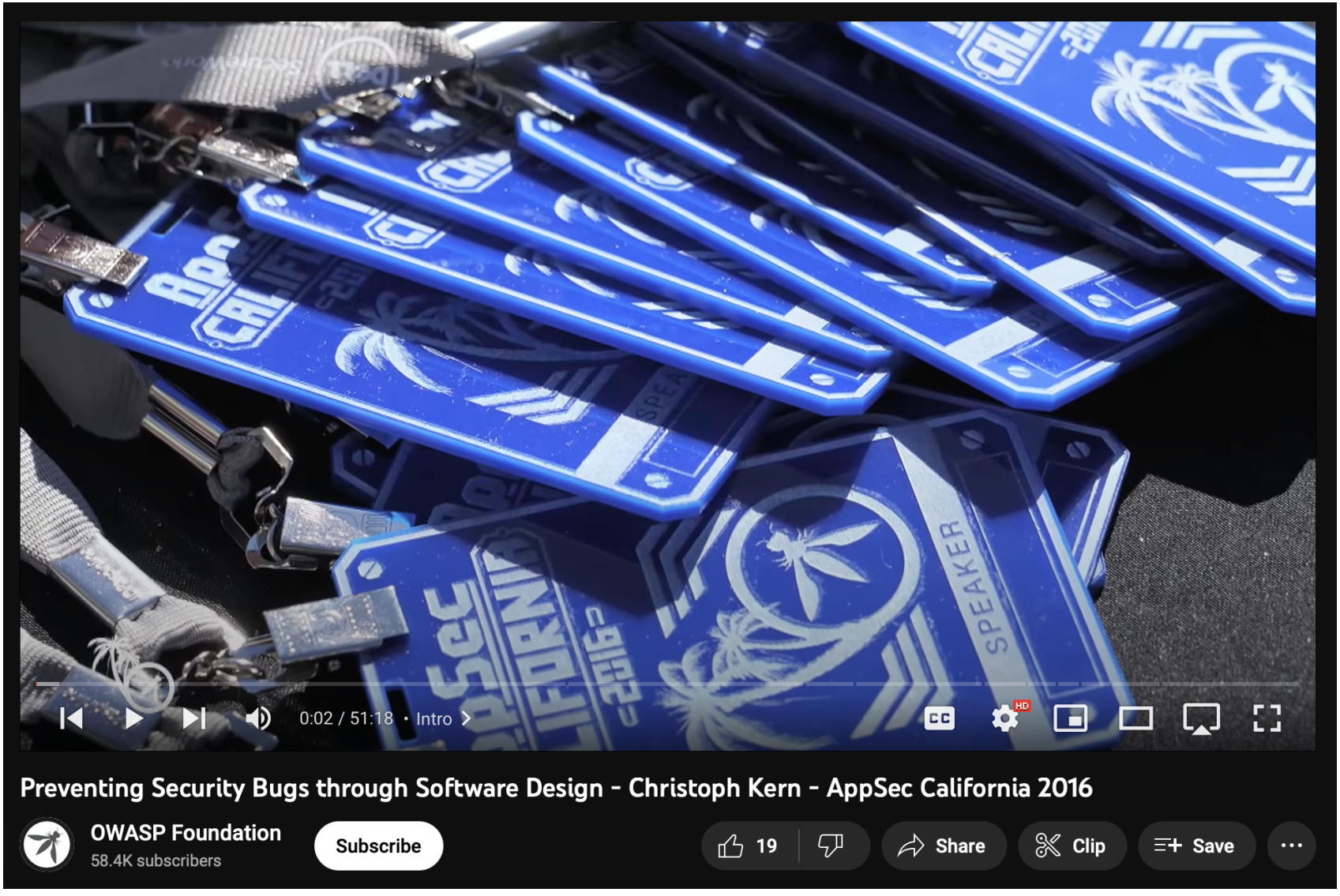


Figure 4: Preventing Security Bugs Through Software Design

Next Time

- Clean Code