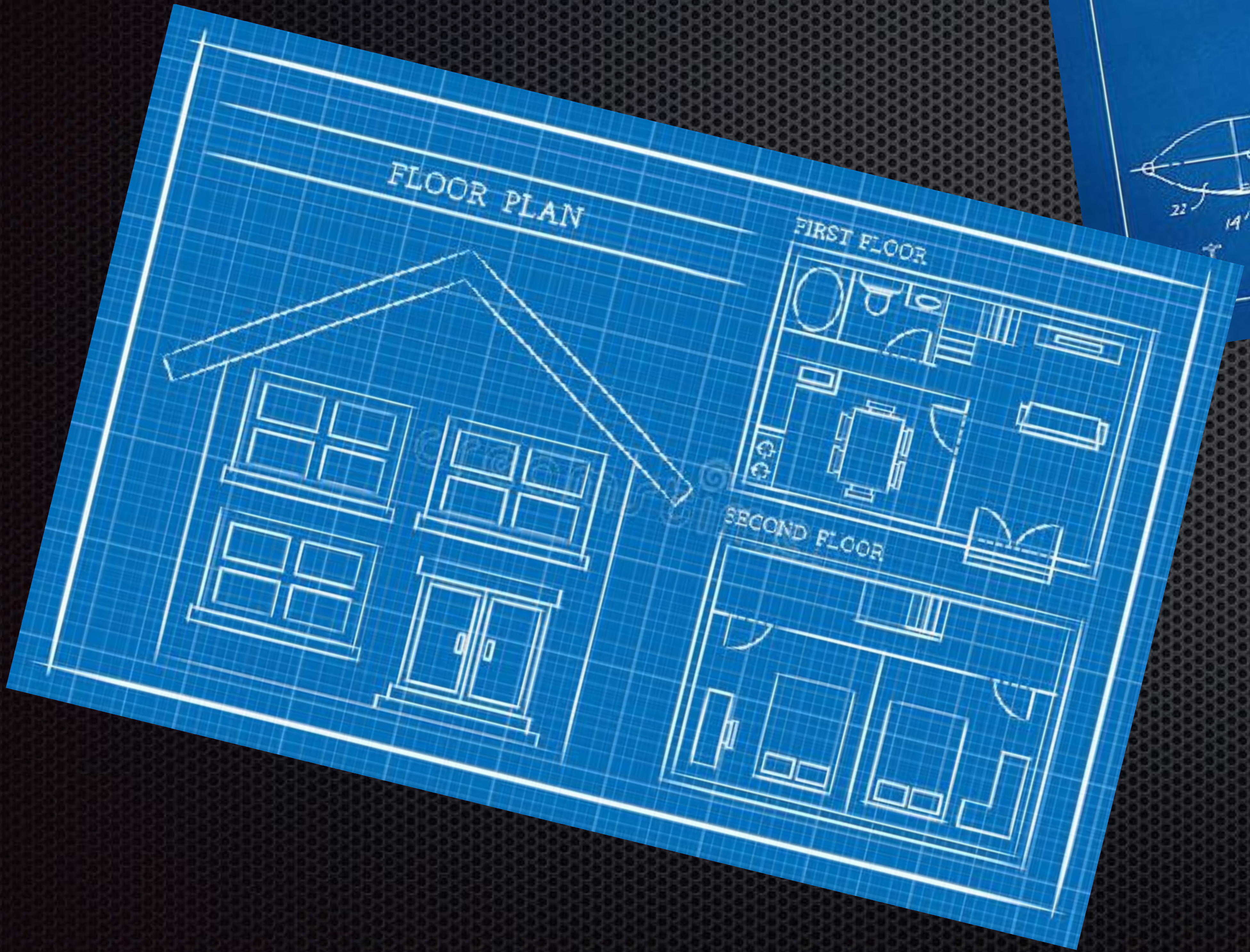
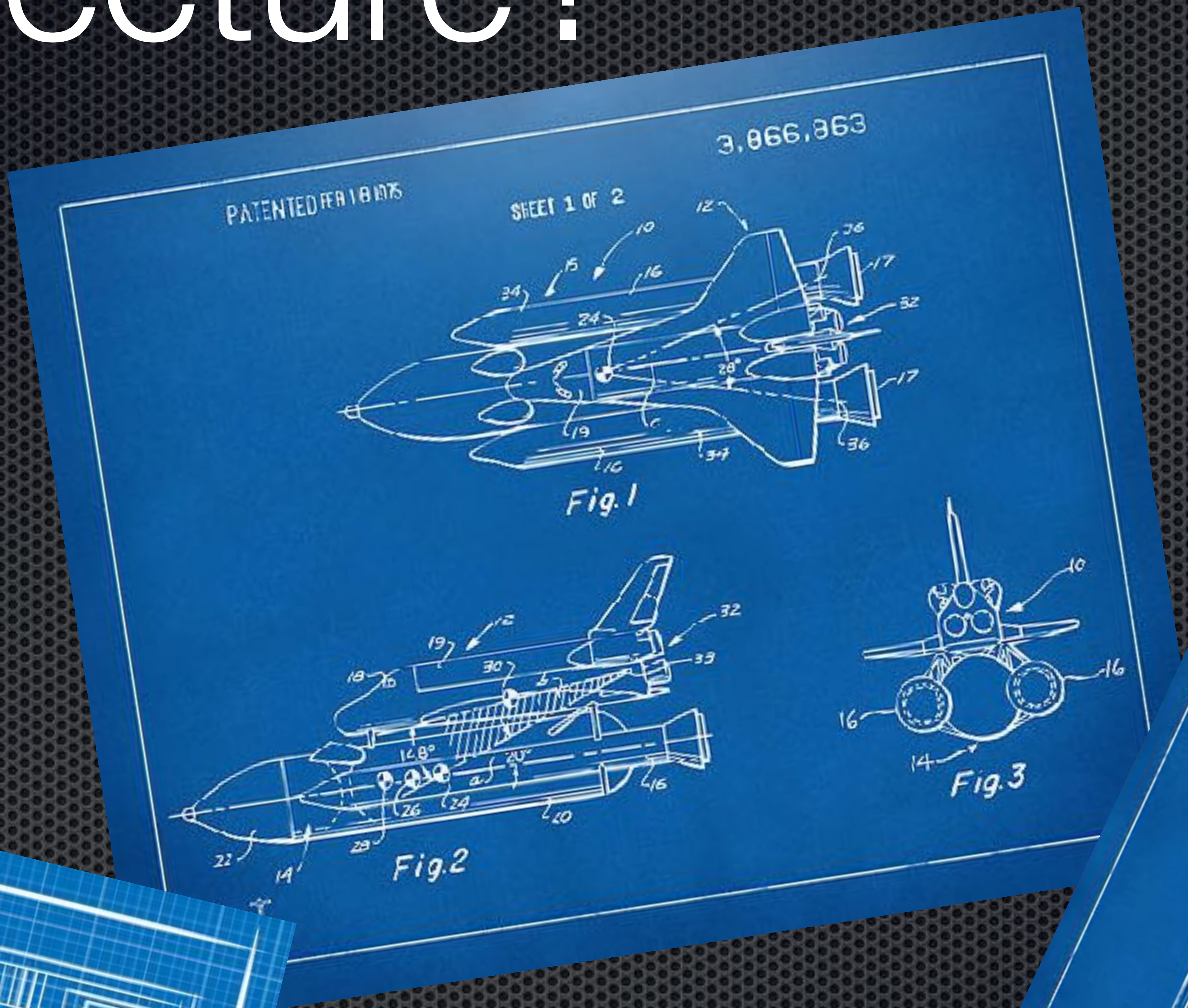
Clean Architecture

Software Structure and Design

What Is Architecture?





“The goal of software architecture is to minimize the human resources required to build and maintain the required system.”

Uncle Bob

Design Principles (SOLID)

- Single Responsibility Principle
- Open-Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Single Responsibility Principle

- A module should
  - have one, and only one, reason to change
  - be responsible to one, and only one, user or stakeholder

Single Responsibility Principle

```
class UserSettings {
    constructor(user) {
        this.user = user;
    }

    changeSettings(settings) {
        if (this.verifyCredentials()) {
            // ...
        }
    }

    verifyCredentials() {
        // ...
    }
}
```

```
class UserAuth {
    constructor(user) {
        this.user = user;
    }

    verifyCredentials() {
        // ...
    }
}


class UserSettings {
    constructor(user) {
        this.user = user;
        this.auth = new UserAuth(user);
    }

    changeSettings(settings) {
        if (this.auth.verifyCredentials()) {
            // ...
        }
    }
}
```

Open-Closed Principle

- A software artifact should be
  - open for extension
  - closed for modification

Open-Closed Principle

```
function makeAjaxCall(url) {
    // request and return promise
}

function makeHttpCall(url) {
    // request and return promise
}

class HttpRequester {
    constructor(environment) {
        this.environment = environment;
    }

    fetch(url) {
        if (this.environment === 'browser') {
            return makeAjaxCall(url).then((response) => {
                // transform response and return
            });
        } else if (this.environment === 'node') {
            return makeHttpCall(url).then((response) => {
                // transform response and return
            });
        }
    }
}
```

```
class AjaxAdapter extends Adapter {
    constructor() {
        super();
        this.name = 'ajaxAdapter';
    }

    request(url) {
        // request and return promise
    }
}

class NodeAdapter extends Adapter {
    constructor() {
        super();
        this.name = 'nodeAdapter';
    }

    request(url) {
        // request and return promise
    }
}

class HttpRequester {
    constructor(adapter) {
        this.adapter = adapter;
    }

    fetch(url) {
        return this.adapter.request(url)
            .then((response) => {
             // transform response and return
            });
    }
}
```

Liskov Substitution Principle

- “If for each object ***o1*** of type <S> there is an object ***o2*** of type <T> such that for all programs **P** defined in terms of <T>, the behavior of **P** is unchanged when ***o1*** is substituted for ***o2*** then <S> is a subtype of <T>.”
- “Barbara Liskov, “Data Abstraction and Hierarchy,” SIGPLAN Notices 23, 5 (May 1988).”

Liskov Substitution Principle

```
class Rectangle {
    constructor() {
        this.width = 0;
        this.height = 0;
    }

    render(area) {}

    setWidth(width) {this.width = width; }

    setHeight(height) {this.height = height;}

    getArea() {
        return this.width * this.height;
    }
}

class Square extends Rectangle {
    setWidth(width) {
        this.width = width;
        this.height = width;
    }

    setHeight(height) {
        this.width = height;
        this.height = height;
    }
}

function renderLargeRectangles(rectangles) {
    rectangles.forEach((rectangle) => {
        rectangle.setWidth(4);
        rectangle.setHeight(5);
        const area = rectangle.getArea(); // 25 / 20.
        rectangle.render(area);
    });
}

const rectangles = [new Rectangle(), new Rectangle(), new Square()];
renderLargeRectangles(rectangles);
```

```
class Shape {
    setColor(color) {
        // ...
    }

    render(area) {
        // ...
    }
}

class Rectangle extends Shape {
    constructor(width, height) {
        super();
        this.width = width;
        this.height = height;
    }

    getArea() {
        return this.width * this.height;
    }
}

class Square extends Shape {
    constructor(length) {
        super();
        this.length = length;
    }

    getArea() {
        return this.length * this.length;
    }
}

function renderLargeShapes(shapes) {
    shapes.forEach((shape) => {
        const area = shape.getArea();
        shape.render(area);
    });
}

const shapes = [new Rectangle(4, 5), new Rectangle(4, 5), new Square(5)];
renderLargeShapes(shapes);
```

Interface Segregation Principle

- no client should be forced to depend on methods it does not use

```
class DOMTraverser {
    constructor(settings) {
        this.settings = settings;
        this.setup();
    }

    setup() {
        this.rootNode = this.settings.rootNode;
        this.settings.animationModule.setup();
    }

    traverse() {
        // ...
    }
}

const $ = new DOMTraverser({
    rootNode: document.getElementsByTagName('body'),
    animationModule() {}
    // ...
});
```

```
class DOMTraverser {
    constructor(settings) {
        this.settings = settings;
        this.options = settings.options;
        this.setup();
    }

    setup() {
        this.rootNode = this.settings.rootNode;
        this.setupOptions();
    }

    setupOptions() {
        if (this.options.animationModule) {
            // ...
        }
    }

    traverse() {
        // ...
    }
}

const $ = new DOMTraverser({
    rootNode: document.getElementsByTagName('body'),
    options: {
        animationModule() {}
    }
});
```

Dependency Inversion Principle

- High-level modules should not depend on low-level modules. Both should depend on abstractions
- Abstractions should not depend on details. Details should depend on abstractions

Dependency Inversion Principle

```
class InventoryRequester {
    constructor() {
        this.REQ_METHODS = ['HTTP'];
    }

    requestItem(item) {
        // ...
    }
}

class InventoryTracker {
    constructor(items) {
        this.items = items;

        // BAD: We have created a dependency on a specific request
implementation.
        // We should just have requestItems depend on a request method:
`request`
        this.requester = new InventoryRequester();
    }

    requestItems() {
        this.items.forEach((item) => {
            this.requester.requestItem(item);
        });
    }
}

const inventoryTracker = new InventoryTracker(['apples', 'bananas']);
inventoryTracker.requestItems();
```

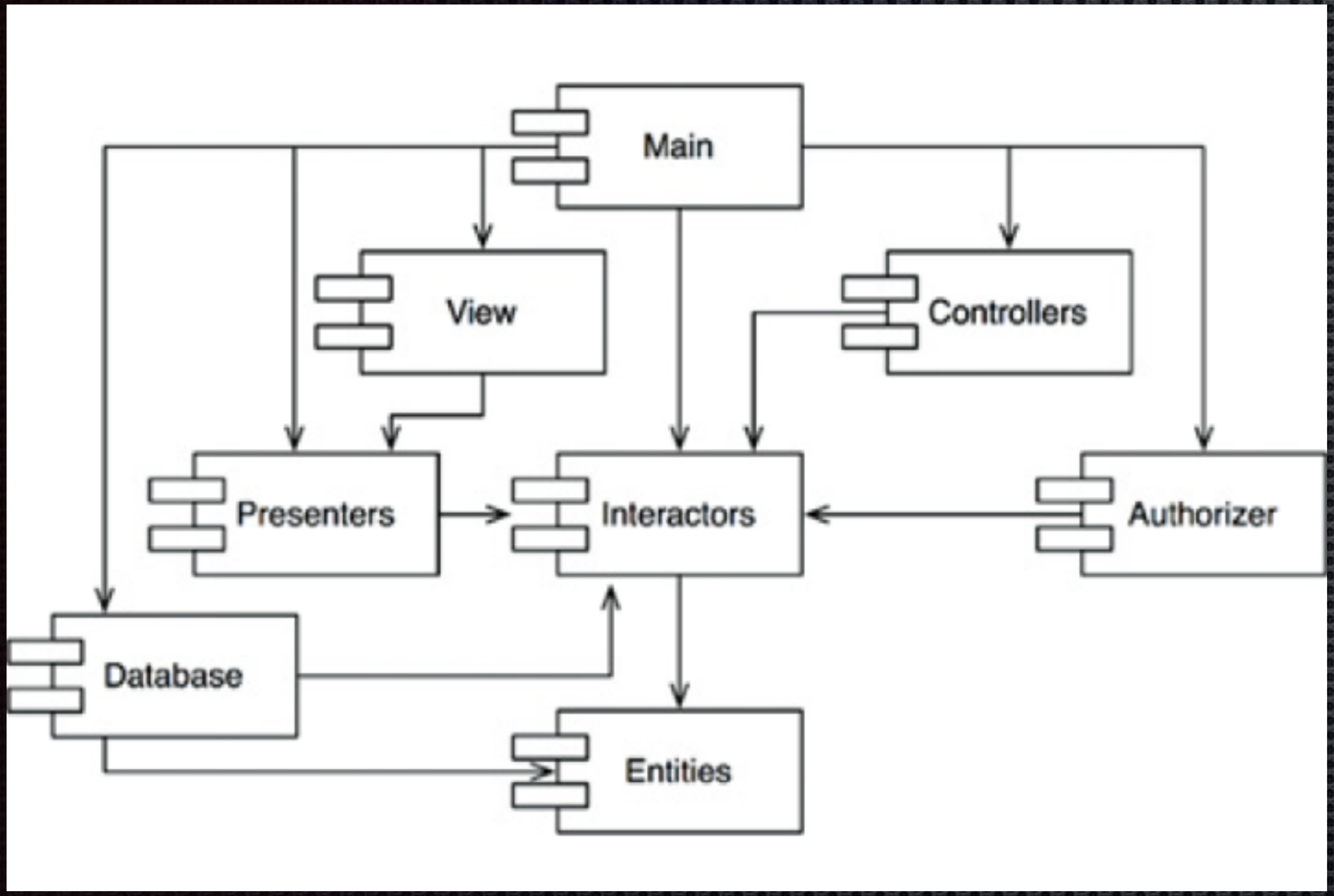
```
class InventoryTracker {
    constructor(items, requester) {
        this.items = items;
        this.requester = requester;
    }

    requestItems() {
        this.items.forEach((item) => {
            this.requester.requestItem(item);
        });
    }
}

class InventoryRequesterV1 {
    constructor() {
        this.REQ_METHODS = ['HTTP'];
    }

    requestItem(item) {
        // ...
    }
}

class InventoryRequesterV2 {
    constructor() {
        this.REQ_METHODS = ['WS'];
    }

    requestItem(item) {
        // ...
    }
}

// By constructing our dependencies externally and injecting them, we can easily
// substitute our request module for a fancy new one that uses WebSockets.
const inventoryTracker = new InventoryTracker(['apples', 'bananas'], new InventoryRequesterV2());
inventoryTracker.requestItems();
```

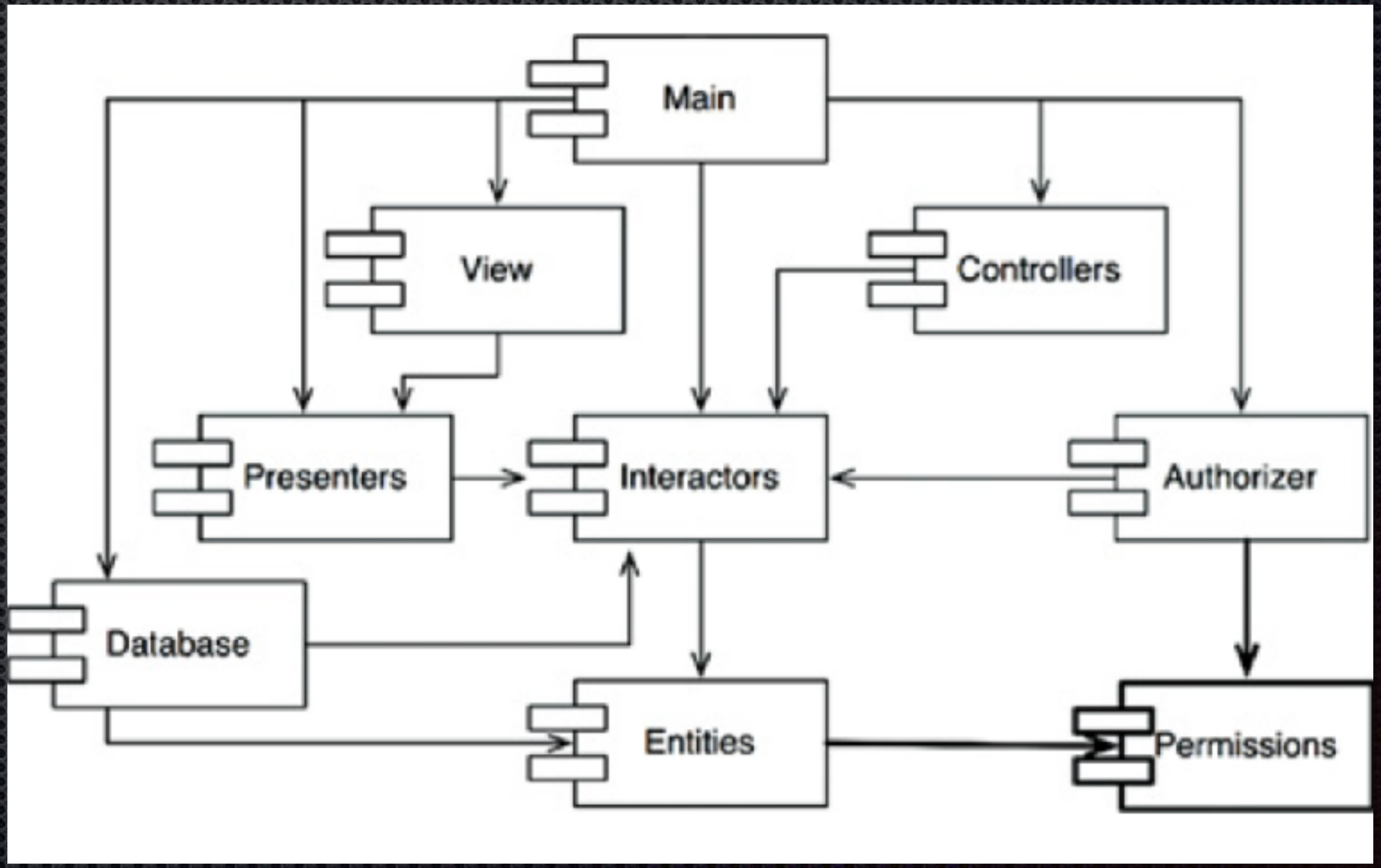
Component Principles

- Acyclic Dependencies Principle
- Stable Dependencies Principle
- Stable Abstractions Principle

Acyclic Dependencies Principle

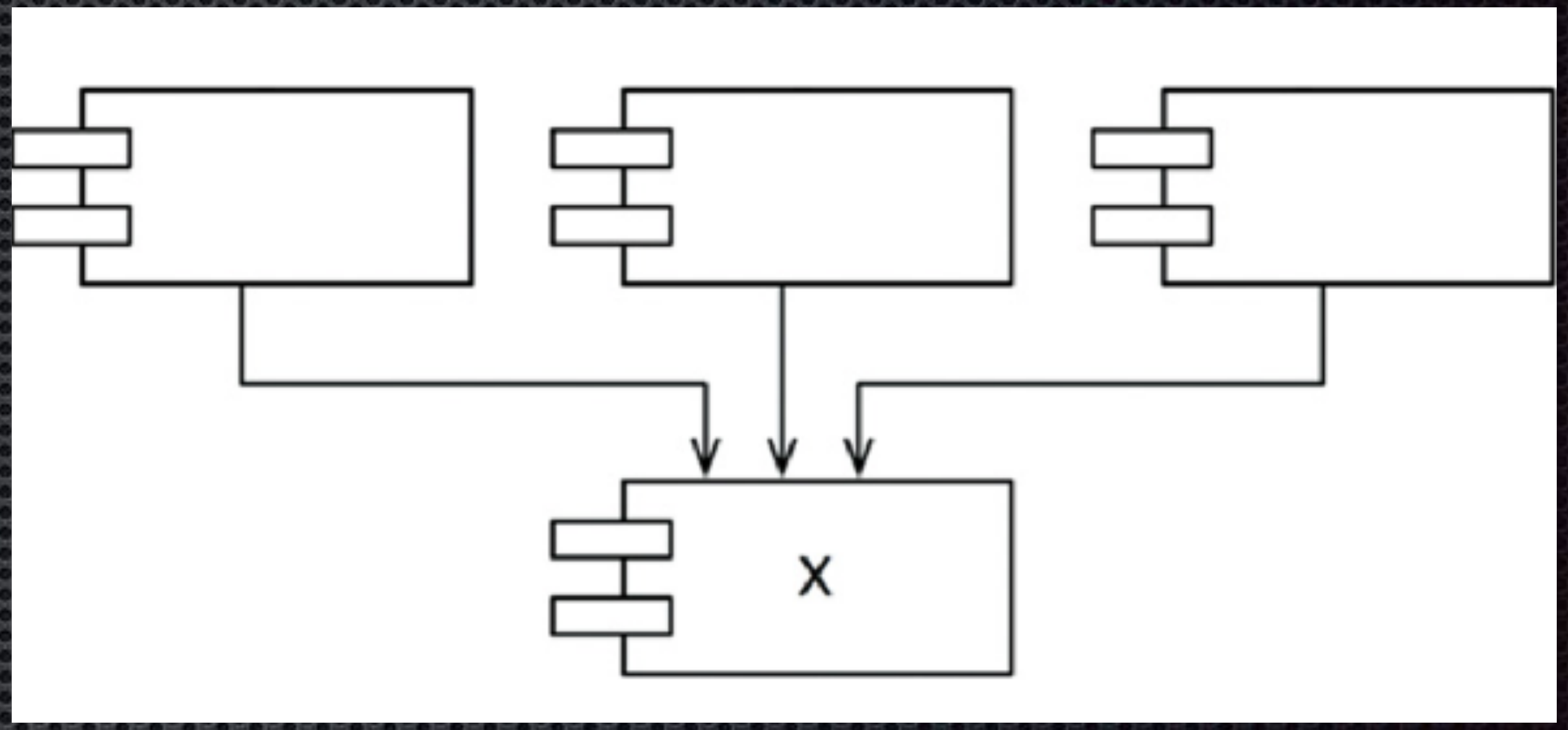
- Allowed no cycles in the component dependency graph



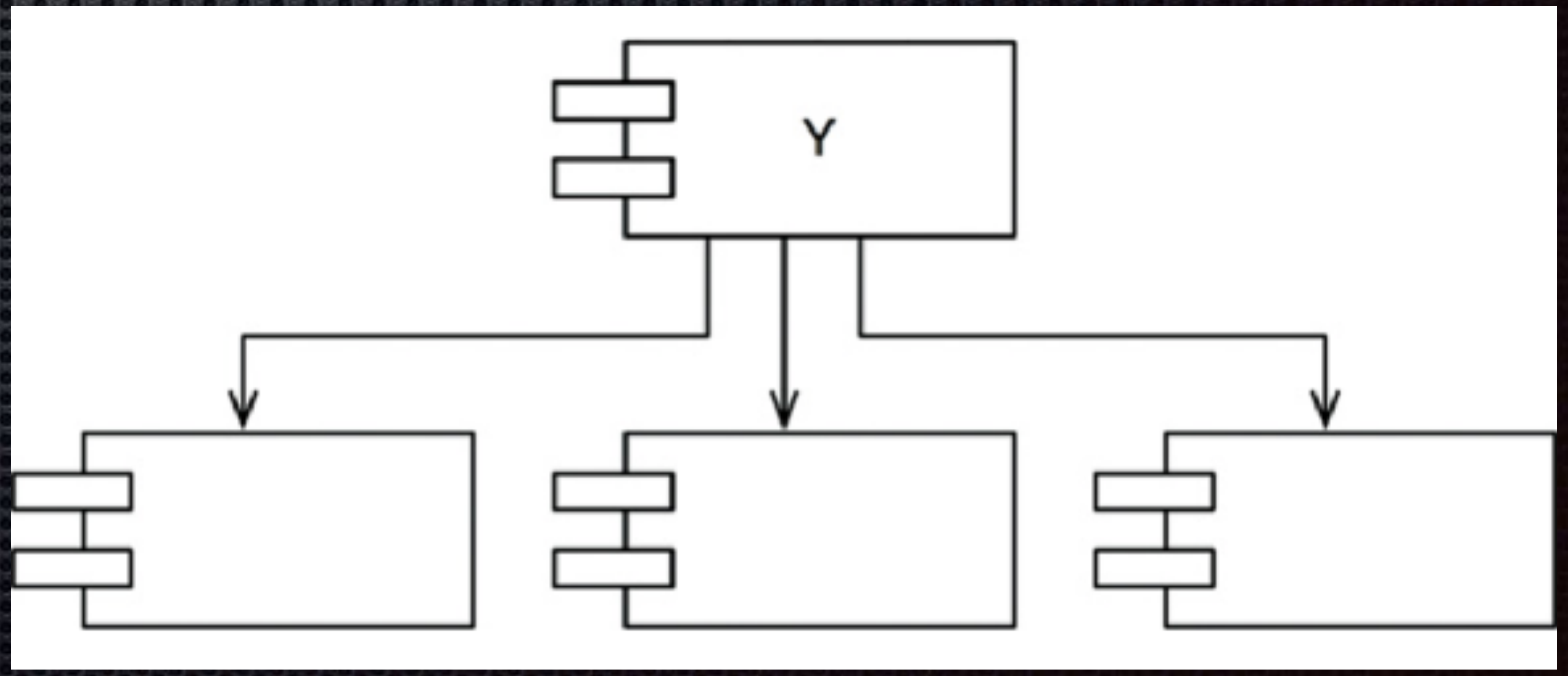


Stable Dependencies Principle

- Depend in the direction of stability
- Fan-in: incoming dependencies
- Fan-out: outgoing dependencies
- I: Instability = Fan-out / (Fan-in + Fan-out)



I = 0 / (0+3) = 0

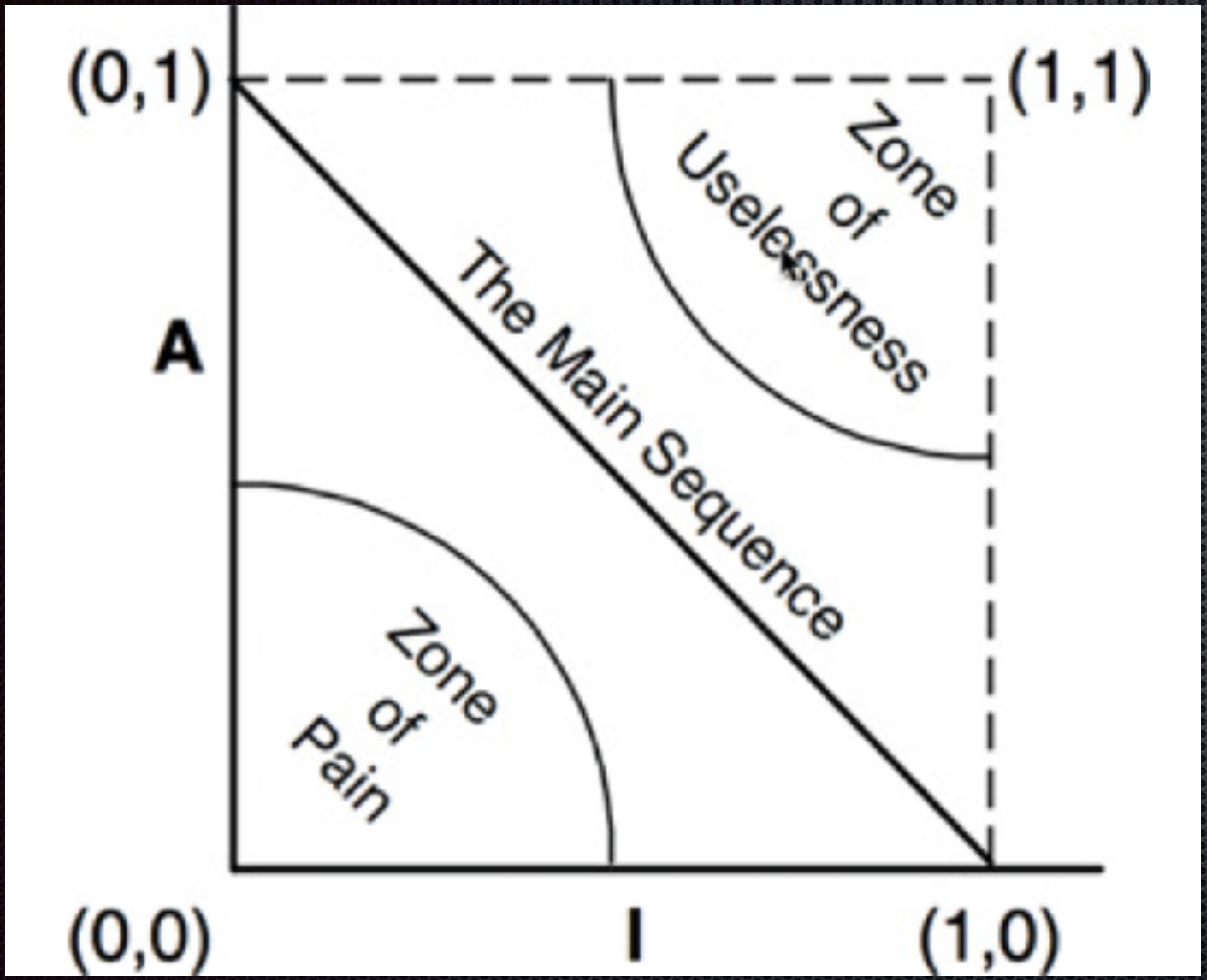


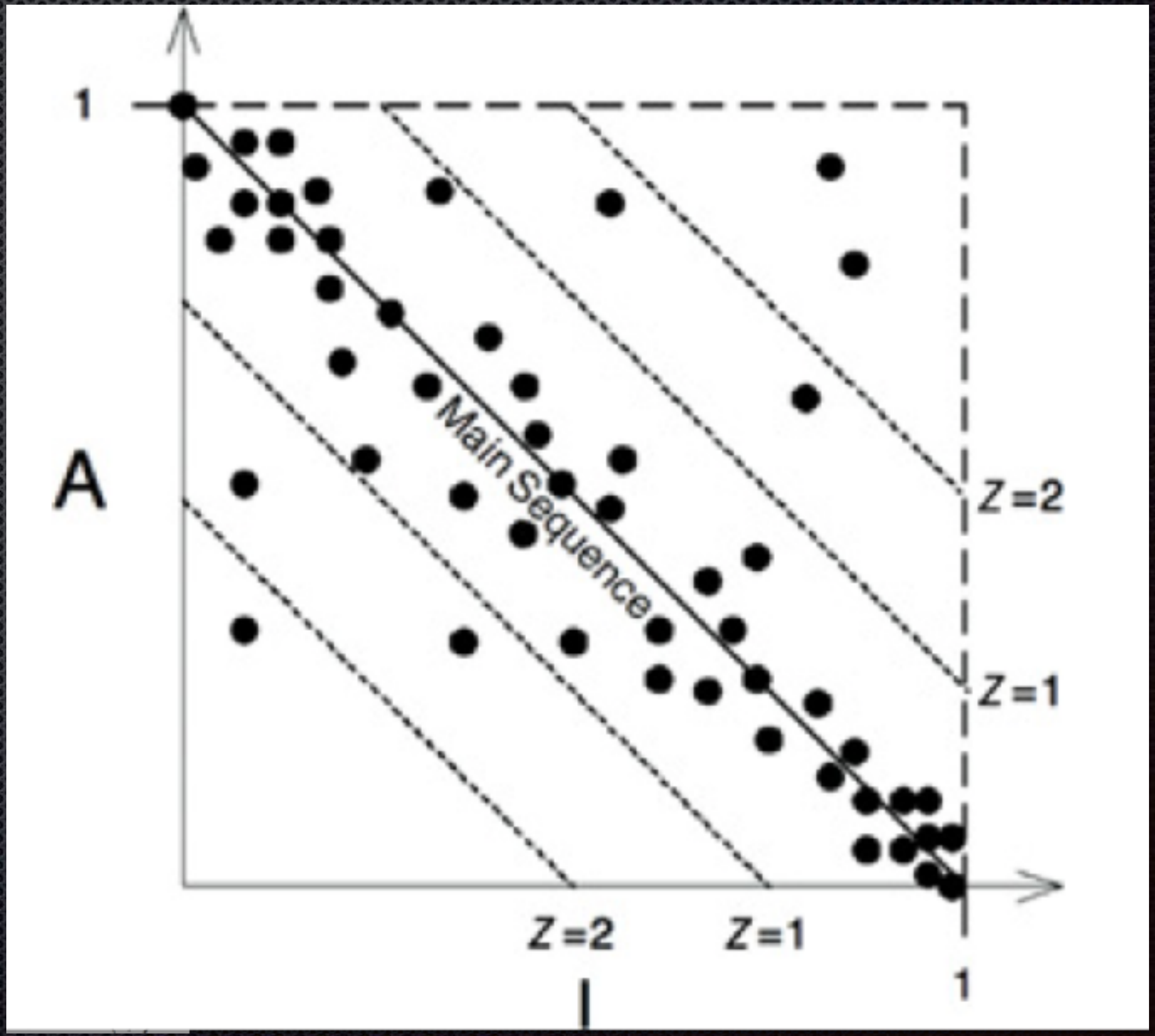
I = 3 / (0+3) = 1

Stable Abstractions Principle

- A component should be as abstract as it is stable
- Nc: The number of classes in the component
- Na: The number of abstract classes and interfaces in the component
- A: Abstractness. A = Na / Nc

Stable Abstractions Principle



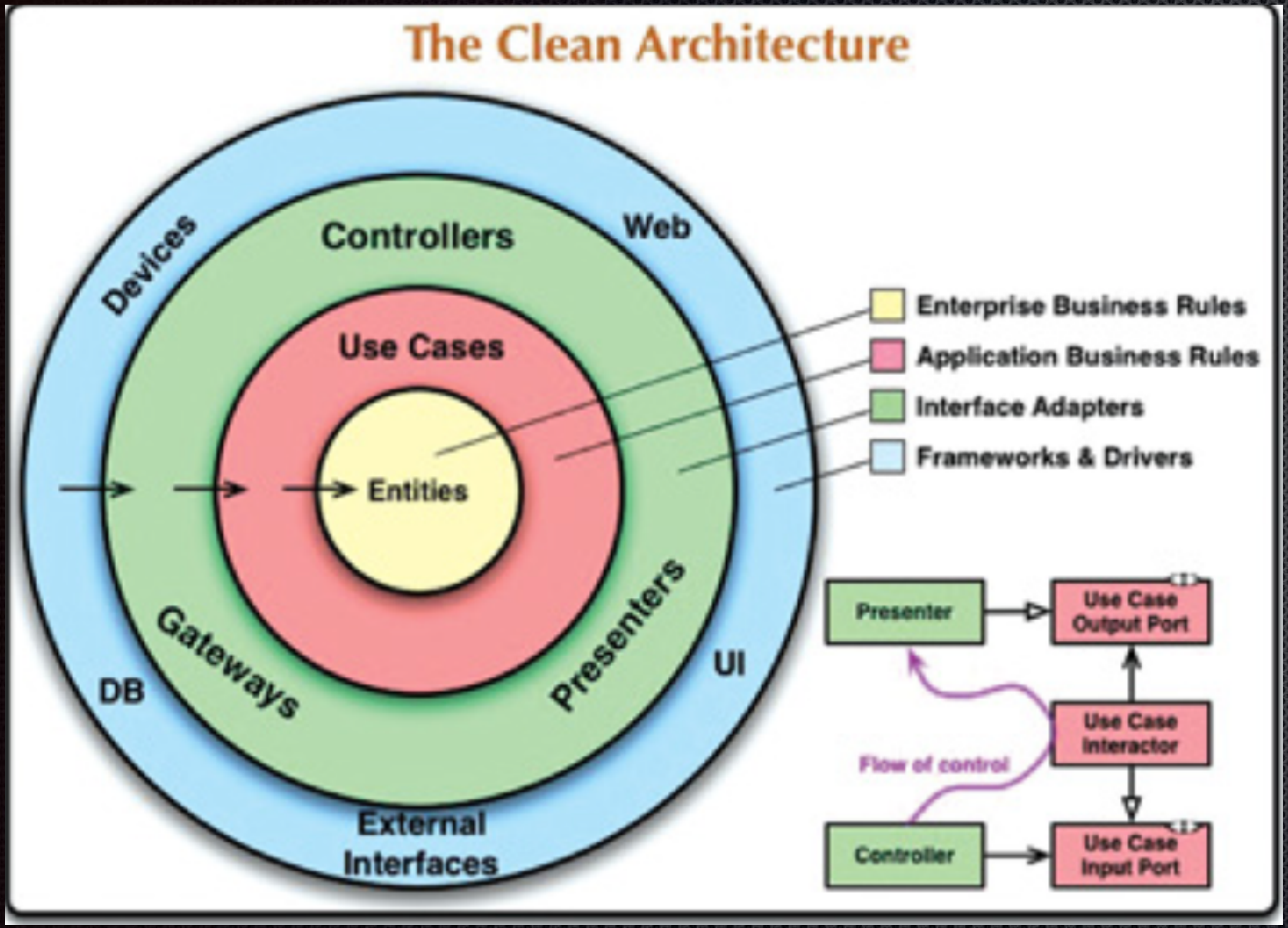


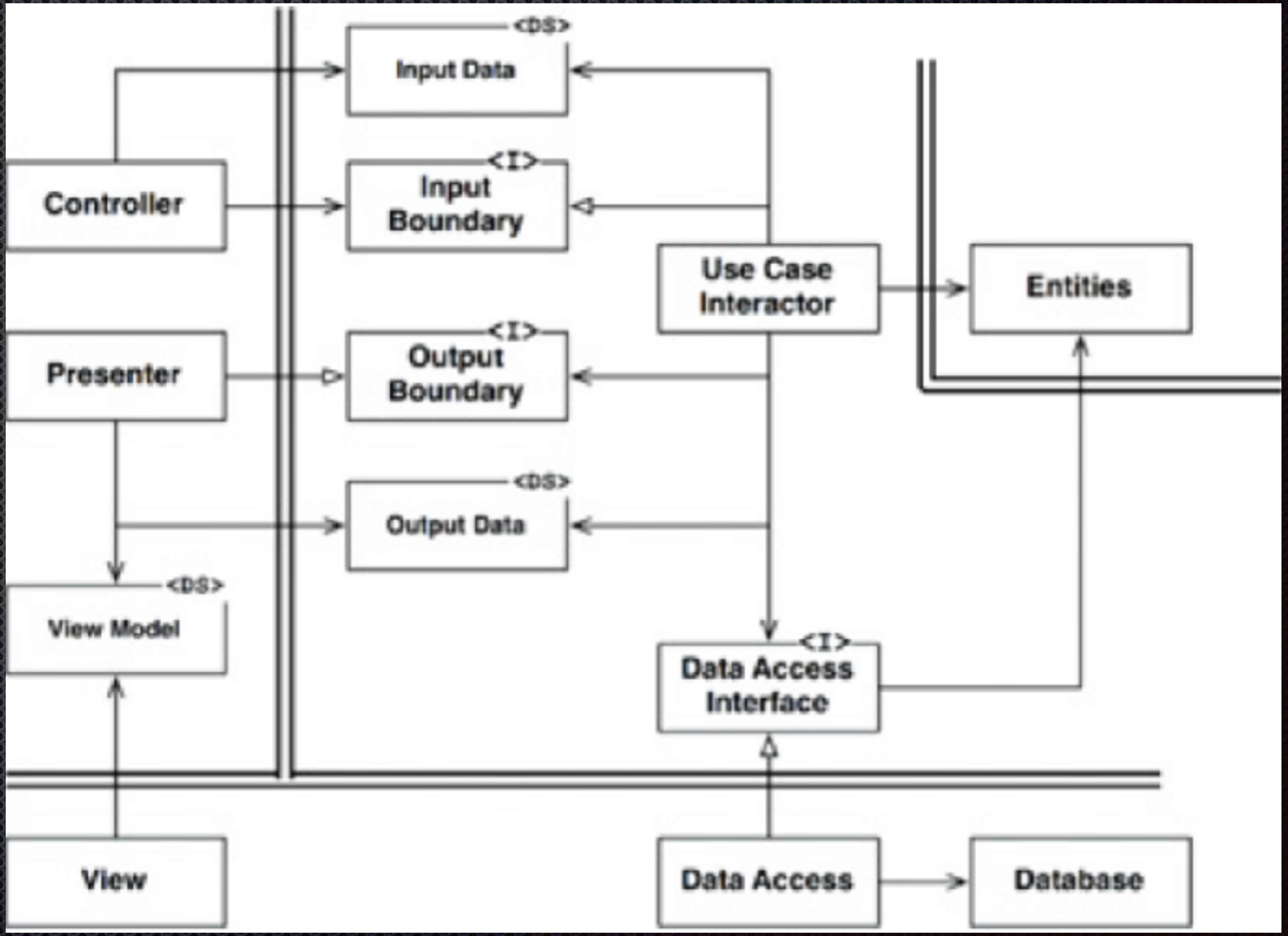
“A good architect maximizes the number of decisions not made.”

“The strategy is to leave as many options open as possible, for as long as possible.”

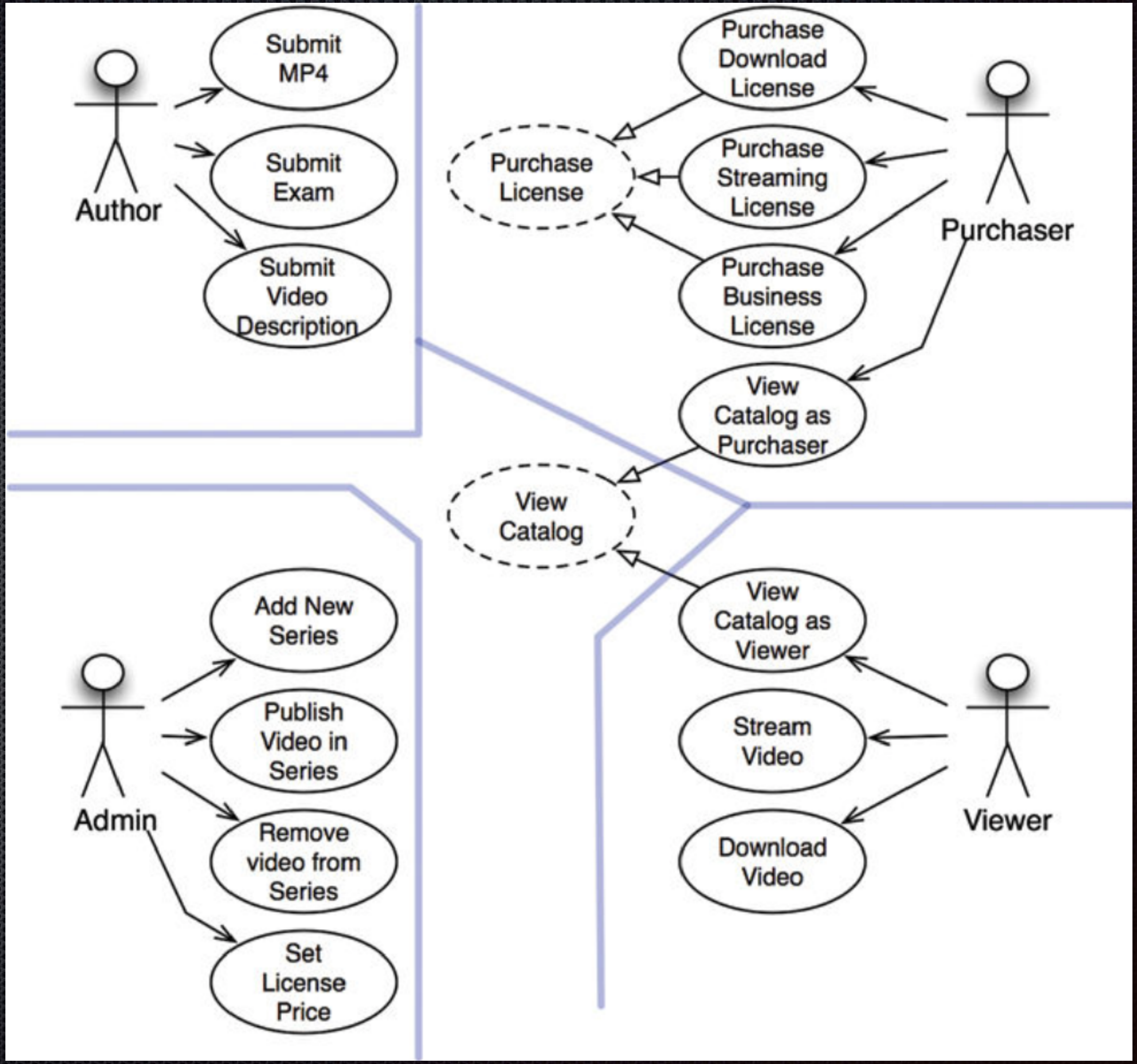
Uncle Bob

The Clean Architecture

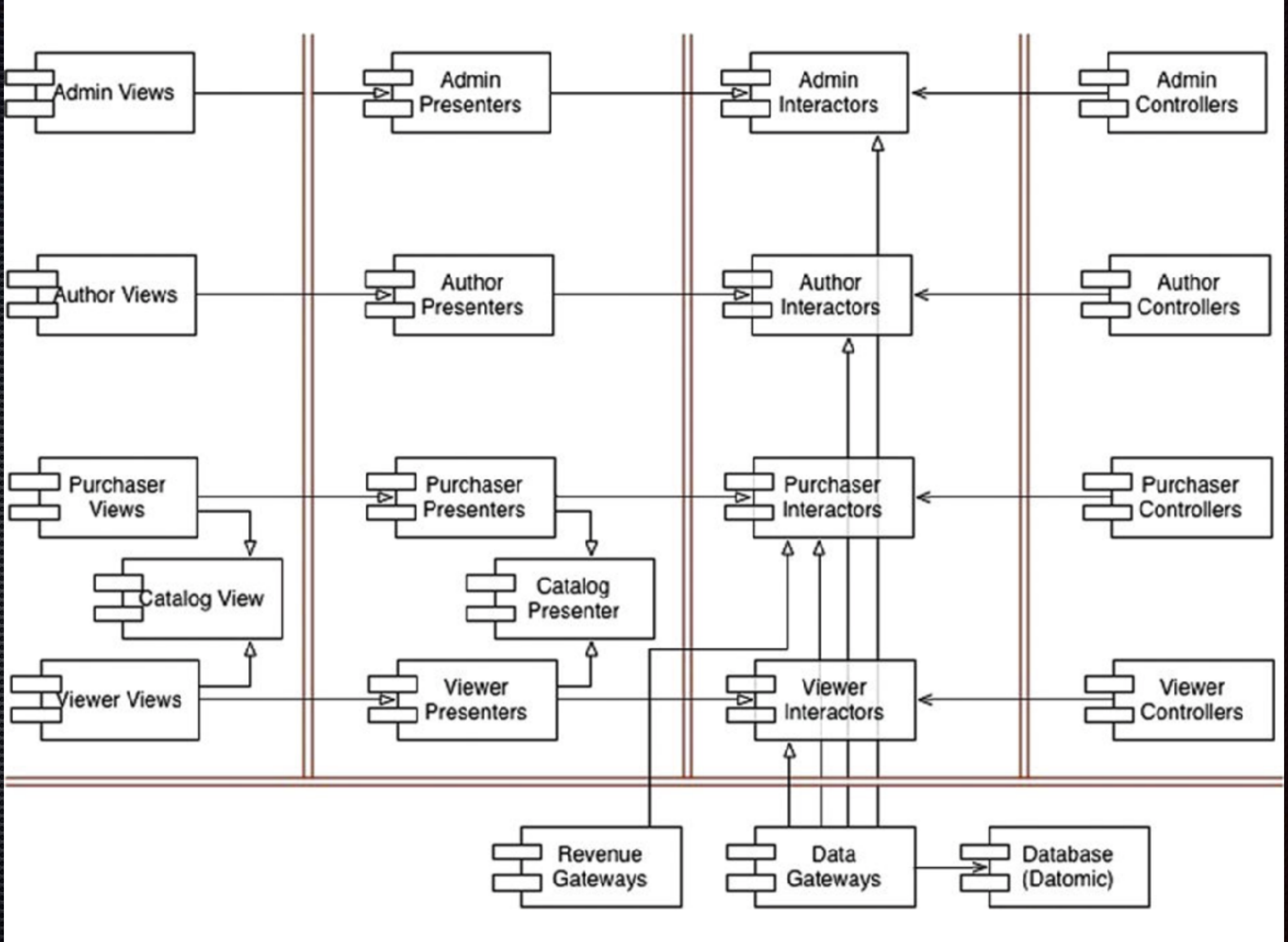




Case Study: Video Sales



Case Study: Video Sales



Thanks for your attention.

Any question?