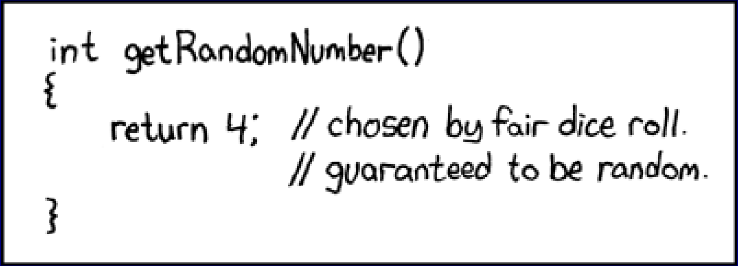
# SOLID Object Oriented Design

Justin Schiff

February 13, 2019

# SOLID Object Oriented Design

Welcome!



# On Design Thinking

Design is a philosophy and mentality of adapting to change.

- Change is the only constant
- Requirements change
- Technologies change
- Data changes

What is a good methodology to gracefully adapt to these changes?

- Over-engineering
- Under-engineering

**Tip**

Once well adapted to design thinking code reviews are a great place to apply the methodology.

# Different Design Methodologies

There are many philosophies of design.

### Functional Programming

- Isolating state upwards
- Making as many pure functions as possible without side effects.
- Lisp-based programming languages facilitate this design philosophy.

### Procedural Programming

- Basically what we all first do as programmers.
- Write what works, lines execute one after another.

### Object Oriented Design.

- Concepts can emerge from our program and express themselves as objects.
- Polymorphism removes branching.
- Objects have interfaces that expose their functionality.

# Mentality

Programming in these design philosophies is not following a set of rules. It's more a set of questions you ask yourself while you are programming. There is also some understanding of human psychology embedded in these principles.

- Duplication is cheaper than the wrong abstraction. (As opposed to DRY)
- An object in motion remains in motion.
- Make the change easy, then make the easy change.

`You see this code in your system:`

```
if (record.type == 'student') {
  return `${record.name} - ${record.grade}`;
} else if (record.type == 'witch') {
  return `${record.utf8Glyph} ${record.favoriteSpell}`;
} else if (record.type == 'teacher') {
  return `${record.title} ${record.name} (${record.roomNumber})`;
}
// The client contacts your PM and says we need to support 'janitor'
// as well. What are you going to do next?
```

# Mentality

Before you know it...

```
if (record.type == 'student') {
} else if (record.type == 'witch') {
} else if (record.type == 'janitor') {
} else if (record.type == 'staff' && record.building == 1) {
} else if (record.type == 'staff' && record.building == 2) {
} else if (record.type == 'buildingMaint') {
} else if (record.type == 'foodServices') {
} else if (record.type == 'teacher') {
}
// etc...
```

# SOLID

Ok so we've beaten around the bush a bit. Here they are:

### Principles

- <u>S</u> ingle Responsibility
- <u>O</u> pen/Closed Principle
- <u>L</u> iskov Substitution
- <u>I</u> nterface Segregation
- <u>D</u> ependency Inversion

# Single Responsibility

A class should have only one clear reason for why you would want to change it.

- Easiest way to demonstrate this is to think of the largest object in your system. This is called a God object. Most frequently this is User and the main domain object in your system.
- Classes that have 1 responsibility are easier to test (see my next talk :))
- Easier to change.
- Classes should be cohesive. Meaning code in the class should change together. Common patterns that have low cohesion are code lumped in `utils` files.

# Single Responsibility

```
class User extends BaseModel {
  //...attributes
  EMAIL_REGEX: /.../
  beforeCreate() {
    this.token = GenerateUniqueToken();
    this.save();
  }
  onSave() {
    if (!EMAIL_REGEX.test(this.email)) {
      // send validation.
    }
  }
  sendInvitation() {
    var token = this.token;
    var message = {
      subject: '',
      body: '...token...', // etc.
    }
    Mailer({smtp_settings: /*...*/})
  }
}
```

# Single Responsibility

```
class TokenableModel extends BaseModel {
  onSave() {this.token ||= GenerateUniqueToken();}
  toUnique() {return this.token;}
}
class User extends TokenableModel { // <--
  //...attributes
  EMAIL_REGEX: /regex/
  onSave() {
    if (!EMAIL_REGEX.test(this.email)) {
      // send validation.
    }
  }
  sendInvitation() {
    var token = this.toUnique(); // <--
    var message = {
      subject: '',
      body: '...token...', // etc.
    }
    Mailer({smtp_settings: /*...*/})
  }
}
```

# Single Responsibility

```
class TokenableModel extends BaseModel {
  onSave() {this.token ||= GenerateUniqueToken();}
  toUnique() {return this.token;}
}
class Invitation extends TokenableModel {
  user: foreignKey(),
  onCreate() {
    Mailer.sendInvitation(this)
  }
}
class User extends BaseModel { // <--
  //...attributes
  EMAIL_REGEX: /regex/
  onSave() {
    if (!EMAIL_REGEX.test(this.email)) {
      // send validation.
    }
  }
}
```

# Single Responsibility

```
class TokenableModel extends BaseModel {/*...*/}
class Invitation extends TokenableModel {/*...*/}
class Email {
  constructor(str) { this.str = str; }
  isValid() { return /regex/.test(this.str); }
  // lets you add functionality specifically related to emails here.
  domain() { this.string.split('@').slice(-1).pop(); }
}

class User extends BaseModel {
  //...attributes
  validations: [
    email: {
      validator: Email,
      message: 'You must enter a valid email.'
    }
  ]
}
```

# Open/Closed Principles

Code should be open for extension, and closed for modification.

- Able to change behavior without changing the code inside the class.

**Tip**

Generally achieved with dependency injection and polymorphism.

# Open/Closed Principles

```
// Example from before violates OCP
if (record.type == 'student') {
  return `${record.name} - ${record.grade}`;
} else if (record.type == 'witch') {
  return `${record.utf8Glyph} ${record.favoriteSpell}`;

  } else if (record.type == 'teacher') {
  return `${record.title} ${record.name} (${record.roomNumber})`;
}
```

```
// Follows OCP
class Student {
  toDisplayName() {return `${record.name} - ${record.grade}`;}
}
class Teacher { /*etc...*/}

// Now wherever this is just needs to send a message to the class.
displayName = record.toDisplayName();
```

# Open/Closed Principles

Another Example. . .

```
// Violates OCP
class User {
  sendUserPostedEvent(post) {
    NotificationService.sendNotifications(post);
    PostMailer.notifyAllUsers(post);
    post.textSubscribers.each(user => {
      TextService.sendPostUpdateText(user);
    });
  }
}
```

**Note**

This class is not open for extension. If we want to change any of the functionality we must modify the code in this class.

# Open/Closed Principles

```
class PostedEvent {
  constructor(post, observers) {
    this.post = post;
    this.observers = observers;
  }
  notifyObservers() {
    this.observers.forEach(observer => observer.notify(this.post));
  }
}
// Usage.. now you can have as many observers as you want.
// They all respond to a standard interface.
const event = new PostedEvent(post, [
  PostNotificationObserver,
  PostMailerObserver,
  PostTexterObserver
]);
event.notifyObservers(); // somewhere in our code.
class PostMailerObserver { // One of these would look like this.
  constructor(post) { this.post = post; }
  notify() {NodeMailer.sendEmail(/* ... */)}
}
```

# Open/Closed Principles

One step further.

```
class CompositeObserver {
  constructor(observers) { this.observers = observers; }
}
// Now our posted event doesn't have to take an array. It can take any
// observer that responds to the `notify` message.
class PostedEvent {
  constructor(post, observer) {
    this.post = post; this.observer = observer;
  }
  notifyObserver() {
    this.observer.notify();
  }
}

const compositeObserver = new CompositeObserver([
  PostNotificationObserver,
  PostMailerObserver,
  PostTexterObserver
])
const event = new PostedEvent(post, compositeObserver);
```

# Liskov Substitution

Subclasses should be able to take the place of their parent class.

- This is probably the hardest to implement in common frameworks.
- If you are building a subclass of something make sure to adhere to the contract the superclass.
- Also applies to return values.

# Liskov Substitution

Example:

```
class PaymentProcessor {
  charge() { return '/receipt'; }
} // Even without this.
class StripeProcessor extends PaymentProcessor {
  charge() {
    stripe.create_subscriber(/* ... */);
    const receipt_url = '...';
    return receipt_url;
  }
}
class AmazonPaymentProcessor extends PaymentProcessor {
  charge() {
    Amazon.charge_card(/* ... */);
    const receipt_url = '...';
    return receipt_url;
  }
}
class FreeProcessor extends PaymentProcessor {
  charge() {} // we aren't complying to the implicit interface here.
}
```

# Liskov Substitution

If you are using typescript you can use an interface.

```
interface IPaymentProcessor {
  charge() : string;
}

class FreeProcessor implements IPaymentProcessor {
  charge() {
    const {id} = createFreeReceipt();
    return `/free_receipt/${id}`;
  }
}
```

If not this is a subtle thing you should be thinking about. LSP can have subtle violations that can bite you at runtime.

# Interface Segregation

Many client-specific interfaces are better than one general-purpose interface.

- Dynamic languages do not really have this problem.
- When using a typed language this is something to consider. (Typescript)

```
// NO
interface IValidator {
  validate(value: string)
  validateNumber(value: number)
};

// YES
interface IStringValidator {
  validate(value: string)
}

interface INumberValidator {
  validate(value: number)
}
```

## Dependency Inversion

Depend on abstractions not concretions.

- Loose Coupling.
- Don't mix high level business logic with low level implementation.
- Using Liskov and Open-Closed will often call upon this pattern.

# Dependency Inversion

```
class ImageGetter { // Violates Dependency Inversion.
  getImage() {
    let webcam = new NativeCameraInterface();
    return webcam.requestImage();
  }
}
let getter = new ImageGetter() // calling code would look like...
getter.getImage()

// Obeys Dependency Inversion.
class ImageGetter {
  constructor(imageReader) {
    this.imageReader = imageReader;
  }
  getImage() {
    return imageReader.requestImage();
  }
}
// calling code.
let getter = new ImageGetter(new NativeCameraInterface())
getter.getImage();
```

# Dependency Inversion

Notice that now that we have injected the dependency instead of depending on it directly. We can use any object the responds to the `requestImage` message.

```
class NativeCameraInterface() {
  requestImage() {}
}

class TestImageInterface() {
  requestImage() {}
}

// calling code can now use any implementation it wants
let getter = new ImageGetter(new NativeCameraInterface())
getter.getImage();
```

# Guidelines

The Code.

## Methodology

You will find that these principles are more a method to expand your thinking about code.

- We should use these to inform our code reviews.
- Thinking about objects and where dependencies are is a prime topic for code review and language agnostic.
- Do NOT treat them as rules to be applied with hammer. Each principle is solving a problem, do not solve problems that you don't have.
- Most importantly do not frontload these principles. Code what works first and then ask yourself if it could be a nicer api.

# Thank You!

https://gist.github.com/lolzballs/2152bc0f31ee0286b722