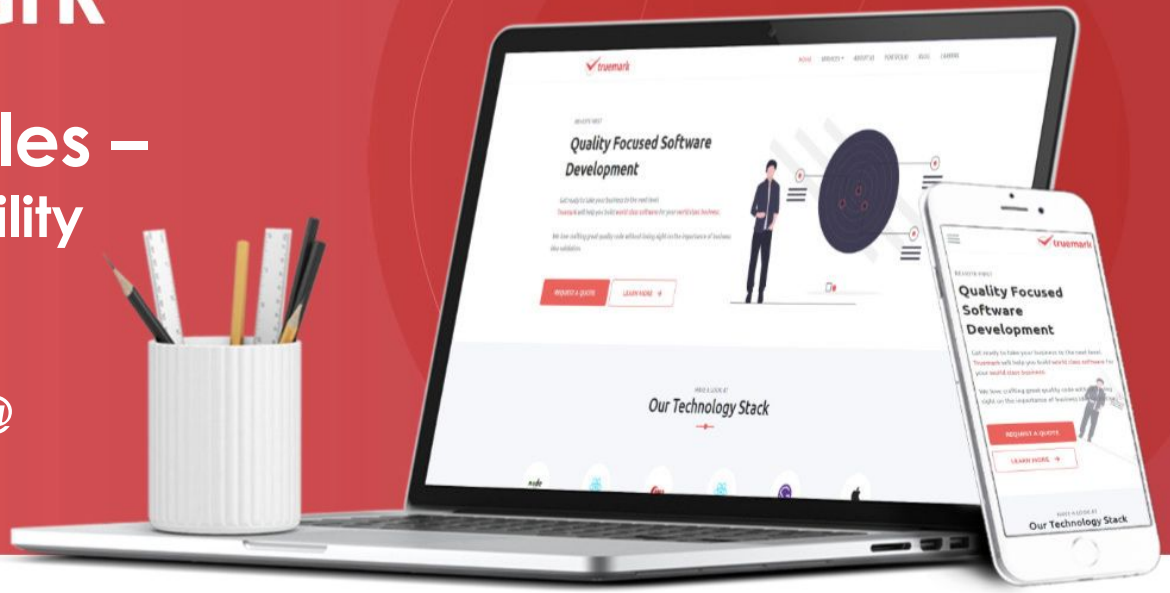
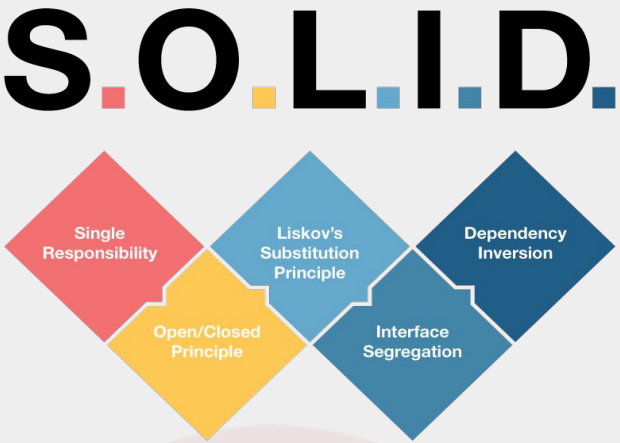
truemark

SOLID Principles – Single Responsibility Principle

Anmol Shah (ASE @ Truemark)



truemark | SOLID Introduction



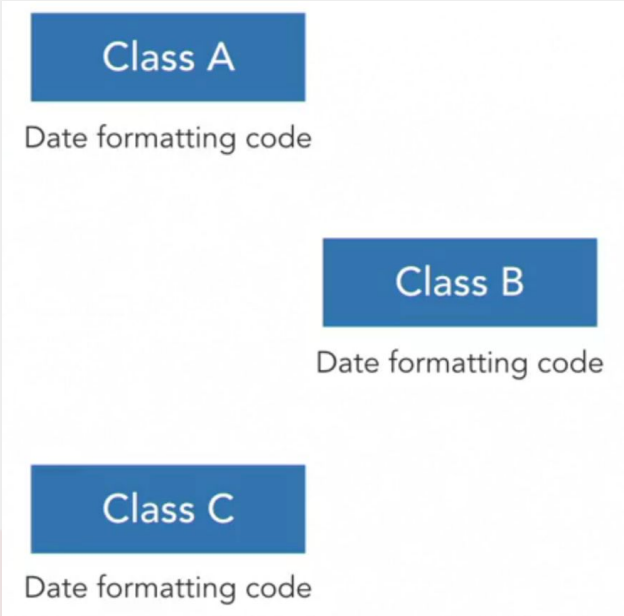
- Introduced by Robert C. Martin in his paper in early 2000s but the acronym was introduced later by Michael Feathers
- Basically a set of principles for object-oriented design (with focus on designing the classes)
- It is necessary to design classes in such a way that changes can be controlled and predictable

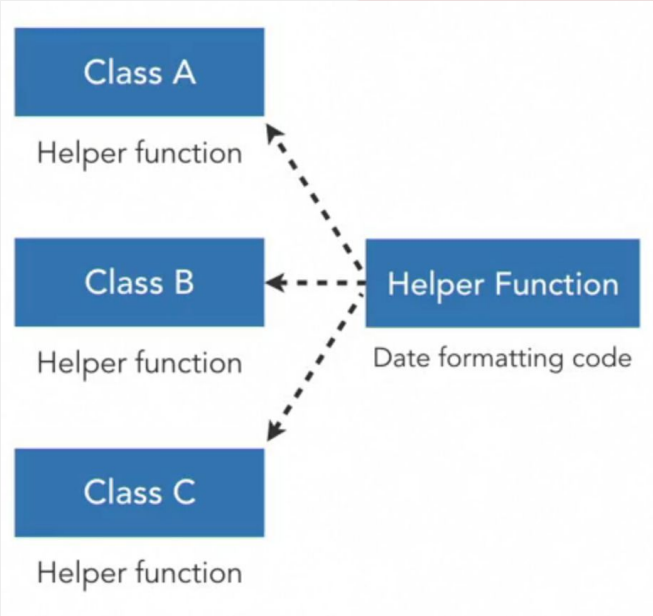
- SOLID design helps to decouple code and make modification easier
- Testable and easily understandable
- Dependency is the key problem in software development at all scales
- Eliminating duplication in programs eliminates dependency
- Incorporating design in the classes helps to create readable code that many developers can collaboratively work on

truemark | Code Smells

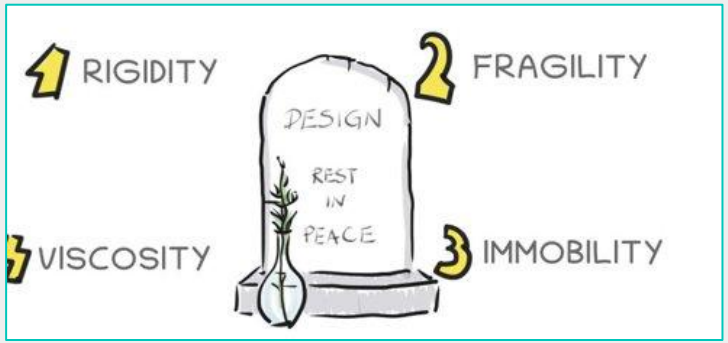
- According to wikipedia, code smells are certain structures in the code that indicate violation of fundamental design principles and negatively impact design quality.
- But do note that code smells are not bugs, compiler errors or non-functional code.

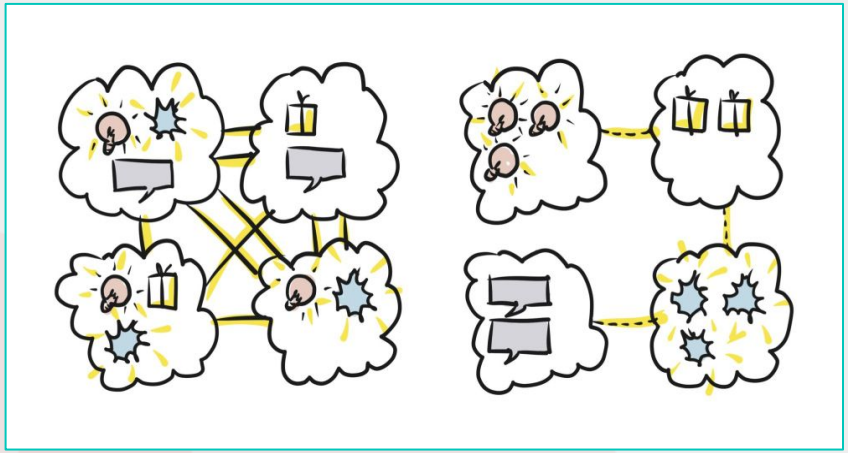
truemark | Code Smell Example





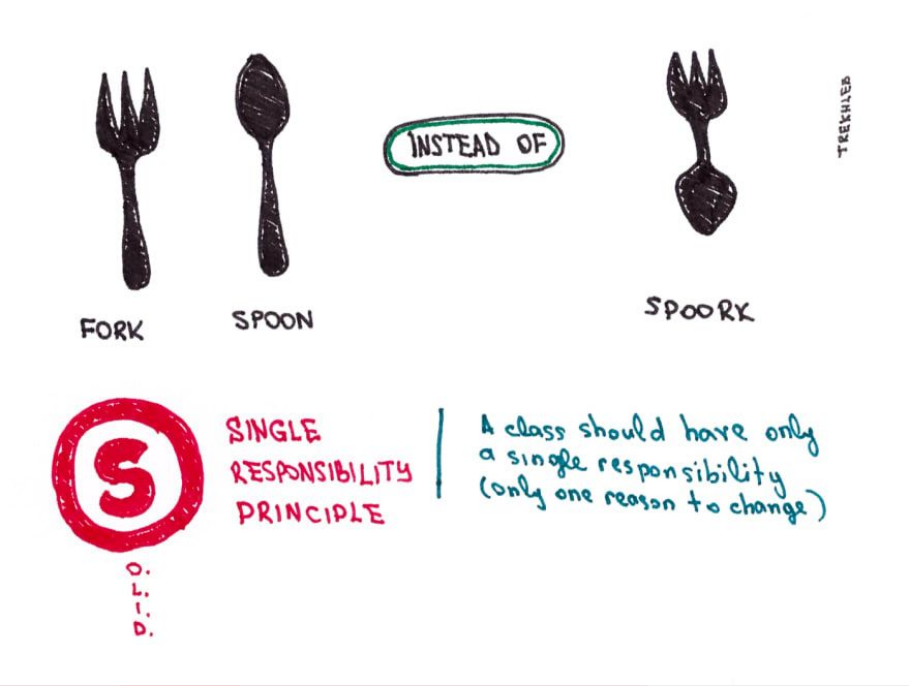
truemark | Symptoms of a rotting software





changes hard to apply	simple change impacts numerous modules	implementing simple changes takes forever
changing one place harms another	fixing a bug causes x others	modules are not reusable because of their dependencies
rewriting a code instead of reusing existing one	easier to do "hacks" than go "by the book"	environment is slow and inefficient

truemark | The Single Responsibility Principle



- According to this principle – "There should not be more than one reason for a class to change."
- Can be adjusted and extended quickly without producing bugs
- Classes narrowly do what they were intended to do

Importance

- Avoids modules incompatibility even when team members edit the same class for different reasons
- Makes version control easier
- Reduces dependencies between classes
- Easier to scale and maintain

truemark |

Violation of the Principle

- In the '**User**' class, the functionality to generate a pay slip of an employee based on their salary is put inside.
- To generate a pay slip all we need to do is instantiate a user object and call the `generate_payslip` method.
- Now, there is a new requirement. We want to send the generated payslip as an email.

```
class User
  def initialize(employee, month)
    @employee = employee
    @month = month
  end

  def generate_payslip
    # Code to read from database,
    # generate payslip
    # and write it to a file
  end
end
```

```
month = 11
user = User.new(employee, month)
user.generate_payslip
```

www.truemark.dev

truemark |

- We have added a new method `send_email` which is generating the payslip before sending out the email
- How do we refactor the code such that it also abides by the single responsibility principle?

```ruby
class User
  def initialize(employee, month)
    @employee = employee
    @month = month
  end

  def generate_payslip
    # Code to read from database,
    # generate payslip
    # and write it to a file
  end

  def send_email
    # generate payslip
    generate_payslip
    # code to send email
    employee.email
    month
  end
end
```

truemark | Good Practice of Coding

- This approach helps to decouple the responsibilities and ensures a predictable change.
- Each class has its own responsibility now since the class **PayslipGenerator** is just handling the generation of a pay slip
- Meanwhile, the class **PayslipMailer** is used to send those generated payslips as emails
- It also helps to predict any changes in functionality.

```ruby
class PayslipGenerator
  def initialize(employee, month)
    @employee = employee
    @month = month
  end

  def generate_payslip
    # Code to read from database,
    # generate payslip
    # and write it to a file
  end
end

class PayslipMailer
  def initialize(employee)
    @employee = employee
  end

  def send_mail
    # code to send email
    employee.email
    month
  end
end
```

truemark | Refactored example with JavaScript

```
class User {
  constructor(employee, month) {
    this.employee = employee;
    this.month = month;
  }

  sendEmail() {
    // code to send email
    this.employee.email;
    return this.month;
  }

  generatePayslip() {
    // Code to read from database,
    // generate payslip
    // and write it to a file
    this.sendEmail();
  }
}

const month = 11;
const user = new User(employee, month);
user.generatePayslip();
```

```
class PayslipGenerator {
  constructor(employee, month) {
    this.employee = employee;
    this.month = month;
  }

  generatePayslip() {
    // Code to read from database,
    // generate payslip
    // and write it to a file
  }
}

class PayslipMailer {
  constructor(employee) {
    this.employee = employee;
  }

  sendMail() {
    //code to send email
    this.employee.email;
    return month;
  }
}

const month = 11;
// generate payslip
const generator = new PayslipGenerator(employee, month);
generator.generatePayslip();
// send email
const mailer = new PayslipMailer(employee, month);
mailer.sendMail();
```

- Moving the business logic from controllers to Service objects
- Grouping the methods and constants inside Modules
- Extract the model logic into Concerns
- Creating utility classes for maximum code reusability

- Provides a principled way to manage dependency
- Results in code that are flexible, robust, and reusable
- A well-designed codebase is adaptable, simple to modify, and pleasurable to work with

truemark

THANK YOU

Pabitranagar, Gongabu
Kathmandu, Nepal 44600

+977 980-3572935
+977 984-9247553

hello@truemark.dev

Technologies: Spree | Reacjs | Gatsby Js | Ruby on Rails

www.truemark.dev