
Table of Contents

Introduction	1.1
Creational Patterns	1.2
Builder	1.2.1
Factory	1.2.2
Abstract Factory	1.2.3
Factory Method	1.2.4
Prototype	1.2.5
Singleton	1.2.6
Object Pool	1.2.7
Revealing Constructor	1.2.8
Structural Patterns	1.3
Adapter	1.3.1
Composite	1.3.2
Proxy	1.3.3
Flyweight	1.3.4
Facade	1.3.5
Bridge	1.3.6
Decorator	1.3.7
Private Class Data	1.3.8
Behavioral Patterns	1.4
Template Method	1.4.1
Mediator	1.4.2
Chain Of Responsibility	1.4.3
Observer	1.4.4
Strategy	1.4.5
Command	1.4.6
State	1.4.7

Visitor	1.4.8
Memento	1.4.9
Interpreter	1.4.10
Null Object	1.4.11
Iterator	1.4.12
Middleware	1.4.13
Clean Code Patterns	1.5
Extract Method	1.5.1
Clarify Responsibility	1.5.2
Remove Duplications	1.5.3
Keep Refactoring	1.5.4
Always Unit Test	1.5.5
Create Data Type	1.5.6
Comment to Better Name	1.5.7
Consistent Naming	1.5.8
If-else over ternary operator	1.5.9
Composition over Inheritance	1.5.10
Too Many Returns	1.5.11
Private to Interface	1.5.12
Anti Patterns	1.6
Big Ball of Mud	1.6.1
Singleton	1.6.2
Mad Scientist	1.6.3
Spaghetti Code	1.6.4
It Will Never Happen	1.6.5
Error Codes	1.6.6
Commented Code	1.6.7
Abbreviations	1.6.8
Prefixes	1.6.9
Over Patternized	1.6.10

Design Patterns Handbook

This handbook is full of examples to inspire usage and easier understanding of design patterns. What actually is a pattern?

Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution. -- Alexander

When we start learning design patterns, we might tend to use them whenever possible. That can lead to over-engineering, which produces more code than it is necessary to implement the needed requirement. It is also called as over using of design patterns.

One way to escape over-engineering is to use Test Driver Development. TDD is made of 3 phases. First, write a simple test. Second, make the test pass, by implementing behavior that is required by the test. Third, refactor what we have coded to make the test pass. The third phase is the ideal place to think about design patterns, because when we want to refactor the code, we have a problem “how to make the code better?”. The design patterns are here to help us with problems, not to be a fancy toys to make our code better.

Always remember, when we use design patterns to solve code problem we face to, the intention is to lower complexity, never increase. That way we only produce low complexity code that is going to be readable and easily maintainable.

Interesting links and reference materials:

- [Examples of GoF Design Patterns in Java's core libraries](#)
- [Design patterns deep dive](#) by SourceMaking
- Refactoring to Patterns by Joshua Kerievsky

Creational Patterns

Creational patterns are trying to help with object creation issues.

Builder

- Separates object construction from its representation

Factory

- Creates an instance of classes based on input

Abstract Factory

- Creates an instance of several families of classes

Factory Method

- Creates an instance of several derived classes

Prototype

- A fully initialized instance to be copied or cloned

Object Pool

- Avoid expensive acquisition and release of resources by recycling objects that are no longer in use

Singleton

- A class of which only a single instance can exist

Builder

The main reason to use Builder pattern is to avoid constructors with huge amount of parameters. Some people say we should have a limit for number of parameters and it should be around 3 parameters per constructor. We should keep the number parameters around 3 because if not, it is becoming a piece of code that is difficult to read, test and manage. People, who developed feelings for a code, might say that constructors with many parameters are ugly and not friendly.

The goal when using Builder patten is to create fluent interface. Here is an example of fluent interface:

```
new UserBuilder().withUserName("john").withAge(21).build();
```

Example - User builder

Here is an example a build that constructs a user.

```
public class User {

    private String firstName;
    private String lastName;

    private User(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public static class UserBuilder {

        private String firstName;
        private String lastName;

        public UserBuilder withFirstName(String firstName) {
            this.firstName = firstName;
            return this;
        }

        public UserBuilder withLastName(String lastName) {
            this.lastName = lastName;
            return this;
        }

        public User build() {
            return new User(firstName, lastName);
        }
    }
}
```

Then we can use the builder's fluent interface to create a new object of `User` class.

```
User john = new User.UserBuilder()  
    .withFirstName("John")  
    .withLastName("Feleti")  
    .build();
```

Builder in JavaScript using async / await

When we are writing integration tests, we might need to insert some data, that are expected to be present, into database. We can write a SQL script or put SQLs into code that would be executed before each test. But then it becomes difficult to update this script. As we add more features we might need to have multiple SQL scripts to cover various scenarios. Better is to split SQL inserts into methods of builder and let the user, the one who writes the integration tests, to choose what should be inserted.

Since we want to treat the test code as production code, we should follow DRY principle. Then it becomes handy to put data creation into builders that we can reuse in test code. Here is a builder that will help us to insert data into database so we can run our integration tests.


```
module.exports = class {
  constructor() {
    this.promises = []
  }

  withUser() {
    this.promises.push(async () => {
      const userService = await Q.container.getAsync('userService')
      await userService.save(new User('John'))
    })
    return this
  }

  withAccount() {
    this.promises.push(async () => {
      const accountService = await Q.container.getAsync('accountService')
      await accountService.save(new Account('My Account'))
    })
    return this
  }

  async build() {
    for (let promise of this.promises) {
      await promise()
    }
  }
}
```

Then we can use the builder in a test as follows.

```
const TestQueryBuilder = require('../fixtures/test_data_builder')

describe('User Is Able to Access his Account', function() {

  beforeEach(async function() {
    await new TestQueryBuilder().withUser().withAccount().build()
  })

  describe('login()', function() {
    it('fails to login because user is not linked with account',
    async function() {
      // write test code here
    })
  })
})
```

Factory

Factory pattern becomes handy we want to dynamically create instance based on some inputs.

Example - Animal factory

Lets say we got a file of various data, it is full of cats and dogs. We want to read the file and create specific objects based on a discriminator. Something like

```
AnimalFactory.create("DOG", "Alex") .
```

```
abstract class Animal {
    public abstract String getName();
}

class Dog extends Animal {

    private String name;

    public Dog(String name) {
        this.name = name;
    }

    @Override
    public String getName() {
        return name;
    }
}

class Cat extends Animal {

    private String name;

    public Cat(String name) {
        this.name = name;
    }
}
```

```
@Override
public String getName() {
    return name;
}

class AnimalFactory {

    public static Animal create(String type, String name) {
        if (type.equals("CAT")) {
            return new Cat(name);
        } else if (type.equals("DOG")) {
            return new Dog(name);
        }
        return null;
    }
}
```

Then we can easily create object using a static method `create` .

```
Animal cat = AnimalFactory.create("CAT", "Meow");
Animal dog = AnimalFactory.create("DOG", "Johnson");
```

If there are too many variables that should be passed into create method, we might want to combine this pattern with Builder pattern. When we call `create` method, it would return `AnimalBuilder` . It would become something like this: `AnimalBuilder buidler = AnimalFactory.create("DOG");`

Example - Factory in JavaScript

Here is an example of factory function that creates a specific instance of image class.

```
function createImage(name) {
  if (name.match(/\.(jpeg$/)) {
    return new JpegImage(name);
  } else if (name.match(/\.(png$/)) {
    return new PngImage(name);
  } else {
    throw new Error('Unsupported image format')
  }
}

class JpegImage {
  constructor(name) {
    this.name = name;
  }
}

class PngImage {
  constructor(name) {
    this.name = name;
  }
}

const jpeg = createImage('my.jpeg');
console.log(jpeg instanceof JpegImage);

const png = createImage('my.png');
console.log(png instanceof PngImage);

createImage('my.gif');
```

Example - Enforce encapsulation

This example shows factory method that not only creates a new instances, but also hides some properties from the user.

```
function createImage(name) {
  if (name.match(/\.(jpeg$/)) {
    return createJpeg(name);
```

```
    } else if (name.match(/\.\png/)) {
      return createPng(name);
    } else {
      throw new Error('Unsupported image format')
    }
  }
}

function createPng(name) {
  const privateProperties = {};

  class PngImage {
    constructor(name) {
      this.name = name;
      privateProperties.veryInternal = `very special and interna
1 thing that nobody can change`
    }
    getSomethingSpecialThatNobodyCanModify() {
      return privateProperties.veryInternal;
    }
  }
  const image = new PngImage(name);
  return image;
}

function createJpeg(name) {
  const privateProperties = {};
  class JpegImage {
    constructor(name) {
      this.name = name;
      privateProperties.veryInternal = `another very special and
internal thing that nobody can change`
    }
    getSomethingSpecialThatNobodyCanModify() {
      return privateProperties.veryInternal;
    }
  }

  const image = new JpegImage(name);

  return image;
}
```

```
}

const jpeg = createImage('my.jpeg');
console.log(jpeg);
console.log(jpeg.getSomethingSpecialThatNobodyCanModify());

const png = createImage('my.png');
console.log(png);
console.log(png.getSomethingSpecialThatNobodyCanModify());
```

Abstract Factory

The abstract factory pattern provides a way to encapsulate a group of individual factories that have a common theme without specifying their concrete classes.

Frameworks that are providing dependency injection, like Spring, are solving this issue and probably, we won't need Abstract Factory there.

Example - Animal factory

Each domain object (like Cat and Dog) are going to have its own factory. That way, we would never have to call `new Cat()` or `new Dog()` constructors. Instead we tell the factory the inputs and the factory makes sure we get an instance.

This way, we can easily change Cat and Dog implementations without affecting user of our factories.

```
interface Animal {
    String getName();
}

class Cat implements Animal {

    private String name;

    public Cat(String name) {
        this.name = name;
    }

    @Override
    public String getName() {
        return name;
    }
}

class Dog implements Animal {
```



```
    private String name;

    public Dog(String name) {
        this.name = name;
    }

    @Override
    public String getName() {
        return name;
    }
}

interface AnimalAbstractFactory {
    Animal create();
}

class CatFactory implements AnimalAbstractFactory {

    private String name;

    public CatFactory(String name) {
        this.name = name;
    }

    @Override
    public Animal create() {
        Cat cat = new Cat(name);
        return cat;
    }
}

class DogFactory implements AnimalAbstractFactory {

    private String name;

    public DogFactory(String name) {
        this.name = name;
    }
}
```

```
        @Override
        public Animal create() {
            return new Dog(name);
        }
    }

    class AnimalFactory {

        public static Animal create(AnimalAbstractFactory factory) {
            Animal animal = factory.create();
            return animal;
        }
    }
```

`AnimalAbstractFactory` can be implemented as interface or abstract class, it depends what is more useful for your problem.

Then we need to create an object, we create specific factory and as a user of such code, we no idea what instance is going to be created (because probably we shouldn't care).

```
Animal meow = AnimalFactory.create(new CatFactory("Meow"));
Animal gilbert = AnimalFactory.create(new DogFactory("Gilbert"));
;
```

Factory Method

Factory Method is using methods to create objects. Factory Method should define abstract methods in an abstract class or interface and classes that implement those are responsible for providing an instance.

Example - Multi-platform UI

We want to create a class design where we support two different architectures, Mac and Windows. The example is oversimplified but should provide basic idea.

```
interface Dialog {  
    void show();  
}  
  
abstract class AbstractDialog implements Dialog {  
  
    abstract TextField createTextField();  
    abstract Button createButton();  
}  
  
class MacDialog extends AbstractDialog {  
  
    @Override  
    public void show() {  
        createTextField();  
        createButton();  
    }  
  
    @Override  
    TextField createTextField() {  
        return new MacTextField();  
    }  
  
    @Override  
    Button createButton() {  
        return new MacButton();  
    }  
}
```

```
    }  
}  
  
class WindowsDialog extends AbstractDialog {  
  
    @Override  
    public void show() {  
        createTextField();  
        createButton();  
    }  
  
    @Override  
    TextField createTextField() {  
        return new WindowsTextField();  
    }  
  
    @Override  
    Button createButton() {  
        return new WindowsButton();  
    }  
}  
  
interface TextField {  
}  
  
class MacTextField implements TextField {  
}  
  
class WindowsTextField implements TextField {  
}  
  
interface Button {  
}  
  
class MacButton implements Button {  
}  
  
class WindowsButton implements Button {  
}
```

Then we just use specific implementations of class that defined abstract factory methods.

```
Dialog dialog = new WindowsDialog();  
Dialog dialog = new MacDialog();
```

Example - Factory method with Template pattern

Usually, factory method is combined with Template pattern. Factory methods are responsible for object creation and Template for behavior of those newly created objects. We would have to implement `show` method in `AbstractDialog` that will represent the template that uses the newly created objects. Then we should remove implementation of `show` method in `WindowsDialog` and `MacDialog`.

```
abstract class AbstractDialog implements Dialog {  
  
    @Override  
    public void show() {  
        createTextField().position(0, 0).show();  
        createButton().position(0, 100).show();  
    }  
  
    abstract TextField createTextField();  
    abstract Button createButton();  
}
```

Prototype

Prototype design pattern suggest to create a clone of a prototype rather than using `new` keyword and deep-copy all values to a cloned instance.

Example - Simple Prototype

In order to implement prototype pattern, we can simply implement `Cloneable` interface from Java.

```
class HumanCell implements Cloneable {  
  
    private String dna;  
  
    public HumanCell(String dna) {  
        this.dna = dna;  
    }  
  
    public String getDna() {  
        return dna;  
    }  
  
    // all complex objects must be recreated (lists, maps, POJOs  
)  
    public Object clone() throws CloneNotSupportedException {  
        return new HumanCell(dna);  
    }  
}
```

Then we can clone human cells like this.

```
HumanCell cell = new HumanCell("DNA123");  
HumanCell identical = (HumanCell) cell.clone();
```

Example - Define own prototype interface

We can create our own interface that will define how a prototype should be copied into a new class.

```
interface Cell {
    Cell split();
    String getDna();
}

class SingleCellOrganism implements Cell {

    private String dna;

    public SingleCellOrganism(String dna) {
        this.dna = dna;
    }

    public String getDna() {
        return dna;
    }

    // all complex objects must be recreated (lists, maps, POJOs
    )
    public Cell split() {
        return new SingleCellOrganism(dna);
    }
}
```

Then we can use it as follows and we can exclude casting Object into a specific type.

```
Cell cell = new SingleCellOrganism("DNA123");
Cell identical = i1.split();
```

Singleton

Singleton provides a way to make sure there is only one instance of a class in one JVM instance.

Because many people used Singleton "everywhere", it became an anti-pattern. Singleton became an anti-pattern because it is difficult to create tests for code that has hundreds of singletons that make mocking really difficult. Another point against singletons are frameworks like Spring can make sure a bean is created only once and we do not need to write any extra code to achieve that.

Before Java had enums, we used to implement singletons to mimic enums.

Example - Thread-safe singleton

Here is an example of thread-safe Singleton. Lets explain key points of this implementation:

- volatile is used to keep singleton in main memory, if volatile is not used, it could be in CPU cache, which could cause issues in multi-threaded environment
- constructor is private, only the singleton can instantiate it's self
- call of the constructor is synchronized on class level, to avoid two threads doing the same thing


```
public class ThreadSafeSingleton {  
  
    private static volatile ThreadSafeSingleton instance;  
  
    private ThreadSafeSingleton() {  
    }  
  
    public static ThreadSafeSingleton getInstance() {  
        if (instance == null) {  
            synchronized (ThreadSafeSingleton.class) {  
                instance = new ThreadSafeSingleton();  
            }  
        }  
        return instance;  
    }  
}
```

Here is how we get the instance of such a singleton.

```
ThreadSafeSingleton single = ThreadSafeSingleton.getInstance();
```

Hack singleton

Anyway, there is a way to hack singleton and create new instances. We can use reflection and set the constructor as accesible.

```
ThreadSafeSingleton instanceOne = ThreadSafeSingleton.getInstance();
ThreadSafeSingleton instanceTwo = null;
try {
    Constructor[] constructors = ThreadSafeSingleton.class.getDeclaredConstructors();
    for (Constructor constructor : constructors) {
        //Below code will destroy the singleton pattern
        constructor.setAccessible(true);
        instanceTwo = (ThreadSafeSingleton) constructor.newInstance();
        break;
    }
} catch (Exception e) {
    e.printStackTrace();
}

// see, we got two instances that have different has codes!
System.out.println(instanceOne.hashCode());
System.out.println(instanceTwo.hashCode());
```

Object Pool

Object Pool is useful when creating a new instance costs a lot of time. We can make performance boost by keeping pool of reusable objects instead of making new objects on request.

Example - Connection pool

Good example to explain object pool is a database connection pool.

```
class ConnectionPool {

    private Set<Connection> available = new HashSet<>();
    private Set<Connection> taken = new HashSet<>();

    public ConnectionPool() {
        available.add(new Connection());
        available.add(new Connection());
        available.add(new Connection());
    }

    public Connection getConnection() {
        Connection connection = available.iterator().next();
        available.remove(connection);
        taken.add(connection);
        return connection;
    }

    public void returnConnection(Connection connection) {
        taken.remove(connection);
        available.add(connection);
    }

    public String toString() {
        return "Available: " + available.size() + ", taken: " +
taken.size();
    }
}

class Connection {
}
```

We can test the connection pool. We first create a new pool and then we obtain and return a connection.

```
ConnectionPool pool = new ConnectionPool();  
System.out.println(pool);  
  
Connection connection = pool.getConnection();  
System.out.println(pool);  
  
pool.returnConnection(connection);  
System.out.println(pool);
```

Here is the output of our test.

```
Available: 3, taken: 0  
Available: 2, taken: 1  
Available: 3, taken: 0
```

Revealing Constructor

Revealing constructor is used in JavaScript, for example, to implement `Promise` class. The constructor of `Promise` class takes function with two parameters (`resolve` and `reject`) into constructor. It is perfect way to hide implementation details and there is no way somebody can mess around with `resolve` and `reject` after the instance is created.

Example - Read-only event emitter

The `emit` function of `ReadOnlyEmitter` can be called only in function that is passed into the constructor.

```
const EventEmitter = require('events');

class ReadOnlyEmitter extends EventEmitter {
  constructor (executor) {
    super();
    const emit = this.emit.bind(this);
    this.emit = undefined; // hide emit function
    executor(emit);
  }
};

const ticker = new ReadOnlyEmitter((emit) => {
  let tickCount = 0;
  setInterval(() => emit('tick', tickCount++), 1000);
});

ticker.on('tick', (count) => console.log('Tick', count));
```

Structural Patterns

Structural patterns are to ease the design by identifying a simple way to realize relationships between entities.

Adapter

- Match interfaces of different classes

Composite

- A tree structure of simple and composite objects

Proxy

- An object representing another object

Flyweight

- A fine-grained instance used for efficient sharing

Facade

- A single class that represents an entire subsystem

Bridge

- Separates an object's interface from its implementation

Decorator

- Add responsibilities to objects dynamically

Private Class Data

- Restricts accessor/mutator access

Adapter

Adapter lets classes work together that could not otherwise because of incompatible interfaces.

Example - User adaptation

Lets say we work for eBay company and there are other users we need to integrate with our systems. Our, eBay, user is defined by `EbayUser` interface. External users are represented by class `ExternalUser` , but we do not want to mess with this class in our code and that is why we create an adapter that will adapt external user into our system.

```
interface EbayUser {
    String getUsername();
}

class EbayUserImpl implements EbayUser {

    private String userName;

    public EbayUserImpl(String userName) {
        this.userName = userName;
    }

    @Override
    public String getUsername() {
        return userName;
    }
}

class ExternalUserToEbayUserAdapter implements EbayUser {

    private String userName;

    public ExternalUserToEbayUserAdapter(ExternalUser paypalUser
) {
```



```
        userName = paypalUser.getUsername();
    }

    @Override
    public String getUserName() {
        return userName;
    }
}

class ExternalUser {

    private String username;

    public ExternalUser(String username) {
        this.username = username;
    }

    public String getUsername() {
        return username;
    }
}
```

Here is an example how we could adapt an external user into our system.

```
ExternalUser externalUser = new ExternalUser("john");

EbayUser ebayUser = new ExternalUserToEbayUserAdapter(externalUser);
```

Composite

When ever we need to a tree structure, we use composite pattern.

Example - Simple Composite

There is a way to create really simple tree structure using composite patter (but we have to omit few classes in order to make it really simplistic).

```
class TreeNode {  
  
    private String name;  
    private List<TreeNode> children = new ArrayList<>();  
  
    public TreeNode(String name) {  
        this.name = name;  
    }  
  
    public void addChild(TreeNode child) {  
        children.add(child);  
    }  
  
    public boolean isLeaf() {  
        return children.size() == 0;  
    }  
}
```

Now we can create a tree using just TreeNode.

```
TreeNode root = new TreeNode("Root");  
root.addChild(new TreeNode("Node 1"));  
root.addChild(new TreeNode("Leaf 1"));
```

Example - Composite with an interface

Here is an example of composite pattern. `Leaf` can't have any children. `Node` can contain multiple implementations of `Component` interface. Node class is enabling the tree structure.

```
interface Component {
    void sayYourName();
}

class Node implements Component {

    private String name;

    private List<Component> elements = new ArrayList<>();

    public Node(String name) {
        this.name = name;
    }

    @Override
    public void sayYourName() {
        System.out.println(name);
    }

    public void add(Component component) {
        elements.add(component);
    }
}

class Leaf implements Component {

    private String name;

    public Leaf(String name) {
        this.name = name;
    }

    @Override
    public void sayYourName() {
        System.out.println(name);
    }
}
```

Here is how to create a tree using the classes above.

```
Node root = new Node("Root");
root.add(new Node("Node 1"));
root.add(new Leaf("Leaf 1"));
```

Complex example with simple algorithm

A mathematical formula can be represented as a tree. Lets try it out and see how composite works in real life.

```
interface ArithmeticExpression {
    int evaluate();
}

class CompositeOperand implements ArithmeticExpression {

    private String operator;
    private List<ArithmeticExpression> expressions = new ArrayList<>();

    public CompositeOperand(String operator) {
        this.operator = operator;
    }

    public void add(ArithmeticExpression expression) {
        expressions.add(expression);
    }

    @Override
    public int evaluate() {

        List<Integer> evaluated = new ArrayList<>();

        for (ArithmeticExpression expression : expressions) {
            evaluated.add(expression.evaluate());
        }

        Integer result = null;
        for (Integer number : evaluated) {
```

```
        if (operator.equals("*")) {
            if (result == null) {
                result = number;
            } else {
                result *= number;
            }
        } else if (operator.equals("+")) {
            if (result == null) {
                result = number;
            } else {
                result += number;
            }
        } else if (operator.equals("-")) {
            if (result == null) {
                result = number;
            } else {
                result -= number;
            }
        }
    }

    return result;
}

class NumericOperand implements ArithmeticExpression {

    private int number;

    public NumericOperand(int number) {
        this.number = number;
    }

    @Override
    public int evaluate() {
        return number;
    }
}
```

Now we can construct mathematical formula. We are going to use this $((7+3)*(5-2))$.

```
// create root
CompositeOperand multiply = new CompositeOperand("");

// create +
CompositeOperand firstSum = new CompositeOperand("+");
firstSum.add(new NumericOperand(7));
firstSum.add(new NumericOperand(3));
multiply.add(firstSum);

// create -
CompositeOperand secondSum = new CompositeOperand("-");
secondSum.add(new NumericOperand(5));
secondSum.add(new NumericOperand(2));
multiply.add(secondSum);

// evaluate all children, then do math operation
int result = multiply.evaluate();
System.out.println(result);
```

The program will print out the following.

30

Proxy

A proxy is an object that controls access to another object. They both must have the same interface (methods and methods signatures).

- Provide a surrogate or placeholder for another object to control access to it.
- Use an extra level of indirection to support distributed, controlled, or intelligent access.
- Add a wrapper and delegation to protect the real component from undue complexity.

A proxy is useful to do data validation, security, caching, logging, lazy initialization or implement a remote object logic.

Example - Secured commands

We are going to use Proxy pattern to secure some commands that should be accessible only for administrators. `CommandExecutorProxy` class makes sure the user who is trying to execute a command is the administrator.


```
interface CommandExecutor {
    void execute(String s);
}

class CommandExecutorImpl implements CommandExecutor {

    @Override
    public void execute(String s) {
        System.out.println("> " + s);
    }
}

class CommandExecutorProxy implements CommandExecutor {

    private CommandExecutor executor = new CommandExecutorImpl();
    ;

    private boolean isAdmin;

    public CommandExecutorProxy(String username, String password
) {
        // TODO: find out if user is admin
        isAdmin = true;
    }

    @Override
    public void execute(String command) {
        if (isAdmin) {
            executor.execute(command);
        } else {
            throw new RuntimeException("Not authorized!");
        }
    }
}
```

Here is how we could use the code created above. If `john` is not an admin, the application should throw exception.

```
CommandExecutor executorProxy = new CommandExecutorProxy("john",  
    "my-secret");  
executorProxy.execute("ls -l");
```

Example - Proxy in JavaScript

Here is an example how to implement Proxy in JavaScript.

```
function createProxy(subject) {
  const proto = Object.getPrototypeOf(subject)
  function Proxy(subject) {
    this.subject = subject;
  }
  Proxy.prototype = Object.create(proto);
  // proxies function
  Proxy.prototype.hello = function() {
    return this.subject.hello() + ' there!';
  }
  // delegate function
  Proxy.prototype.goodbye = function() {
    return this.subject.goodbye.apply(this.subject, arguments)
  }
  return new Proxy(subject);
}

class MyObject {
  hello() {
    return "Hello";
  }
  goodbye() {
    return "Goodbye";
  }
}

const myObject = new MyObject();
const myProxiedObject = createProxy(myObject);
console.log(myProxiedObject instanceof MyObject); // return true
console.log(myProxiedObject.hello());
console.log(myProxiedObject.goodbye());
```

Or there is easier way to create proxy, taking advantage of dynamic typing in JavaScript.

```
function createProxy(subject) {
  return {
    hello: () => subject.hello() + ' there!',
    goodbye: () => subject.goodbye(),
  };
}

const myObject = new MyObject();
const myProxiedObject = createProxy(myObject);
console.log(myProxiedObject instanceof MyObject); // return false
console.log(myProxiedObject.hello());
console.log(myProxiedObject.goodbye());
```

ES2015 contains Proxy implementation.

```
class MyObject {
  constructor(name) {
    this.name = name;
  }
}

const myObj = new MyObject("Celeste");
const myProxied = new Proxy(myObj, {
  get: (target, property) => target[property].toUpperCase()
});
console.log(myProxied.name); // prints "CELESTE"
```

Flyweight

Flyweight is intended to use sharing to support large numbers of fine-grained objects efficiently.

Good example to explain usage Flyweight is UIKit by Apple. When we create TableView that is can display millions rows to the user, we know that user can see only few at the time. What Apple engineers did that they are reusing components that are on the table view to avoid any performance issues.

Example - UI component factory

When our UI table should display a button, it will use ButtonFactory to get a button (keeping track of what buttons are not used and visible is omitted). Number that is the parameter in `get` method could be an ID of a button in a table row (but it is not really important in this example).

This example intent is to show that Flyweight is some kind of cash for objects that are memory-heavy or difficult to create.

```
class Button {  
}  
  
class ButtonFactory {  
  
    private Map<Integer, Button> cache = new HashMap<>();  
  
    public Button get(Integer number) {  
        if (cache.containsKey(number)) {  
            return cache.get(number);  
        } else {  
            Button button = new Button();  
            cache.put(number, button);  
            return button;  
        }  
    }  
}
```

Then we would use the factory whenever we need an instance of a button that can be reused.

```
ButtonFactory buttonFactory = new ButtonFactory();  
  
Button button1a = buttonFactory.get(1);  
Button button1b = buttonFactory.get(1);
```

Facade

Facade purpose is to hide complex libraries API and provide simple class or set of classes that are meant to be used by users of a library.

Example - Document reader

We are going to use Facade pattern to provide API for complex readers that can read various document types, like MS Word or Text files.

These are the readers that should be hidden from users.

```
class TextReader {  
  
    public TextReader(String document) {  
    }  
  
    public String getText() {  
        return "Content of a simple text fiel";  
    }  
}  
  
class WordReader {  
  
    public WordReader(String document) {  
    }  
  
    public String getText() {  
        return "Content of a MS Word document.";  
    }  
}
```

We create a facade class called `DocumentReader` to provide easy API for library users. We usually put facade classes into library root, so it is the first thing a library user sees.

```
class DocumentReader {  
  
    public String read(String document) {  
        if (document.endsWith(".docx")) {  
            return new WordReader(document).getText();  
        } else if (document.endsWith(".txt")) {  
            return new TextReader(document).getText();  
        }  
  
        return "";  
    }  
}
```

Here is how users of our library would use the facade.

```
DocumentReader reader = new DocumentReader();  
  
String wordText = reader.read("resume.docx");  
String text = reader.read("resume.txt");
```


Bridge

- Decouple an abstraction from its implementation so that the two can vary independently.
- Publish interface in an inheritance hierarchy, and bury implementation in its own inheritance hierarchy
- Beyond encapsulation, to insulation (object wrapping)

At first sight, the Bridge pattern looks a lot like the Adapter pattern in that a class is used to convert one kind of interface to another. However, the intent of the Adapter pattern is to make one or more classes' interfaces look the same as that of a particular class. The Bridge pattern is designed to separate a class's interface from its implementation so you can vary or replace the implementation without changing the client code. [more](#)

Example - Document reader

We are going to create document parser that is responsible to read file and return its content as text. Implementation of `DocumentParser` will be different for MS Word, PDF and so on. Instead of using implementations of `DocumentParser` interface, we want to create `DocumentReader` because it will provide API for reading files.

```
interface DocumentParser {
    String parse();
}

class WordParser implements DocumentParser {

    private String fileName;

    public WordParser(String fileName) {
        this.fileName = fileName;
    }

    @Override
    public String parse() {
        return "Content of Word document.";
    }
}

interface DocumentReader {
    String getText();
}

class DocumentReaderImpl implements DocumentReader {

    private DocumentParser documentParser;

    public DocumentReaderImpl(DocumentParser documentParser) {
        this.documentParser = documentParser;
    }

    @Override
    public String getText() {
        // TODO: here we would do more stuff if needed...
        return documentParser.parse();
    }
}
```

Here is an example how to read MS Word using Bridge pattern.

```
DocumentReader reader = new DocumentReaderImpl(new WordParser("document.docx"));  
String text = reader.getText();
```

Decorator

- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- Client-specified embellishment of a core object by recursively wrapping it.
- Wrapping a gift, putting it in a box, and wrapping the box.

Example - Cars

We create few classes and then we can dynamically build cars, without changing the code.

```
interface Car {
    void assemble();
}

class BasicCar implements Car {

    @Override
    public void assemble() {
        System.out.println("Basic car.");
    }
}

class CarDecorator implements Car {

    private Car car;

    public CarDecorator(Car car) {
        this.car = car;
    }

    @Override
    public void assemble() {
        car.assemble();
    }
}
```

```
class SportCar extends CarDecorator {

    public SportCar(Car car) {
        super(car);
    }

    @Override
    public void assemble() {
        super.assemble();
        System.out.println("Sport car.");
    }
}

class LuxuryCar extends CarDecorator {

    public LuxuryCar(Car car) {
        super(car);
    }

    @Override
    public void assemble() {
        super.assemble();

        System.out.println("Luxury car.");
    }
}
```

Here is how to build Mustang and LaFerrari using the classes wrapping.

```
Car mustang = new SportCar(new BasicCar());
mustang.assemble();

System.out.println();

Car laFerrari = new SportCar(new LuxuryCar(new BasicCar()));
laFerrari.assemble();
```

The code prints out the following and we can observe how car was assembled.

Basic car.

Sport car.

Basic car.

Luxury car.

Sport car.

Private Class Data

- Control write access to class attributes
- Separate data from methods that use it
- Encapsulate class data initialization
- Providing new type of `final` - final after constructor

Example - Hide attributes by keeping them elsewhere

In this example, we want to hide user's password variable.

```
class User {
    private UserData userData;

    public User(String name, String password) {
        userData = new UserData(name, password);
    }

    public String getName() {
        return userData.getName();
    }
}

class UserData {
    private String name;
    private String password;

    public UserData(String name, String password) {
        this.name = name;
        this.password = password;
    }

    public String getName() {
        return name;
    }
}
```

User of the code does not have to know how is data internally stored in class, he uses the code as normally. But he can't access `getPassword()` method anywhere.

```
User john = new User("John", "secret");
```

This is simplified example, we could be hiding variables, methods or some functionality at initialization of an object.

Behavioral Patterns

Behavioral patterns help to solve interaction or communication of objects.

Template method

- Defer the exact steps of an algorithm to a subclass

Mediator

- Defines simplified communication between classes

Chain of responsibility

- A way of passing a request between a chain of objects

Observer

- A way of notifying change to a number of classes

Strategy

- Encapsulates an algorithm inside a class

Command

- Encapsulate a command request as an object

State

- Alter an object's behavior when its state changes

Visitor

- Defines a new operation to a class without change

Memento

- Capture and restore an object's internal state

Interpreter

- A way to include language elements in a program

Null Object

- Designed to act as a default value of an object

Iterator

- Sequentially access the elements of a collection

Template Method

Template method defines the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Base class declares algorithm 'placeholders', and derived classes implement the placeholders.

Example - Building a house

We are going to make "algorithm" to build a house. First we want to build foundation, then walls, then windows and roof as the last thing.

```
abstract class HouseTemplate {

    public void build() {
        buildFoundation();
        buildWalls();
        buildWindows();
        buildRoof();
    }

    protected abstract void buildFoundation();
    protected abstract void buildWalls();
    protected abstract void buildWindows();
    protected abstract void buildRoof();
}

class WoodenHouse extends HouseTemplate {

    @Override
    protected void buildFoundation() {
        System.out.println("Foundation");
    }

    @Override
    protected void buildWalls() {
        System.out.println("Wooden Walls");
    }

    @Override
    protected void buildWindows() {
        System.out.println("Windows");
    }

    @Override
    protected void buildRoof() {
        System.out.println("Wooden Roof");
    }
}
```

Then we build a wooden house like this.

```
WoodenHouse house = new WoodenHouse();  
house.build();
```

Default implementations

We might find that some steps are the same in all implementations of template class. That might be case to put these implementations up, to the template class. We might move `buildFoundation` and `buildRoof` into template class because those could be implemented in the same way for all house types. And if not, then we can always override `buildFoundation` and `buildRoof` methods.

```
abstract class HouseTemplate {

    public void build() {
        buildFoundation();
        buildWalls();
        buildWindows();
        buildRoof();
    }

    protected abstract void buildFoundation() {
        System.out.println("Foundation");
    }

    protected abstract void buildWalls();

    protected void buildWindows() {
        System.out.println("Windows");
    }

    protected abstract void buildRoof();
}

class WoodenHouse extends HouseTemplate {

    @Override
    protected void buildWalls() {
        System.out.println("Wooden Walls");
    }

    @Override
    protected void buildRoof() {
        System.out.println("Wooden Roof");
    }
}
```

Mediator

Mediator provides way to implement many-to-many relationships between interacting peers. If peers interact directly between each other, it means they are tangled and we end up with spaghetti code.

Imagine if all the airplanes would try to agree who is landing first between each other, rather than contacting the air traffic control tower. Mediator is the air traffic control tower.

Example - Chat

User belongs to a group, if he wants to say something, he is always "redirected" to group. Group is the mediator, providing way users can talk together.

```
class Group {

    private List<User> users = new ArrayList<>();

    public void add(User user) {
        users.add(user);
    }

    public void say(String message, User author) {
        users.stream()
            .filter(user -> user != author)
            .forEach(user -> System.out.println(user.getName
() + ": " + message));
    }
}

class User {

    private Group group;
    private String name;

    public User(Group group, String name) {
        this.group = group;
        this.name = name;
    }

    public Group getGroup() {
        return group;
    }

    public String getName() {
        return name;
    }

    public void say(String message) {
        group.say(message, this);
    }
}
```


Here is example of usage.

```
Group group = new Group();

User john = new User(group, "John");
group.add(john);
User jimmy = new User(group, "Jimmy");
group.add(jimmy);
User emily = new User(group, "Emily");
group.add(emily);

jimmy.say("Hi everyone!");
```

When we run the code, Jimmy says "Hi everyone!" and users are notified with the new message.

```
John: Hi everyone!
Emily: Hi everyone!
```

Chain of Responsibility

Chain of responsibility pattern provides a way to create a linked list or pipeline of objects that might be able solve an issue the pipeline is asked to solve.

Example - Image processor

We want to create processor that will decide itself what algorithm should be executed for an given image type.

```
interface ImageTransformer {
    void setNext(ImageTransformer transformer);
    void resize(String image);
}

class JpegTransfomer implements ImageTransformer {

    private ImageTransformer next;

    @Override
    public void setNext(ImageTransformer transformer) {
        this.next = transformer;
    }

    @Override
    public void resize(String image) {
        if (image.endsWith(".jpeg")) {
            System.out.println(getClass().getSimpleName() + " is
processing: " + image);
        } else {
            if (next != null) {
                next.resize(image);
            }
        }
    }
}
```

```
class PngTransfomer implements ImageTransformer {

    private ImageTransformer next;

    @Override
    public void setNext(ImageTransformer transformer) {
        this.next = transformer;
    }

    @Override
    public void resize(String image) {
        if (image.endsWith(".png")) {
            System.out.println(getClass().getSimpleName() + " is
processing: " + image);
        } else {
            if (next != null) {
                next.resize(image);
            }
        }
    }
}
```

Then we can create the cain of image transformers and submit a request to process an image.

```
ImageTransformer chain = new PngTransfomer();
ImageTransformer jpegTransfomer = new JpegTransfomer();
chain.setNext(jpegTransfomer);

String png = "image.png";
String jpeg = "image.jpeg";

chain.resize(png);
chain.resize(jpeg);
```

Here is the output.

```
PngTransfomer is processing: image.png  
JpegTransfomer is processing: image.jpeg
```

Observer

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

Example - Propagate status change

When user status changes, we need to do many things: update UI, publish message so other users are notified about status change and maybe we know that we will add more reactions later on.

```
class StatusSubject {  
  
    private int status;  
  
    private List<StatusObserver> observers = new ArrayList<>();  
  
    public void add(StatusObserver observer) {  
        observers.add(observer);  
    }  
  
    public void updateStatus(int status) {  
        this.status = status;  
    }  
  
    public void notifyObservers() {  
        observers.stream().forEach(it -> it.update());  
    }  
}  
  
abstract class StatusObserver {  
  
    private StatusSubject subject;  
  
    public StatusObserver(StatusSubject subject) {  
        this.subject = subject;  
        subject.add(this);  
    }  
}
```

```
    }

    public StatusSubject getSubject() {
        return subject;
    }

    public abstract void update();
}

class PublishMessageToQueue extends StatusObserver {

    public PublishMessageToQueue(StatusSubject subject) {
        super(subject);
    }

    @Override
    public void update() {
        System.out.println(getClass().getSimpleName() + " updating...");
    }
}

class UpdateUIStatusBarObserver extends StatusObserver {

    public UpdateUIStatusBarObserver(StatusSubject subject) {
        super(subject);
    }

    @Override
    public void update() {
        System.out.println(getClass().getSimpleName() + " updating...");
    }
}
```

Here is how to construct the observer and notify observers.

```
StatusSubject subject = new StatusSubject();  
new UpdataUIStatusBarObserver(subject);  
new PublishMessageToQueue(subject);  
  
subject.updateStatus(1);  
subject.notifyObservers();
```

Here is the output.

```
UpdataUIStatusBarObserver updating...  
PublishMessageToQueue updating...
```

Strategy

- Define the interface of an interchangeable family of algorithms
- Bury algorithm implementation details in derived classes
- Derived classes could be implemented using the Template Method pattern
- Clients of the algorithm couple themselves strictly to the interface

Example - Checking data providers

We are going to use Strategy pattern to implement various strategies to check websites like Google, eBay and so on.


```
interface Strategy {
    void solve();
}

class GoogleStrategy implements Strategy {
    @Override
    public void solve() {
        System.out.println("Checking Google");
    }
}

class EbayStrategy implements Strategy {
    @Override
    public void solve() {
        System.out.println("Checking eBay");
    }
}

class StrategyExecutor {
    void execute() {
        Strategy[] strategies = {new GoogleStrategy(), new EbayStrategy()};
        Arrays.stream(strategies).forEach(Strategy::solve);
    }
}
```

Then we use the executor to run our strategies.

```
new StrategyExecutor().execute();
```

Here is the output.

```
Checking Google
Checking eBay
```

Using Strategy to replace complex logic in single class

Lets consider we have create a class that is doing a simple thing. It will acquire an item from a storage. Then we did many modification in time and the class become really complex. Such a class is then difficult to modify and test. Also it is difficult to easily get an idea what is the class actually doing.

```
const _ = require('lodash')

Q.Errors.declareError('NoProxyAvailableError')

module.exports = class {
  constructor(
    domainRedisService = 'redis:domainRedisService',
    proxyRedisService = 'redis:proxyRedisService',
    providerService,
    policyService,
    proxyPolicyMongoService = 'mongo:proxyPolicyMongoService',
    policyRedisService = 'redis:policyRedisService',
    lockFactory
  ) {
    this.domainRedisService = domainRedisService
    this.proxyRedisService = proxyRedisService
    this.providerService = providerService
    this.policyMongoService = policyService
    this.proxyPolicyMongoService = proxyPolicyMongoService
    this.policyRedisService = policyRedisService
    this.lockFactory = lockFactory
  }

  async tryToAcquireProxy(domain) {
    await this.tryToCreateProxySet()
    await this.tryToCreateDefaultDomainSet()

    const domainExists = await this.domainRedisService.domainExists(domain)
    if (!domainExists) {
      await this.tryToCreateDomain(domain)
    }
  }
}
```

```
    }
    const proxy = await this.acquireProxy(domain)

    Q.log.info({ domain, proxy }, 'Acquired proxy based on origin or host')
    return proxy
  }

  async acquireProxy(domain) {
    const proxyIndex = await this.domainRedisService.getProxyIndex(domain)
    if (!proxyIndex) {
      throw new Q.Errors.NoProxyAvailableError('Not able to find proxy index for domain', { domain })
    }
    const proxy = await this.proxyRedisService.findByName(proxyIndex)
    proxy.proxyIndex = proxyIndex
    if (!proxy) {
      throw new Q.Errors.NoProxyAvailableError('Not able to find proxy for domain', { domain })
    }
    if (proxy && proxy.unlimited) {
      return proxy
    }
    await this.domainRedisService.updateTimestamp(domain, proxyIndex)
    return proxy
  }

  async tryToCreateProxySet() {
    const proxySetAvailable = await this.proxyRedisService.exists()
    if (proxySetAvailable) return
    let lock
    try {
      lock = await this.lockFactory.acquire('proxy:createIndexSet:lock', { timeout: 20000, retries: 20, delay: 1000 })
      if (!lock) return
      const proxies = await this.providerService.findProxies(1,
```

```

1000000)

    if (!await this.proxyRedisService.exists()) {
        Q.log.info({ proxies: proxies.length }, 'Creating index
set of proxies in Redis')
        await this.proxyRedisService.create(proxies)
    }
} finally {
    if (lock) {
        await lock.release().catch(Error, err => {
            Q.log.warn({ err }, 'unable to release lock')
        })
    }
}
}

async tryToCreateDefaultDomainSet() {
    const defaultDomainExists = await this.domainRedisService.do
mainExists('default')
    if (!defaultDomainExists) {
        await this.tryToCreateDomain('default')
    }
}

async tryToCreateDomain(domain) {
    const foundPolicy = await this.tryToCreateAndGetPolicy(domai
n)
    if (!foundPolicy) {
        await this.domainRedisService.cloneDomain(domain, 'default
')
        return
    }
    let lock
    try {
        lock = await this.lockFactory.acquire(`proxy:${domain}:loc
k`, { timeout: 20000, retries: 20, delay: 1000 })
        if (!lock) return
        Q.log.info({ domain }, 'Trying to insert proxies into Redi
s')
        const maxProxies = 1000000

```

```
    const proxies = await this.proxyPolicyMongoService.findByDomain(domain, 1, maxProxies)
    if (_.isEmpty(proxies)) {
      Q.log.info({ domain }, 'No proxies found and domain set wont be created in Redis')
      return
    }
    Q.log.info({ domain, proxies: proxies.length }, 'Proxies found and domain set will be created in Redis')
    await this.domainRedisService.insertProxies(domain, proxies)
  } finally {
    if (lock) {
      await lock.release().catch(Error, err => {
        Q.log.warn({ err }, 'unable to release lock')
      })
    }
  }
}

async tryToCreateAndGetPolicy(domain) {
  const policyFromRedis = await this.policyRedisService.findPolicy(domain)
  if (policyFromRedis) {
    return policyFromRedis
  }

  const policyFromMongo = await this.policyMongoService.findOneByDomain(domain)
  if (policyFromMongo) {
    await this.policyRedisService.createPolicy(policyFromMongo)
    return policyFromMongo
  }

  const defaultPolicyFromMongo = await this.policyMongoService.findOneByDomain('default')
  if (defaultPolicyFromMongo) {
    await this.policyRedisService.createPolicy(defaultPolicyFromMongo)
  }
}
```

```
        return defaultPolicyFromMongo
    }
}

async removeAllDomains() {
    await this.domainRedisService.removeAll()
}
}
```

First indicator that something is wrong is number of parameters in the constructor. Many parameters in the constructor tells us there are too many responsibilities in this class. We need to solve it by moving the code somewhere else. But how do we know what code is supposed to be moved and where?

WIP: Dec 5 2017

Command

- Encapsulate a request as an object, thereby letting you parametrize clients with different requests, queue or log requests, and support undoable operations.
- Promote "invocation of a method on an object" to full object status
- An object-oriented callback

Example - Unify REST and RCP calls

Command patter is going to help us to encapsulate REST and RCP call.

```
class State {  
  
    private String data1;  
    private String data2;  
  
    public String getData1() {  
        return data1;  
    }  
  
    public void setData1(String data1) {  
        this.data1 = data1;  
    }  
  
    public String getData2() {  
        return data2;  
    }  
  
    public void setData2(String data2) {  
        this.data2 = data2;  
    }  
}  
  
interface Command {  
    void execute();  
}
```

```
class RestCommand implements Command {

    private State state;

    public RestCommand(State state) {
        this.state = state;
    }

    @Override
    public void execute() {
        System.out.println(getClass().getSimpleName() + " executing...");
    }
}

class RPCCommand implements Command {

    private State state;

    public RPCCommand(State state) {
        this.state = state;
    }

    @Override
    public void execute() {
        System.out.println(getClass().getSimpleName() + " executing...");
    }
}

class CommandExecutor {

    private List<Command> commands = new ArrayList<>();

    public void add(Command command) {
        commands.add(command);
    }

    public void execute() {
```



```
        commands.stream().forEach(Command::execute);  
    }  
}
```

Here is how we could call the commands. But we could also run just one command, it does not mean we have to always put all command into a collection of commands.

```
State state = new State();  
  
Command restCommand = new RestCommand(state);  
Command rpcCommand = new RPCCommand(state);  
  
CommandExecutor executor = new CommandExecutor();  
executor.add(restCommand);  
executor.add(rpcCommand);  
executor.execute();
```

Here is the output.

```
RestCommand executing...  
RPCCommand executing...
```

State

- Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- An object-oriented state machine
- wrapper + polymorphic wrappee + collaboration

Example - UI states

We are going to use state pattern to implement different states of a login screen.

```
class TextField {  
}  
  
class Button {  
}  
  
interface State {  
    void usernameTextField(TextField textField);  
  
    void passwordTextField(TextField textField);  
  
    void loginButtonTextField(Button button);  
}  
  
class EmptyState implements State {  
  
    @Override  
    public void usernameTextField(TextField textField) {  
        // textField.setEnabled(true);  
    }  
  
    @Override  
    public void passwordTextField(TextField textField) {  
        // textField.setEnabled(true);  
    }  
}
```

```
@Override
public void loginButtonTextField(Button button) {
    // textField.setEnabled(false);
}
}

class ReadyState implements State {

    @Override
    public void usernameTextField(TextField textField) {
        // textField.setEnabled(true);
    }

    @Override
    public void passwordTextField(TextField textField) {
        // textField.setEnabled(true);
    }

    @Override
    public void loginButtonTextField(Button button) {
        // textField.setEnabled(true);
    }
}

class UserLoginUI {
    private State state;

    public UserLoginUI() {
        this.state = new EmptyState();
    }

    public void applyState(State state) {
        this.state = state;
        this.state.usernameTextField(null);
        this.state.passwordTextField(null);
        this.state.loginButtonTextField(null);
    }
}
```

Here is how we would use the states.

```
UserLoginUI loginUI = new UserLoginUI();

// user just opened the login dialog
loginUI.applyState(new EmptyState());

// when user fills in username and password
loginUI.applyState(new ReadyState());

// when user clicks on Login button and it fails
//loginUI.setState(new ErrorState());
```

Visitor

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Example - Shopping cart

We will implement a shopping cart full of vegetables that we will be able to visit and get its weight (so we can then get the price for it).

```
interface Visitorable {
    int accept(Visitor visitor);
}

interface Visitor {
    int getWeight(Apple apple);
    int getWeight(Banana banana);
}

class VisitorImpl implements Visitor {

    @Override
    public int getWeight(Apple apple) {
        return apple.getWeight();
    }

    @Override
    public int getWeight(Banana banana) {
        return banana.getWeight();
    }
}

class Apple implements Visitorable {

    private int weight;
```

```
    public Apple(int weight) {
        this.weight = weight;
    }

    public int getWeight() {
        return weight;
    }

    @Override
    public int accept(Visitor visitor) {
        return visitor.getWeight(this);
    }
}

class Banana implements Visitorable {

    private int weight;

    public Banana(int weight) {
        this.weight = weight;
    }

    public int getWeight() {
        return weight;
    }

    @Override
    public int accept(Visitor visitor) {
        return visitor.getWeight(this);
    }
}
```

Now we are going to create apple and banana and visit them in order to get its weight. It is going to return sum equal to 250.

```
Visitor visitor = new VisitorImpl();
Visitorable[] visitorables = {new Apple(100), new Banana(150)};
Long sum = Arrays
    .stream(visitorables)
    .collect(Collectors.summarizingInt(it -> it.accept(visitor)))
    .getSum();
System.out.println(sum);
```

Memento

Memento patter helps us to persist state of application in given time, so we can easily roll back or traverse through history of changes.

Example - Save game progress

We are going to use memento pattern to save game whenever user reaches a checkpoint.


```
class Game {
    private Checkpoint checkpoint;

    public void setCheckpoint(Checkpoint checkpoint) {
        this.checkpoint = checkpoint;
    }
}

class Checkpoint implements Cloneable {

    private int exp;

    public Checkpoint(int exp) {
        this.exp = exp;
    }

    public Checkpoint clone() {
        return new Checkpoint(exp);
    }
}

class GameProgressKeeper {

    private List<Checkpoint> checkpoints = new ArrayList<>();

    public void save(Checkpoint checkpoint) {
        checkpoints.add(checkpoint.clone());
    }

    public Checkpoint getLastOne() {
        return checkpoints.get(checkpoints.size() - 1);
    }
}
```

Here is how game could be played and what would have to happen in order to persist or roll back game state.

```
Game game = new Game();

// playing game... until the player reaches the first checkpoint
Checkpoint checkpoint1 = new Checkpoint(100); // other values...
game.setCheckpoint(checkpoint1);

GameProgressKeeper progressKeeper = new GameProgressKeeper();
progressKeeper.save(checkpoint1);

// continue playing game... until the player reaches the second
// checkpoint
// but the player died, roll back the game to the last saved che
// ckpoint
progressKeeper.getLastOne();
game.setCheckpoint(checkpoint1);
```

Interpreter

- Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
- Map a domain to a language, the language to a grammar, and the grammar to a hierarchical object-oriented design.

Example - Expression with variables

We are going to use interpreter to evaluate mathematical expression with variables.

```
class Evaluator {
    private Expression syntaxTree;

    public Evaluator(String expression) {
        Stack<Expression> expressionStack = new Stack<>();

        for (String token : expression.split(" ")) {

            if (token.equals("+")) {
                Expression left = expressionStack.pop();
                Expression right = expressionStack.pop();
                Expression subExpression = new Plus(left, right)
;
                expressionStack.push(subExpression);
            } else if (token.equals("-")) {
                Expression right = expressionStack.pop();
                Expression left = expressionStack.pop();
                Expression subExpression = new Minus(left, right
);
                expressionStack.push(subExpression);
            } else if (Character.isDigit(token.charAt(0))) {
                expressionStack.push(new Number(Integer.valueOf(
token)));
            } else {
```

```
        expressionStack.push(new Variable(token));
    }
}
syntaxTree = expressionStack.pop();
}

public int interpret(Map<String, Expression> variables) {
    return syntaxTree.intepret(variables);
}
}

interface Expression {
    int intepret(Map<String, Expression> variables);
}

class Number implements Expression {

    private Integer number;

    public Number(Integer number) {
        this.number = number;
    }

    @Override
    public int intepret(Map<String, Expression> variables) {
        return number;
    }
}

class Plus implements Expression {

    private Expression left;
    private Expression right;

    public Plus(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

    @Override
```

```
    public int intepret(Map<String, Expression> variables) {
        return left.intepret(variables) + right.intepret(variables);
    }
}

class Minus implements Expression {

    private Expression left;
    private Expression right;

    public Minus(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

    @Override
    public int intepret(Map<String, Expression> variables) {
        return left.intepret(variables) - right.intepret(variables);
    }
}

class Variable implements Expression {

    private String name;

    public Variable(String name) {
        this.name = name;
    }

    @Override
    public int intepret(Map<String, Expression> variables) {
        return variables.get(name).intepret(variables);
    }
}
```

We define the math formula using [polish reversed notation](#): 52 1 2 + -. It is going to return 49.

```
// reverse polish notation
String expression = "52 1 2 + -";
Evaluator evaluator = new Evaluator(expression);

// variables is the context
Map<String, Expression> variables = new HashMap<>();
variables.put("w", new Number(5));
variables.put("x", new Number(10));
variables.put("z", new Number(25));

int result = evaluator.interpret(variables);
System.out.println(result);
```

Null Object

The intent of a Null Object is to encapsulate the absence of an object by providing a substitutable alternative that offers suitable default do nothing behavior. In short, a design where "nothing will come of nothing"

Example - Empty iterator instead of null

We are going to use null object to represent empty iterator. Then we use this iterator whenever we would want to return null.

```
interface Iterator<T> {
    boolean hasNext();
    T next();
}

class IteratorImpl<T> implements Iterator {

    @Override
    public boolean hasNext() {
        return true; // TODO: here would be checking if the collection has more items to iterate through
    }

    @Override
    public T next() {
        return null;
    }
}

class NullIterator<T> implements Iterator {

    @Override
    public boolean hasNext() {
        return false;
    }

    @Override
    public T next() {
        return null;
    }
}
```

Instead of returning or working with null, we return instance of `NullIterator` .


```
class MyCollection {  
    private List list;  
  
    public Iterator iterator() {  
        if (list == null) {  
            return new NullIterator();  
        }  
        return new IteratorImpl(); // we would have to finish im  
plementation of this iterator  
    }  
}
```

Iterator

Iterator patterns provide way to access collection elements in sequential way.

Example - Iterating through collection

We are going to implement our own iterator to to iterate. This way we will practice iterator pattern and see how an iterator can be created.

```
interface Iterator<T> {
    boolean hasNext();
    T getNext();
}

class IteratorImpl<T> implements Iterator {

    private java.util.Iterator<T> values;

    public IteratorImpl(List<T> values) {
        this.values = values.iterator();
    }

    public boolean hasNext() {
        return values.hasNext();
    }

    public T getNext() {
        return values.next();
    }
}

// we could add Iterable interface and would define iterator() method
class UserCollection {

    private List<User> users = new ArrayList<>();
```

```
    public void add(User user) {
        users.add(user);
    }

    public Iterator<User> iterator() {
        return new IteratorImpl<>(users);
    }
}

class User {
    private String name;

    public User(String name) {
        this.name = name;
    }

    public String toString() {
        return name;
    }
}
```

Here is how we could use this iterator.

```
UserCollection users = new UserCollection();
users.add(new User("John"));
users.add(new User("Jimmy"));

Iterator<User> iterator = users.iterator();
while (iterator.hasNext()) {
    User user = iterator.getNext();
    System.out.println(user);
}
```

Example - Accessing database rows

Here we create another iterator to move away from "iterator" keyword. In this case, `ResultSet` is the iterator.

```
import java.util.*;

// simplified result set that returns on next string
interface ResultSet {
    String nextString();
    boolean hasNext();
}

class OneRowResultSet implements ResultSet {

    private Queue<String> stack;

    public OneRowResultSet(List<String> values) {
        this.stack = new ArrayDeque<>();
        stack.addAll(values);
    }

    @Override
    public String nextString() {
        return stack.poll();
    }

    @Override
    public boolean hasNext() {
        return !stack.isEmpty();
    }
}
```

Now we can use result set, fill it with values and iterate through it.

```
List<String> values = new ArrayList<>();
values.add("A");
values.add("B");
values.add("C");

ResultSet resultSet = new OneRowResultSet(values);

while (resultSet.hasNext()) {
    String value = resultSet.nextString();
    System.out.println(value);
}
```

The code above goes through all records and prints out the following.

```
A
B
C
```

Middleware

Middleware pattern is used to handle a request and a response in code (in pipeline-like manner). It can decide whether to pass execution to next middleware, it can call its own code before or after it passes execution to the next one. The middleware do not know about each other.

Middleware is created from Chain of Responsibility and Intercepting Filter patterns.

Usually, a message, context or request and response are send to each middleware function. Then it is mutated or cloned and then mutated. That way, we can extend code by adding a new middleware.

Example

We are going to create a http server that will have an ability to add middle-wares. The main reason of this code is to show how middle wares can be implemented.

```
const http = require('http');

// wrapper to easier start of http server
class Server {
  async start(port, requestHandler) {
    const server = http.createServer(requestHandler);
    server.listen(port, (err) => {
      if (err) {
        return console.log('Error', err)
      }
      console.log(`Server is listening on ${port}`)
    });
  }
}

// class to keep middle-wares in a linked list
class Node {
  constructor(value) {
```

```
    this.value = value;
    this.next = null;
  }
}

class Balast {
  constructor() {
    this.middlewares = { // linked list of middle wares
      head: undefined,
      tail: undefined
    };
  }

  // adds new middleware
  use(middleware) {
    const newNode = new Node(middleware);
    if (this.middlewares.head === undefined) {
      this.middlewares.head = newNode;
    } else {
      if (this.middlewares.tail) {
        this.middlewares.tail.next = newNode;
      } else {
        this.middlewares.head.next = newNode;
        this.middlewares.tail = newNode;
      }
    }
  }

  // triggers chain of middle-wares
  async serve(request, response) {
    const current = this.middlewares.head;
    if (current) {
      const nextOne = current.next || (() => {
      });
      await current.value(request, response, nextOne);
    }
  }

  // starts a server and registers handler that triggers chain o
  f middle-wares
```

```
async listen(port) {
  const requestHandler = async (request, response) => {
    console.log('request handler: start');
    await this.serve(request, response);
    await response.end(null);
    console.log('request handler: done');
  };
  await new Server().start(port, requestHandler);
}

const balast = new Balast(); // just random name of our http server

const parseToJson = async (request, response, next) => {
  request.myData = {some: 'json value'};
  console.log('parseToJson', request.myData);
  await next.value(request, response, next.next);
};
balast.use(parseToJson);

const enhanceMyData = async (request, response, next) => {
  request.myData.enhancedField = 123;
  console.log('enhanceMyData', request.myData);
  await next.value(request, response, next.next);
};
balast.use(enhanceMyData);

const stringifyData = async (request, response, next) => {
  let content = JSON.stringify(request.myData);
  console.log("stringifyData", content);
  await next.value(request, response, next.next);
};
balast.use(stringifyData);

const writeData = async (request, response, next) => {
  let content = JSON.stringify(request.myData);
  console.log("writeData", content);
  await response.write(content);
};
balast.use(writeData);
```



```
const port = 3000;  
Promise.resolve(balast.listen(port));
```

Clean Code Patterns

While creational, structural and behavioral pattern help us to solve specific problems, clean code patterns help us to produce code that is readable, easily understandable and easy to change. Some of them are called low-level refactoring patterns but some are rather ideas and principles to examine and be inspired by.

Clean code patterns are activities that are supposed to be done continuously over the time. Clean code patterns become are supposed to become a good habit, rather than one time solution for a problem.

| The more you don't pay, the worse your fees and payments become.

This category of design patterns is based on the following points:

- 1. Remove duplication.
- 2. Simplify your code.
- 3. Clarify you code's intent.

| -- Ward Cunningham

There is a subtle difference between code which is new, the code which is new and kept refactored and code that nobody every did refactor over many years.

With this chapter, I hope to provide more insight on how to make code that does not rot.

Extract Method

Instead of having multiple lines of code with unclear purpose, we extract that code into well named methods. By extracting these lines of code into methods, we usually give couple of variables clear scope and meaning.

This pattern is also called low-level refactoring. We need to be careful when we use this pattern. Even it is simple to do it, it does not mean we are supposed to do it always.

Rules:

- a new method needs to have a name that is easy to understand
- a new method can't use shortcuts or any abbreviations
- number of lines in method should be small, but there are some unique exceptions

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

-- Martin Fowler

This pattern is not suggesting we can't have long methods. There are good reasons to have a long method that does one thing well, but maybe it is the nature of that one thing to be long. Here is an example that could be reasonable long method. In this example, let's say we are reading data from a data source and would rather keep it atomic. I have seen these kinds of methods that were reading data from old mainframes in foreign language and the most readable way to keep them, was to keep that code together.

```
class DbEntry {
    int id;
    int index1;
    int index2;
    int index3;
    int index4;
    int index5;
    int index6;
    int index7;
    int index8;
    int index9;
    int index10;
}

public DbEntry build(CrazyOldEntity entity) {
    DbEntry entry = new DbEntry();
    entry.id = entity.getKey();
    entry.index1 = entity.getKattu();
    entry.index2 = entity.getKrekos();
    entry.index3 = entity.getIKA();
    entry.index4 = entity.getKiitos();
    entry.index5 = entity.getKiitoksia();
    entry.index6 = entity.getPerkele();
    entry.index7 = entity.getUksi();
    entry.index8 = entity.getOlut();
    entry.index9 = entity.getNimi();
    entry.index10 = entity.getMarenhaimen();
    return entry;
}
```

Example

Lets say we have the following code.

```
public String getNameWithBirthDate() {
    java.util.Calendar c = java.util.Calendar.getInstance();
    c.set(2005, java.util.Calendar.NOVEMBER, 20);

    String name = "John ";
    name += c.toInstant().toString();
    return name;
}
```

The code gets messy when we start playing with `Calendar` class. When we apply `Define Method` pattern, we extract all the code to separate unit of functionality.

```
public String getNameWithBirthDate() {
    Instant instant = getBirthDate();
    return getName(instant);
}

private String getName(Instant instant) {
    String name = "John ";
    name += instant.toString();
    return name;
}

private Instant getBirthDate() {
    java.util.Calendar c = java.util.Calendar.getInstance();
    c.set(2005, java.util.Calendar.NOVEMBER, 20);
    return c.toInstant();
}
```

Clarify Responsibility

What ever code we produce, we always clarify responsibility by refactoring. We might want to pick some of other design patterns to clarify the responsibility.

Example - Builder

Now a days there are not many opportunities to find code that is really wrong and really dumb. I got lucky today because I got this beauty. This class pretends to be responsible for user creation. But is does also generates user ID using a datasource. There are also other things wrong:

- stores and resets state in static variable
- is using Singleton pattern without reason
- all methods are static
- it is difficult to test
- is not thread safe, if this is called from multiple threads, it mixes up couple of users
- is using ancient naming conventions
- and there is no test for this class (maybe luckily, there is not test)

Here is the class, properly aged for 4 years in production.

```
import org.apache.commons.lang.StringEscapeUtils;
import org.apache.commons.validator.routines.EmailValidator;
import org.mongodb.morphia.Datastore;

import java.util.HashSet;
import java.util.List;
import java.util.Set;

/**
 * Allows easy creation of valid, unique users in a given databa
se.
 * <p>
 * This user will have a set ID based on the database is it inte
```

ended for. The password will also be encrypted based on the PasswordService's encryptPassword function. All fields besides password and username will be Javascript escaped for Mongo. The username will be validated as an email address.

```
*
* @since June 2013
*/
public final class UserBuilder {

    /**
     * User builder instance to be returned.
     */
    private static final UserBuilder USER_INSTANCE = new UserBuilder();

    /**
     * The user to be CREATED.
     */
    private static UserDTO sUser = new UserDTO();

    /**
     * The plaintext version of the user's password.
     */
    private static String sPlaintextPassword = null;

    /**
     * The datastore which the user will be intended for.
     */
    private static Datastore sDatastore = null;

    /**
     * Private constructor to prevent instances being CREATED.
     */
    private UserBuilder() {
        // hide the constructor
    }

    /**
     * Reset the builder to its initial state. Mainly for testing.
     */
}
```

```
    */
    protected static void reset() {
        sUser = new UserDTO();
        sDatastore = null;
        sPlaintextPassword = null;
    }

    /**
     * Returns the CREATED user after validating its fields for
     the datastore.
     * <p>
     * The user will have its password encrypted, and its ID wil
     l be set based on the provided datastore.
     *
     * @return The CREATED User object.
     */
    public static UserDTO create() throws RuntimeException {
        checkAllFieldsExist();
        if (sDatastore == null) {
            throwRuntimeException("Must provide datastore.");
        }
        sUser.setID(generateID());
        validatePassword();
        sUser.setCredentials(encryptPassword(sPlaintextPassword)
);
        validateUserEmail();

        final UserDTO toReturn = sUser;
        reset();
        return toReturn;
    }

    /**
     * Returns the CREATED user after validating its fields for
     the datastore.
     * <p>
     * The user will have its password encrypted, but no ID will
     be set and the datastore (if provided) will be ignored.
     *
     * @return The CREATED User object.
     */
```



```
    */
    protected static UserDTO createWithoutDatastore() throws RuntimeException {
        checkAllFieldsExist();
        validatePassword();
        sUser.setCredentials(encryptPassword(sPlaintextPassword)
    );
        validateUserEmail();

        final UserDTO toReturn = sUser;
        reset();
        return toReturn;
    }

    /**
     * The user will be validated and its ID will be set for the given datastore.
     *
     * @param ds The database this user will be intended for after creation.
     * @return The UserBuilder instance.
     */
    public static UserBuilder forDatabase(final Datastore ds) {
        sDatastore = ds;
        return USER_INSTANCE;
    }

    /**
     * Set the username, which should be a valid email address.
     *
     * @param username The email and username for this user.
     * @return The UserBuilder instance.
     */
    public static UserBuilder withEmailAsUsername(final String username) {
        sUser.setUsername(username);
        sUser.setEmail(username);
        return USER_INSTANCE;
    }
}
```

```
/**
 * Set the password, which will be encrypted.
 * <p>
 * The password should meet the following criteria: -8 characters or longer -Contains at least three of these character types: -lowercase letter -uppercase letter -special character -number
 *
 * @param password The user's password.
 * @return The UserBuilder instance.
 */
public static UserBuilder withPassword(final String password) {
    sPlaintextPassword = password;
    return USER_INSTANCE;
}

/**
 * Set the username, which should be a valid email address.
 *
 * @param firstName The user's first name.
 * @param lastName The user's last name.
 * @return The UserBuilder instance.
 */
public static UserBuilder withName(final String firstName, final String lastName) {
    sUser.setFirstName(StringEscapeUtils.escapeJavaScript(firstName));
    sUser.setLastName(StringEscapeUtils.escapeJavaScript(lastName));
    return USER_INSTANCE;
}

/**
 * Set user's type.
 *
 * @param type The users type.
 * @return The UserBuilder instance.
 */
public static UserBuilder withType(final String type) {
```

```
sUser.setType(StringEscapeUtils.escapeJavaScript(type));
return USER_INSTANCE;
}

/**
 * Set the user's roles. These will be used with Shiro's authentication.
 *
 * @param roles The roles for this user.
 * @return The UserBuilder instance.
 */
public static UserBuilder withRoles(final String[] roles) {
    final Set<String> roleSet = new HashSet<>();
    for (final String r : roles) {
        roleSet.add(StringEscapeUtils.escapeJavaScript(r));
    }
    sUser.setRoles(roleSet);
    return USER_INSTANCE;
}

public static UserBuilder withAuthorizedGroups(final List<BatchGroup> authorizedGroups) {
    sUser.setAuthorizedGroups(authorizedGroups);
    return USER_INSTANCE;
}

private static void checkAllFieldsExist() throws RuntimeException {
    if (fieldInvalid(sUser.getUsername())) {
        throwRuntimeException("Must provide username.");
    }
    if (fieldInvalid(sPlaintextPassword)) {
        throwRuntimeException("Must provide password.");
    }
    if (fieldInvalid(sUser.getFirstName())) {
        throwRuntimeException("Must provide first name.");
    }
    if (fieldInvalid(sUser.getLastName())) {
        throwRuntimeException("Must provide last name.");
    }
}
```

```
        if (fieldInvalid(sUser.getType())) {
            throwRuntimeException("Must provide type.");
        }
        final Set roles = sUser.getRoles();
        if (roles == null || roles.size() == 0) {
            throwRuntimeException("Must provide roles.");
        }
    }

    private static boolean fieldInvalid(final String str) {
        return str == null || str.length() < 1;
    }

    private static void validateUserEmail() throws RuntimeException {
        final EmailValidator ev = EmailValidator.getInstance();
        if (!(ev.isValid(sUser.getEmail()) && ev.isValid(sUser.getUsername())) {
            throwRuntimeException("Username was not a valid email address.");
        }
    }

    /**
     * Ensures that the password meets the following criteria: -
     * 8 characters or longer -Contains at least three of these character
     * types: -lowercase letter -uppercase letter -special character
     * -number
     */
    private static void validatePassword() throws RuntimeException {
        validatePassword(sPlaintextPassword);
    }

    public static void validatePassword(final String plainTextPassword) {
        if (plainTextPassword.length() < 8) {
            throwRuntimeException("Password is shorter than 8 characters.");
        }
    }
}
```

```
    }
    short hasNumber = 0;
    short hasLower = 0;
    short hasUpper = 0;
    short hasSpecial = 0;
    for (int i = 0; i < plainTextPassword.length(); ++i) {
        final char c = plainTextPassword.charAt(i);
        if ('a' <= c && c <= 'z') {
            hasLower = 1;
        } else if ('A' <= c && c <= 'Z') {
            hasUpper = 1;
        } else if ('0' <= c && c <= '9') {
            hasNumber = 1;
        } else {
            hasSpecial = 1;
        }
    }
    if (hasLower + hasNumber + hasUpper + hasSpecial >= 3) {
        return;
    }
    throwRuntimeException("Password must contain 3 of: lower
case, uppercase, numbers, special chars.");
}

    public static void withDefaultProficiency(final String profi
ciency) {
        sUser.setProficiency(proficiency);
    }

    private static String encryptPassword(final String cleartext
) {
        final PasswordService qps = new PasswordService();
        return qps.encryptPassword(cleartext);
    }

    private static String generateID() {
        final IdDAO IdDAO = new IdDAO(sDatastore);
        return IdDAO.generateId(User.class).toString();
    }
}
```

```
private static void throwRuntimeException(final String message) throws RuntimeException {
    reset();
    throw new RuntimeException(message);
}

public static String getPassword(final String plainTextPassword) {
    validatePassword(plainTextPassword);
    return encryptPassword(plainTextPassword);
}
}
```

Normally we would start with a test and then we could refactor the code in a responsible way. But this code is extremely hard to test. You would have to probably connect to database get MongoDB to test creation of users. Rather than that, we dare to modify it and as we refactor, we create tests to verify it is all working.

We need to clarify responsibility of this class first. The most important thing to do is to remove dependency on `Database` and `PasswordService` classes. In order to do that, we will have to:

- remove all static keywords from the code, we want to work with instance, not with a static variable
- extract code that generates password to other class, e.g. `IdGenerator`
- create methods `withEncryptedPassword(String password)` and `withId(String id)`
- replace `forDatabase` with `withId(String id)` method
- remove all class fields
- rename `create()` to `build()` because this class will be a builder that builds stuff
- remove `throwRuntimeException` method because it brings more code than actually executing
- fix misleading naming
- remove useless comments
- replace `fieldInvalid` method with a standard `isBlank` method
- move `validatePassword` to another class, e.g. `PasswordValidator`

Extract `generateID` into its own interface, because `IdGenerator` goes into database and we want to make our code testable. This way, we create a classes with clear responsibility, generate id.

```
public interface IdGenerator {  
    String generateID();  
}
```

Implement `IdGenerator` for MongoDB.

```
import org.mongodb.morphia.Datastore;  
  
@Component  
public class IdGeneratorMongo implements IdGenerator {  
  
    @Autowired  
    @Qualifier("userDatastore")  
    private Datastore datastore;  
  
    public String generateID() {  
        IdDAO idDAO = new IdDAO(datastore);  
        return idDAO.generateId(User.class).toString();  
    }  
}
```

Create `PasswordValidator` , to creates a class with clear responsibility, and it is validation of password.

```
/**
 * Ensures that the password that is 8 characters or longer, con
 * tains at least three of these
 * character types: lowercase letter, uppercase letter, special
 * character, number
 */
public class PasswordValidator {

    public void validatePassword(String plainTextPassword) {
        if (plainTextPassword.length() < 8) {
            throw new RuntimeException("Password is shorter than
8 characters.");
        }
        short hasNumber = 0;
        short hasLower = 0;
        short hasUpper = 0;
        short hasSpecial = 0;
        for (int i = 0; i < plainTextPassword.length(); ++i) {
            final char c = plainTextPassword.charAt(i);
            if ('a' <= c && c <= 'z') {
                hasLower = 1;
            } else if ('A' <= c && c <= 'Z') {
                hasUpper = 1;
            } else if ('0' <= c && c <= '9') {
                hasNumber = 1;
            } else {
                hasSpecial = 1;
            }
        }
        if (hasLower + hasNumber + hasUpper + hasSpecial >= 3) {
            return;
        }
        throw new RuntimeException("Password must contain 3 of:
lowercase, uppercase, numbers, special chars.");
    }
}
```

Here is the refactored `UserBuilder` . `UserBuilder` has clarified responsibility: build a valid user object.


```
import org.apache.commons.lang.StringEscapeUtils;
import org.apache.commons.lang3.StringUtils;
import org.apache.commons.validator.routines.EmailValidator;

import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class UserBuilder {

    private UserDTO user;

    public UserBuilder() {
        this.user = new UserDTO();
    }

    public UserDTO build() throws RuntimeException {
        checkAllFieldsExist(user);
        validatePassword(user);
        validateUserEmail(user);
        return user;
    }

    public UserBuilder withId(String id) {
        user.setID(id);
        return this;
    }

    public UserBuilder withEmailAsUsername(final String email) {
        user.setUsername(email);
        user.setEmail(email);
        return this;
    }

    public UserBuilder withPassword(final String password) {
        user.setCredentials(password);
        return this;
    }
}
```

```
    public UserBuilder withName(final String firstName, final String lastName) {
        user.setFirstName(StringEscapeUtils.escapeJavaScript(firstName));
        user.setLastName(StringEscapeUtils.escapeJavaScript(lastName));
        return this;
    }

    public UserBuilder withType(String type) {
        user.setType(StringEscapeUtils.escapeJavaScript(type));
        return this;
    }

    public UserBuilder withRoles(String[] roles) {
        Set<String> roleSet = new HashSet<>();
        for (final String r : roles) {
            roleSet.add(StringEscapeUtils.escapeJavaScript(r));
        }
        user.setRoles(roleSet);
        return this;
    }

    public UserBuilder withAuthorizedGroups(List<BatchGroup> authorizedGroups) {
        user.setAuthorizedGroups(authorizedGroups);
        return this;
    }

    public UserBuilder withDefaultProficiency(String proficiency) {
        user.setProficiency(proficiency);
        return this;
    }

    private void checkAllFieldsExist(UserDTO user) throws RuntimeException {
        if (StringUtils.isBlank(user.getUsername())) {
            throw new RuntimeException("Must provide username.");
        }
    }
}
```

```
    }
    if (StringUtils.isBlank(user.getCredentials())) {
        throw new RuntimeException("Must provide password.")
    }
    if (StringUtils.isBlank(user.getFirstName())) {
        throw new RuntimeException("Must provide first name.");
    }
    if (StringUtils.isBlank(user.getLastName())) {
        throw new RuntimeException("Must provide last name.");
    }
    if (StringUtils.isBlank(user.getType())) {
        throw new RuntimeException("Must provide type.");
    }
    final Set roles = user.getRoles();
    if (roles == null || roles.size() == 0) {
        throw new RuntimeException("Must provide roles.");
    }
}

private void validateUserEmail(UserDTO user) throws RuntimeException {
    EmailValidator validator = EmailValidator.getInstance();
    if (!(validator.isValid(user.getEmail()) && validator.isValid(user.getUsername()))) {
        throw new RuntimeException("Username was not a valid email address.");
    }
}

private void validatePassword(UserDTO user) throws RuntimeException {
    new PasswordValidator().validatePassword(user.getCredentials());
}
}
```

In order to see how the newly refactored class is going to be used, have a look at this sample of test code that was created while refactoring.

```
import org.jetbrains.spek.api.Spek
import org.jetbrains.spek.api.dsl.given
import org.jetbrains.spek.api.dsl.it
import org.jetbrains.spek.api.dsl.on
import kotlin.test.assertEquals
import kotlin.test.assertFailsWith
import kotlin.test.assertNull

class UserBuilderTest : Spek({
    given("user details") {
        val id = "id"
        val username = "user@email.com"
        val password = "Password123!"
        val encryptedPassword = PasswordService().encryptPassword(password)

        val firstName = "firstName"
        val lastName = "lastName"
        val type = "type"
        val roles = arrayOf("role")
        val authorizedGroups = mutableListOf(BatchGroup())
        val proficiency = "proficiency"

        on("build user with all fields") {
            val user = UserBuilder()
                .withId(id)
                .withEmailAsUsername(username)
                .withPassword(encryptedPassword)
                .withName(firstName, lastName)
                .withType(type)
                .withRoles(roles)
                .withAuthorizedGroups(authorizedGroups)
                .withDefaultProficiency(proficiency)
                .build()

            it("contains all all fields") {
                assertEquals(user.ID, id)
                assertEquals(user.username, username)
            }
        }
    }
})
```

```
        assertEquals(user.email, username)
        assertEquals(user.credentials, encryptedPassword
    )

        assertEquals(user.firstName, firstName)
        assertEquals(user.lastName, lastName)
        assertEquals(user.type, type)
        assertEquals(user.roles.stream().findFirst().get
    (), roles.get(0))
        assertEquals(user.authorizedGroups, authorizedGr
    oups)

        assertEquals(user.proficiency, proficiency)
    }
}

on("build user only with mandatory fields") {
    val user = UserBuilder()
        .withqId(id)
        .withEmailAsUsername(username)
        .withPassword(encryptedPassword)
        .withName(firstName, lastName)
        .withType(type)
        .withRoles(roles)
        .build()

    it("contains mandatory fields") {
        assertEquals(user.ID, id)
        assertEquals(user.username, username)
        assertEquals(user.email, username)
        assertEquals(user.credentials, encryptedPassword
    )

        assertEquals(user.firstName, firstName)
        assertEquals(user.lastName, lastName)
        assertEquals(user.type, type)
        assertEquals(user.roles.stream().findFirst().get
    (), roles.get(0))
        assertNull(user.authorizedGroups)
        assertNull(user.proficiency)
    }
}
```

```
on("build user with no fields") {  
  it("throws validation error") {  
    assertFailsWith(RuntimeException::class) {  
      UserBuilder().build()  
    }  
  }  
}  
}  
})
```

Remove Duplications

We need to only remove duplications that are somehow related. Code that is somehow related is a code that shares common domain.

Lets explain that on the following examples.

Reuse code which is already there

If we have a class that is doing the same thing but returning different type of results, we definitely want to remove duplicates.

We can see here that the `executeAndGetString` method and `executeAndGetDouble` methods are used for the same purpose, they share the domain. They are both used for the same purpose, to get average of certain set of numbers.

```
public class Executor {  
  
    public String executeAndGetString(int i) {  
        OptionalDouble average = IntStream.range(0, i).average()  
    ;  
        return String.valueOf(average.getAsDouble());  
    }  
  
    public double executeAndGetDouble(int i) {  
        OptionalDouble average = IntStream.range(0, i).average()  
    ;  
        return average.getAsDouble();  
    }  
}
```

We will refactor that code and remove duplications. Then we actually find we can reuse `extractAndGetDouble` method.

```
class RefactoredExecutor {  
  
    public String executeAndGetString(int i) {  
        return String.valueOf(executeAndGetDouble(i));  
    }  
  
    public double executeAndGetDouble(int i) {  
        OptionalDouble average = IntStream.range(0, i).average()  
;  
        return average.getAsDouble();  
    }  
}
```

Do not remove duplicates always

There are some occasions, when we shouldn't reuse the code and we should rather keep it separate. This can be difficult to judge, but let's decide after we see this example.

In this example, there are two different entities, Apple and Car. These two different entities, have different methods `getName` and `getCount`. Anyway, it seems there is a code duplication. What happens if we remove this duplicated code?


```
import java.util.stream.LongStream;

class Apple {
    private long size;

    public String getName() {
        long count = LongStream.range(0, size).count();
        return String.valueOf(count);
    }
}

class Car {
    private long power;

    public String getCount() {
        long count = LongStream.range(0, power).count();
        return String.valueOf(count);
    }
}
```

First option to remove duplicate is to create a parent for both classes.

```
class Parent {
    public String getString(long number) {
        long count = LongStream.range(0, number).count();
        return String.valueOf(count);
    }
}

class Apple extends Parent {
    private long size;

    public String getName() {
        return getString(size);
    }
}

class Car extends Parent {
    private long power;

    public String getCount() {
        return getString(power);
    }
}
```

That would be probably the worst thing we could do. We would make very strong relationship between the classes that had nothing in common previously. Since this is not the way, what else we can do?

We can replace inheritance by composition. That means, we can move the code to common class, like `Utils`, and call it whenever we need to get the string. The following solution is better than the one using inheritance. But still, it says there is some kind of relationship between `Apple` and `Car`. But in reality, there is no relationship between `Apple` and `Car`.

```
class Utils {
    public static String getString(long number) {
        long count = LongStream.range(0, number).count();
        return String.valueOf(count);
    }
}

class Apple {

    private long size;

    public String getName() {
        return Utils.getString(size);
    }
}

class Car {

    private long power;

    public String getCount() {
        return Utils.getString(power);
    }
}
```

Other point to consider is: what would happen if I have to change behavior of `getName` method in `Apple` class. I would have to update `Utils.getString` method, but then I would change behavior of `Car` class.

One solution could be add another method to the `Utils` method and add the new requested behavior there. This is would be most tempting and usually is most preferred, because it follows the same principle, keep these kind of methods in `Utils` class. Many junior developers would follow this way. But it is wrong. First, it creates duplicated code and second, it is moving code that belongs to `Car` and `Apple` classes somewhere else. It makes the code less understandable. Also, what if someone decides to reuse this code in `Universe` class? We are not far away from a disaster.

```
class Utils {  
    public static String getStringForCar(long number) {  
        long count = LongStream.range(0, number).count();  
        return String.valueOf(count);  
    }  
    public static String getStringForApple(long number, String otherInput) {  
        long count = LongStream.range(0, number).count();  
        return otherInput + " " + String.valueOf(count);  
    }  
}
```

What is the other solution? The logical implication is to stop using

`Utils.getName` in `Apple` class and create its own implementation. But then we would end up with the same code as we started, with duplicates. And we would probably make it more complicated for other developers. They could ask, "Why did they created `Utils` method if the method is used only on one place? That makes no sense!".

Reusing code and removing duplicated is very important, but everything has its price. It is up to us to judge and decide, how much are we willing to pay.

Keep Refactoring

There are the following false assumptions about code:

- it can be written well during when it is first implemented
- should be changed only if required by business
- not every class, file or function needs a test

We all know the code needs to be continuously integrated, we need to merge our code often to avoid long living branches in our source code repository. We know the code needs to be continuously deployed, to get feedback if the code is deployable and working properly. Also, we need to remember that the code needs to be continuously evolved and thus refactored.

How to make refactoring possible

In order to be able to refactor the code, we need to rely on tests. Tests are mandatory building blocks of software. If the application is not fully covered by tests, developers cannot consider a project as a success. Even when it was delivered on time.

The code needs to be maintainable. In other words, other people who did not develop the code, must be willing to change it. They need to do it in a confident way. If they change the code, they run the tests. If tests fail, they know they did something wrong and that they need to go back and rethink the change.

Sign of poor quality: "Applications or service is redeployed multiple times a day in order to fix a single bug."

When we should refactor

This needs to be discussed within the team. But the general idea is to make the code better with each pull request. There could be too many changes in a pull request. Therefore we should refactor only what is related to our changes.

If we got spaghetti code, we might end up with huge refactoring pull requests, because everything is connected to everything. These refactorings should replace spaghetti code with better structured code. So other pull requests will be not that big.

Those who are suggesting to put the refactoring changes into a special "refactoring" pull requests, to keep changes separate and easier code review process, might be wrong. They might ask you to put the refactoring changes into other pull requests, but it might be difficult to split the changes into multiple pull requests. The refactoring itself can be difficult. If we add one more difficulty, which is, how to split the changes to multiple pull request, we might end up not doing the refactoring at all. Because the refactoring become nearly impossible. Instead, suggest to and put the refactoring changes into separate commits.

Always Unit Test

There is a concept of [test pyramid](#). It suggests that there should be more unit tests than integration tests. And there should be more integration tests than e2e tests.

When to write unit test

When we should create a unit test?

- If we follow Test Driven Development, we always start with a test. Most likely, it is going to be a unit test.
- If we don't follow TDD, we should create a unit test for each unit, a unit is a method or function.
- When we are fixing a bug (and no tests are failing), we should first replicate that by in a unit test.

Are there any exceptions

What are the exceptions? There can be a reason to not to write a unit test. The easy identifier of such a situation is length of setup for a unit test. If we have to mock half of the application, and our setUp method has a lot of lines, lets say more than 50, we might face to code which is not testable, it contains too many static initialization blocks and too many final keywords. We need to use tools like PowerMock, that can modify byte code and make the code testable by force.

Are there any excuses

Many.

One excuse I was personally fighting with was, "Should I write unit tests for code that might not be used, because it is just PoC or it is just a new product that have very low chance of success?". Why did I asked this question? Because I have spend really a lot of time to make sure our code is nearly 100% covered by tests.

We have spent a lot of time by implementing all the business logic. Keep high test coverage required extra effort. Then a project was thrown away. All that work vanished for ever.

Why to bother with tests anyway? First logical implication is that we never know what project is going to stay with us and what project is thrown away. But more importantly, we are professionals and we need to keep output of our work at the top level. We always need to create tests keeping test pyramid in mind. More unit tests than integration tests and more integration tests than end to end tests.

Create Data Type

For more complex and growing systems, it is more readable and maintainable to create type instead of using too many parameters or maps.

There is one exception. The exception is when we really need to handle dynamically changing number of parameters.

Too many parameters

As the code grows, we can end up with methods that have too many parameters.

```
class UserService {  
    public void save(int id,  
                    String firsrtname,  
                    String lastname,  
                    String street,  
                    String city,  
                    String apt,  
                    String ssn,  
                    boolean active,  
                    boolean update) {  
        // for example, it gets saved into database  
    }  
}
```

When we have so many parameters, we want to create an envelop data types that would wrap these objects. The easiest solution is to use a map, because it is provided by language itself. Lets see what is wrong with that.

Using Maps everywhere instead of classes

Lets look at an example. The controller gets some parameters. We have got only three and we could just pass them into service, imagine there 20 parameters that needs to be wrapped before they are sent to service.

It is questionable if using maps everywhere is wrong for dynamically typed languages. It depends, but many times, it is better to have data type for better maintainability.

```
class Controller {
    private Service service = new Service();
    public Map<String, Object> get(int id, String type, String name) {
        Map<String, Object> query = new HashMap<>();
        query.put("ID", id);
        query.put("TYPE", type);
        query.put("NAME", name);
        return service.find(query);
    }
}

class Service {
    public Map<String, Object> find(Map<String, Object> query) {
        HashMap<String, Object> result = new HashMap<>();
        int id = (int) query.get("ID");
        String type = (String) query.get("TYPE");
        String name = (String) query.get("NAME");
        // imagine we call some DAO to get data from database
        // String someData = dao.find(id, type, name);
        result.put("SOME_KEY", "some data from db");
        return result;
    }
}
```

What is wrong with using maps:

- We are losing information about type
- It is not clear what is the signature of the method, maps are too generic
- Keys in the maps are all around the code and it is difficult to do refactoring (we can create helper class to store the keys, but it is just creating another class which shouldn't be even created)

Using Data Type instead of Maps or too many parameters

Lets create data types instead of using the maps or instead of having so many parameters.

```
class Request {
    private final int id;
    private final String type;
    private final String name;

    public Request(int id, String type, String name) {
        this.id = id;
        this.type = type;
        this.name = name;
    }

    public int getId() {
        return id;
    }
    public String getType() {
        return type;
    }
    public String getName() {
        return name;
    }
}

class Result {
    private final String someData;

    public Result(String someData) {
        this.someData = someData;
    }

    public String getSomeData() {
        return someData;
    }
}
```

```
class Controller {
    private Service service = new Service();
    public Result get(int id, String type, String name) {
        Request request = new Request(id, type, name);
        return service.find(request);
    }
}

class Service {
    public Result find(Request query) {
        int id = query.getId();
        String type = query.getType();
        String name = query.getName();
        // imagine we call some DAO to get data from database
        // String someData = dao.find(id, type, name);
        return new Result("some data from db");
    }
}
```

Comment to Better Name

It is very easy to name methods, files, classes or functions, if we don't care about other people, or about future us. This technique help to figure out better name for while we code.

The idea is to first start with a comment in the code. Then translate that comment to name.

From a good comment to a good method name

Lets see an example. We want to create something what produces users and sends the produced users to a queue. We start with comment in the code.

```
// produces users to a queue
```

Then we translate this comment to class name, for now.

```
class UserQueueProducer {  
}
```

Then we want to implement a method that will do the work. We write a command what that method is going to do.

```
class UserQueueProducer {  
    // produces a user and sends it to the queue  
}
```

Then we translate the comment to a method name.

```
class UserQueueProducer {  
    // produces a user and sends it to the queue  
    public void produceUserAndSendUserToQueue() {  
    }  
}
```

We always review the name, if it contains any duplications, we remove them. Or sometimes, we need to change the name to make it really understandable.

```
class UserQueueProducer {  
    public void produceAndSendUserToQueue() {  
    }  
}
```

Good comment, bad name

Here is an example that shows only the first part done well, the comment provides enough information to justify what is going to happen. We have created a very good comment for the method call, `Move all the remaining urls in one slot`. But the name of the method does not reflect that good intention.

```
StagingPlanRepository stagingPlanRepository = new StagingPlanRepository();  
  
// Move all the remaining urls in one slot  
stagingPlanRepository.updateSlotForAllUrls(1);
```

Here is how the method should be named. The consequence of making the method with proper name, can mean we need to create another method and fix the confusion.

```
class StagingPlanRepository {  
    public void moveAllRemainingUrlsInOneSlot() {  
        updateSlotForAllUrls(1);  
    }  
  
    public void updateSlotForAllUrls(int slots) {  
    }  
}
```

Ideally, we would also rename `updateSlotForAllUrls` method. But that might be difficult, because we will need to go through the method code and figure out what the method is really doing.

Consistent Naming

The same things should be always use the same name in the code. Coding is not an writing exercise where we are asked to use as many synonym as possible to create high quality literature.

Look at that method variables. `offset` is renamed to `startsWith` and then renamed to `startFrom` . Then `limit` parameter is renamed to `numItems` and then `howMany` parameter.

```
class Controller {
    private Service service;

    Response listJobs(@RequestParam("offset") final int startsWith
, @RequestParam("limit") int numItems) {
        return Response.from(service.find(startsWith, numItems)).build();
    }
}

class Service {
    List<User> find(int startFrom, int howMany) {
        return null; // return something...
    }
}
```


If-else over ternary operator

We need to avoid very long lines that are using ternary operator, because it makes code very difficult to read.

Straightforward usage of ternary operator is OK, for example: `String name = user ? user.getName() : "";`

Here is an example, last line in a controller that decides what is going to be returned.

```
return result != null ? ResponseEntity.ok(result).header("Location", "/v1/task/" + result.getId()).build() : Response.status(Status.BAD_REQUEST).header("ErrorMessage", "Invalid taskdata").build();
```

What if we want to refactor such a code? We first split it into if-else branches and then we figure out what is actually happening.

Composition over Inheritance

Better to reference other classes using composition than binding code together using inheritance.

The problem

Lets look at crazy example that would over use inheritance.

```
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

@RestController
@RequestMapping("/")
public class Controller extends Service {

    @GetMapping("users")
    public ResponseEntity<List<User>> users() {
        return ResponseEntity.ok(getUsers());
    }
}

class Service extends Repository {
    List<User> getUsers() {
        List<String> users = super.findUsers();
        return users.stream().map(User::new).collect(Collectors.toList());
    }
}

class User {
```

```
private final String name;

User(String name) {
    this.name = name;
}

public String getName() {
    return name;
}
}

class Repository {
    List<String> findUsers() {
        ArrayList<String> users = new ArrayList<>();
        users.add("John");
        return users;
    }
}
```

Solution

The solution is to use composition. The code becomes more testable.

To make the code really testable, we would probably have to create couple interfaces for the service and repository. That would allow us to mock the code and write proper unit tests.

```
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

@RestController
@RequestMapping("/")
public class CompoController {
```

```
private final Service service;

public CompoController(Service service) {
    this.service = service;
}

@GetMapping("users")
public ResponseEntity<List<User>> users() {
    List<User> users = service.getUsers();
    return ResponseEntity.ok(users);
}
}

@org.springframework.stereotype.Service
class Service {
    private final Repository repository;

    Service(Repository repository) {
        this.repository = repository;
    }

    List<User> getUsers() {
        List<String> users = repository.findUsers();
        return users.stream().map(User::new).collect(Collectors.toList());
    }
}

class User {
    private final String name;

    User(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

```
@org.springframework.stereotype.Repository
class Repository {
    List<String> findUsers() {
        ArrayList<String> users = new ArrayList<>();
        users.add("John");
        return users;
    }
}
```

Too Many Returns

Too many returns (accompanied with too many if branches) lead to long, complex and hard to maintain methods.

Example

Lets take this example of validator. It takes an object and validates its properties. As the object grows, the method grows. As the validation logic grows, the method grows and gets more and more complex. We can already see multiple inner

```
if s.
```

```
const Validator = require('validator')
const _ = require('lodash')

module.exports = class {
  validate({imageUrl, proxy, useProxy, width, height, keepRatio,
    quality}) {
    if (!Validator.isURL(imageUrl) && !imageUrl.startsWith('data
:image')) {
      return 'imageUrl needs to be a valid URL'
    }

    if (proxy && useProxy) {
      if (!Validator.isURL(proxy)) {
        return 'proxy needs to be a valid URL'
      }
      if (!_.isBoolean(useProxy)) {
        return 'useProxy needs to be a boolean'
      }
    }

    if (!_.isNumber(width)) {
      return 'width needs to be a number'
    }

    if (!_.isNumber(height)) {
      return 'height needs to be a number'
    }

    if (!_.isBoolean(keepRatio)) {
      return 'keepRatio needs to be a boolean'
    }

    if (quality) {
      if (!_.isNumber(quality)) {
        return 'quality needs to be a number'
      }
    }
  }
}
```

Instead of too many `if` branches each with its own `return` statement, we can create an object that will avoid that.

```
const Validator = require('validator')
const _ = require('lodash')

function validate({imageUrl, proxy, useProxy, width, height, keepRatio, quality}) {
  return new RequestValidator()
    .withUrl(imageUrl)
    .withProxy(proxy, useProxy)
    .withWidth(width)
    .withHeight(height)
    .withKeepRatio(keepRatio)
    .withQuality(quality)
    .validate()
}

module.exports = validate

class RequestValidator {
  constructor() {
    this.errors = []
  }

  withUrl(imageUrl) {
    if (!Validator.isURL(imageUrl) && !imageUrl.startsWith('data:image')) {
      this.errors.push('imageUrl needs to be a valid URL')
    }
    return this
  }

  withProxy(proxy, useProxy) {
    if (proxy && useProxy) {
      if (!Validator.isURL(proxy)) {
        this.errors.push('proxy needs to be a valid URL')
      }
    }
    if (!_isBoolean(useProxy)) {
      this.errors.push('useProxy needs to be a boolean')
    }
  }
}
```



```
    }
  }
  return this
}

withWidth(width) {
  if (!_.isNumber(width)) {
    this.errors.push( 'width needs to be a number')
  }
  return this
}

withHeight(height) {
  if (!_.isNumber(height)) {
    this.errors.push( 'height needs to be a number')
  }
  return this
}

withKeepRatio(keepRatio) {
  if (!_.isBoolean(keepRatio)) {
    this.errors.push( 'keepRatio needs to be a boolean')
  }
  return this
}

withQuality(quality) {
  if (quality && !_.isNumber(quality)) {
    this.errors.push( 'quality needs to be a number')
  }
  return this
}

validate() {
  if (!_.isEmpty(this.errors)) {
    return _.join(this.errors, ', ')
  }
}
}
```


Private to Interface

The code grows as the code is maintained and enhanced. If there is not enough refactoring and deliberate actions, quite few private methods can pop up there. That would be probably fine but more private methods we have, more difficult the testing is. If we have 1 public method in one class and none private methods, it is very easy to test it because there is nothing hidden. But if we have a class with 1 public method and 10 private methods that do all the work, it can become very difficult to test all the if-else or try-catch branches.

Consider the following scenario. We have a class that has about 1 000 lines and contains 1 public method and 10 private methods. We change the logic of one private method in order to fix a bug. Then we run the test and nothing fails. We found that that part of code is not covered by tests. We have to options. First, try to create the test and get into that branch, which might be very difficult and even if we do it, we will have to probably prepare a lot of dummy data or we will have to do a lot of mocking. In some cases, we might be forced to do both. If we do write a test that requires a lot of mocking, the readability and maintainability of that test is absolutely not good. We always need to keep in mind that there will be other people changing the code and also the tests should be treated as production code. The other option is to move that private method somewhere else and make it testtable. We can create an interface that contains that method definition. Then we create implementation of that interface and we move the private method there and thus we make it public. Then we can create unit test and test the new class. In the existing code, we reference the new interface.

Example

Lets consider we have the following code and we want to change `findRole` method.

```
class Db {
    public String execute(String sql) {
        return "dummy role";
    }
}

class User {
    private int id;
    private String role;

    public User(int id, String role) {
        this.id = id;
        this.role = role;
    }
}

class UserService {

    private Db db;

    public User createUser(int id, int roleId) {
        validate(id);
        String role = findRole(roleId);
        return new User(id, role);
    }

    private void validate(int id) {
        if (id <= 0) throw new RuntimeException("Not valid ID, use positive number");
    }

    private String findRole(int roleId) {
        return db.execute("SELECT role FROM roles WHERE role_id = " + roleId);
    }
}
```

Instead of updating tests for UserService we create a new set of classes to handle logic of `findRole` method.

```
class Db {
    public String execute(String sql) {
        return "dummy role";
    }
}

class User {
    private int id;
    private String role;

    public User(int id, String role) {
        this.id = id;
        this.role = role;
    }
}

interface RoleService {
    String findRole(int userId);
}

class DefaultRoleService implements RoleService {

    private Db db = new Db();

    @Override
    public String findRole(int roleId) {
        if (roleId <= 0) throw new RuntimeException("Role ID needs to be positive.");
        return db.execute("SELECT role FROM roles WHERE role_id = " + roleId);
    }
}

class NewUserService {

    private RoleService roleService = new DefaultRoleService();

    public User createUser(int id, int roleId) {
        validate(id);
    }
}
```

```
        String role = roleService.findRole(roleId);
        return new User(id, role);
    }

    private void validate(int id) {
        if (id <= 0) throw new RuntimeException("Not valid ID, use positive number");
    }
}
```

Anit Patterns

To know what is good, we need to know what is wrong.

Big Ball of Mud

Big Ball of Mud anti pattern, first described by [Brian Foote and Joseph Yoder](#), is result of under-engineering. It is described as:

- everything talks to everything
- data are shared without limitations
- meaningless or misleading naming
- functions are not atomic and they have many responsibilities
- code is duplicated
- difficult to follow the code flow
- not clear intent of the code
- code is patched multiple times without refactoring

Here are some root causes of Big Ball of Mud project:

- Lack of upfront design
- Late changes to the requirements
- Late changes to the architecture
- Piecemeal growth

If we ever get to work with Big Ball of Mud, we need to follow the Boy Scouts of America.

Leave the campground cleaner than you found it.

But before we can follow that rule, we need to get understanding of that code, which is the most difficult thing to do on that kind of project.

Example of Big Ball of Mud pattern

How does a big ball of mud looks? Lets try to create a concise example of Big Ball of Mud. The following code demonstrates the feelings we get when we start reading Big Ball of Mud code. We don't know where it starts, we don't what it actually does, we don't know why it does certain things and we can see there are

confusing comments that were added as requirements changed. Also we can see there was never any design applied on that code and certainly, no refactoring was done, like never ever. One could doubt that this code passed a code review.

```
public class Executor {

    public UKF data;
    public RenderHtml service;

    public Executor(UKF data, RenderHtml service) {
        this.data = data;
        this.service = service;
    }

    public String makeData() {
        String show = service.show();
        System.out.println(show);
        // need to return empty string or it fails in other serv
ices
        return "";
    }

    public String makeOtherData() {
        String show = service.show();
        System.out.println(show);
        // need to return empty string or it fails in other serv
ices
        return "";
    }
}
```

```
public class RenderHtml {

    public static UKF ukf = new UKF();

    public static String show() {
        String input = "";
        // TODO: refactor this, probably shouldn't use toString
        input += ukf.toString();
        return input;
    }
}
```

```
public class UKF {
    private String u;
    private Integer k;

    public void doIt() {
        Executor executor = new Executor(new UKF(), new RenderHtml());
        String s = executor.makeData();
        RenderHtml.ukf.u = s;
        // JIRA-123 changed value to 100 because otherwise it makes no sense!
        RenderHtml.ukf.k = 100;
    }
}
```

Imagine these classes are used by many other classes in our application. Would you dare to change `RenderHtml.ukf.k = 100;` to `RenderHtml.ukf.k = 101;` ? If yes, why? If no, why? Nobody can tell, that is the effect of Big Ball of Mud pattern. It results in code nobody wants to change.

Singleton

Singleton pattern is mentioned in both good patterns and also in anti patterns.

Why? I think because singleton is easy to implement and many people wanted to use design patterns where ever it is possible, even when they didn't have a need to solve an issue. Then we ended up with code that is:

- difficult to use
- complex
- difficult to test

Some people suggest that Singleton is an anti-pattern because it brings more evil than good. Still, Singleton pattern is useful when it is used correctly.

A lesson to learn here is "Use Singleton only when you face an issue, never just because you think it might make the code more cool or fancy".

Example of Singleton pattern

How to create an anti pattern of Singleton pattern? It should be easy. We just need to use Singleton pattern when we shouldn't use it.

```
public class UserFactory {

    private static UserFactory userFactory;

    private UserFactory() {
    }

    public static UserFactory getInstance() {
        if (userFactory == null) userFactory = new UserFactory()
;
        return userFactory;
    }

    public User createUser(String name) {
        return new User(name);
    }
}

class User {
    String name;

    public User(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}

class Main {
    public static void main(String [] args) {
        User john = UserFactory.getInstance().createUser("John")
;
        System.out.println(john.getName());
    }
}
```

There is no reason why a developer who is using this code couldn't create an instance of `User` class by him self. The code above seems to be over-engineered.

Mad Scientist

Mad scientist's code is code produced by highly motivated, intelligent, productive but mad developer. Lets look at these aspects:

- **motivated** - usually works all the time, day and night, working day or weekend
- **intelligent** - is able to find interesting solutions for many issues
- **productive** - writes a LOT of code, I mean, like a lot!
- **mad** - cares only about functionality, not about how is the code written and tested, usually cares only if the code works while the code is produced

The code produced by mad scientist is not horrible, it just does not follow common practices, usually is not formatted, is not testable and not very well tested. Also, code is using very encrypted shortcuts. A good sign you work with code produced by mad scientist, is that you don't want to touch it.

The code usually works on their machines. They don't really enjoy communication with other developers, because the other guys are just too stupid. They think "cmon, everybody should get what is the code about by actually READING the code".

Sometimes, management thinks they are super awesome guys saving the universe. But people who are around them know how much damage they do to the company.

Example

I picked this class because I think it is crazy enough. Long methods, not testable code and many comments that are not solved (and probably won't be solved any soon). It is using shortcuts, like `PS` , `SEG` , `S` .

If anyone thinks this code is testable, try to test that constructor ;-)

```
class ParagraphText<PS, SEG, S> extends TextFlowExt {  
  
    // FIXME: changing it currently has not effect, because
```

```

    // Text.impl_selectionFillProperty().set(newFill) doesn't work
    // properly for Text node inside a TextFlow (as of JDK8-b100).
    private final ObjectProperty<Paint> highlightTextFill = new
SimpleObjectProperty<>(Color.WHITE);
    public ObjectProperty<Paint> highlightTextFillProperty() {
        return highlightTextFill;
    }

    private final Var<Integer> caretPosition = Var.newSimpleVar(
0);
    public Var<Integer> caretPositionProperty() { return caretPosition; }
    public void setCaretPosition(int pos) { caretPosition.setValue(pos); }
    private final Val<Integer> clampedCaretPosition;

    private final ObjectProperty<IndexRange> selection = new SimpleObjectProperty<>(
StyledTextArea.EMPTY_RANGE);
    public ObjectProperty<IndexRange> selectionProperty() { return selection; }
    public void setSelection(IndexRange sel) { selection.set(sel); }

    private final Paragraph<PS, SEG, S> paragraph;

    private final Path caretShape = new CaretPath();
    private final Path selectionShape = new SelectionPath();

    private final CustomCssShapeHelper<Paint> backgroundShapeHelper;
    private final CustomCssShapeHelper<BorderAttributes> borderShapeHelper;
    private final CustomCssShapeHelper<UnderlineAttributes> underlineShapeHelper;

    // proxy for caretShape.visibleProperty() that implements unbind() correctly.
    // This is necessary due to a bug in BooleanPropertyBase#unb

```

```

ind().
    // See https://bugs.openjdk.java.net/browse/JDK-8130458
    private final Var<Boolean> caretVisible = Var.newSimpleVar(false);
    {
        caretShape.visibleProperty().bind(caretVisible);
    }

    ParagraphText(Paragraph<PS, SEG, S> par, Function<StyledSegment<SEG, S>, Node> nodeFactory) {
        this.paragraph = par;

        getStyleClass().add("paragraph-text");

        int parLen = paragraph.length();
        clampedCaretPosition = caretPosition.map(i -> Math.min(i, parLen));
        clampedCaretPosition.addListener((obs, oldPos, newPos) -> requestLayout());

        selection.addListener((obs, old, sel) -> requestLayout());

        Val<Double> leftInset = Val.map(insetsProperty(), Insets::getLeft);
        Val<Double> topInset = Val.map(insetsProperty(), Insets::getTop);

        // selection highlight
        selectionShape.setManaged(false);
        selectionShape.setFill(Color.DODGERBLUE);
        selectionShape.setStrokeWidth(0);
        selectionShape.layoutXProperty().bind(leftInset);
        selectionShape.layoutYProperty().bind(topInset);
        getChildren().add(selectionShape);

        // caret
        caretShape.getStyleClass().add("caret");
        caretShape.setManaged(false);
        caretShape.setStrokeWidth(1);

```



```

        caretShape.layoutXProperty().bind(leftInset);
        caretShape.layoutYProperty().bind(topInset);
        getChildren().add(caretShape);

        // XXX: see the note at highlightTextFill
//        highlightTextFill.addListener(new ChangeListener<Paint
>() {
//            @Override
//            public void changed(ObservableValue<? extends Pain
t> observable,
//                                Paint oldFill, Paint newFill) {
//                for(PumpedUpText text: textNodes())
//                    text.impl_selectionFillProperty().set(newF
ill);
//            }
//        });

        // populate with text nodes
        par.getStyledSegments().stream().map(nodeFactory).forEac
h(n -> {
            if (n instanceof TextExt) {
                TextExt t = (TextExt) n;
                // XXX: binding selectionFill to textFill,
                // see the note at highlightTextFill
                JavaFXCompatibility.Text_selectionFillProperty(t
).bind(t.fillProperty());
            }
            getChildren().add(n);
        });

        // set up custom css shape helpers
        Supplier<Path> createBackgroundShape = () -> {
            Path shape = new BackgroundPath();
            shape.setManaged(false);
            shape.layoutXProperty().bind(leftInset);
            shape.layoutYProperty().bind(topInset);
            return shape;
        };
        Supplier<Path> createBorderShape = () -> {
            Path shape = new BorderPath();

```

```

        shape.setManaged(false);
        shape.layoutXProperty().bind(leftInset);
        shape.layoutYProperty().bind(topInset);
        return shape;
    };

    Supplier<Path> createUnderlineShape = () -> {
        Path shape = new UnderlinePath();
        shape.setManaged(false);
        shape.layoutXProperty().bind(leftInset);
        shape.layoutYProperty().bind(topInset);
        return shape;
    };

    Consumer<Collection<Path>> clearUnusedShapes = paths ->
getChildren().removeAll(paths);
    Consumer<Path> addToBackground = path -> getChildren().a
dd(0, path);
    Consumer<Path> addToForeground = path -> getChildren().a
dd(path);
    backgroundShapeHelper = new CustomCssShapeHelper<>(
        createBackgroundShape,
        (backgroundShape, tuple) -> {
            backgroundShape.setStrokeWidth(0);
            backgroundShape.setFill(tuple._1);
            backgroundShape.getElements().setAll(getRang
eShape(tuple._2));
        },
        addToBackground,
        clearUnusedShapes
    );
    borderShapeHelper = new CustomCssShapeHelper<>(
        createBorderShape,
        (borderShape, tuple) -> {
            BorderAttributes attributes = tuple._1;
            borderShape.setStrokeWidth(attributes.width)
;
            borderShape.setStroke(attributes.color);
            if (attributes.type != null) {
                borderShape.setStrokeType(attributes.typ
e);

```

```

        }
        if (attributes.dashArray != null) {
            borderShape.getStrokeDashArray().setAll(
attributes.dashArray);
        }
        borderShape.getElements().setAll(getRangeSha
pe(tuple._2));
    },
    addToBackground,
    clearUnusedShapes
);
underlineShapeHelper = new CustomCssShapeHelper<>(
    createUnderlineShape,
    (underlineShape, tuple) -> {
        UnderlineAttributes attributes = tuple._1;
        underlineShape.setStroke(attributes.color);
        underlineShape.setStrokeWidth(attributes.wid
th);
        underlineShape.setStrokeLineCap(attributes.c
ap);
        if (attributes.dashArray != null) {
            underlineShape.getStrokeDashArray().setA
ll(attributes.dashArray);
        }
        underlineShape.getElements().setAll(getUnder
lineShape(tuple._2));
    },
    addToForeground,
    clearUnusedShapes
);
}

public Paragraph<PS, SEG, S> getParagraph() {
    return paragraph;
}

public Var<Boolean> caretVisibleProperty() {
    return caretVisible;
}

```

```
public ObjectProperty<Paint> highlightFillProperty() {
    return selectionShape.fillProperty();
}

public double getCaretOffsetX() {
    layout(); // ensure layout, is a no-op if not dirty
    Bounds bounds = caretShape.getLayoutBounds();
    return (bounds.getMinX() + bounds.getMaxX()) / 2;
}

public Bounds getCaretBounds() {
    layout(); // ensure layout, is a no-op if not dirty
    return caretShape.getBoundsInParent();
}

public Bounds getCaretBoundsOnScreen() {
    layout(); // ensure layout, is a no-op if not dirty
    Bounds localBounds = caretShape.getBoundsInLocal();
    return caretShape.localToScreen(localBounds);
}

public Bounds getRangeBoundsOnScreen(int from, int to) {
    layout(); // ensure layout, is a no-op if not dirty
    PathElement[] rangeShape = getRangeShapeSafely(from, to)
;

    // switch out shapes to calculate the bounds on screen
    // Must take a copy of the list contents, not just a ref
    erence:
    List<PathElement> selShape = new ArrayList<>(selectionSh
ape.getElements());
    selectionShape.getElements().setAll(rangeShape);
    Bounds localBounds = selectionShape.getBoundsInLocal();
    Bounds rangeBoundsOnScreen = selectionShape.localToScree
n(localBounds);
    selectionShape.getElements().setAll(selShape);

    return rangeBoundsOnScreen;
}
```

```
public Optional<Bounds> getSelectionBoundsOnScreen() {
    if(selection.get().getLength() == 0) {
        return Optional.empty();
    } else {
        layout(); // ensure layout, is a no-op if not dirty
        Bounds localBounds = selectionShape.getBoundsInLocal
();
        return Optional.ofNullable(selectionShape.localToScr
een(localBounds));
    }
}

public int getCurrentLineStartPosition() {
    return getLineStartPosition(clampedCaretPosition.getValu
e());
}

public int getCurrentLineEndPosition() {
    return getLineEndPosition(clampedCaretPosition.getValue(
));
}

public int currentLineIndex() {
    return getLineOfCharacter(clampedCaretPosition.getValue(
));
}

public int currentLineIndex(int position) {
    return getLineOfCharacter(position);
}

private void updateCaretShape() {
    PathElement[] shape = getCaretShape(clampedCaretPosition
.getValue(), true);
    caretShape.getElements().setAll(shape);
}

private void updateSelectionShape() {
    int start = selection.get().getStart();
    int end = selection.get().getEnd();
}
```

```

        selectionShape.getElements().setAll(getRangeShapeSafely(
start, end));
    }

    /**
     * Gets the range shape for the given positions within the t
ext, including the newline character, if range
     * defined by the start/end arguments include it.
     *
     * @param start the start position of the range shape
     * @param end the end position of the range shape. If {@code
end == paragraph.length() + 1}, the newline character
     * will be included in the selection by selecting
the rest of the line
     */
    private PathElement[] getRangeShapeSafely(int start, int end
) {
        PathElement[] shape;
        if (end <= paragraph.length()) {
            // selection w/o newline char
            shape = getRangeShape(start, end);
        } else {
            // Selection includes a newline character.
            if (paragraph.length() == 0) {
                // empty paragraph
                shape = createRectangle(0, 0, getWidth(), getHei
ght());
            } else if (start == paragraph.length()) {
                // selecting only the newline char

                // calculate the bounds of the last character
                shape = getRangeShape(start - 1, start);
                LineTo lineToTopRight = (LineTo) shape[shape.len
gth - 4];
                shape = createRectangle(lineToTopRight.getX(), l
ineToTopRight.getY(), getWidth(), getHeight());
            } else {
                shape = getRangeShape(start, paragraph.length())
;

                // Since this might be a wrapped multi-line para

```

```

graph,
    // there may be multiple groups of (1 MoveTo, 3
LineTo objects) for each line:
    // MoveTo(topLeft), LineTo(topRight), LineTo(bot
tomRight), LineTo(bottomLeft)

    // We only need to adjust the top right and bott
om right corners to extend to the
    // width/height of the line, simulating a full l
ine selection.
    int length = shape.length;
    int bottomRightIndex = length - 3;
    int topRightIndex = bottomRightIndex - 1;
    LineTo lineToTopRight = (LineTo) shape[topRightI
ndex];
    shape[topRightIndex] = new LineTo(getWidth(), li
neToTopRight.getY());
    shape[bottomRightIndex] = new LineTo(getWidth(),
getHeight());
    }
    }

    if (getLineCount() > 1) {
        // adjust right corners of wrapped lines
        boolean wrappedAtEndPos = (end > 0 && getLineOfChara
cter(end) > getLineOfCharacter(end - 1));
        int adjustLength = shape.length - (wrappedAtEndPos ?
0 : 5);
        for (int i = 0; i < adjustLength; i++) {
            if (shape[i] instanceof MoveTo) {
                ((LineTo)shape[i + 1]).setX(getWidth());
                ((LineTo)shape[i + 2]).setX(getWidth());
            }
        }
    }

    return shape;
}

private PathElement[] createRectangle(double topLeftX, doubl

```

```

    e topLeftY, double bottomRightX, double bottomRightY) {
        return new PathElement[] {
            new MoveTo(topLeftX, topLeftY),
            new LineTo(bottomRightX, topLeftY),
            new LineTo(bottomRightX, bottomRightY),
            new LineTo(topLeftX, bottomRightY),
            new LineTo(topLeftX, topLeftY)
        };
    }

    private void updateBackgroundShapes() {
        int start = 0;

        // calculate shared values among consecutive nodes
        FilteredList<Node> nodeList = getChildren().filtered(nod
e -> node instanceof TextExt);
        for (Node node : nodeList) {
            TextExt text = (TextExt) node;
            int end = start + text.getText().length();

            Paint backgroundColor = text.getBackgroundColor();
            if (backgroundColor != null) {
                backgroundShapeHelper.updateSharedShapeRange(bac
kgroundColor, start, end);
            }

            BorderAttributes border = new BorderAttributes(text)
;
            if (!border.isNullValue()) {
                borderShapeHelper.updateSharedShapeRange(border,
start, end);
            }

            UnderlineAttributes underline = new UnderlineAttribu
tes(text);
            if (!underline.isNullValue()) {
                underlineShapeHelper.updateSharedShapeRange(unde
rline, start, end);
            }
        }
    }

```



```

        start = end;
    }

    borderShapeHelper.updateSharedShapes();
    backgroundShapeHelper.updateSharedShapes();
    underlineShapeHelper.updateSharedShapes();
}

@Override
protected void layoutChildren() {
    super.layoutChildren();
    updateCaretShape();
    updateSelectionShape();
    updateBackgroundShapes();
}

private static class CustomCssShapeHelper<T> {

    private final List<Tuple2<T, IndexRange>> ranges = new L
inkedList<>();
    private final List<Path> shapes = new LinkedList<>();

    private final Supplier<Path> createShape;
    private final BiConsumer<Path, Tuple2<T, IndexRange>> co
nfigureShape;
    private final Consumer<Path> addToChildren;
    private final Consumer<Collection<Path>> clearUnusedShap
es;

    CustomCssShapeHelper(Supplier<Path> createShape, BiConsu
mer<Path, Tuple2<T, IndexRange>> configureShape,
                        Consumer<Path> addToChildren, Consu
mer<Collection<Path>> clearUnusedShapes) {
        this.createShape = createShape;
        this.configureShape = configureShape;
        this.addToChildren = addToChildren;
        this.clearUnusedShapes = clearUnusedShapes;
    }
}

```

```

    /**
     * Calculates the range of a value (background color, underline, etc.) that is shared between multiple
     * consecutive {@link TextExt} nodes
     */
    private void updateSharedShapeRange(T value, int start,
int end) {
        Runnable addNewValueRange = () -> ranges.add(Tuples.
t(value, new IndexRange(start, end)));

        if (ranges.isEmpty()) {
            addNewValueRange.run();;
        } else {
            int lastIndex = ranges.size() - 1;
            Tuple2<T, IndexRange> lastShapeValueRange = ranges.get(lastIndex);
            T lastShapeValue = lastShapeValueRange._1;

            // calculate smallest possible position which is
            consecutive to the given start position
            final int prevEndNext = lastShapeValueRange.get2
().getEnd() + 1;
            if (start <= prevEndNext && // Consecutive?

                lastShapeValue.equals(value)) { // Same style?

                    IndexRange lastRange = lastShapeValueRange._
2;
                    IndexRange extendedRange = new IndexRange(lastRange.getStart(), end);
                    ranges.set(lastIndex, Tuples.t(lastShapeValue, extendedRange));
                } else {
                    addNewValueRange.run();
                }
        }
    }

    /**

```

```

        * Updates the shapes calculated in {@link #updateShared
ShapeRange(Object, int, int)} and configures them
        * via {@code configureShape}.
        */
    private void updateSharedShapes() {
        // remove or add shapes, depending on what's needed
        int neededNumber = ranges.size();
        int availableNumber = shapes.size();

        if (neededNumber < availableNumber) {
            List<Path> unusedShapes = shapes.subList(neededN
umber, availableNumber);
            clearUnusedShapes.accept(unusedShapes);
            unusedShapes.clear();
        } else if (availableNumber < neededNumber) {
            for (int i = 0; i < neededNumber - availableNumb
er; i++) {
                Path shape = createShape.get();

                shapes.add(shape);
                addToChildren.accept(shape);
            }

            // update the shape's color and elements
            for (int i = 0; i < ranges.size(); i++) {
                configureShape.accept(shapes.get(i), ranges.get(
i));
            }

            // clear, since it's no longer needed
            ranges.clear();
        }
    }

    private static class BorderAttributes extends LineAttributes
Base {

        final StrokeType type;

```

```

        BorderAttributes(TextExt text) {
            super(text.getBorderStrokeColor(), text.getBorderStr
okewidth(), text.borderStrokeDashArrayProperty());
            type = text.getBorderStrokeType();
        }

        @Override
        public boolean equals(Object obj) {
            if (obj instanceof BorderAttributes) {
                BorderAttributes attributes = (BorderAttributes)
obj;
                return super.equals(attributes) && Objects.equal
s(type, attributes.type);
            } else {
                return false;
            }
        }

        @Override
        public String toString() {
            return String.format("BorderAttributes[type=%s %s]",
type, getSubString());
        }
    }

    private static class UnderlineAttributes extends LineAttribu
tesBase {

        final StrokeLineCap cap;

        UnderlineAttributes(TextExt text) {
            super(text.getUnderlineColor(), text.getUnderlineWid
th(), text.underlineDashArrayProperty());
            cap = text.getUnderlineCap();
        }

        @Override
        public boolean equals(Object obj) {
            if (obj instanceof UnderlineAttributes) {
                UnderlineAttributes attr = (UnderlineAttributes)

```

```

obj;
        return super.equals(attr) && Objects.equals(cap,
attr.cap);
    } else {
        return false;
    }
}

@Override
public String toString() {
    return String.format("UnderlineAttributes[cap=%s %s]
", cap, getSubString());
}

}

private static class LineAttributesBase {

    final double width;
    final Paint color;
    final Double[] dashArray;

    public final boolean isNullValue() { return color == nul
l || width == -1; }

    /**
     * Java Quirk! Using {@code t.get[border/underline]DashA
rray()} throws a ClassCastException
     * "Double cannot be cast to Number". However, using {@c
ode t.getDashArrayProperty().get()}
     * works without issue
     */
    LineAttributesBase(Paint color, Number width, ObjectProp
erty<Number[]> dashArrayProp) {
        this.color = color;
        if (color == null || width == null || width.doubleVa
lue() <= 0) {
            // null value
            this.width = -1;
            dashArray = null;
        } else {

```

```

        // real value
        this.width = width.doubleValue();

        // get the dash array - JavaFX CSS parser seems
to return either a Number[] array
        // or a single value, depending on whether only
one or more than one value has been
        // specified in the CSS
        Object dashArrayProperty = dashArrayProp.get();
        if (dashArrayProperty != null) {
            if (dashArrayProperty.getClass().isArray())
{
                Number[] numberArray = (Number[]) dashAr
rayProperty;

                dashArray = new Double[numberArray.lengt
h];

                int idx = 0;
                for (Number d : numberArray) {
                    dashArray[idx++] = (Double) d;
                }
            } else {
                dashArray = new Double[1];
                dashArray[0] = ((Double) dashArrayProper
ty).doubleValue();
            }
        } else {
            dashArray = null;
        }
    }

    @Override
    public boolean equals(Object obj) {
        if (obj instanceof UnderlineAttributes) {
            UnderlineAttributes attr = (UnderlineAttributes)
obj;

            return Objects.equals(width, attr.width)
                && Objects.equals(color, attr.color)
                && Arrays.equals(dashArray, attr.dashArr
ay);

```

```
        } else {
            return false;
        }
    }

    protected final String getSubString() {
        return String.format("width=%s color=%s dashArray=%s", width, color, Arrays.toString(dashArray));
    }
}
```

If I get any feeling when looking at this code, the feeling is "Please don't make me modify that code, pleaseeeeeee%^&*(*&^%~!!!! Nooooooooo!!!!".

Spaghetti Code

We have spaghetti code in the project when the flow is conceptually like a bowl of spaghetti, i.e. twisted and tangled. There are other [Italian type of code](#) as well.

Example

With spaghetti code, we never know where the code starts and where the code ends.

```
class DefaultModule {
    public static Processor proc() {
        return new Processor();
    }
}

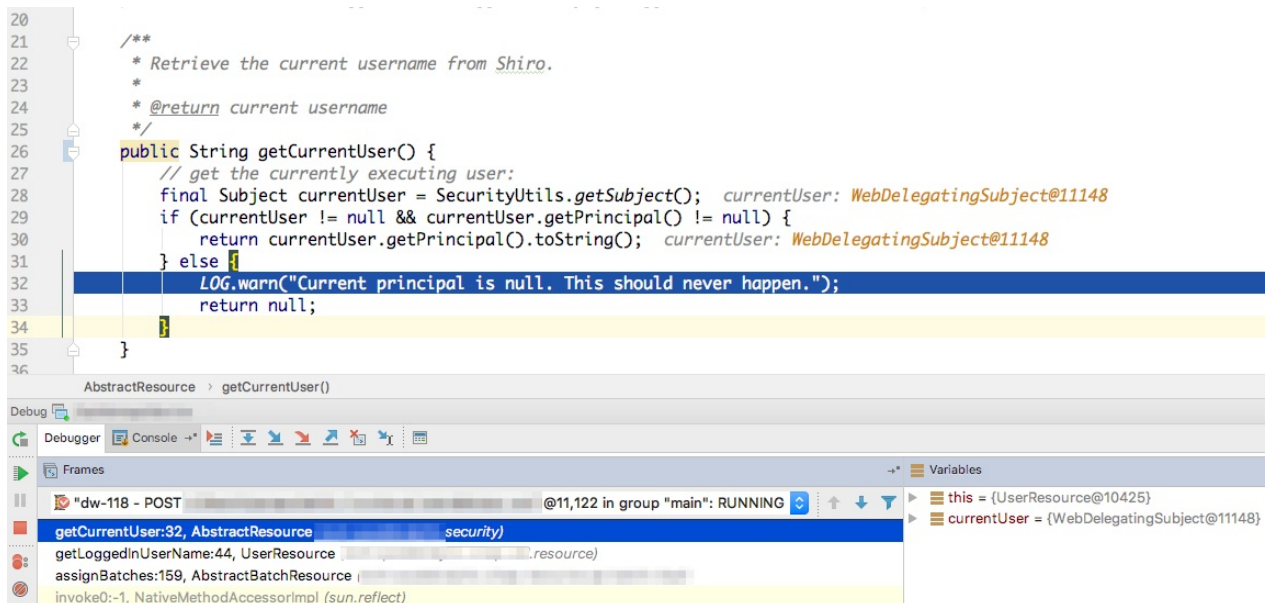
class IQModule {
    public Processor process() {
        Processor proc = DefaultModule.proc();
        return proc;
    }
}

class Processor {
    public void exec() {
        Processor process = new IQModule().process();
        process.exec();
    }
}
```


It Will Never Happen

Natural part of programmers' work is to try to predict what will or not happen. For example, if user does not provide password while authenticating, the program reports a failure. This is correct assumption and this logical flow should be implemented.

Sometime, we go too far and we try to predict what will never happen. Look at this code.



```
20
21
22  /**
23   * Retrieve the current username from Shiro.
24   *
25   * @return current username
26   */
27  public String getCurrentUser() {
28      // get the currently executing user:
29      final Subject currentUser = SecurityUtils.getSubject(); currentUser: WebDelegatingSubject@11148
30      if (currentUser != null && currentUser.getPrincipal() != null) {
31          return currentUser.getPrincipal().toString(); currentUser: WebDelegatingSubject@11148
32      } else {
33          LOG.warn("Current principal is null. This should never happen.");
34          return null;
35      }
36  }
```

The screenshot shows the method `getCurrentUser()` in the `AbstractResource` class. The code has an `if` statement that checks if the `currentUser` is not null and has a principal. If true, it returns the principal's string representation. If false, it logs a warning and returns null. The warning message is "Current principal is null. This should never happen." The debug console shows the method is being called from `dw-118 - POST` and the `currentUser` variable is of type `WebDelegatingSubject@11148`.

The original author was sure that else branch should never be executed. But when I was debugging the application, I found it is actually called. But it shouldn't be called, how come it is called now? This application is deployed in production and this code is not causing issues. Everything works perfectly. This else branch is executed because the code around was changed and the other code around stopped carrying what `getCurrentUser` method returns.

What we should do when we think this code will never happen? We could consider these options.

- don't implement that "will or should never happen" code, if it should never happen, it shouldn't happen to be implemented
- throw an exception, indicating this should never happen, so users of the code will know something is very wrong
- log fatal error, so users of the code will know something is very wrong

Error Codes

When the error codes escape from a system to clients, then they are supposed to find the solution based on the code. Forcing users to decrypt error code and find a solution is very irritating.

```
public class ExceptionErrorCodes {  
  
    public static final int FIELD_MUST_BE_EMPTY = 101;  
  
    public static final int FIELD_MUST_NOT_BE_EMPTY = 102;  
  
    public static final int CANNOT_RETRIEVE_TASK = 103;  
  
    public static final int FIELD_CANNOT_BE_MODIFIED = 104;  
  
    public static final int MYBATIS_EXCEPTION = 103;  
  
    public static final int CSV_INVALID_FIELD_EXCEPTION = 104;  
  
    public static final int CSV_NO_HEADER_FOUND_EXCEPTION = 105;  
  
    public static final int FILE_UPLOAD_EXCEPTION = 107;  
  
    public static final int CATCHALL_EXCEPTION = 999;  
}
```

Commented Code

If we fail to create meaningful code with meaningful names, we tend to create comments to explain our intentions and give other people clues.

It might seem, that better no comment than comment that is false. The comments can get old because automated refactoring can't fix all the comments.

```
Map<String, Object> result = new HashMap<>();
if (numItems < 0) {
    numItems = 0; // use -1 to get all without limit
} else if (numItems == 0) {
    numItems = 10; // default to max of 20 items in a list
}
```

Abbreviations

Abbreviations in code are causing higher cognitive load for developers. They should be avoided. There is always a better name to discover.

Look at `numItems` method parameter. `numItems` can mean couple of things - number of items, numerical items or number items. But in reality, it is limit that specifies maximum number of items that are requested. We would do better job if we just stick to `limit` as the name for that variable.

```
Response listJobs(@RequestParam("offset") final int startswith,  
@RequestParam("limit") int numItems)
```

Prefixes

Somehow, package name is not enough to identify a class. We should think harder before we decide to prefix all classes names with, for example company name or company first letter.

Company or product prefixes

Lets say we work for company named `UmpaLumpaOfScience` that is developing a product named `science`. We should probably avoid using these kind of prefixes.

If there is a conflict in names in one class, probably something else is wrong our class design and we need to rethink how our classes exchange data and how they work together.

```
public class UUser {  
}  
  
public class UScienceUser extends UUser {  
}
```

m for member **s** for static

Just don't do this.

```
public class Task {  
  
    private static final String[] sExcludedFields = new String[]{"  
mCreated", "mId"};  
  
    private Integer mId;  
  
    private String mTemplate;
```

```
public Integer getId() {
    return mId;
}

public setId(Integer id) {
    return mId = id;
}

public String getTemplate() {
    return mTemplate;
}

public getTemplate(String template) {
    mTemplate = template;
}

@Override
public String toString() {
    return ToStringBuilder.reflectionToString(this, SHORT_PREFIX
_STYLE);
}

@Override
public int hashCode() {
    return HashCodeBuilder.reflectionHashCode(this, sExcludedFie
lds);
}

@Override
public boolean equals(final Object obj) {
    return EqualsBuilder.reflectionEquals(this, obj, EXCLUDE_FIE
LDS);
}
}
```

Over Patternized

Over patternization happens when we go wild and we use every design pattern we have heard of without applying any reasoning.

Example - Hello World

Lets say we want to print `Hello World!` message. One, simple approach would be this one. Just print out the message, which should be enough for the first version of the hello world application.

```
class PrintHelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello World!");  
    }  
}
```

The other approach is to throw there couple of design patterns (BDUF - big design up front). We are going to use `Facade` pattern because amount of classes is too high to expect someone would try to understand what is happening under the hood. `Facade` patter helps us to wrap all the logic. It first creates a message using `Builder` pattern pattern. Then it acquires command factory using `Factory` and `Singleton` patterns. Then we create a new command that will be responsible for printing out the text, `Command` pattern. When we have command pattern, we just call `execute` which prints out the message.

```
class PrintHelloWorld {  
    public static void main(String[] args) {  
        Facade facade = new Facade();  
        facade.print("Hello World!");  
    }  
}  
  
class Facade {  
    public void print(String message) {
```



```
        Message m = new MessageBuilder().withMessage(message).build();
        CommandFactory commandFactory = SystemOutPrintlnCommandFactory.getInstance();
        Command command = commandFactory.createCommand(m);
        command.execute();
    }
}

interface Message {
    String getMessage();
    void setMessage(String message);
}

class DefaultMessage implements Message {
    private String message;

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}

class MessageBuilder {
    private final Message message;

    public MessageBuilder() {
        this.message = new DefaultMessage();
    }

    public MessageBuilder withMessage(String message) {
        this.message.setMessage(message);
        return this;
    }

    public Message build() {
        return message;
    }
}
```

```
    }
}

interface CommandFactory {
    Command createCommand(Message message);
}

class SystemOutPrintlnCommandFactory implements CommandFactory {

    private static SystemOutPrintlnCommandFactory instance = new
SystemOutPrintlnCommandFactory();

    public static CommandFactory getInstance() {
        return instance;
    }

    private SystemOutPrintlnCommandFactory() {}

    @Override
    public Command createCommand(Message message) {
        if (message instanceof DefaultMessage) {
            return new SystemOutPrintlnCommand(message);
        }
        throw new RuntimeException("No suitable command found");
    }
}

interface Command {
    void execute();
}

class SystemOutPrintlnCommand implements Command {

    private Message message;

    public SystemOutPrintlnCommand(Message message) {
        this.message = message;
    }

    @Override
```

```
public void execute() {  
    System.out.println(message.getMessage());  
}  
}
```