Object Oriented Design

Serhiy Oplakanets @ Lohika, 2010

Agenda

- Symptoms of Bad Design
- Basic Principles of Object Oriented Design

Symptoms of Bad Design

- Rigidity
- Fragility
- Immobility
- Viscosity

Rigidity

- Changes are difficult and painful.
- Every change requires cascade of changes in dependent modules.
- Scope of change is unpredictable.
- Your manager has a favorite scope multiplier, usually more than 2.

Fragility

- Closely related to *Rigidity*.
- You can never predict the impact of change.
- You never know what will break.
- The "christmas tree" *(Hi, Jilles!)*

Basic Principles of OOD

- **S**ingle Responsibility Principles
- **O**pen-Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

Single Responsibility Principle

- A class should have one and only one reason to change.

How to Spot SRP Violation?

- Member groups or even coalitions :)
- Boolean flags
- Hard to name classes
- Monster classes: *Manager, *Controller, *Util, Context, ...
- Long unit tests
- Hard to test-double units

Open-Closed Principle

- Modules should be open for extension but closed for modification.

Friends of OCP

- Design Patterns:
  - Visitor
  - Decorator

Liskov Substitution Principle

- *Let q(x) be a property provable about objects x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T.*
- Basically means that children classes should not break parent's interface

How to Spot LSP Violation?

- Derivative that tries to do less than base class
- *instanceof* checks
- Hiding or stubbing parent methods
  - `void someMethod() {}`
  - `void someMethod(){throw new ShouldNotBeCalledException("...");}`
- *Polymorphic if statements :)*

Dependency Inversion Principle

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

DIP Violation Example

```
class UserDao {
  User find(String login, String password) {
    Database database = new MySQLDatabase();
    // run queries and so on to create user ...
    return user;
  }
}
```

DIP Example. Refactored

```
class UserDao {
  DataSource dataSource;

  UserDao(DataSource dataSource) {
    this.dataSource = dataSource;
  }

  User find(String login, String password) {
    // use dataSource to create user object ...
    return user;
  }
}
```

DIP Example.Continued

```
interface DataSource {
  Map<Key, Value> read(String container);
  void update(String container, Map<Key, Value> fields);
  void create(String container, Map<Key, Value> fields);
}
```

Why adhere to DIP?

Friends of DIP

- Abstractions(Interfaces/Abstract Classes/...)
- Patterns
  - Factory, Abstract Factory
  - Adapter
  - Service Locator
  - Dependency Injection
- Inversion of Control Principle

Questions?

- Have I mentioned good OO Design is hard to implement, btw?