# Принципы проектирования и дизайна ПО

Лекция №12

Агошков Илья 2016

# GoF design patterns

| Creational | Structural | Behavioral |
|---|---|---|
| Factory method | Adapter | Chain of responsibility |
| Abstract factory | Bridge | Command |
| Singleton | Composite | Interpreter |
| Builder | Decorator | Iterator |
| Prototype | Facade | Mediator |
| | Flyweight | Memento |
| | Proxy | Observer |
| | | State |
| | | Strategy |
| | | Template method |
| | | Visitor |

# State

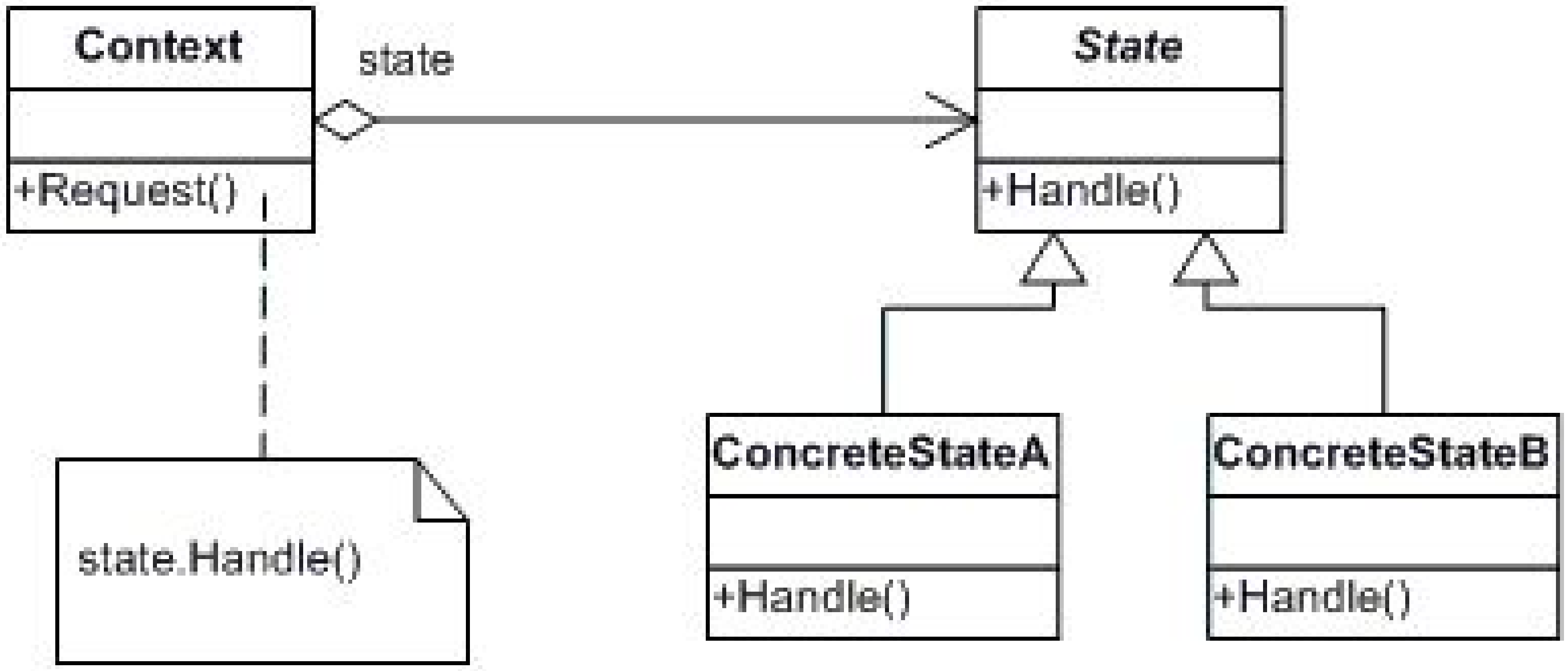
## Intent

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

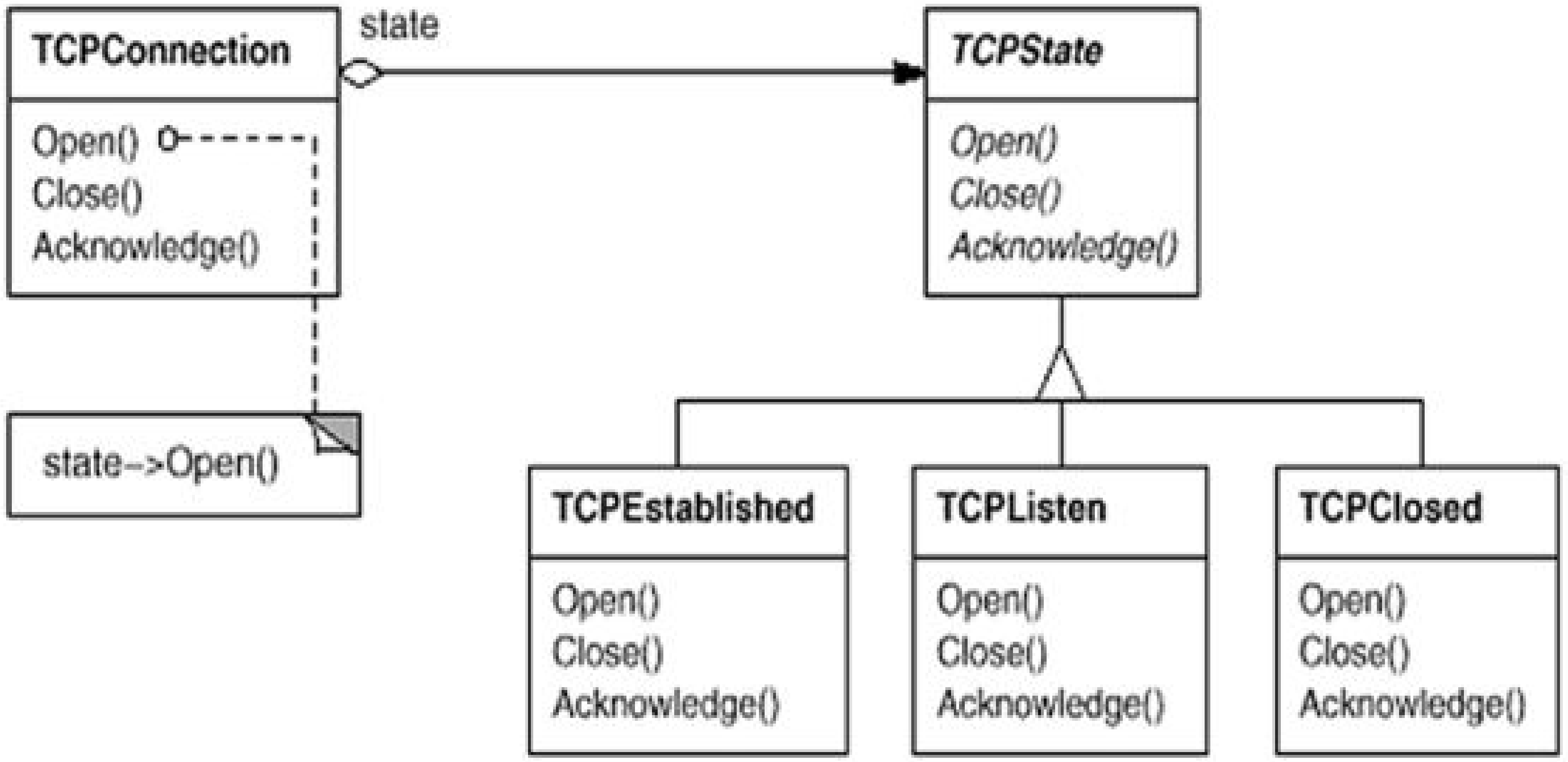
## Similar to

Strategy

# State


Context
+Request()
state
State
+Handle()
ConcreteStateA
+Handle()
ConcreteStateB
+Handle()
state.Handle()

# State


TCPConnection
Open()
Close()
Acknowledge()
state
TCPState
Open()
Close()
Acknowledge()
state->Open()
TCPEstablished
Open()
Close()
Acknowledge()
TCPListen
Open()
Close()
Acknowledge()
TCPClosed
Open()
Close()
Acknowledge()

## State

### Applicability

Use the State pattern in either of the following cases:
• An object’s behavior depends on its state, and it must change its behavior at run-time depending on that state.
• Operations have large, multipart conditional statements that depend on the object’s state. This state is usually represented by one or more enumerated constants. Often, several operations will contain this same conditional structure. The State pattern puts each branch of the conditional in a separate class. This lets you treat the object’s state as an object in its own right that can vary independently from other objects.

# Command

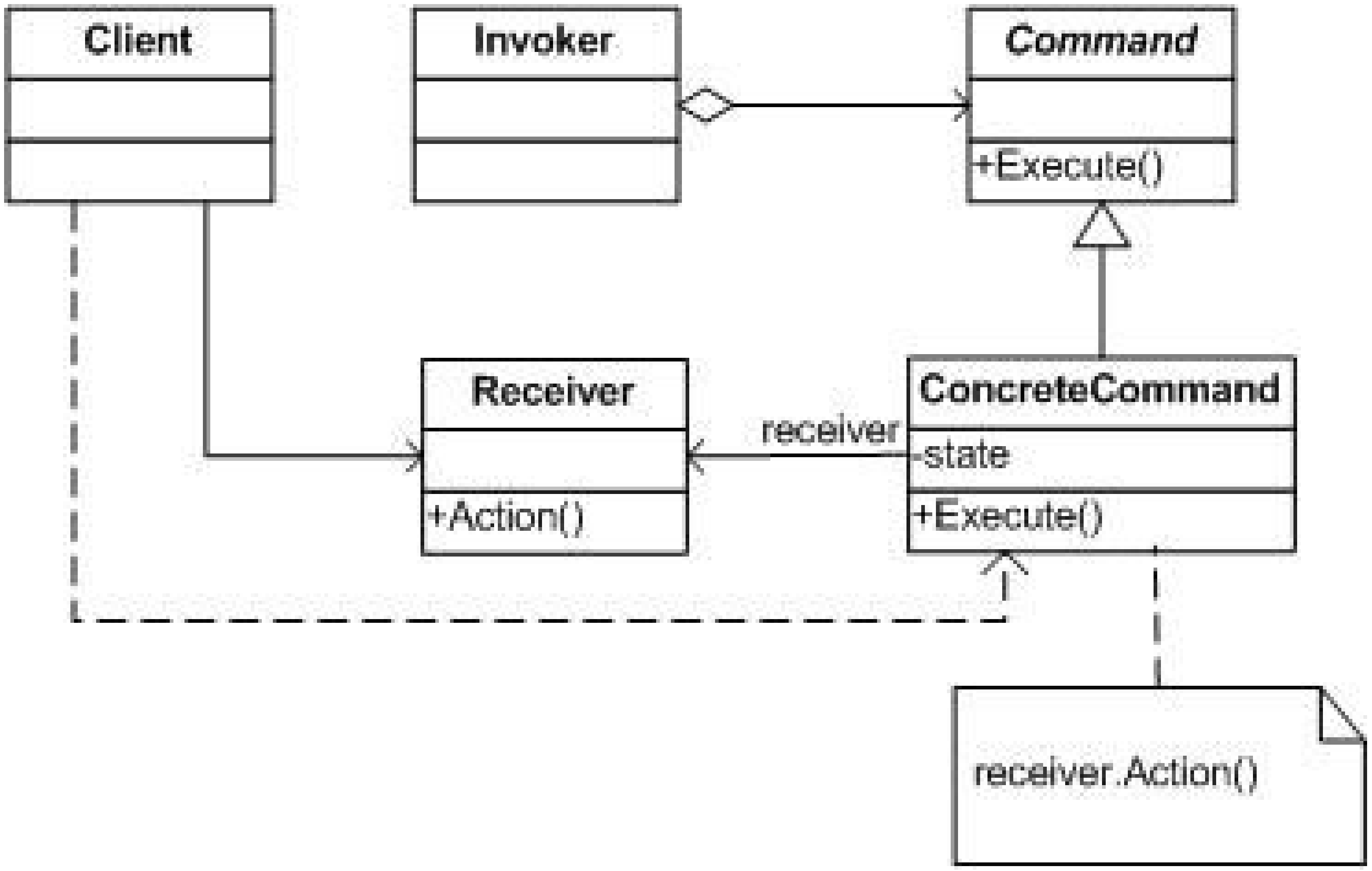
## Intent

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

## Also known as

Action, Transaction

# Command


Client
Invoker
Command
+Execute()
Receiver
+Action()
receiver
ConcreteCommand
-state
+Execute()
receiver.Action()

# Command

## Applicability

Use the Command pattern when you want to

- parameterize objects by an action to perform. You can express such parameterization in a procedural language with a callback function, that is, a function that's registered somewhere to be called at a later point. Commands are an object-oriented replacement for callbacks.
- specify, queue, and execute requests at different times. A Command object can have a lifetime independent of the original request. If the receiver of a request can be represented in an address space independent way, then you can transfer a command object for the request to a different process and fulfill the request there.
- support undo. The Command's Execute operation can store state for reversing its effects in the command itself. The Command interface must have an added Unexecute operation that reverses the effects of a previous call to Execute. Executed commands are stored in a history list. Unlimited-level undo and redo is achieved by traversing this list backwards and forwards calling Unexecute and Execute, respectively.

# Command

## Applicability

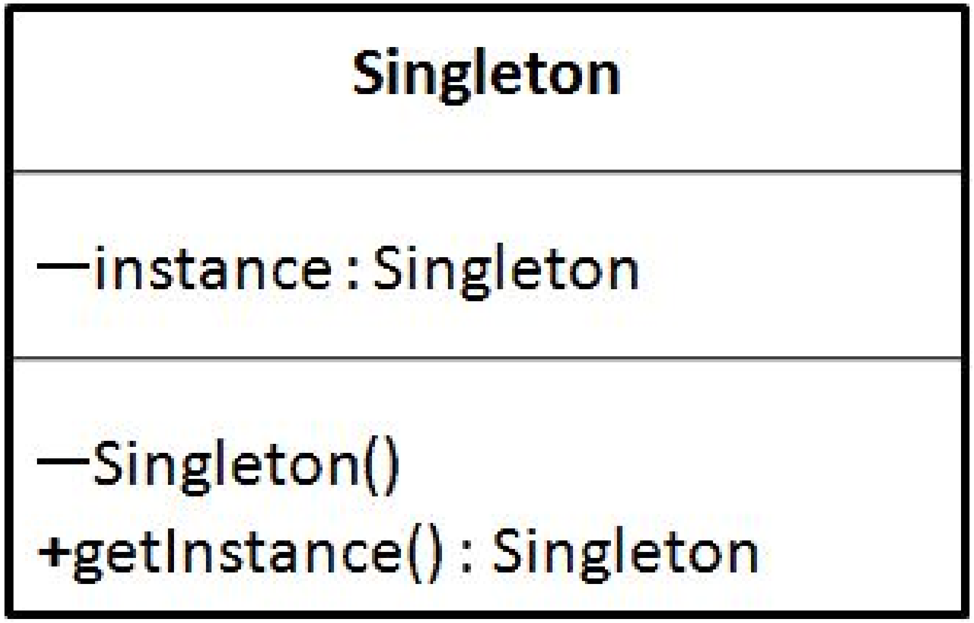
Use the Command pattern when you want to
• support logging changes so that they can be reapplied in case of a system crash. By augmenting the Command interface with load and store operations, you can keep a persistent log of changes. Recovering from a crash involves reloading logged commands from disk and reexecuting them with the Execute operation.
• structure a system around high-level operations built on primitives operations. Such a structure is common in information systems that support transactions. A transaction encapsulates a set of changes to data. The Command pattern offers a way to model transactions. Commands have a common interface, letting you invoke all transactions the same way. The pattern also makes it easy to extend the system with new transactions.

## Singleton

### Intent

Ensure a class only has one instance, and provide a global point of access to it.

# Singleton


Singleton
—instance : Singleton
—Singleton()
+getInstance() : Singleton

## Global state and singletons

## Global state

```
class X {
    …
    X() { … }
    public int doSomething() { … }
}

int a = new X().doSomething();
int b = new X().doSomething();
```

Does a == b or a != b?

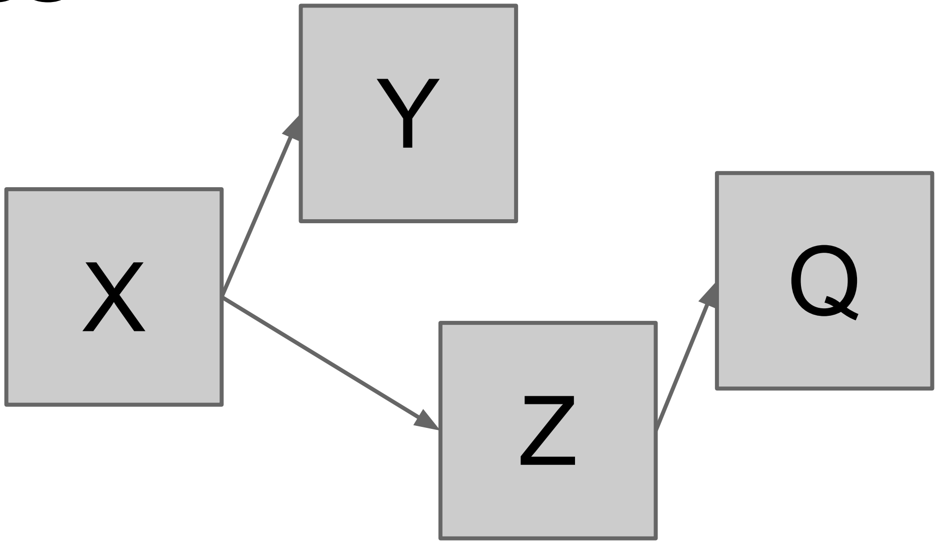
## Global state

- Object state is transient and subject to garbage collection
- Class state is persistent to lifetime of JVM

## Global state

a = new X(); -->

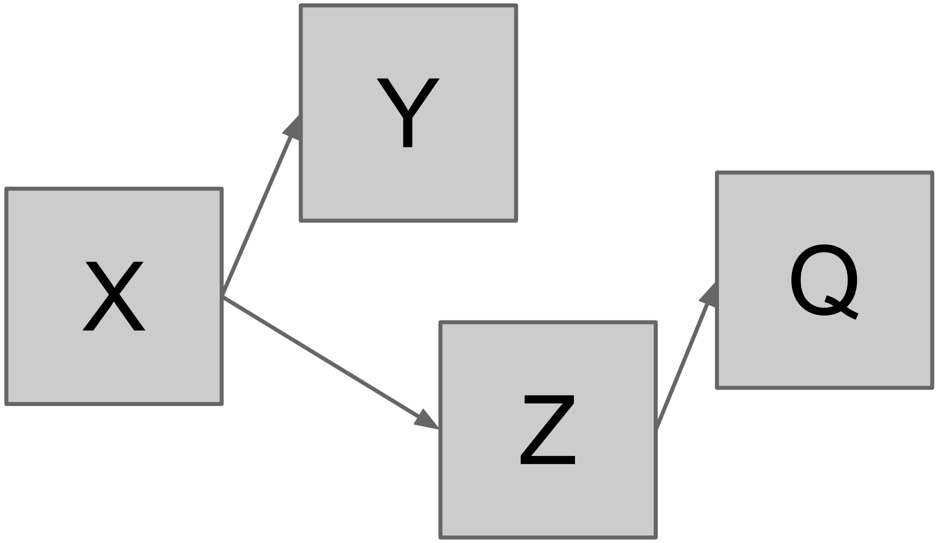
a.doSomething()



a == b

b = new X(); -->

b.doSomething()



# Global state

**a = new X(); -->**

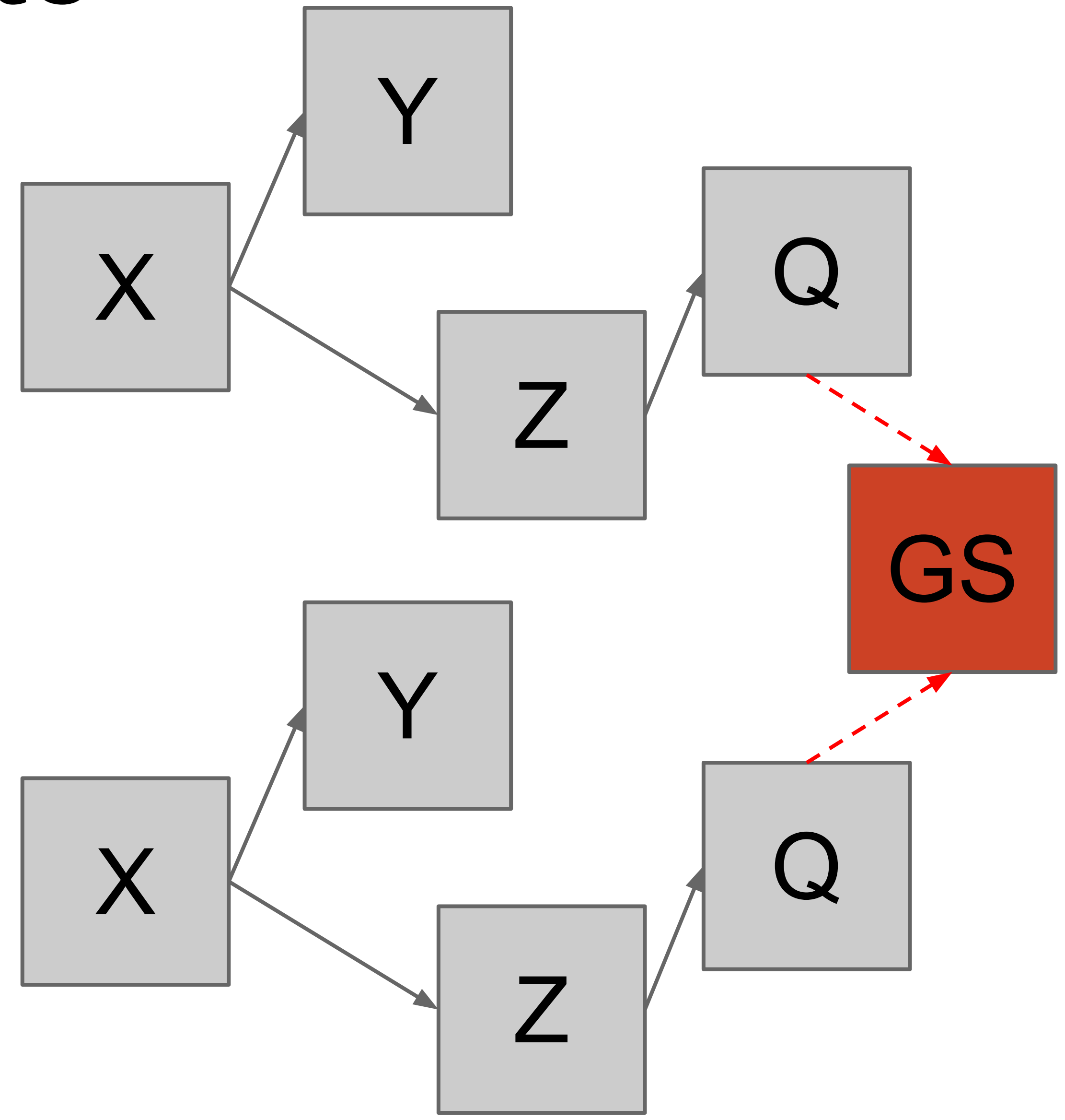
**a.doSomething()**

**a == b**

**a != b**

**b = new X(); -->**

**b.doSomething()**



# Singleton: Good vs Bad

- Application global vs JVM global
- Usually we have One application per JVM
  - So we incorrectly assume that Application Global = JVM Global
- Each test is a different configuration of a portion of our application
  - For tests it is very important that
  - Application Global != JVM Global

## Singleton: Good vs Bad

```
class AppSettings {
    private static AppSettings instance = 
new AppSettings();
    private Object state1;
    private Object state2;
    private Object state3;
    private AppSettings() {...}

    public static AppSettings getInstance() {
        return instance;
    }
}
```

# Singleton: Good vs Bad

```
class App {
    int method() {
        return App.getInstance().doX();
    }
}

void testApp() {
    // ???
}
```

# Singleton: Good vs Bad

```
class AppSettings {
    private Object state1;
    private Object state2;
    private Object state3;
    public AppSettings() {...}
}
```

- Class no longer ensures it’s “singletoness”
- Application code only creates one

## Singleton: Good vs Bad

```
class App {
    AppSettings settings;
    App(AppSetting settings) {
        this.settings = settings;
    }
    int method() {
        return settings.doX();
    }
}
void testApp() {
    new App(new AppSettings(...)).method()
}
```

## Deceptive API

- API that lies about what it needs
- Unforeseen action at a distance

# Deceptive API

```
public void testCharge() {
    CreditCard cc;
    cc = new CreditCard(“1234567890123”);
    cc.charge(100);
}
```

- After running a test you receive an SMS with text “Your credit card have been charged $100”
- Spooky action at a distance!
- It never passed in isolation

## Deceptive API

```
public void testCharge() {
    CreditCard cc;
    cc = new CreditCard(“1234567890123”);
    cc.charge(100);
}
```

```
java.lang.NullPointerException
at company.CreditCard.charge(CreditCard.java:48)
```

# Deceptive API

```
public void testCharge() {
    CreditCardProcessor.init(...);
    CreditCard cc;
    cc = new CreditCard(“1234567890123”);
    cc.charge(100);
}
```

```
java.lang.NullPointerException
at company.CreditCardProcessor.start(CreditCardProcessor.java:129)
```

## Deceptive API

```
public void testCharge() {
    OfflineQueue.start();
    CreditCardProcessor.init(...);
    CreditCard cc;
    cc = new CreditCard(“1234567890123”);
    cc.charge(100);
}
```

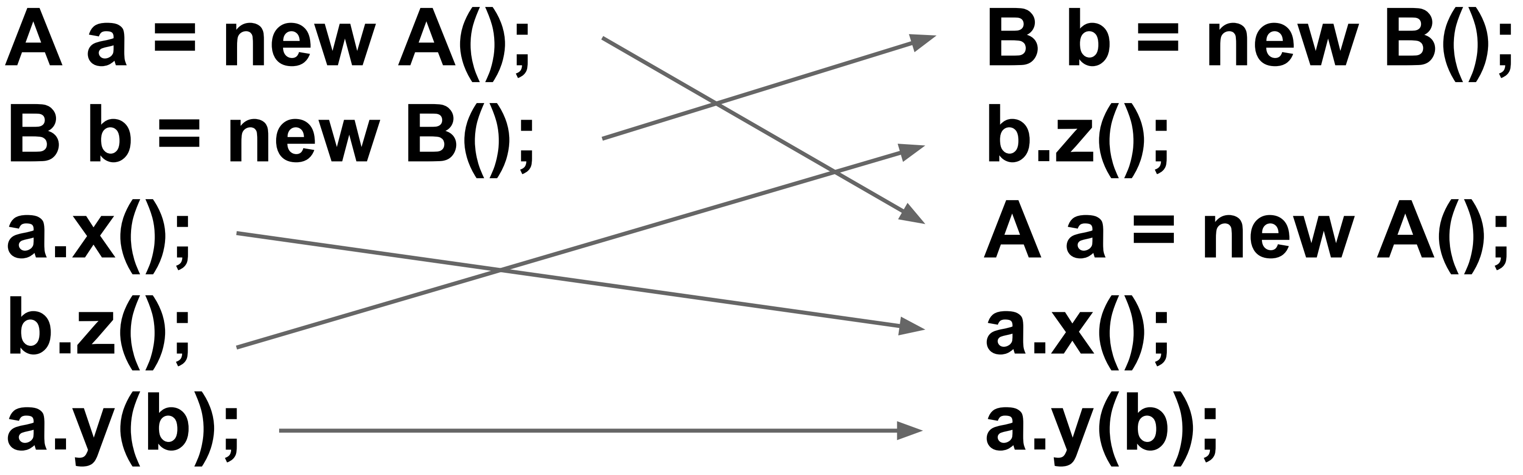
```
java.lang.NullPointerException
at company.OfflineQueue.start(OfflineQueue.java:16)
```

# Deceptive API

```
public void testCharge() {
    Database.connect(...);
    OfflineQueue.start();
    CreditCardProcessor.init(...);
    CreditCard cc;
    cc = new CreditCard(“1234567890123”);
    cc.charge(100);
}
```

CreditCard API lies!
It pretends it does not need CreditCardProcessor even though in reality it does!

# Deceptive API



```
A a = new A();
B b = new B();
a.x();
b.z();
a.y(b);
```

```
B b = new B();
b.z();
A a = new A();
a.x();
a.y(b);
```

- code should be commutative (a * b = b * a)
- The above code will fail if there is Global State
- Dependency Injection orders code naturally

## Better API

```
public void testCharge() {



    CreditCard cc;
    cc = new CreditCard(“1234567890123”);
    cc.charge(100);
}
```

## Better API

```
public void testCharge() {


    ccProc = new CreditCardProcessor(queue);
    CreditCard cc;
    cc = new CreditCard(“1234567890123”, ccProc);
    cc.charge(100);
}
```

# Better API

```
public void testCharge() {

    queue = new OfflineQueue(db);
    ccProc = new CreditCardProcessor(queue);
    CreditCard cc;
    cc = new CreditCard(“1234567890123”, ccProc);
    cc.charge(100);
}
```

## Better API

```
public void testCharge() {
    db = new Database(...);
    queue = new OfflineQueue(db);
    ccProc = new CreditCardProcessor(queue);
    CreditCard cc;
    cc = new CreditCard(“1234567890123”, ccProc);
    cc.charge(100);
}
```

Dependency Injection enforces the order of initialization at compile time

## Better API

```
public void testCharge() {
    db = new Database(...);
    queue = new OfflineQueue(db);
    ccProc = new CreditCardProcessor(queue);
    CreditCard cc;
    cc = new CreditCard(“1234567890123”, ccProc);
    cc.charge(100);
}
```

**Each layer can be tested in isolation. This is a major benefit of DI.**
**Pass in a *test double* CCProcessor, and you don’t need any of the others.**

## Review

- Global state is the root of all test problems
- Global state can not be controlled from tests
- Singleton is a common form of encapsulating global state
    - Only the Singletons that enforce their own "singletones" are a problem

## FizzBuzz

FizzBuzz is a game that has gained in popularity as a programming assignment to weed out non-programmers during job interviews. The object of the assignment is less about solving it correctly according to the below rules and more about showing the programmer understands basic, necessary tools such as `if`-/`else`-statements and loops. The rules of FizzBuzz are as follows:

For numbers 1 through 100,

- if the number is divisible by 3 print Fizz;
- if the number is divisible by 5 print Buzz;
- if the number is divisible by 3 and 5 (15) print FizzBuzz;
- else, print the number.

## FizzBuzz enterprise edition

https://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition