Java in Real Life

Eugene Dzhurinsky

March 12, 2012

Thinking in Object-Oriented way

Benefits of encapsulation

- Keep code and data together.
- Single point of modification.
- One class - one responsibility.
- Easy maintenance.
- Unit tests.

Thinking in Object-Oriented way

Benefits of encapsulation

- Keep code and data together.
- Single point of modification.
- One class - one responsibility.
- Easy maintenance.
- Unit tests.

Thinking in Object-Oriented way

Benefits of encapsulation

- Keep code and data together.
- Single point of modification.
- One class - one responsibility.
- Easy maintenance.
- Unit tests.

Thinking in Object-Oriented way

Benefits of encapsulation

- Keep code and data together.
- Single point of modification.
- One class - one responsibility.
- Easy maintenance.
- Unit tests.

Thinking in Object-Oriented way

Benefits of encapsulation

- Keep code and data together.
- Single point of modification.
- One class - one responsibility.
- Easy maintenance.
- Unit tests.

Thinking in Object-Oriented way

Wrong class hierarchy

```
class Engine {
  protected int power;
  protected int minRpm;
  protected int maxRpm;
  //getters/setters/constructor
}

class Vehicle extends Engine {
  private int capacity;
  private int volume;
  // getters/setters/constructor
}
```

Thinking in Object-Oriented way

Prefer composition over inheritance. Is-a versus has-a principle.

- Problems with having complex data hierarchy when modifying superclasses.
- Tightly coupling children class with ancestor one.
- Hierarchy design bugs makes it hard to refactor.
- Keeping unnecessary data in children from ancestor.
- Breaking incapsulation with protected field access. Ability to break contract of ancestor class in child class.

Thinking in Object-Oriented way

Prefer composition over inheritance. Is-a versus has-a principle.

- Problems with having complex data hierarchy when modifying superclasses.
- Tightly coupling children class with ancestor one.
- Hierarchy design bugs makes it hard to refactor.
- Keeping unnecessary data in children from ancestor.
- Breaking incapsulation with protected field access. Ability to break contract of ancestor class in child class.

Thinking in Object-Oriented way

Prefer composition over inheritance. Is-a versus has-a principle.

- Problems with having complex data hierarchy when modifying superclasses.
- Tightly coupling children class with ancestor one.
- Hierarchy design bugs makes it hard to refactor.
- Keeping unnecessary data in children from ancestor.
- Breaking incapsulation with protected field access. Ability to break contract of ancestor class in child class.

Thinking in Object-Oriented way

Prefer composition over inheritance. Is-a versus has-a principle.

- Problems with having complex data hierarchy when modifying superclasses.
- Tightly coupling children class with ancestor one.
- Hierarchy design bugs makes it hard to refactor.
- Keeping unnecessary data in children from ancestor.
- Breaking incapsulation with protected field access. Ability to break contract of ancestor class in child class.

Thinking in Object-Oriented way

Prefer composition over inheritance. Is-a versus has-a principle.

- Problems with having complex data hierarchy when modifying superclasses.
- Tightly coupling children class with ancestor one.
- Hierarchy design bugs makes it hard to refactor.
- Keeping unnecessary data in children from ancestor.
- Breaking incapsulation with protected field access. Ability to break contract of ancestor class in child class.

Thinking in Object-Oriented way

Better class hierarchy

```
class Engine {
  protected int power;
  protected int minRpm;
  protected int maxRpm;
  //getters/setters/constructor
}

class Vehicle {
  private Engine engine;
  private int capacity;
  private int volume;
  // getters/setters/constructor
}
```

Thinking in Object-Oriented way

Ad-hoc polymorphism, method overloading.

```
interface TaxCalculator {
  Number calculate(int interest, int grossIncome);
  Number calculate(double interest, int grossincome);
}

class CalculatorImpl implements TaxCalculator {
  // method implementations
}

class Bank {
  private TaxCalculator calculator = new CalculatorImpl();
  public Number calculate(int grossIncome) {
    int interest = getInterest(...);
    return calculator.calculate(interest, grossIncome);
  }
}
```

Thinking in Object-Oriented way

Subtype polymorphism. Liskov substitution principle

```java
interface TaxCalculator {
  Number calculate(int interest, int grossIncome);
}

class TaxCalculatorWithVAT implements TaxCalculator {
  Integer calculate(int interest, int grossIncome) {...}
}

class TaxCalculatorNoVAT implements TaxCalculator {
  Float calculate(int interest, int grossIncome) {...}
}

class Bank {
  private final TaxCalculator calculator;
  public Bank(TaxCalculator calculator) { this.calculator = calculator; };
  public Number calculate(int grossIncome) {
    int interest = getInterest(...);
    return calculator.calculate(interest, grossIncome);
  }
}
```

Thinking in Object-Oriented way

Parametric polymorphism. Generics.

```java
interface List<T> {
  void append(T item);
  void prepend(T item);
  T removeFirst();
  T removeLast();
}

class ArrayList<E> implements List<E> {
  private E[] items = new E[100];
  private int currentIdx = 0;
  public void append(E item) {...}
  public void prepend(E item) {...}
  E removeFirst() {...}
  E removeLast() {...}
}

void main() {
        List<String> stringList = new ArrayList<String>();
        stringList.append("new string here");
}
```

Thinking in Object-Oriented way

Abstract class definition. Purpose. Is-a versus Has-a.

```java
class Engine {
  //  engine properties omitted
  public void start() {};
  public void go() {};
  public void stop() {};
}

abstract class Vehicle {
  protected final Engine;
  protected Vehicle(Engine engine) { this.engine = engine; }
  public abstract void move();
}

class Truck extends Vehicle {
  private final int capacity;
  public Vehicle(Engine engine, int capacity) {
    super(engine);
    this.capacity = capacity;
  }
  public void move() {
    engine.start();
    engine.go();
    engine.stop();
  }
}
```

Thinking in Object-Oriented way

Interface definition. Contracts.

```
class Engine {  ... }

interface Movable {
  void move();
}

abstract class Vehicle {
  protected final Engine;
  protected Vehicle(Engine engine) { this.engine = engine; }
}

class Truck extends Vehicle implements Moveable {
 public Vehicle(Engine engine) {
    super(engine);
  }
  public void move() {
    engine.start();
    engine.go();
    engine.stop();
  }
}
```

Thinking in Object-Oriented way

Multiple inheritance - safe way. Diamond problem.

```
class Engine { ... }

interface Movable {
  void move();
}

interface Unloadable {
  void unload();
}

abstract class Vehicle { ... }

class Truck extends Vehicle implements Moveable, Unloadable {
  private final Unloadable trunk;
  public Vehicle(Engine engine, Unloadable trunk) {
    super(engine);
    this.trunk = trunk;
  }
  public void move() { ... }
  public void unload() {
    trunk.unload();
  }
}
```

Thinking in Object-Oriented way

Design Patterns

- Program to interfaces - not implementations.
- Prefer composition over inheritance.
- Open-close principle.

Thinking in Object-Oriented way

Design Patterns

- Program to interfaces - not implementations.
- Prefer composition over inheritance.
- Open-close principle.

Thinking in Object-Oriented way

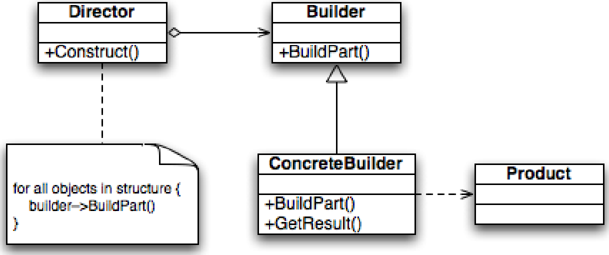
Design Patterns

- Program to interfaces - not implementations.
- Prefer composition over inheritance.
- Open-close principle.

Thinking in Object-Oriented way

Types of design patterns :: Creational patterns.

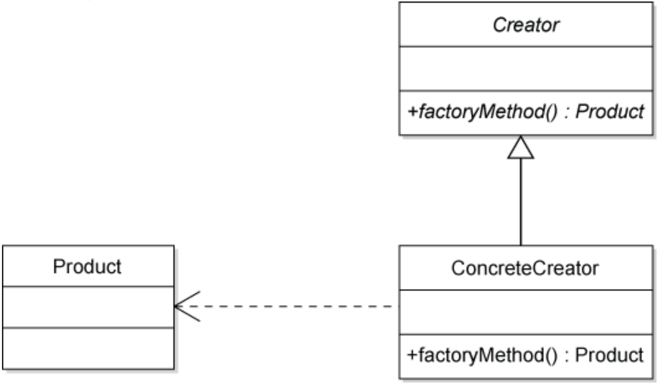
Builder



Thinking in Object-Oriented way

Types of design patterns :: Creational patterns.

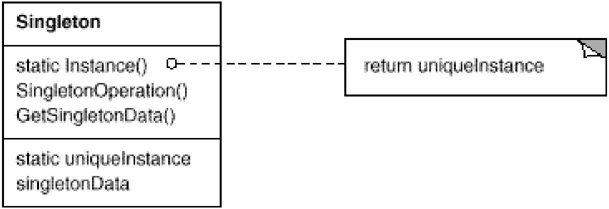
Factory method



Thinking in Object-Oriented way

Types of design patterns :: Creational patterns.

Singleton



Singleton

static Instance()
SingletonOperation()
GetSingletonData()

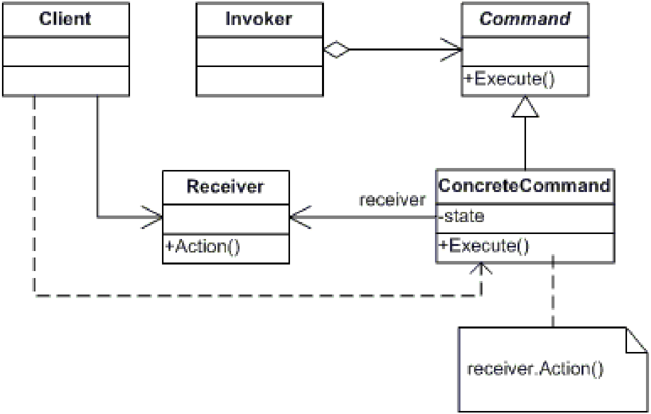
static uniqueInstance
singletonData

return uniqueInstance

Thinking in Object-Oriented way

Types of design patterns :: Behavioral patterns.

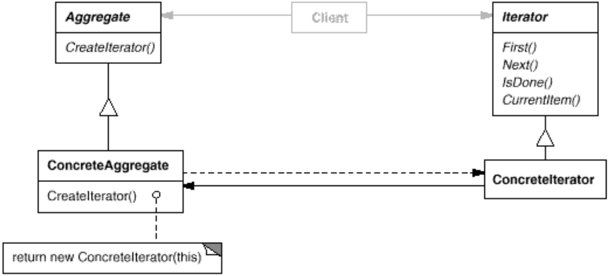
Command



Thinking in Object-Oriented way

Types of design patterns :: Behavioral patterns.

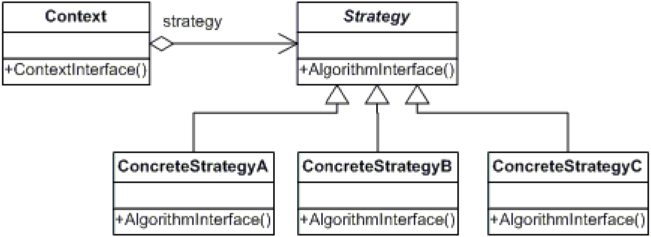
Iterator



Thinking in Object-Oriented way

Types of design patterns :: Behavioral patterns.

Strategy



Java and WEB applications.

HTTP Protocol.

HTTP protocol overview

- Request / response model
- Stateless
- Request headers
- Response headers
- Cookies
- Web sockets

Java and WEB applications.

HTTP Protocol.

HTTP protocol overview

- Request / response model
- Stateless
- Request headers
- Response headers
- Cookies
- Web sockets

Java and WEB applications.

HTTP Protocol.

HTTP protocol overview

- Request / response model
- Stateless
- Request headers
- Response headers
- Cookies
- Web sockets

Java and WEB applications.

HTTP Protocol.

HTTP protocol overview

- Request / response model
- Stateless
- Request headers
- Response headers
- Cookies
- Web sockets

Java and WEB applications.

HTTP Protocol.

HTTP protocol overview

- Request / response model
- Stateless
- Request headers
- Response headers
- Cookies
- Web sockets

Java and WEB applications.

HTTP Protocol.

HTTP protocol overview

- Request / response model
- Stateless
- Request headers
- Response headers
- Cookies
- Web sockets

Java and WEB applications.

HTML.

HTML overview

- HTML markup
- Hyperlinks
- Forms
- CSS
- JavaScript

Java and WEB applications.

HTML.

HTML overview

- HTML markup
- Hyperlinks
- Forms
- CSS
- JavaScript

Java and WEB applications.

HTML.

HTML overview

- HTML markup
- Hyperlinks
- Forms
- CSS
- JavaScript

Java and WEB applications.
HTML.

HTML overview

- HTML markup
- Hyperlinks
- Forms
- CSS
- JavaScript

Java and WEB applications.

HTML.

HTML overview

- HTML markup
- Hyperlinks
- Forms
- CSS
- JavaScript

Data Definition Language

- Tables.
- Indexes. Unique indexes. Primary keys.
- Views.

Java and WEB applications.

Databases.

Data Definition Language

- Tables.
- Indexes. Unique indexes. Primary keys.
- Views.

Java and WEB applications.

Databases.

Data Definition Language

- Tables.
- Indexes. Unique indexes. Primary keys.
- Views.

Java and WEB applications.

JDBC.

Database connectivity

- JDBC overview.
- Database drivers.
- Connections.
- Statements and prepared statements.
- Result sets.

Java and WEB applications.

JDBC.

Database connectivity

- JDBC overview.
- Database drivers.
- Connections.
- Statements and prepared statements.
- Result sets.

Java and WEB applications.

JDBC.

Database connectivity

- JDBC overview.
- Database drivers.
- Connections.
- Statements and prepared statements.
- Result sets.

Java and WEB applications.

JDBC.

Database connectivity

- JDBC overview.
- Database drivers.
- Connections.
- Statements and prepared statements.
- Result sets.

Java and WEB applications.

JDBC.

Database connectivity

- JDBC overview.
- Database drivers.
- Connections.
- Statements and prepared statements.
- Result sets.

Java and WEB applications.

Servlets and Java Server Pages.

Java Servlets

- javax.servlet.http.HttpServlet
- javax.servlet.http.HttpServletRequest and javax.servlet.http.HttpServletResponse
- Sessions. Customer identification.
- web.xml definition.

Java and WEB applications.

Servlets and Java Server Pages.

Java Servlets

- javax.servlet.http.HttpServlet
- javax.servlet.http.HttpServletRequest and javax.servlet.http.HttpServletResponse
- Sessions. Customer identification.
- web.xml definition.

Java and WEB applications.

Servlets and Java Server Pages.

Java Servlets

- javax.servlet.http.HttpServlet
- javax.servlet.http.HttpServletRequest and javax.servlet.http.HttpServletResponse
- Sessions. Customer identification.
- web.xml definition.

Java and WEB applications.

Servlets and Java Server Pages.

Java Servlets

- javax.servlet.http.HttpServlet
- javax.servlet.http.HttpServletRequest and javax.servlet.http.HttpServletResponse
- Sessions. Customer identification.
- web.xml definition.

Java and WEB applications.

Servlets and Java Server Pages.

Java Server Pages

- JSP Model 1 and JSP Model 2. Scriplets, JSTL and Expression Langiage.
- Model/View/Controller.
- Entry point.
- Request forwarding and including.

Java and WEB applications.

Servlets and Java Server Pages.

Java Server Pages

- JSP Model 1 and JSP Model 2. Scriplets, JSTL and Expression Langiage.
- Model/View/Controller.
- Entry point.
- Request forwarding and including.

Java and WEB applications.

Servlets and Java Server Pages.

Java Server Pages

- JSP Model 1 and JSP Model 2. Scriplets, JSTL and Expression Langiage.
- Model/View/Controller.
- Entry point.
- Request forwarding and including.

Java and WEB applications.

Servlets and Java Server Pages.

Java Server Pages

- JSP Model 1 and JSP Model 2. Scriplets, JSTL and Expression Langiage.
- Model/View/Controller.
- Entry point.
- Request forwarding and including.

Java and WEB applications.

Servlet Containers.

Servlet containers

- Apache Tomcat
- Jetty

Java and WEB applications.

Servlet Containers.

Servlet containers

- Apache Tomcat
- Jetty

Software engineering

Real life development

- Metodologies (Waterfall, Agile, RUP, XP)
- Outsourcing. Bodyshops.
- Freelancing. Scriptlance, Elance, Odesk, Rentacoder.
- Shareware. Software directories. Digital river.

Software engineering

Real life development

- Metodologies (Waterfall, Agile, RUP, XP)
- Outsourcing. Bodyshops.
- Freelancing. Scriptlance, Elance, Odesk, Rentacoder.
- Shareware. Software directories. Digital river.

Software engineering

Real life development

- Metodologies (Waterfall, Agile, RUP, XP)
- Outsourcing. Bodyshops.
- Freelancing. Scriptlance, Elance, Odesk, Rentacoder.
- Shareware. Software directories. Digital river.

Software engineering

Real life development

- Metodologies (Waterfall, Agile, RUP, XP)
- Outsourcing. Bodyshops.
- Freelancing. Scriptlance, Elance, Odesk, Rentacoder.
- Shareware. Software directories. Digital river.