Java in Real Life

Eugene Dzhurinsky

March 14, 2012

Thinking in Object-Oriented way

Benefits of encapsulation

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

Thinking in Object-Oriented way

Benefits of encapsulation

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

Thinking in Object-Oriented way

Benefits of encapsulation

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

Thinking in Object-Oriented way

Benefits of encapsulation

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

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

```
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.

```
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.

```
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
Unit testing. JUnit 4.

Interface definition

```
interface TaxCalculator {
        Number calculateTax(Number income);
}
```

Thinking in Object-Oriented way
Unit testing. JUnit 4.

Unit test stub

```
public class TaxCalculatorTest {

  private TaxCalculator calculator;

  @Test(expected = IllegalArgumentException.class)
  public void testNullArgument() {
    calculator.calculateTax(null);
  }

  @Test(expected = IllegalArgumentException.class)
  public void testNegativeAmount() {
    calculator.calculateTax(Double.valueOf(-100.0d));
  }


  @Test
  public void testNormalExecution() {
    assertEquals(Double.valueOf(20.0),calculator.calculateTax(100.0d));
    assertEquals(Double.valueOf(0.0),calculator.calculateTax(0.0d));
    // more assertions on corner cases
  }

}
```

Thinking in Object-Oriented way
Unit testing. JUnit 4.

Tax calculator implementation

```
public class TaxCalculatorImpl implements TaxCalculator {

  private final double taxRate;

  public TaxCalculatorImpl(double taxRate) { this.taxRate = taxRate; }

  @Override
  Double calculateTax(Number amount) {
    if (amount == null) {
      throw new IllegalArgumentException("Null is not allowed as amount");
    }
    final amountValue = amount.doubleValue();
    if (amountValue() < 0) {
      throw new IllegalArgumentException("Negative amount is not allowed");
    }
    return taxRate * amountValue;
  }
}
```

Unit test complete

```
public class TaxCalculatorTest {

  private TaxCalculator calculator;

  @Before
  public void setUp() throws Exception {
    calculator = new TaxCalculatorImpl(20.0d);
  }

  // test methods
}
```

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

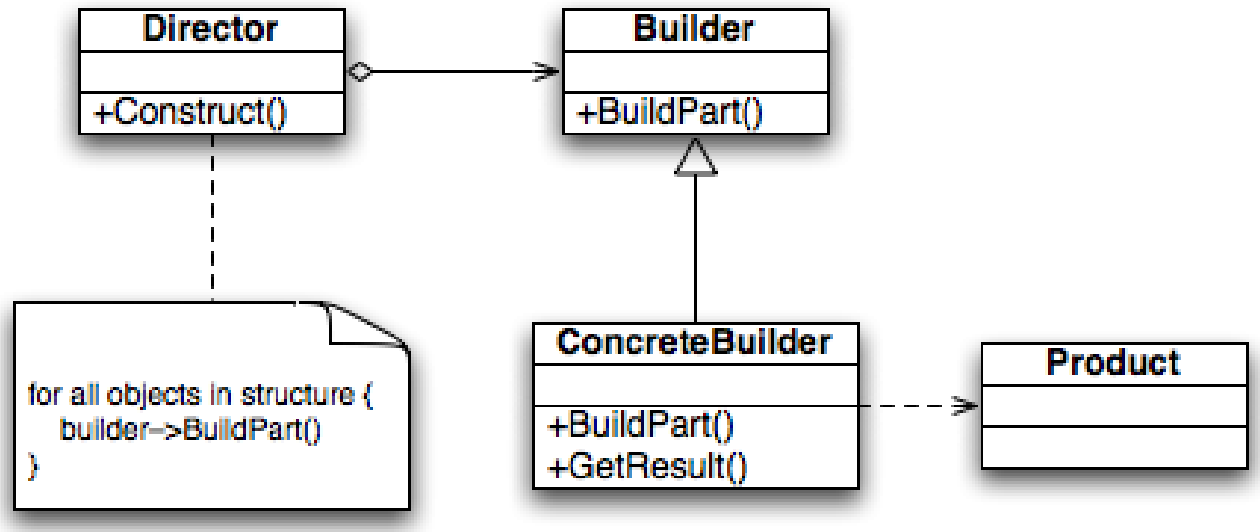
Design Patterns

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

Thinking in Object-Oriented way

Types of design patterns :: Creational patterns.

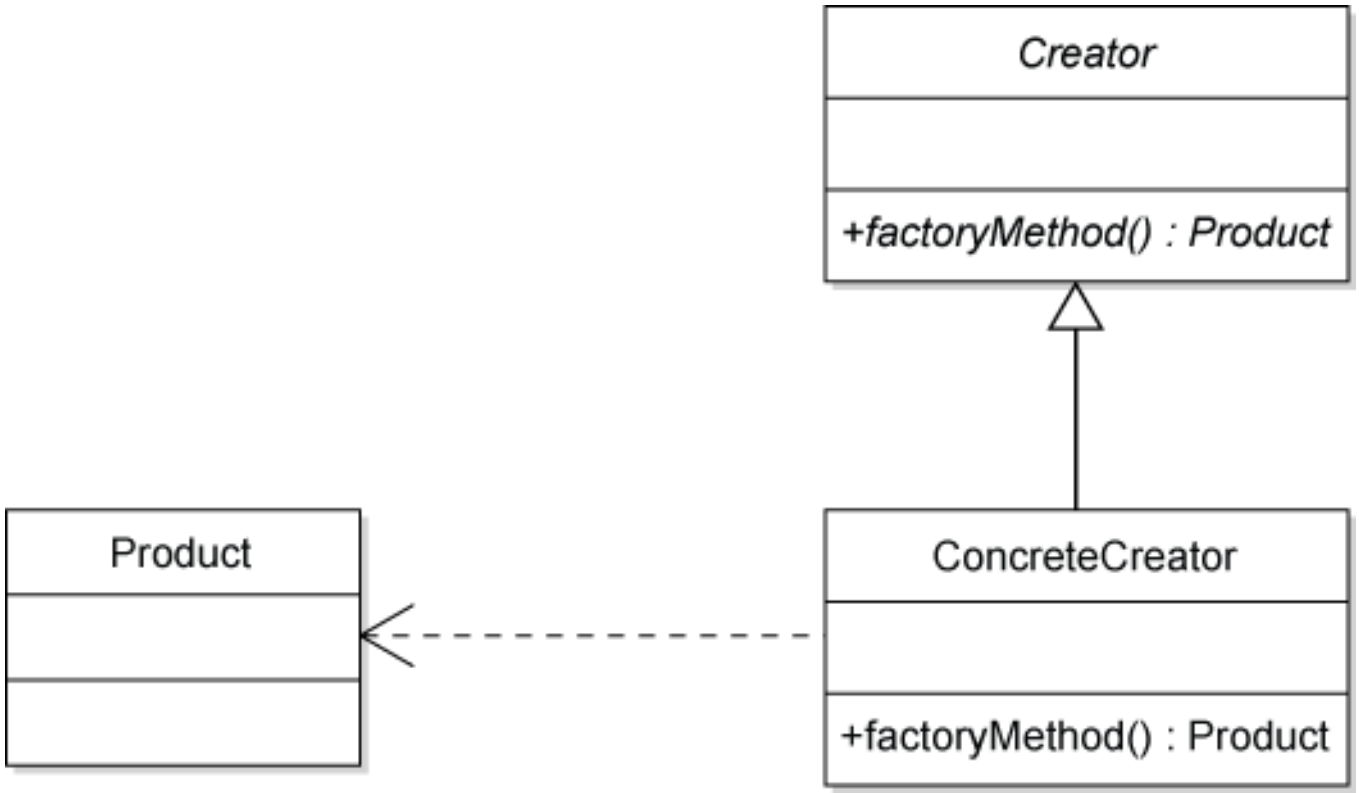
Builder



Thinking in Object-Oriented way

Types of design patterns :: Creational patterns.

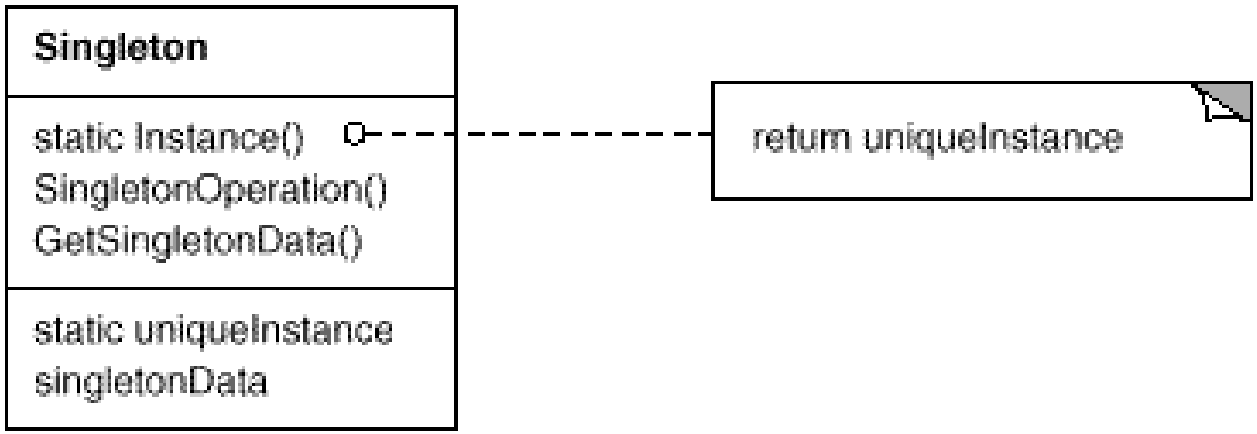
Factory method



Thinking in Object-Oriented way

Types of design patterns :: Creational patterns.

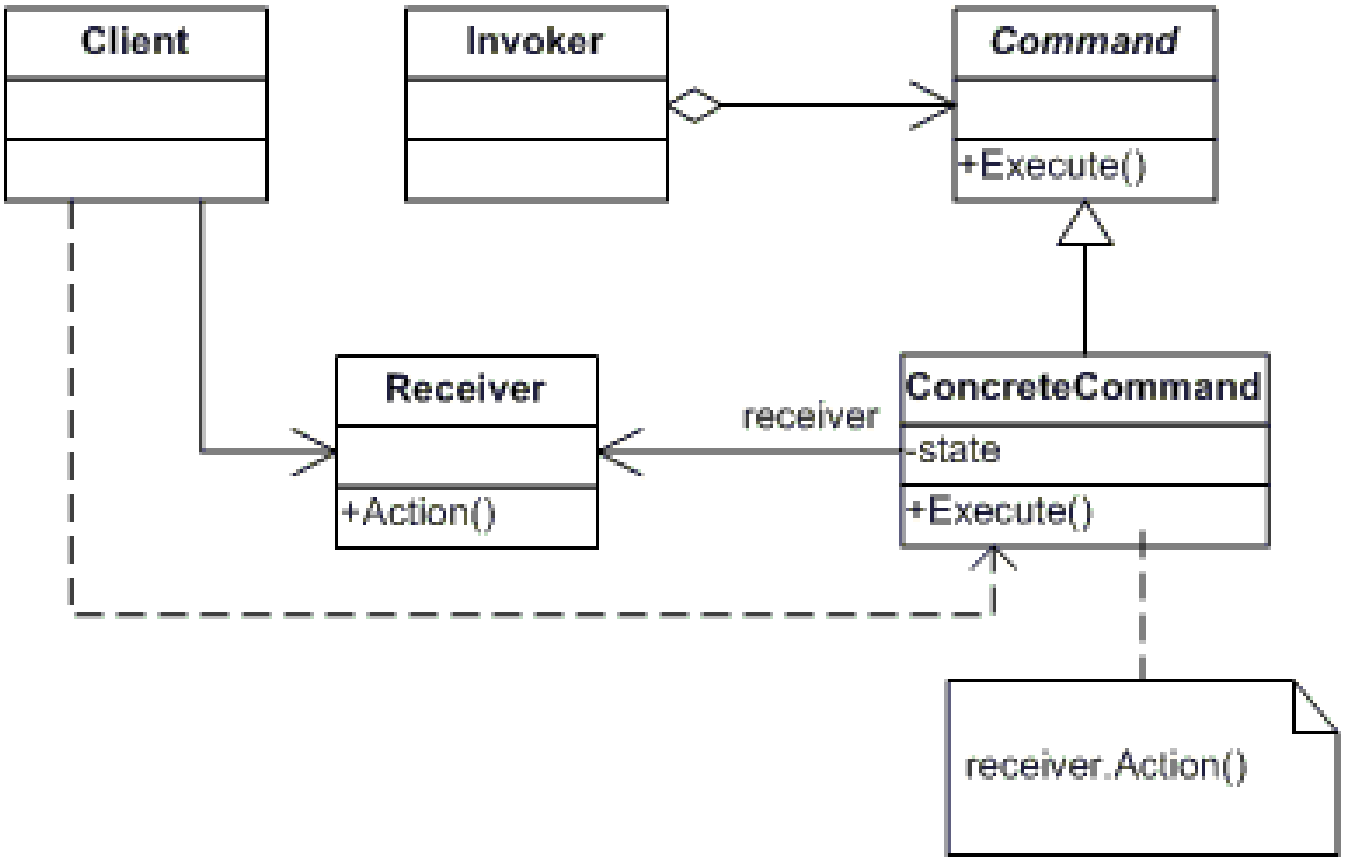
Singleton



Thinking in Object-Oriented way

Types of design patterns :: Behavioral patterns.

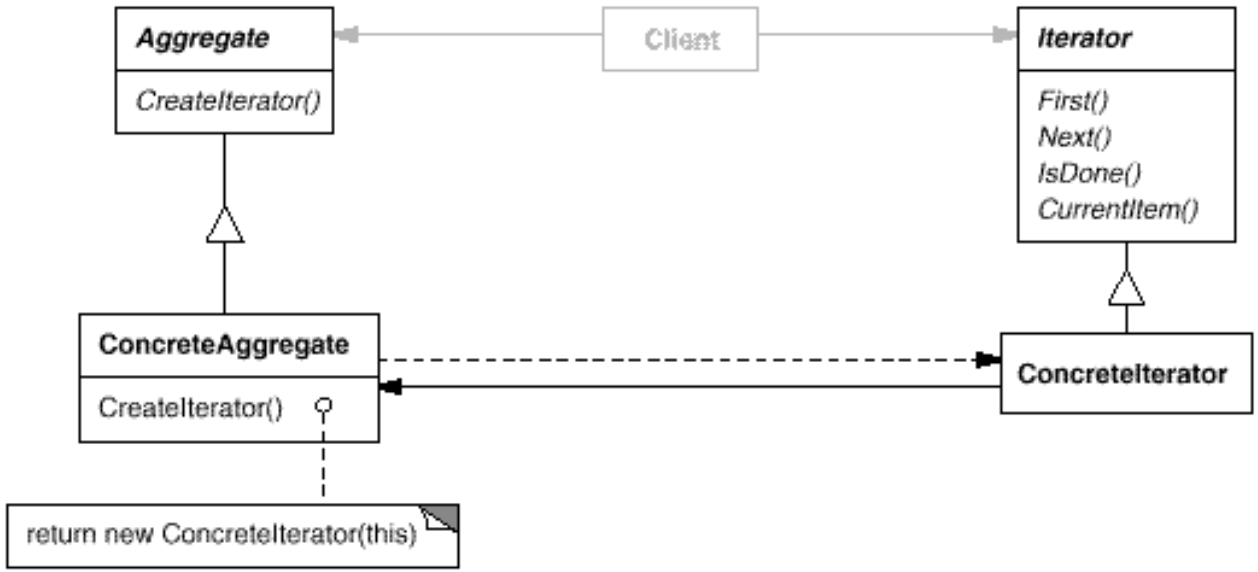
Command



Thinking in Object-Oriented way

Types of design patterns :: Behavioral patterns.

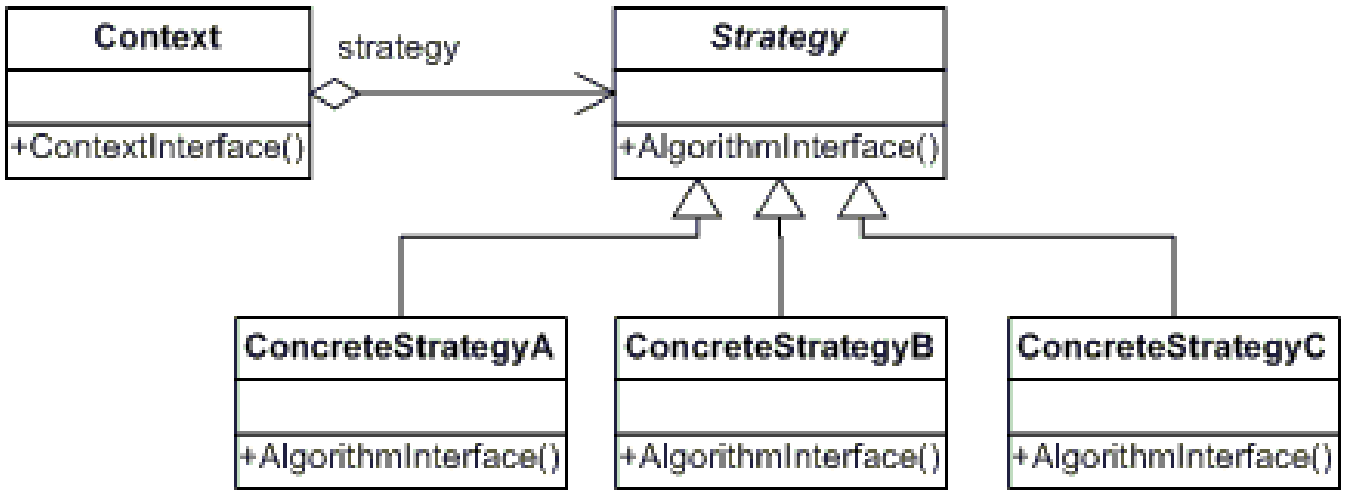
Iterator



Thinking in Object-Oriented way

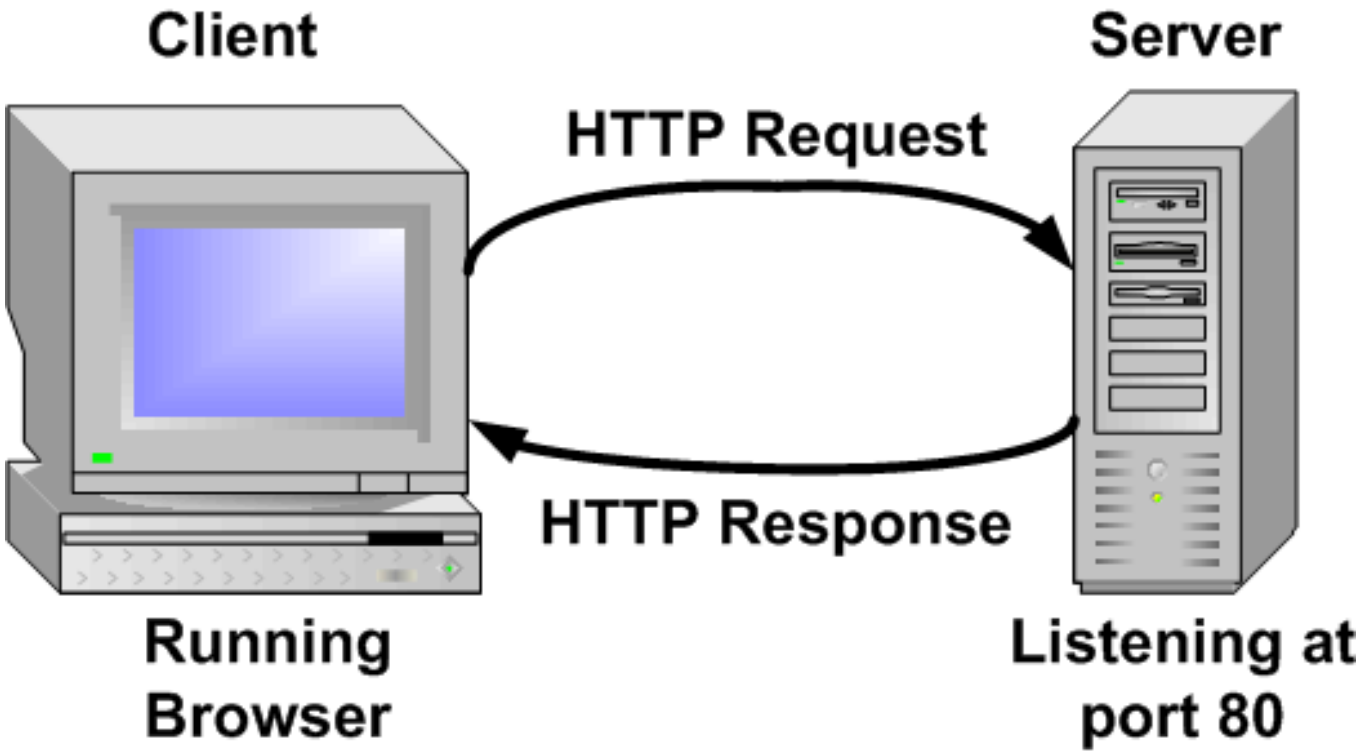
Types of design patterns :: Behavioral patterns.

Strategy



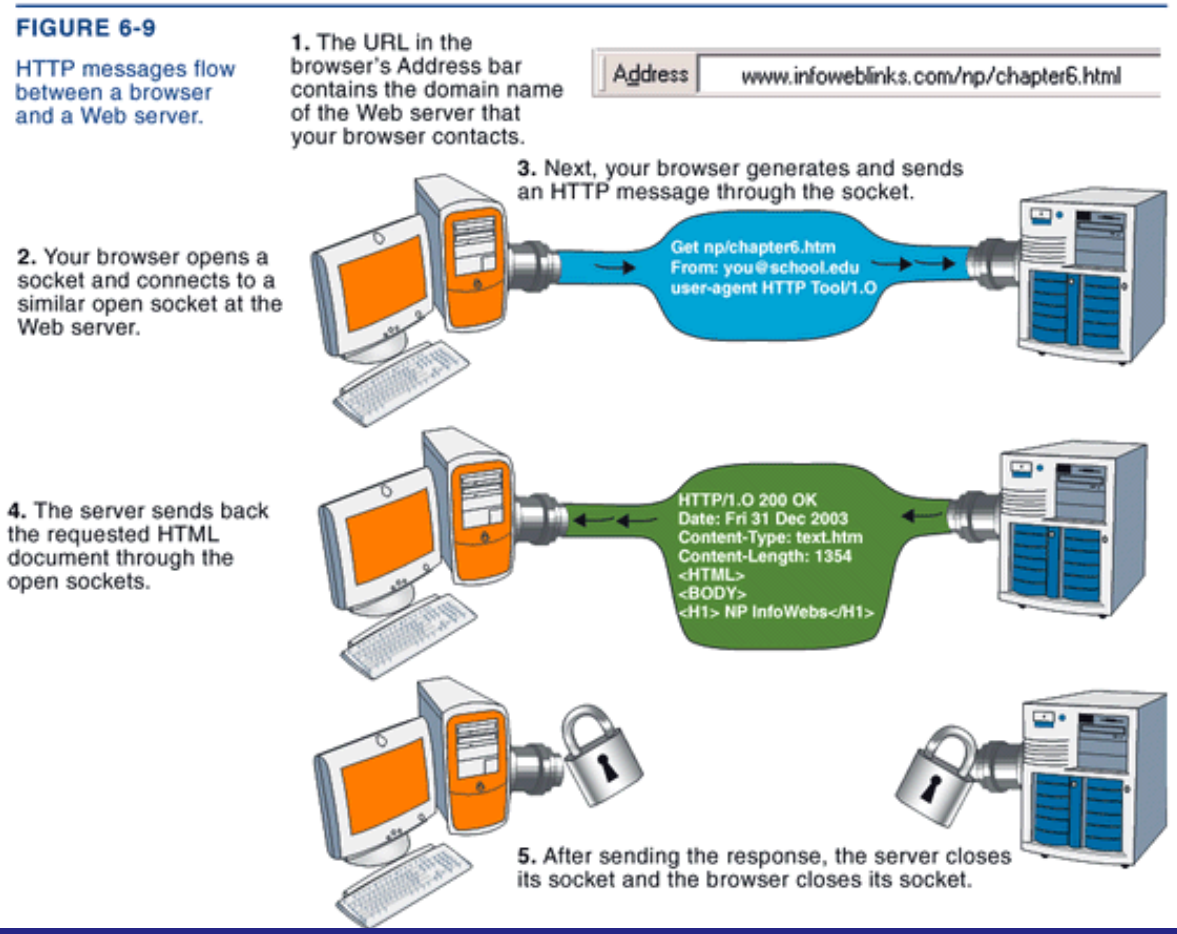
Java and WEB applications.
HTTP Protocol.

Request / response model



Java and WEB applications.
HTTP Protocol.

Stateless



Java and WEB applications.
HTTP Protocol.

Request headers

```
GET /links/widgets/zoneit.js HTTP/1.1
Host: widgets.dzone.com
User-Agent: Mozilla/5.0 (X11; FreeBSD amd64; rv:10.0.2) Gecko/20100101 Firefox/10.0.2
Accept: */*
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Referer: http://java.dzone.com/
Cookie: __qca=1194188492-91087860-55883650
Pragma: no-cache
Cache-Control: no-cache
```

Java and WEB applications.
HTTP Protocol.

Response headers

```
HTTP/1.1 200 OK
Date: Tue, 13 Mar 2012 13:36:28 GMT
Server: Apache/2.2.11 (Unix) DAV/2 SVN/1.5.5 Resin/4.0.4 PHP/5.2.13
X-Powered-By: PHP/5.2.13
Last-Modified: Tue, 13 Mar 2012 13:34:54 GMT
ETag: "416b0bcb5a020fbeaeae5d0c6cb68d87"
Expires: Sun, 19 Nov 1978 05:00:00 GMT
Cache-Control: must-revalidate
Content-Encoding: gzip
Vary: User-Agent
Keep-Alive: timeout=15, max=500
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html;
charset=utf-8
```

Java and WEB applications.
HTML.

HTML overview

Java and WEB applications.
HTML.

HTML example

```
<html>
<head>
    <title>Page Title</title>
    <link href="style.css" type="text/css" rel="stylesheet" media="screen" />
</head>
<body>
    <div id="header">
    </div>
    <div id="navigation">
        <a href="index.html">Home</a> | <a href="about.html">About</a> |
                <a href="contact.html">Contact</a>
    </div>
    <div id="left-sidebar">
    </div>
    <div id="content-area">
    </div>
    <div id="right-sidebar">
    </div>
    <div id="footer">
    </div>
</body>
</html>
```

Java and WEB applications.
Databases. Data Definition Language (DDL)

Tables

```
CREATE TABLE customer (
   customer_id int primary key auto_increment,
   firstname varchar(255) not null,
   lastname varchar(255) not null,
   email varchar(100) not null,
   gender char(1) null
)
```

Java and WEB applications.
Databases. Data Definition Language (DDL)

Indexes. Unique indexes. Primary keys.

- Indexes. Hash and B-Tree indexes. Purpose.

```
create index customer_gender on customer(gender);
```

- Unique indexes

```
create unique index customer_firstname_lastname
        on customer(firstname,lastname);
```

Java and WEB applications.
JDBC.

JDBC Overview

Java and WEB applications.
JDBC.

Example connecting to database and fetching results

```
interface CustomerEnumerator {
  List<Customer> enumerate(String firstName) throws EnumerateCustomerException;
}
class JDBCCustomerEnumerator implements CustomerEnumerator {
  public List<Customer> enumerate(String firstName) {
    Class.forName("com.mysql.jdbc.Driver").newInstance();
    Connection dbh = null; PreparedStatement psth = null; ResultSet res = null;
    try {
      DriverManager.
        getConnection("jdbc:mysql://localhost:3306/customers","dbUser","c00lpwd");
      PreparedStatement psth = dbh.prepareStatement("select firstname,lastname,gender
        from customers where firstname=?");
      psth.setString(1,firstName);
      ResultSet res = psth.executeQuery();
      List<Customer> customers = new ArrayList<Customer>(100);
      while (res.hasNext()) {
        customers.add(new Customer(res.getString(1),res.getString(2),res.getString(3));
      }
      return customers;
    } catch (SQLException e) { throw new EnumerateCustomerException(e);}
      finally { DbUtils.closeQuietly(dbh, psth, res);}
  }
}
```

Java and WEB applications.
Servlets.

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.

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.

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.

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.
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.
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.
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.
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.

Suggested readings

- **Steve McConnell.** Code Complete: A Practical Handbook of Software Construction. **ISBN-10: 0735619670.**
- **Joshua Bloch.** Effective Java. **ISBN-10: 0321356683.**
- **Bruce Eckel.** Thinking in Java. **ISBN-10: 0131872486.**
- **Kathy Sierra.** Head First Java. **ISBN-10: 0596009208.**
- **Elisabeth Freeman.** Head First Design Patterns. **ISBN-10: 0596007124.**
- **Martin Fowler.** Refactoring: Improving the Design of Existing Code. **ISBN-10: 0201485672.**
- **Craig Larman.** Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. **ISBN-10: 0131489062.**