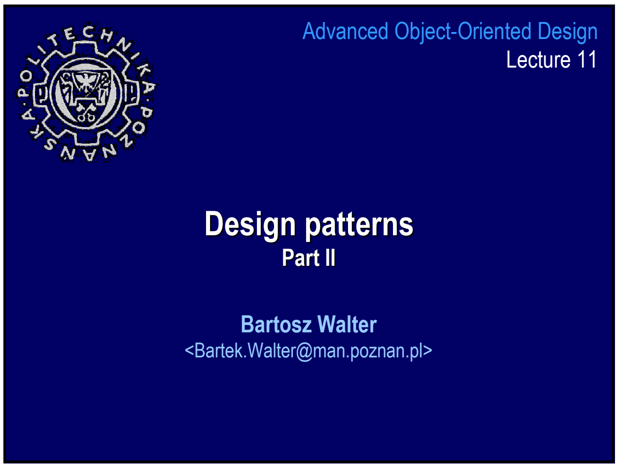
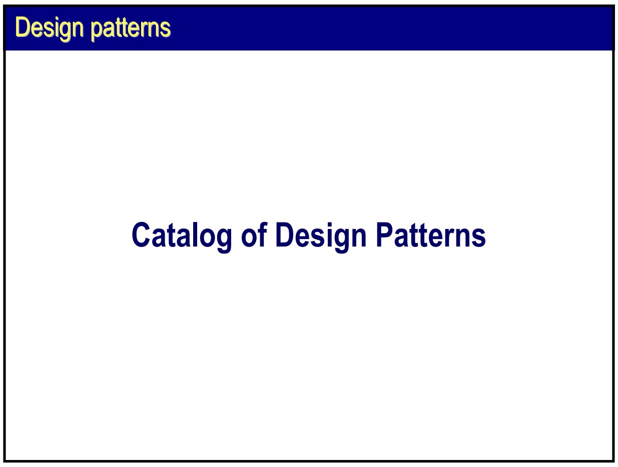


Advanced Object-Oriented Design
Lecture 11

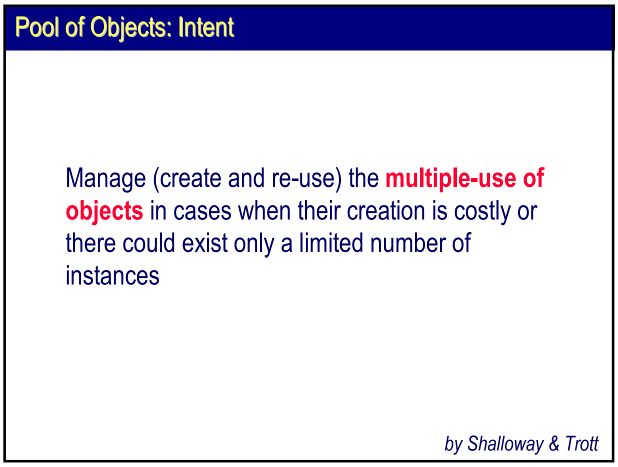
Design patterns
Part II

Bartosz Walter
<Bartek.Walter@man.poznan.pl>



Design patterns

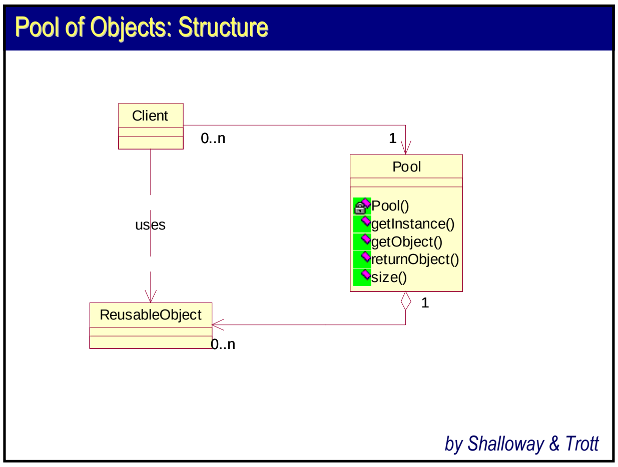
Catalog of Design Patterns



Pool of Objects: Intent

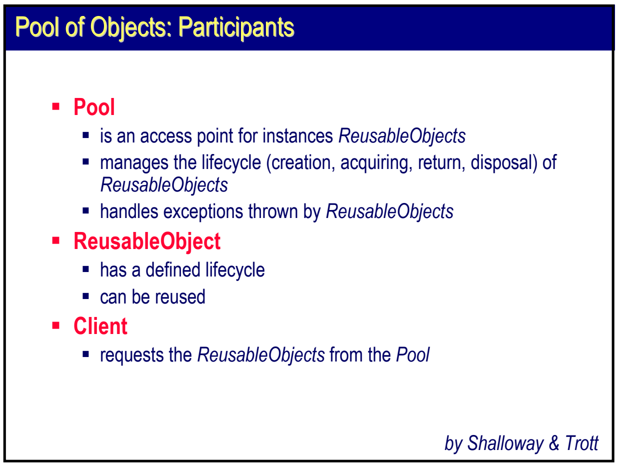
Manage (create and re-use) the **multiple-use of objects** in cases when their creation is costly or there could exist only a limited number of instances

by Shalloway & Trott



Pool of Objects: Structure

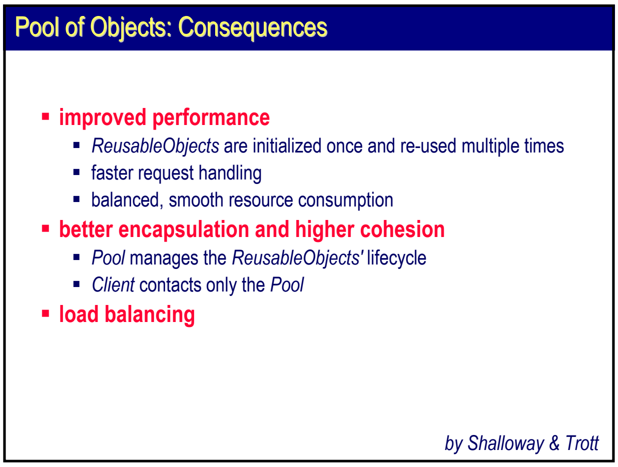
by Shalloway & Trott



Pool of Objects: Participants

- **Pool**
  - is an access point for instances *ReusableObjects*
  - manages the lifecycle (creation, acquiring, return, disposal) of *ReusableObjects*
  - handles exceptions thrown by *ReusableObjects*
- **ReusableObject**
  - has a defined lifecycle
  - can be reused
- **Client**
  - requests the *ReusableObjects* from the *Pool*

by Shalloway & Trott



Pool of Objects: Consequences

- **improved performance**
  - *ReusableObjects* are initialized once and re-used multiple times
  - faster request handling
  - balanced, smooth resource consumption
- **better encapsulation and higher cohesion**
  - *Pool* manages the *ReusableObjects'* lifecycle
  - *Client* contacts only the *Pool*
- **load balancing**

by Shalloway & Trott

Pool of Objects: Example

javax.sql.DataSource is a generic interface for database connection factories. It defines a *getConnection()* method, which provides a client with an wrapped connection instance.

Invoking *close()* on the wrapper does not close the connection itself; it simply returns it to the *DataSource*, which therefore acts as a self-recovering pool of objects.

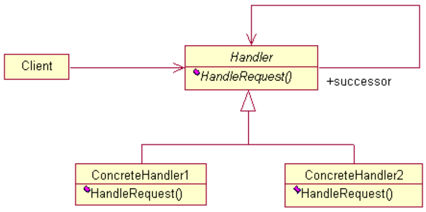
Pools are often implemented as singletons, usually they are also multithreaded (with *getObject()* and *returnObject()* synchronized)

Chain of Responsibility: Intent

- Avoid coupling the sender of a request to its receiver by **giving more than one object a chance to handle the request**
- Chain the receiving objects and **pass the request along the chain** until an object handles it

by the Gang of Four

Chain of Responsibility: Structure



by the Gang of Four

Chain of Responsibility: Participants

- **Handler**
  - defines an interface for handling requests
  - (optional) implements the successor link
- **Concrete Handler**
  - handles requests it is responsible for
  - can access its successor
  - if the *ConcreteHandler* can handle the request, it does so; otherwise it forwards the request to its successor
- **Client**
  - initiates the request to a *ConcreteHandler* object on the chain

by the Gang of Four

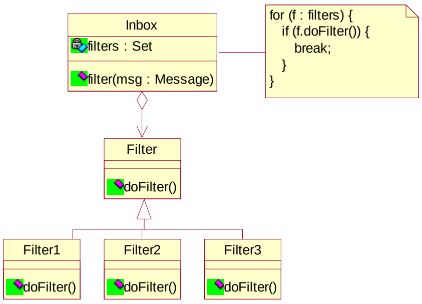
Chain of Responsibility: Consequences

Chain of Responsibility allows for:

- reduced coupling
  - Object does not need to know which other object handles the request
  - both Sender and Receiver have no knowledge of each other
  - chain elements do not know the chain's structure
- added flexibility in assigning responsibilities to objects
  - responsibilities can be freely distributed among chain elements
- no guarantee of receipt

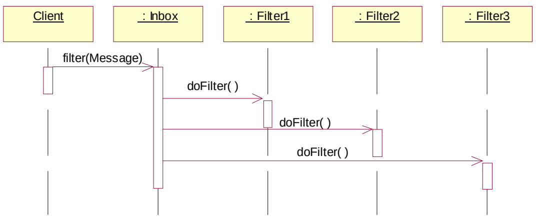
by the Gang of Four

Chain of Responsibility: Example 1

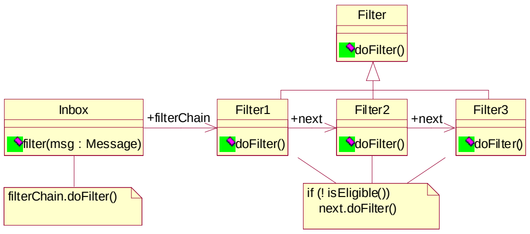


```
for (f : filters) {
    if (f.doFilter()) {
        break;
    }
}
```

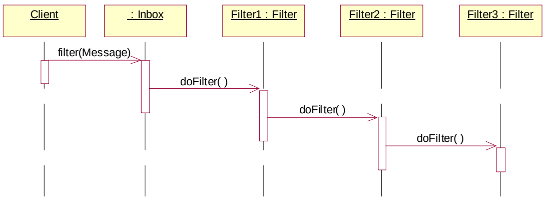
Chain of Responsibility: Example 1



Chain of Responsibility: Example 2



Chain of Responsibility: Example 2

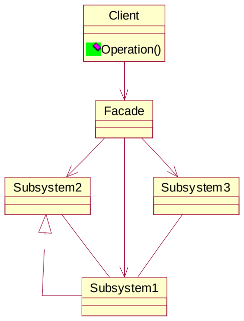


Facade: Motivation

- Provide a **unified interface to a set of interfaces** in a subsystem
- Facade **defines a higher-level interface** that makes the subsystem easier to use

by the Gang of Four

Facade: Structure



by the Gang of Four

Facade: Participants

- **Facade**
  - knows which subsystem classes are responsible for handling the request
  - delegates client requests to appropriate subsystem objects
- **subsystem classes**
  - implement subsystem functionality
  - handle work assigned to them by the *Facade* object
  - have no knowledge of the facade, i.e. they keep no references to it

by the Gang of Four

Facade: Consequences

- **shielding clients from subsystem components**
  - clients may communicate with subsystems via *Facade*
  - subsystems are easier to use and cheaper in maintenance
- **flexible access to subsystems**
  - clients can call both *Facade* and underlying subsystems
- **promotion of looser coupling between the subsystems and their clients**
  - subsystems can be hidden from the clients
  - subsystems can be easily replaced
- **provisioning of only partial functionality by the *Facade***

by the Gang of Four

Facade: Example

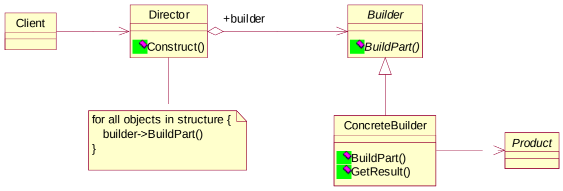
```
public class Email {                                   // facade
   MimeMessage msg = null;                             // subsystem1
   Session session = Session.getInstance(null, props); // subsystem2

   public Email(String subject, String text) {
      msg = new MimeMessage(session);
      msg.setFrom(DEFAULT_FROM);
      msg.setSubject(subject);
      msg.setText(text, "UTF-8");
   }

   public void sendTo(String[] to) {
      msg.setRecipients(Message.RecipientType.TO, convert(to));
      Transport transport = session.getTransport("smtp");
      transport.sendMessage(msg, msg.getAllRecipients()
   }

   public void sendTo(String[] to, String[] cc) {
      msg.setRecipients(Message.RecipientType.TO, convert(to));
      msg.setRecipients(Message.RecipientType.CC, convert(Cc));
      Transport transport = session.getTransport("smtp");
      transport.sendMessage(msg, msg.getAllRecipients()
   }
}
```

Builder: Motivation

- **Separate the construction** of a complex object **from its representation**
- The **same construction process** can create **different representations**

by the Gang of Four

Builder: Structure



by the Gang of Four

Builder: Participants

- **Builder**
  - specifies an abstract interface for creating parts of a *Product*
- **Concrete Builder**
  - constructs and assembles parts of the product by implementing the *Builder* interface
  - defines and keeps track of representation it creates
  - provides an interface for retrieving the *Product*
- **Director**
  - constructs an object using the *Builder* interface
- **Product**
  - includes classes that define its parts, including interfaces for assembling the parts into the final result

by the Gang of Four

Builder: Consequences

- ***Product's* internal representations may vary**
- **isolation the construction code from representation code**
- **finer control over the construction process**
- **improved testability**

by the Gang of Four

Builder: Example

```
public class Director {
   Builder roofBuilder = new RoofBuilder();
   Builder wallBuilder = new WallBuilder();
   Builder foundationBuilder = new FoundationBuilder();

   public House assemble() {
      Roof roof = (Roof) roofBuilder.build();
      Part wall = (Wall) roofBuilder.build();
      Part foundation = (Foundation) foundationBuilder.build();

      House house = new House();
      house.setFoundation (foundation);
      house.setWall(wall);
      house.setRoof(roof);

      return house;
   }
}

public interface Builder {
   public Part build();
}
```

Memento: Motivation

Without violating encapsulation, **capture and externalize an object's internal state** so that the object can be **restored to this state later**

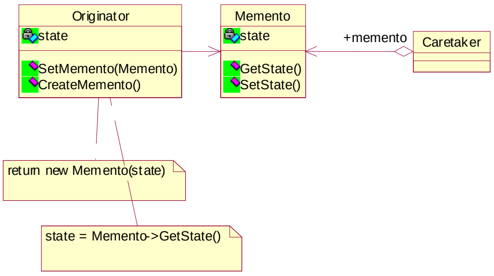
by the Gang of Four

Memento: Applicability

- A snapshot of an **object state must be saved** so that it can be restored later
- A direct interface to obtaining the state would expose implementation details and **break the encapsulation**

by the Gang of Four

Memento: Structure



by the Gang of Four

Memento: Participants

- **Memento**
  - stores original state of the Originator state
  - protects agains access by objects other that the Originator
- **Originator**
  - creates a Memento containing a snapshot of its current internal state
  - uses the Memento to restore its internal state
- **Caretaker**
  - is responsible for Memento safekeeping
  - never examines the content of a memento

by the Gang of Four

Memento: Implementation

- **Two interfaces of Memento**
  - narrow – *Caretaker* can only pass *Memento* to other objects
  - wide – *Originator* sees all data needed to restore its state
- **Java implementation**
  - *Memento* as an inner class of the *Originator*
  - *Memento* and *Originator* within a common package (methods accessible at default security level)
- **C++ implementation**
  - *Originator* is a friend class to *Memento*

Memento: Example

```
public class Account {
   private int balance = 0;

   public void credit(int amount) {
      balance += amount;
   }

   public void debit(int amount) {
      balance -= amount;
   }

   public void setMemento(Memento memento) {
      memento.restoreState();
   }

   public Memento createMemento() {
      Memento mementoToReturn = new Memento();
      mementoToReturn.setState();

      return mementoToReturn;
   }
```

Memento: Example

```
public class Account {
   // continued...

   class Memento {
      int mementoBalance = 0;

      public void setState() {
         mementoBalance = balance;
      }

      public void restoreState() {
         balance = mementoBalance;
      }
   }
}
```

Memento: Example

```
public class Bank {
   public static void main(String[] args) {
      Account.Memento caretaker = null;

      Account account = new Account("John Smith");
      account.credit(200);    // balance = 200

      account.createMemento();

      account.debit(100);     // balance = 100
      account.debit(50);      // balance = 50

      account.setMemento();

      account.debit(50);      // balance = 150
   }
}
```

Memento: Consequences

- **preserving encapsulation boundaries**

  it shields other objects from potentially complex *Originator* internals
- **simplifying the Originator**

  all storage management burden put on *Originator*

  clients ask the *Originator* to store and restore the *Mementos*
- **defining narrow and wide interfaces**
- **potential memory overhead**

  Originator may copy large amounts of state data

  Caretaker in unaware of the *Memento's* size

by the Gang of Four

Prototype: Motivation

Specify the kinds of objects to create using a prototypical instance, and **create new objects by cloning this prototype**

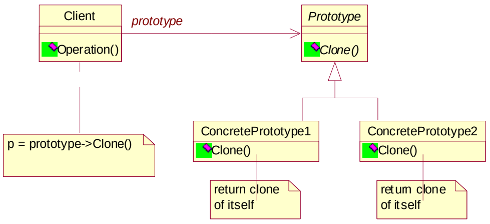
by the Gang of Four

Prototype: Applicability

- Classes to instatiate are **specified at runtime**.
- To avoid **parallel hierarchies of factories and products**.
- Instances of a class can have one of **only few different combinations of state**.

by the Gang of Four

Prototype: Structure



by the Gang of Four

Prototype: Participants

- **Prototype**
  - declares an interface for cloning itself.
- **Concrete Prototype**
  - implements an operation for cloning itself.
- **Client**
  - creates a new object by asking a prototype to clone itself.

by the Gang of Four

Prototype: Consequences

- **manipulating products at runtime**

  new prototypical instances can register to be available for cloning
- **specyfying new object by varying values and structure**

  Prototype creates new objects as instatiation does
- **reduced subclassing**

by the Gang of Four

Prototype: Example

```
public class Employee {
   private String name = null;
   private Date birthday = null;

   public Employee(String name, Date birthday) {
       this.name = name;
       this.birthday = birthday;
   }
}
```

```
Employee emp = new Employee("John Smith", new Date());
Employee emp2 = (Employee) emp.clone();

java.lang.CloneNotSupportedException
```

Prototype: Example

```
public class Employee implements Cloneable {
   private String name = null;
   private Date birthday = null;

   public Employee(String name, Date birthday) {
       this.name = name;
       this.birthday = birthday;
   }

   public Object clone() throws CloneNotSupportedException {
       // some specific cloning (eg. deep copy)
       return super.clone();
   }
}
```

```
Employee emp = new Employee("John Smith", new Date());
Employee emp2 = (Employee) emp.clone();
assertEquals(emp, emp2);
```

State: Motivation

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

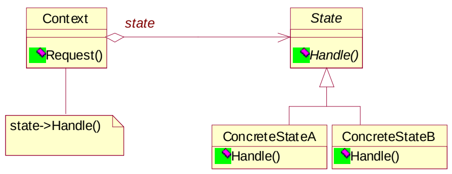
by the Gang of Four

State: Applicability

- **Object's behavior depends on its state**
- The behavior must **change at runtime**
- Operations have **large, multipart conditionals** that depend on the object's state
- Each state is a self-contained object, that can vary **independently from other objects**

by the Gang of Four

State: Structure



by the Gang of Four

State: Participants

- **Context**
  - defines the interface of interest to clients
  - maintains an instance of a *ConcreteState* subclass that defines the current state
- **State**
  - defines an interface for encapsulating the behavior associated with a particular state of the *Context*
- **Concrete State subclasses**
  - each subclass implements a behavior associated with a state of the *Context*

by the Gang of Four

State: Consequences

- **partition of the behavior for different states**

  state-specific code lives in *State* subclasses

  intent of the state-dependent behavior is clearer
- **explicit state transitions**
- **protection from inconsistent internal states**
- **possible sharing of state objects**

  States are stateless

by the Gang of Four

State: Example

```java
public class Account {
   private int balance = 0;
   private String owner = null;
   private boolean isOpen = false;

   public Account(String owner, int balance) {
      this.owner = owner;
      this.balance = balance;
      this.isOpen = true;
   }
   public void credit(int amount) {
      if (isOpen) {
         balance += amount;
      } else {
         alert("The account is closed!");
      }
   }
}
```

State: Example

```java
public interface AccountState {
    public void credit(Account acc, int amount);
}

public class AccountOpen implements AccountState {
    public void credit(Account acc, int amount) {
        acc.balance += amount;
    }
}

public class AccountClosed implements AccountState {
    public void credit(Account acc, int amount) {
        alert("The account is closed!");
    }
}
```

State: Example

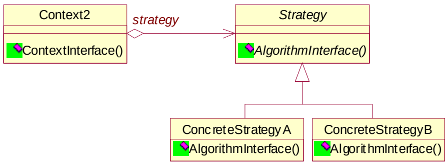
```
public class Account {
   private int balance = 0;
   private String owner = null;
   private AccountState state = null;

   public Account(String owner, int balance) {
      this.owner = owner;
      this.balance = balance;
      this.state = new AccountOpen();
   }

   public void credit(int amount) {
      this.state.credit(this, amount);
   }

   public void close() {
      this.state = new AccountClosed();
   }
}
```

Strategy: Applicability

- **Define a family of algorithms**, encapsulate each one, and **make them interchangeable**
- Strategy **lets the algorithm to vary independently from clients** that use it

by the Gang of Four

Strategy: Structure



by the Gang of Four

Strategy: Participants

- **Strategy**
  - declares an interface common to all supported algorithms. *Context* uses this interface to call the algorithm defined by a *ConcreteStrategy*
- **Concrete Strategy**
  - implements the algorithm using the *Strategy* interface
- **Context**
  - is configured with a *ConcreteStrategy* object
  - maintains a reference to a *Strategy* object
  - may define an interface that lets *Strategy* access its data

by the Gang of Four

Strategy: Consequences

- **reusable families of related algorithms**
- **flexible alternative to plain subclassing**

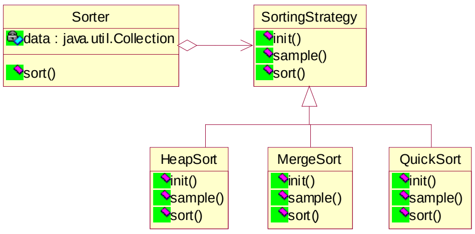
  algorithms can be exchanged dynamically

  clear separation of common and specific behavior
- **elimination of complex conditionals**
- **often client must be aware of different Strategies**
- **possible high complexity of Strategy interface**

  all Strategies must share common interface, regardless from their needs
- **increased number of objects within application**

by the Gang of Four

Strategy: Example 1



Strategy: Example 2

```
public class BannerGenerator {
   OuputDevice device = null;

   public BannerGenerator() {
      device = new Screen();
   }

   public void generate(String text1, String text2) {
      device.CR();
      device.print(text1);
      device.LF();
      device.CR();
      device.print(text2);
      device.FF();
   }
}
```

Strategy: Example 2

```
interface OutputDevice {
   public String print(String text);
   public String CR();
   public String LF();
   public String FF();
}

class Screen implements OutputDevice {
   public String print(String text) {
      System.out.print(text);
   }
   public String CR() {
      System.out.print("\n");
   }
   public String LF() {
      System.out.print("\n");
   }
   public String FF() {
      System.out.print("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
   }
}
```

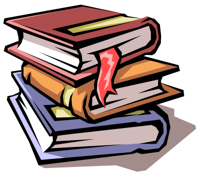
Strategy: Example 2

```
interface OutputDevice {
   public String print(String text);
   public String CR();
   public String LF();
   public String FF();
}

class Printer implements OutputDevice {
   private PrnDriver prn = new PrnDriver();

   public String print(String text) {
      prn.print(text);
   }
   public String CR() {
      prn.return();
   }
   public String LF() {
      prn.nextLine();
   }
   public String FF() {
      prn.nextPage();
   }
}
```

Design patterns

... to be continued again...

Readings



1. Gamma E. et al., *Design Patterns. Elements of Reuseable Object-Oriented Software.* Addison-Wesley, 1995
2. Eckel B., *Thinking in patterns.* http://www.bruceeckel.com
3. Cooper J., *Java. Wzorce Projektowe.* Helion, 2001
4. Shalloway A., Trott J., *Projektowanie zorientowane obiektowo. Wzorce projektowe.* Wydanie II. Helion, 2005

Q & A