Software Development Good Practices

Design Patterns

Olivier Boissier

Mines Saint-Etienne

Olivier.Boissier@emse.fr

Fall 2019

Outline

- **Motivations**
- Definitions
- Design Patterns
- Anti Patterns

Motivations
Usual life of a software source code

- Once upon a time, my software had
  - a beautiful design and coding
- … then …
  - A first problem or new requirement arrived
- … then …
  - A second one and an other, …

- Maintenance becomes a nightmare
- And now, design and code are full of ”horrible” adaptations

Motivations (cont'd)
… to Application Frameworks

- A class is a mechanisms for encapsulation,
  - it embodies a certain service providing the data and behavior that is useful in the context of some application.
  - A single class is rarely the complete solution to some real problem

- There is a growing body of research in describing the manner in which collections of classes work *together* in the solution of problems.

- *Application frameworks* and *design patterns* are two ideas that became popular in this context

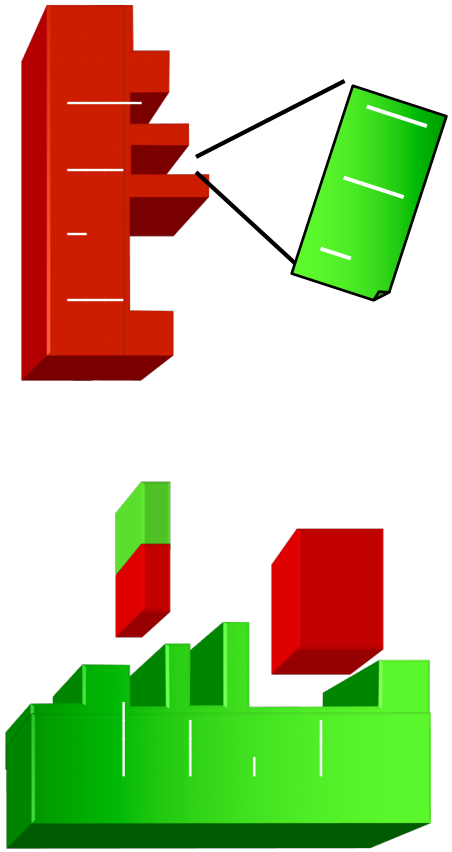
Application Frameworks

- An application framework is a set of classes that:
  - cooperate closely with each other
  - and **together** embody a reusable design for a general category of problems

- Although one might abstract and discuss the design elements that lie behind the framework, the framework itself is a set of specific classes that are typically implemented only on a specific platform or limited set of platforms

- The framework dictates the overall structure and behavior of the application.
- It describes how responsibilities are partitioned between various components and how these components interact

A Framework as an Upside-down Library



- In a traditional application the application-specific code defines the overall flow of execution through the program occasionally invoking library-supplied code
- A framework reverts this relation: the flow of control is dictated by the framework and the creator of a new application merely changes some of the methods invoked by the framework
- Inheritance is often used as powerful mechanism for achieving this. Alternatively application-specific components that obey a specified interface are plugged in

Application Framework Example (1)

- GUI application frameworks simplify the creation of graphical user interfaces for software systems
- A GUI application framework implements the behavior expected from a graphical user interface (hold windows, buttons, menu's, textfields, etc. - move and resize windows - handle mouse events on buttons and menus- ...)
- A new application is build by specifying and arranging the necessary elements (buttons, menu's, textfields, etc.) and by redefining certain methods (responses to the mouse and key events - …)

Application Framework Example (2)

- Simulation frameworks simplify the creation of simulation style applications
- A simulation framework provides a general-purpose class for managing the types of objects in the simulation
- The heart of a simulation framework is a procedure that cycles through the list of objects introduced in the simulation asking each to update itself
- The framework knows nothing about the particular application it is going to be used for: billiard balls, fish in a tank, rabbits and wolves in an ecological game, etc.

Why using Principles and Patterns?

- Principles:
  - organisation of the dependence relations between classes, modules, components
- Patterns:
  - Abstraction of problems and proposal of abstract solutions
- For:
  - Systematic (software-)development:
  - Documenting expert knowledge
  - Use of generic solutions
  - Raising the abstraction level
  - Raising quality of solutions

Some criteria

- Modularity
  - Ease the management (object technology)
- Cohesion
  - measure of relatedness or consistency in the functionality of a software unit
  - Degree of implemented functionalities strongly related to the concern that it is meant to model.
  - Strong cohesion is good quality
- Coupling
  - Degree with which methods of different modules are dependent on each other
  - A loose coupling is good quality
- Reusability
  - Libraries, frameworks

Cohesion: ``bad'' example

Why?

```
public class GameBoard {

public GamePiece[ ][ ] getState() { … }
 // Méthode copiant la grille dans un tableau temporaire, résultat de l'appel de la méthode.

public Player isWinner() { … }
 // vérifie l'état du jeu pour savoir s'il existe un gagnant, dont la référence est retournée.
 // Null est retourné si aucun gagnant.

public boolean isTie() { … }
 //retourne true si aucun déplacement ne peut être effectué, false sinon.

public void display () { … }
// affichage du contenu du jeu. Espaces blancs affichés pour chacune des
// références nulles.
}
```

GameBoard is responsible of the rules of the game and of the display

Cohesion: ``good'' example

```
public class GameBoard {

public GamePiece[ ][ ] getState() { … }

public Player isWinner() { … }

public boolean isTie() { … }

}

public class BoardDisplay {

public void displayBoard (GameBoard gb) { … }
// affichage du contenu du jeu. Espaces blancs affichés pour chacune des
// références nulles.
}
```

Coupling: example

Is it loosely coupled?

```
void initArray(int[] iGradeArray, int nStudents) {
   int i;
   for (i = 0; i < nStudents; i++) {
      iGradeArray[i] = 0;
   }
}
```

Coupling between client and initArray by ``nStudents'' parameter

Is it loosely coupled?

```
void initArray(int[ ] iGradeArray) {
   int i;
   for (i=0; i < iGradeArray.length; i++)  {
      iGradeArray[i] = 0;
   }
}
```

Loose coupling
Through the use of ``length'' attribute

The SOLID Principles

Guidelines that, when followed, can dramatically enhance the maintainability of software.

- **Single Responsibility Principle (SRP)**. The SRP states that each class or similar unit of code should have one responsibility only and, therefore, only one reason to change.
- **Open / Closed Principle (OCP)**. The OCP states that all classes and similar units of source code should be open for extension but closed for modification.
- **Liskov Substitution Principle (LSP)**. The LSP specifies that functions that use pointers of references to base classes must be able to use objects of derived classes without knowing it.
- **Interface Segregation Principle (ISP)**. The ISP specifies that clients should not be forced to depend upon interfaces that they do not use. Instead, those interfaces should be minimised.
- **Dependency Inversion Principle (DIP)**. The DIP states that high level modules should not depend upon low level modules and that abstractions should not depend upon details.

Design Principles Example

```
class StudentRecord {
   private Name lastName;
   private Name firstName;
   private long ID;
   public Name getLastName() { return lastName; }
   … // etc
}
```

```
class SortedList {
   Object[] sortedData = new Object[size];
   public void add(StudentRecord X) {
        // …
        Name a = X.getLastName();
        Name b = sortedData[k].getLastName();
        if (a.lessThan(b)) … else … //do something
   }
… }
```

Design Principles
Example Solution 1

```
class StudentRecord {
   private Name lastName;
   private Name firstName;
   private long ID;
   public boolean lessThan(Object X) {
    return lastName.lessThan(X.lastName);}
   … // etc
}
```

```
class SortedList {
   Object[] sortedData = new Object[size];
   public void add(StudentRecord X) {
        // …
        if (X.lessThan(sortedData[k])) … else … //do something
   }
… }
```

Design Principles
Example Solution 2

```
Interface Comparable {
    public boolean lessThan(Object X);
    public boolean greaterThan(Object X);
    public boolean equal(Object X);
}
```

```
class StudentRecord implements Comparable {
    private Name lastName;
    private Name firstName;
    private long ID;
    public boolean lessThan(Object X) {
          return lastName.lessThan(((StudentRecord)X).lastName);}
    … // etc
}
```

```
class SortedList {
    Object[] sortedData = new Object[size];
    public void add(Comparable X) {
          // …
          if (X.lessThan(sortedData[k])) … else … //do something
    } … }
```

Coming back to OCP

- The open-closed principle states that a software module should be:
  - *Open for extension* — It should be possible to alter the behavior of a module or add new features to the module functionality.
  - *Closed for modification* — Such a module should not allow its code to be modified.

Becoming a Chess Master

- First learn rules and physical requirements
  - e.g., names of pieces, legal movements, chess board geometry and orientation, etc.
- Then learn principles
  - e.g., relative value of certain pieces, strategic value of center squares, power of a threat, etc.
- However, to become a master of chess, one must study the games of other masters
  - These games contain patterns that must be understood, memorized, and applied repeatedly
- There are hundreds of these patterns

Becoming a Software Designer Master

- First learn the rules
  - e.g., the algorithms, data structures and languages of software
- Then learn the principles
  - e.g., structured programming, modular programming, object oriented programming, generic programming, etc.
- However, to truly master software design, one must study the designs of other masters
  - These designs contain patterns must be understood, memorized, and applied repeatedly
- There are hundreds of these patterns

Outline

- Motivations
- **Definitions**
- Design Patterns
- Anti Patterns

Definition : Pattern

- Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

C. Alexander

A pattern is a common solution to a common problem in a given context

More on Patterns

- « Patterns help you build on the collective experience of skilled software engineers. »
- « They capture existing, well-proven experience in software development and help to promote good design practice »
- « Every pattern deals with a specific, recurring problem in the design or implementation of a software system »
- « Patterns can be used to construct software architectures with specific properties... »

Ecosystem of Patterns

- Programming Patterns (idioms)
  - low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.
  - e.g. singleton, string copy in C (while (*d++=*s++);
- Design Patterns
  - A design pattern provides a scheme for refining the subsystems or components of a software system, or the relation ships between them. It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context.
- Architectural Patterns
  - fundamental structural organization schema for software systems. It provides a set of predefined subsystems, their responsibilities, and includes rules and guidelines for organizing the relationships between them.
  - e.g. layers, distribution, security, MVC…

Ecosystem of Patterns (2)

- Requirement Patterns
- Testing Patterns
- Project Management Patterns
- Process Patterns
- Organizational Patterns
- …
- Anti-Patterns

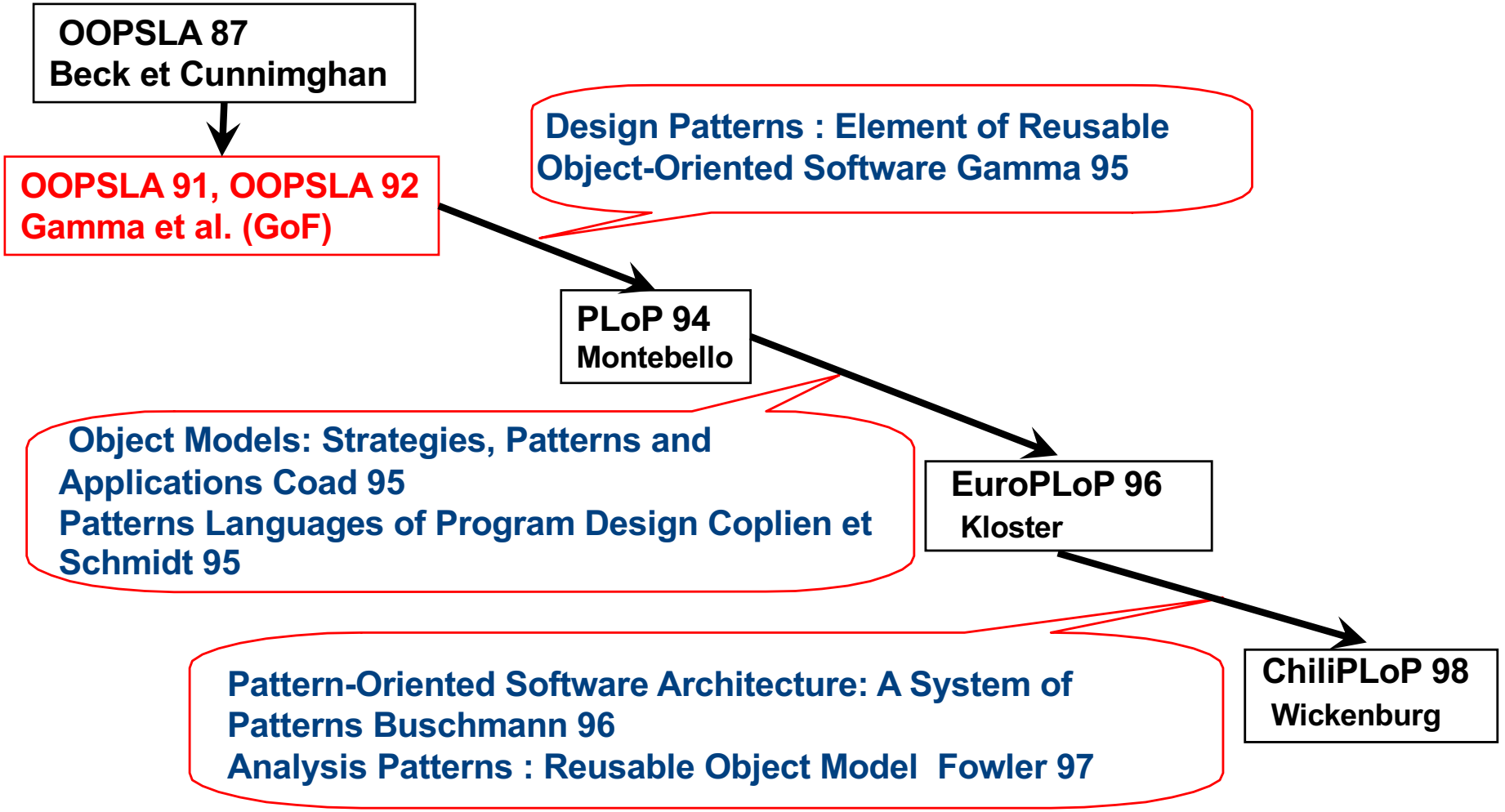
Definition : Design Pattern

A **design pattern** is a **general reusable** solution to a commonly occurring problem in software design.

A design pattern is **not** a **finished** design that can be transformed directly into code. It is a **description** or **template** for how to solve a problem that can be used in many different situations.

Object-oriented design patterns typically show **relationships** and **interactions** between classes or objects, without specifying the final application classes or objects that are involved.

History


OOPSLA 87
Beck et Cunnimghan
OOPSLA 91, OOPSLA 92
Gamma et al. (GoF)
Design Patterns : Element of Reusable Object-Oriented Software Gamma 95
PLoP 94
Montebello
Object Models: Strategies, Patterns and Applications Coad 95
Patterns Languages of Program Design Coplien et Schmidt 95
EuroPLoP 96
Kloster
Pattern-Oriented Software Architecture: A System of Patterns Buschmann 96
Analysis Patterns : Reusable Object Model Fowler 97
ChiliPLoP 98
Wickenburg

Outline

- Motivations
- Definitions
- **Design Patterns**
- Anti Patterns

Design Pattern Description

1. Pattern Name
   - A short mnemonic to increase your design vocabulary.
2. Problem
   - Description when to apply the pattern (conditions that have to be met before it makes sense to apply the pattern).
3. Solution
   - The elements that make up the design, their relationships, responsibilities and collaborations.
4. Consequences
   - Costs and benefits of applying the pattern. Language and implementation issues as well as impact on system flexibility, extensibility, or portability.
   - The goal is to help understand and evaluate a pattern.

Template for GoF Design Pattern

1. **Pattern Name and Classification**
   - A descriptive and unique name that helps in identifying and referring to the pattern.
- **Intent**
   - A description of the goal behind the pattern and the reason for using it.
- **Also Known As**
   - Other names for the pattern.

2. **Motivation (Forces)**
   - A scenario consisting of a **problem** and a **context** in which this pattern can be used.
- **Applicability**
   - Situations in which this pattern is usable; the context for the pattern.

Template for GoF Design Pattern (cont'd)

3. **Structure**
   - A graphical representation of the pattern. Class diagrams and Interaction diagrams may be used for this purpose.
- **Participants**
  - A listing of the classes and objects used in the pattern and their roles in the design.
- **Collaboration**
  - A description of how classes and objects used in the pattern interact with each other.

4. **Consequences**
   - A description of the results, side effects, and trade offs caused by using the pattern.
   - Assessments of resource usage (memory usage, running time)
   - Influence on flexibility, extendibility and portability

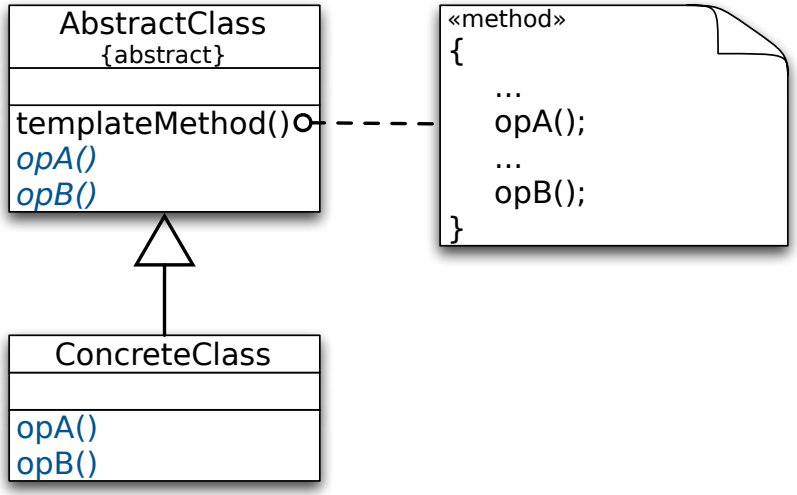
Template for GoF Design Pattern (cont'd)

5. **Implementation**
   - A description of an implementation of the pattern; the solution part of the pattern.
- **Sample Code**
  - An illustration of how the pattern can be used in a programming language.
- **Known Uses**
  - Examples of real usages of the pattern.
- **Related Patterns**
  - Other patterns that have some relationship with the pattern; discussion of the differences between the pattern and similar patterns.

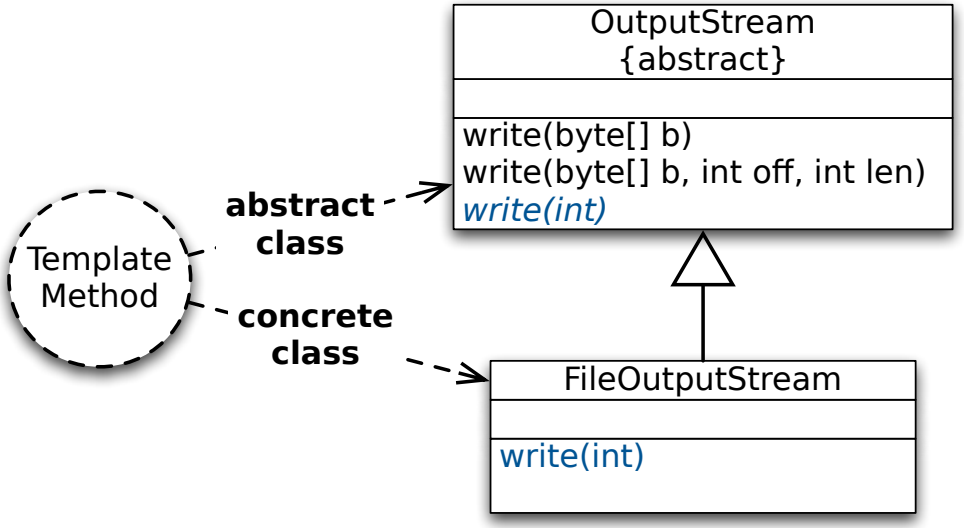
Documenting used design pattern in a design

- Use the participant name of the pattern to specify a class' role in the implementation of patterns

Template Method Pattern



Use of the Template Method Pattern in Java



Purposes of Design Patterns

Elements of Reusable Software	patterns foster reusability
Reuse of Design	rather than code
Communication	design vocabulary
Documentation	information chunks
Language Design	high level languages
Teaching	passing on culture

Design Patterns Classification

Creational Patterns:
are concerned with the process of object creation

Structural Patterns:
are concerned with how classes and objects are composed to form larger structures

Behavioural Patterns:
are concerned with algorithms and the assignment of responsi-bilities between objects

Class Patterns deal with static relationships between classes and subclasses

Object Patterns deal with object relationships which can be changed at run time

Creational Patterns

- Abstracts the instantiation process:
  - Encapsulates knowledge about which concrete classes to use
  - Hides how the instances of these classes are created and put together
- Gives a lot of flexibility in **what** gets created, **who** creates it, **how** it gets created, and **when** it it gets created
- A **class creational pattern** uses inheritance to vary the class that is instantiated
- An **object creational pattern** delegates instantiation to another object

Creational Patterns (cont'd)

- **Abstract Factory** Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- **Builder** Separate the construction of a complex object from its representation so that the same construction process can create different representations.
- **Factory Method** Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- **Prototype** Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- **Singleton** Ensure a class only has one instance, and provide a global point of access to it.

Structural Patterns

- Structural design patterns are concerned with how classes and objects are composed to form larger structures

- A **class structural pattern** uses inheritance to compose interfaces or implementation; compositions are fixed at design time (e.g. Adaptor)

- An **object structural pattern** describes ways to compose objects to realise new functionality; the added flexibility of object composition comes from the ability to change the composition at run-time (e.g. Object Adaptor, Bridge, Façade, Proxy)

Structural Patterns (cont'd)

- **Adapter** Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- **Bridge** Decouple an abstraction from its implementation so that the two can vary independently.
- **Composite** Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- **Decorator** Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- **Facade** 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.
- **Flyweight** Use sharing to support large numbers of fine-grained objects efficiently.
- **Proxy** Provide a surrogate or placeholder for another object to control access to it.

Behavioral Patterns

- Behavioral design patterns are concerned with algorithms and the assignment of responsibilities between objects
- **Class behavioral patterns** use inheritance to distribute behavior between classes
- **Object behavioral patterns** use object composition rather than inheritance; some describe how a group of peer objects cooperate to perform a tasks no single object can carry out by itself; others are concerned with encapsulating behavior in an object and delegating request to it

Behavioral Patterns (cont'd)

- **Chain of Responsibility** 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.
- **Command** Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- **Interpreter** Given a language, define a represention for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
- **Iterator** Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- **Mediator** Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Behavioral Patterns (cont'd)

- **Memento** Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
- **Observer** Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **State** Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- **Strategy** Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- **Template Method** Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- **Visitor** Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

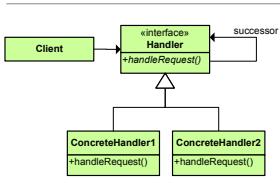
GoF Design Patterns

(Gamma, Helm, Johnson, Vlissides)

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter
				Template Method
	Object	Abstract Factory	Adapter (object)	Chain of Responsibility
		Prototype	Bridge	Command
		Builder	Composite	Iterator
		Singleton	Decorator	Mediator
			Facade	Memento
			Flyweight	Observer
			Proxy	State
				Strategy
				Visitor

based on book of Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Elements of Reusable Object-Oriented Software

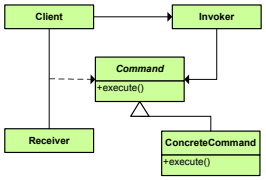
C Abstract Factory
S Adapter
S Bridge
C Builder
B Chain of Responsibility
B Command
S Composite
S Decorator
S Facade
C Factory Method
S Flyweight
B Interpreter
B Iterator
B Mediator
B Memento
C Prototype
S Proxy
B Observer
C Singleton
B State
B Strategy
B Template Method
B Visitor



Chain of Responsibility

Type: Behavioral

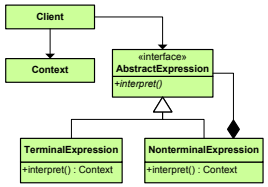
What it is:
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.



Command

Type: Behavioral

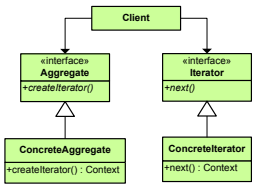
What it is:
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.



Interpreter

Type: Behavioral

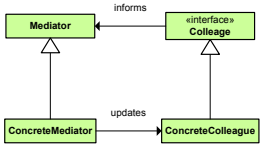
What it is:
Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.



Iterator

Type: Behavioral

What it is:
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.



Mediator

Type: Behavioral

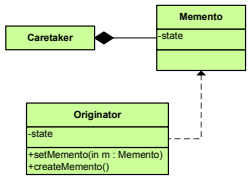
What it is:
Define an object that encapsulates how a set of objects interact. Promotes loose coupling by keeping objects from referring to each other explicitly and it lets you vary their interactions independently.

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison Wesley Longman, Inc.

Memento

Type: Behavioral

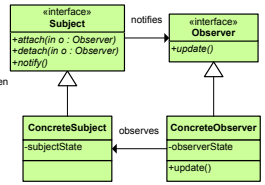
What it is:
Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.



Observer

Type: Behavioral

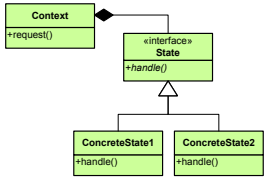
What it is:
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



State

Type: Behavioral

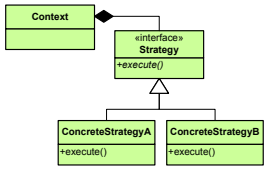
What it is:
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.



Strategy

Type: Behavioral

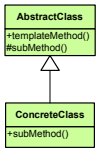
What it is:
Define a family of algorithms, encapsulate each one, and make them interchangeable. Lets the algorithm vary independently from clients that use it.



Template Method

Type: Behavioral

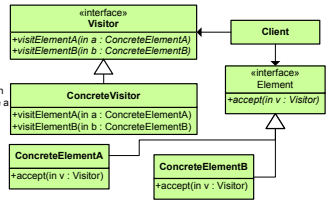
What it is:
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

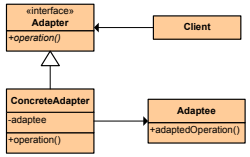


Visitor

Type: Behavioral

What it is:
Represent an operation to be performed on the elements of an object structure. Lets you define a new operation without changing the classes of the elements on which it operates.

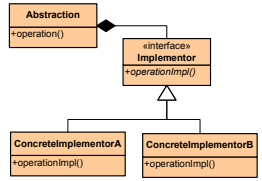




Adapter

Type: Structural

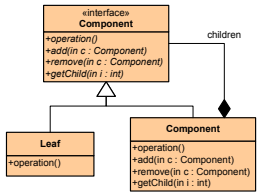
What it is:
Convert the interface of a class into another interface clients expect. Lets classes work together that couldn't otherwise because of incompatible interfaces.



Bridge

Type: Structural

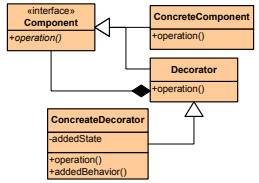
What it is:
Decouple an abstraction from its implementation so that the two can vary independently.



Composite

Type: Structural

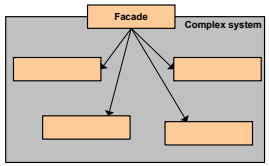
What it is:
Compose objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and compositions of objects uniformly.



Decorator

Type: Structural

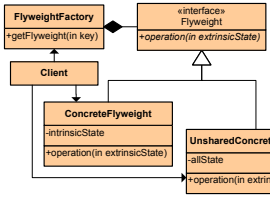
What it is:
Attach additional responsibilities to an object dynamically. Provide a flexible alternative to sub-classing for extending functionality.



Facade

Type: Structural

What it is:
Provide a unified interface to a set of interfaces in a subsystem. Defines a high-level interface that makes the subsystem easier to use.



Flyweight

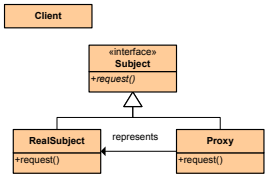
Type: Structural

What it is:
Use sharing to support large numbers of fine grained objects efficiently.

Proxy

Type: Structural

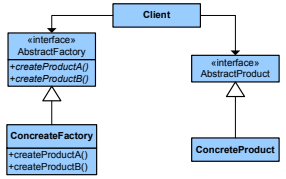
What it is:
Provide a surrogate or placeholder for another object to control access to it.



Abstract Factory

Type: Creational

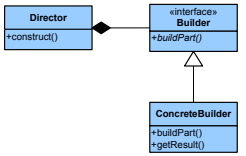
What it is:
Provides an interface for creating families of related or dependent objects without specifying their concrete class.



Builder

Type: Creational

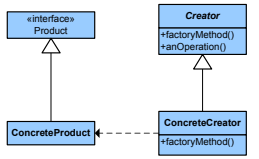
What it is:
Separate the construction of a complex object from its representing so that the same construction process can create different representations.



Factory Method

Type: Creational

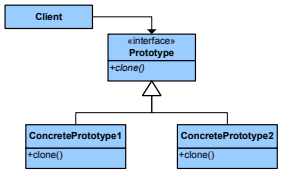
What it is:
Define an interface for creating an object, but let subclasses decide which class to instantiate. Lets a class defer instantiation to subclasses.



Prototype

Type: Creational

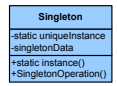
What it is:
Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.



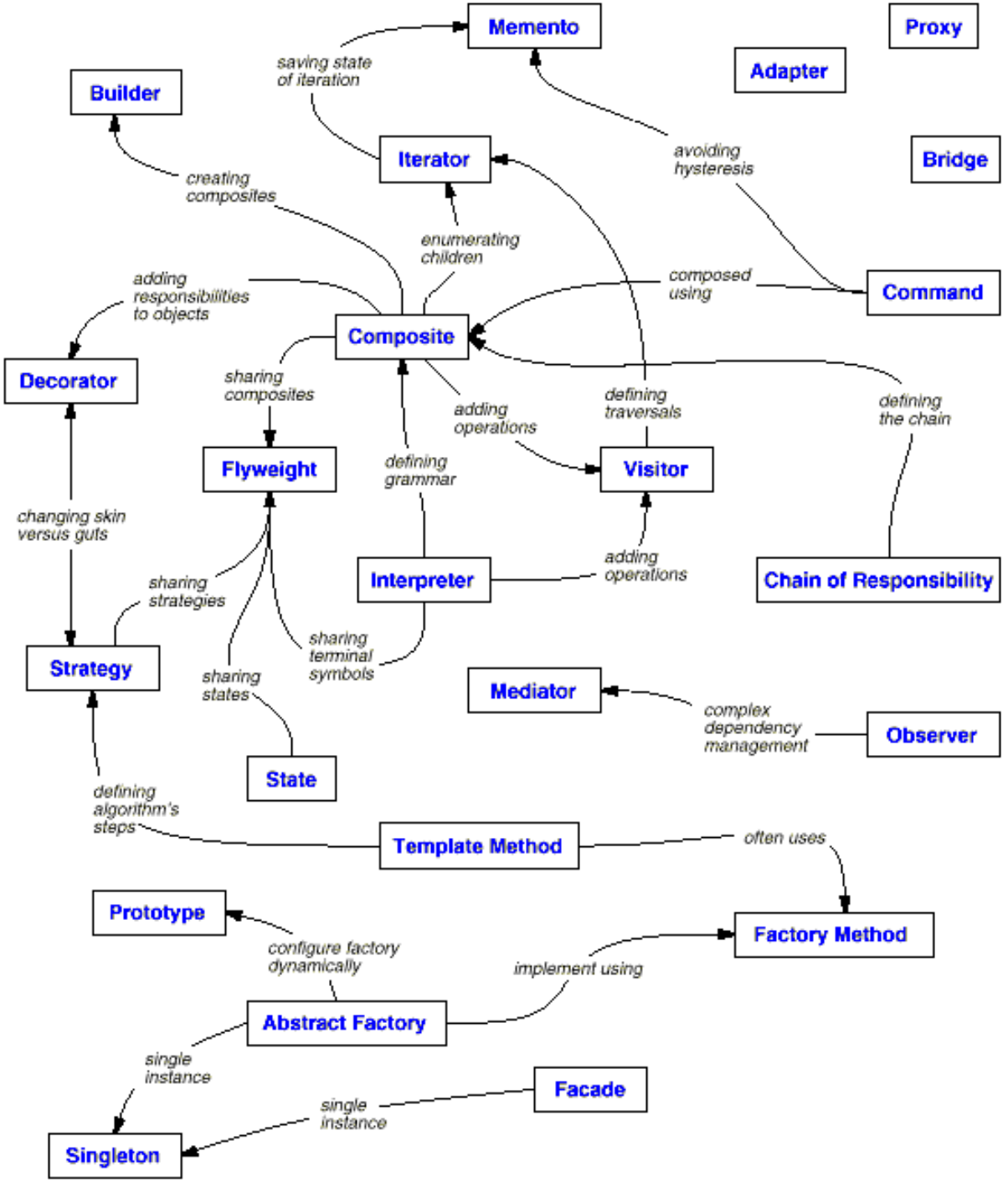
Singleton

Type: Creational

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



Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison Wesley Longman, Inc.



Design pattern relationships

Outline

- Motivations
- Definitions
- Design Patterns
- **Anti Patterns**

- If the following patterns occur in your software project, you're doing it wrong!!!

Anti-Patterns: Programming

- **The Blob**. (aka “God Class”) One object("blob") has the majority of the responsibilities, while most of the others just store data or provide only primitive services.
  - *Solution*: refactoring
- **The Golden Hammer**.A favorite solution ("Golden Hammer") is applied to every single problem:With a hammer, every problem looks like a nail.
  - *Solution*: improve level of education
- **Copy-and-Paste Programming**. Code is reused in multiple places by being copied and adjusted.This causes a maintenance problem.
  - *Solution*: Black box reuse, identifying of common features.
- **Spaghetti Code.** The code is mostly unstructured; it's neither particularly modular nor object-oriented; control flow is obscure.
  - *Solution*: Prevent by designing first, and only then implementing. Existing spaghetti code should be refactored.
- **Mushroom Management.** Developers are kept away from users.
  - *Solution*: Improve contacts.

Anti-Patterns: Architecture

- Vendor Lock-In. A system is dependent on a proprietary architecture or data format.
  - Solution: Improve portability, introduce abstractions.
- Design by Committee.The typical anti-pattern of standardizing committees, that tend to satisfy every single participant, and create overly complex and ambivalent designs ("A camel is a horse designed by a committee"). Known examples: SQL and COBRA.
  - Solution: Improve group dynamics and meetings (teamwork)
- **Reinvent the Wheel**. Due to lack of knowledge about existing products and solutions, the wheel gets reinvented over and over, which leads to increased development costs and problems with deadlines.
  - *Solution*: Improve knowledge management.

Anti-Patterns: Management

- Intellectual Violence. Someone who has mastered a new theory, technique or buzzwords, uses his knowledge to intimidate others.
  - Solution:Ask for clarification!
- Project Mismanagement. The manager of the project is unable to make decisions.
  - Solution: Admit having the problem; set clear short-term goals.

Drawbacks of Patterns

- Patterns do not lead to direct code reuse.
- Individual Patterns are deceptively simple.
- Composition of different patterns can be very complex.
- Teams may suffer from pattern overload.
- Patterns are validated by experience and discussion rather than by automated testing.
- Integrating patterns into a software development process is a humanintensive activity.

From « System of Patterns » Buschmann&All and « Design Pattern » Gamma&Al

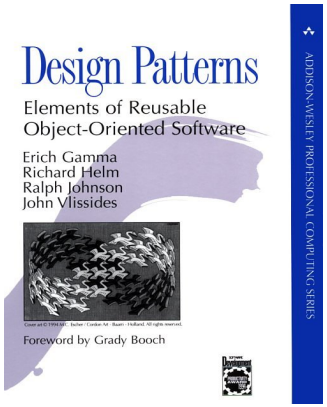
References

- ” Pattern Languages of Program Design ”, Coplien J.O., Schmidt D.C., Addison-Wesley, 1995.
- ” Pattern languages of program design 2 ”, Vlissides, et al, ISBN 0-201-89527-7, Addison-Wesley
- ” Pattern-oriented software architecture, a system of patterns ”, Buschmann, et al, Wiley
- ” Advanced C++ Programming Styles and Idioms ”, Coplien J.O., Addison-Wesley, 1992.
- S.R. Alpert, K.Brown, B.Woolf (1998) The Design Patterns Smalltalk Companion, Addison-Wesley (Software patterns series).
- J.W.Cooper (1998), The Design Patterns Java Companion, http://www.patterndepot.com/put/8/JavaPatterns.htm.
- S.A. Stelting, O.Maasen (2002) Applied Java Patterns, Sun Microsystems Press.
- Communications of ACM, October 1997, vol. 40 (10).
- Thinking in Patterns with Java http://mindview.net/Books/TIPatterns/

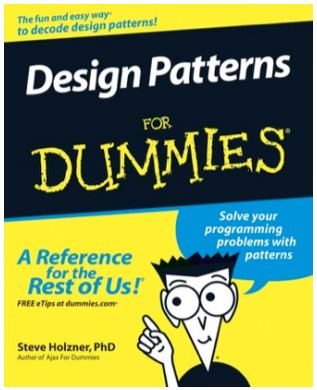
References

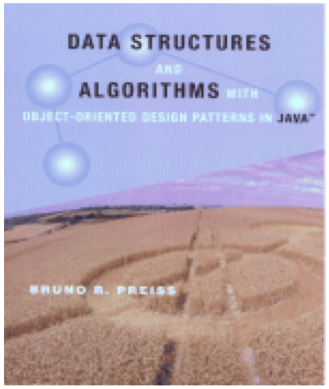
- http://hillside.net/
- http://www.c2.com/ppr (Portland Pattern Repository)
- http://www.industriallogic.com/papers/learning.html (A Learning Guide To Design Patterns)
- http://www.corej2eepatterns.com/index.htm (Core J2EE Patterns)
- https://www.oodesign.com/ (OODesign)
- https://www.youtube.com/playlist?list=PLF206E906175C7E07 Series of mini-tutorials (~15 min each) by Derek Banas about design patterns
- http://www.blackwasp.co.uk/gofpatterns.aspx BlackWasp Gang of Four Design Patterns
- https://martinfowler.com/eaaCatalog/index.html
- http://www.cloudcomputingpatterns.org/ (Cloud Computing Patterns)
- https://resources.oreilly.com/examples/9780596007126/ Head First Design Patterns Source Code (2014 update)

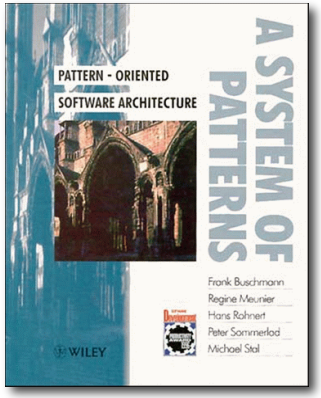
References

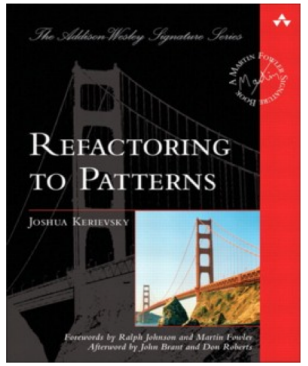

Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

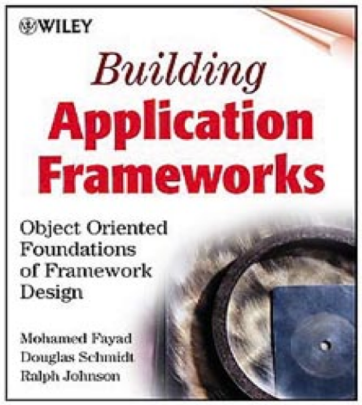

O'REILLY
Head First
Design Patterns
A Brain-Friendly Guide
Eric Freeman & Elisabeth Robson

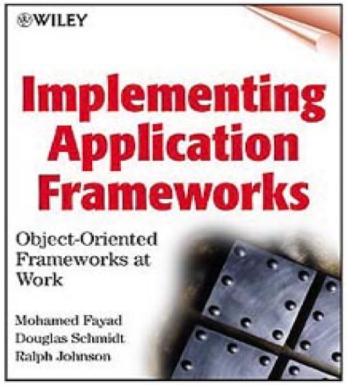

The fun and easy way to decode design patterns!
Design Patterns
FOR
DUMMIES
Solve your programming problems with patterns
A Reference for the Rest of Us!
FREE eTips at dummies.com
Steve Holzner, PhD

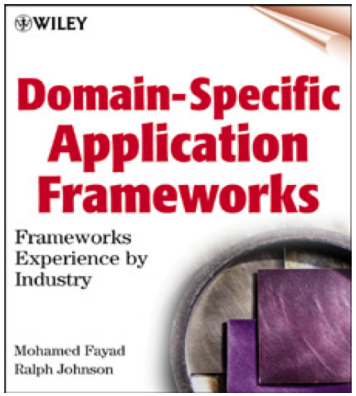

DATA STRUCTURES
AND
ALGORITHMS WITH
OBJECT-ORIENTED DESIGN PATTERNS IN JAVA
BRUNO R. PREISS


PATTERN - ORIENTED
SOFTWARE ARCHITECTURE
A SYSTEM OF PATTERNS
Frank Buschmann
Regine Meunier
Hans Rohnert
Peter Sommerlad
Michael Stal
WILEY

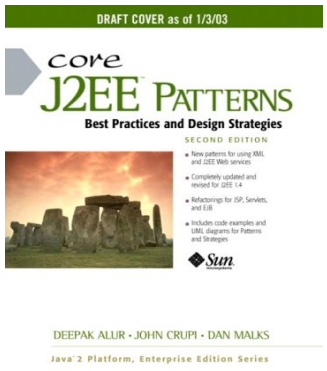

The Addison Wesley Signature Series
REFACTORING
TO PATTERNS
JOSHUA KERIEVSKY


WILEY
Building
Application
Frameworks
Object Oriented Foundations of Framework Design
Mohamed Fayad
Douglas Schmidt
Ralph Johnson


WILEY
Implementing
Application
Frameworks
Object-Oriented Frameworks at Work
Mohamed Fayad
Douglas Schmidt
Ralph Johnson


WILEY
Domain-Specific
Application
Frameworks
Frameworks Experience by Industry
Mohamed Fayad
Ralph Johnson


Douglas Schmidt
Michael Stal
Hans Rohnert
Frank Buschmann
PATTERN-ORIENTED
SOFTWARE
ARCHITECTURE
Volume 2
Patterns for Concurrent and Networked Objects
WILEY
SOFTWARE DESIGN PATTERNS


DRAFT COVER as of 1/3/03
core
J2EE PATTERNS
Best Practices and Design Strategies
SECOND EDITION
DEEPAK ALUR • JOHN CRUPI • DAN MALKS
Java 2 Platform, Enterprise Edition Series