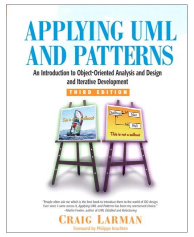
CMPE 202

Introduction to Design Patterns

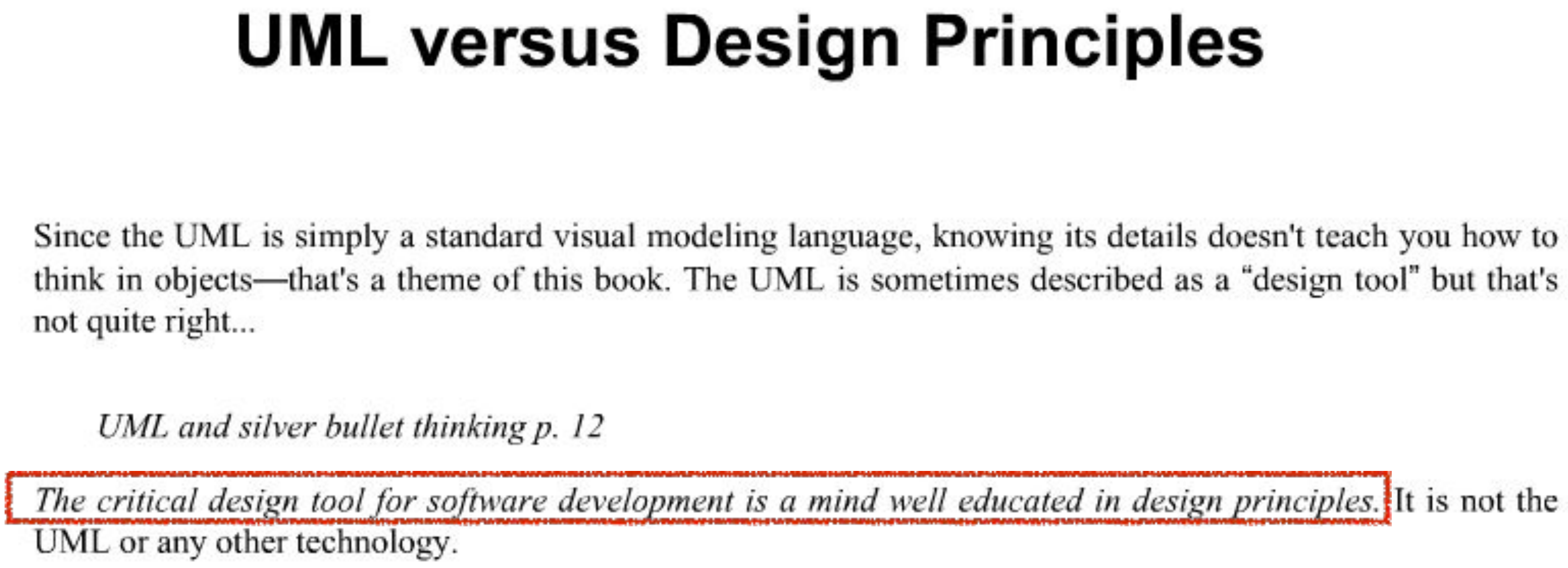

APPLYING UML AND PATTERNS

An Introduction to Object-Oriented Analysis and Design and Iterative Development

THIRD EDITION

CRAIG LARMAN

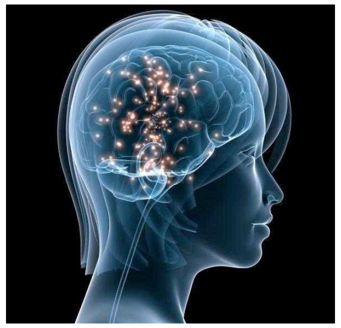
UML != Design


UML versus Design Principles

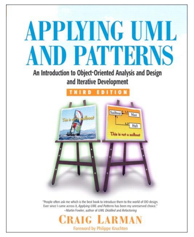
Since the UML is simply a standard visual modeling language, knowing its details doesn't teach you how to think in objects—that's a theme of this book. The UML is sometimes described as a "design tool" but that's not quite right...

UML and silver bullet thinking p. 12

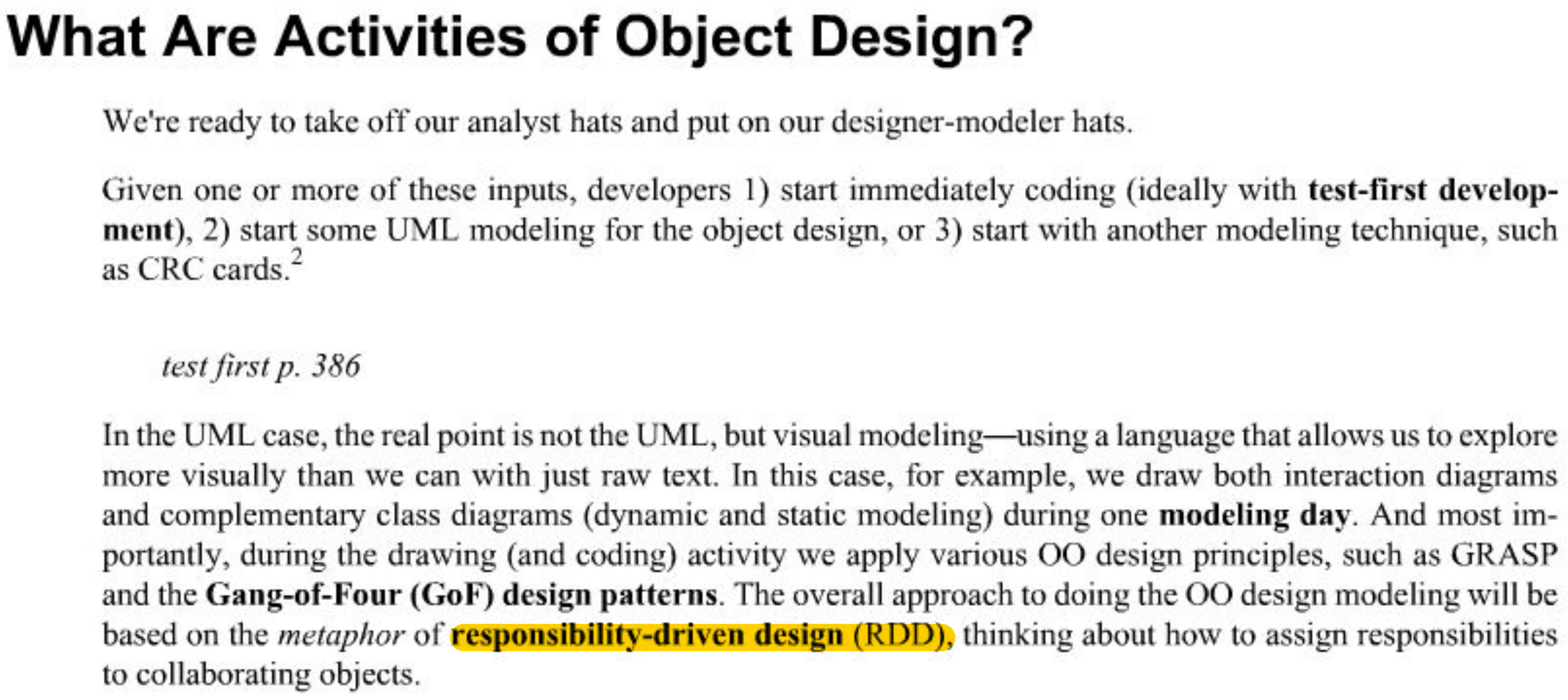
The critical design tool for software development is a mind well educated in design principles. It is not the UML or any other technology.



= Design



Why not just dive into Coding?



What Are Activities of Object Design?

We're ready to take off our analyst hats and put on our designer-modeler hats.

Given one or more of these inputs, developers 1) start immediately coding (ideally with **test-first development**), 2) start some UML modeling for the object design, or 3) start with another modeling technique, such as CRC cards.[2]

test first p. 386

In the UML case, the real point is not the UML, but visual modeling—using a language that allows us to explore more visually than we can with just raw text. In this case, for example, we draw both interaction diagrams and complementary class diagrams (dynamic and static modeling) during one **modeling day**. And most importantly, during the drawing (and coding) activity we apply various OO design principles, such as GRASP and the **Gang-of-Four (GoF) design patterns**. The overall approach to doing the OO design modeling will be based on the *metaphor* of **responsibility-driven design** (RDD), thinking about how to assign responsibilities to collaborating objects.

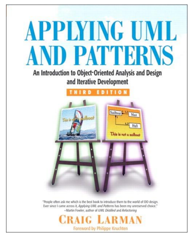
Why not do both?

Test-Driven Development

1. Write Code
2. Write Unit Tests (Design by Contract)
3. Do some modeling
   (CRC cards, maybe UML)
4. Refactor

Responsibility-Driven Design

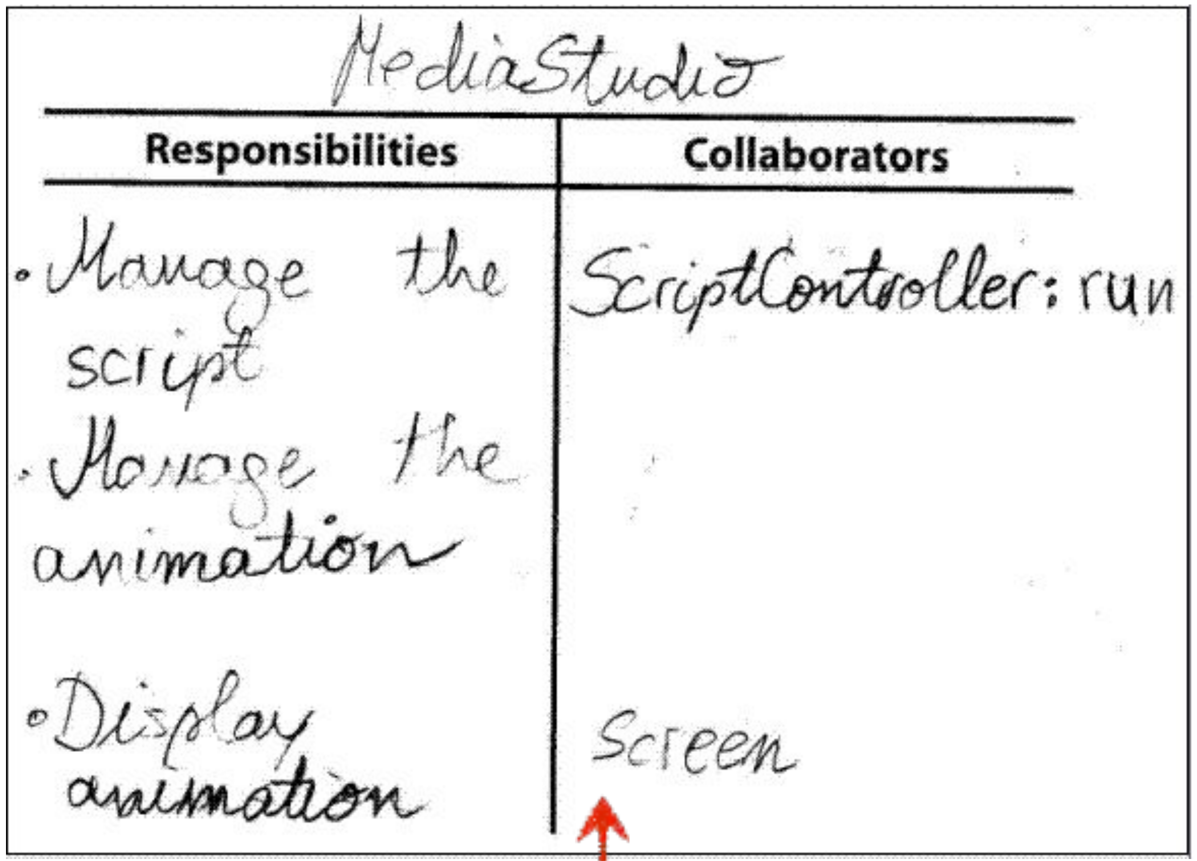
1. Use "Visual" Models (UML)
2. Design with Patterns
3. Think about "Collaborating" Objects
4. Understand the "Vision"


APPLYING UML AND PATTERNS

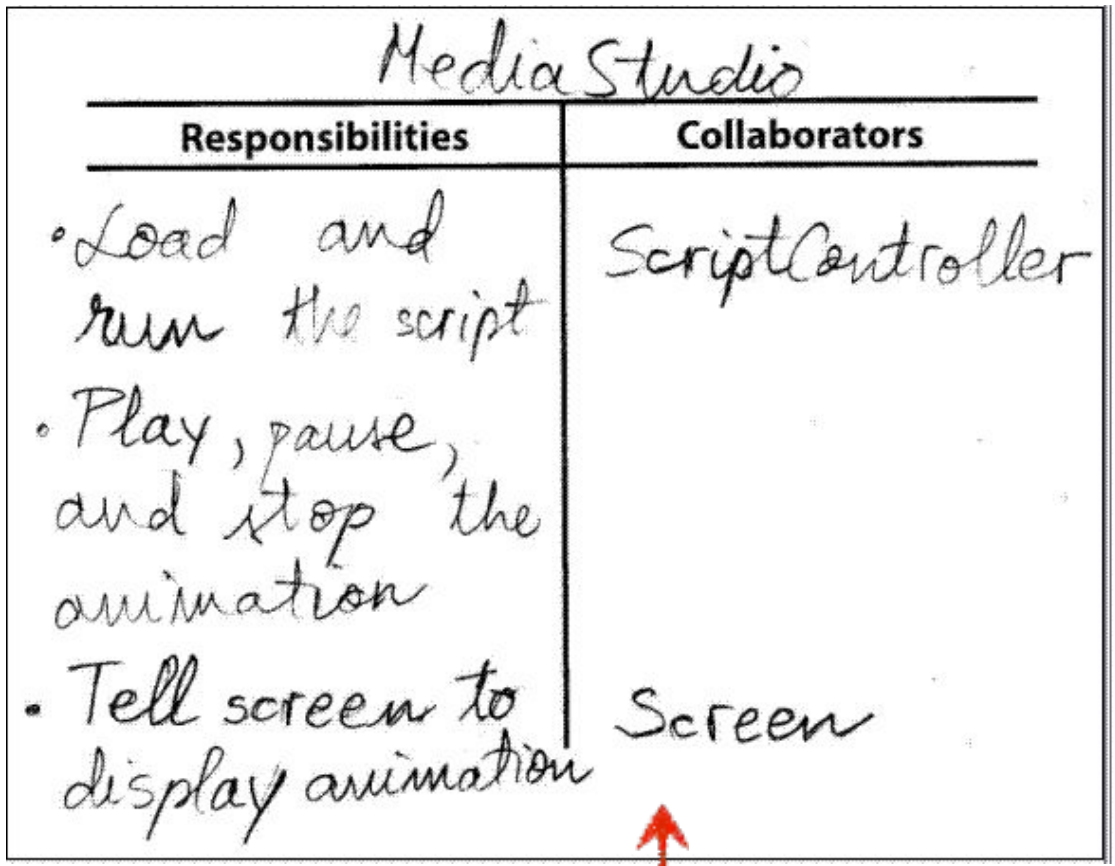
CRAIG LARMAN

CRC Cards & Example

Class-Responsibility-Collaboration Cards


MediaStudio

Responsibilities	Collaborators
• Manage the script	ScriptController: run
• Manage the animation	
• Display animation	Screen


MediaStudio

Responsibilities	Collaborators
• Load and run the script	ScriptController
• Play, pause, and stop the animation	
• Tell screen to display animation	Screen

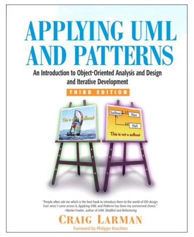
This Example is Incorrect

This Example is Better

Why?

http://coweb.cc.gatech.edu/cs2340/5583

http://c2.com/doc/oopsla89/paper.html



Responsibility Driven Design = Simulation

A popular way of thinking about the design of software objects and also larger-scale components[3] is in terms of *responsibilities*, *roles*, and *collaborations*. This is part of a larger approach called **responsibility-driven design** or RDD [WM02].

In RDD, we think of software objects as having responsibilities—an abstraction of what they do. The UML defines a *responsibility* as "a contract or obligation of a classifier" [OMG03b]. Responsibilities are related to the obligations or behavior of an object in terms of its role. Basically, these responsibilities are of the following two types: *doing* and *knowing*.

Doing responsibilities of an object include:

- doing something itself, such as creating an object or doing a calculation
- initiating action in other objects
- controlling and coordinating activities in other objects

Knowing responsibilities of an object include:

- knowing about private encapsulated data
- knowing about related objects
- knowing about things it can derive or calculate

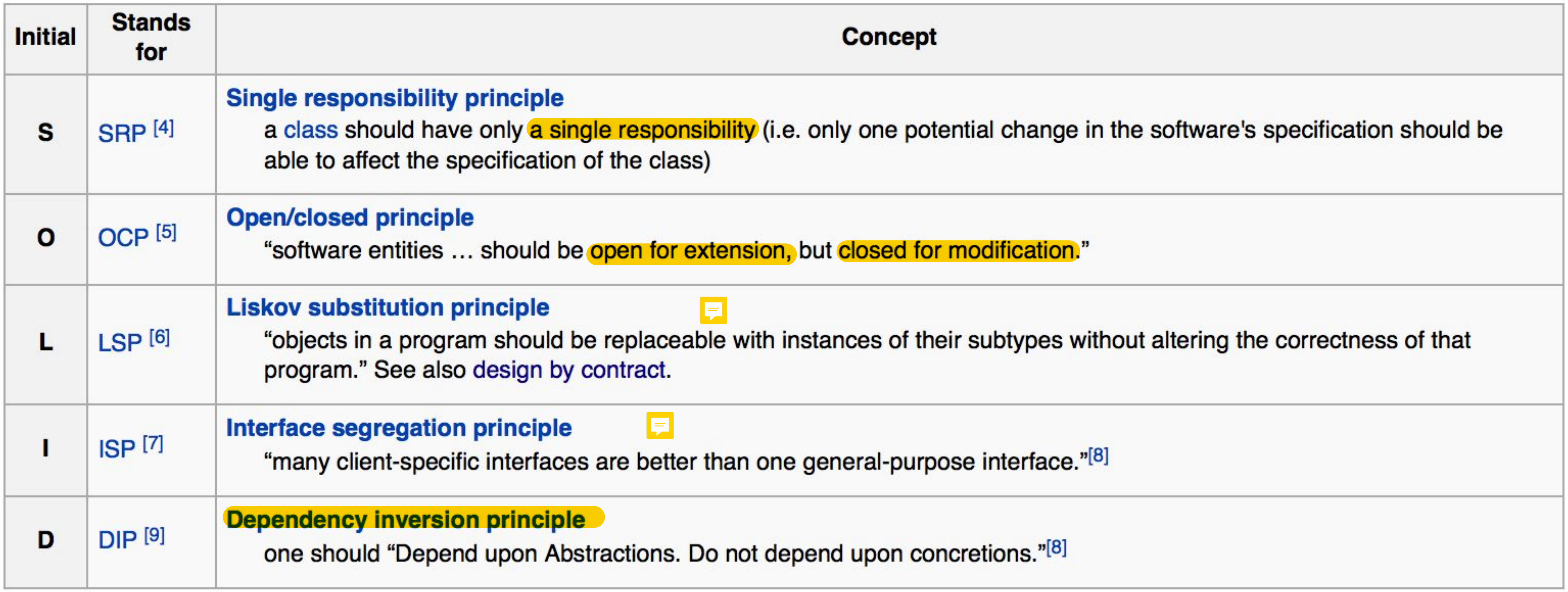
<u>Objects have two types of Responsibilities:</u>

1. What I need to do (i.e. my "Job")
2. What I need to know to get my job done

RDD is a Metaphor

RDD is a general metaphor for thinking about OO software design. Think of software objects as similar to people with responsibilities who collaborate with other people to get work done. RDD leads to viewing an OO design as a *community of collaborating responsible objects.*

SOLID Principles



Initial	Stands for	Concept
S	SRP [4]	**Single responsibility principle** a class should have only a single responsibility (i.e. only one potential change in the software's specification should be able to affect the specification of the class)
O	OCP [5]	**Open/closed principle** “software entities … should be open for extension, but closed for modification.”
L	LSP [6]	**Liskov substitution principle** “objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.” See also design by contract.
I	ISP [7]	**Interface segregation principle** “many client-specific interfaces are better than one general-purpose interface.”[8]
D	DIP [9]	**Dependency inversion principle** one should “Depend upon Abstractions. Do not depend upon concretions.”[8]

wikipedia.o
rg

Single Responsibility Principle (SRP)

- Consider a class that opens a connection to the database, pulls out some table data, and writes the data to a file. This class has multiple reasons to change:
  - adoption of a new database,
  - modified file output format,
  - deciding to use an ORM, etc.
- In terms of the SRP, we'd say that this class is doing too much.

Open/Closed Principle (OCP)

- A class that does what it needs to flawlessly without assuming that people should come in and change it later.
- It's closed for modification, but it can be extended by, for instance, inheriting from it and overriding or extending certain behaviors.

OCP example

- An example of running afoul of the open-closed principle would be to have a switch statement somewhere that you needed to go in and add to every time you wanted to add a menu option to your application.
- Consider a smartphone - core phone functionality *closed for modification open to extension*

OCP

```
public class Rectangle { …..}
public class AreaCalculator {
        public double area(Rectangle[] shapes)
          {
    …….
           }
       }
```

OCP

```
// Enhance AreaCalculator to handle rectangle
and circle

public double area(Object[] shapes) {

……….

// check if rectangle or circle and apply the
correct formula


}
```

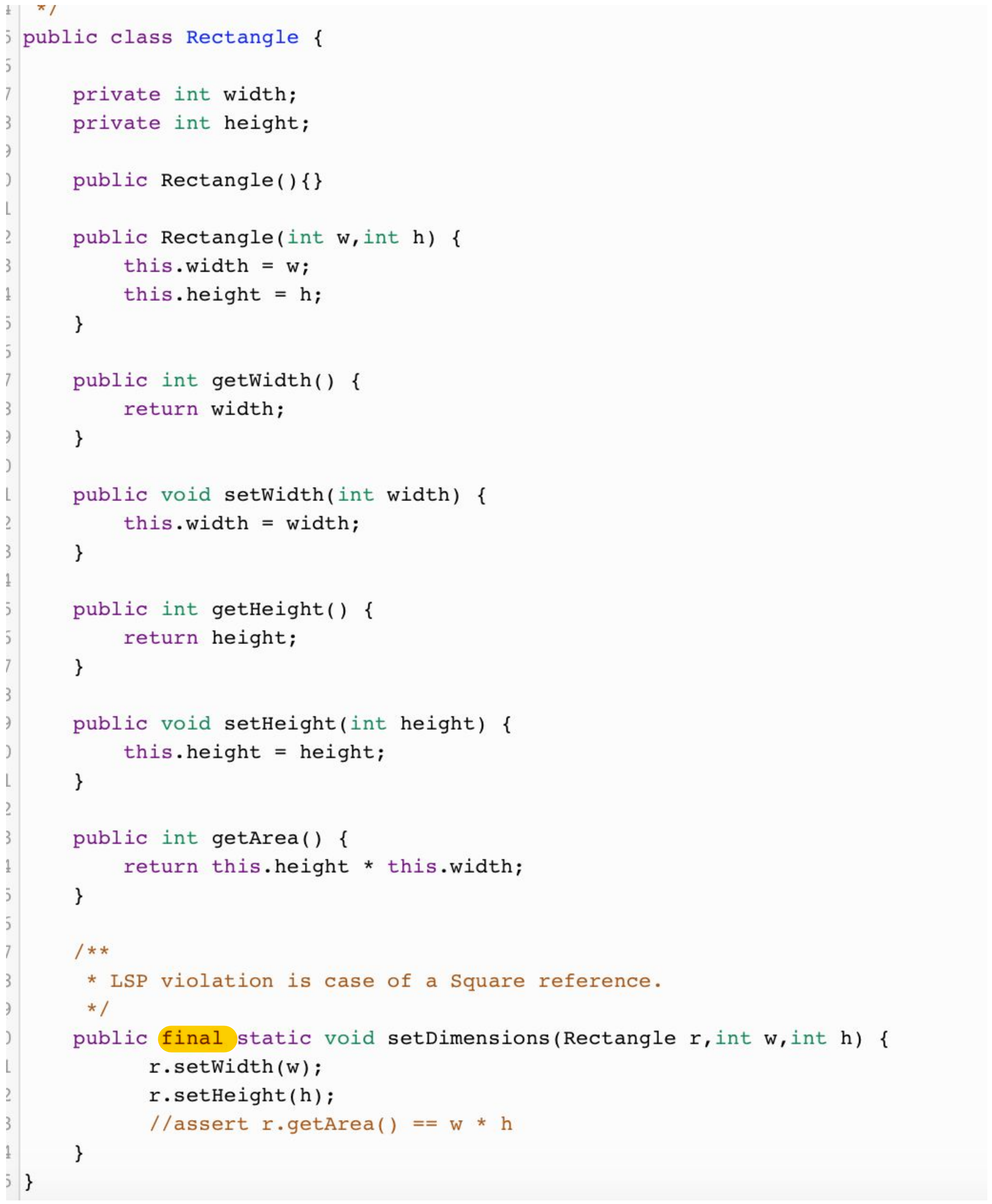
For every new shape **AreaCalculator** needs to be edited – it is open for modification – violation of OCP!

OCP

```
public abstract class Shape {
        public abstract  double area();
}
public class Rectangle exends Shape {
        public double area() {….}  //override
}
public class Circle extends Shape {
        public double area() { …} //override
}
```

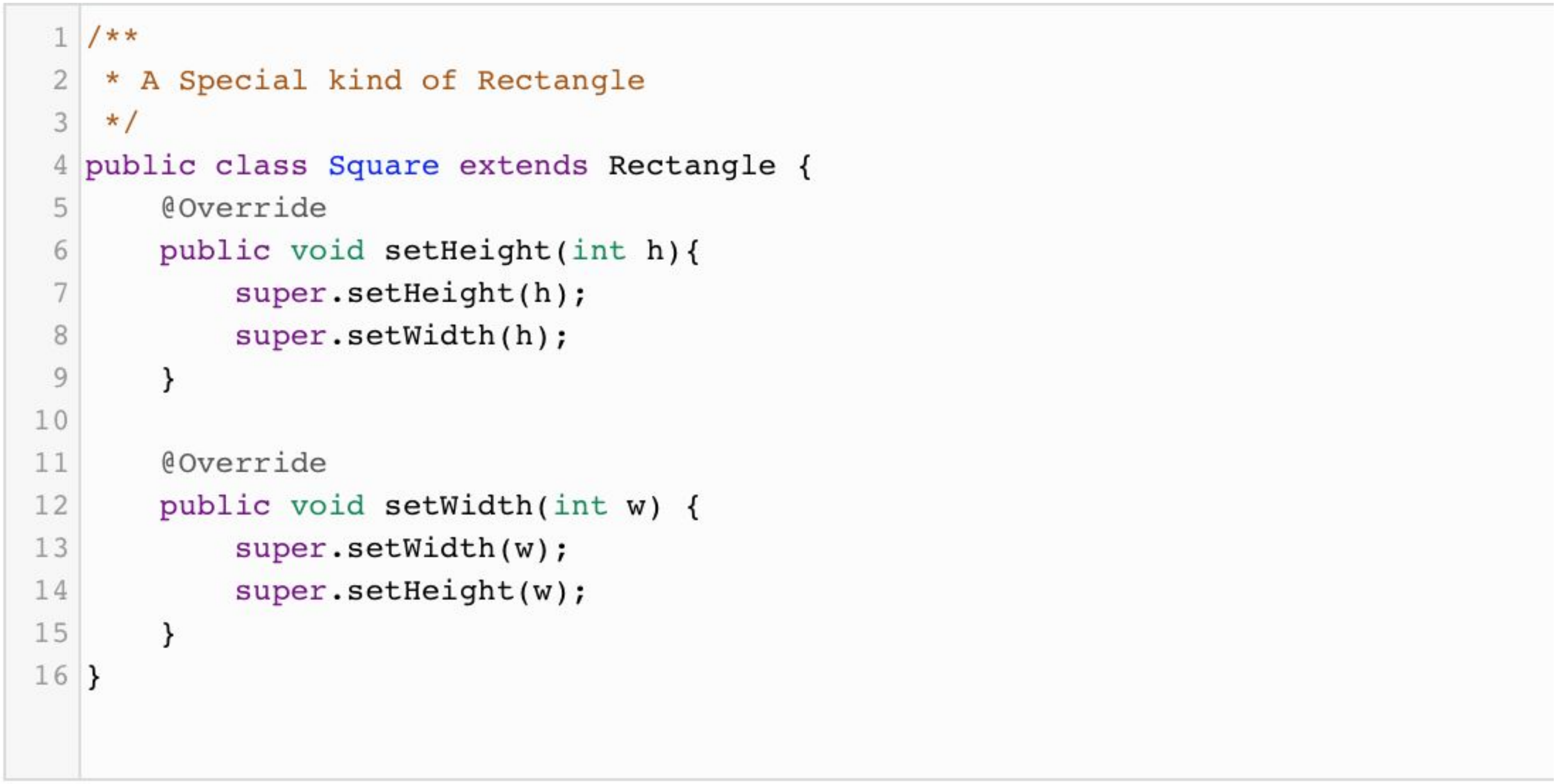
- public double area(Shape[] shapes)
- {
- double total = 0;
- ……. for(Shape s: shapes) {
- total + = s.area();
- }
- return total;

Liskov Substitution Principle



```
 */
public class Rectangle {

    private int width;
    private int height;

    public Rectangle(){}

    public Rectangle(int w,int h) {
        this.width = w;
        this.height = h;
    }

    public int getWidth() {
        return width;
    }

    public void setWidth(int width) {
        this.width = width;
    }

    public int getHeight() {
        return height;
    }

    public void setHeight(int height) {
        this.height = height;
    }

    public int getArea() {
        return this.height * this.width;
    }

    /**
     * LSP violation is case of a Square reference.
     */
    public final static void setDimensions(Rectangle r,int w,int h) {
          r.setWidth(w);
          r.setHeight(h);
          //assert r.getArea() == w * h
    }
}
```

LCP



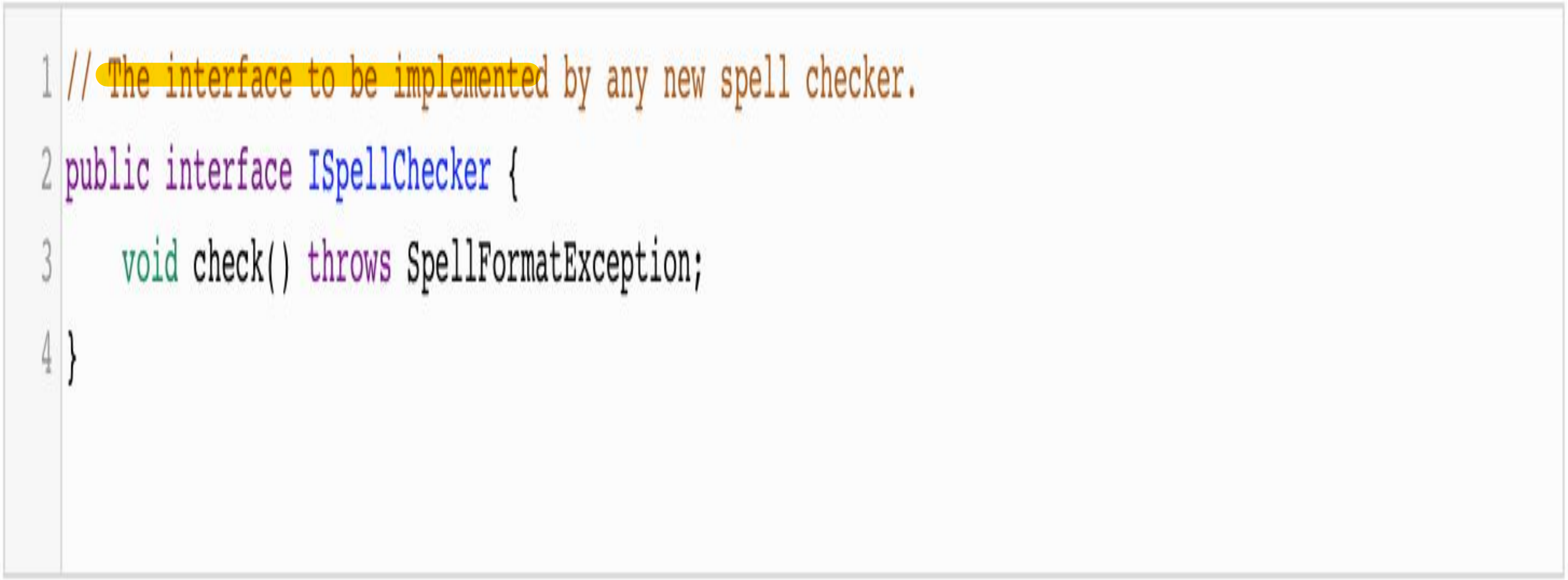
```java
/**
 * A Special kind of Rectangle
 */
public class Square extends Rectangle {
    @Override
    public void setHeight(int h){
        super.setHeight(h);
        super.setWidth(h);
    }

    @Override
    public void setWidth(int w) {
        super.setWidth(w);
        super.setHeight(w);
    }
}
```

Dependency Inversion

```java
public class Emailer{
    private SpellChecker spellChecker;
    public Emailer(SpellChecker sc) {
        this.spellChecker = sc;
    }
    public void checkEmail() {
        this.spellChecker.check();
    }
}
```

And the `Spellchecker` class:

```java
public class SpellChecker {
    public void check() throws SpellFormatException {
    }
}
```



```java
// The interface to be implemented by any new spell checker.
public interface ISpellChecker {
    void check() throws SpellFormatException;
}
```

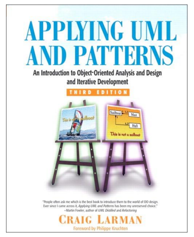
Dependency Inversion

Now, the `Emailer` class accepts only an `ISpellChecker` reference on the constructor. Below, we changed the `Emailer` class to not care/depend on the implementation (concrete class) but rely on the interface (`ISpellChecker`)

```java
public class Emailer{
    private ISpellChecker spellChecker;
    public Emailer(ISpellChecker sc) {
        this.spellChecker = sc;
    }
    public void checkEmail() {
        this.spellChecker.check();
    }
}
```

And we have many implementations for the `ISpellChecker`:

```java
public class SpellChecker implements ISpellChecker {
    @Override
    public void check() throws SpellFormatException {
    }
}

public class GreekSpellChecker implements ISpellChecker {
    @Override
    public void check() throws SpellFormatException {
    }
}
```



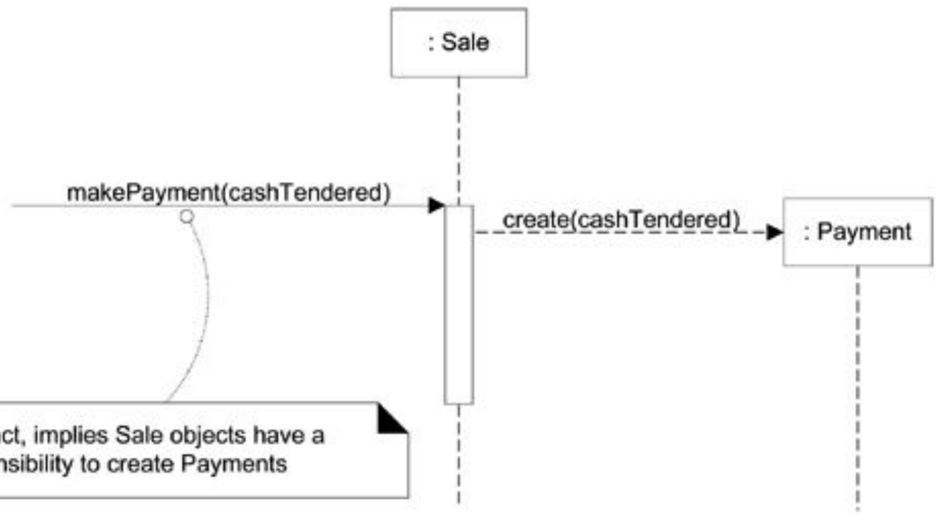
GRASP

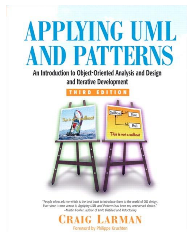
General Responsibility Assignment Software Patterns (or Principles)

Key point: GRASP names and describes some basic principles to assign responsibilities, so it's useful to know—to support RDD.

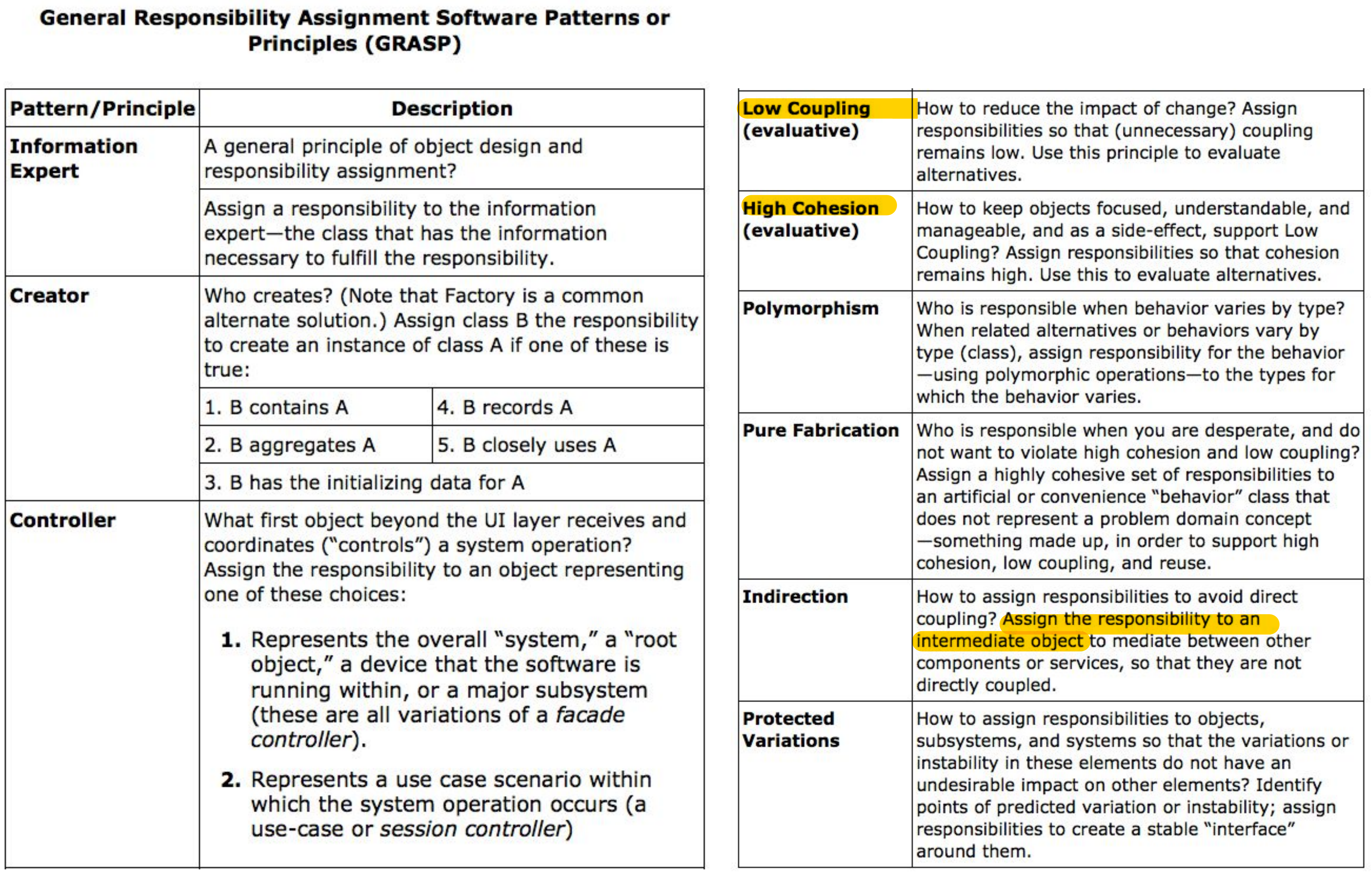
17.5. What's the Connection Between Responsibilities, GRASP, and UML Diagrams?

You can think about assigning responsibilities to objects while coding or while modeling. Within the UML, drawing interaction diagrams becomes the occasion for considering these responsibilities (realized as methods).


: Sale
makePayment(cashTendered)
create(cashTendered)
: Payment
abstract, implies Sale objects have a responsibility to create Payments


APPLYING UML AND PATTERNS
An Introduction to Object-Oriented Analysis and Design and Iterative Development
THIRD EDITION
CRAIG LARMAN

GRASP Patterns Quick Reference



General Responsibility Assignment Software Patterns or Principles (GRASP)

Pattern/Principle	Description	
Information Expert	A general principle of object design and responsibility assignment?	
	Assign a responsibility to the information expert—the class that has the information necessary to fulfill the responsibility.	
Creator	Who creates? (Note that Factory is a common alternate solution.) Assign class B the responsibility to create an instance of class A if one of these is true:	
	1. B contains A	4. B records A
	2. B aggregates A	5. B closely uses A
	3. B has the initializing data for A	
Controller	What first object beyond the UI layer receives and coordinates ("controls") a system operation? Assign the responsibility to an object representing one of these choices: **1.** Represents the overall "system," a "root object," a device that the software is running within, or a major subsystem (these are all variations of a *facade controller*). **2.** Represents a use case scenario within which the system operation occurs (a use-case or *session controller*)	
Low Coupling (evaluative)	How to reduce the impact of change? Assign responsibilities so that (unnecessary) coupling remains low. Use this principle to evaluate alternatives.	
High Cohesion (evaluative)	How to keep objects focused, understandable, and manageable, and as a side-effect, support Low Coupling? Assign responsibilities so that cohesion remains high. Use this to evaluate alternatives.	
Polymorphism	Who is responsible when behavior varies by type? When related alternatives or behaviors vary by type (class), assign responsibility for the behavior—using polymorphic operations—to the types for which the behavior varies.	
Pure Fabrication	Who is responsible when you are desperate, and do not want to violate high cohesion and low coupling? Assign a highly cohesive set of responsibilities to an artificial or convenience "behavior" class that does not represent a problem domain concept—something made up, in order to support high cohesion, low coupling, and reuse.	
Indirection	How to assign responsibilities to avoid direct coupling? Assign the responsibility to an intermediate object to mediate between other components or services, so that they are not directly coupled.	
Protected Variations	How to assign responsibilities to objects, subsystems, and systems so that the variations or instability in these elements do not have an undesirable impact on other elements? Identify points of predicted variation or instability; assign responsibilities to create a stable "interface" around them.	

Name:	**Creator**
Problem:	Who creates an A?
Solution: (this can be viewed as advice)	Assign class B the responsibility to create an instance of class A if one of these is true (the more the better): • B "contains" or compositely aggregates A. • B records A. • B closely uses A. • B has the initializing data for A.

Name:	**Controller**
Problem:	What first object beyond the UI layer receives and coordinates ("controls") a system operation?
Solution: (advice)	Assign the responsibility to an object representing one of these choices: • Represents the overall "system," a "root object," a device that the software is running within, or a major subsystem (these are all variations of a *facade controller*). • Represents a use case scenario within which the system operation occurs (a use case or *session controller*)

Name:	**Low Coupling**
Problem:	How to reduce the impact of change?
Solution: (advice)	Assign responsibilities so that (unnecessary) coupling remains low. Use this principle to evaluate alternatives.

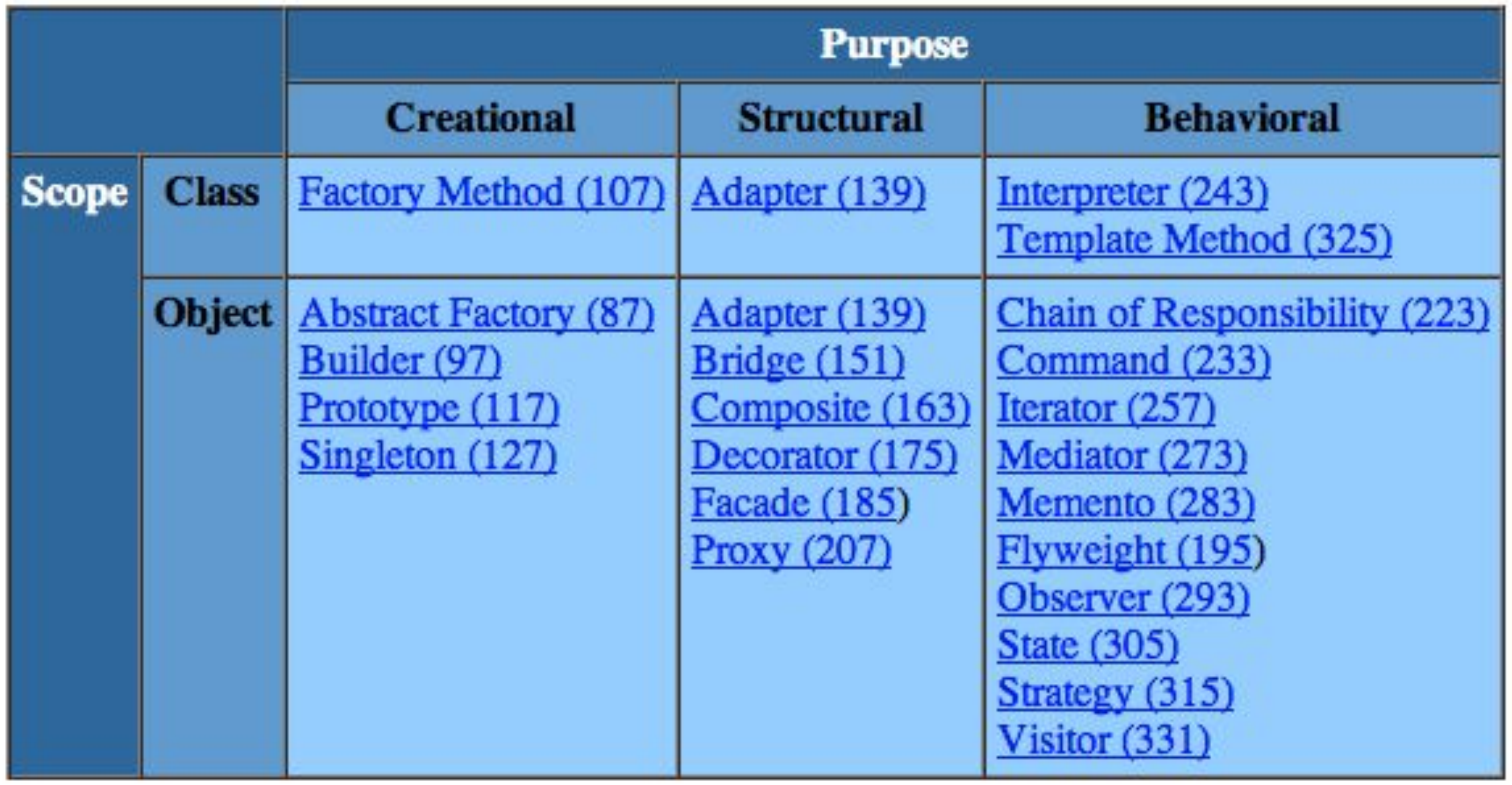
Name:	**High Cohesion**
Problem:	How to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling?
Solution: (advice)	Assign responsibilities so that cohesion remains high. Use this to evaluate alternatives.

Information Expert explains why the *Board* is chosen:

Name:	**Information Expert**
Problem:	What is a basic principle by which to assign responsibilities to objects?
Solution: (advice)	Assign a responsibility to the class that has the information needed to fulfill it.

Scope & Purpose of a Pattern

We classify design patterns by two criteria (Table 1.1). The first criterion, called **purpose**, reflects what a pattern does. Patterns can have either **creational**, **structural**, or **behavioral** purpose. Creational patterns concern the process of object creation. Structural patterns deal with the composition of classes or objects. Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility.



		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method (107)	Adapter (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Flyweight (195) Observer (293) State (305) Strategy (315) Visitor (331)

How to change Class relationships at compile time →

How to change Object relationships at runtime →

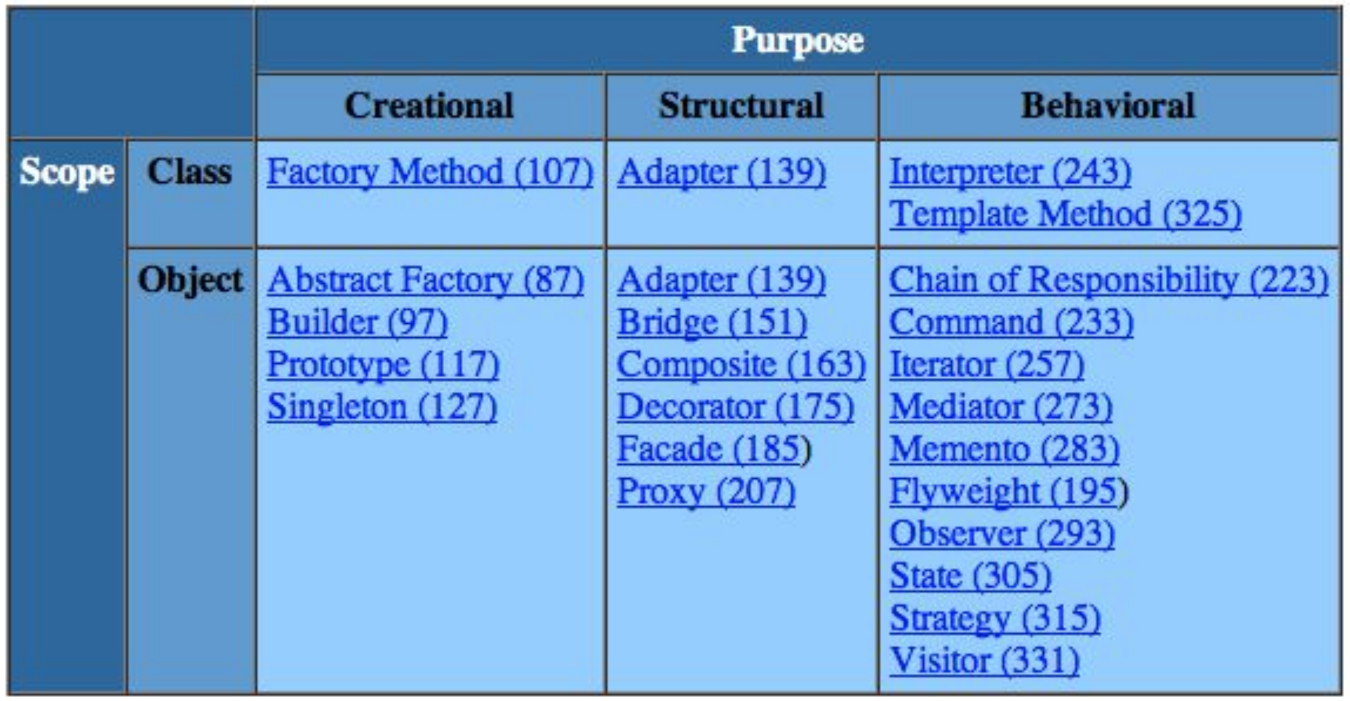
↑ Decouple Constructors

↑ How to Create Large Compositions

↑ How to divide Responsibilities

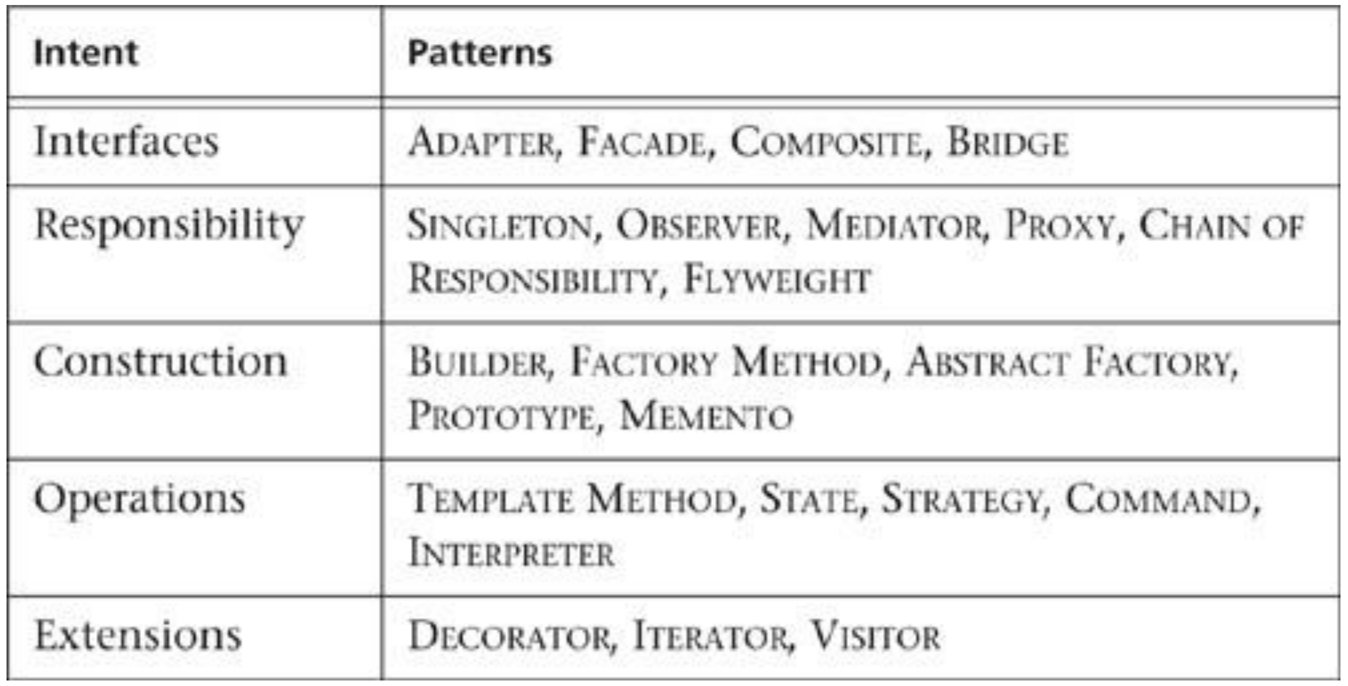
Categorization

Classic Gang of Four Patterns Categorization



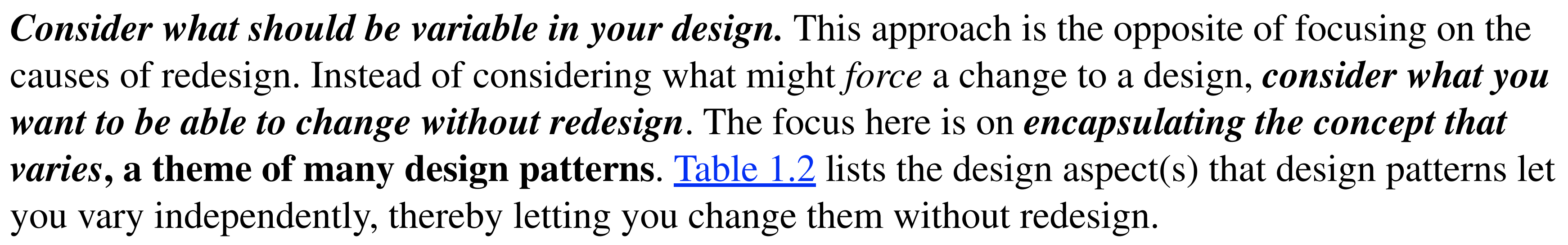
		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method (107)	Adapter (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Flyweight (195) Observer (293) State (305) Strategy (315) Visitor (331)

Design Patterns in Java (Book's) Categorization

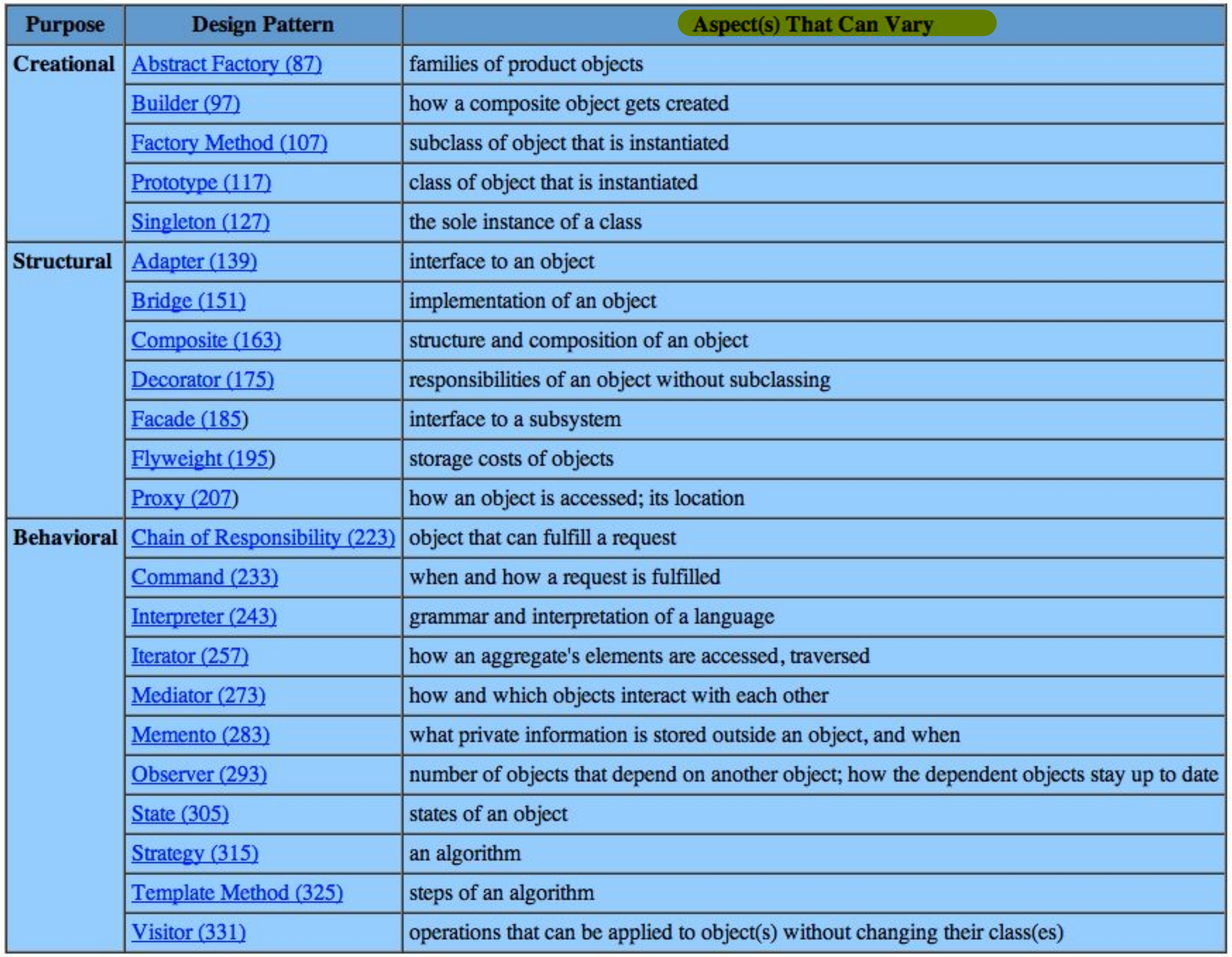


Intent	Patterns
Interfaces	ADAPTER, FACADE, COMPOSITE, BRIDGE
Responsibility	SINGLETON, OBSERVER, MEDIATOR, PROXY, CHAIN OF RESPONSIBILITY, FLYWEIGHT
Construction	BUILDER, FACTORY METHOD, ABSTRACT FACTORY, PROTOTYPE, MEMENTO
Operations	TEMPLATE METHOD, STATE, STRATEGY, COMMAND, INTERPRETER
Extensions	DECORATOR, ITERATOR, VISITOR

Encapsulate the "Concept" that Varies...



Consider what should be variable in your design. This approach is the opposite of focusing on the causes of redesign. Instead of considering what might *force* a change to a design, ***consider what you want to be able to change without redesign***. The focus here is on ***encapsulating the concept that varies*, a theme of many design patterns**. Table 1.2 lists the design aspect(s) that design patterns let you vary independently, thereby letting you change them without redesign.



Purpose	Design Pattern	Aspect(s) That Can Vary
Creational	Abstract Factory (87)	families of product objects
	Builder (97)	how a composite object gets created
	Factory Method (107)	subclass of object that is instantiated
	Prototype (117)	class of object that is instantiated
	Singleton (127)	the sole instance of a class
Structural	Adapter (139)	interface to an object
	Bridge (151)	implementation of an object
	Composite (163)	structure and composition of an object
	Decorator (175)	responsibilities of an object without subclassing
	Facade (185)	interface to a subsystem
	Flyweight (195)	storage costs of objects
	Proxy (207)	how an object is accessed; its location
Behavioral	Chain of Responsibility (223)	object that can fulfill a request
	Command (233)	when and how a request is fulfilled
	Interpreter (243)	grammar and interpretation of a language
	Iterator (257)	how an aggregate's elements are accessed, traversed
	Mediator (273)	how and which objects interact with each other
	Memento (283)	what private information is stored outside an object, and when
	Observer (293)	number of objects that depend on another object; how the dependent objects stay up to date
	State (305)	states of an object
	Strategy (315)	an algorithm
	Template Method (325)	steps of an algorithm
	Visitor (331)	operations that can be applied to object(s) without changing their class(es)

Goal:

"Adapt to changes without a re-design"

Key Ideas behind Design Patterns

- They are an encoding of OO folklore!
- They are at a mid-range level of abstraction
  - That is, a the level of "Communicating Collaborating Objects".
  - Not low-level (i.e. Hash Tables).
  - Not high-level (i.e. Domain Specific Patterns)
- They are not "recipes"!
  - Thus, do not take the pattern as a prescription for implementation
- GoF Patterns assume features similar to Smalltalk/C++

Study Guide

1. Know the GoF Pattern Catalog, Know each Pattern by Name
2. Know their Purpose Category (Creational, Structural, Behavioral)
3. Know what “aspect of variation” they encapsulate
4. Know the short description of their problem/solution statement

Through the Labs and your Project work:

Know the implementation (in Java code) for the following patterns:

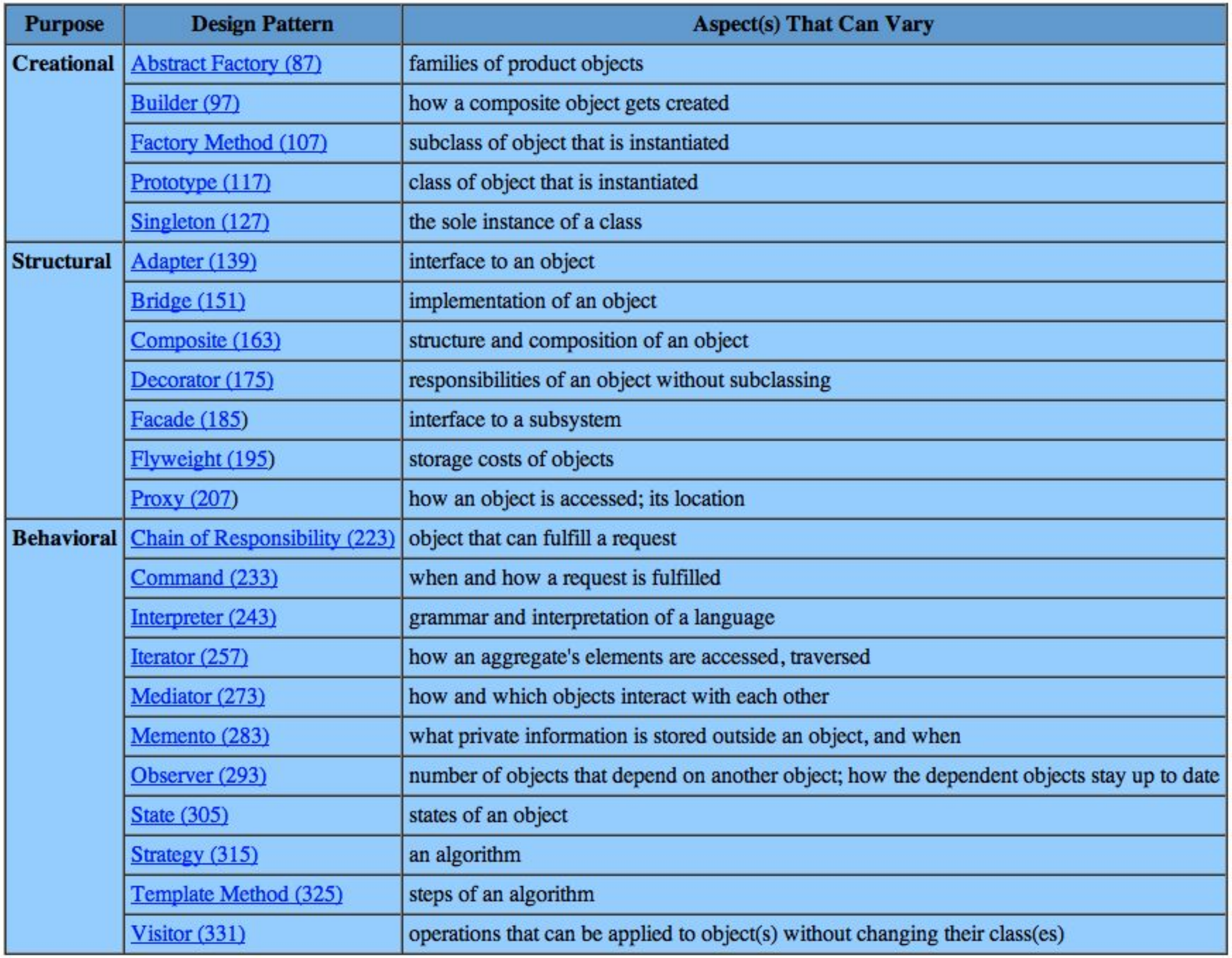
1. Observer
2. Composite
3. Strategy
4. Factory Method
5. Decorator
6. Command
7. Chain of Responsibility
8. Adapter
9. Singleton
10. State
11. Proxy
12. Iterator

Also: Know the how the patterns above apply to the MVC architecture

Recommended Additional Resources:

Quick Reference

Design Pattern Catalog



Purpose	Design Pattern	Aspect(s) That Can Vary
Creational	Abstract Factory (87)	families of product objects
	Builder (97)	how a composite object gets created
	Factory Method (107)	subclass of object that is instantiated
	Prototype (117)	class of object that is instantiated
	Singleton (127)	the sole instance of a class
Structural	Adapter (139)	interface to an object
	Bridge (151)	implementation of an object
	Composite (163)	structure and composition of an object
	Decorator (175)	responsibilities of an object without subclassing
	Facade (185)	interface to a subsystem
	Flyweight (195)	storage costs of objects
	Proxy (207)	how an object is accessed; its location
Behavioral	Chain of Responsibility (223)	object that can fulfill a request
	Command (233)	when and how a request is fulfilled
	Interpreter (243)	grammar and interpretation of a language
	Iterator (257)	how an aggregate's elements are accessed, traversed
	Mediator (273)	how and which objects interact with each other
	Memento (283)	what private information is stored outside an object, and when
	Observer (293)	number of objects that depend on another object; how the dependent objects stay up to date
	State (305)	states of an object
	Strategy (315)	an algorithm
	Template Method (325)	steps of an algorithm
	Visitor (331)	operations that can be applied to object(s) without changing their class(es)

▾ Creational Patterns

Abstract Factory (87)
Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Builder (97)
Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Factory Method (107)
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 (117)
Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Singleton (127)
Ensure a class only has one instance, and provide a global point of access to it.

▾ Structural Patterns

Adapter (139)
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 (151)
Decouple an abstraction from its implementation so that the two can vary independently.

Composite (163)
Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Decorator (175)
Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Facade (185)
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 (195)
Use sharing to support large numbers of fine-grained objects efficiently.

Proxy (207)
Provide a surrogate or placeholder for another object to control access to it.

▾ Behavioral Patterns

Chain of Responsibility (223)

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 (233)

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

Interpreter (243)

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

Iterator (257)

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

Mediator (273)

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.

Memento (283)

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

Observer (293)

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

State (305)

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

Strategy (315)

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

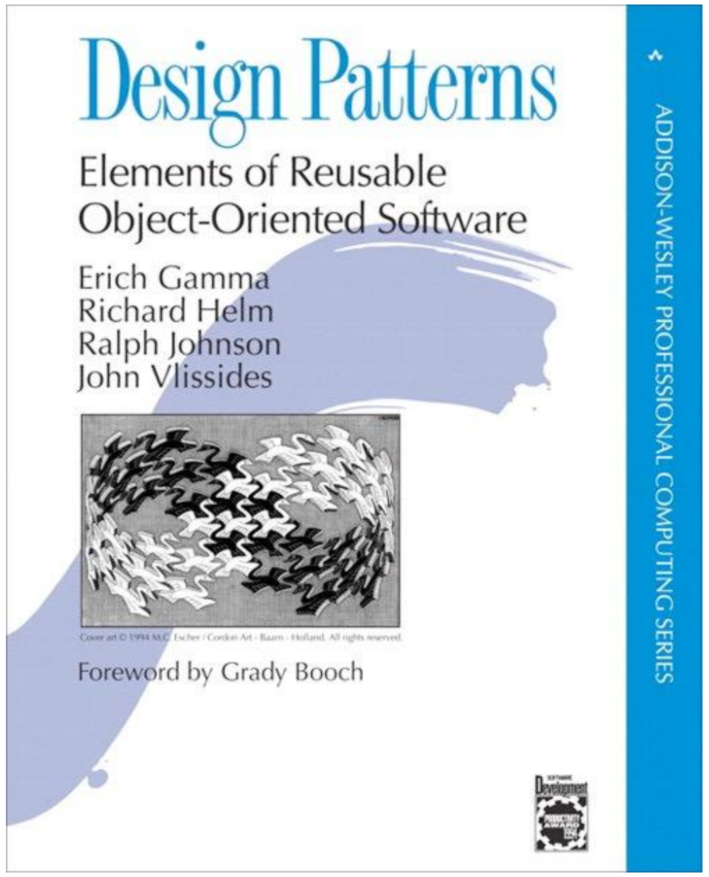
Template Method (325)

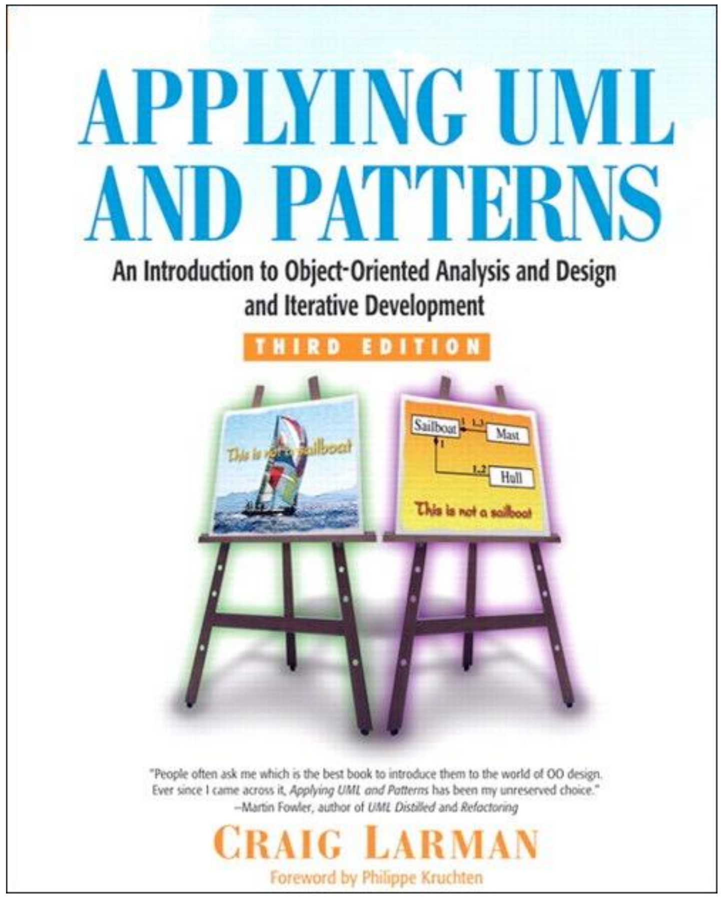
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

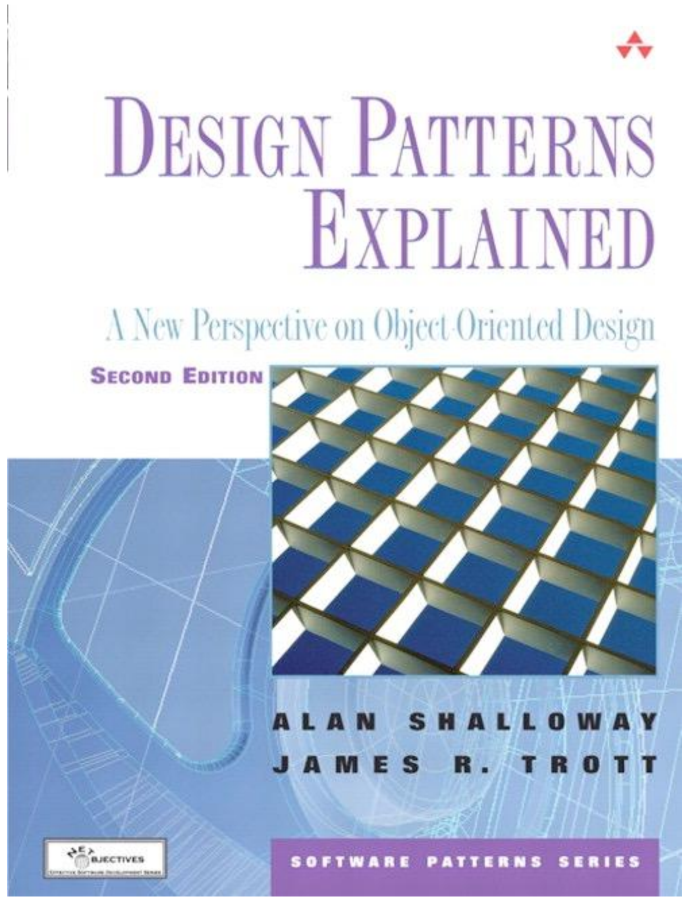
Visitor (331)

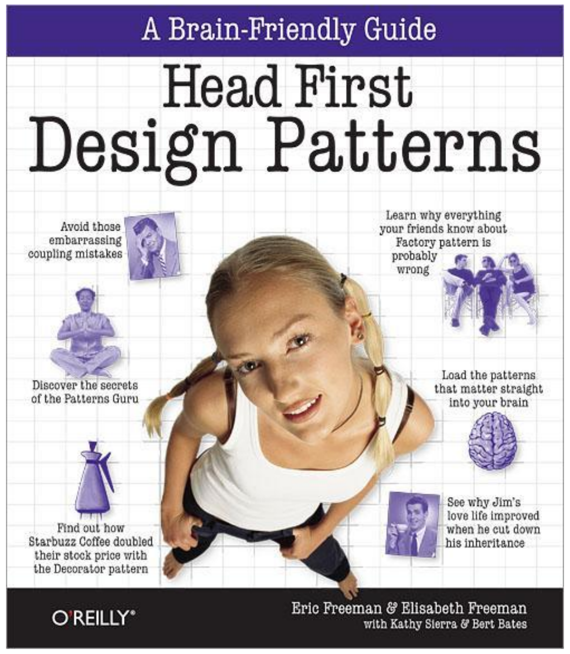
Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

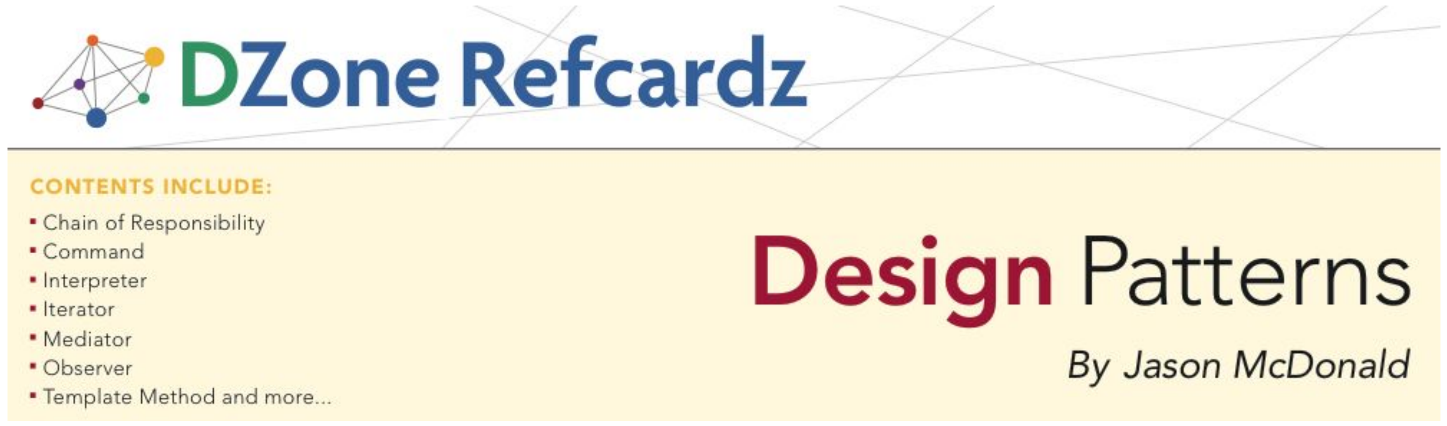
Resources for this Tutorial


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


APPLYING UML AND PATTERNS
An Introduction to Object-Oriented Analysis and Design and Iterative Development
THIRD EDITION
CRAIG LARMAN
Foreword by Philippe Kruchten


DESIGN PATTERNS EXPLAINED
A New Perspective on Object Oriented Design
SECOND EDITION
ALAN SHALLOWAY
JAMES R. TROTT
SOFTWARE PATTERNS SERIES


A Brain-Friendly Guide
Head First Design Patterns
O'REILLY
Eric Freeman & Elisabeth Freeman
with Kathy Sierra & Bert Bates


DZone Refcardz

CONTENTS INCLUDE:
- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Observer
- Template Method and more...

Design Patterns

By Jason McDonald