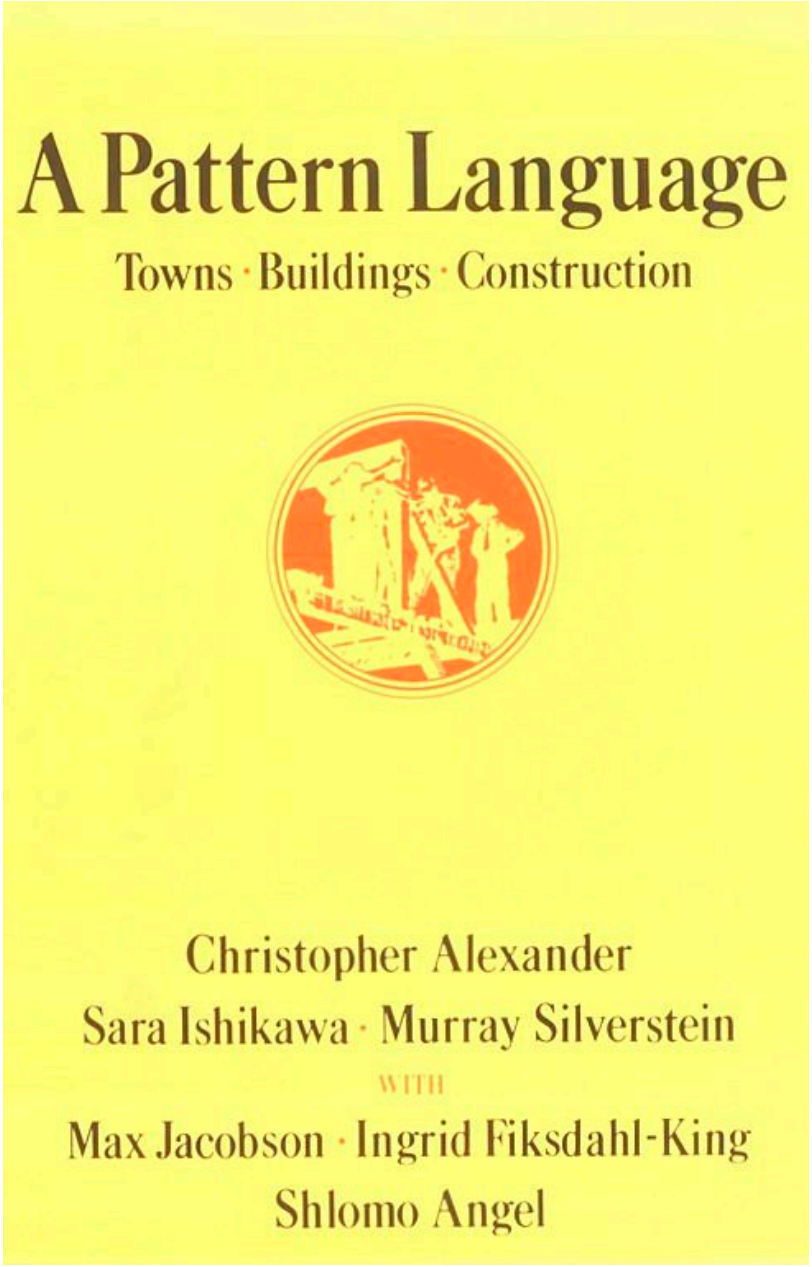
SEG 2105 - LECTURE 06

Design Patterns

A design pattern is the re-usable form of a solution to a design problem. The idea was introduced by the architect Christopher Alexander and has been adapted for various other disciplines, notably software engineering.

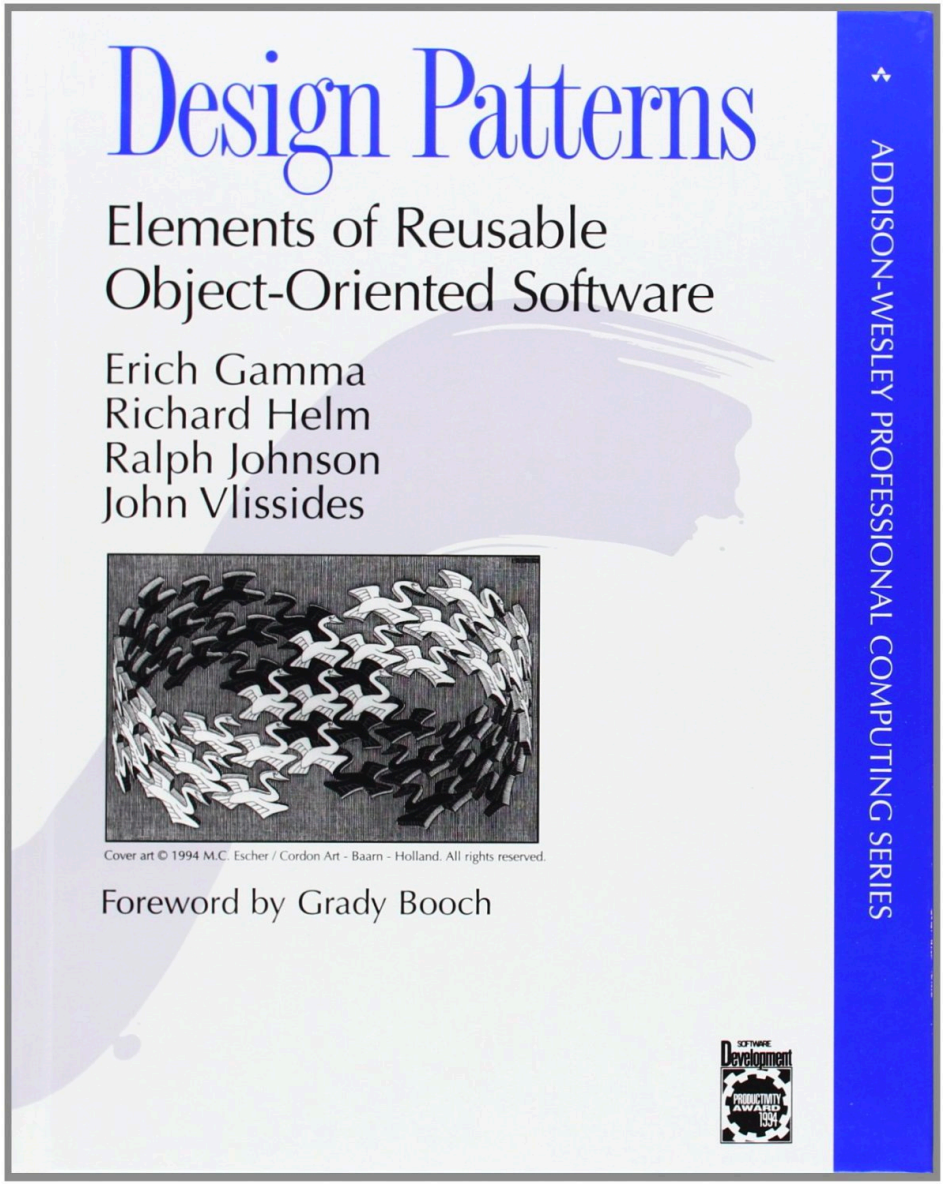
https://en.wikipedia.org/wiki/Design_pattern


A Pattern Language
Towns · Buildings · Construction
Christopher Alexander
Sara Ishikawa · Murray Silverstein
WITH
Max Jacobson · Ingrid Fiksdahl-King
Shlomo Angel

[…] each pattern represents our **current best guess** as to what arrangement of the physical environment will work to **solve the problem presented**. The empirical questions center on the problem—**does it occur** and is it felt **in the way we describe** it?—and the solution—does the arrangement we propose solve the problem? And the asterisks represent our degree of faith in these hypotheses.

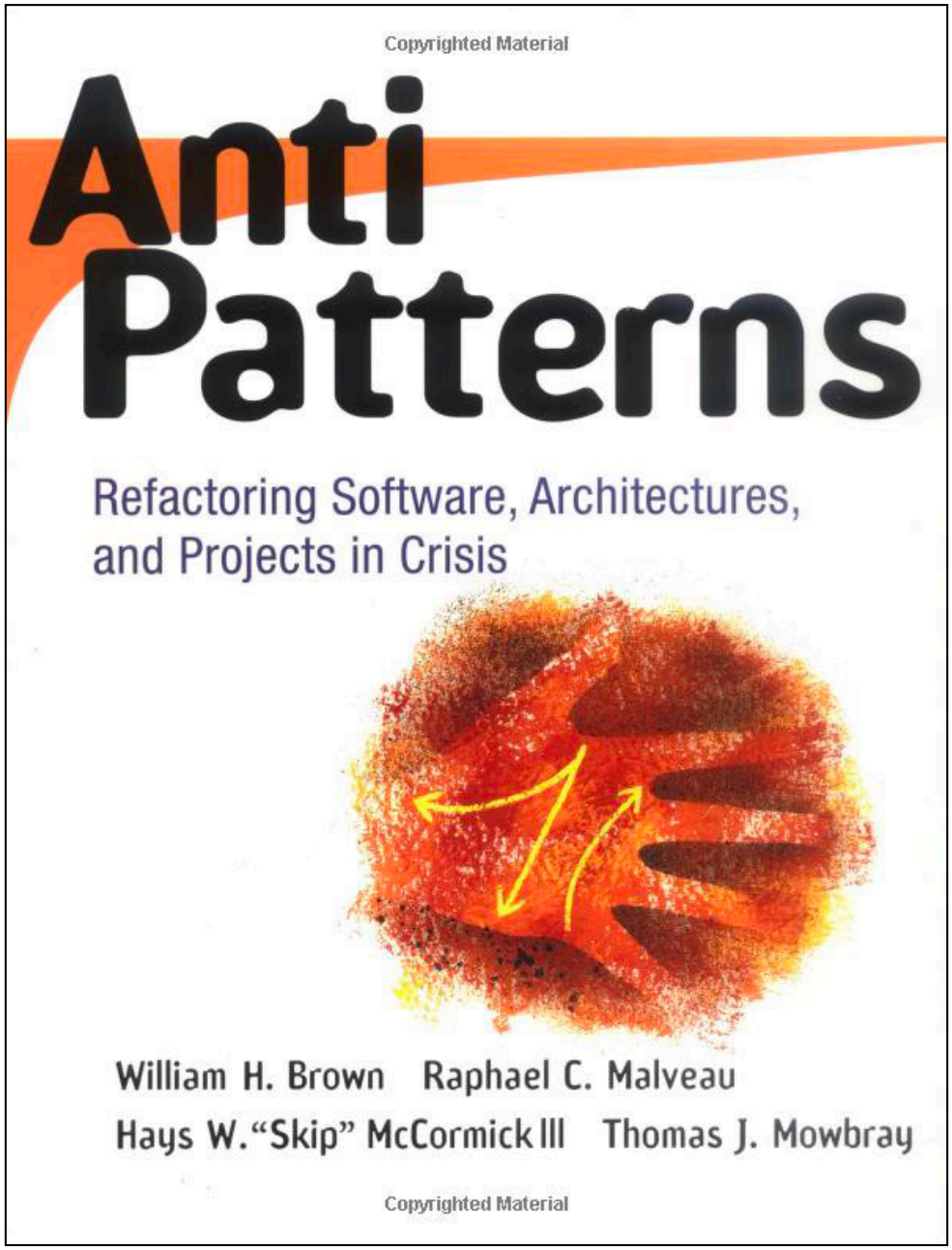
https://en.wikipedia.org/wiki/A_Pattern_Language

http://synapse9.com/signals/2015/05/06/whats-pattern-language/


Design Patterns
Elements of Reusable
Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

A "**design pattern**" is generally understood as both a "**simplifying ideal**" for solving a problem that **commonly reoccurs** in various situations. It is then also something to refer to like a reference book, of workable design variations, linked to all the places the recurrent problem arises. So, design patterns are **not exactly 'instructions' but 'guides'**.

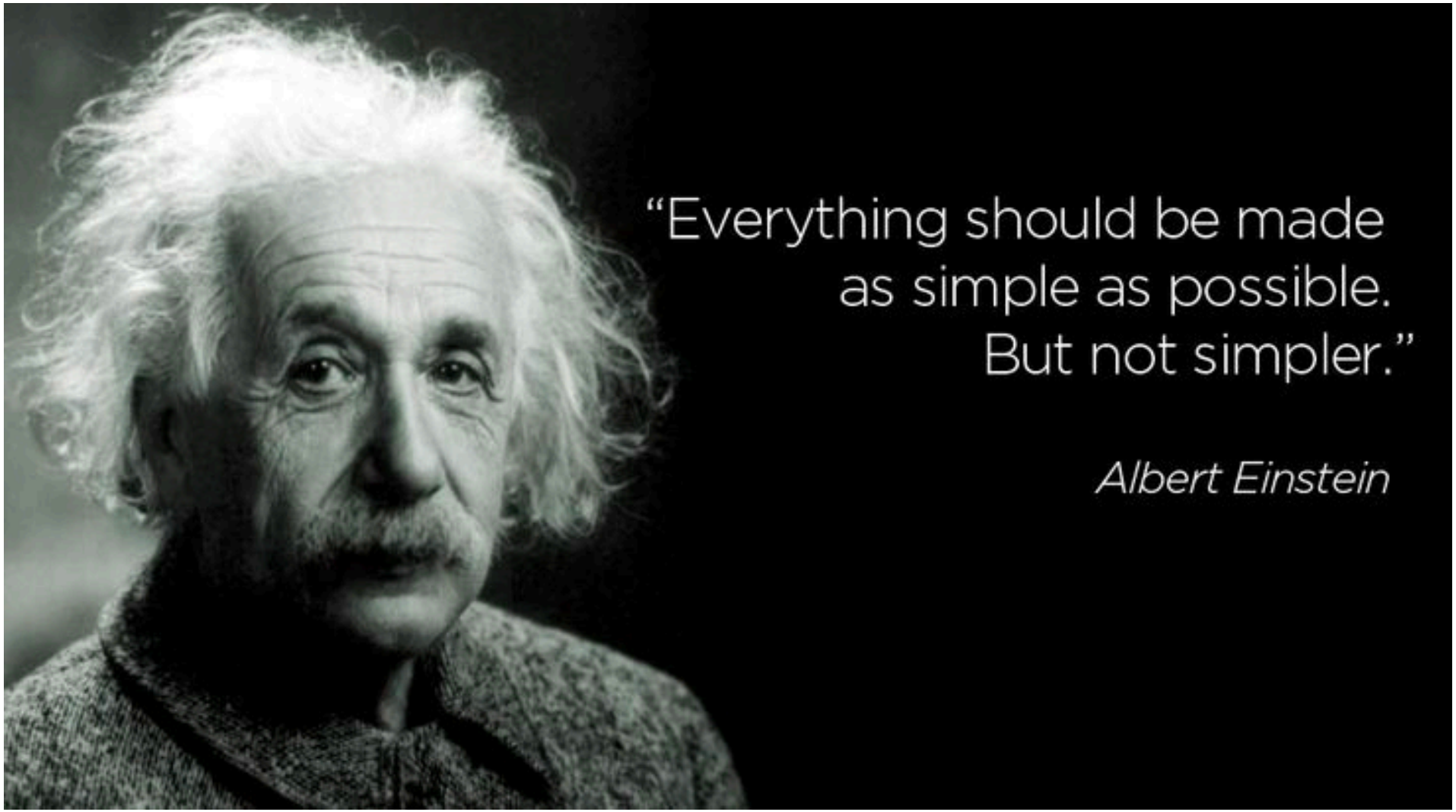
https://www.synapse9.com/drafts/2015_PLoP-draft.pdf


Copyrighted Material
Anti
Patterns
Refactoring Software, Architectures,
and Projects in Crisis
William H. Brown Raphael C. Malveau
Hays W. "Skip" McCormick III Thomas J. Mowbray
Copyrighted Material

Recurring aspects of design

- Outline of a re-usable solution
- To a general problem
- Encountered in a particular context

A good pattern should



A good pattern should...

Studying patterns is an effective way . . .

Pattern Description

- Context:
- Problem:
- Forces:
- Solution:
- Antipatterns(Optional)
- Related patterns: (Optional)
- References

Context:
The general situation in which the pattern applies

Problem:
A short sentence or two raising the main difficulty.

Forces:
The issues or concerns to consider when solving the problem

Solution:
The recommended way to solve the problem in the given context. ‘to balance the forces’

Antipatterns: (Optional)
Solutions that are inferior or do not work in this context.

Related patterns: (Optional)
Patterns that are similar to this pattern.

References:
Who developed or inspired the pattern.

ABSTRACTION-OCCURRENCE

PATTERN

Context

- Often in a domain model you find a set of related objects (occurrences).
  - The members of such a set share common information
  - but also differ from each other in important ways.

CAPITOL CORRIDOR®
Train Schedule
AUGUST 22, 2016

CAPITOL CORRIDOR
JOINT POWERS AUTHORITY

WEEKDAYS: Westbound
Auburn-Sacramento-Emeryville/SF-Oakland-San Jose

CODE	STATION	TRAIN:	521	523	525	527	709	529	531	711	535	537	541	713	543	545	715	547	549	717	551	553
							SAN JOAQUIN			SAN JOAQUIN		#3537		SAN JOAQUIN	#3543		SAN JOAQUIN		#3549	SAN JOAQUIN		
COX	Colfax	Depart										10:00a			1:25p				4:45p			
ARN	Auburn/Conheim	Depart						6:30a				10:30a			1:55p				5:10p			
RLN	Rocklin	Depart		#3623	#3625			6:53a				10:45a			2:10p				5:15p			
RSV	Roseville	Depart		4:35a	5:25a			7:03a				11:00a			2:25p				5:40p			
SAC	Sacramento	Arrive		5:15a	6:05a			7:32a				11:55a			3:20p				6:15p			
		Depart	4:30a	5:30a	6:20a	7:00a	Through Trains from Bakersfield	7:40a	8:55a	Through Trains from Bakersfield	10:10a	12:10p	2:10p	Through Trains from Bakersfield	3:35p	4:40p	Through Trains from Bakersfield	5:40p	6:55p	Through Trains from Bakersfield	9:10p	10:30p
DAV	Davis	Depart	4:45a	5:45a	6:35a	7:15a		7:55a	9:10a		10:25a	12:25p	2:25p		3:50p	4:55p		5:55p	7:11p		9:25p	10:45p
SUI	Suisun/Fairfield	Depart	5:09a	6:09a	6:59a	7:39a		8:19a	9:34a		10:49a	12:49p	2:49p		4:14p	5:19p		6:19p	7:34p		9:49p	11:09p
MTZ	Martinez	Depart	5:29a	6:29a	7:19a	7:59a	For current schedule, visit amtrak.com	8:39a	9:54a	For current schedule, visit amtrak.com	11:09a	1:09p	3:09p	For current schedule, visit amtrak.com	4:34p	5:39p	For current schedule, visit amtrak.com	6:39p	7:54p	For current schedule, visit amtrak.com	10:09p	11:29p
RIC	Richmond–BART	Depart	5:55a	6:55a	7:45a	8:25a		9:05a	10:20a		11:35a	1:35p	3:35p		5:00p	6:05p		7:05p	8:20p		10:35p	11:55p
BKY	Berkeley	Depart	6:03a	7:03a	7:53a	8:33a		9:13a	10:28a		11:43a	1:43p	3:43p		5:08p	6:13p		7:13p	8:28p		10:43p	12:03a
EMY	Emeryville	Depart	6:10a	7:10a	8:00a	8:40a		9:20a	10:35a		11:50a	1:50p	3:48p		5:15p	6:18p		7:20p	8:35p		10:50p	12:10a
OKJ	Oakland Jack London	Arrive	6:21a	7:21a	8:11a	8:51a		9:38a	⊕10:51a		12:01p	2:01p	3:50p		5:26p	6:20p		7:31p	⊕ 8:51p		11:08p	12:28a
		Depart	6:23a	7:23a	8:13a	8:53a					12:06p	2:03p	4:08p		5:28p	6:38p		7:33p				
OAC	Oakland Coliseum–BART	Depart	6:32a	7:32a	8:22a	9:02a			11:00a		12:15p	2:12p			5:40p			7:44p	⊕ 9:00p			
HAY	Hayward	Depart	6:43a	7:43a	8:33a	9:13a						2:23p			5:52p			7:59p	No Stop			
FMT	Fremont/Centerville	Depart	6:59a	7:59a	8:49a	9:29a						2:39p			6:09p			8:17p	No Stop			
GAC	Santa Clara/Great America	Depart	7:16a	8:16a	9:06a	9:46a						2:56p			6:27p			8:34p	No Stop			
SCC	Santa Clara/University	Depart	7:24a	8:24a	9:14a	9:54a						3:04p			6:35p			8:42p	No Stop			
SJC	San Jose	Arrive	7:38a	8:38a	9:28a	10:13a						3:18p			6:48p			8:58p	11:55p	#4068		

WEEKDAYS: Westbound
Emeryville to San Francisco Bus Connection

	CONNECTING TRAIN	521	523	525	527	709	529	531	711	535	537	541	713	543	545	715	547	549	717	551	553
Ar	Emeryville-Amtrak	6:08a	7:08a	7:58a	8:38a	10:09a	9:18a	10:33a	1:58p	11:48a	1:48p	3:48p	4:26p	5:13p	6:18p	7:19p	7:18p	8:33p	⊕9:46p	10:48p	12:08a
	Connecting Bus #	6621/3321	6623/3323	6625/3325	6627/3327	6609/3309	6629/3329	6631	6611	6635	6637	6641	6613	6643	6645/3345	6647	6647	6649	6617	6651	6653
Lv	Emeryville-Amtrak	6:10a	7:10a	8:00a	8:40a	10:10a	9:20a	10:35a	2:00p	11:50a	1:50p	3:50p	4:30p	5:15p	6:20p	7:20p	7:20p	8:35p	9:50p	10:50p	12:10a
	SF Transbay Terminal	⊕6:40a	⊕7:50a	⊕8:35a	⊕9:15a	10:40a	⊕9:50a	⊕11:05a	⊕2:30p	⊕12:20p	⊕2:20p	⊕4:20p	⊕5:00p	⊕5:45p	⊕6:50p	⊕7:50p	⊕7:50p	⊕9:00p	⊕10:15p	⊕11:15p	12:35a
	SF Financial-Hyatt	⊕6:40a	⊕7:50a	⊕8:35a	⊕9:15a	10:50a	⊕10:00a	⊕11:15a	No Stop	No Stop	No Stop	No Stop	No Stop	No Stop	⊕6:50p	No Stop	⊕8:00p	No Stop	No Stop	No Stop	No Stop
	SF Pier 39	No Stop	No Stop	No Stop	No Stop	11:05a	⊕10:15a	⊕11:25a	No Stop	No Stop	No Stop	No Stop	No Stop	No Stop	No Stop	No Stop	No Stop	No Stop	No Stop	No Stop	No Stop
	SF Shop Center, 835 Market	No Stop	No Stop	No Stop	No Stop	10:50a	No Stop	No Stop	No Stop	No Stop	No Stop	No Stop	No Stop	No Stop	No Stop	No Stop	No Stop	No Stop	No Stop	No Stop	No Stop	No Stop

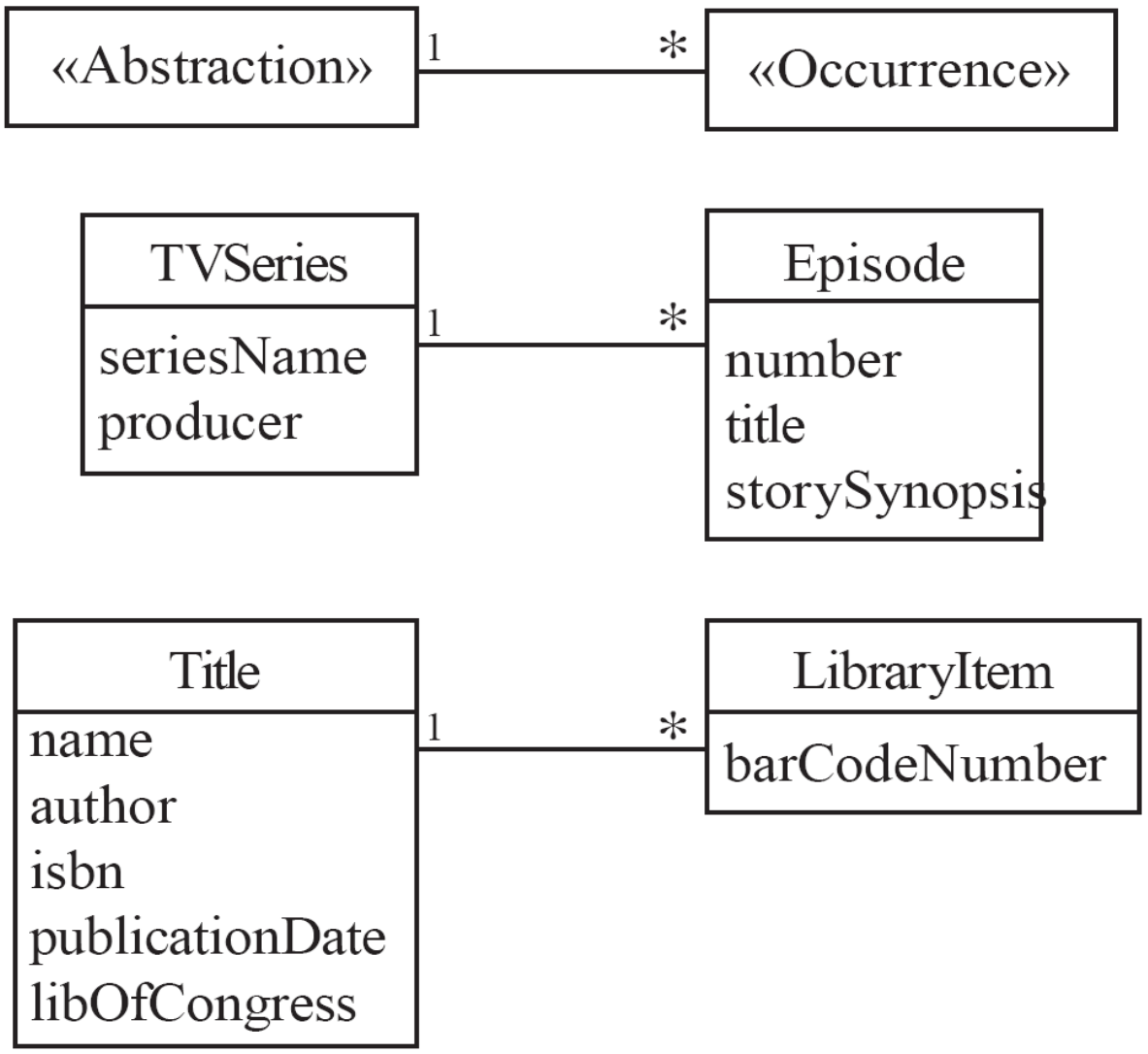
PROBLEM

- What is the best way to represent such sets of occurrences in a class diagram?

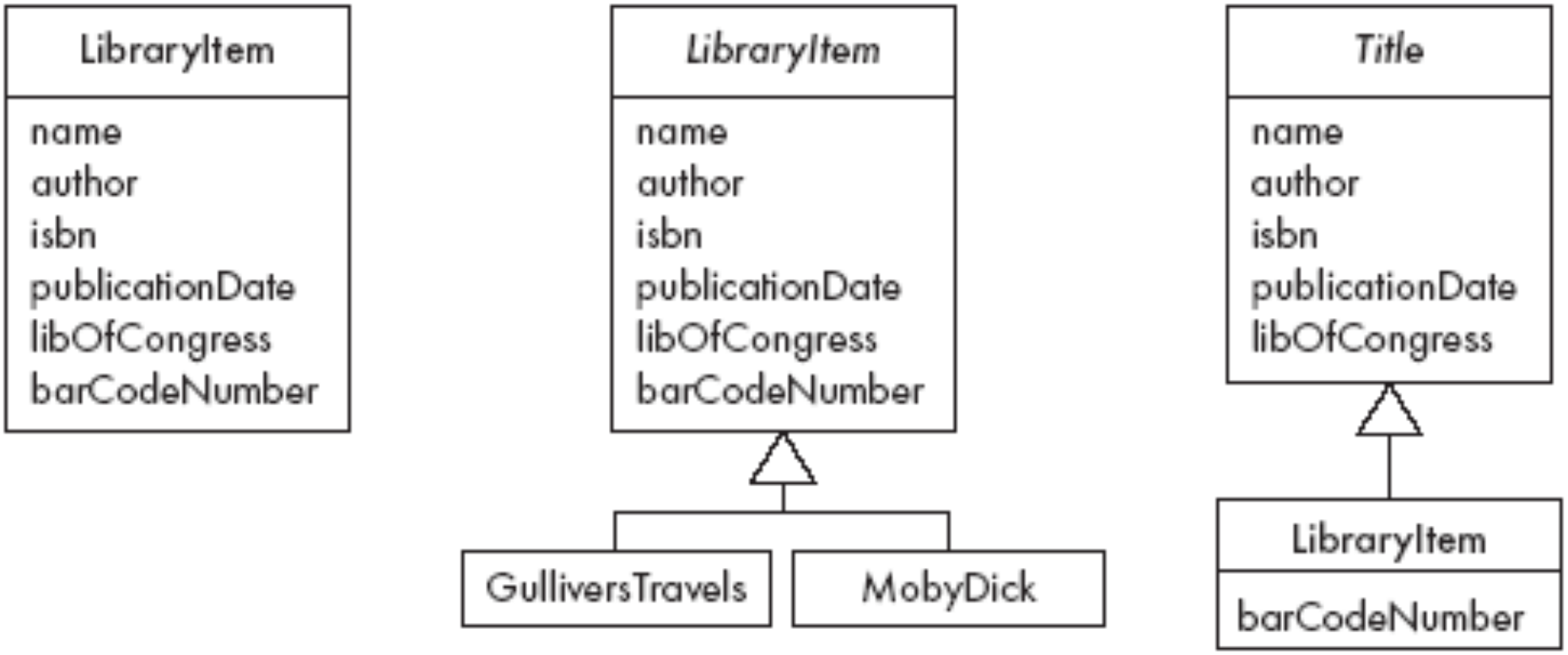
FORCES

- You want to represent the members of each set of occurrences without duplicating the common information

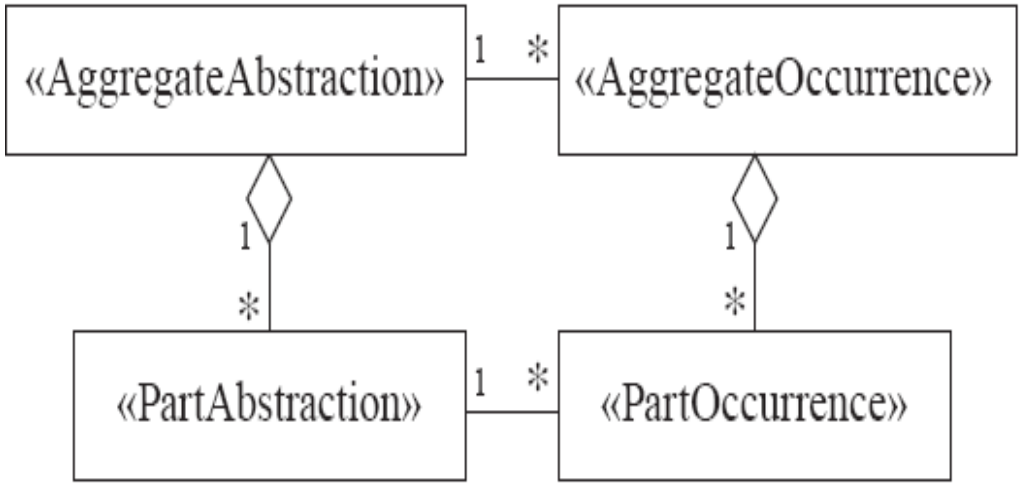
SOLUTION

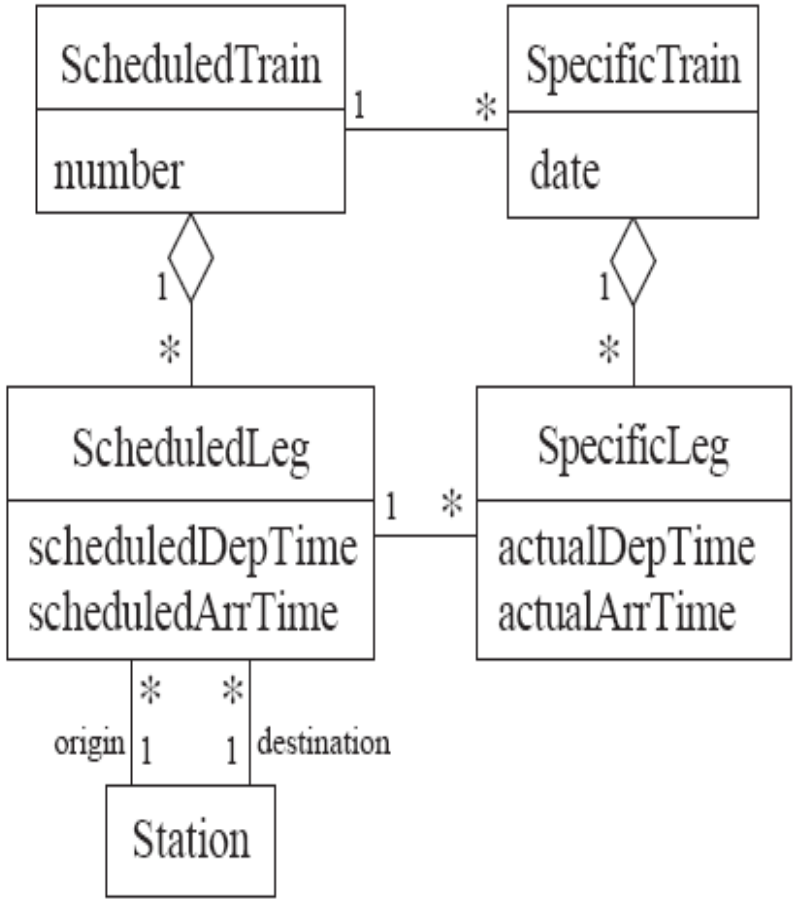

«Abstraction»
1
*
«Occurrence»
TVSeries
seriesName
producer
1
*
Episode
number
title
storySynopsis
Title
name
author
isbn
publicationDate
libOfCongress
1
*
LibraryItem
barCodeNumber

ANTIPATTERNS



SQUARE VARIANT



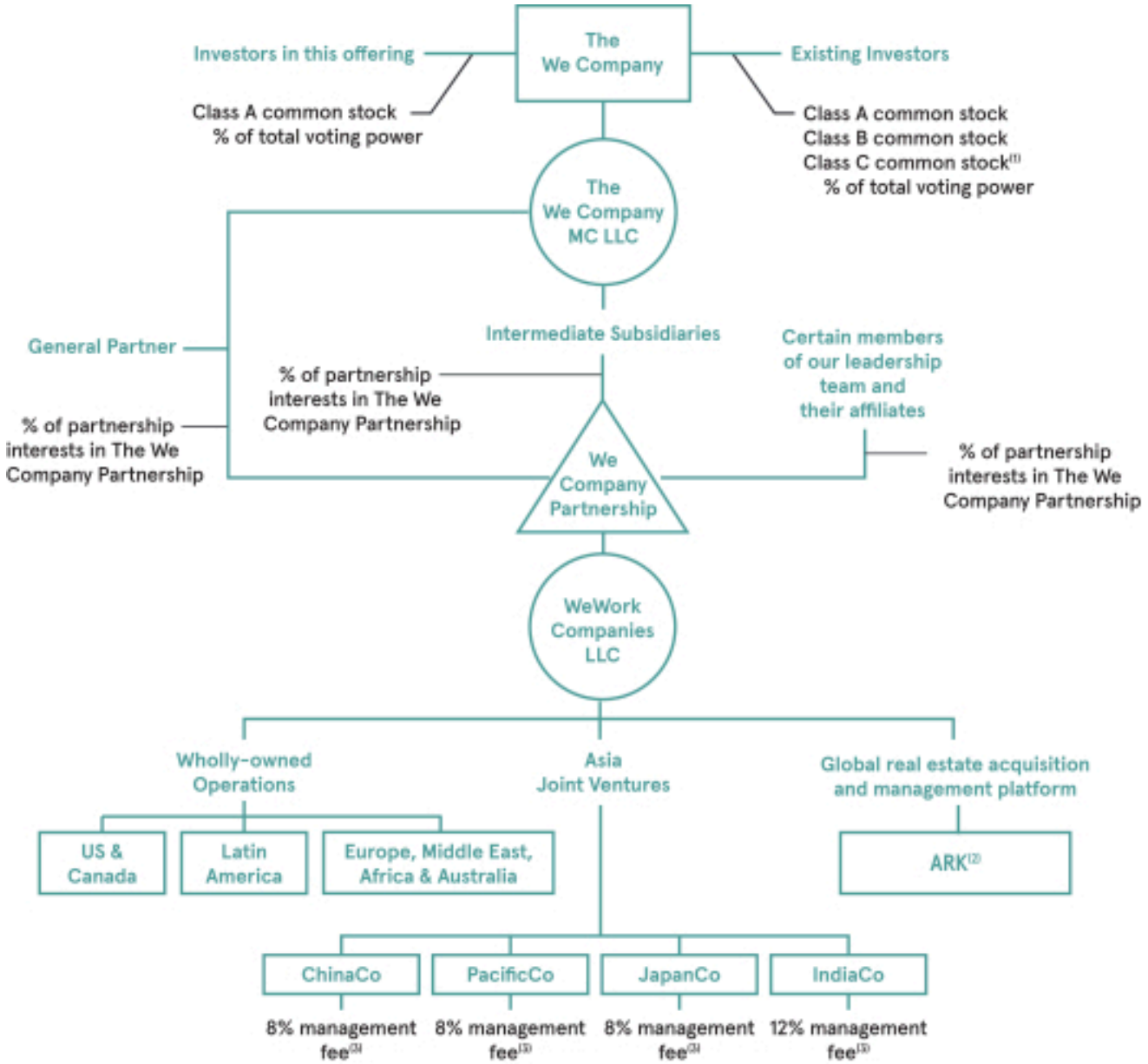


GENERAL HIERARCHY

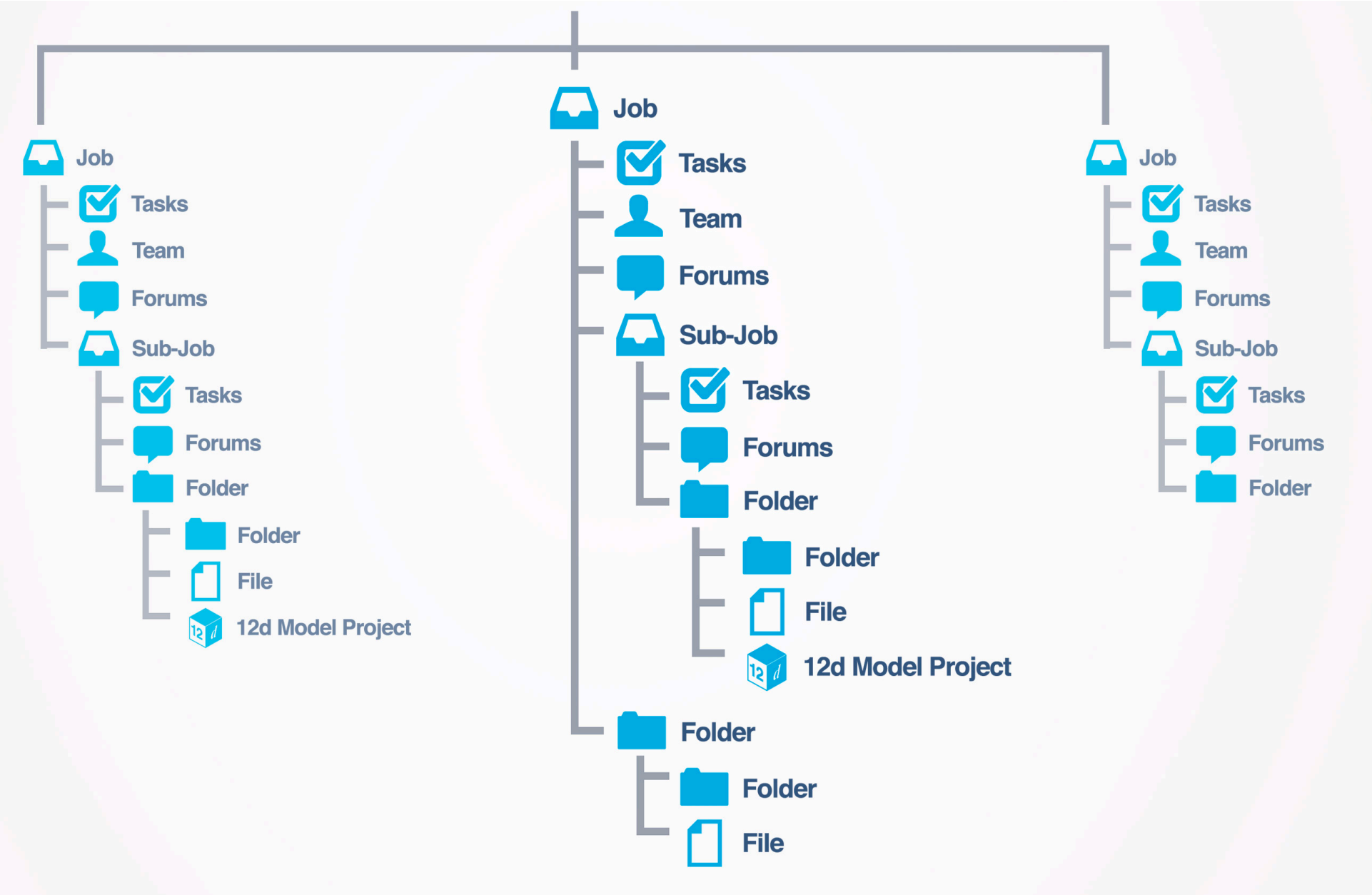
PATTERN

Context

- Objects in a hierarchy can have one or more objects above them (superiors),
  - and one or more objects below them (subordinates).
- Some objects cannot have any subordinates



https://www.businessinsider.my/wework-adopts-up-c-structure-2019-8/


Job
Tasks
Team
Forums
Sub-Job
Tasks
Forums
Folder
Folder
File
12d Model Project
Job
Tasks
Team
Forums
Sub-Job
Tasks
Forums
Folder
Folder
File
12d Model Project
Folder
Folder
File
Job
Tasks
Team
Forums
Sub-Job
Tasks
Forums
Folder

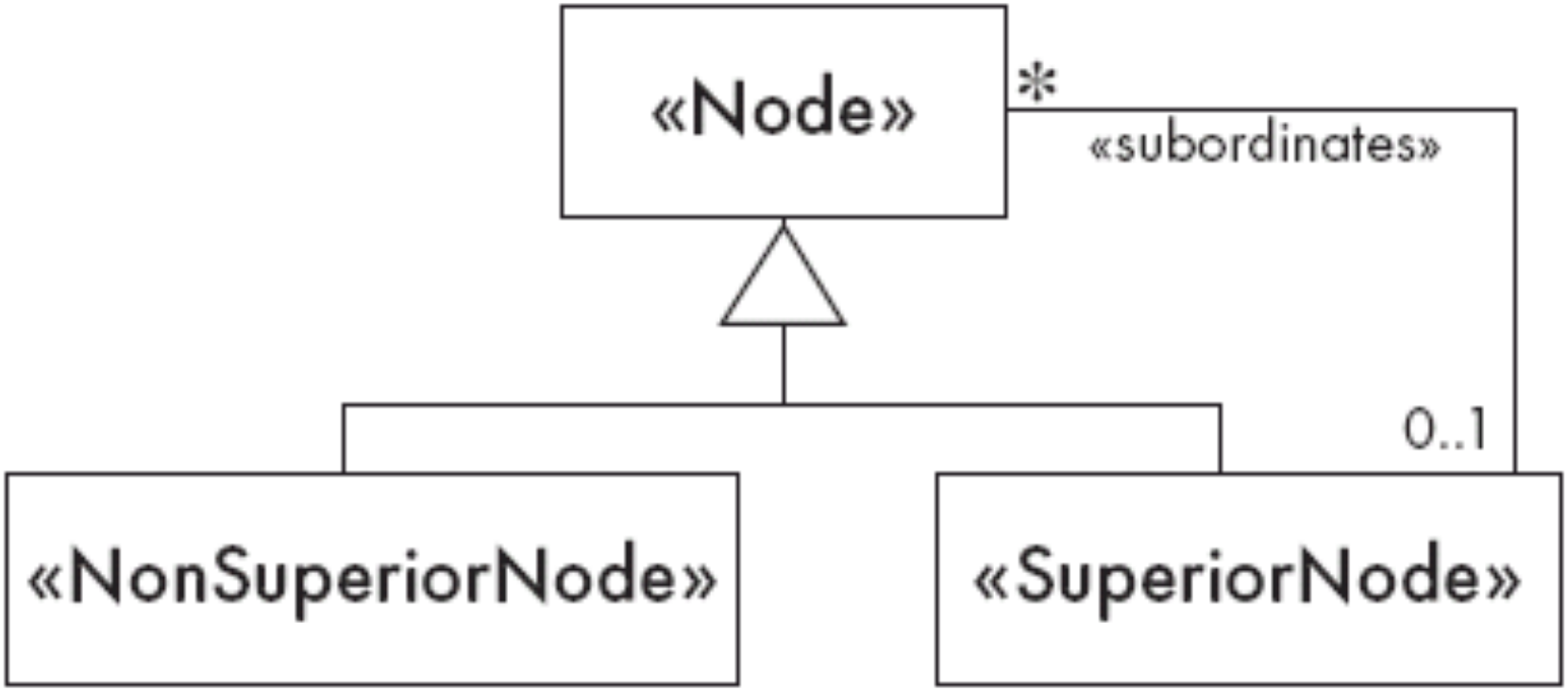
PROBLEM

- How do you represent a hierarchy of objects, in which some objects cannot have subordinates?

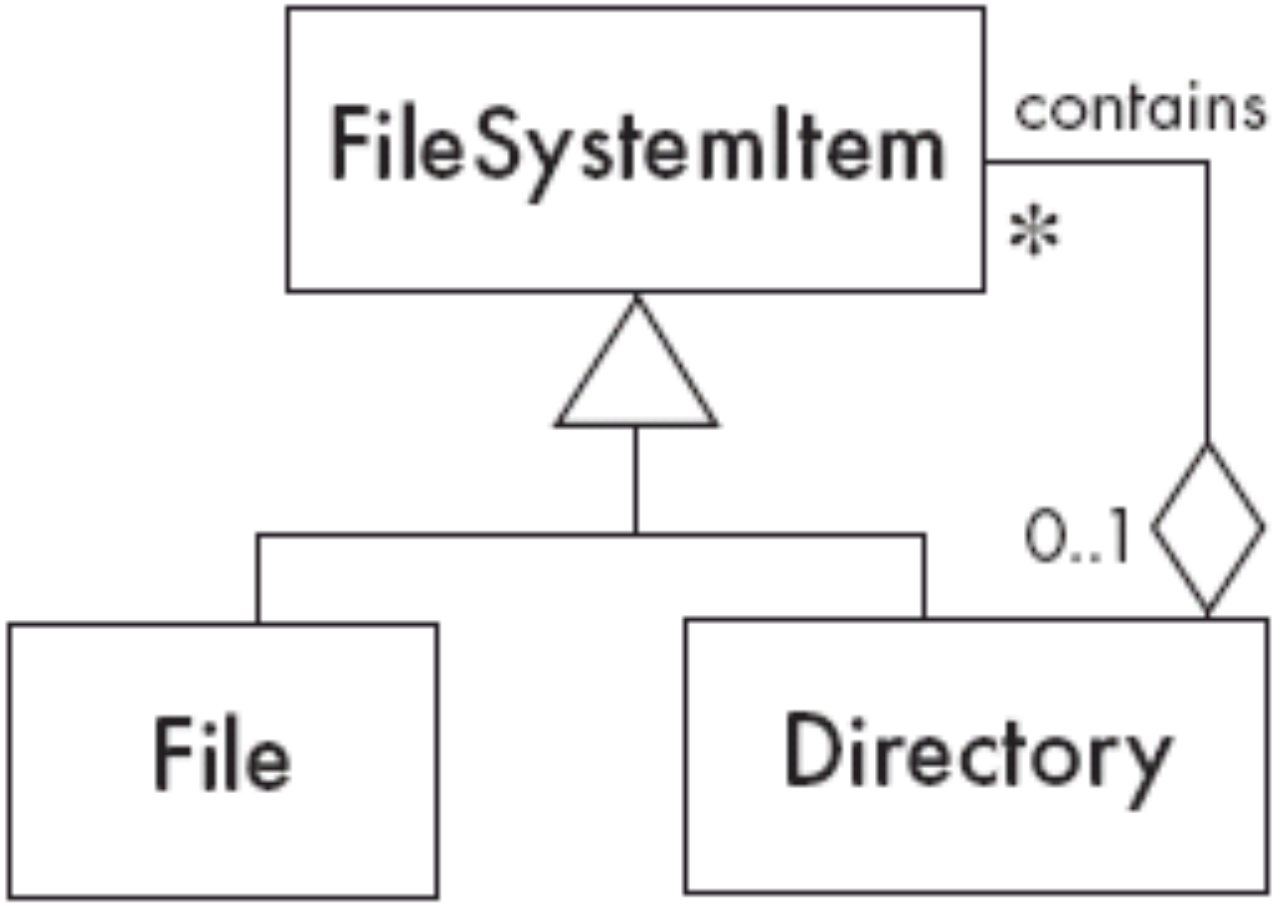
FORCES

- You want a flexible way of representing the hierarchy
  - that prevents certain objects from having subordinates
- All the objects have many common properties and operations

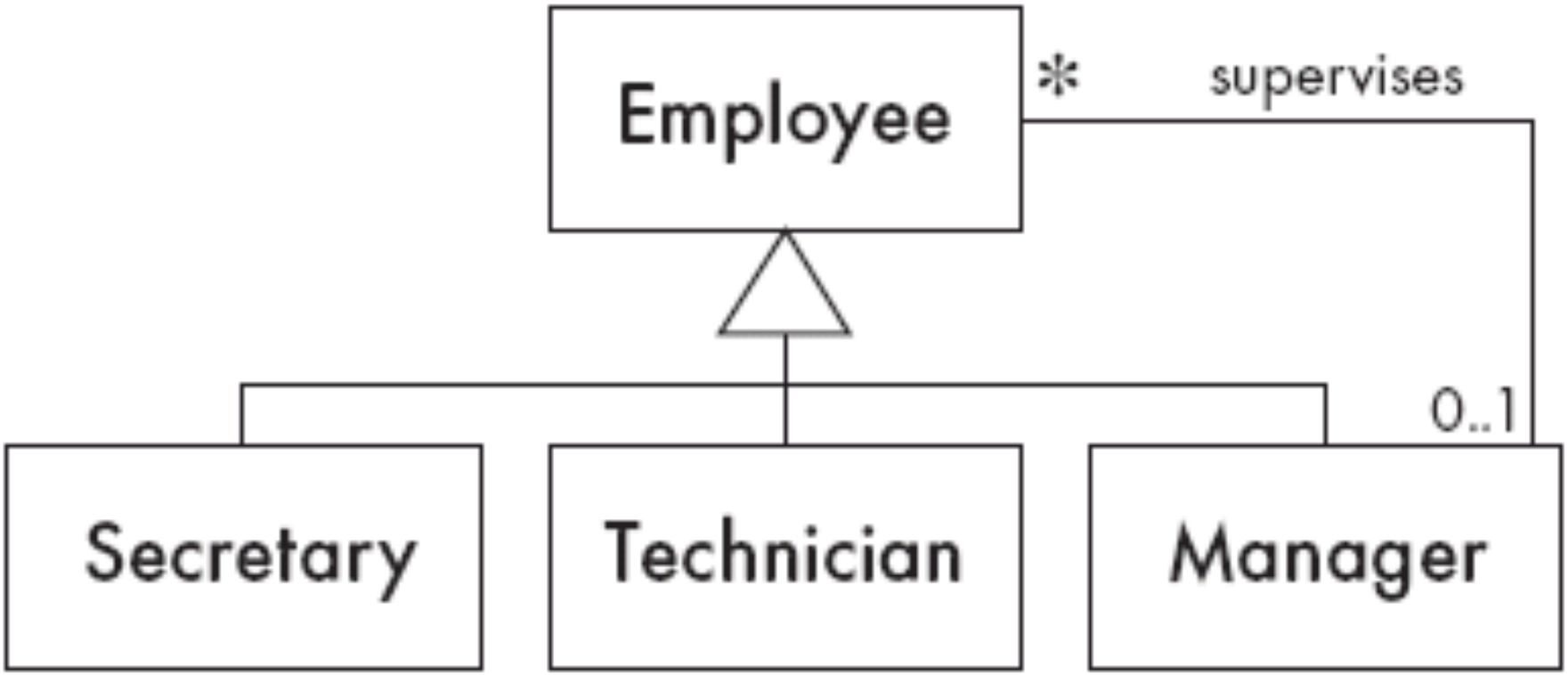
Solution



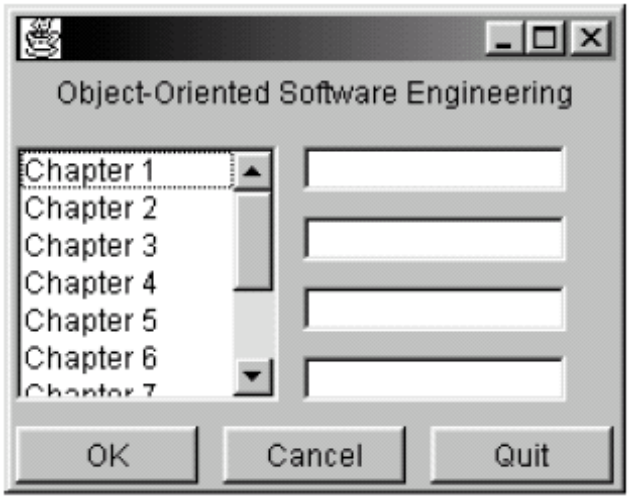
Example: File System



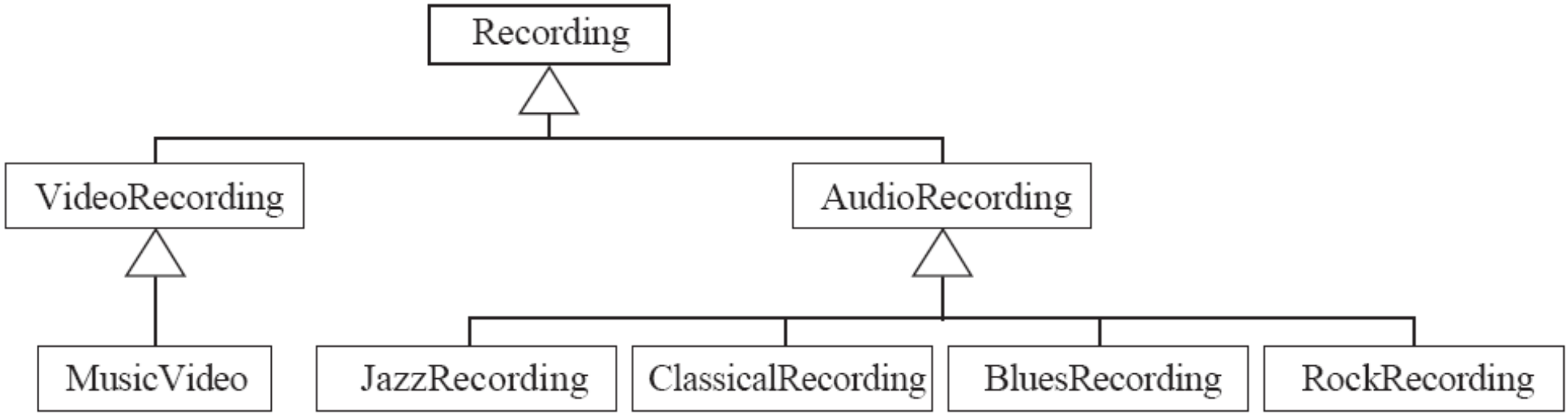
Example: Employees



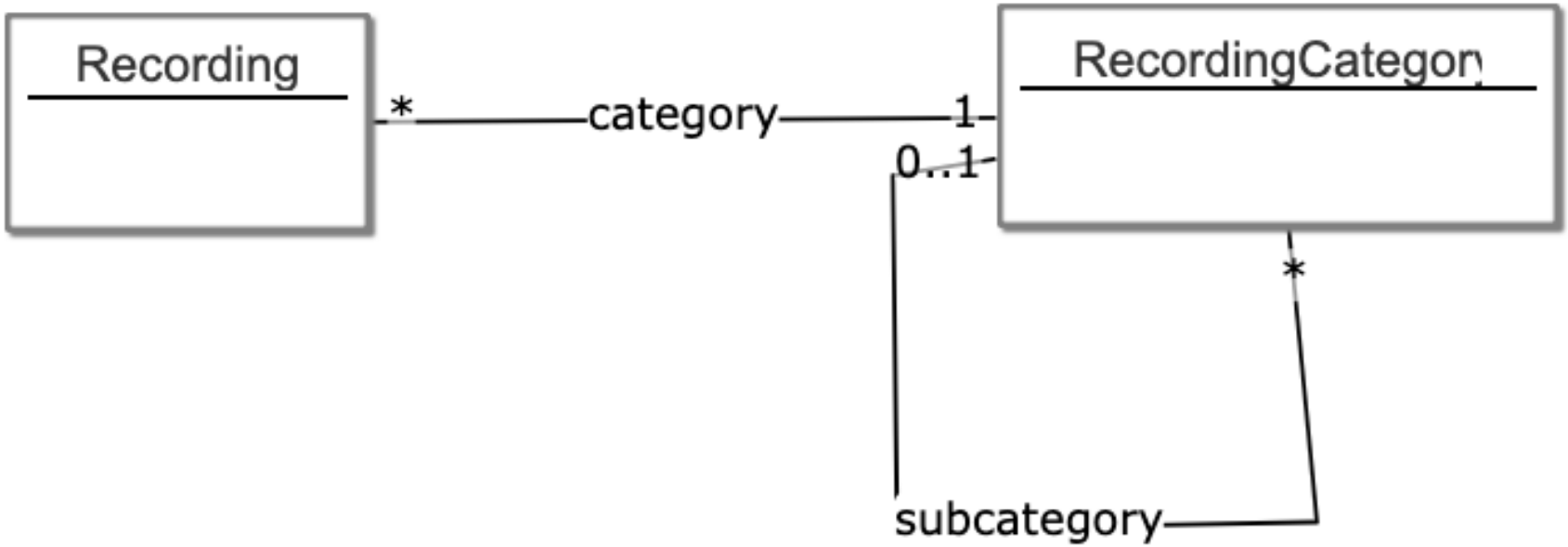
EXAMPLE: GUIS



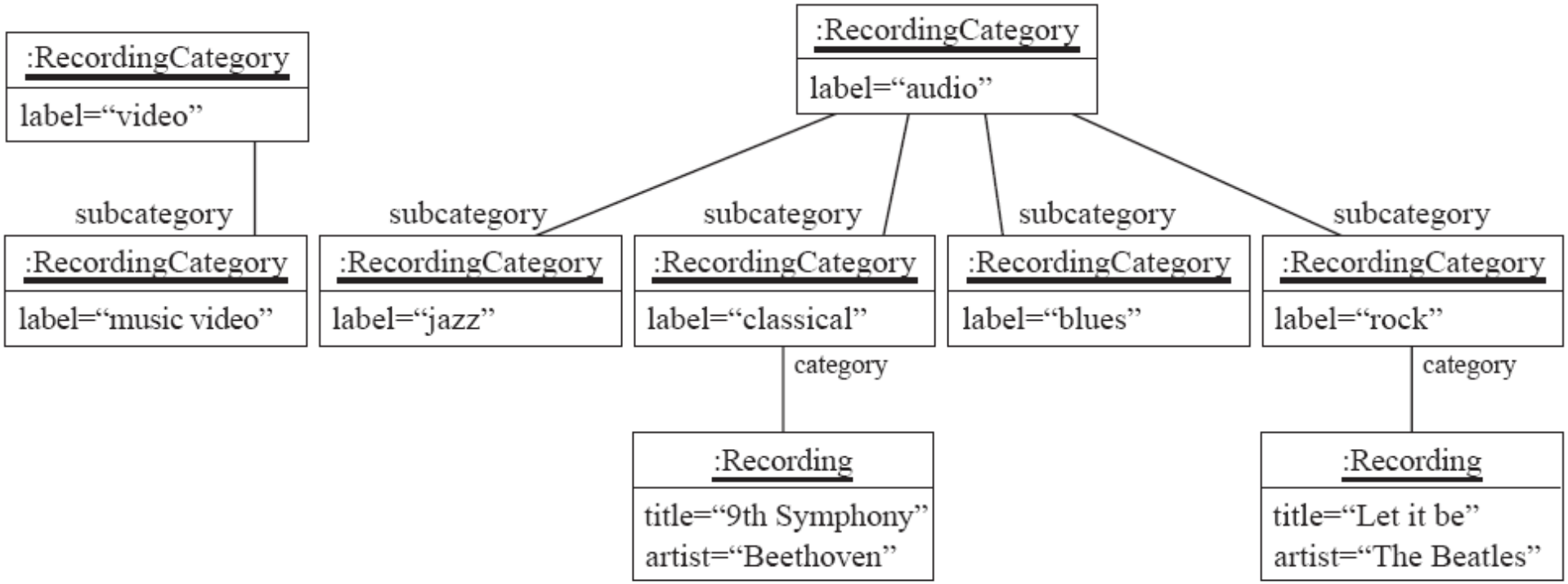
ANTI-PATTERN



THE BETTER SOLUTION



CORRESPONDING OBJECT DIAGRAM



PLAYER-ROLE

PATTERN

Context

- A role is a particular set of properties associated with an object in a particular context.
- An object may play different roles in different contexts.


SUPER
SMASH BROS.™
ULTIMATE

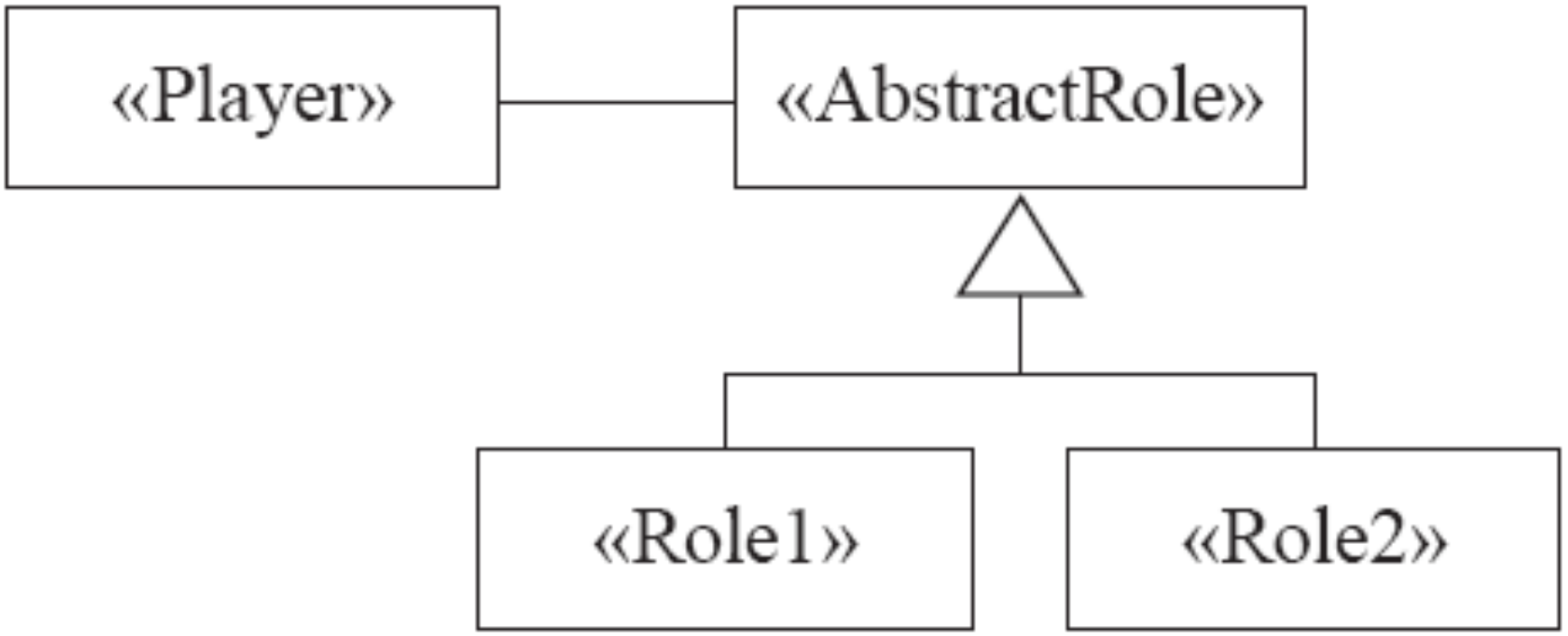
PROBLEM

- How do you best model players and roles so that a player can change roles or possess multiple roles?

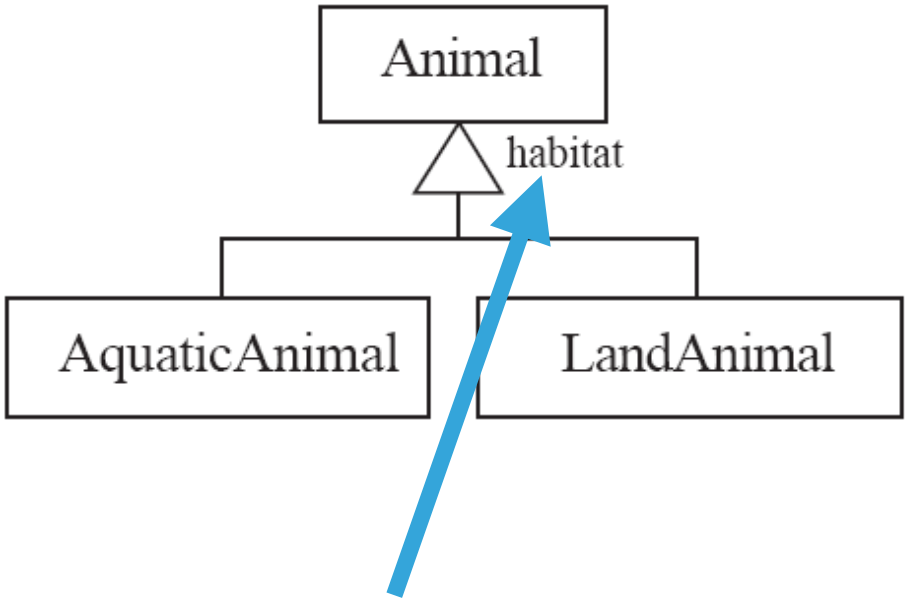
Forces

- It is desirable to improve encapsulation by capturing the information associated with each separate role in a class.
- You want to avoid multiple inheritance.
- You cannot allow an instance to change class

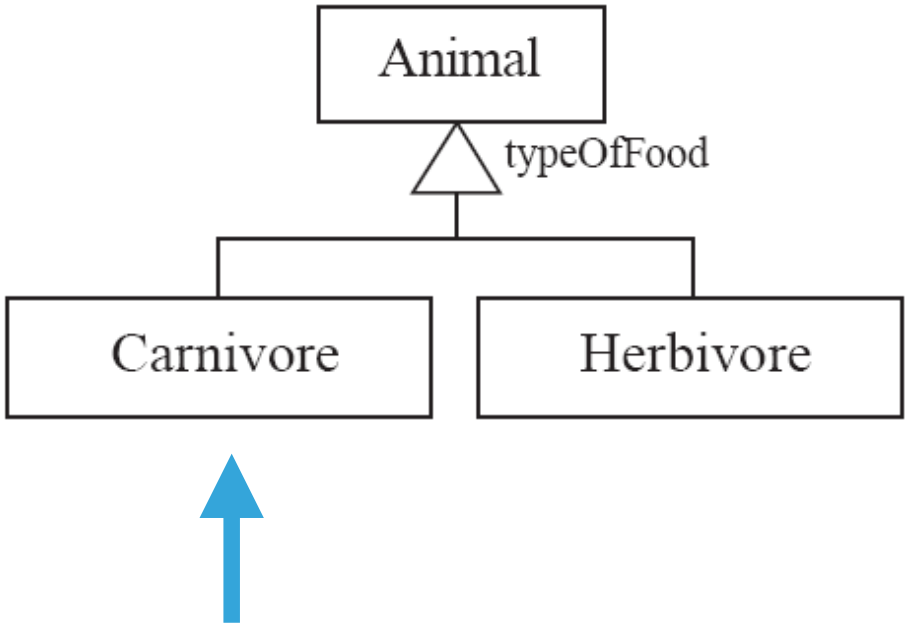
Solution



RECALL...

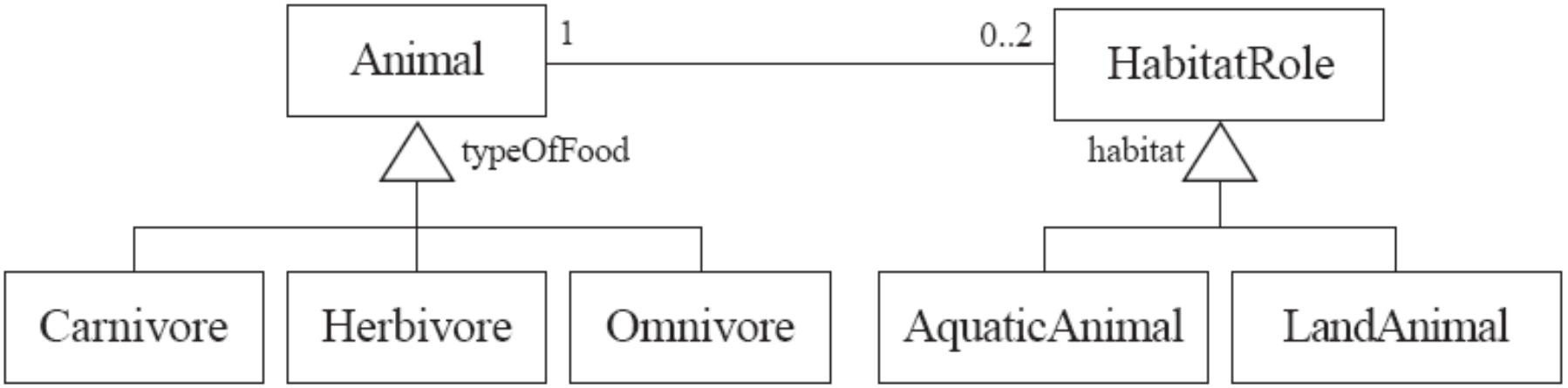


Label (or discriminator) describes the criteria

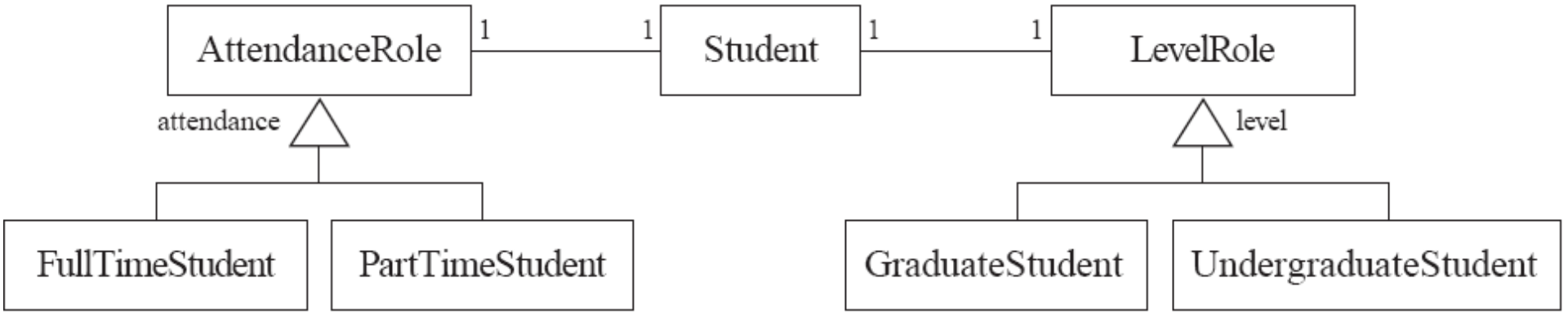


Many ways to generalize

EXAMPLE: ANIMALS



EXAMPLE: STUDENTS

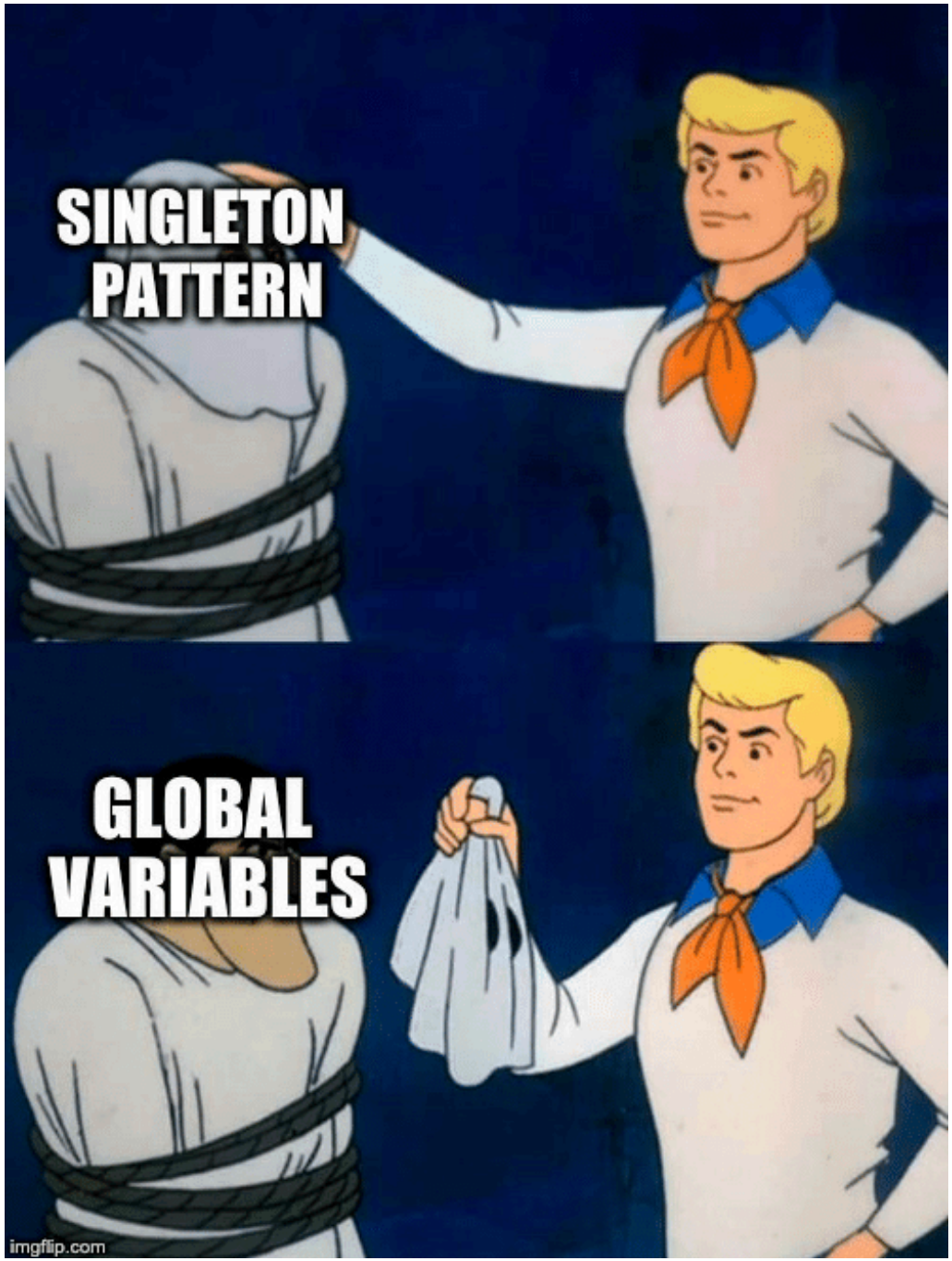


Antipatterns

- Merge all the properties and behaviours into a single «Player» class and not have «Role» classes at all.
- Create roles as subclasses of the «Player» class.

SINGLETON

PATTERN


SINGLETON PATTERN
GLOBAL VARIABLES
imgflip.com

CONTEXT

- It is very common to find classes for which only one instance should exist (singleton)

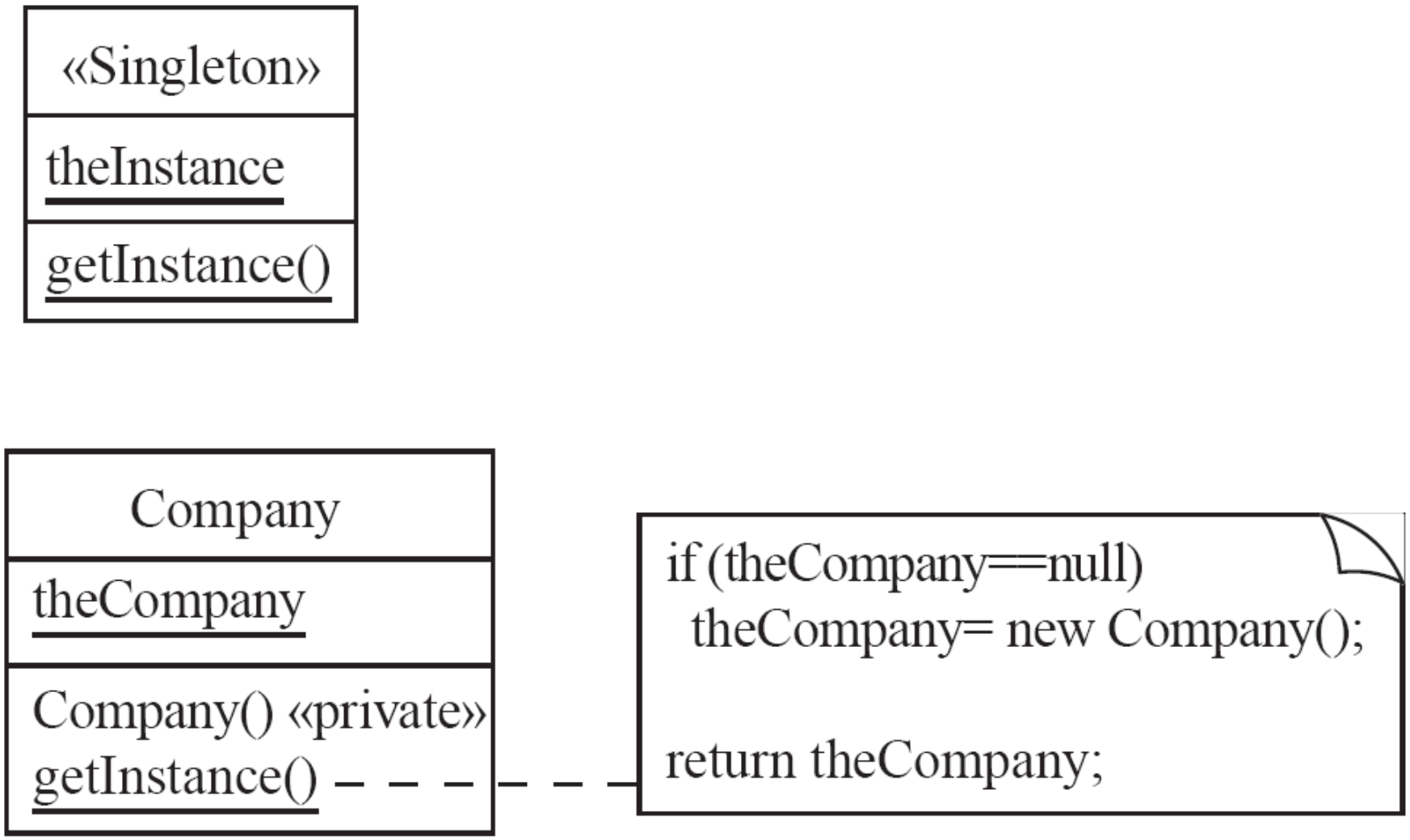
Problem

- How do you ensure that it is never possible to create more than one instance of a singleton class?

Forces

- The use of a public constructor cannot guarantee that no more than one instance will be created.
- The singleton instance must also be accessible to all classes that require it

SOLUTION


«Singleton»
theInstance
getInstance()
Company
theCompany
Company() «private»
getInstance()
if (theCompany==null)
theCompany= new Company();
return theCompany;

Global State

```
class X {

  ... come code / fields ...

  X() { ... }

  public int doSomething(){ ... }
}

int a = new X().doSomething();
int b = new X().doSomething();
```

https://www.youtube.com/watch?v=acjvKJiOvXw

OBSERVER

PATTERN

Context

- When an association is created between two classes, the code for the classes becomes inseparable.
- If you want to reuse one class, then you also have to reuse the other.

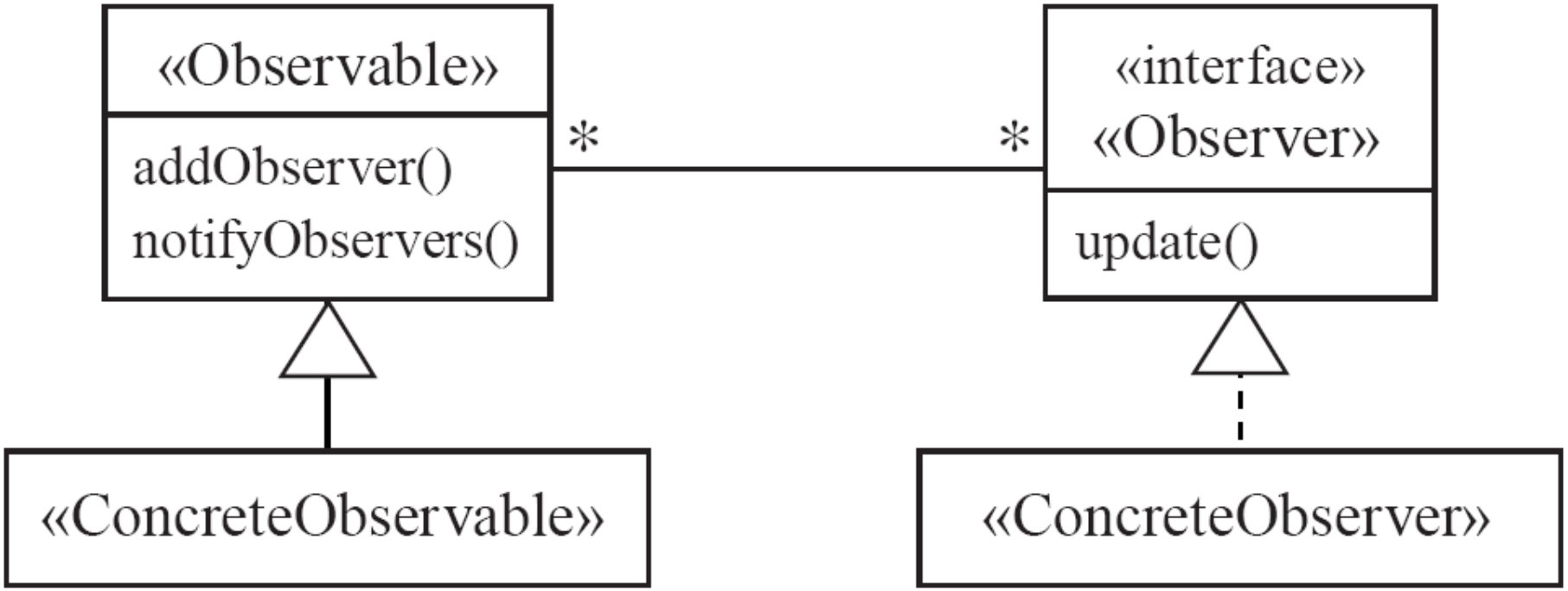
Problem

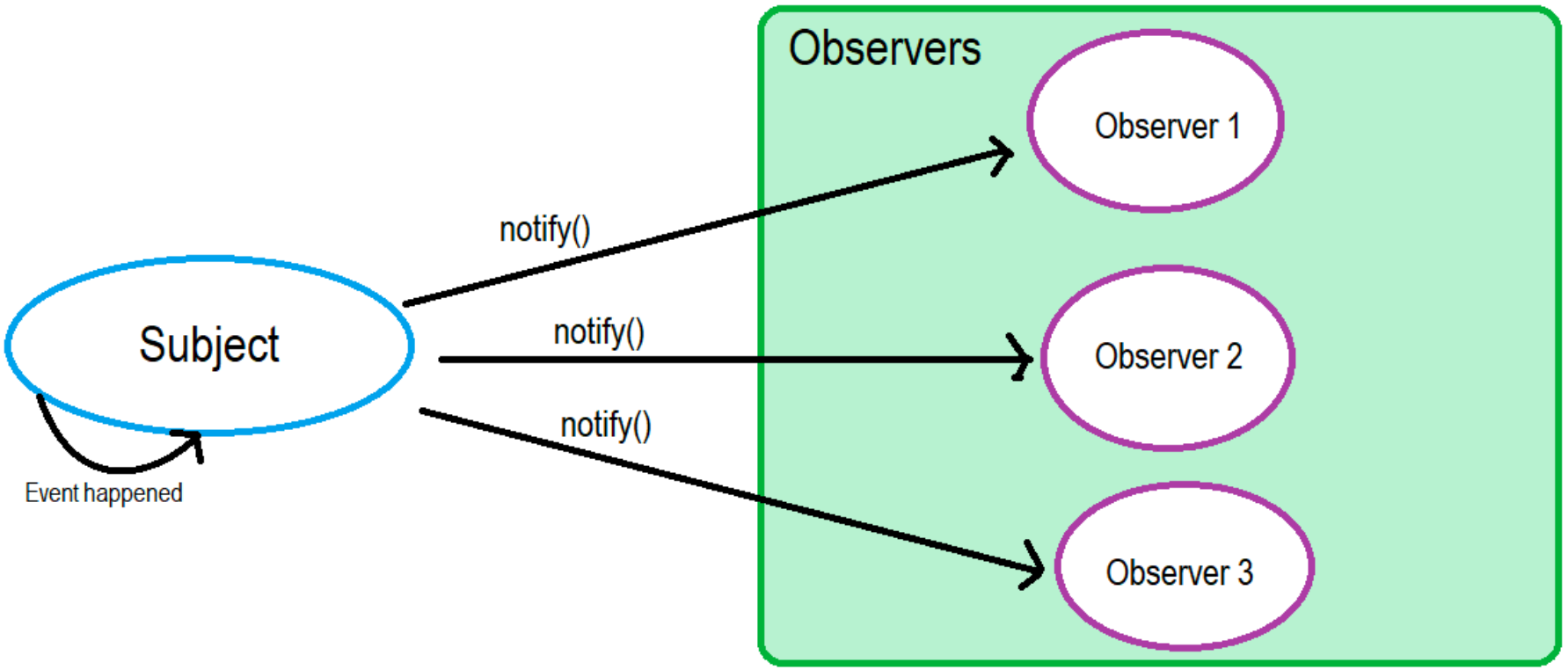
- How do you reduce the interconnection between classes, especially between classes that belong to different modules or subsystems?

Forces

- You want to maximize the flexibility of the system to the greatest extent possible.

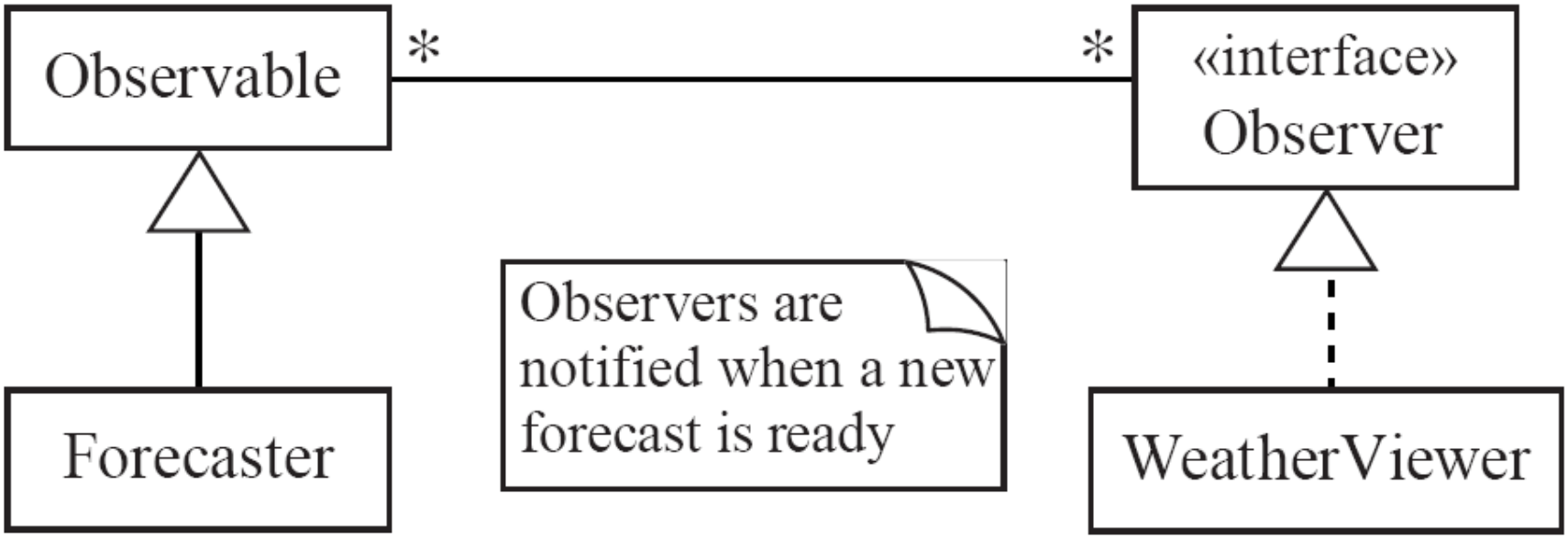
SOLUTION


«Observable»
addObserver()
notifyObservers()
*
*
«interface»
«Observer»
update()
«ConcreteObservable»
«ConcreteObserver»



https://hackernoon.com/observer-vs-pub-sub-pattern-50d3b27f838c

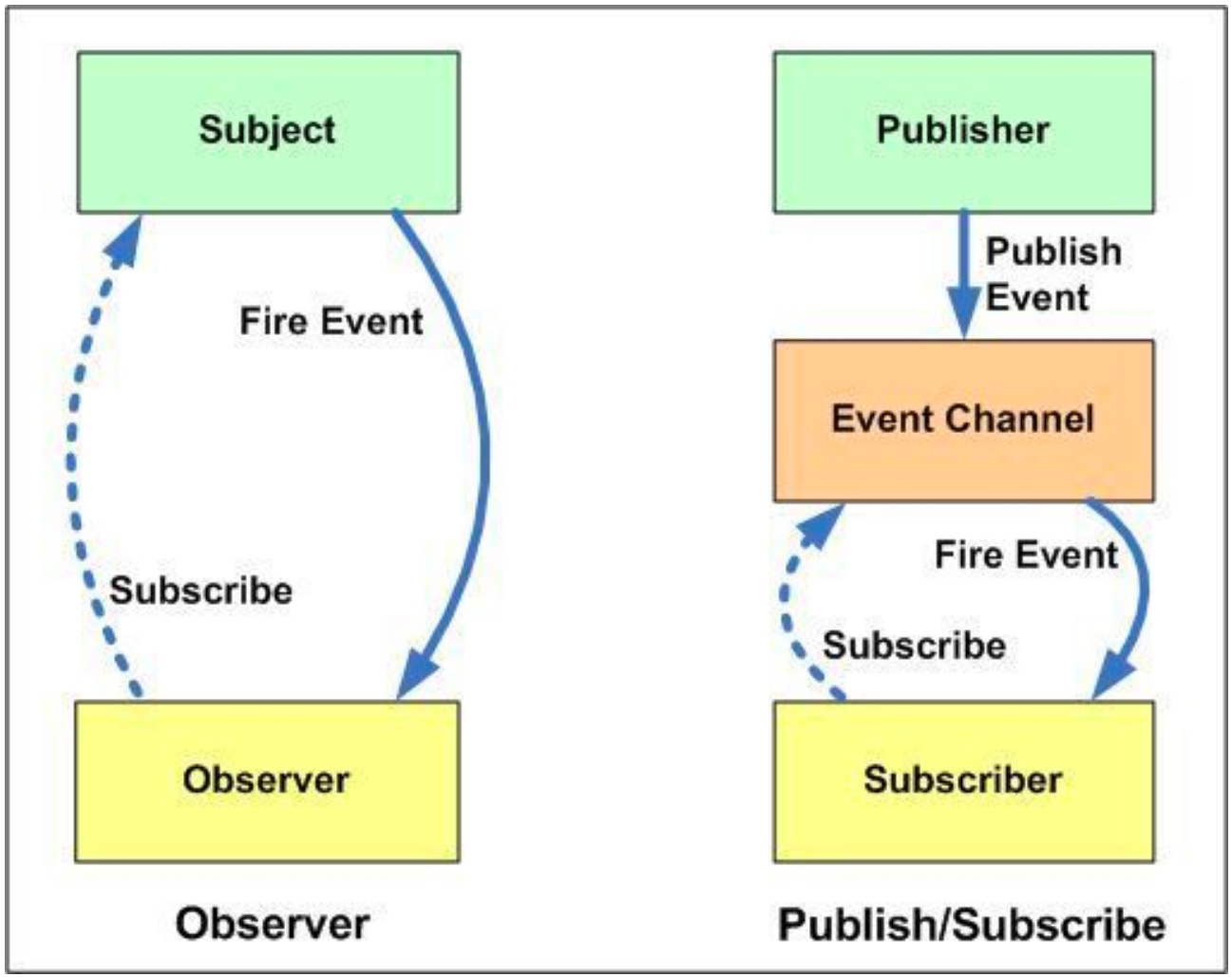
Example: Weather Forecaster



Antipatterns

- Connect an observer directly to an observable so that they both have references to each other.
- Make the observers subclasses of the observable.

PUB/SUB VS OBSERVER



https://hackernoon.com/observer-vs-pub-sub-pattern-50d3b27f838c

DELEGATION

PATTERN

https://blog.kotlin-academy.com/programmer-dictionary-delegation-vs-composition-3025d9e8ae3d

Context

- You are designing a method in a class
  - You realize that another class has a method which provides the required service
- Inheritance is not appropriate
  - E.g. because the isa rule does not apply

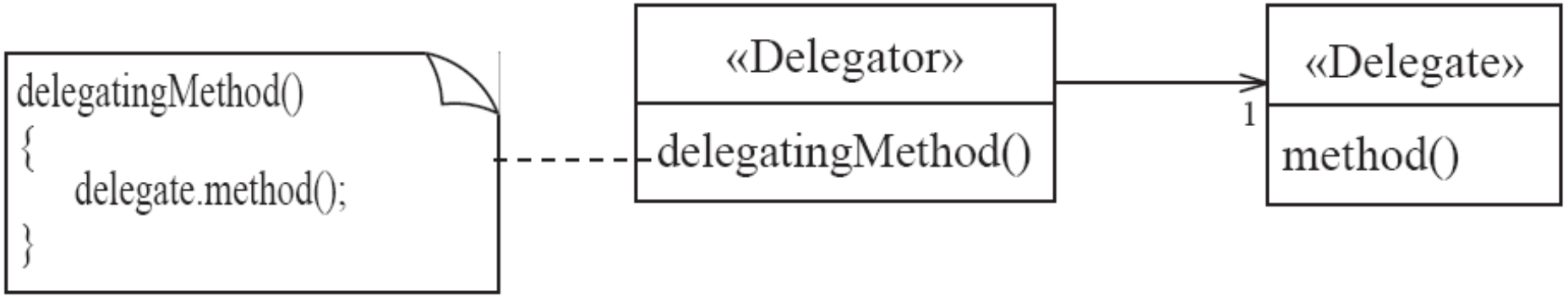
Problem

- How can you most effectively make use of a method that already exists in the other class?

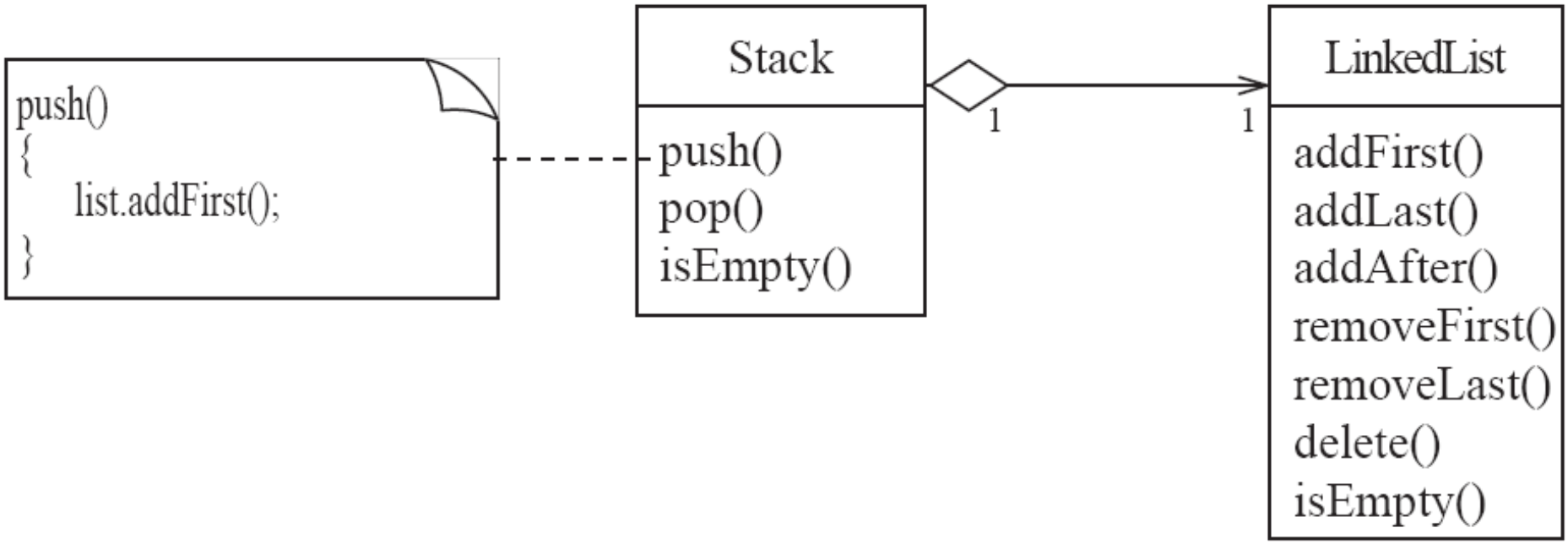
Forces

- You want to minimize development cost by reusing methods

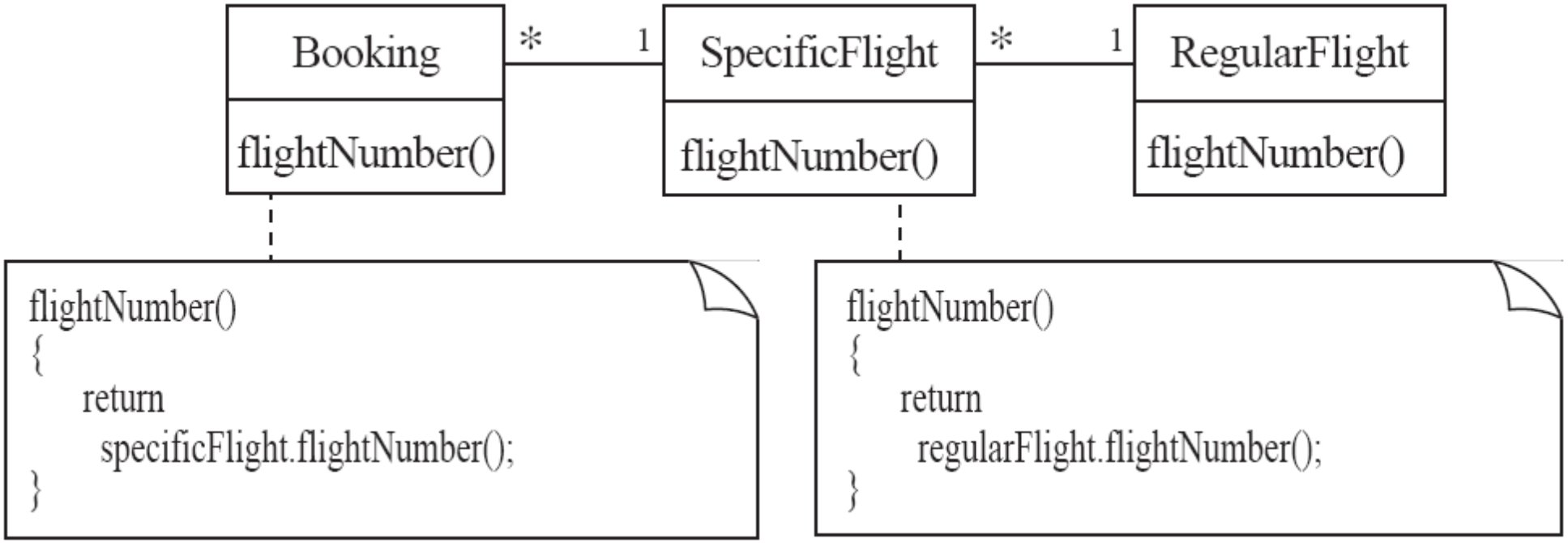
SOLUTION


delegatingMethod()
{
delegate.method();
}
«Delegator»
delegatingMethod()
1
«Delegate»
method()

EXAMPLE: STACK



EXAMPLE: FLIGHTNUMBER()

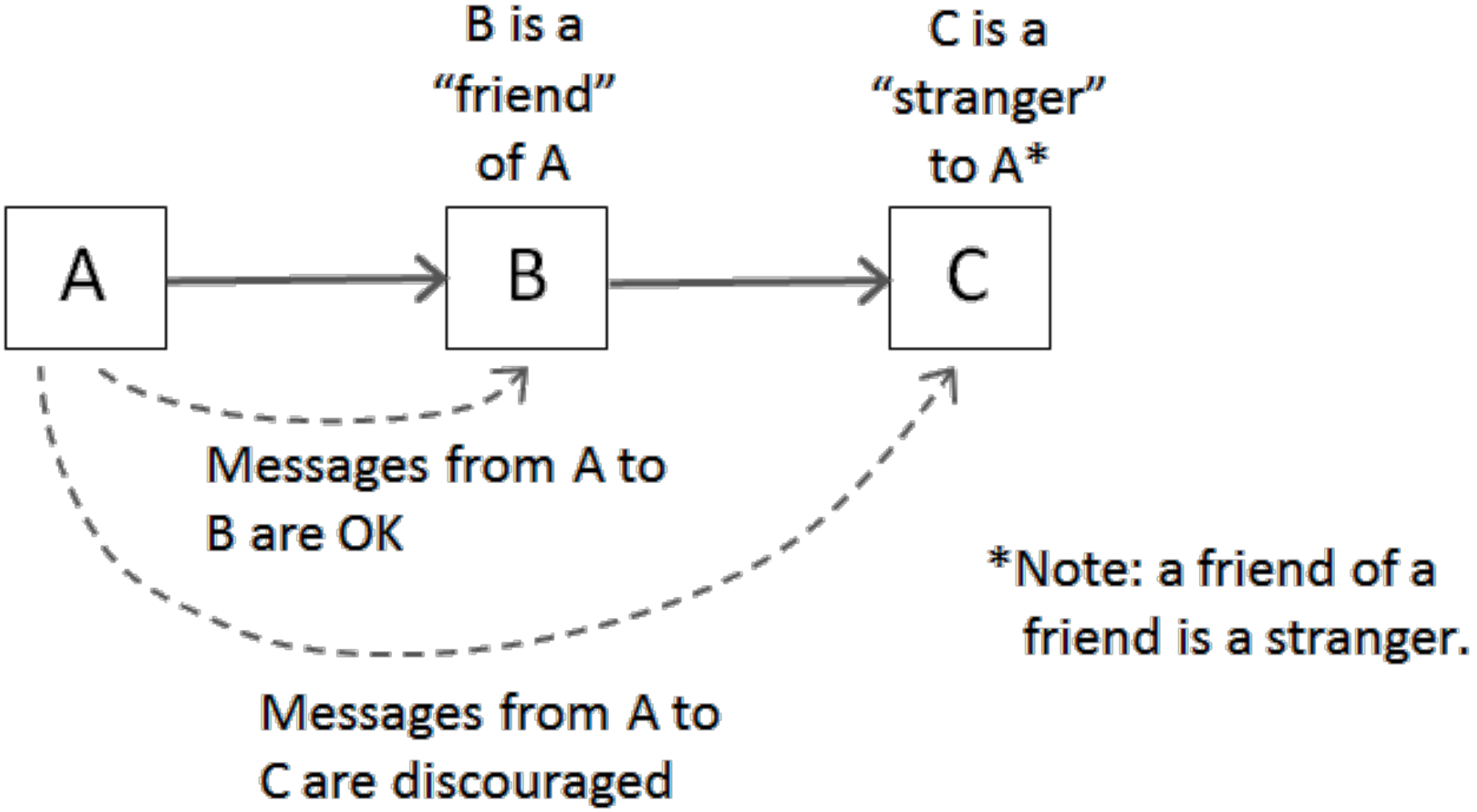


ANTIPATTERNS

- Overuse generalization and *inherit* the method that is to be reused
- Instead of creating a *single* method in the «Delegator» that does nothing other than call a method in the «Delegate»
  - consider having many different methods in the «Delegator» call the delegate's method
- Access non-neighboring classes

```
return specificFlight.regularFlight.flightNumber();

return getRegularFlight().flightNumber();
```

PRINCIPLE OF LEAST KNOWLEDGE (LAW OF DEMETER)



https://en.wikipedia.org/wiki/Law_of_Demeter

https://sce2.umkc.edu/BIT/burrise/pl/design/ood_heuristics.html

ADAPTER

PATTERN

Context

- You are building an inheritance hierarchy and want to incorporate it into an existing class.
- The reused class is also often already part of its own inheritance hierarchy.

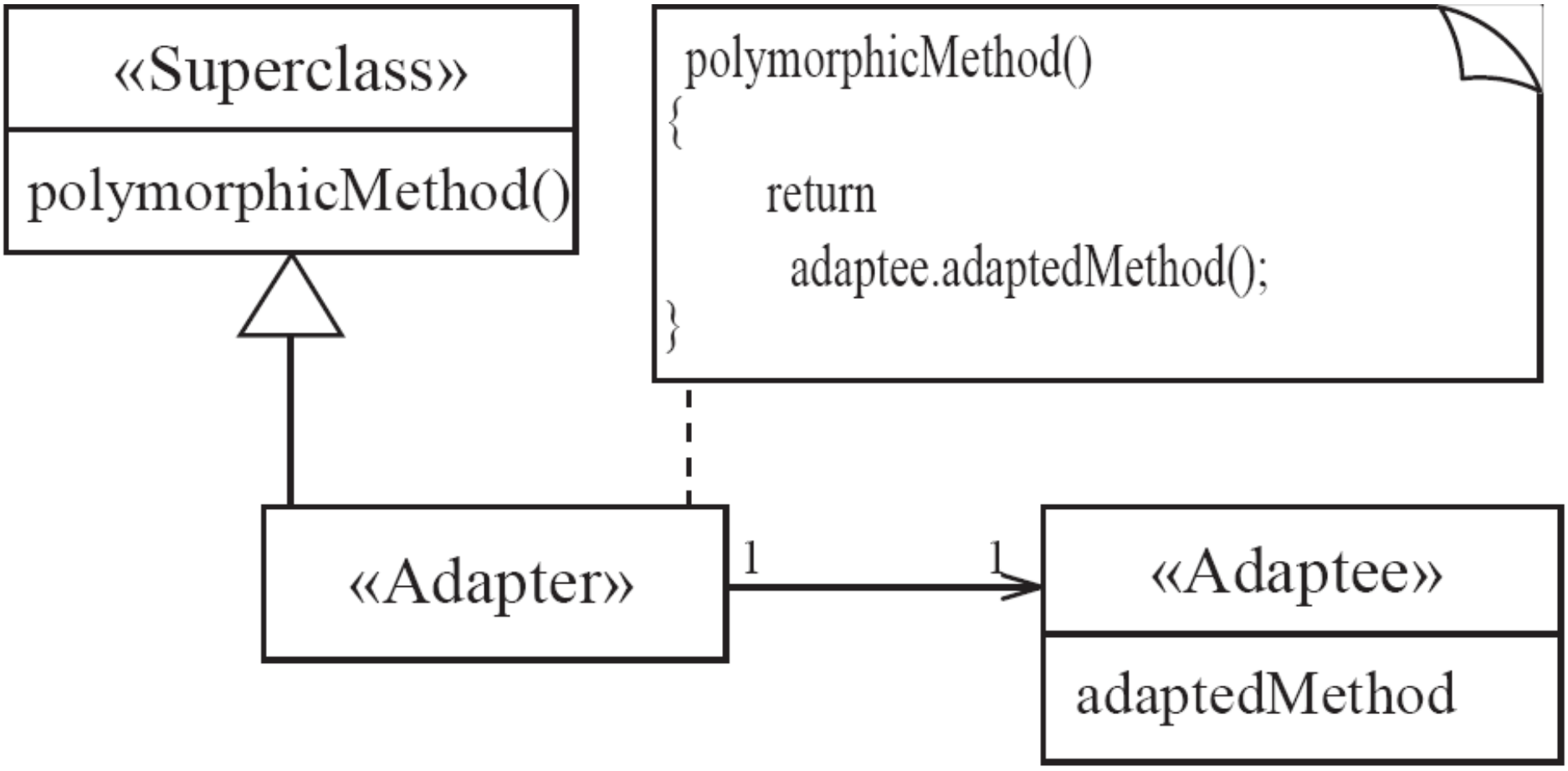
PROBLEM

- How to obtain the power of polymorphism when reusing a class whose methods
  - have the same function
  - but not the same signature
- as the other methods in the hierarchy?

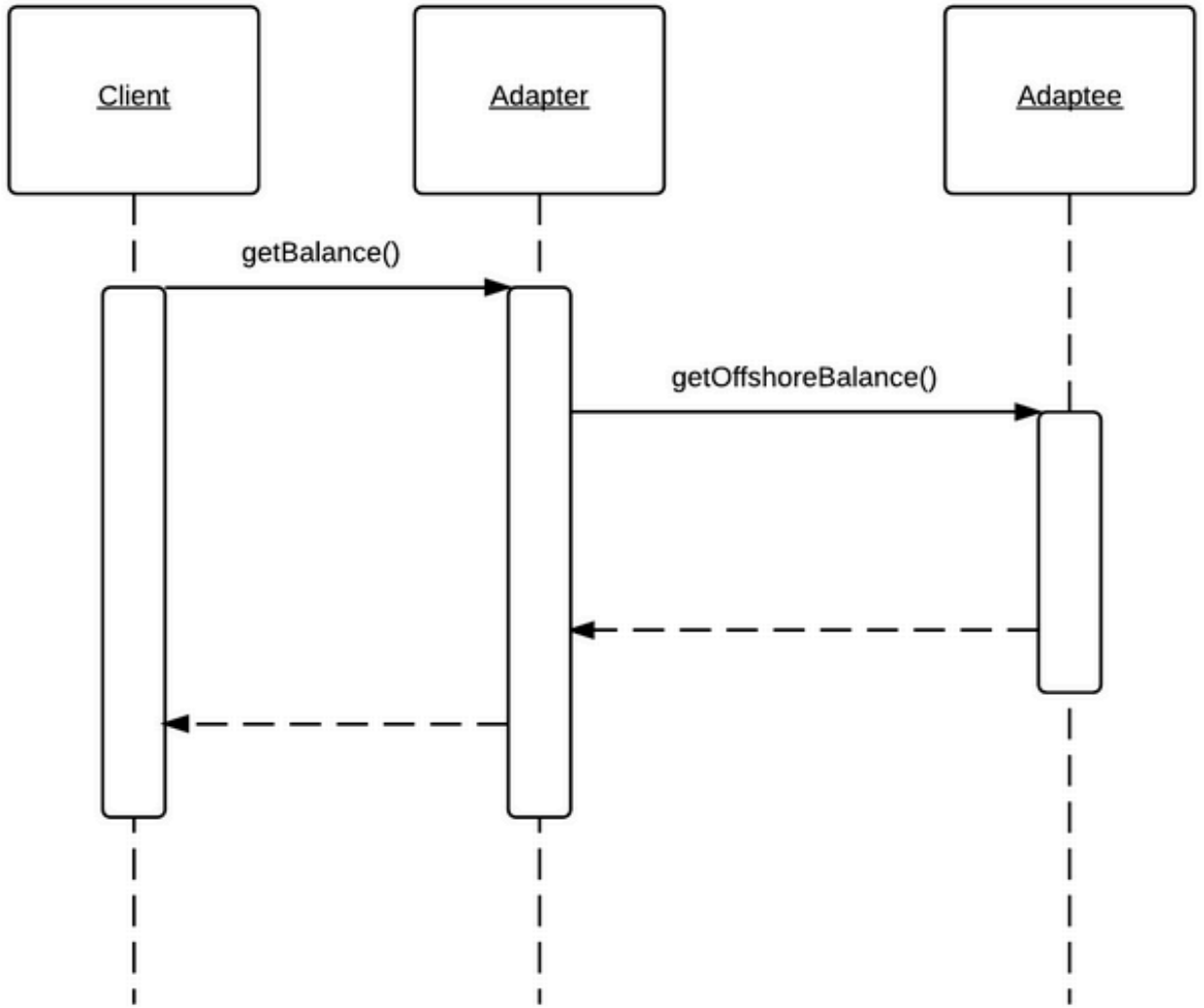
Forces

- You do not have access to multiple inheritance or you do not want to use it.

SOLUTION


«Superclass»
polymorphicMethod()
polymorphicMethod()
{
return
adaptee.adaptedMethod();
}
«Adapter»
1
1
«Adaptee»
adaptedMethod

SOLUTION



https://stacktips.com/tutorials/design-patterns/adapter-design-pattern-in-java/attachment/adapter-design-pattern-sequence-diagram

EXAMPLE: PLUGS

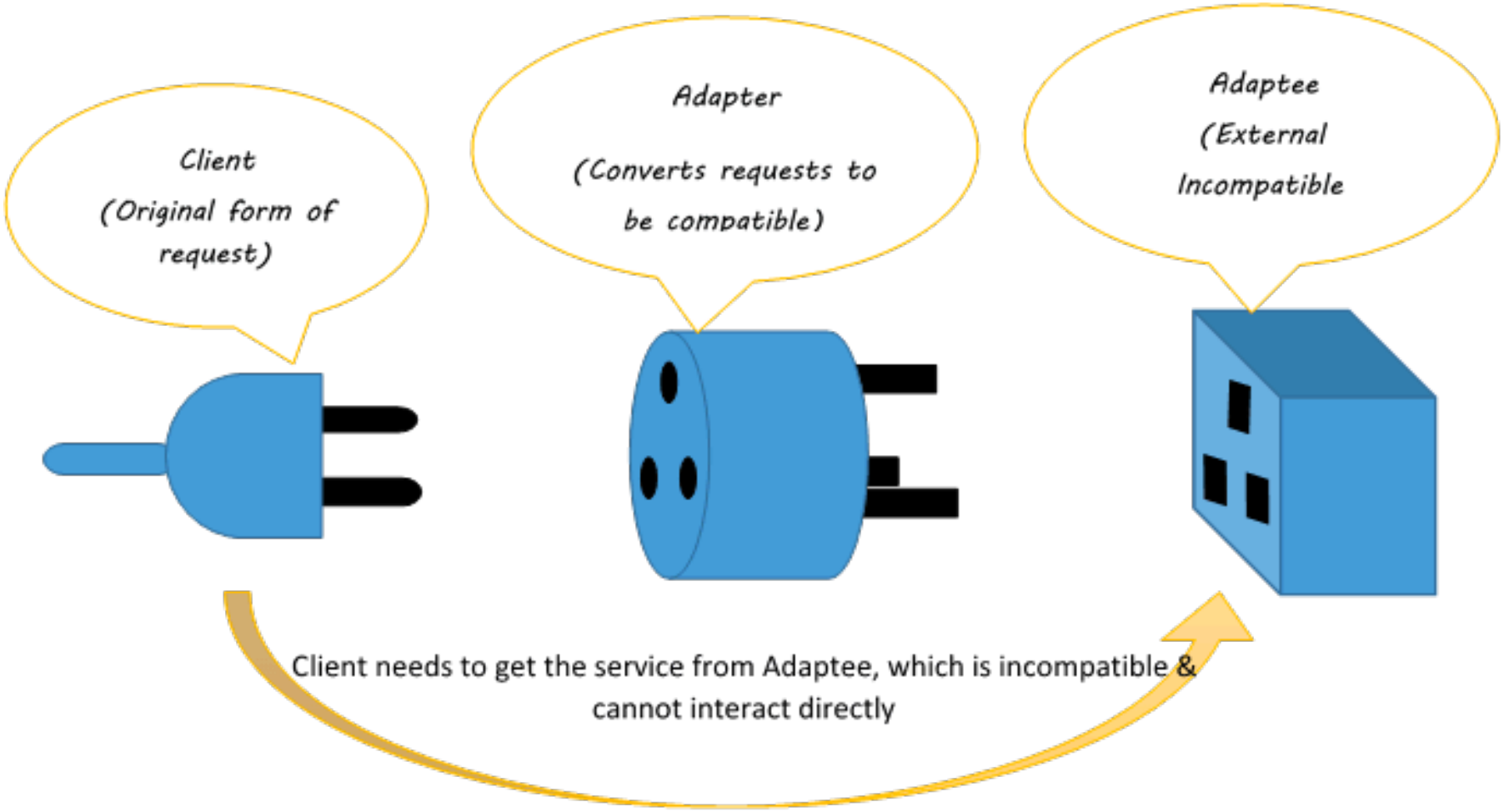
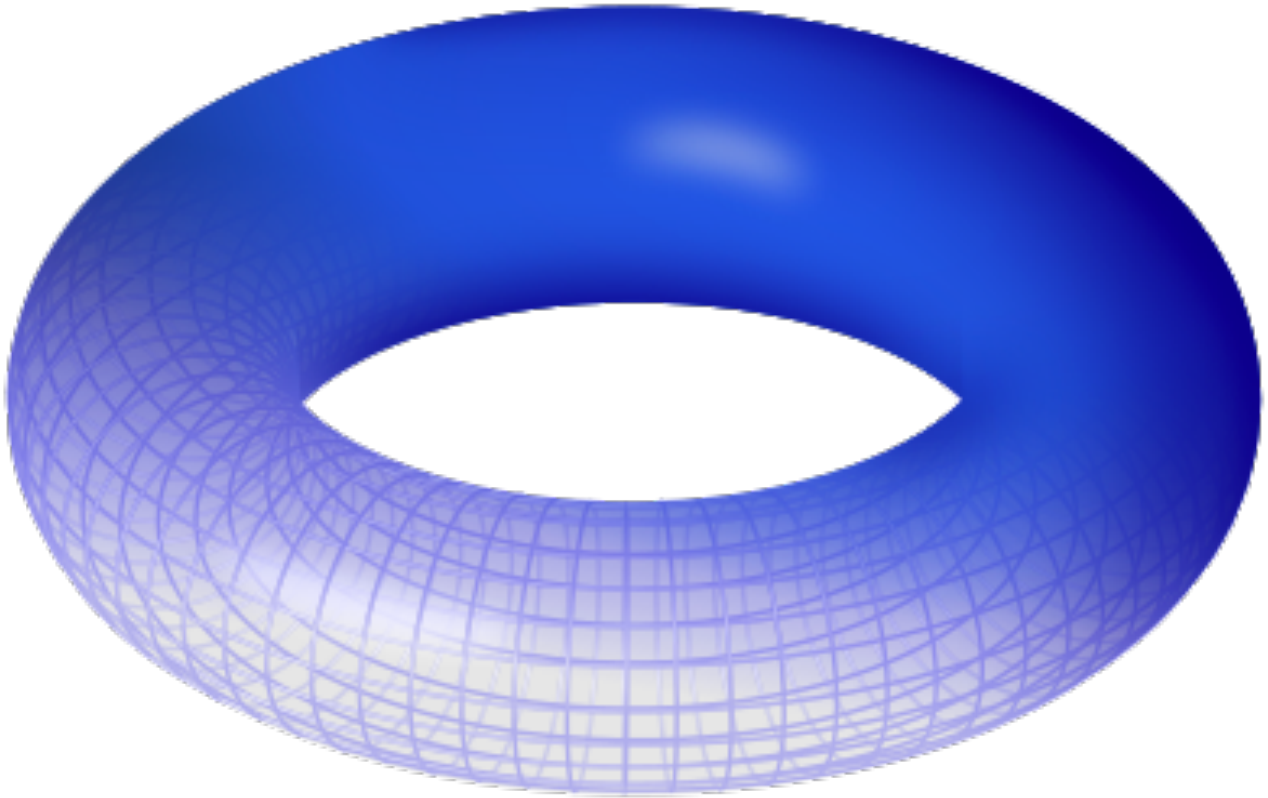


Figure 1-Adapter Pattern Concept

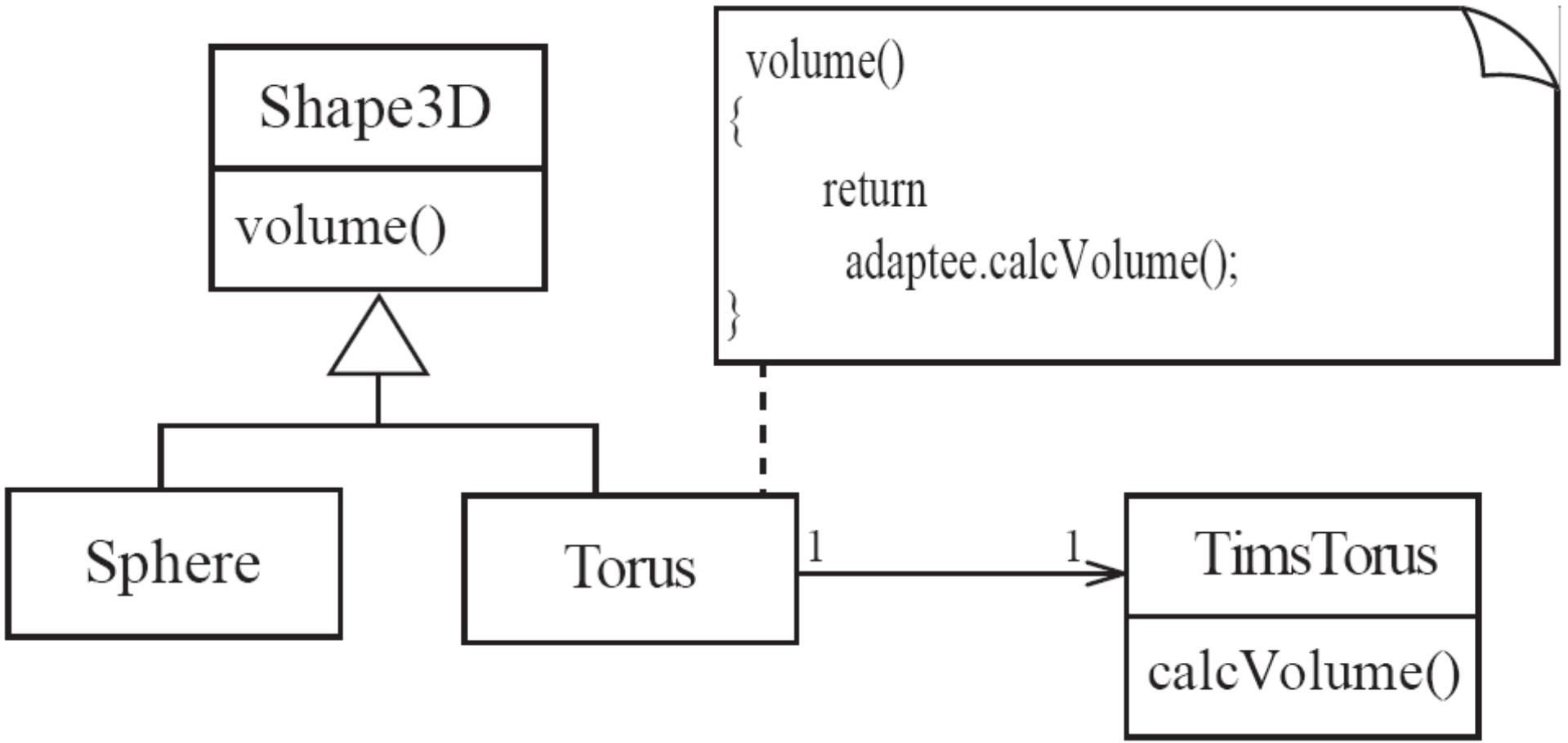
https://www.javagists.com/adapter-design-pattern

What is a Torus?



https://en.wikipedia.org/wiki/Torus

EXAMPLE: SHAPES



FAÇADE

PATTERN


150 ELGIN STREET
shopify
CIBC

Context

- Often, an application contains several complex packages.
- A programmer working with such packages has to manipulate many different classes

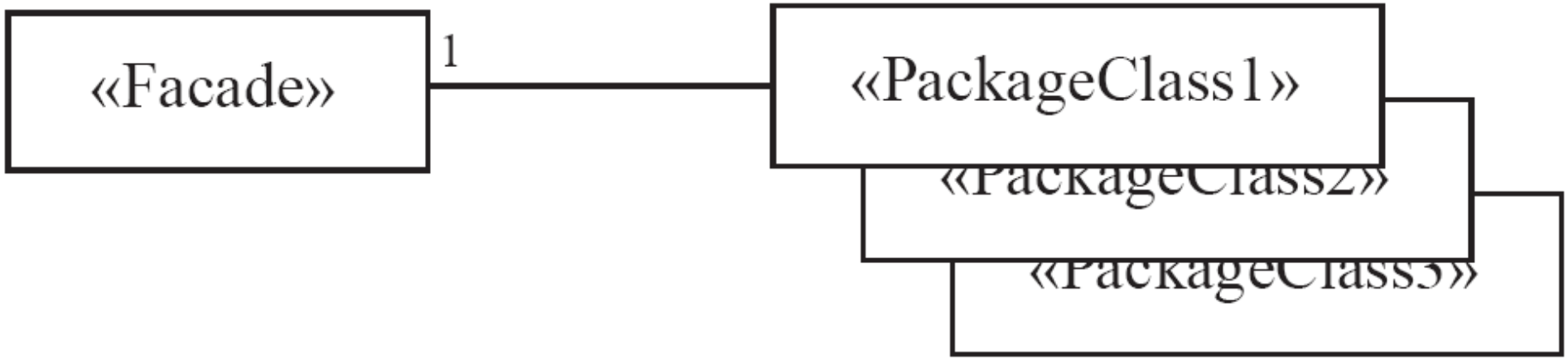
Problem

- How do you simplify the view that programmers have of a complex package?

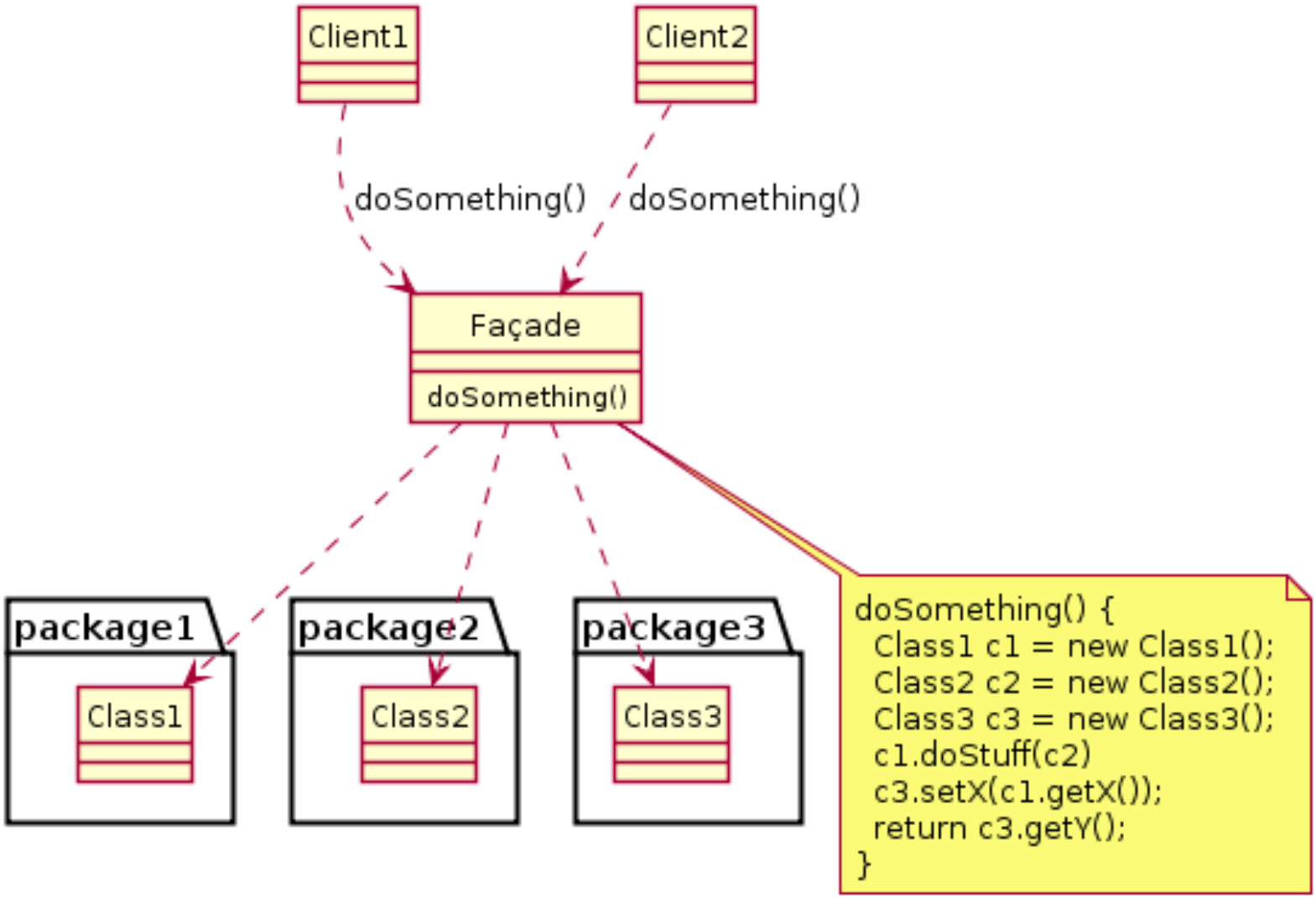
Forces

- It is hard for a programmer to understand and use an entire subsystem
- If several different application classes call methods of the complex package, then any modifications made to the package will necessitate a complete review of all these classes.

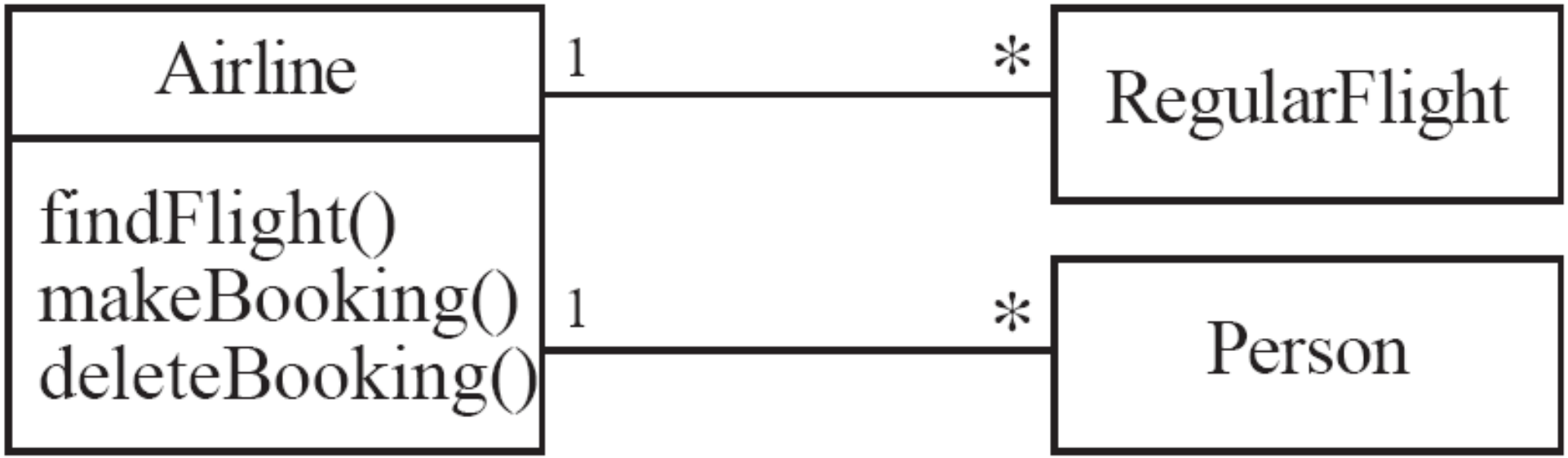
SOLUTION



SOLUTION


Client1
Client2
doSomething()
doSomething()
Façade
doSomething()
package1
Class1
package2
Class2
package3
Class3
doSomething() {
Class1 c1 = new Class1();
Class2 c2 = new Class2();
Class3 c3 = new Class3();
c1.doStuff(c2)
c3.setX(c1.getX());
return c3.getY();
}

EXAMPLE: AIRLINE



IMMUTABLE

PATTERN

immutable adjective

im·mu·ta·ble | \ (ˌ)i(m)-ˈmyü-tə-bəl \

Definition of *immutable*

: not capable of or susceptible to change

https://www.merriam-webster.com/dictionary/immutable

Context

- An immutable object is an object that has a state that never changes after creation

Problem

- How do you create a class whose instances are immutable?

Forces

- There must be no loopholes that would allow ‘illegal’ modification of an immutable object

Solution

- Ensure that the constructor of the immutable class is the only place where the values of instance variables are set or modified.
- Instance methods which access properties must not have side effects.
- If a method that would otherwise modify an instance variable is required, then it has to return a new instance of the class.

EXAMPLE (UMPLE)

```
class Person {
  immutable name;
}
```

```
10. public class Person
11. {
12.
13.   //------------------------
14.   // MEMBER VARIABLES
15.   //------------------------
16.
17.   //Person Attributes
18.   private String name;
19.
20.   //------------------------
21.   // CONSTRUCTOR
22.   //------------------------
23.
24.   public Person(String aName)
25.   {
26.     name = aName;
27.   }
28.
29.   //------------------------
30.   // INTERFACE
31.   //------------------------
32.
33.   public String getName()
34.   {
35.     return name;
36.   }
37.
38.   public void delete()
39.   {}
40.
41.
42.   public String toString()
43.   {
44.     return super.toString() + "["+
45.              "name" + ":" + getName()+ "]";
46.   }
47. }
```

Example (Modifying name)

```
class Person {
  immutable name;
  lazy int grade;

  Person rename(String newName) {
    Person p = new Person(newName);
    p.setGrade(grade);
    return p;
  }
}
```

WHAT IS NAME NOT AVAILABLE AT "CONSTRUCTION"

```
class Person {
  lazy immutable name;
}
```

```
  public Person()
  {
    canSetName = true;
  }

  //------------------------
  // INTERFACE
  //------------------------
  /* Code from template attribute_SetImmutable */
  public boolean setName(String aName)
  {
    boolean wasSet = false;
    if (!canSetName) { return false; }
    canSetName = false;
    name = aName;
    wasSet = true;
    return wasSet;
  }

  public String getName()
  {
    return name;
  }
```

READ-ONLY INTERFACE

PATTERN

CONTEXT

- You sometimes want certain privileged classes to be able to modify attributes of objects that are otherwise immutable

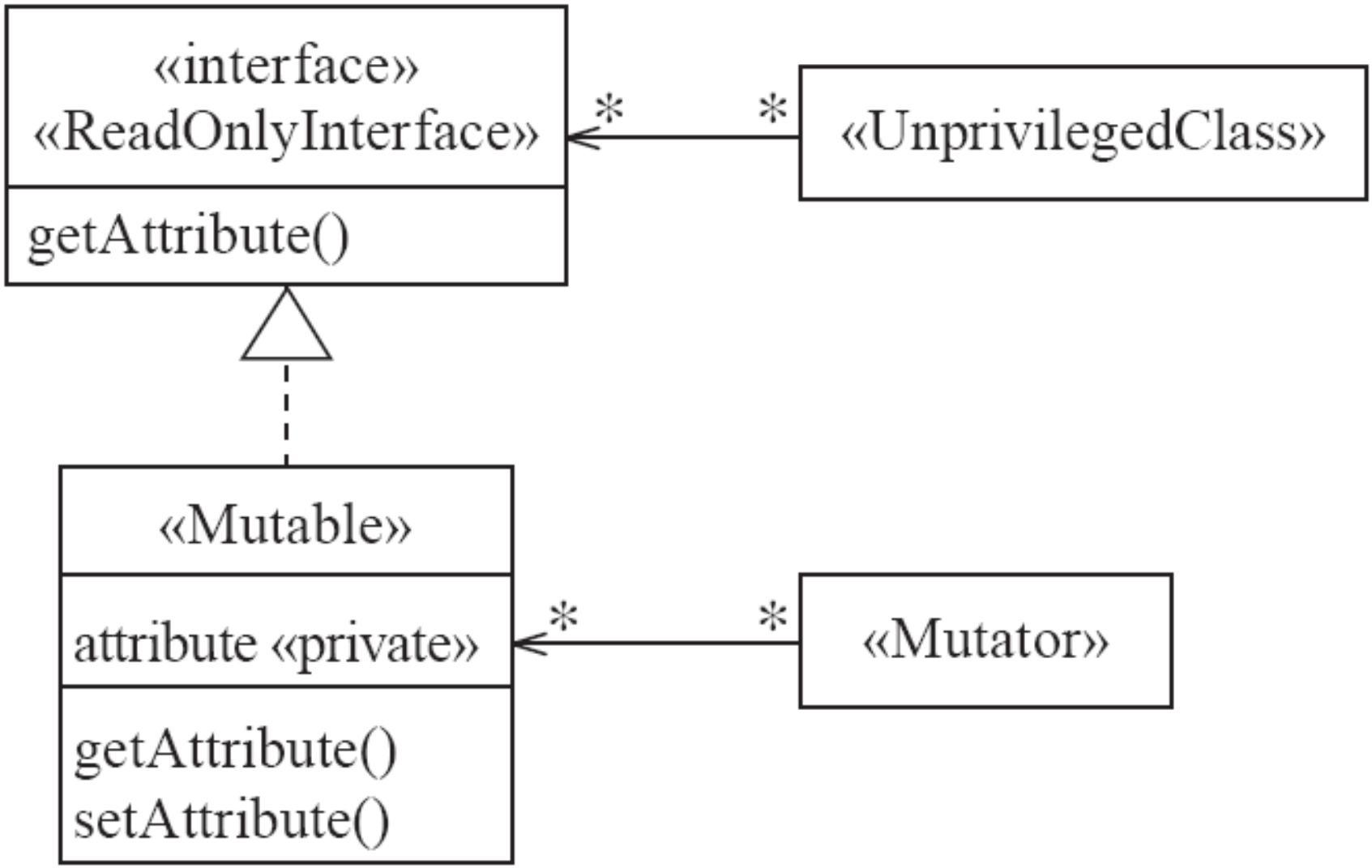
Problem

- How do you create a situation where some classes see a class as read-only whereas others are able to make modifications?

Forces

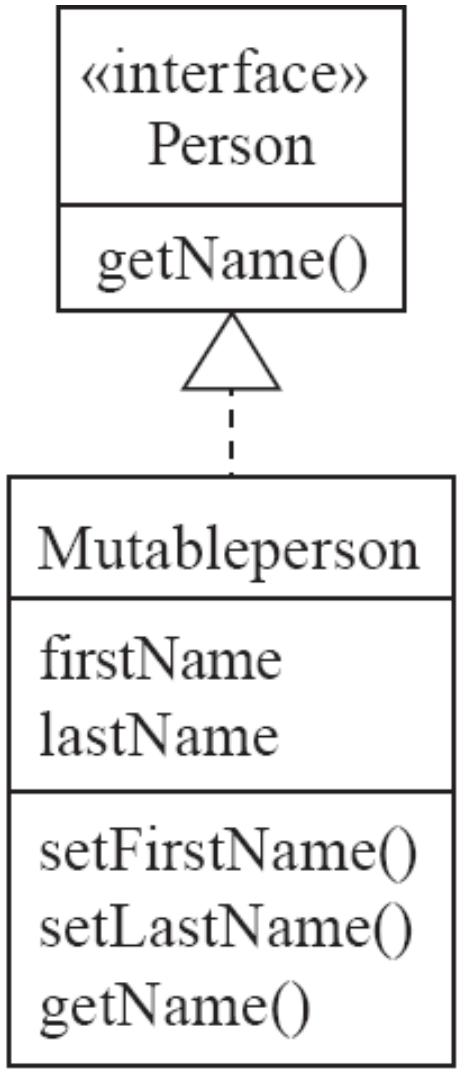
- Restricting access by using the **public**, **protected** and **private** keywords is not adequately selective.
- Making access **public** makes it public for both reading and writing

SOLUTION


«interface»
«ReadOnlyInterface»
getAttribute()
*
*
«UnprivilegedClass»
«Mutable»
attribute «private»
getAttribute()
setAttribute()
*
*
«Mutator»

EXAMPLE: PERSON

```
interface Person {
   String getName();
}

class MutablePerson {
  isA Person;
  firstName;
  lastName;
  name = { firstName + lastName };
}
```



```java
// line 6 "model.ump"
// line 16 "model.ump"
public class MutablePerson implements Person
{

  //------------------------
  // MEMBER VARIABLES
  //------------------------

  //MutablePerson Attributes
  private String firstName;
  private String lastName;

  public MutablePerson(String aFirstName, String aLastName)
  {
    firstName = aFirstName;
    lastName = aLastName;
  }

  //------------------------
  // INTERFACE
  //------------------------

  public boolean setFirstName(String aFirstName)
  {
    boolean wasSet = false;
    firstName = aFirstName;
    wasSet = true;
    return wasSet;
  }

  public boolean setLastName(String aLastName)
  {
    boolean wasSet = false;
    lastName = aLastName;
    wasSet = true;
    return wasSet;
  }

  public String getFirstName()
  {
    return firstName;
  }

  public String getLastName()
  {
    return lastName;
  }

  public String getName()
  {
    return firstName + lastName;
  }
```

Antipatterns

- Make the read-only class a subclass of the «Mutable» class
- Override all methods that modify properties
  - such that they throw an exception

PROXY

PATTERN

Context

- Often, it is time-consuming and complicated to create instances of a class (heavyweight classes).
- There is a time delay and a complex mechanism involved in creating the object in memory

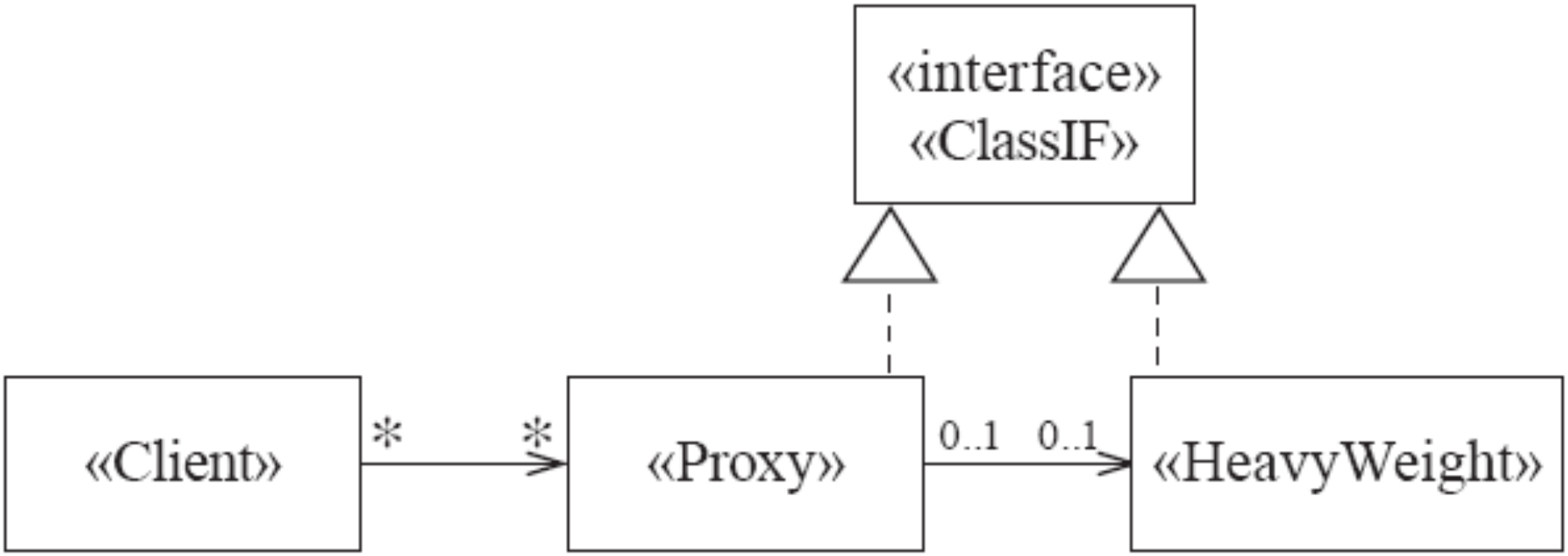
Problem

- How to reduce the need to create instances of a heavyweight class?

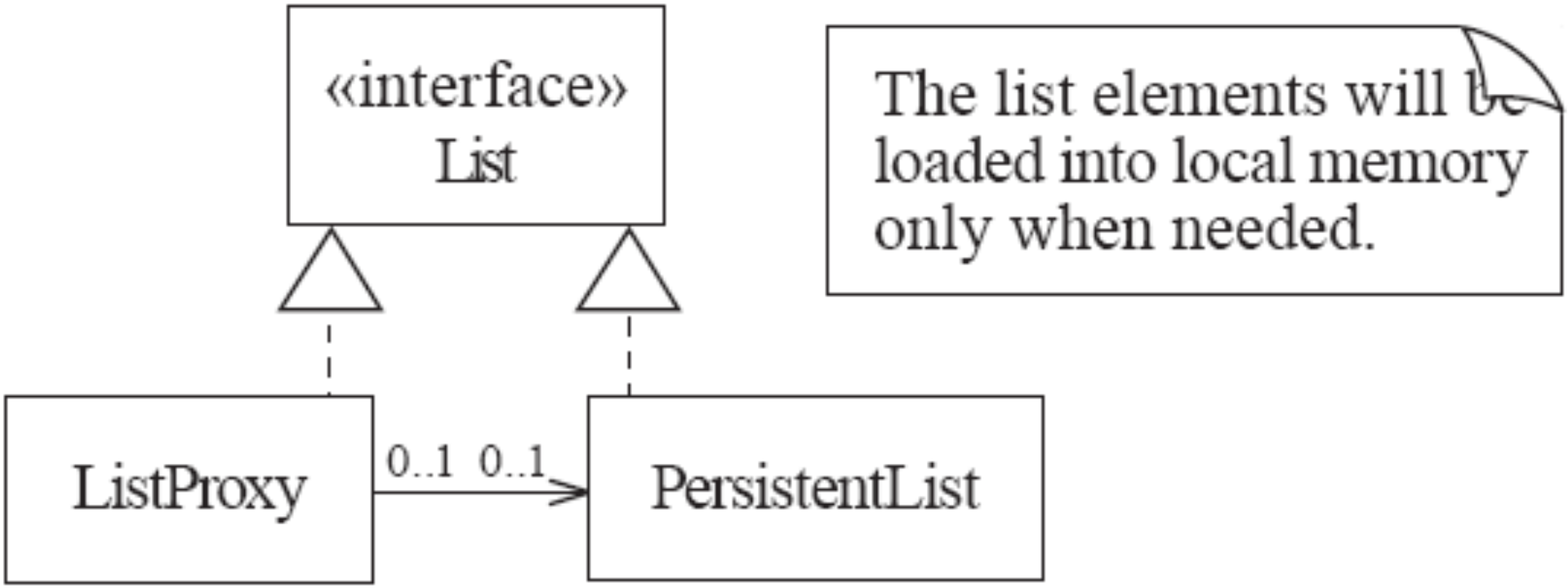
Forces

- We want all the objects in a domain model to be available for programs to use when they execute a system's various responsibilities.
- It is also important for many objects to persist from run to run of the same program

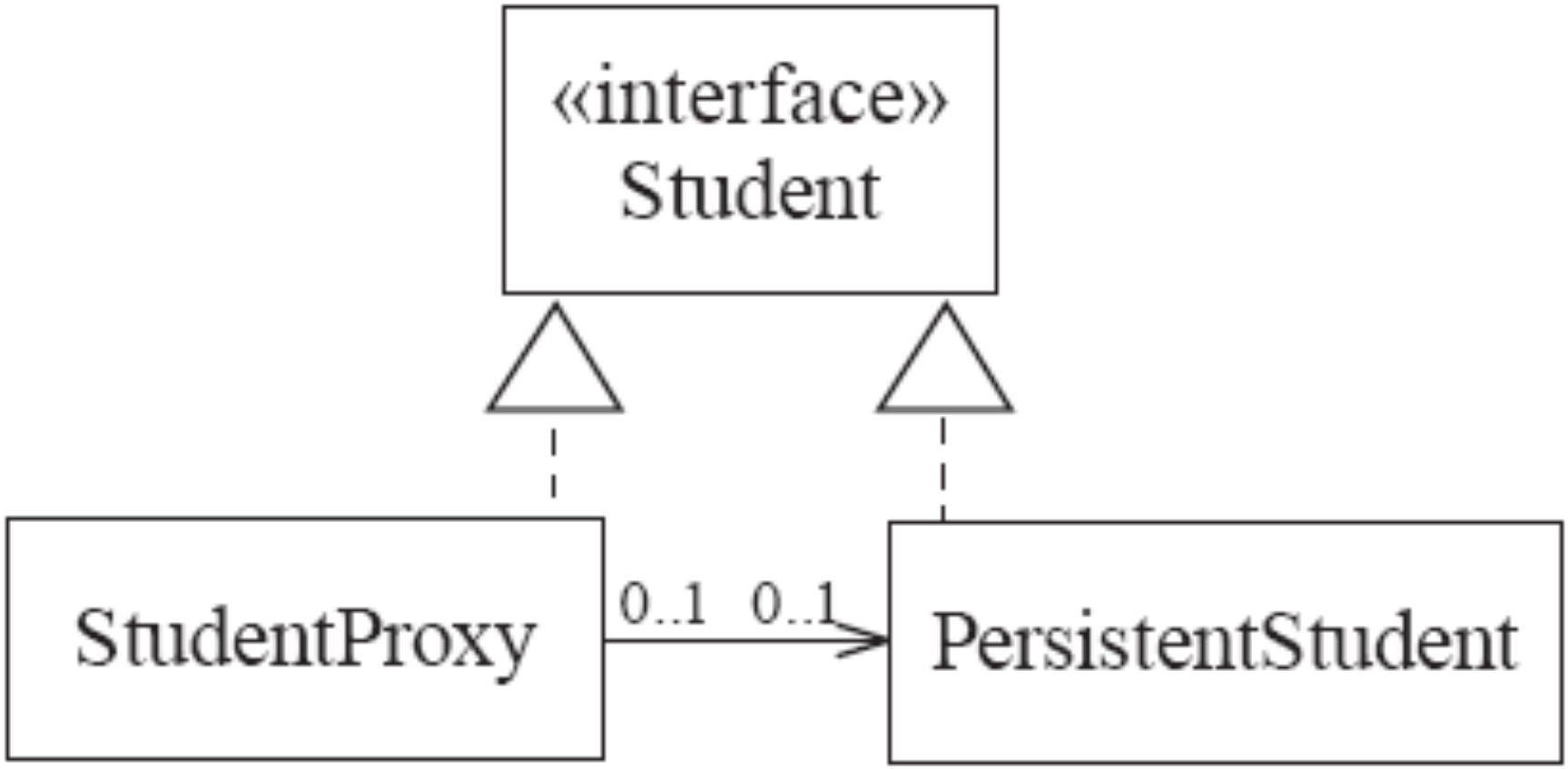
SOLUTION



Example: Lists

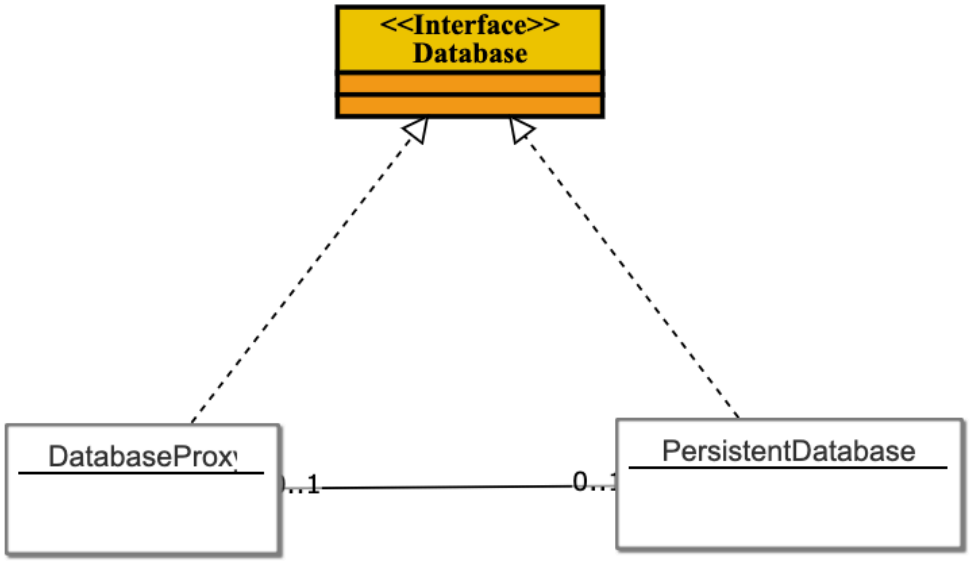


Example: Student

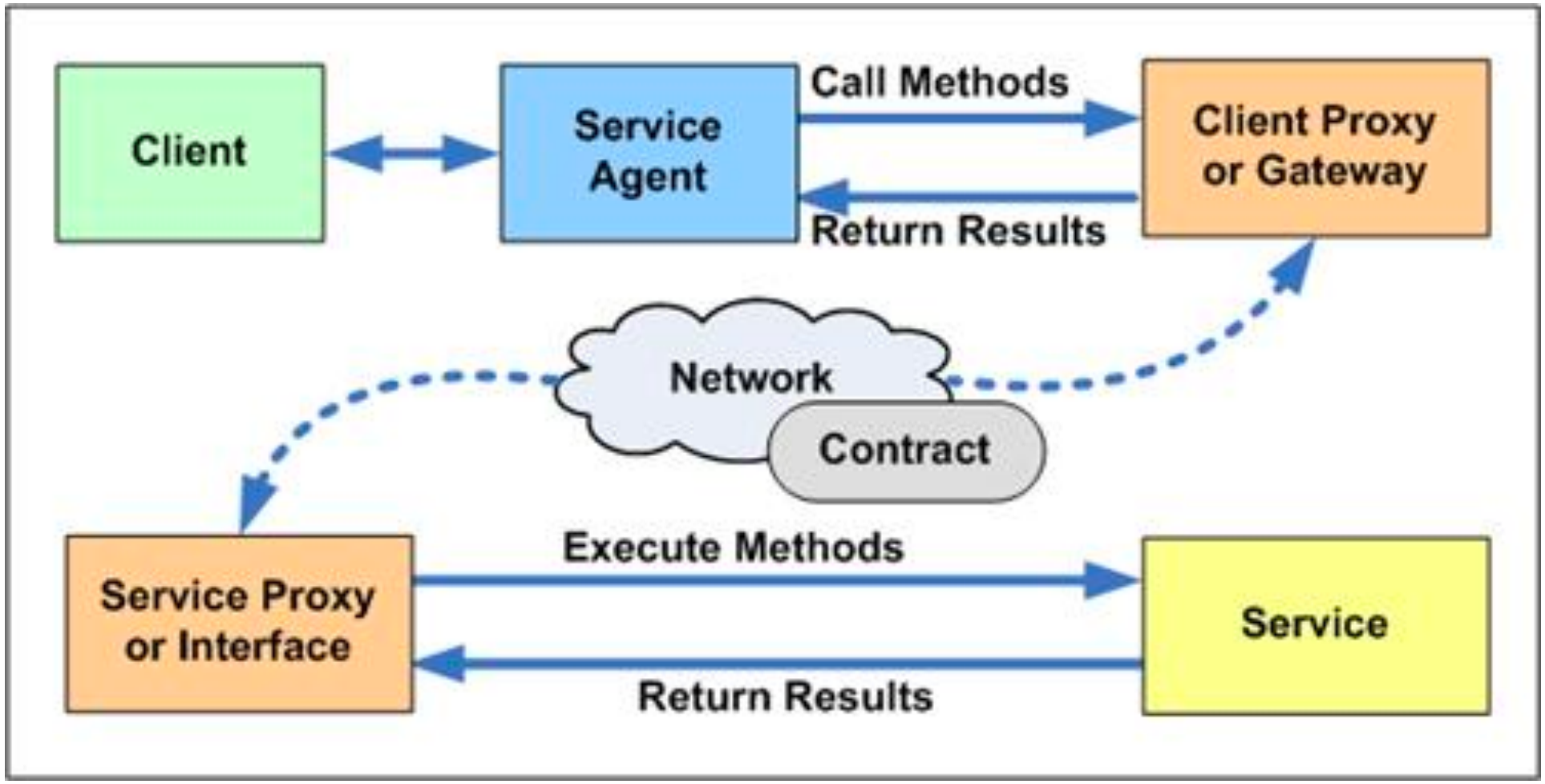


EXAMPLE: DATABASE ACCESS

```
interface Database {}

class DatabaseProxy {

  isA Database;

  0..1 -- 0..1 PersistentDatabase;

}

class PersistentDatabase {

  isA Database;

}
```



EXAMPLE: NETWORK CALLS



https://www.developerfusion.com/article/8307/aspnet-patterns-every-developer-should-know/

FACTORY

PATTERN

Context

- A reusable framework needs to create objecs; however the class of the created objects depends on the application.

Problem

- How do you enable a programmer to add new application-specific class into a system built on such a framework?

Forces

- We want to have the framework create and work with application-specific classes that the framework does not yet know about.

NOVEMBER 11, 2002 *by* JOEL SPOLSKY

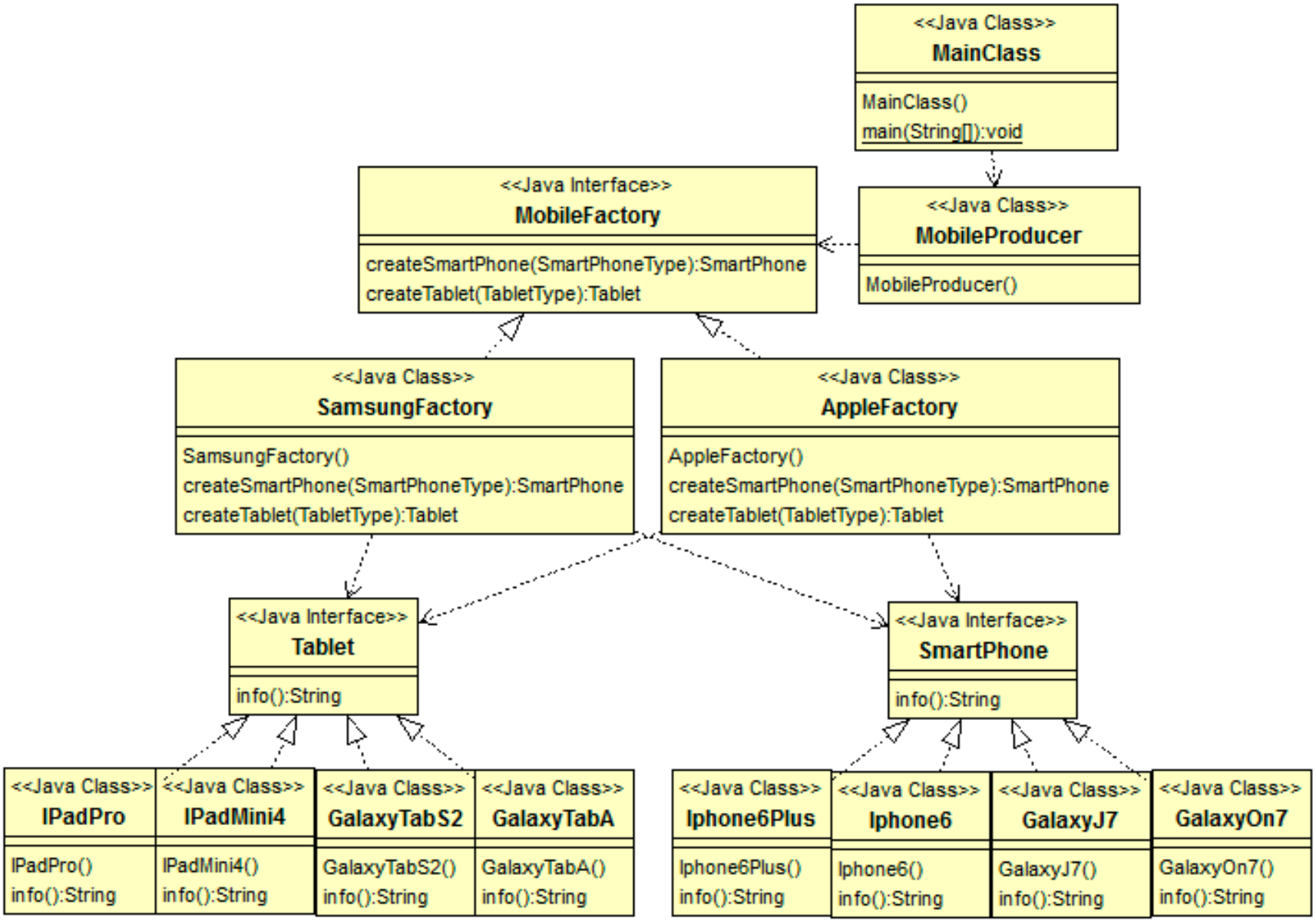
The Law of Leaky Abstractions

TOP 10, ROCK STAR DEVELOPER, NEWS

There's a key piece of magic in the engineering of the Internet which you rely on every single day. It happens in the TCP protocol, one of the fundamental building blocks of the Internet.

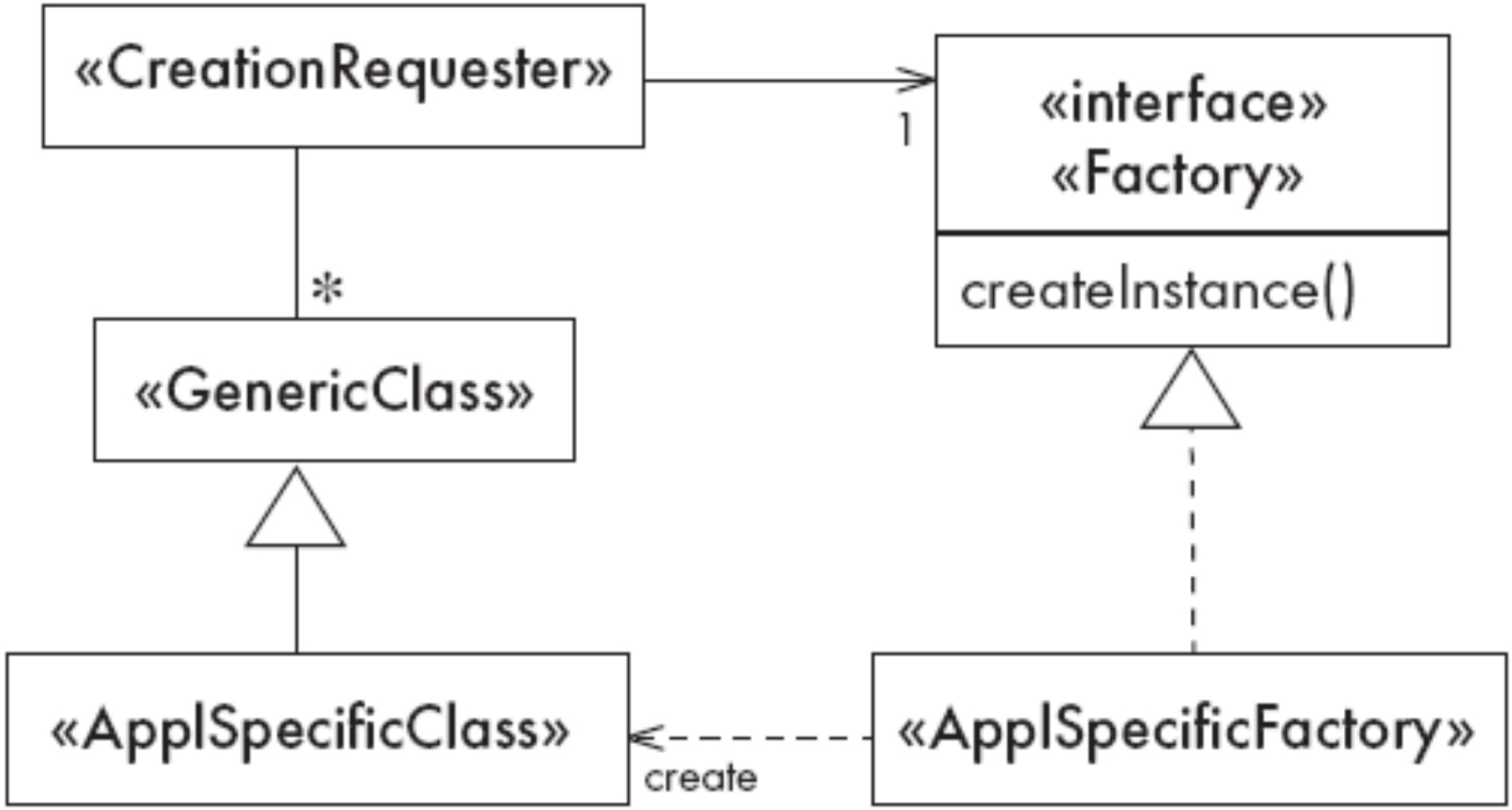
TCP is a way to transmit data that is *reliable*. By this I mean: if you send a message over a network using TCP, it will arrive, and it won't be garbled or corrupted.

https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/



https://grokonez.com/design-pattern/java-design-pattern-abstract-factory-pattern

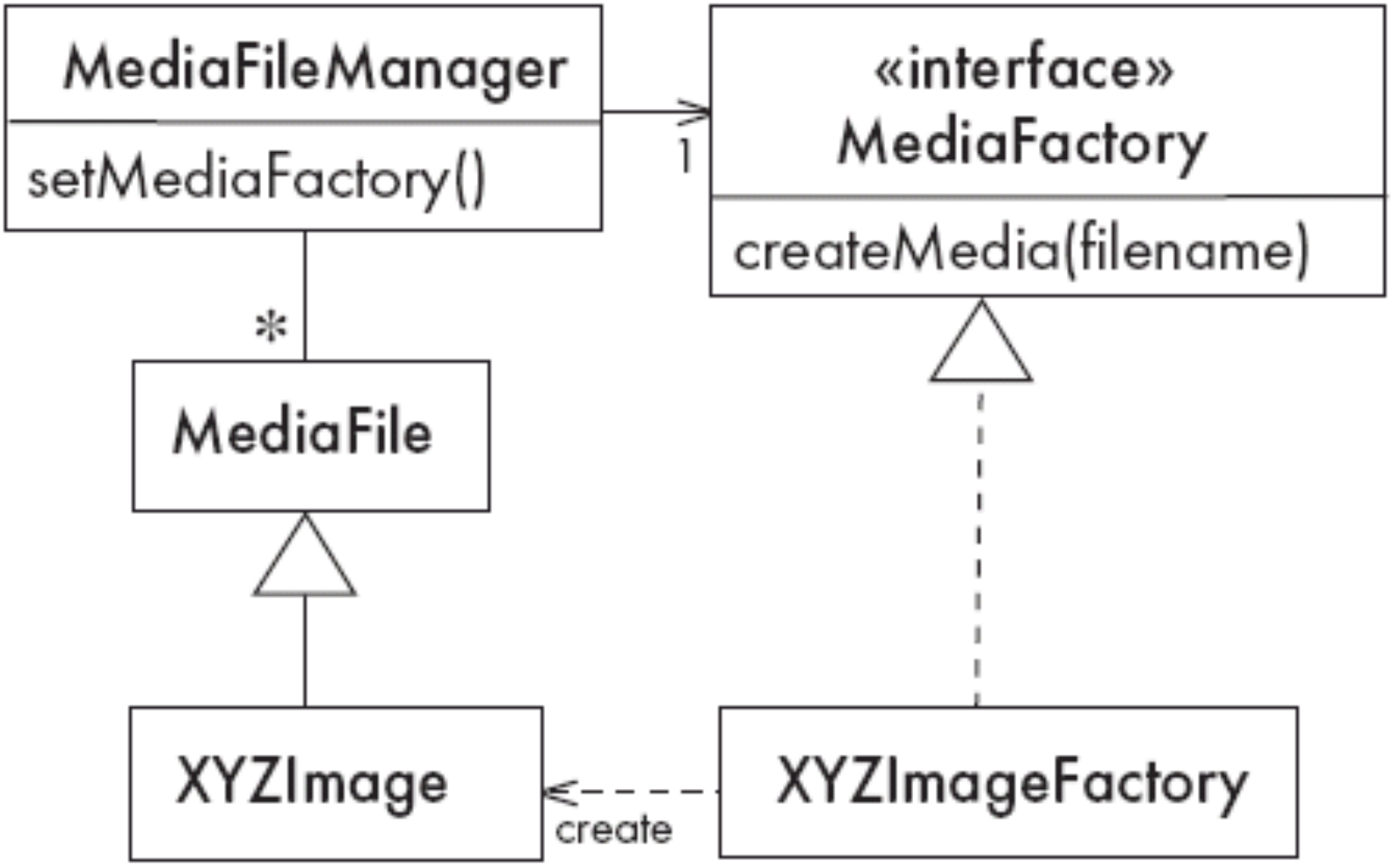
SOLUTION


«CreationRequester»
1
«interface»
«Factory»
createInstance()
*
«GenericClass»
«ApplSpecificClass»
create
«ApplSpecificFactory»

Solution

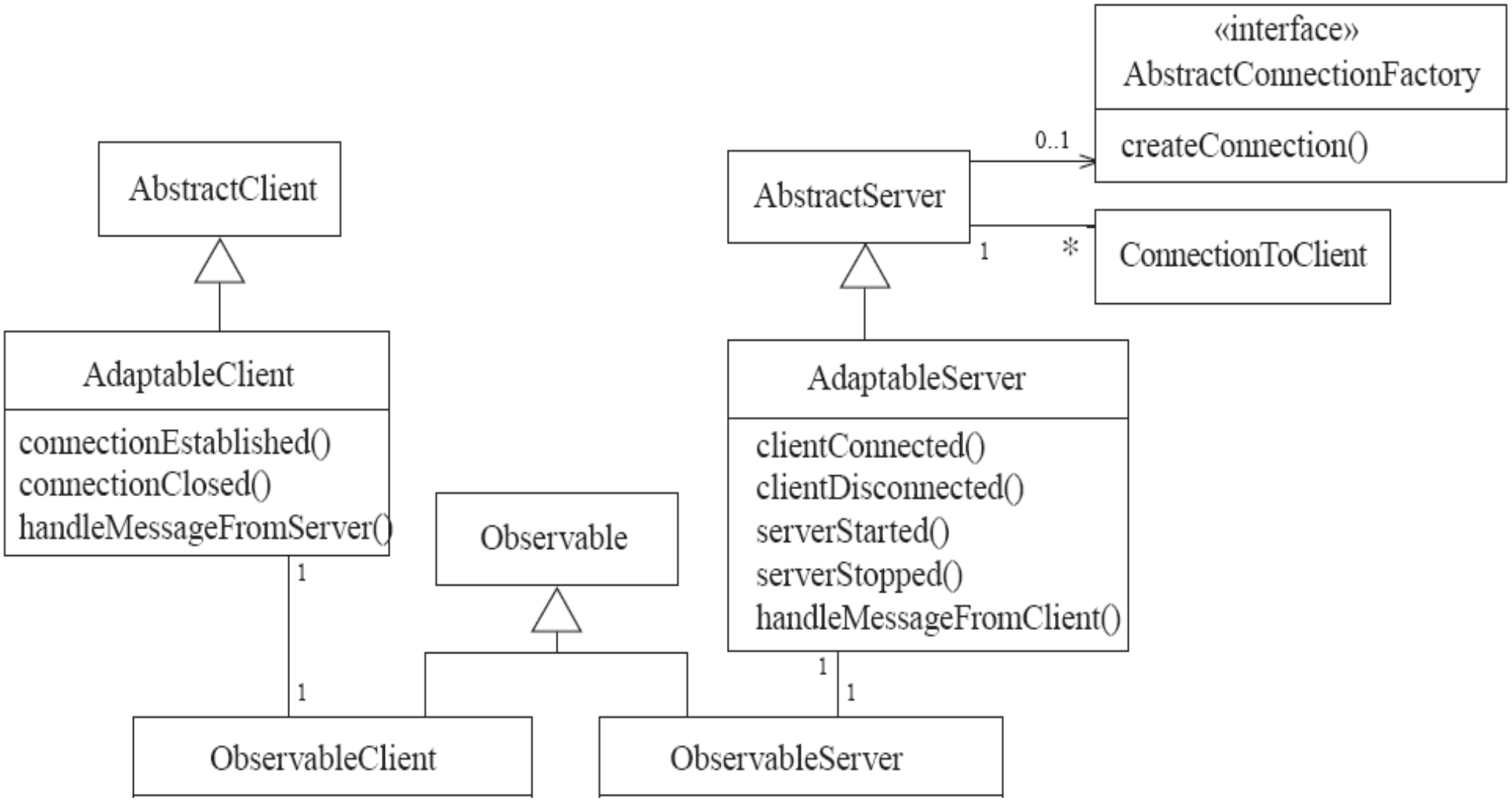
- The framework delegates the creation of application-specific classes to a specialized class, the Factory.
- The Factory is a generic interface defined in the framework.
- The factory interface declares a method whose purpose is to create some subclass of a generic class.

EXAMPLE: MEDIA

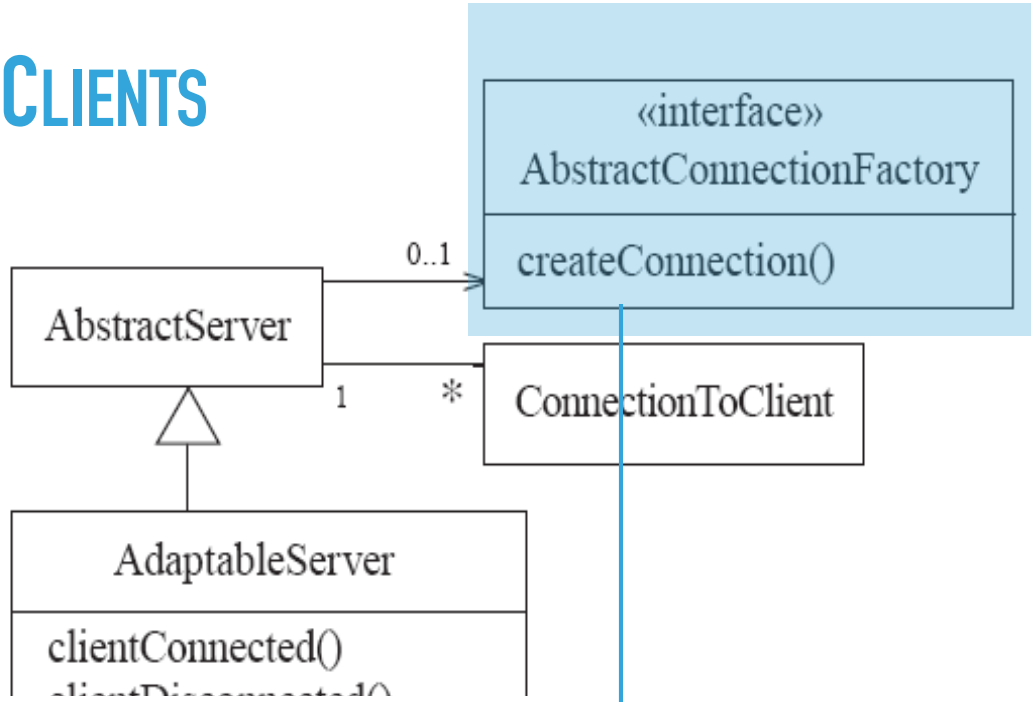


ENHANCING OCSF WITH

DESIGN PATTERNS


«interface»
AbstractConnectionFactory
createConnection()
0..1
AbstractClient
AbstractServer
1
*
ConnectionToClient
AdaptableClient
connectionEstablished()
connectionClosed()
handleMessageFromServer()
AdaptableServer
clientConnected()
clientDisconnected()
serverStarted()
serverStopped()
handleMessageFromClient()
Observable
1
1
1
1
ObservableClient
ObservableServer

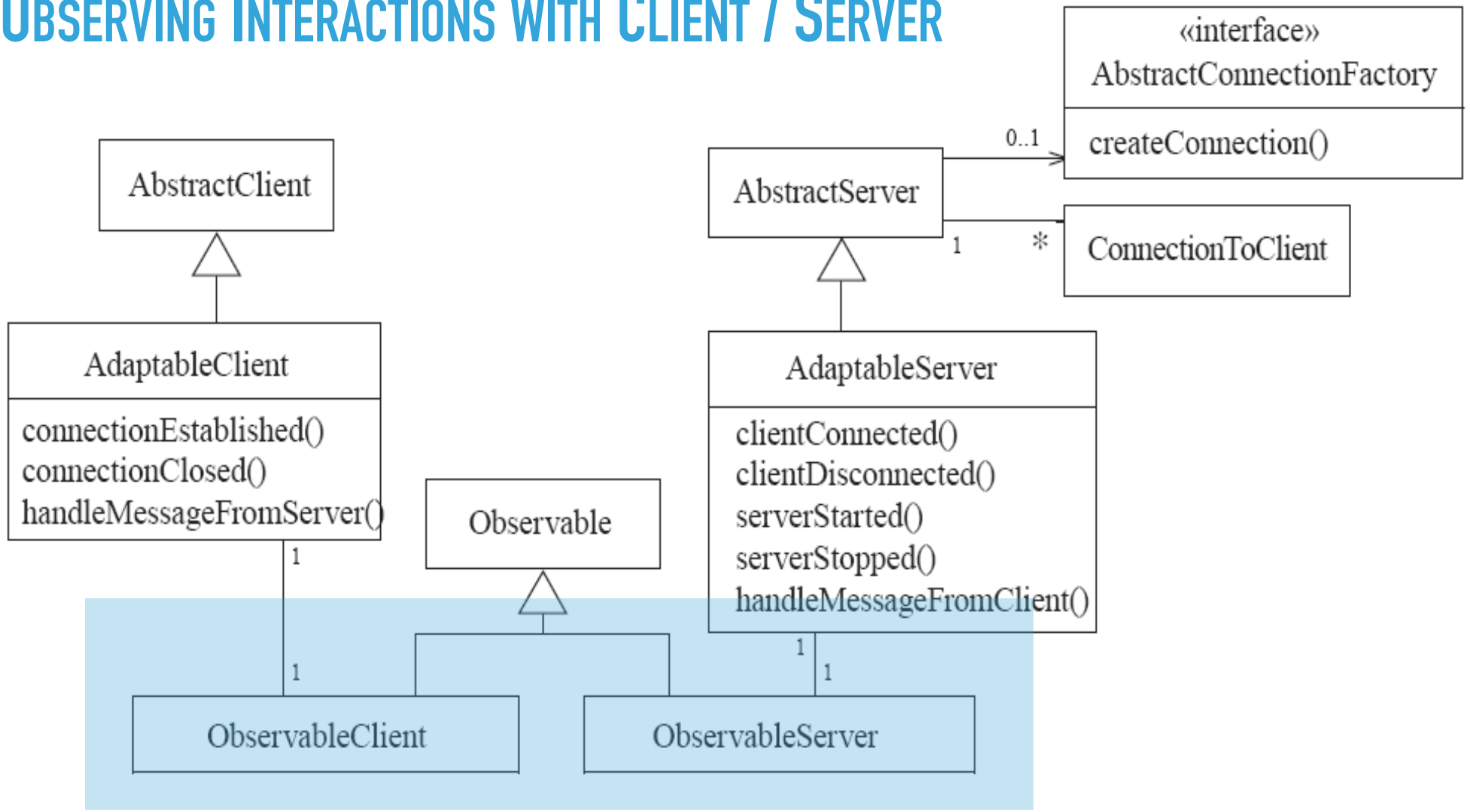
FACTORY PATTERN FOR CONNECTING CLIENTS



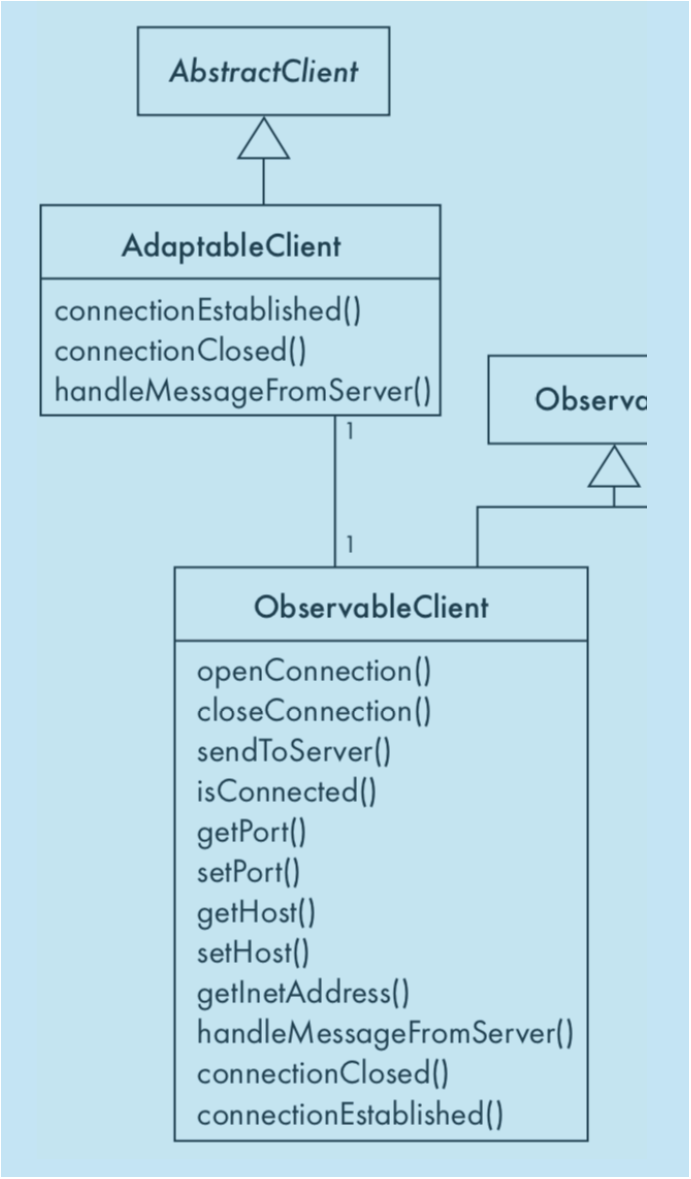

```
setConnectionFactory(new MyConnectionFactory());
```

```
protected ConnectionToClient createConnection(
   ThreadGroup group, Socket clientSocket,
   AbstractServer server) throws IOException
{
   return new Connection(group,clientSocket,server);
}
```

OBSERVING INTERACTIONS WITH CLIENT / SERVER



ADAPTING CLIENT / SERVER TO BE OBSERVABLE



DIFFICULTIES AND RISKS

When Creating Class Diargams

Patterns are not a panacea:

- Whenever you see an indication that a pattern should be applied, you might be tempted to blindly apply the pattern.
- This can lead to unwise design decisions

Resolution

- Always understand in depth the forces that need to be balanced, and when other patterns better balance the forces.
- Make sure you justify each design decision carefully

Developing patterns is hard

- Writing a good pattern takes considerable work.
- A poor pattern can be hard to apply correctly

Resolution

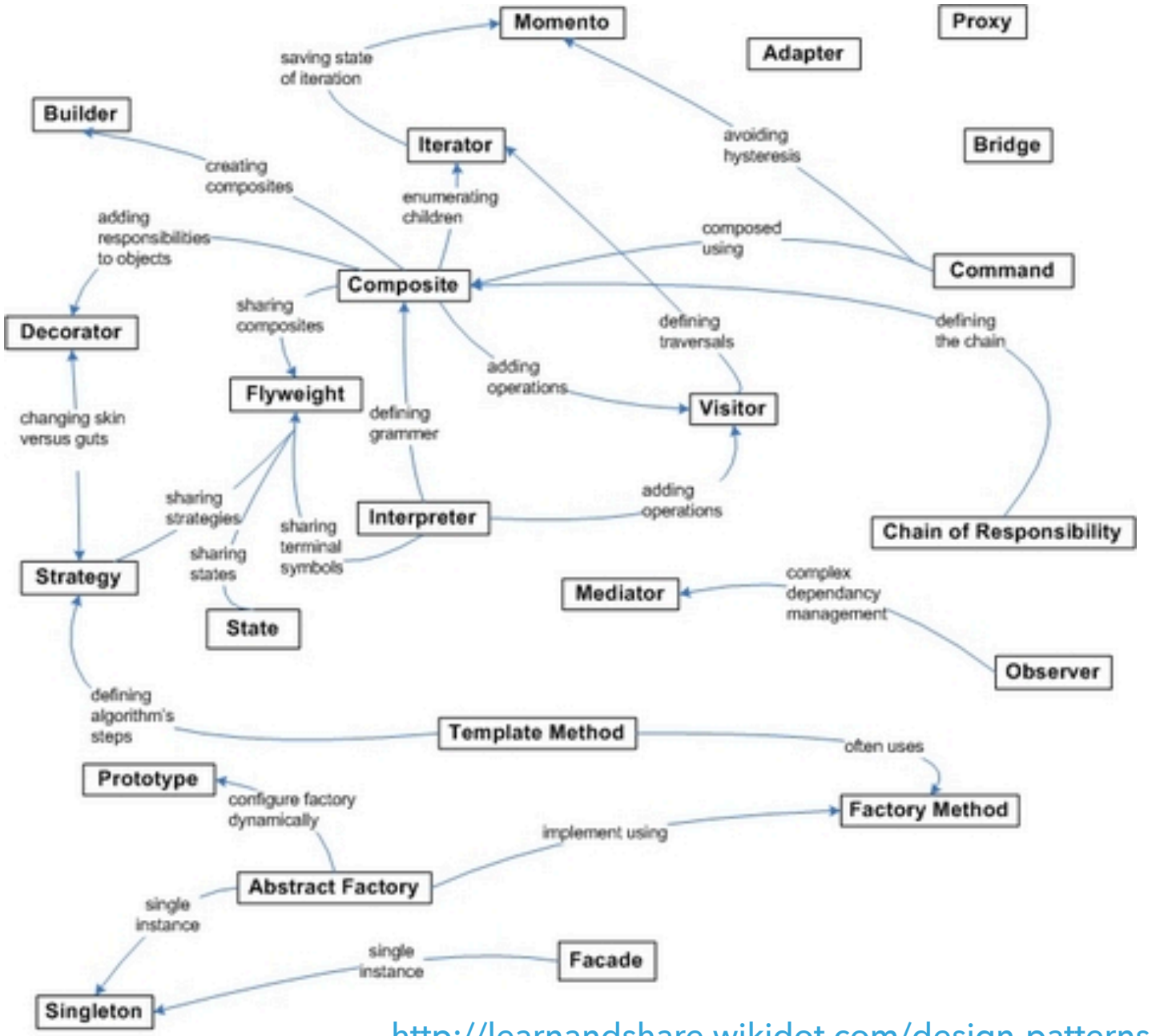
- Do not write patterns for others to use until you have considerable experience both in software design and in the use of patterns.
- Take an in-depth course on patterns.
- Iteratively refine your patterns, and have them peer reviewed at each iteration.

THE 23 GANG OF FOUR DESIGN PATTERNS

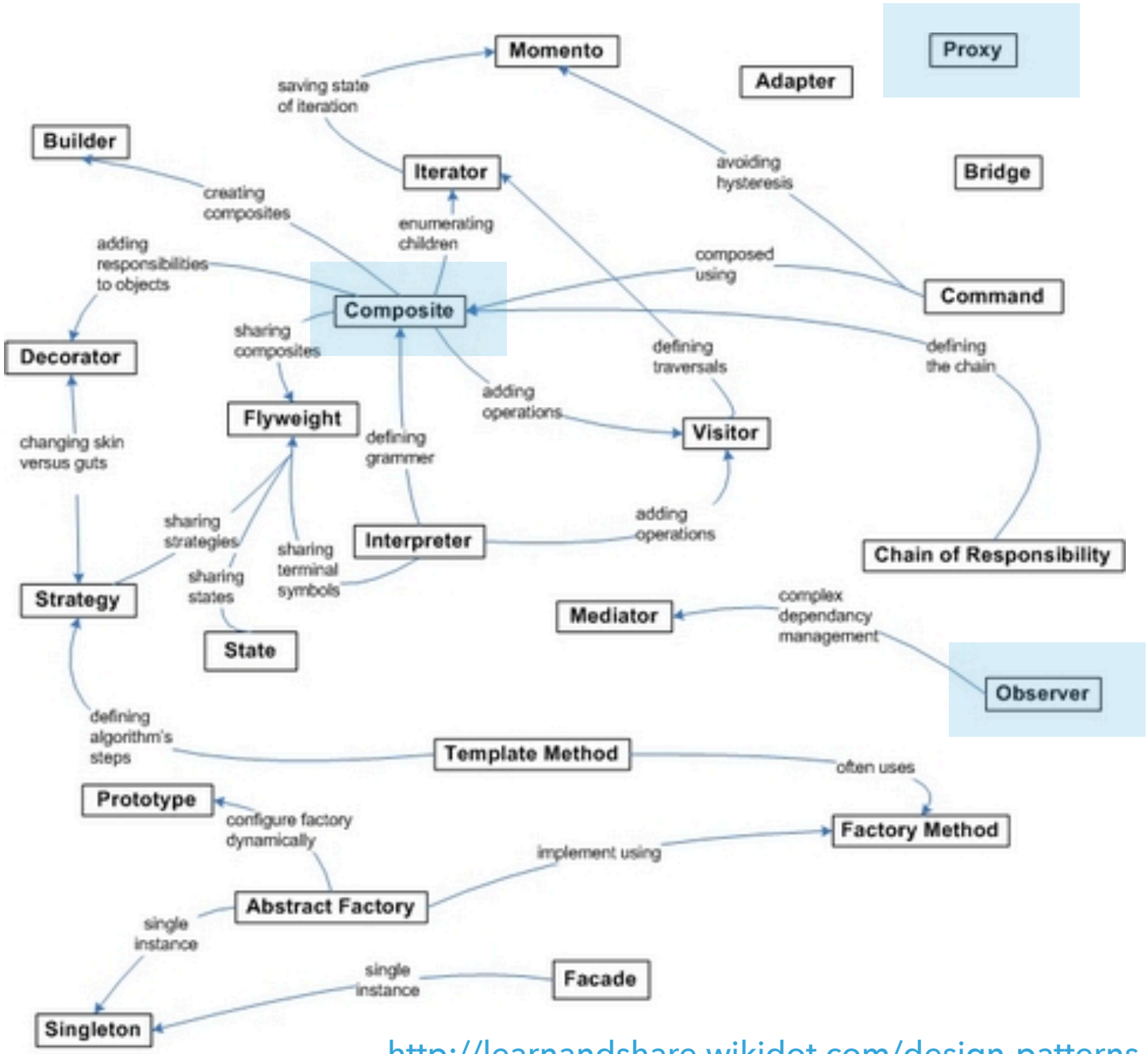
C	Abstract Factory	S	Facade	S	Proxy
S	Adapter	C	Factory Method	B	Observer
S	Bridge	S	Flyweight	C	Singleton
C	Builder	B	Interpreter	B	State
B	Chain of Responsibility	B	Iterator	B	Strategy
B	Command	B	Mediator	B	Template Method
S	Composite	B	Memento	B	Visitor
S	Decorator	C	Prototype		

https://www.script-tutorials.com/design-patterns-in-php/

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



http://learnandshare.wikidot.com/design-patterns



http://learnandshare.wikidot.com/design-patterns