Head First Design Patterns

by Eric Freeman, Elisabeth Freeman, Kathy Sierra, and Bert Bates

ch 0. Backgrounds

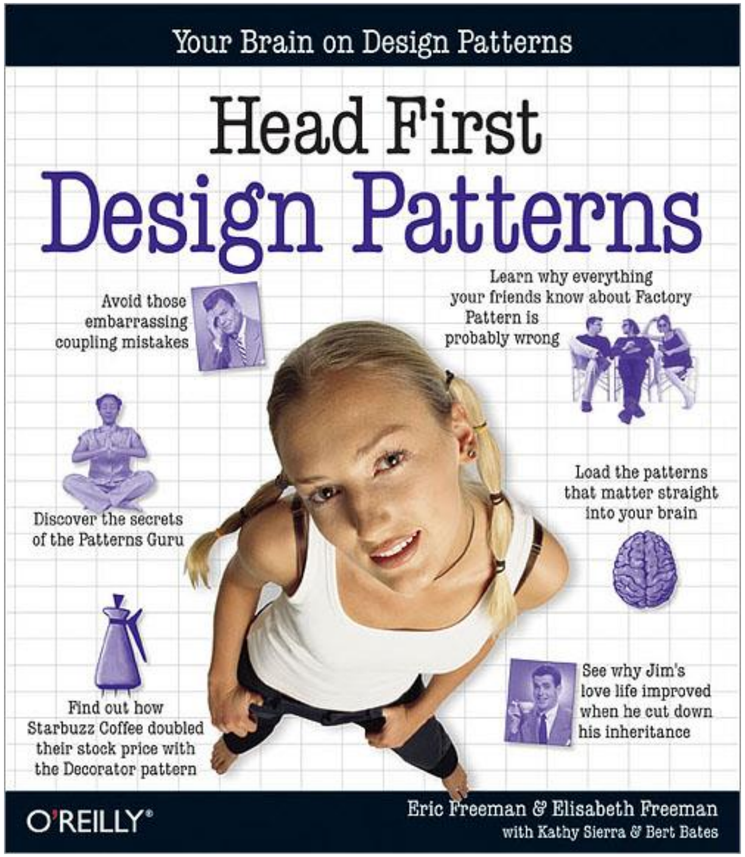
Contents

- **Chapter 0. Backgrounds**
  - 0.1 Introduction to Design Patterns
  - 0.2 Object Oriented Paradigm Review
  - 0.3 Core Concepts in Design
  - 0.4 UML Class Diagram Short Review for Design Patterns

0.1 Introduction to Design Patterns

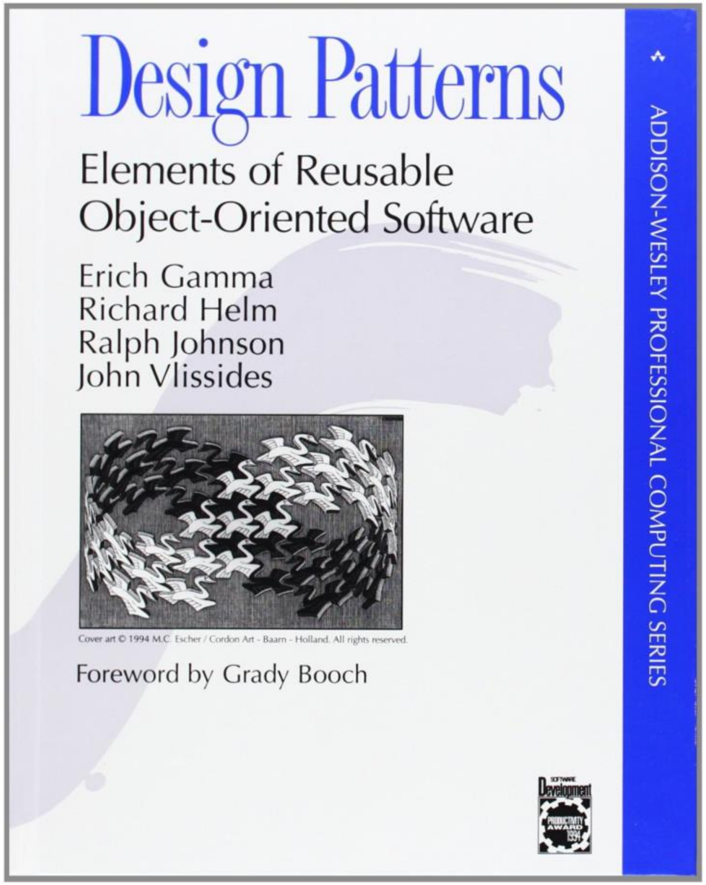
Main Textbook

- **Required**
  - Head First Design Patterns by Eric Freeman, Elisabeth Freeman, Kathy Sierra, and Bert Bates. O'Reilly


Your Brain on Design Patterns
Head First
Design Patterns
Avoid those embarrassing coupling mistakes
Learn why everything your friends know about Factory Pattern is probably wrong
Discover the secrets of the Patterns Guru
Load the patterns that matter straight into your brain
Find out how Starbuzz Coffee doubled their stock price with the Decorator pattern
See why Jim's love life improved when he cut down his inheritance
O'REILLY
Eric Freeman & Elisabeth Freeman
with Kathy Sierra & Bert Bates

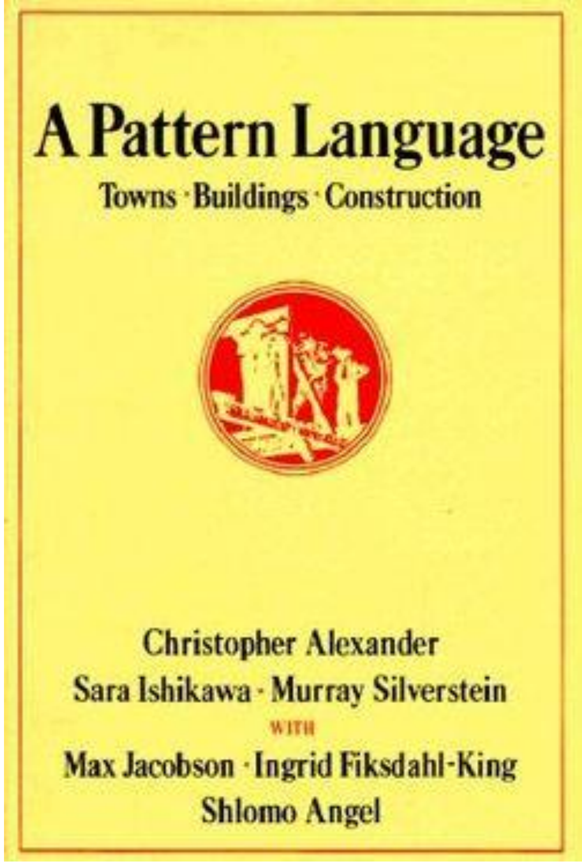
Reference

- **Recommended**
  - Design Patterns. Elements of Reusable Object-Oriented Software by Gamma, Helm, Johnson, and Vlissides (GoF). Addison-Wesley. 1995


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

What is a pattern?

- **Christopher Alexander, an architect,**
  - 건물과 도시 설계 프로세스를 개선하기 위한 방법을 연구
- **common definition:**
  - "A solution to a problem in a context"
- **나중에,**
  - 패턴을 소프트웨어 개발에도 적용할 수 있음을 인식함



A Pattern Language

Towns · Buildings · Construction

Christopher Alexander

Sara Ishikawa · Murray Silverstein

WITH

Max Jacobson · Ingrid Fiksdahl-King

Shlomo Angel

Software Patterns History

- **1977: The architect Christopher Alexander, A Pattern Language: Towns, Buildings, Construction**
- **1987: Kent Beck and Ward Cunningham, OOPSLA Paper**
  - Adopted Alexander's pattern idea for Smalltalk GUI design
- **1991: Erich Gamma, Ph. D. thesis**
- **1995: Gamma, Helm, Johnson, Vlissides (Gang of Four) Design Patterns: Elements of Reusable Object-Oriented Software**
- **1994-present: Pattern Languages of Programs (PLoP) Conferences and books**

Why do we use patterns?

- 객체 지향 소프트웨어를 설계하는 것은 어렵다, 재사용 가능한 객체 지향 소프트웨어를 설계하는 것은 더욱 어렵다. – Erich Gamma
- 경험 있는 설계자는 과거에 잘 작동했던 해결책을 재사용한다.
- 잘 구조화된 객체 지향 소프트웨어에는 클래스 또는 객체들의 반복된 패턴이 나타난다.
- 패턴을 아는 설계자는 더욱 생산적이며 설계 결과는 더 유연하고 재사용가능하다.
- "Someone has already solved your problems"

A Design Pattern

- **"A solution to a problem in a context"**
  - The **context** is the situation in which the pattern applies. This should be a **recurring** situation
  - The **problem** refers to the **goal** you are trying to achieve in this context, but it also refers to any **constraints** that occur in the context
  - The **solution** is **what you're after**: a general design that anyone can apply which resolves the goal and the set of constraints

Example

- **Iterator Pattern**
  - Context
    - You have a collection of objects
  - Problem
    - You need to step through the objects without exposing the collection's implementation
  - Solution
    - Encapsulate the iteration into a separate class

Design Patterns Categories

- **Creational**
  - Address problems of creating an object in a flexible way
  - **Separate creation** from operation/use
- **Structural**
  - Address problems of using OO constructs like inheritance to **organize classes and objects**
- **Behavioral**
  - Address problems of assigning responsibilities to classes
  - Suggest both static **relationships** and patterns of **communication**

Design Patterns Space

		Purpose		
		Defer object creation to another class or object	Describe ways to assemble objects	Describe algorithms and flow control
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter	Interpreter Template
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Command Iterator Mediator Memento Observer State Strategy Visitor

Key Features of Design Patterns

- **Pattern name**
  - Having a concise, meaningful name for a pattern improves communication among developers
- **Intent**
  - The purpose of the pattern
- **Problem**
  - Problem that the pattern is trying to solve
- **Solution**
  - How the pattern provides a solution to the problem in the context where it shows up
  - Emphasizes their relationships, responsibilities and collaborations; rather an abstract description
- **Consequences**
  - The pros and cons of using the pattern (includes impacts on reusability, portability, extensibility)
- **Implementation**
  - How the pattern can be implemented
    - Note: Implementations are just concrete manifestations of the pattern

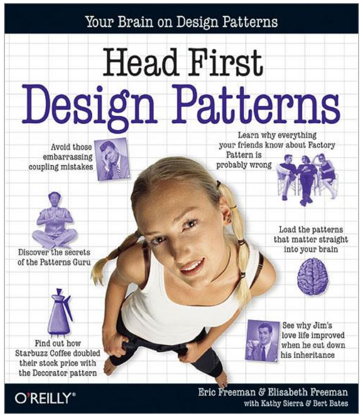
Benefits of Design Patterns

- 성공적인 설계를 재사용 하기 쉽게 해 줌
- 전문적인 설계 지식을 비전문가도 쉽게 접근할 수 있도록 표준 형식을 제공함
- 개발자들 사이의 의사 소통을 원할하게 해 줌
- 설계 수정을 용이하게 해 줌
- 설계 문서화를 개선해 줌
- 설계 이해도를 높여줌

Levels of Patterns

- **Architectual pattern**
  - expresses a fundamental structural organization or schema
  - provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them
  - affects the overall skeletal structure and organization of a software
- **Design pattern**
  - provides a scheme for refining the subsystems or components of a software, or the relationships between them.
  - describes commonly recurring structure of components that solves a general design problem within a particular context
  - does *not* influence overall system structure, but instead define *micro-architectures* of subsystems and components
- **Coding pattern (or programming idiom)**
  - low-level pattern specific to a programming language
  - describes how to implement particular aspects of components or the relationships between them using the features of the given language.

Design Pattern Books


Your Brain on Design Patterns

Head First Design Patterns

Avoid those embarrassing coupling mistakes

Learn why everything your friends know about Factory Pattern is probably wrong

Discover the secrets of the Patterns Guru

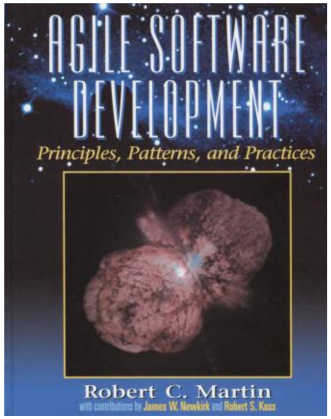
Load the patterns that matter straight into your brain

Find out how Starbuzz Coffee doubled their stock price with the Decorator pattern

See why Jim's love life improved when he cut down his inheritance

O'REILLY

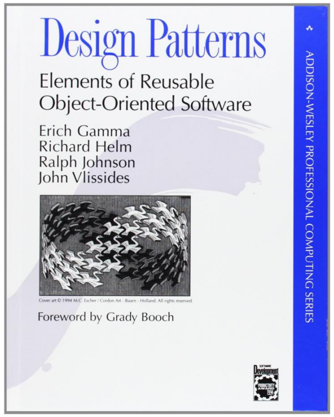
Eric Freeman & Elisabeth Freeman with Kathy Sierra & Bert Bates


AGILE SOFTWARE DEVELOPMENT

Principles, Patterns, and Practices

Robert C. Martin

with contributions by James W. Newkirk and Robert S. Koss

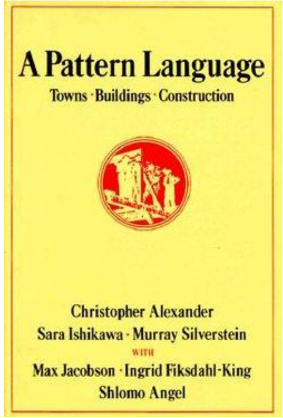

Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES


A Pattern Language

Towns · Buildings · Construction

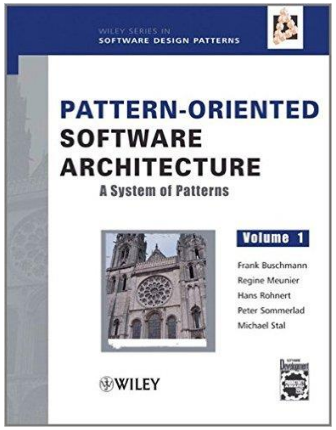
Christopher Alexander

Sara Ishikawa · Murray Silverstein

WITH

Max Jacobson · Ingrid Fiksdahl-King

Shlomo Angel


WILEY SERIES IN SOFTWARE DESIGN PATTERNS

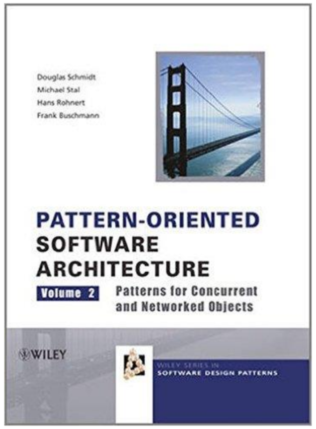
PATTERN-ORIENTED SOFTWARE ARCHITECTURE

A System of Patterns

Volume 1

Frank Buschmann
Regine Meunier
Hans Rohnert
Peter Sommerlad
Michael Stal

WILEY

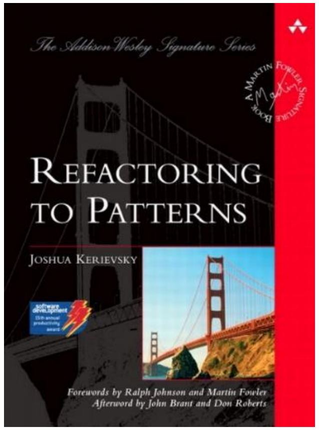

Douglas Schmidt
Michael Stal
Hans Rohnert
Frank Buschmann

PATTERN-ORIENTED SOFTWARE ARCHITECTURE

Volume 2 Patterns for Concurrent and Networked Objects

WILEY

WILEY SERIES IN SOFTWARE DESIGN PATTERNS


The Addison-Wesley Signature Series

A Martin Fowler Signature Book

REFACTORING TO PATTERNS

JOSHUA KERIEVSKY

Forewords by Ralph Johnson and Martin Fowler
Afterword by John Brant and Don Roberts

0.2 Object Oriented Paradigm Review

Introduction to Data Abstraction

- **An *abstract data type* is a user-defined data type that satisfies the following two conditions:**
  - The representation of, and operations on, objects of the type are defined in a single syntactic unit **single syntactic unit**

Weak Points of ADT

- **Reuse**
  - ADTs are difficult to reuse

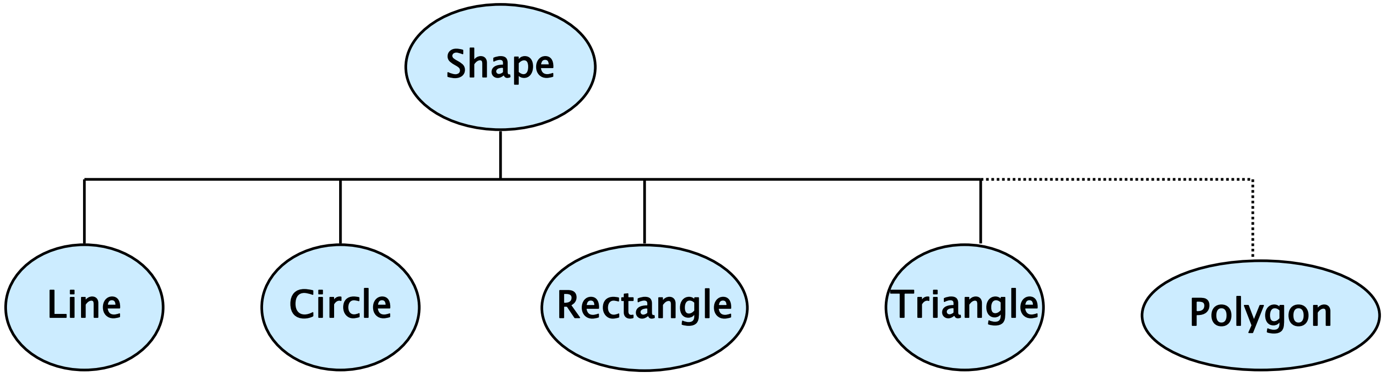
- **No levels**
  - All ADTs are independent and at the same level

Object-Oriented Paradigm

Design by Class

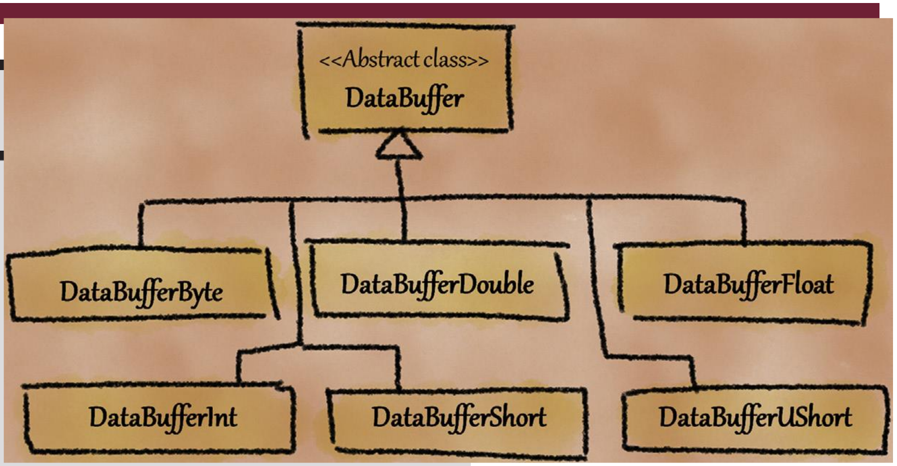
Class = ADT + Inheritance + Polymorphism

* Promotes reusability and flexibility

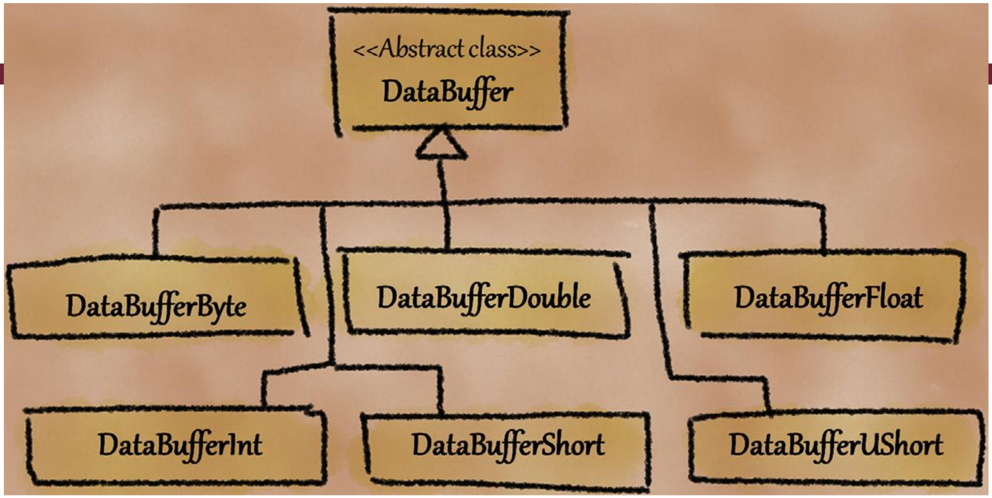


```
    protected int transferType;
```

```
switch (transferType) {
  case DataBuffer.TYPE_BYTE:
    byte bdata[] = (byte[])inData;
    pixel = bdata[0] & 0xff;
    length = bdata.length;
    break;
  case DataBuffer.TYPE_USHORT:
    short sdata[] = (short[])inData;
    pixel = sdata[0] & 0xffff;
    length = sdata.length;
    break;
  case DataBuffer.TYPE_INT:
    int idata[] = (int[])inData;
    pixel = idata[0];
    length = idata.length;
    break;
  default:
    throw new
    UnsupportedOperationException("This method has not been "+
    "implemented for transferType" + transferType);
}
```



- **Unexploited Encapsulation!**



부모 타입

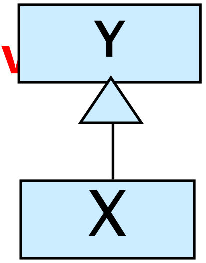
```
protected DataBuffer dataBuffer;
```

```
pixel = dataBuffer.getPixel();
length = dataBuffer.getSize();
```

- **Exploit Encapsulation and Polymorphism !**

Inheritance

- **When class X inherits from class Y**
  - X inherits all the methods of Y
    - Y의 인스턴스에 보낼 수 있는 모든 메시지는 X의 인스턴스에게도 보낼 수 있음
  - X inherits all the data from Y
    - Instances X have instances of all the data of Y
  - As a consequence we can say
    - X is a Y
  - Every instance of X is also an instance of Y
  - **We can use the instance of X wherev instance of type Y is expected**



Polymorphism

- The ability in computer programming to present the same programming interface for differing underlying forms (data types, classes)

- **Types of Polymorphism**
  - Runtime polymorphism (Dynamic polymorphism)
    - Method Overriding
  - Compile time polymorphism (Static polymorphism)
    - Method Overloading

Method Overloading

```
class X
{
   void methodA(int num)
   {
       System.out.println ("methodA:" + num);
   }
   void methodA(int num1, int num2)
   {
       System.out.println ("methodA:" + num1 + "," + num2);
   }
   double methodA(double num) {
       System.out.println("methodA:" + num);
       return num;
   }
}

public class test
{
   public static void main (String args [])
   {
       X Obj = new X();
       double result;
       Obj.methodA(20);
       Obj.methodA(20, 30);
       result = Obj.methodA(5.5);
       System.out.println("Answer is:" + result);
   }
}
```

```
methodA:20
methodA:20,30
methodA:5.5
Answer is:5.5
```

(Run-time) Polymorphism

- **A *polymorphic variable***
  - 한 클래스의 객체나 그 자손들의 객체를 가리킬 수 있는 변수

- 클래스 계층 구조에서 메소드들이 오버라이드된 경우, polymorphic 변수를 통해 메소드가 호출될 때 실제로 어떤 메소드와 바인딩될 지는 동적(run-time)으로 결정됨

- 개발 또는 유지 보수 시 시스템을 쉽게 확장할 수 있게 해줌

Polymorphism

- A property of object oriented software by which an operation (typically virtual) may be performed in different ways in different classes.
  - 동일한 이름을 가진 여러 개의 메소드가 존재함
  - 어떤 메소드가 실제로 실행될 지는 변수에 어떤 객체가 담겨있는지에 따라 결정됨
  - 많은 if-else or switch 문을 제거할 수 있게 해 줌

Method Overriding

```
public class X
{
    public void methodA() //Base class method
    {
        System.out.println ("hello, I'm methodA of class X");
    }
}

public class Y extends X
{
    public void methodA() //Derived Class method
    {
        System.out.println ("hello, I'm methodA of class Y");
    }
}
public class Z
{
   public static void main (String args []) {
       X obj1 = new X(); // Reference and object X
       X obj2 = new Y(); // X reference but Y object
       obj1.methodA();
       obj2.methodA();
   }
}
```

```
hello, I'm methodA of class X
hello, I'm methodA of class Y
```

Abstract Class and Abstract Method (in Java)

- An *abstract method* is one that does not include a definition (it only defines a protocol)
- A class defined with “abstract” keyword is an abstract class
- A class with at least one abstract method should become an abstract class (but not vice versa)
- An abstract class cannot be instantiated
- But, 추상 클래스 타입의 변수는 모든 자식 객체를 가리킬 수 있음

Polymorphism + Abstract class

```
abstract class Animal {
    protected String name;
    abstract public void say();
}

class Cat extends Animal {
    private void meow() { … }
    public void say() { meow(); }
}

abstract class Canine extends Animal {
    protected bool likeBones;
}

class Dog extends Canine {
    private void bark() { … }
    public void say() { bark(); }
}
```

```
Animal  a = null;
Dog  baduki = new Dog();
Cat  nabi = new Cat();

a = baduki; a.say();
a = nabi;  a.say();

Animal c1 = new Animal();
Animal c2 = new Cat();
Cat c3 = new Animal();
Cat c4 = new Cat();

Animal d1 = new Canine();
Animal d2 = new Dog();
Canine d3 = new Dog();
Canine d4 = new Cat();
Dog d5 = new Canine();
Dog d6 = new Dog();
```

Polymorphism + Abstract class

```
abstract class Animal {
     protected String name;
     abstract public void say();
}

class Cat extends Animal {
     private void meow() { … }
     public void say() { meow(); }
}

abstract class Canine extends Animal {
     protected bool likeBones;
}

class Dog extends Canine {
     private void bark() { … }
     public void say() { bark(); }
}
```

```
Animal  a = null;
Dog  baduki = new Dog();
Cat  nabi = new Cat();

a = baduki; a.say();
a = nabi;  a.say();

Animal c1 = new Animal(); //Compile Error!
Animal c2 = new Cat();
Cat c3 = new Animal(); //Compile Error!
Cat c4 = new Cat();

Animal d1 = new Canine(); //Compile Error!
Animal d2 = new Dog();
Canine d3 = new Dog();
Canine d4 = new Cat(); //Compile Error!
Dog d5 = new Canine(); //Compile Error!
Dog d6 = new Dog();
```

How a decision is made about which method to run

- Step 1: 현재 클래스에 구체적인 메소드가 있으면 그 메소드를 실행함

- Step 2: 그렇지 않으면, 바로 위 부모 클래스에 그 메소드가 있는지 찾아보고 있으면 그 메소드를 실행함

- Step 3: 그렇지 않으면, 실행할 구체적인 메소드를 찾을 때까지 Step 2를 반복함

- Step 4: 메소드를 찾지 못하면 오류가 발생함
  - In Java and C++ the program would not have compiled

What is an Interface?

- An interface is similar to an abstract class with the following exceptions:
  - 모든 메소드는 abstract임 (Java 8 부터는 default method와 static 메소드라는 구체적인 메소드를 가질 수 있음)
  - 인터페이스는 인스턴스 변수를 가질 수 없음
  - 단, 상수는 가질 수 있음(public static final variables)
- Interfaces are declared using the "interface" keyword
  - If an interface is public, it must be contained in a file which has the same name.
- 인터페이스는 추상 클래스보다 더 추상적임
- Interfaces are implemented by classes using the "implements" keyword.

Declaring an Interface

In Steerable.java:

```
public interface Steerable
{
  public void turnLeft(int degrees);
  public void turnRight(int degrees);
}
```

When a class "implements" an interface, the compiler ensures that it provides an implementation for all methods defined within the interface.

In Car.java:

```
public class Car extends Vehicle implements Steerable
{
  public int turnLeft(int degrees)
  {    ...            }
  public int turnRight(int degrees)
  {    ...                  }
  public otherMethods1()  {      ...                  }
  public otherMethods2()  {      ...                  }
     …
 }
```

Interfaces as Types

- 클래스가 정의되면, 컴파일러는 클래스를 하나의 새로운 타입으로 간주함

- 인터페이스도 컴파일러는 새로운 타입으로 간주함
  - 변수나 메소드 인자의 타입을 선언하는데 사용될 수 있음

Abstract Classes Versus Interfaces

- 언제 인터페이스 대신 추상 클래스를 사용하는가?
  - 서브 클래스와 슈퍼 클래스의 관계가 진정으로 “is a” 관계일 때

- 언제 추상 클래스 대신 인터페이스를 사용하는가?
  - 정의된 메소드들이 한 클래스의 일부분일 때
  - 이미 다른 클래스를 상속하고 있을 때
  - 어떠한 메소드의 구현도 제공하지 못 하는 경우

Polymorphism with Interface

```
abstract class Animal {
    protected String name;
    abstract public void say();    }

class Cat extends Animal implements Sayable {
    private void meow() { … }
    public void say() { meow(); }    }

abstract  class Canine extends Animal {
    protected boolean likeBones;  }

class Dog extends Canine implements Sayable {
    private void bark() { … }
    public void say() { bark(); }    }

class Robot implements Sayable {
    private void printOut() { … }
    public void say() { printOut(); }  }

interface Sayable {
 public void say(); }
```

```
Dog  baduki = new Dog();
Cat  nabi = new Cat();
Robot robo = new Robot();

Animal  aref = null;
Sayable sref = null;
Canine cref = null;

aref = baduki; aref.say();
aref = nabi; aref.say();
aref = robo; aref.say();

sref = baduki; sref.say();
sref = nabi; sref.say();
sref = robo; sref.say();

cref = baduki; cref.say();
cref = nabi; cref.say();
cref = robo; cref.say();
```

Polymorphism with Interface

```
abstract class Animal {
    protected String name;
    abstract public void say();    }

class Cat extends Animal implements Sayable {
    private void meow() { … }
    public void say() { meow(); }    }

abstract  class Canine extends Animal {
    protected boolean likeBones;  }

class Dog extends Canine implements Sayable {
    private void bark() { … }
    public void say() { bark(); }    }

class Robot implements Sayable {
    private void printOut() { … }
    public void say() { printOut(); }   }

interface Sayable {
 public void say(); }
```

```
Dog  baduki = new Dog();
Cat  nabi = new Cat();
Robot robo = new Robot();

Animal  aref = null;
Sayable sref = null;
Canine cref = null;

aref = baduki; aref.say();
aref = nabi; aref.say();
aref = robo; aref.say();  // Compile Error!

sref = baduki; sref.say();
sref = nabi; sref.say();
sref = robo; sref.say();

cref = baduki; cref.say();
cref = nabi; cref.say(); // Compile Error!
cref = robo; cref.say(); // Compile Error!
```

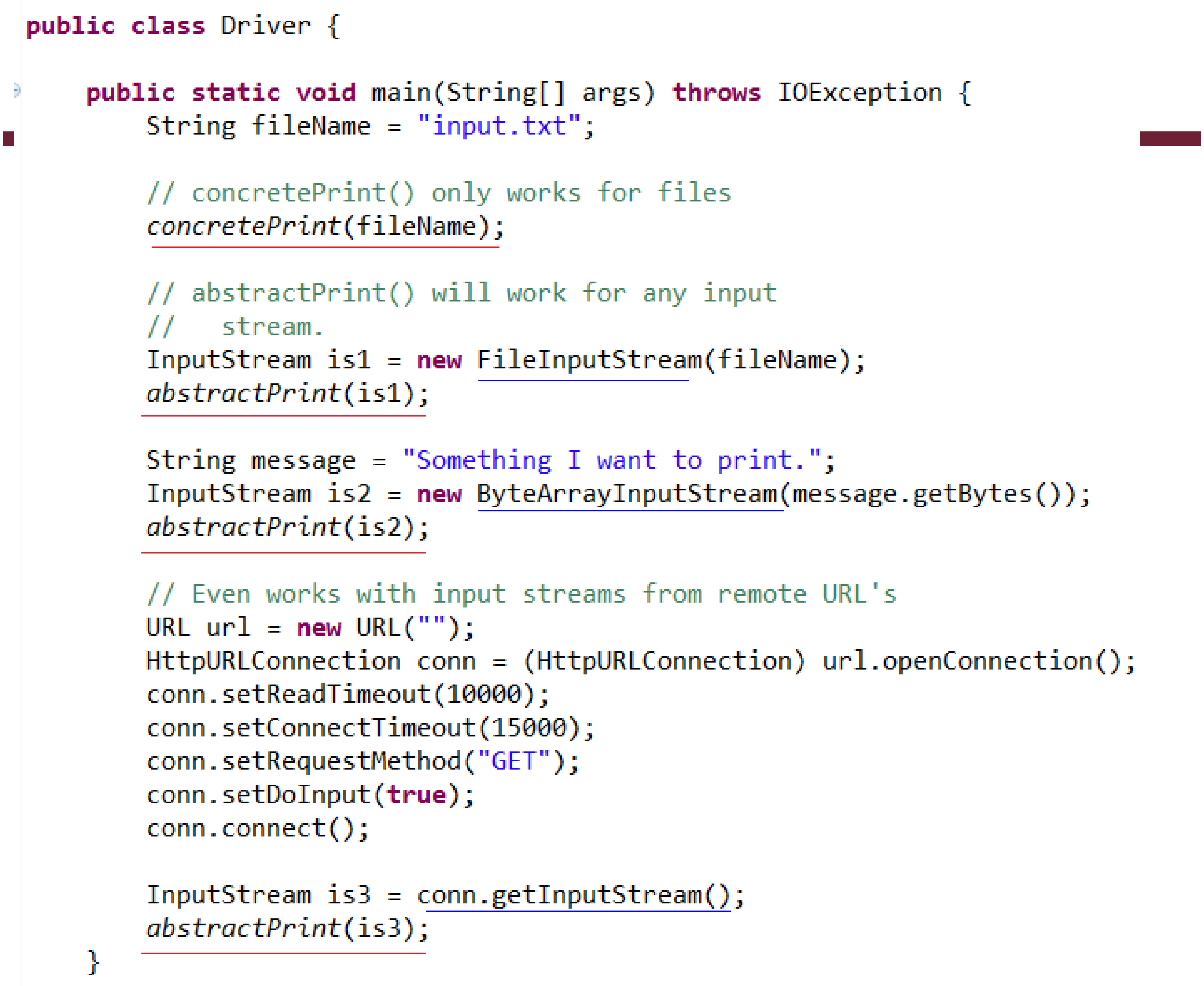
0.3 Core Concepts in Design

Core Concepts in Design

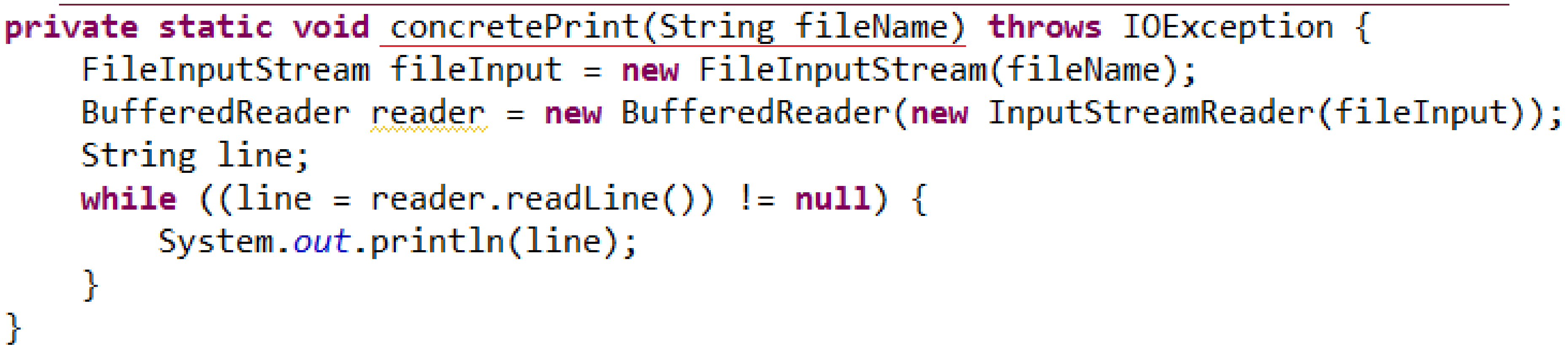
- Abstraction
- Information hiding
- Separation of concerns
- Interfaces
- Modularity
- Divide and conquer

Abstraction

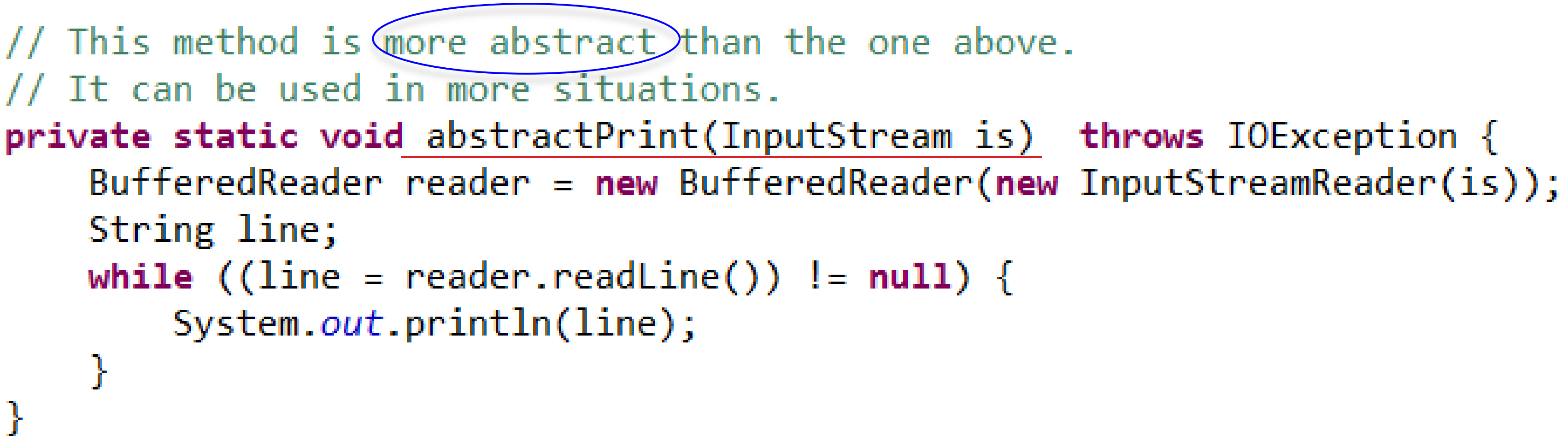
- Hiding details
  - 복잡한 소프트웨어 설계를 단순화하고 관리하기 쉽게 하기 위해
- 문제와 관련된 카테고리와 개념을 구체적인 구현과 분리시켜 줌
  - It means that code can be written so that
    - 구체적인 세부사항에 의존적이지 않게 함(e.g. supporting applications, operating system software or hardware)
    - 추상적인 개념에 의존하도록 함으로써 최소한의 작업으로 수정이나 확장을 가능하게 함



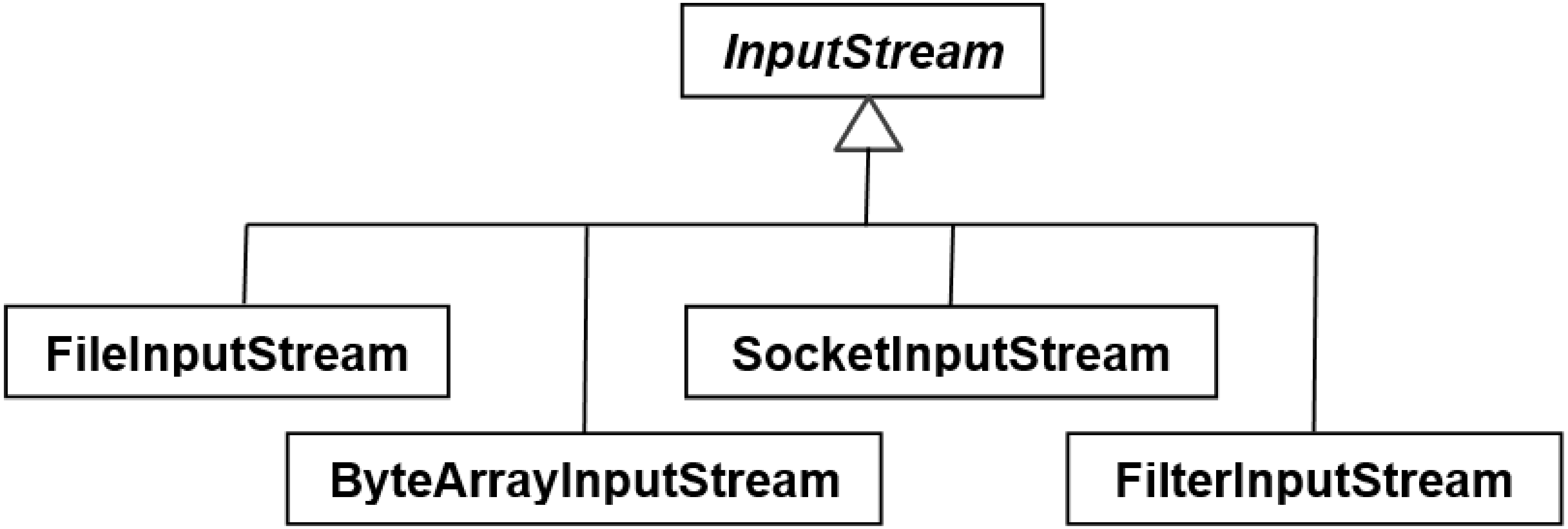
```
public class Driver {

    public static void main(String[] args) throws IOException {
        String fileName = "input.txt";

        // concretePrint() only works for files
        concretePrint(fileName);

        // abstractPrint() will work for any input
        //   stream.
        InputStream is1 = new FileInputStream(fileName);
        abstractPrint(is1);

        String message = "Something I want to print.";
        InputStream is2 = new ByteArrayInputStream(message.getBytes());
        abstractPrint(is2);

        // Even works with input streams from remote URL's
        URL url = new URL("");
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setReadTimeout(10000);
        conn.setConnectTimeout(15000);
        conn.setRequestMethod("GET");
        conn.setDoInput(true);
        conn.connect();

        InputStream is3 = conn.getInputStream();
        abstractPrint(is3);
    }
```



```java
private static void concretePrint(String fileName) throws IOException {
    FileInputStream fileInput = new FileInputStream(fileName);
    BufferedReader reader = new BufferedReader(new InputStreamReader(fileInput));
    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
}
```



```java
// This method is more abstract than the one above.
// It can be used in more situations.
private static void abstractPrint(InputStream is)  throws IOException {
    BufferedReader reader = new BufferedReader(new InputStreamReader(is));
    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
}
```



InputStream is an abstract class with several concrete subclasses!

Information hiding

- 한 모듈의 알고리즘이나 자료 구조와 관련된 내부 설계 결정을 외부로부터 숨김
  - 내부 설계 결정은 변경될 가능성이 큼
  - 향후 유지보수나 수정 시 side effect를 줄여 줌

- Information hiding : Encapsulation

  = Principle : Technique

Evaluate w.r.t. information hiding

```
class Course {
    private Set students;

    public Set getStudents() {
        return students;
    }
    public void setStudents(Set s) {
        students = s;
    }
}
```

Evaluate w.r.t. information hiding

```
class Course {
    private Set students;

    public Set getStudents() {
        return Collections.
               unmodifiableSet(students);
    }
    public void addStudent(Student student) {
        students.add(student);
    }
    public void removeStudent(Student student) {
        students.remove(student);
    }
}
```

Separation of Concerns (SoC)

- 하나의 컴퓨터 프로그램을 각각의 관심사를 해결하는 여러 개의 section으로 분리하자는 설계 원칙
- A concern is a set of information that affects the code of a computer program.
- A program that embodies SoC well is called modular.
- Modularity, and hence separation of concerns, is achieved by encapsulating information inside a section of code that has a well-defined interface.
  - cf) Aspect-oriented Programming (AOP)

Interfaces

- 인터페이스는 모듈들이 서로 의사 소통하는 접점이다
- 추상화 과정에서 예상 입력과 출력을 명확하게 기술하는 인터페이스를 잘 정의해야 함
- 캡슐화가 잘 진행되면, 한 객체는 인터페이스를 통해서만 다른 객체로부터 접근가능함
  - Some programming languages explicitly support interfaces (C#, Java, Objective-C, PHP, etc.)
  - In C++, interfaces are known as abstract base classes and implemented using pure virtual functions.

Modularity

- 하나의 큰 소프트웨어 시스템을 연결된 작은 모듈들로 나눔
- 모듈들은 인터페이스를 통해 서로 연결됨
- Interconnection은 가능한 단순하게 해서 유지 보수 시 side effect를 줄여야 함
    - directly connected: one module can call the other module.
    - indirectly connected: they share common files or global data structures.

Modularity

- Goal of design
  - 시스템을 모듈들로 분할하고 각 모듈에 책임을 할당함
  - 응집도(cohesion)는 높이고 결합도(coupling)는 낮추어야 함

- Modularity는 프로그래머가 다루어야 할 복잡도를 낮춰 줌

- 응집도와 결합도는 설계 결과물을 평가하는 데 있어서 가장 중요한 설계 원칙임

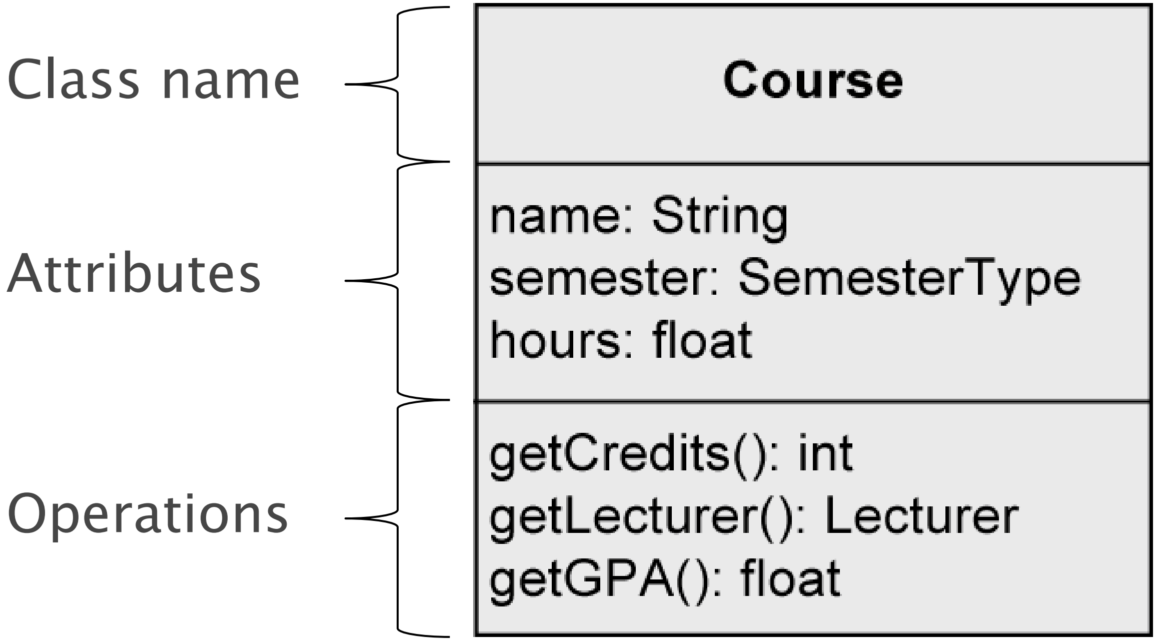
Divide and conquer

- 한 모듈의 세부 설계나 알고리즘 개발 시 적용되는 개념
- 하나의 문제를 점점 더 작은 문제들로 반복해서 쪼갬
- 각각의 작은 문제들을 해결함으로써 최종 문제를 해결함

0.4 UML Class Diagram Short Review for Design Patterns

Class Diagram

- Shows a set of classes, interfaces, and collaborations and their relationships (dependency, generalization, association and realization)
- Represents the <u>static view</u> of a system



Class name

Attributes

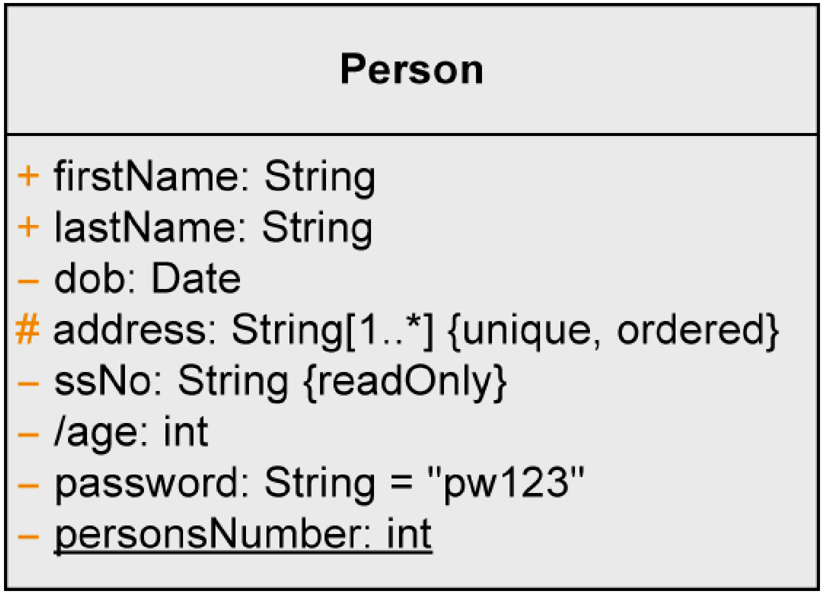
Operations

Course

name: String
semester: SemesterType
hours: float

getCredits(): int
getLecturer(): Lecturer
getGPA(): float

Attribute Syntax - Visibility

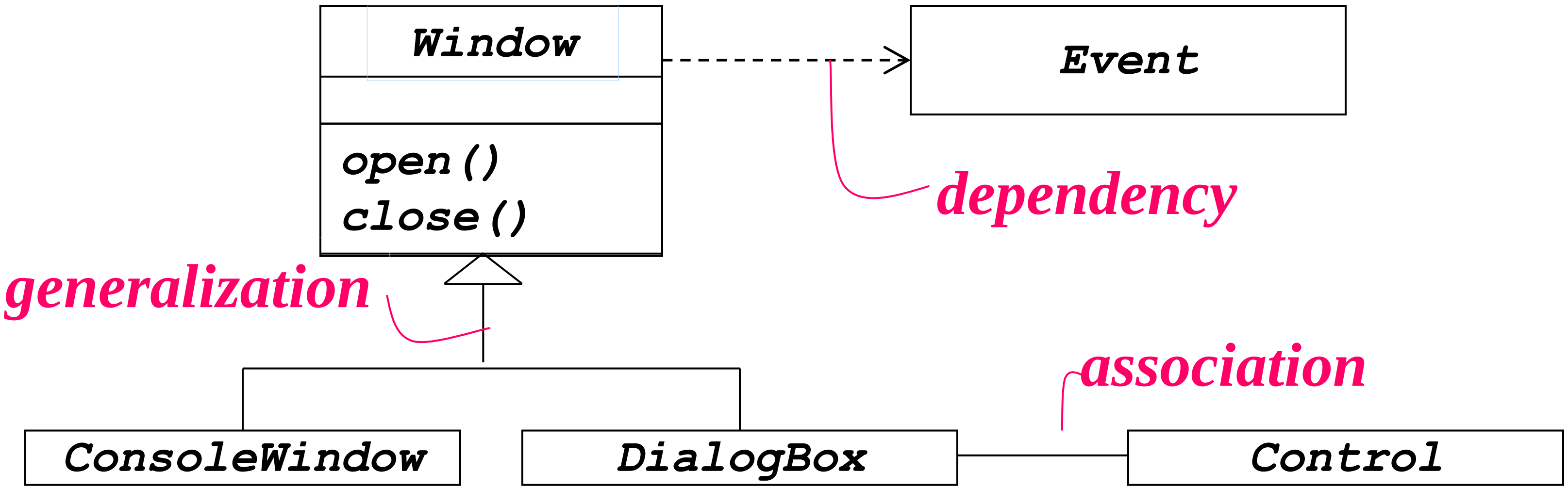


Person

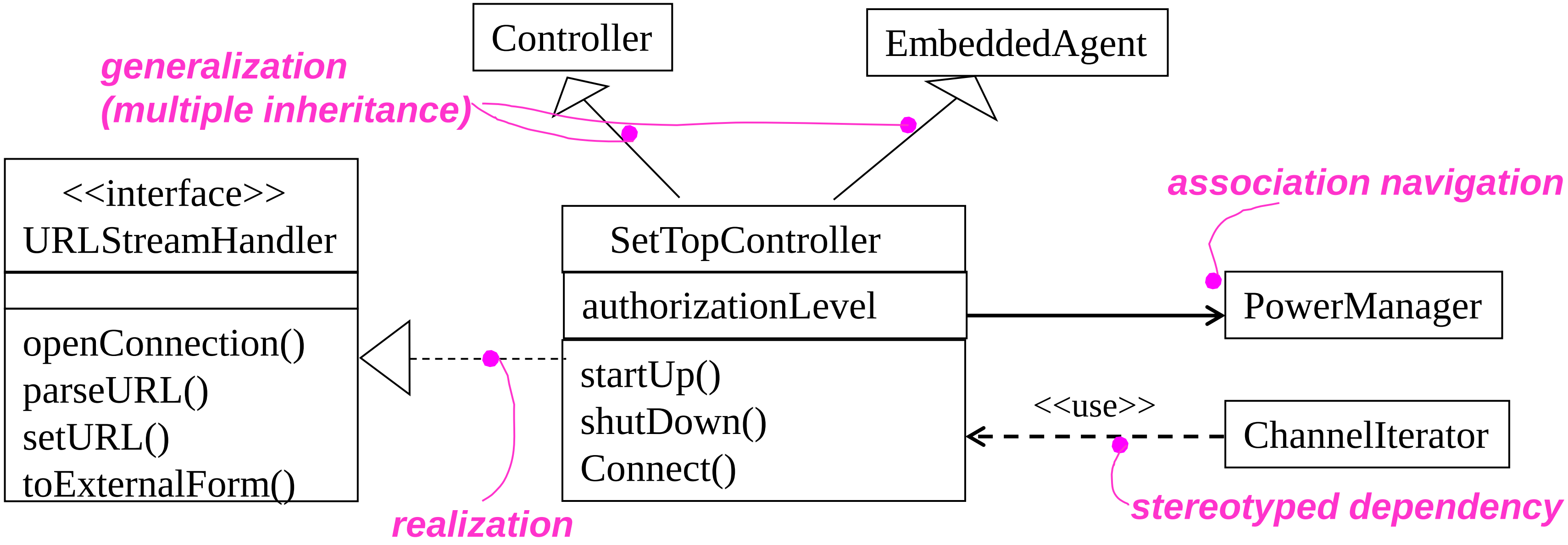
+ firstName: String
+ lastName: String
– dob: Date
# address: String[1..*] {unique, ordered}
– ssNo: String {readOnly}
– /age: int
– password: String = "pw123"
– personsNumber: int

- Who is permitted to access the attribute
  - + ... public: everybody
  - - ... private: only the object itself
  - # ... protected: class itself and subclasses
  - ~ ... package: classes that are in the same package

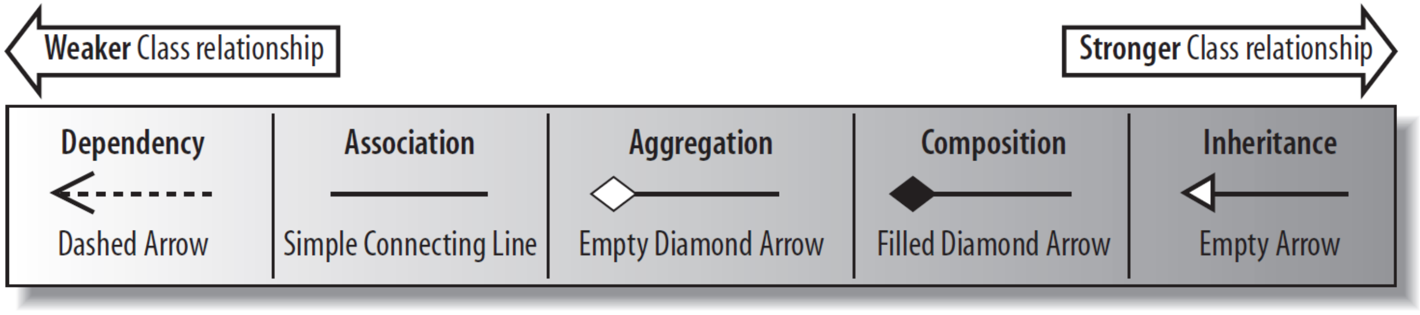
Relationships



Relationships



Types of class relationship



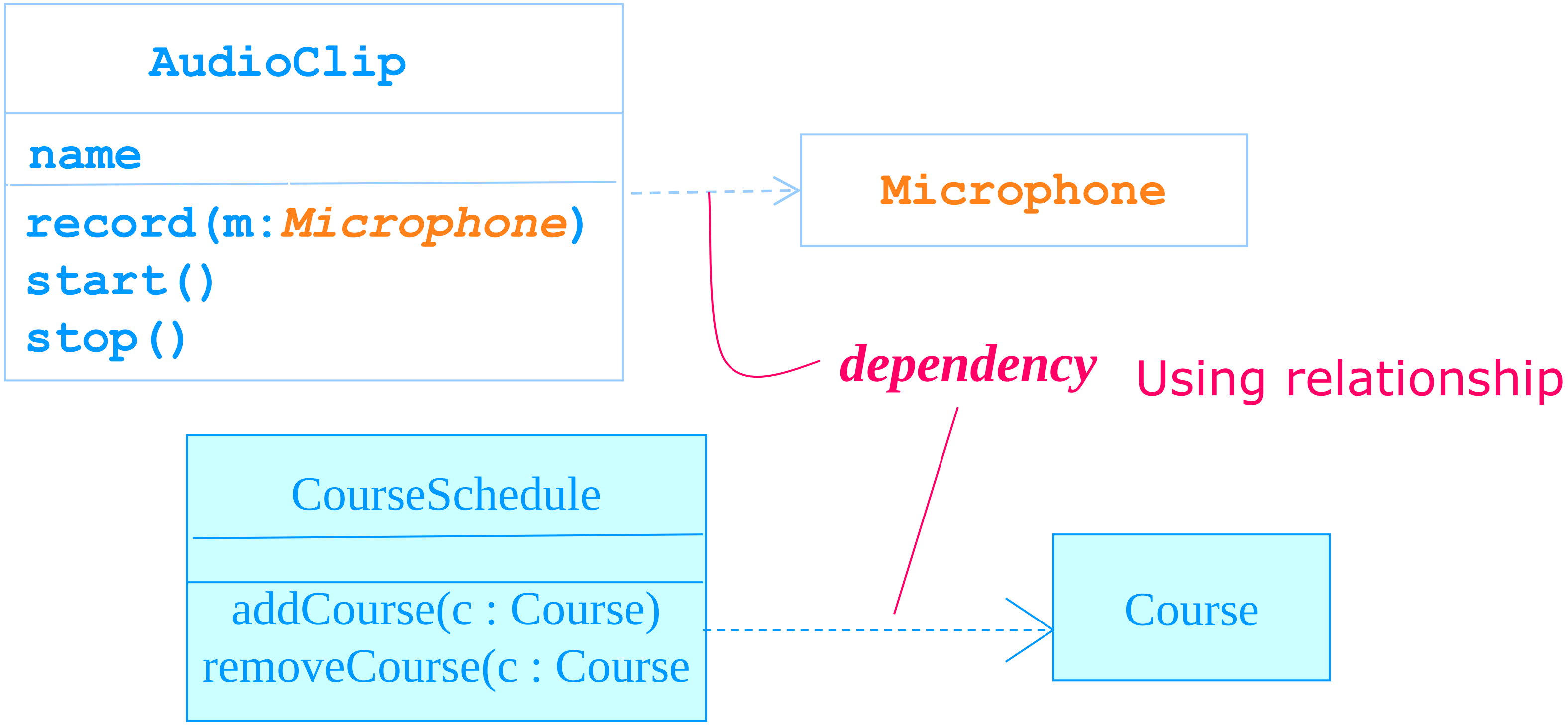
Weaker Class relationship

Stronger Class relationship

Dependency	Association	Aggregation	Composition	Inheritance
Dashed Arrow	Simple Connecting Line	Empty Diamond Arrow	Filled Diamond Arrow	Empty Arrow
When objects of one class **work briefly with** objects of another class	When objects of one class **work with** objects of another class for some prolonged amount of time	When one class **owns but shares a reference** to objects of another class	When one class **contains** objects of another class	When one class **is a** type of another class

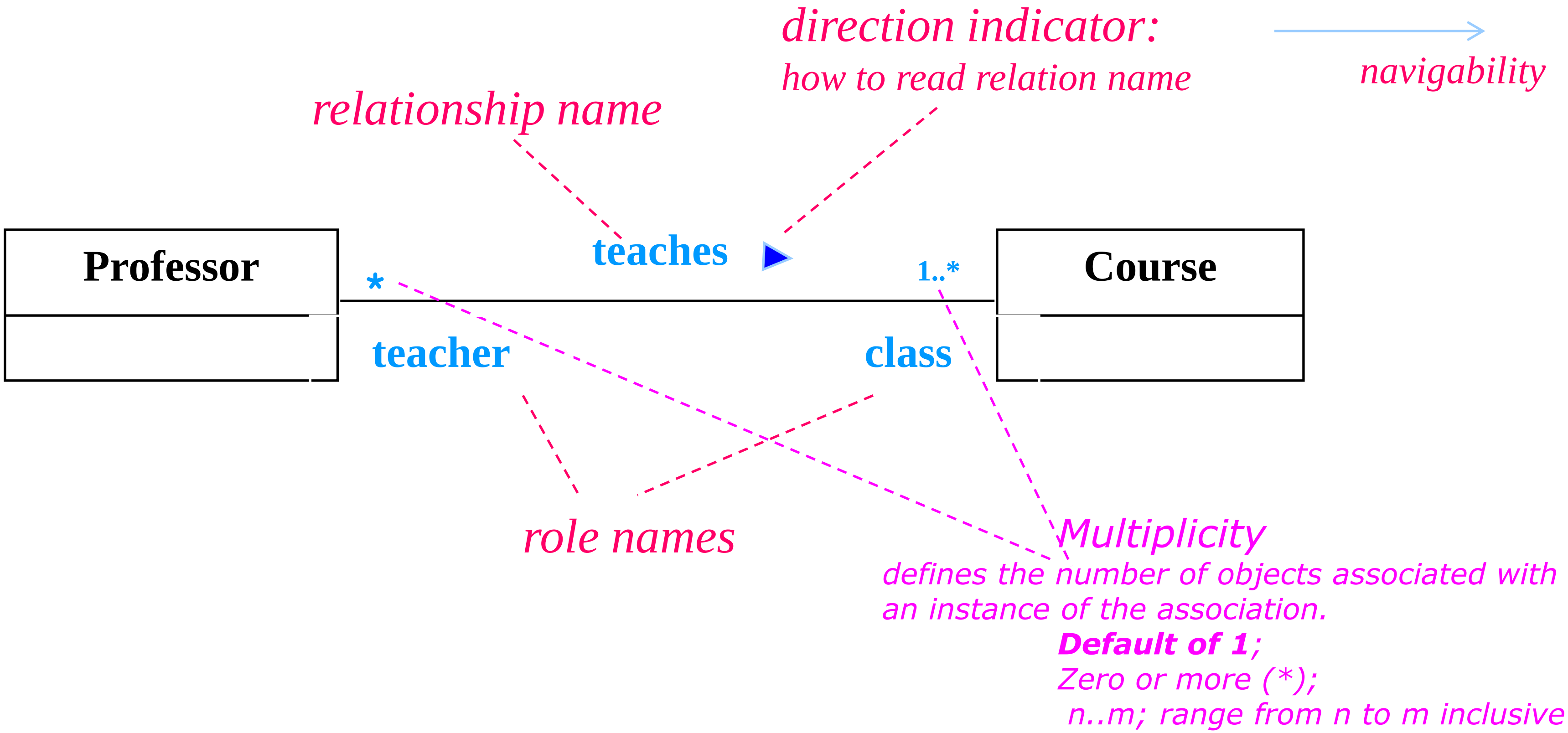
Dependency

- **A change in one thing may affect another.**
  - The most common dependency between two classes is one where one class <<use>>s another as a *parameter to an operation*.



Associations

- Represent conceptual relationships between classes



relationship name

direction indicator:
how to read relation name

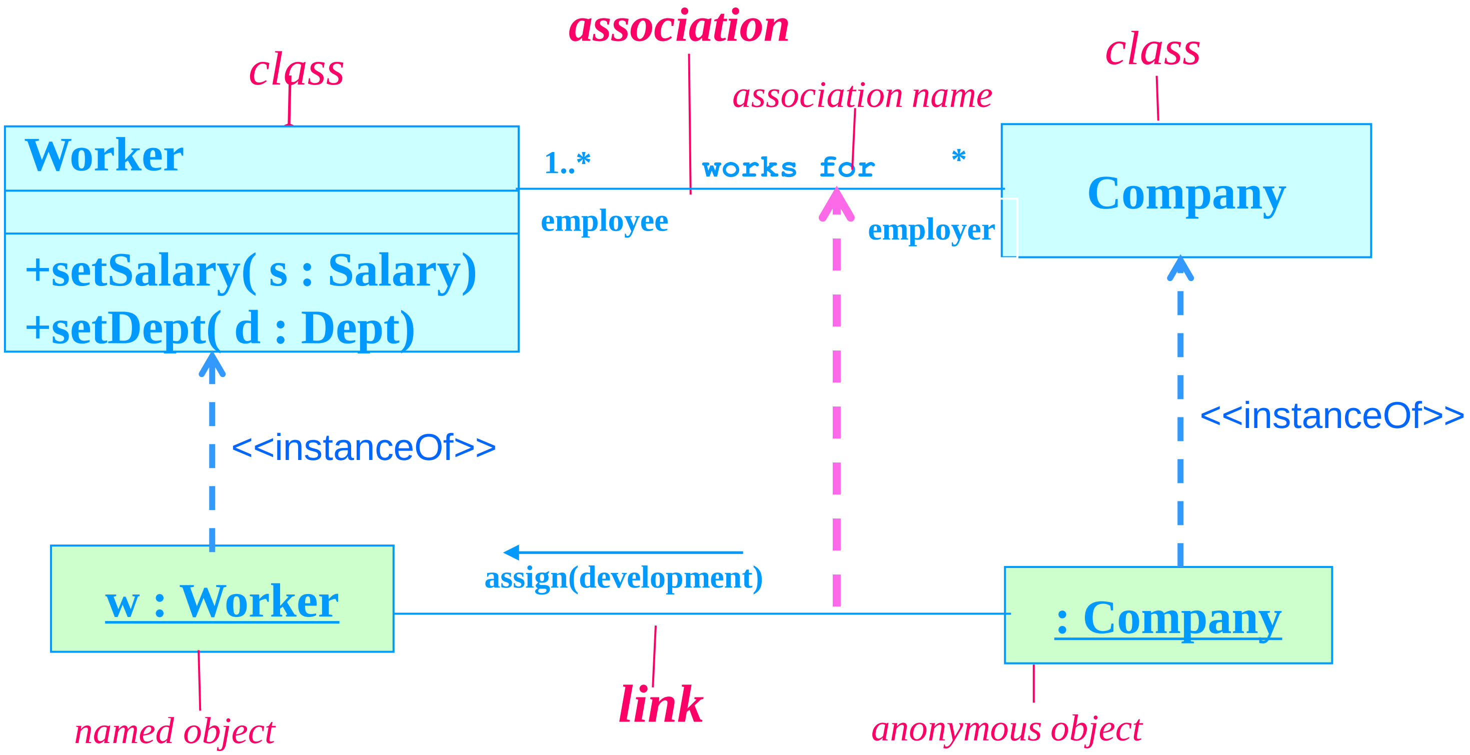
navigability

role names

Multiplicity
defines the number of objects associated with an instance of the association.
***Default of 1**;*
Zero or more (*);
n..m; range from n to m inclusive

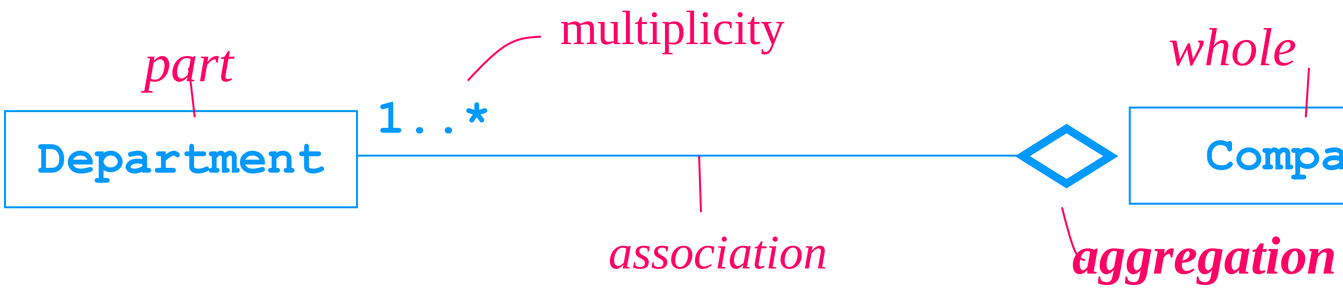
Associations – Links

- link is a semantic connection among objects.
- A link is an instance of an association.



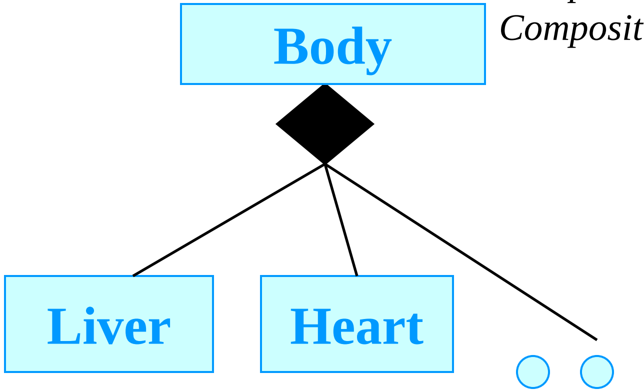
Aggregation & Composition

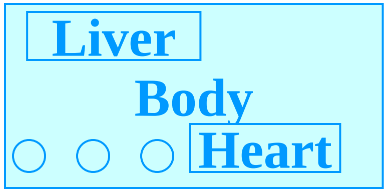
Aggregation - structural association representing <u>“whole/part” relationship</u>.
- “has-a” relationship.

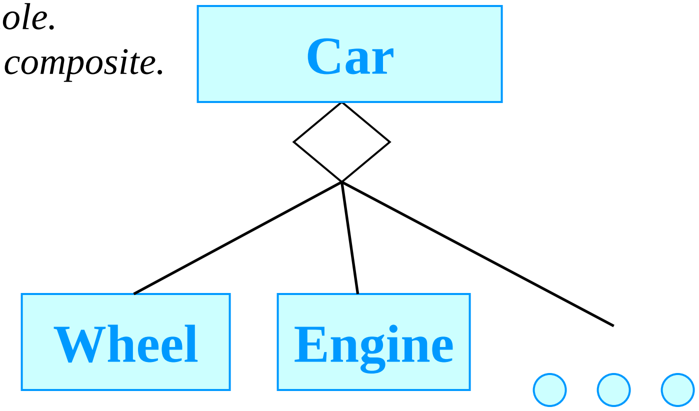


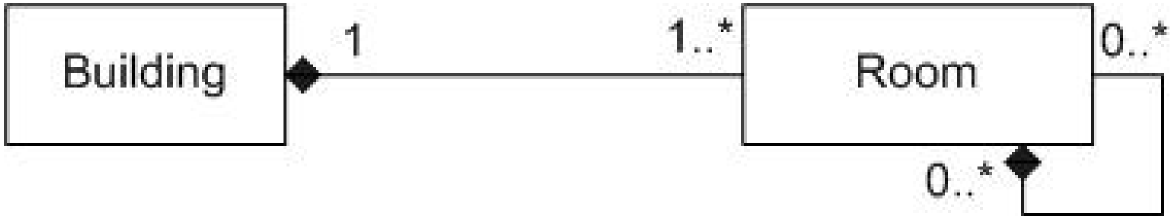
Composition

Composition is a stronger form of aggregation.
Composite parts live and die with the whole.
Composite parts may belong to only one composite.



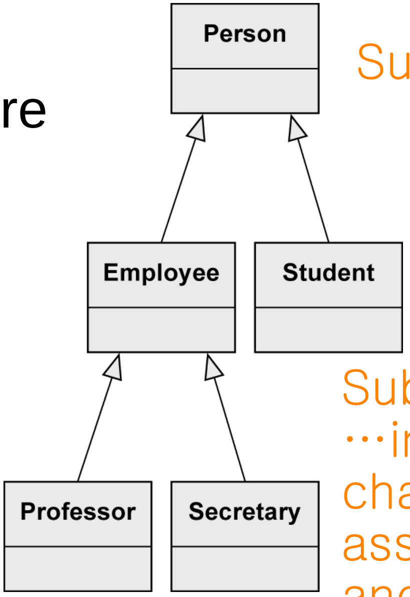






Inheritance (Generalization)

- Characteristics (attributes and operations), associations, and aggregations that are specified for a general class (superclass) are passed on to its subclasses.
- Every instance of a subclass is simultaneously an indirect instance of the superclass.
- Subclass inherits all characteristics, associations, and aggregations of the superclass except private ones.
- Subclass may have further characteristics, associations, and aggregations.
- Generalizations are transitive.



Superclass

Subclasses …inherit characteristics, associations, and aggregations

A Secretary is an Employee and a Person