Design Patterns

ebru@hacettepe.edu.tr
ebruakcapinarsezer@gmail.com
http://yunus.hacettepe.edu.tr/~ebru/
@ebru176
Şubat 2017

Policy

- One midterm (24 points), total 3 uninformed quiz (12 points for each) and final exam (50 points)

-No homework

-No memorization

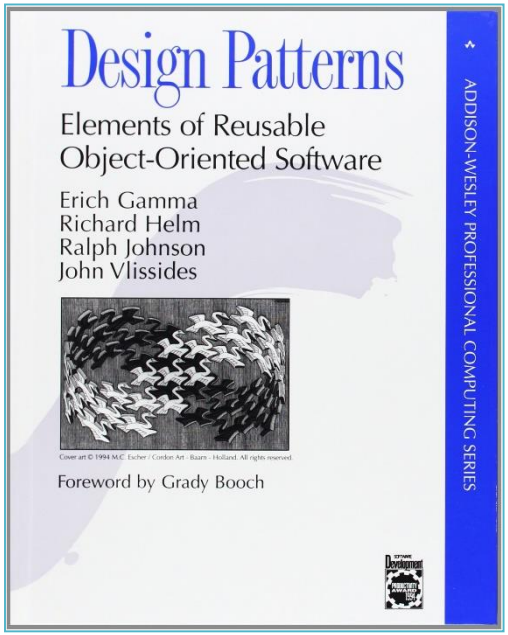
-Explanation, discussion, question-answer

-Slides will be shared on personel web site

-Also answer key of exams

Write «design patterns» to Google, get more than you need

Textbook



Why or Why not

Because of

- me
- hesitate from design
- having more funny options
- bilingual content
- being shy to answer me
- dislike pen and paper

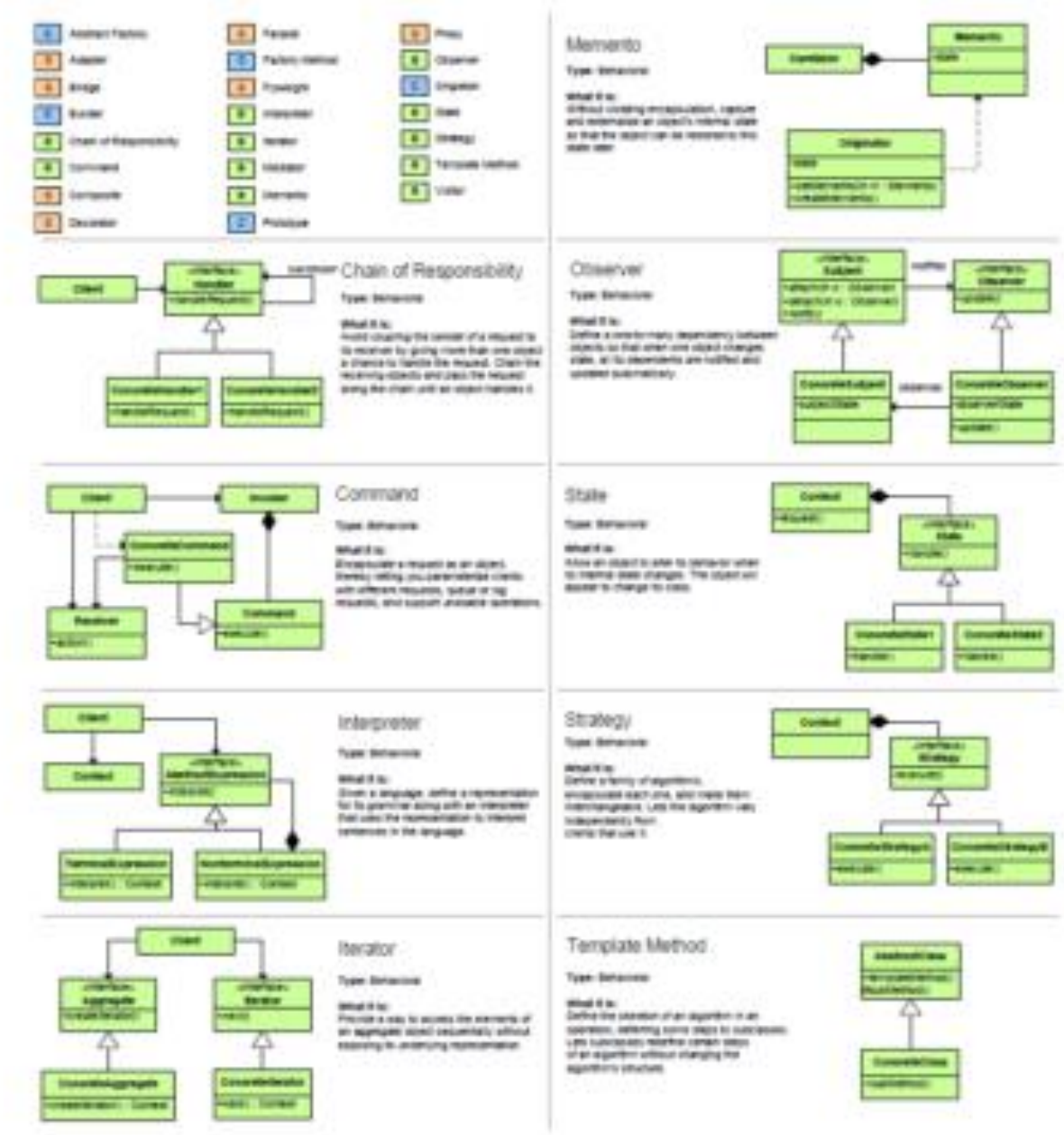
you may withdraw

Because of

- me
- direct focus on design

you may be willing to go on

Content, as expected…



Lets start: what is the definition of…

-Class

-Association

-Inheritance

-Polymorphism

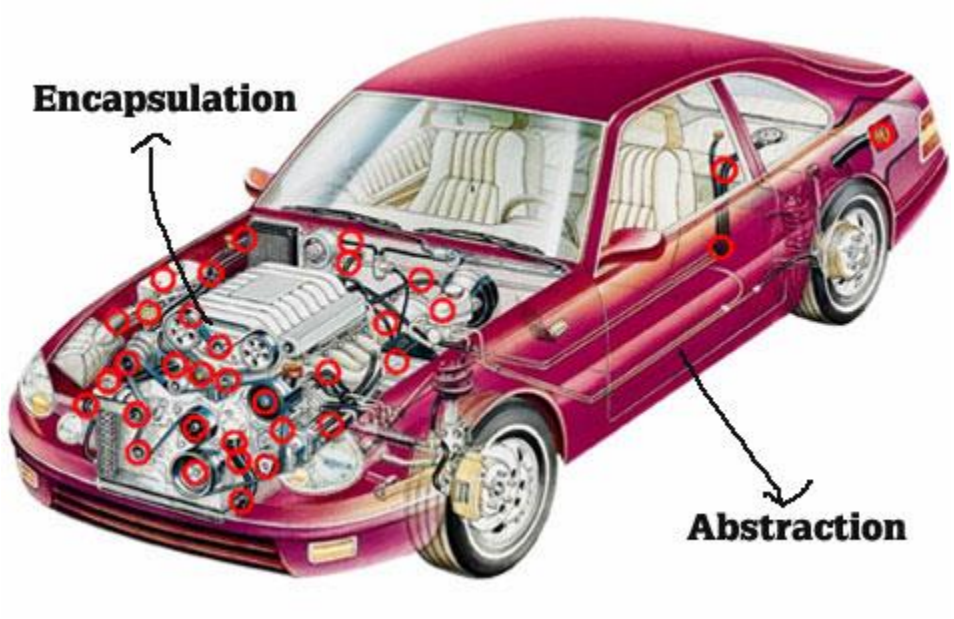
-Encapsulation

-Abstraction

-UML & its diagrams

Fast Review: Confusing Start

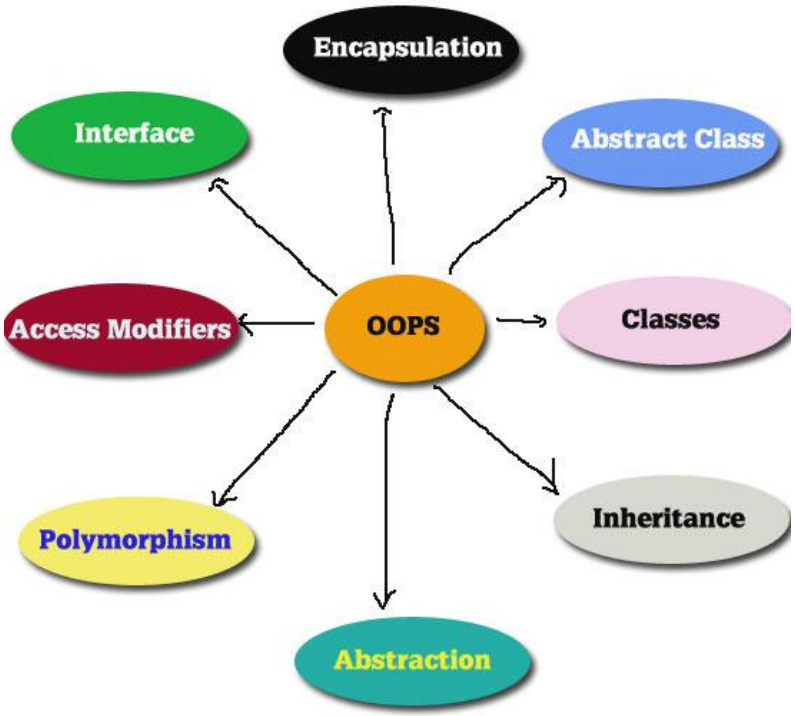
Encapsulation: Hiding implementation details, define expected I/Os

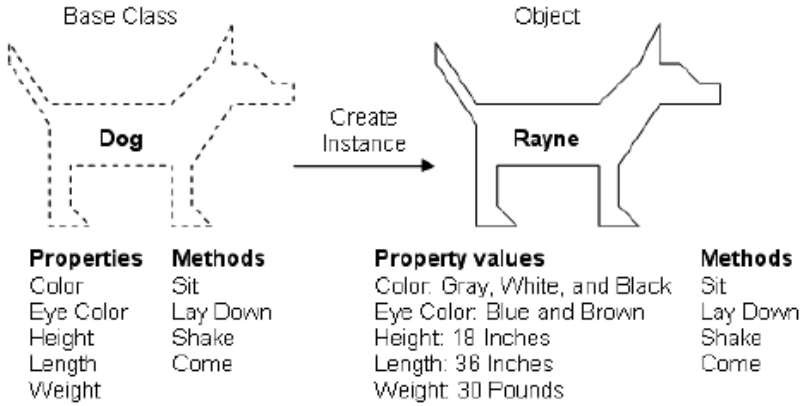


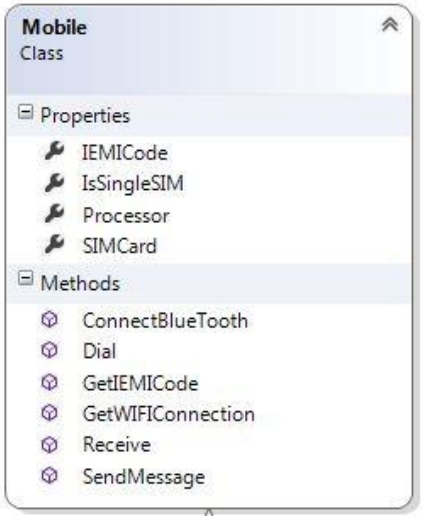
Cause and effect

Abstraction: Hiding whole system, define a way to access the system

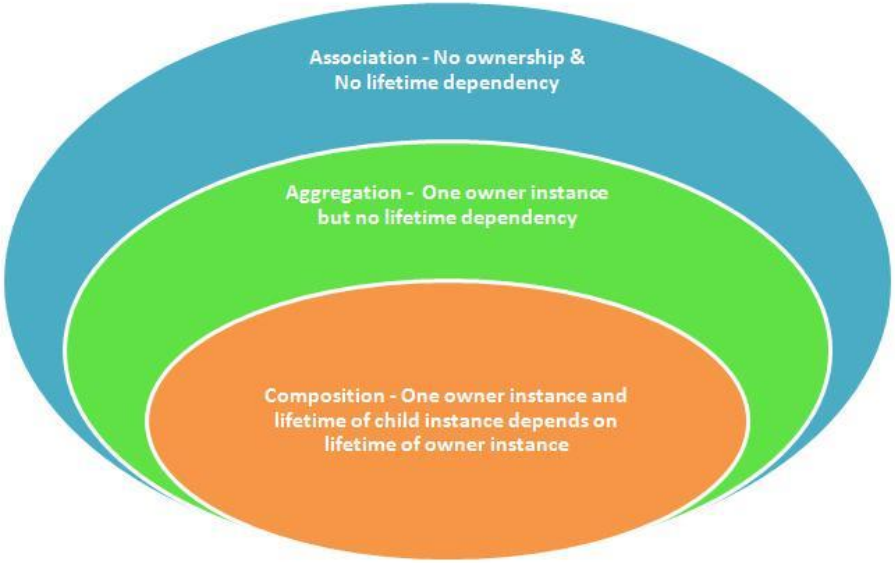
Fast Review: OOP Concepts

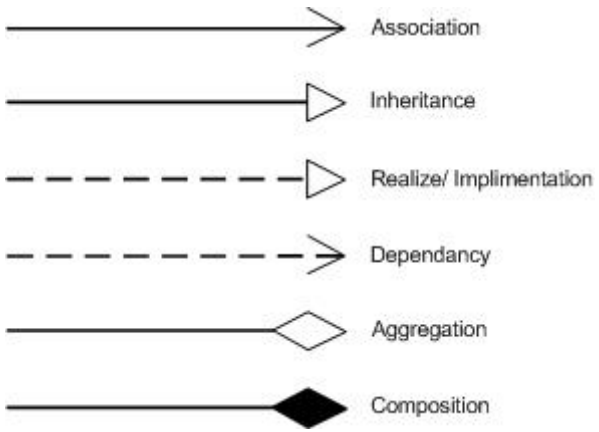






Fast Review: Object Relations





Object Relations and Multiplicities

Multiplicity means to define that how many objects can be related with the objects of correponding class.

Relation means to there are some structural reasons to occur together or unstructural reasons for communication between objects to achieve a specific purpose

Multiplicity:

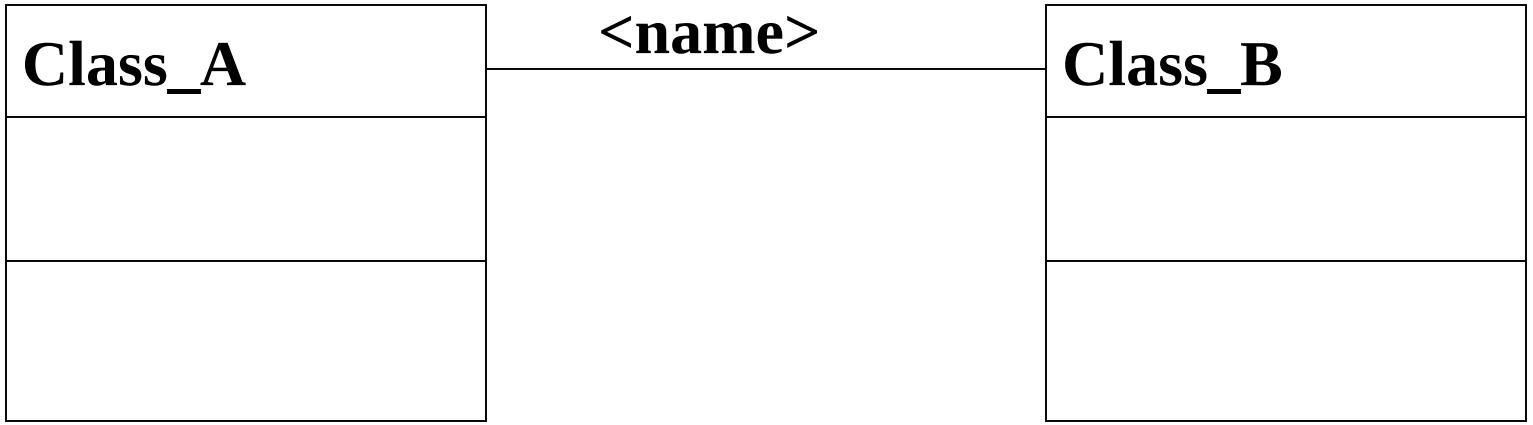
In OO analysis and design this can help to implement, test and debug the code.

Relations:

The relations of ***is_a*** and ***has_a*** are fundamental ways to understand collections of classes.

In an OO implementation these relations will usually be visible in the code. But they are not the **only** interesting relations!

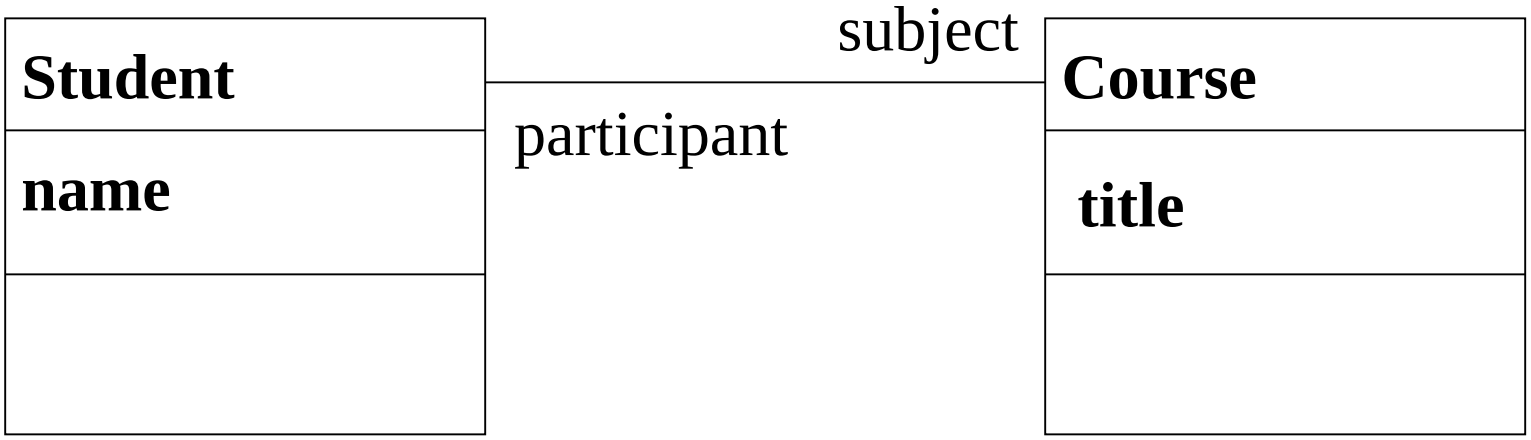
The simplest association is binary and represented by a line e.g.



Normally, we at least annotate the association with a <name>, e.g.

An arrow can be added to show the orientation or asymmetry of the relation.

In this case **studies** is not symmetric.

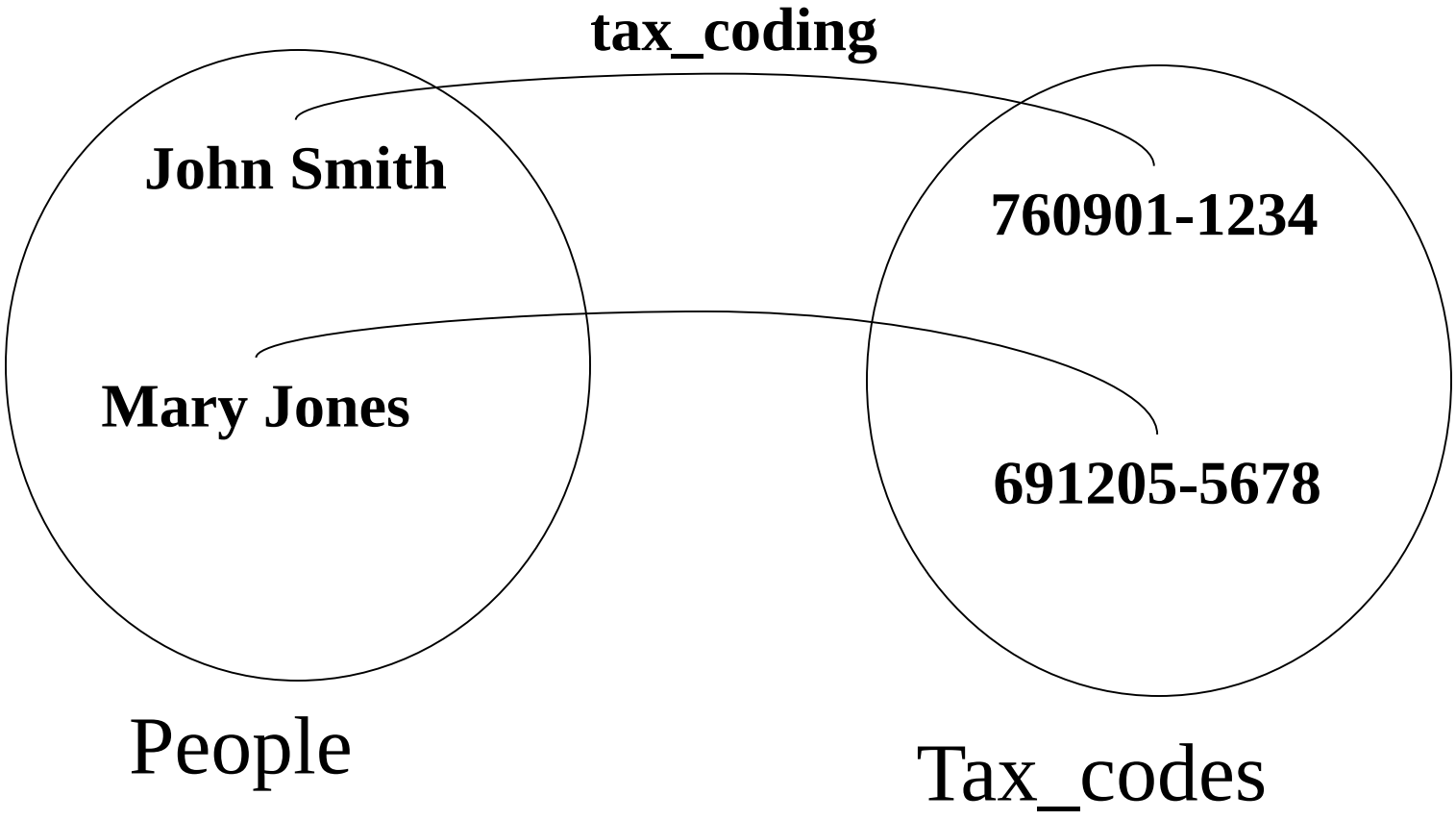


The existence of an association between two classes often indicates some level of **coupling**, in the sense of *related concepts*.

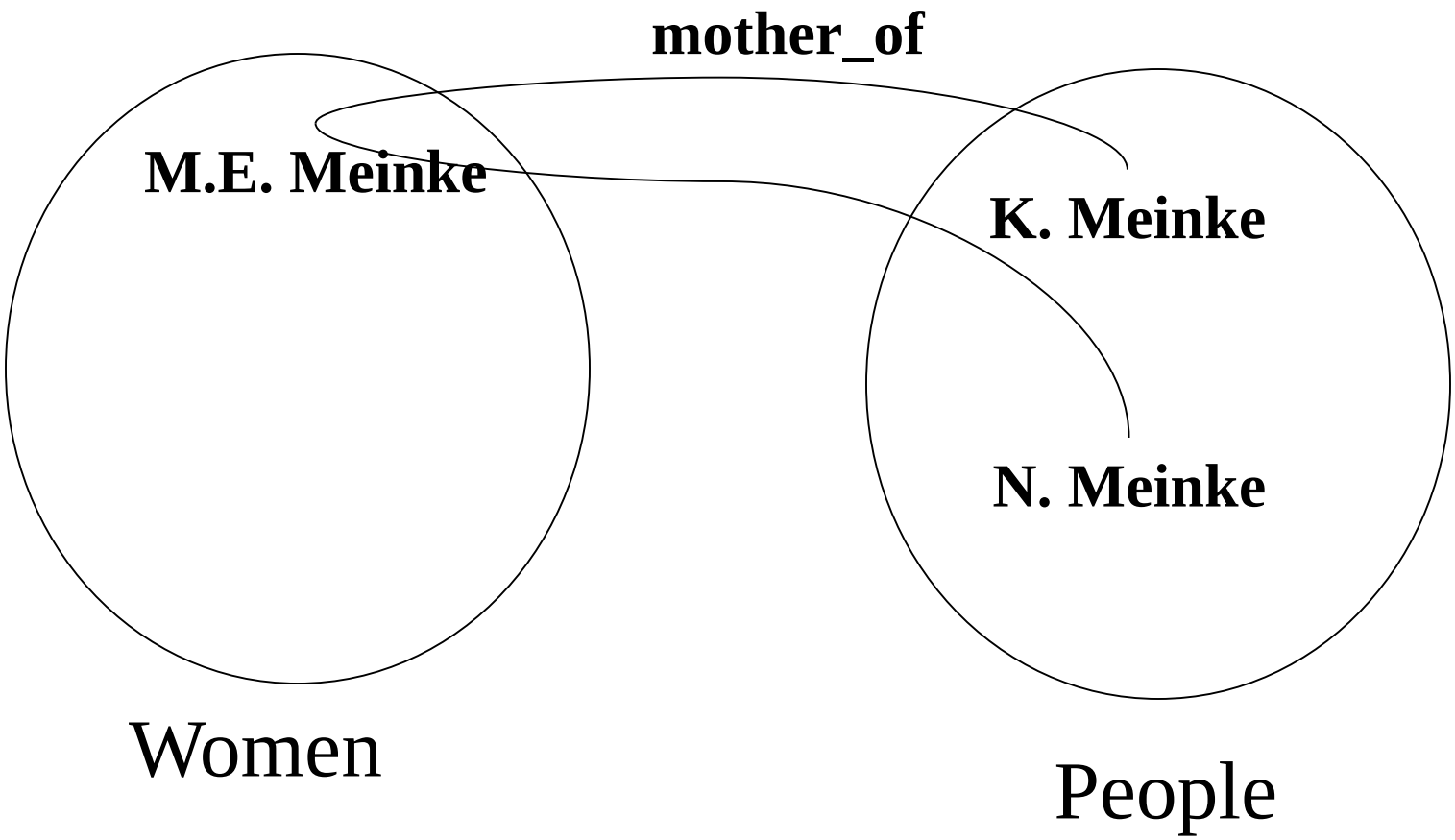
According to standard practice, coupling should be minimized and cohesion maximized.

The commonest multiplicities are:

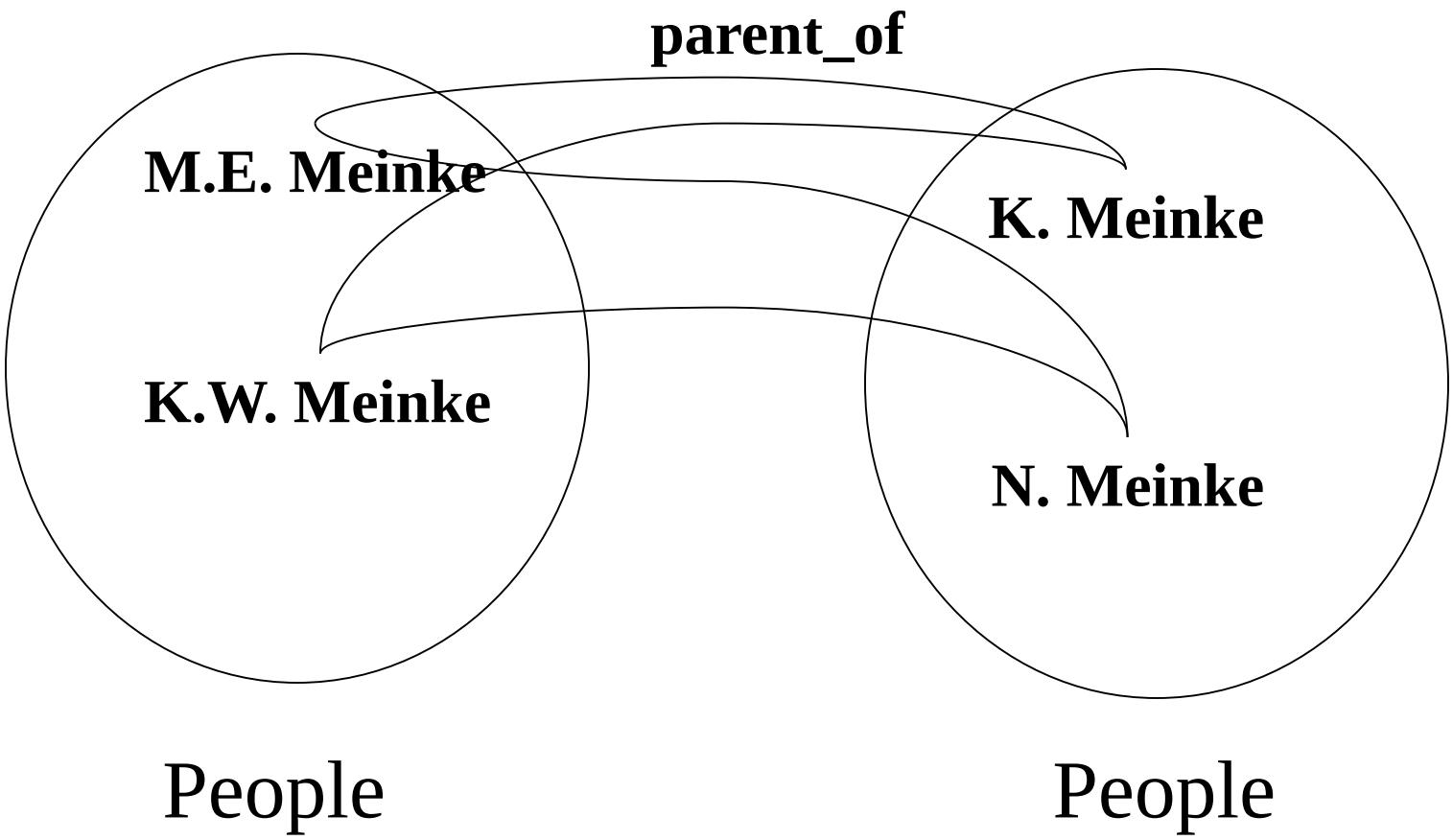
One-to-one



One-to-many



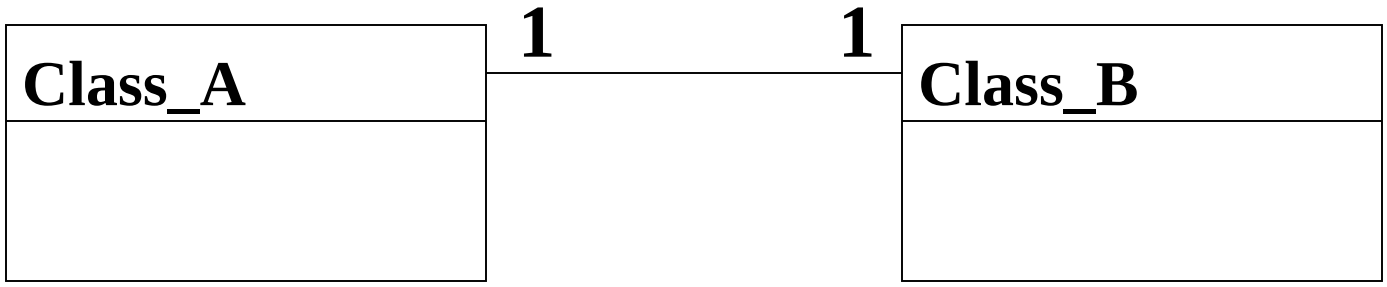
Many-to-many


parent_of
M.E. Meinke
K.W. Meinke
K. Meinke
N. Meinke
People
People

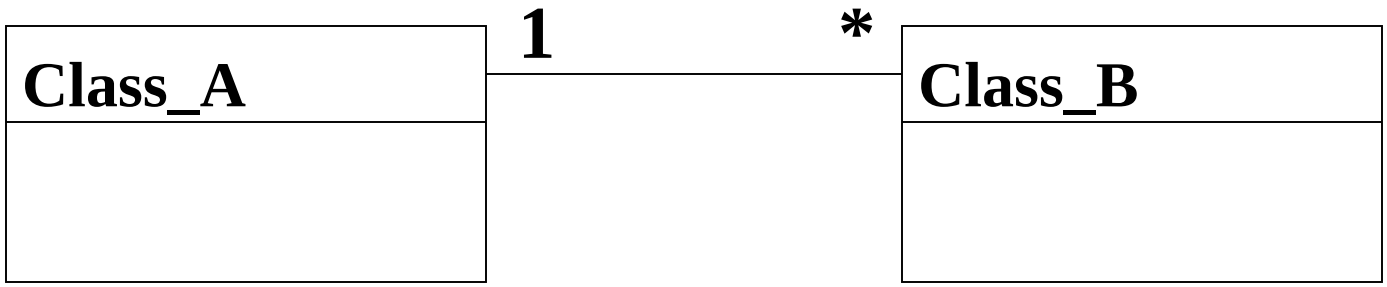
UML has a notation for multiplicity:

1	one and only one
0 .. 1	zero or one
M .. N	from M to N
*	greater than or equal to zero
0 .. *	…same …
M .. *	greater than or equal to M

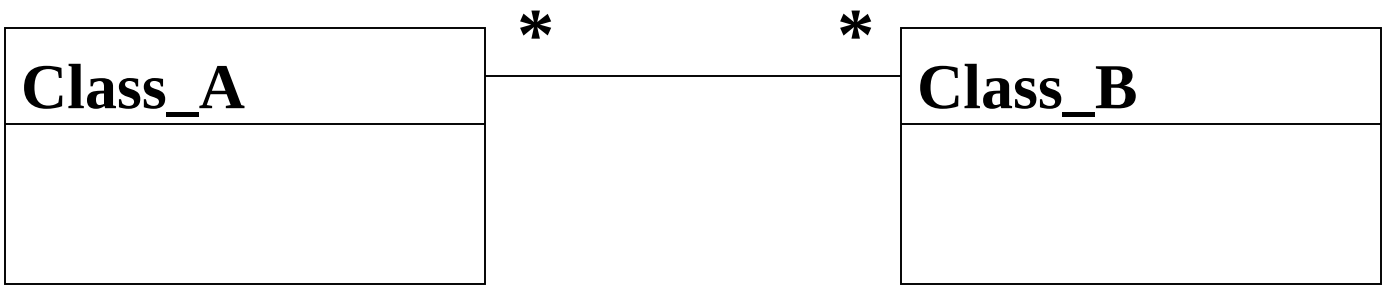
Examples: one-to-one



one-to-many

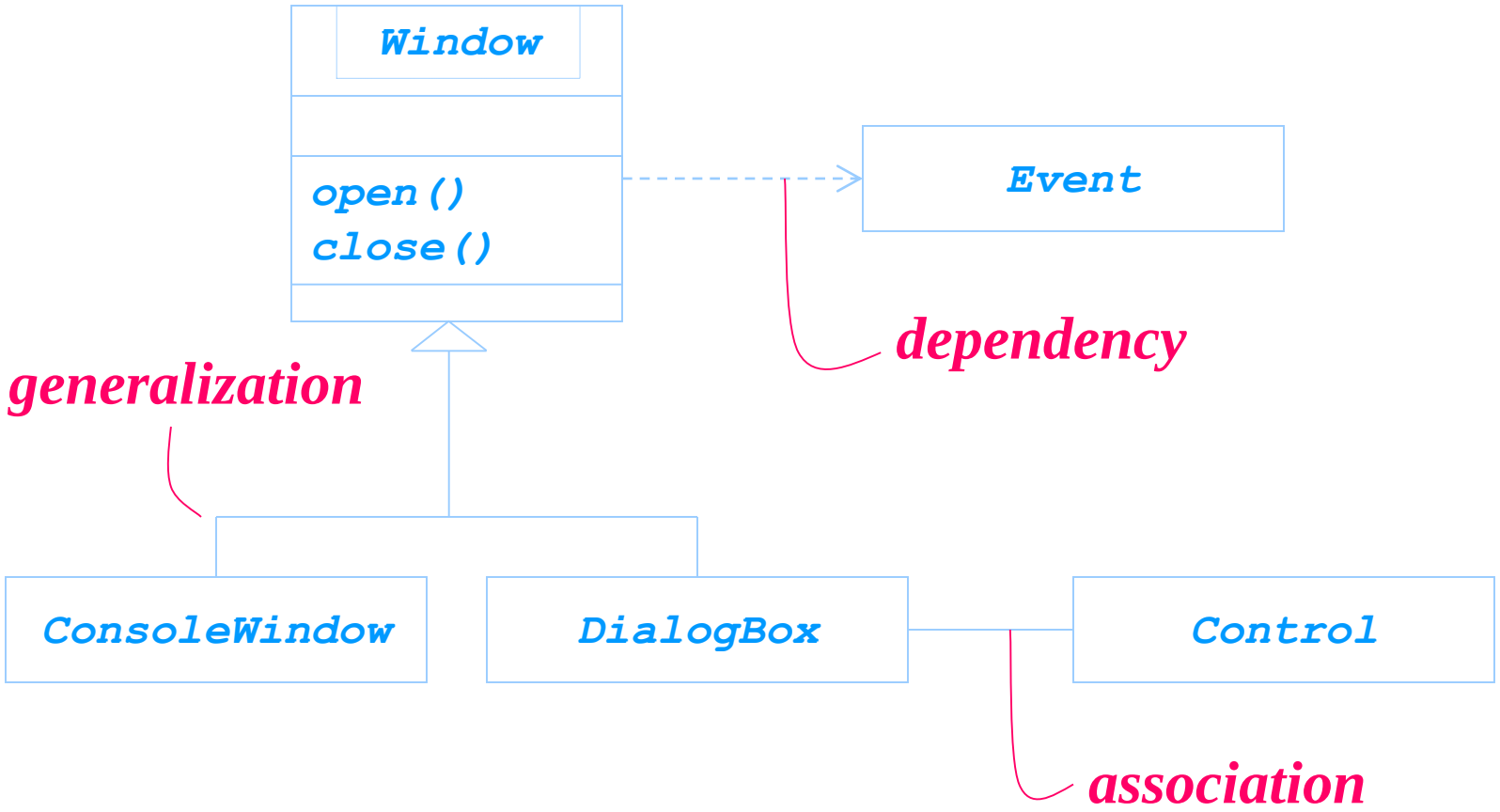


many-to-many



We can also add multiplicity constraints to aggregation and composition relations, e.g. …

Relationships: 3 Kinds

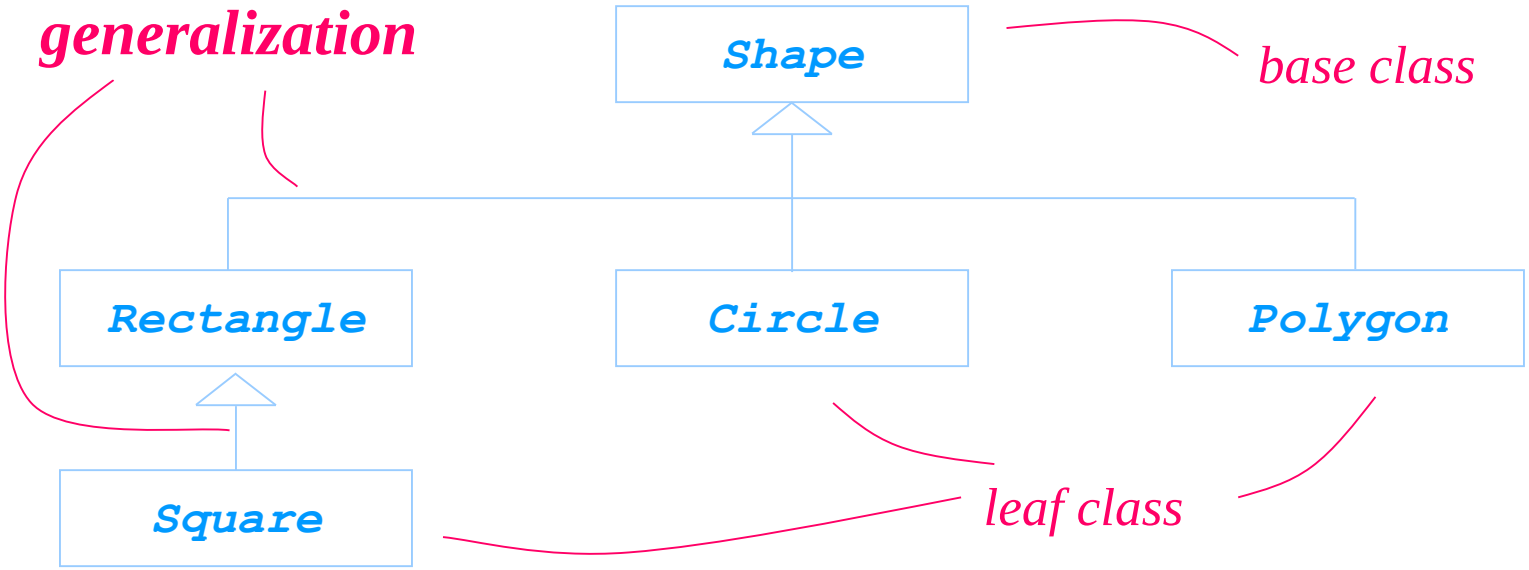


Generalization

Relationship between general thing (parent) and more specific thing (child)

Child “is-a-kind-of” parent.

Child inherits attributes and operations of parent.

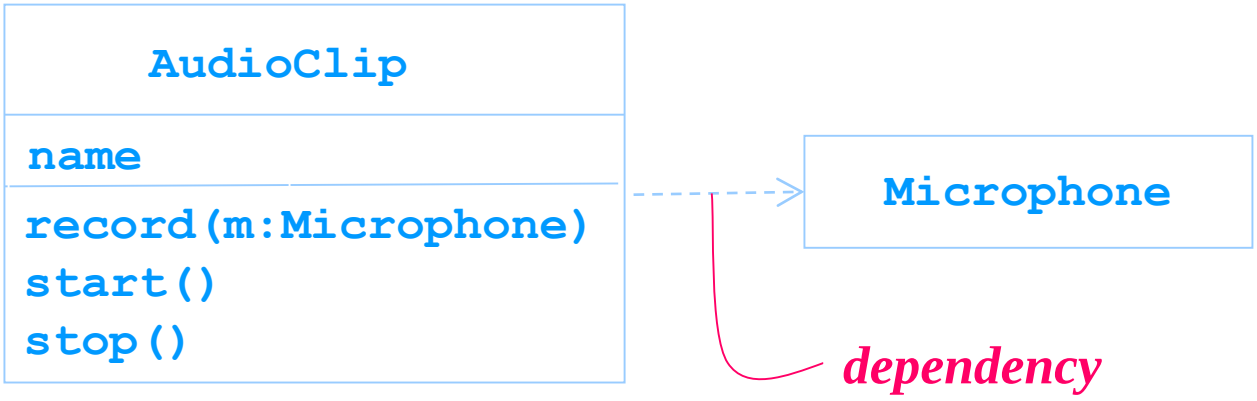


Dependency

A change in one thing may affect another.

“Uses” relationship.

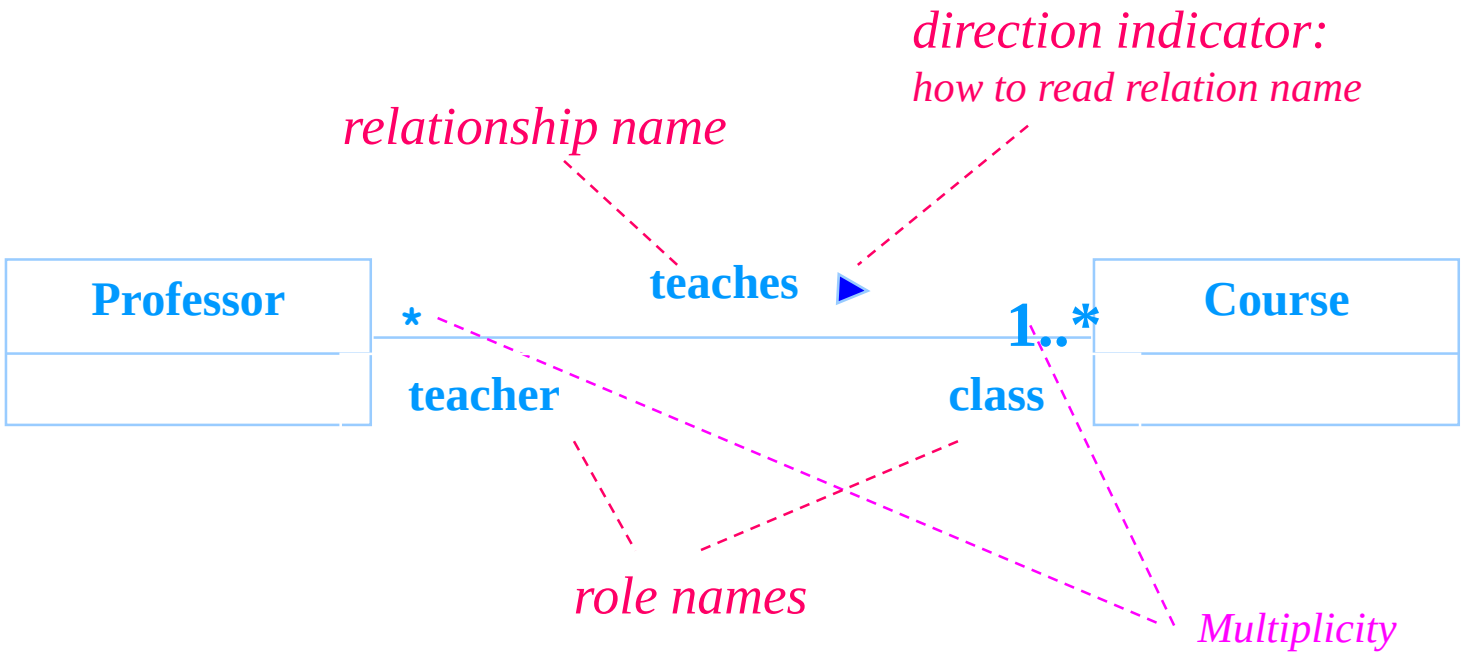
May have a name, but not common.



One important use of dependency

Associations (UML)

- Represent conceptual relationships between classes



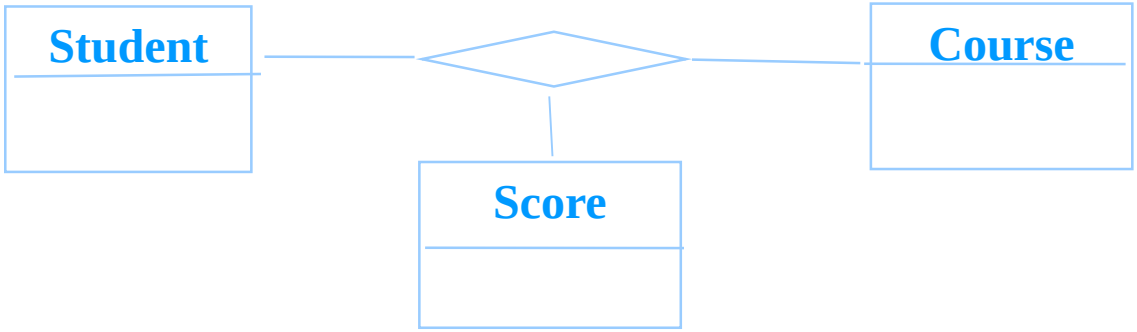
Associations - In Other OOAD

Associations may be binary, ternary, or higher order

binary association

Fusion Style

Ternary association



Associations – A Question

How would you model the following situation?

"You have two files, say Homework1 and MyPet, where Homework1 is accessible only by you, but MyPet is accessible by anybody."

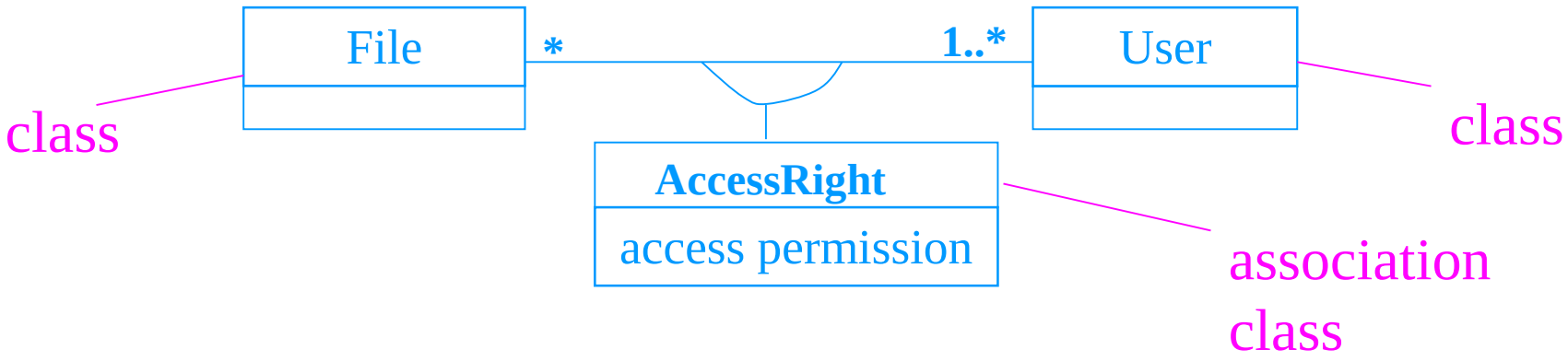
You could create two classes, File and User. Homework1 and MyPet are files, and you are a user.

Approach 1: Now, would you associate the file access right with File?

Approach 2: Or, would you associate the file access right with User?

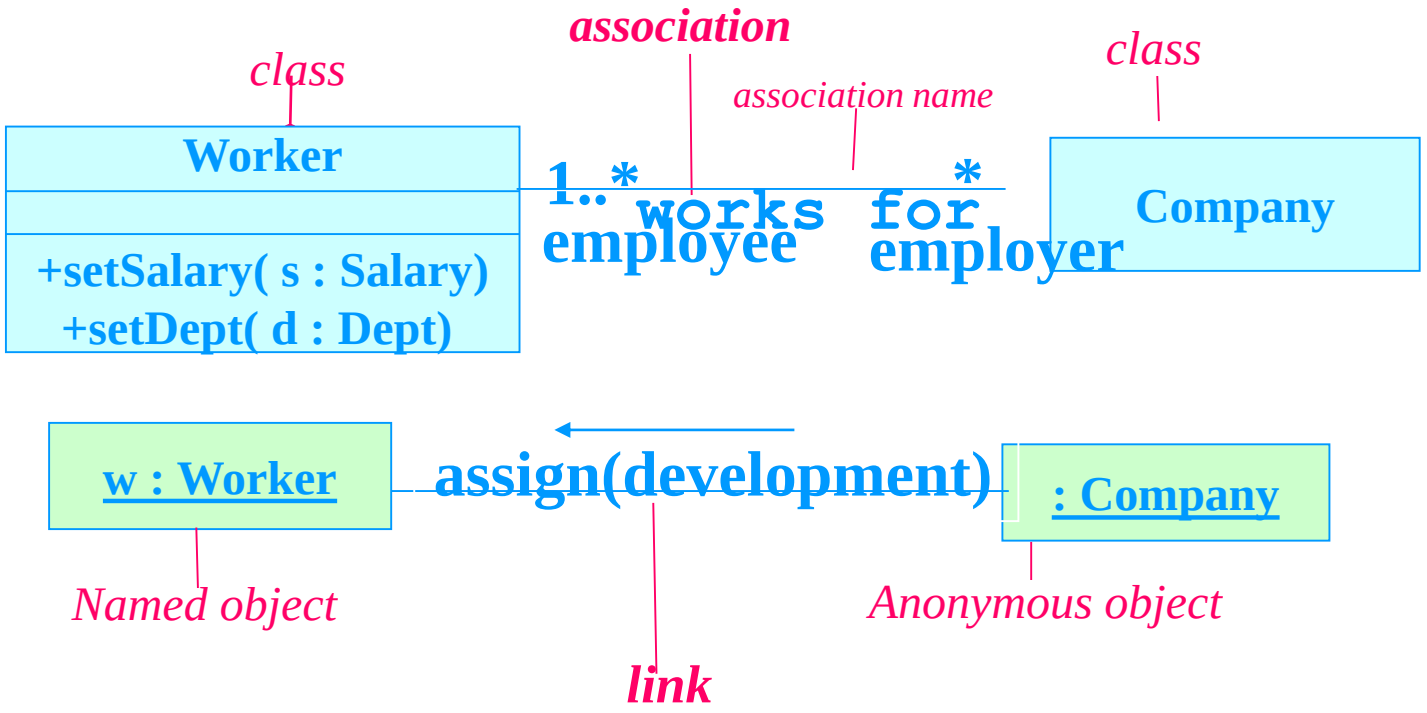
Associations

- UML Association Class



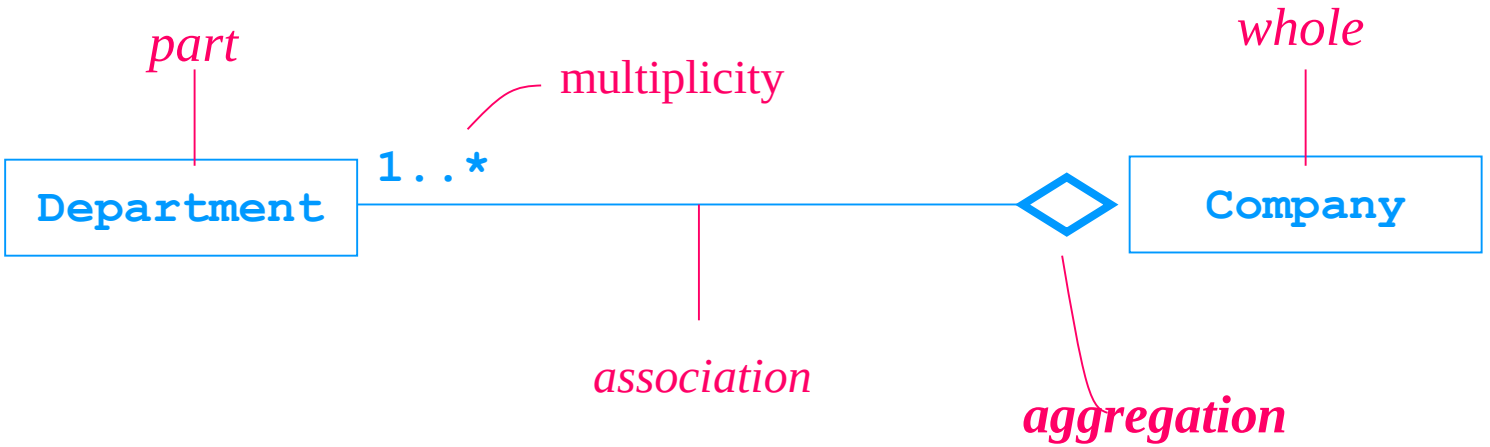
Associations – UML Links

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



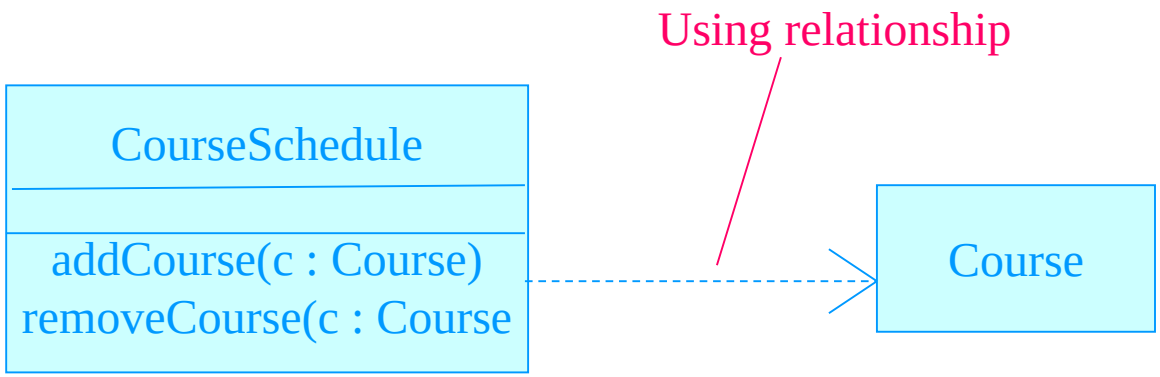
Associations - Aggregation

- structural association representing “whole/part” relationship.
- “has-a” relationship.



Modeling Simple Dependencies

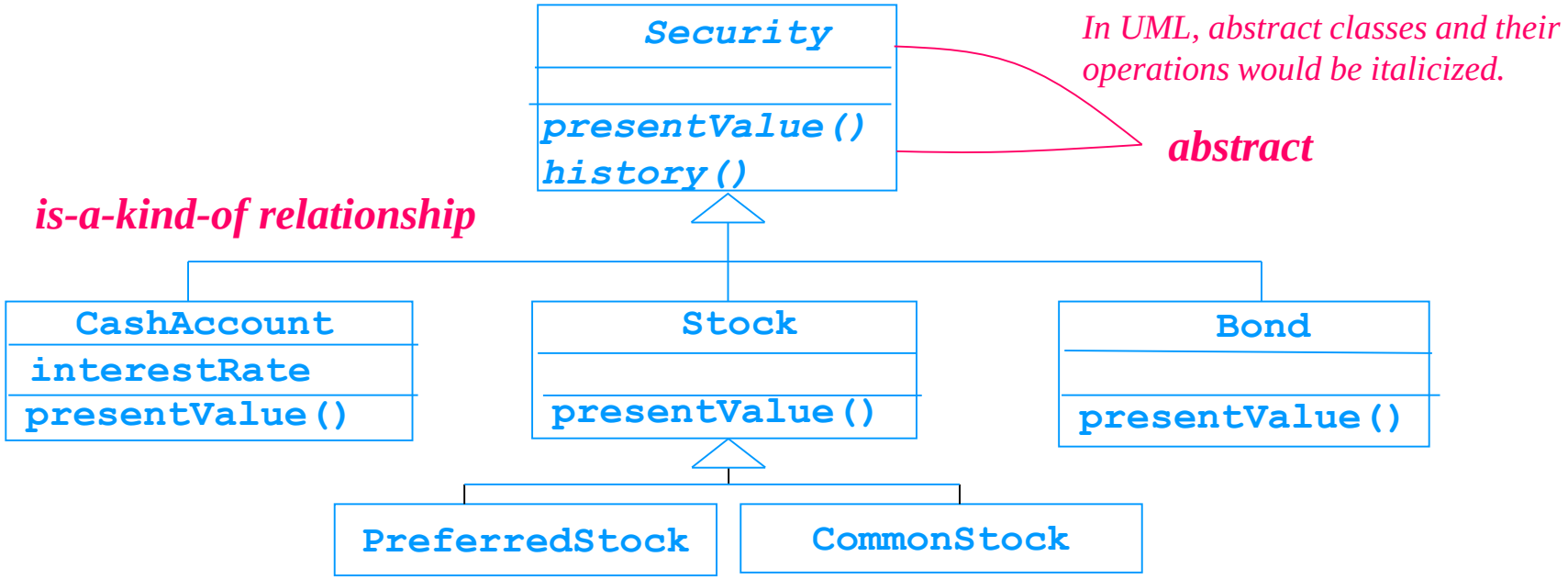
- The most common dependency between two classes is one where one class uses another as a parameter to an operation.
  Create dependency pointing from class with operation to parameter.



Usually initial class diagrams will not have any significant number of dependencies in the beginning of analysis but will as more details are identified.

Modeling Single Inheritance

- Look for common responsibilities, attributes, and operations that are common to two (2) or more classes.
- If necessary, create a new class to assign commonalities.
- Specify that the more-specific classes inherit from the more-general.

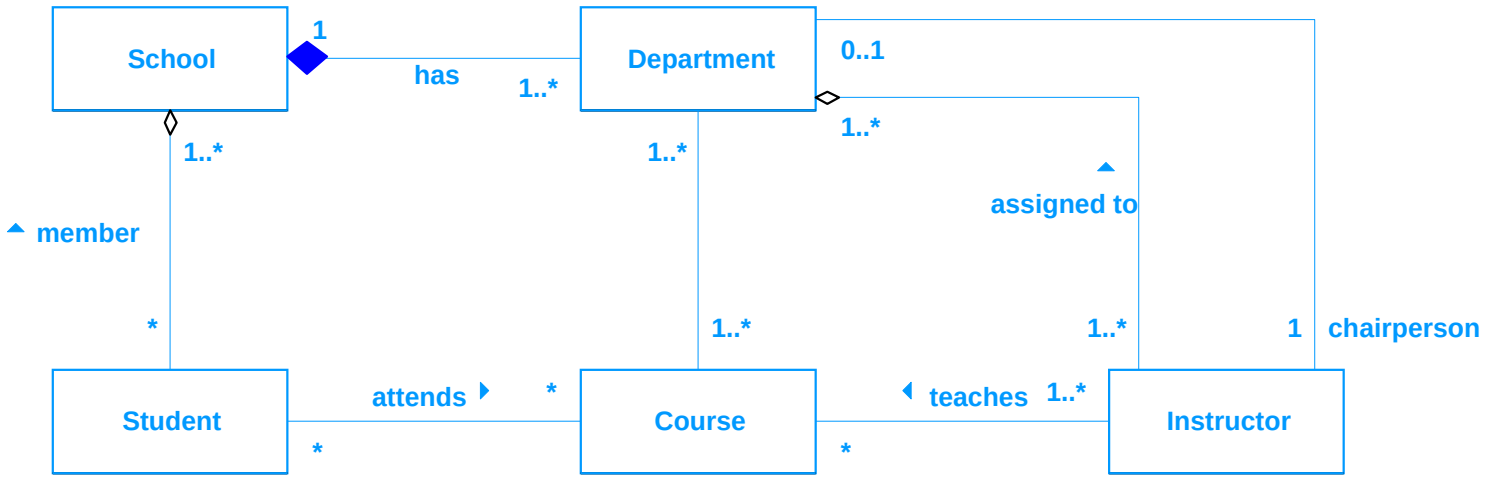


Modeling Single Inheritance (cont'd)

- Abstract
  - Abstraction—the essential characteristics of a thing.
  - Abstract class—cannot be instantiated.
  - Abstract method—has no implementation defined (i.e., no method body).
  - Depicted in italics or with stereotypes.
- Concrete
  - Not abstract.
  - Can have instances.

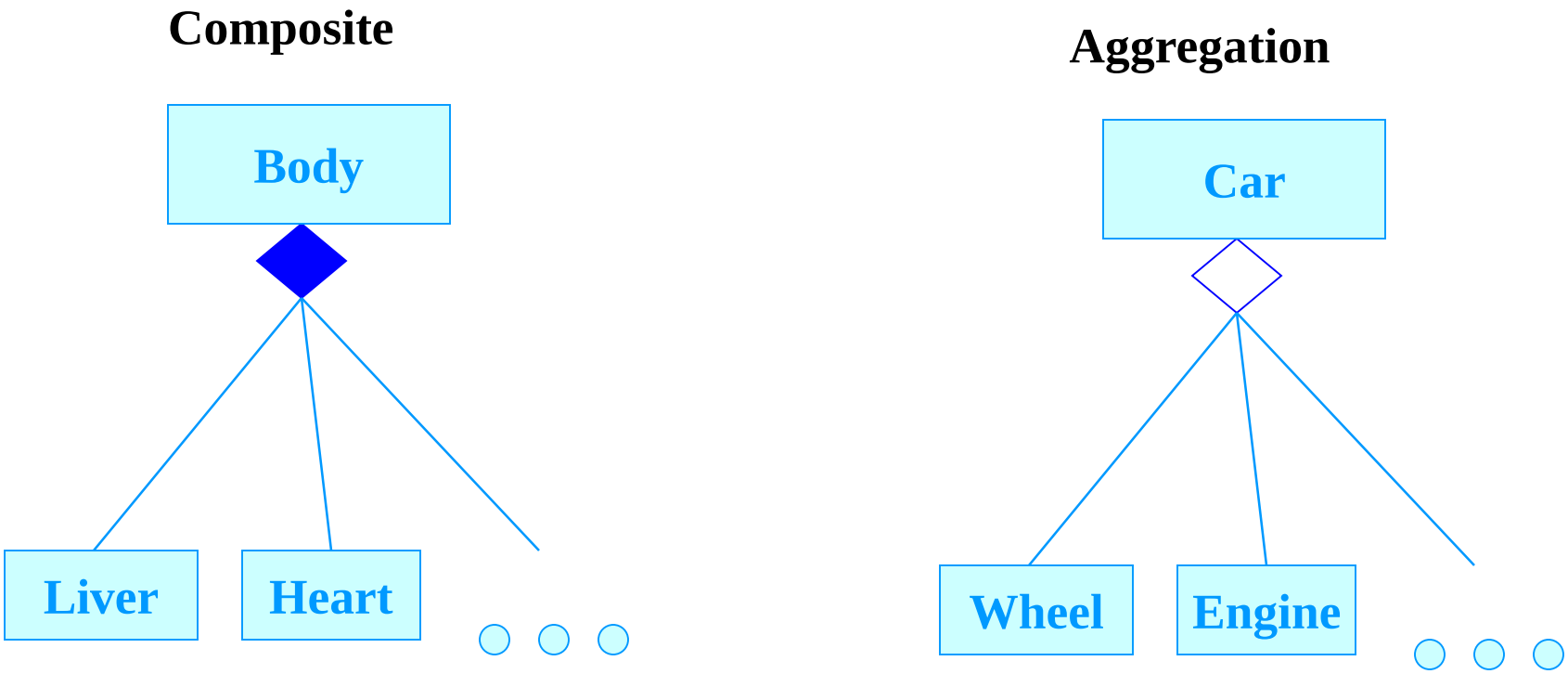
Modeling Structural Relationships

- Considering a bunch of classes and their association relationships



composite symbol (♦)get loaded versus the aggregation

Modeling Structural Relationships



Composite is a stronger form of aggregation. Composite parts live and die with the whole.

Modeling Structural Relationships

Specify an association to create a navigation path between two objects (in either direction).

Specify an association if two objects interact with each other beyond operation arguments.

How do you know that objects of one class must interact with another class?

- Review the scenarios that were derived from Use Cases.
- CRC cards seem much less used in practice..

Specify multiplicity; 1 is assumed.

Specify aggregation when one of the classes represents a whole over the other classes.

Hints & Tips

Modeling relationships

Use dependencies when relationship is not structural.

Use generalization with “is-a” relationship.

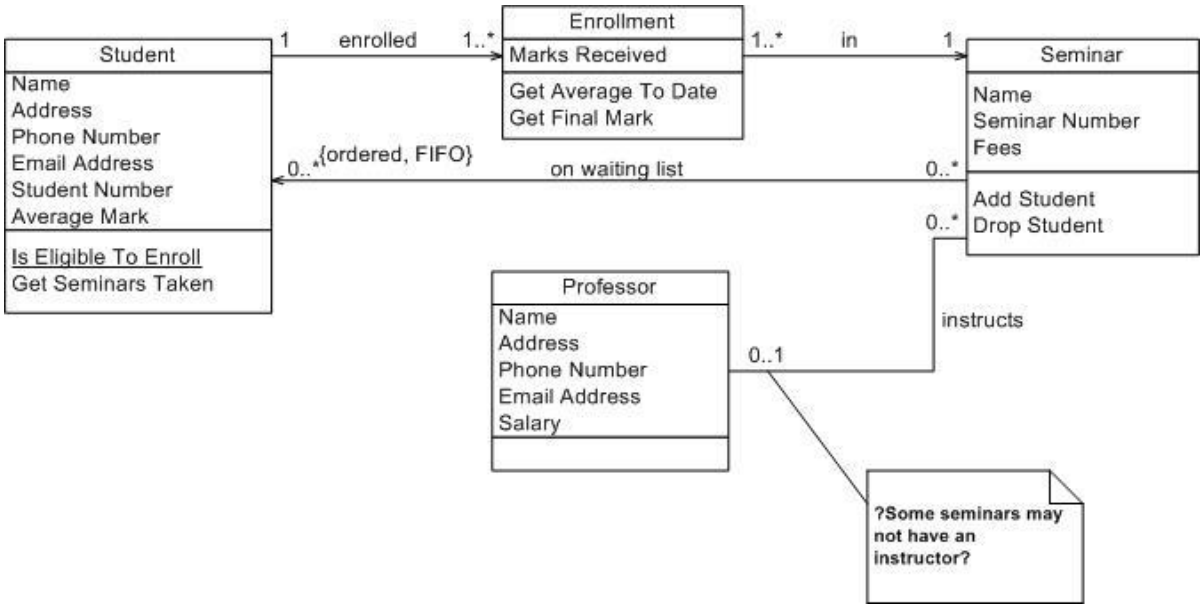
Don’t introduce cyclical generalizations.

Balance generalizations - Not too deep, not too wide.

Use associations where structural relationships exist.

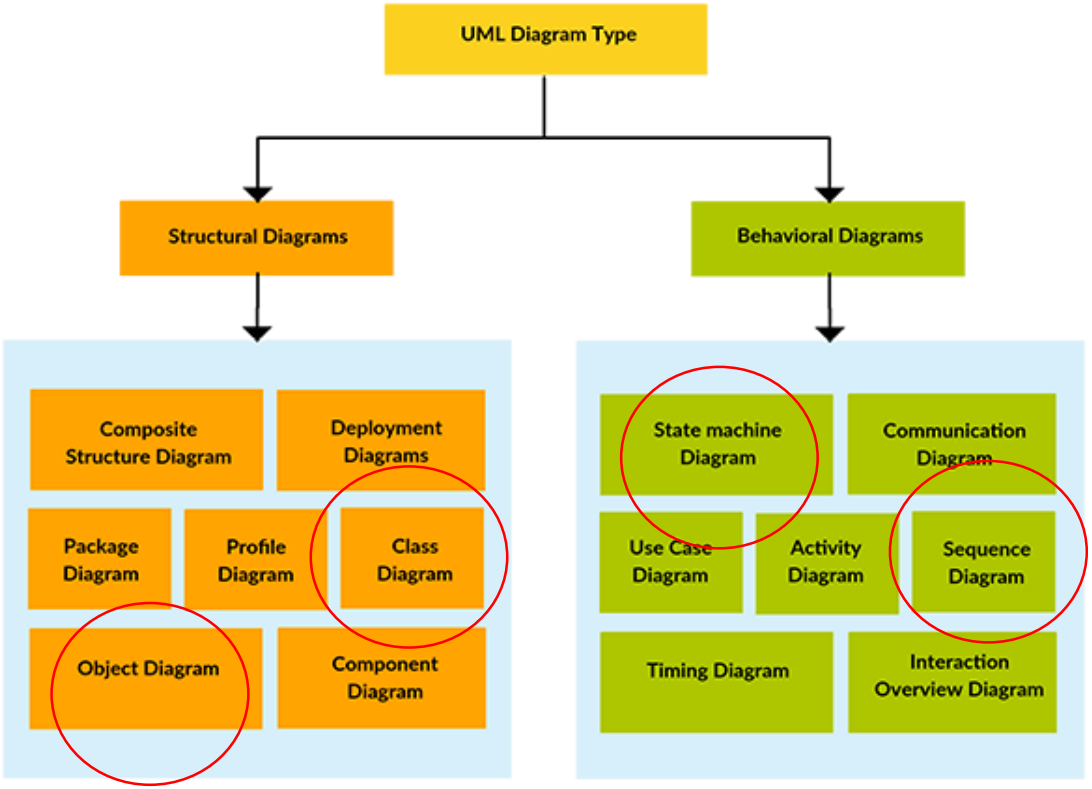
- Drawing a UML relationship
  - Use rectilinear or oblique lines consistently.
  - Avoid lines that cross.
  - Show only relationships necessary to understand a particular grouping of things.
  - Elide redundant associations.

Fast Review: Class Diagram



Here is the relation tuple:
{Type, Multiplicity, Name, Roles}

Fast Review: UML Diagram



What is Analysis and Design?

What is Analysis and Design?

Analysis emphasizes an investigation of the problem and rrequirements, rather than a solution. For example, if a new computerized library information system is desired, how will it be used?

Design emphasizes a conceptual solution that fulfills the requirements, rather than its implementation. For example, a description of a database schema, object model and dynamic model. Ultimately, designs can be implemented.

Analysis and design have been summarized in the phase «do the right thing» (analysis), and «do the thing right» (design)

When?

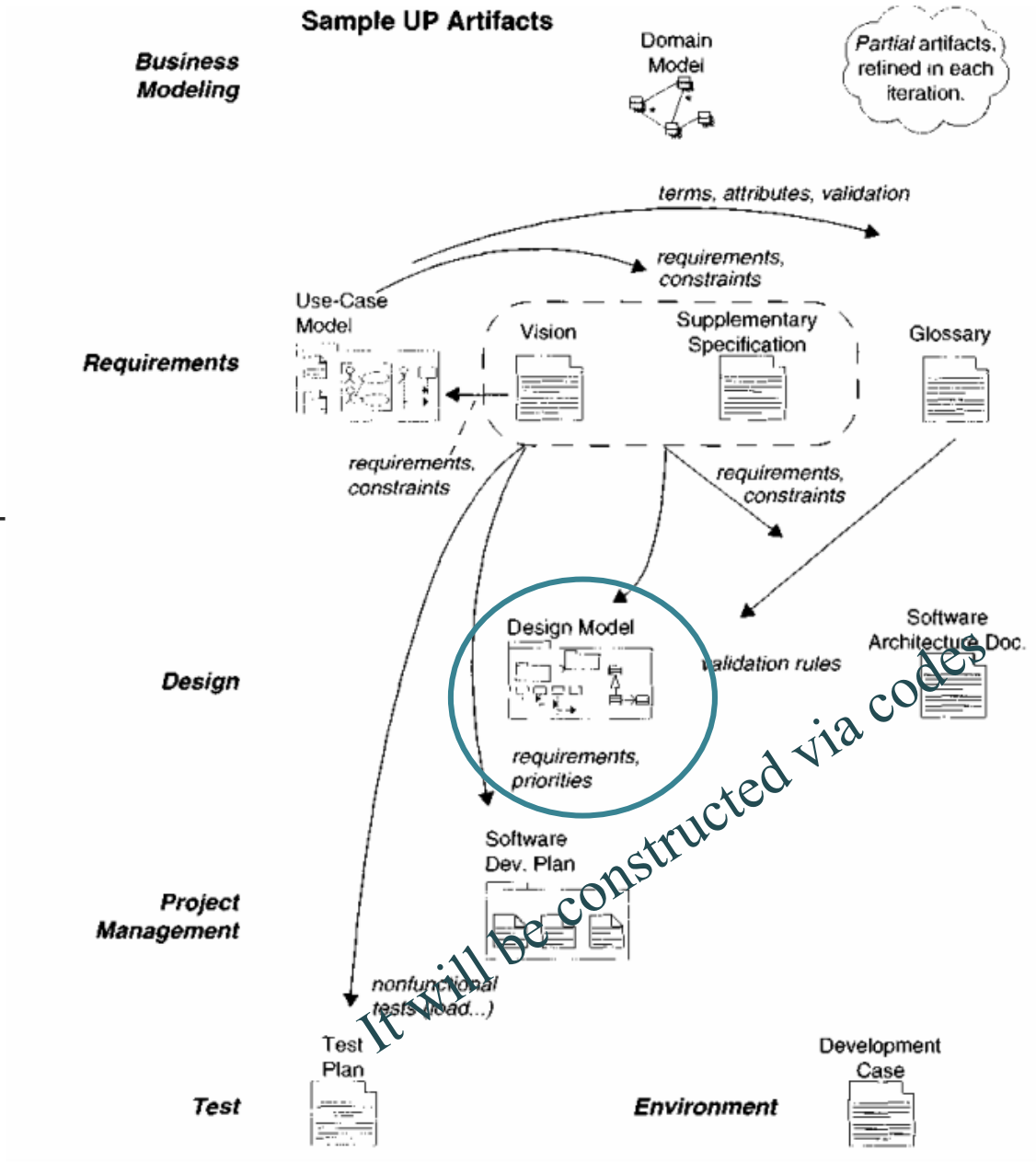
Discipline	Artifact Iteration-*	Incep. 11	Elab. El. .En	Const. CL.Cn	Trans. T1..T2
Business Modeling	Domain Model		s		
Requirements	Use-Case Model	s	r		
	Vision	s	r		
	Supplementary Specification	s	r		
	Glossary	s	r		
Design	Design Model		s	r	
	SW Architecture Document		s		
	Data Model		s	r	
Implementation	Implementation Model		s	r	r
Project Management	SW Development Plan	s	r	r	r
Testing	Test Model		s	r	
Environment	Development Case	s	r		

Sample Development Case of UP artifacts, s - start; r - refine

Inputs?

Inputs?

Requirements are capabilities and conditions to which the system—and more broadly, the project–must conform

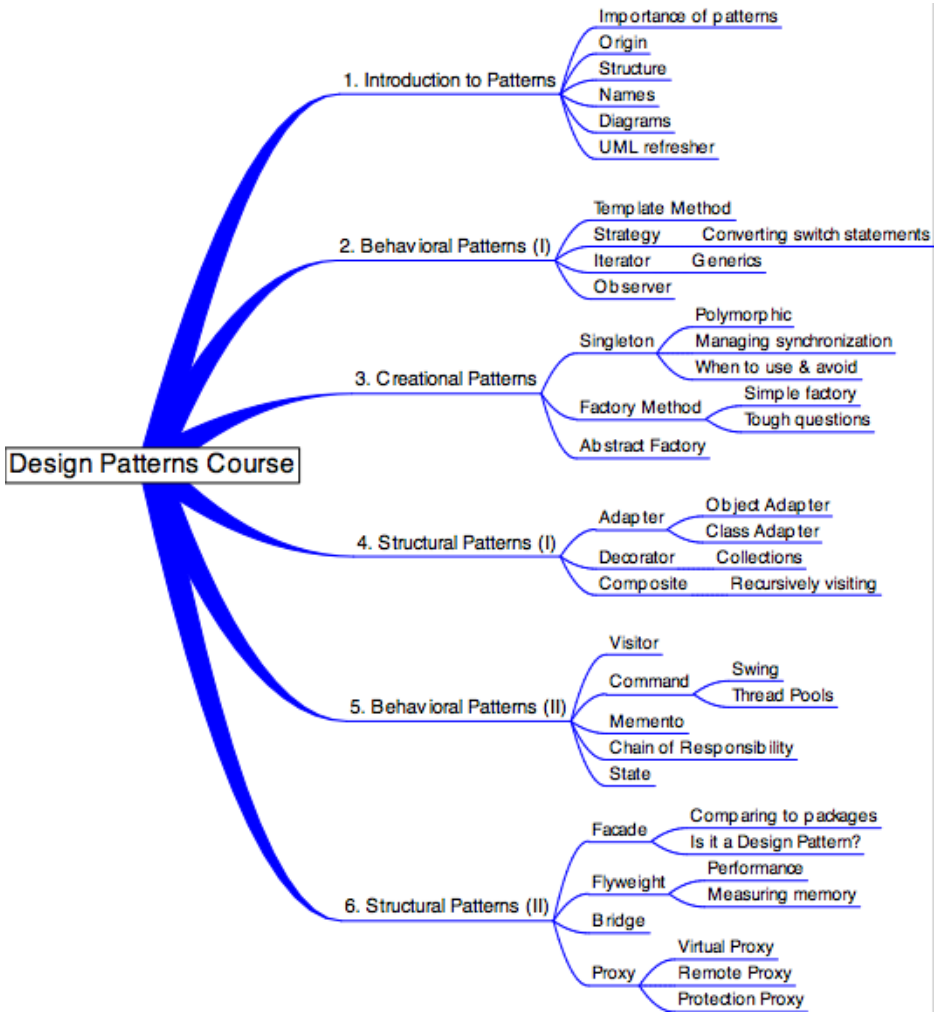


Sample UP artifact influence.

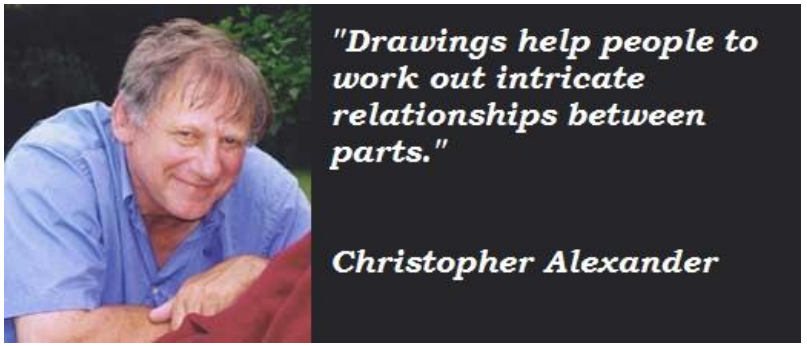
Let’s discuss

Accuracy ?
Flexibility?

Outline of Design Patterns



In 1977…



Christopher Alexander (born 1936), architect, has been worked on the effects of designs on the structures (building, cities, centers) and, he has tried on the design quality

He used «Design Pattern» name to describe the problems that occur and occur in different scales and can be solvable in the same manner

Definition from the father

Design Pattern definition from Alexander:

“Each pattern describes a problem

1)which occurs over and over again in our environment and then

2)describes the core of the solution to that problem, in such a way that

3)you can use this solution a million times over, without ever doing it the same way twice”

Smart Summary

Design pattern is the solution outline for the problems triggered by same reasons

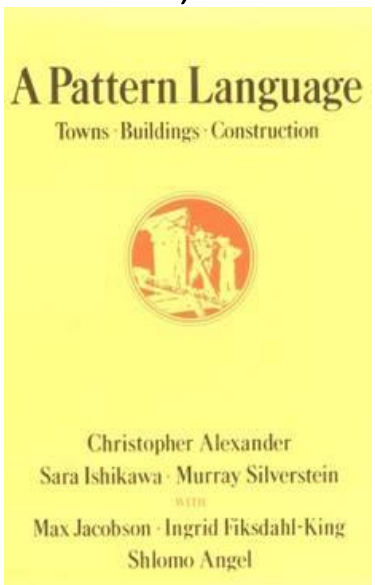
Patterns = problem/solution pairs in context

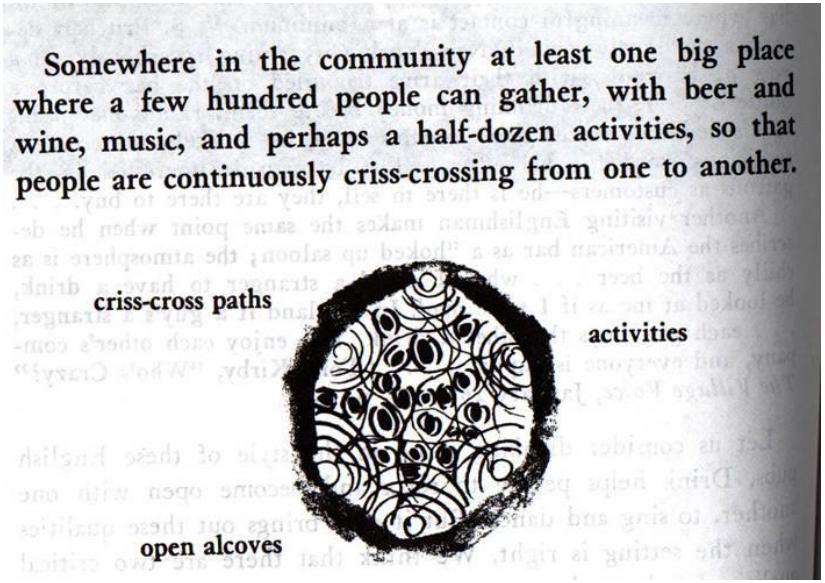
Patterns facilitate reuse of successful software architectures and design

Not code reuse !!!

Instead, solution/strategy reuse

Sometimes, interface reuse





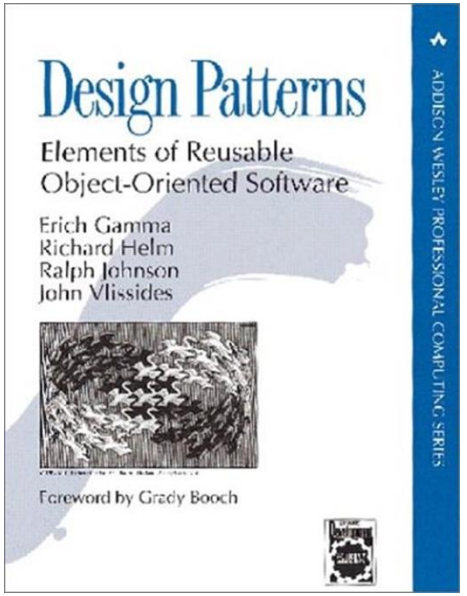
From Architectural design 2 software design

At the early 1990s, software design issues were discussed by considering two basic question

Are there any problems occuring repeatedly and each solution carries some similiar structures?

Is it possible to desing a software by using patterns even domains may change from one to another

The book that started it all
Community refers to authors as the “Gang of Four”
Figures and some text in these slides
come from this book



Components of a Pattern

Pattern name
- identify this pattern; distinguish from other patterns
- define terminology

Pattern alias – "*also known as*"

Real-world example

Context / Problem

Solution
- typically natural language notation

Structure
- class (and possibly object) diagram in solution

Interaction diagram (optional)

Consequences
- advantages and disadvantages of pattern
- ways to address residual design decisions

Components of a Pattern (cont'd)

Implementation

critical portion of plausible code for pattern

Known uses

often systems that inspired pattern

References - See also

related patterns that may be applied in similar cases

Why this book is serious

Because,

-First announcement of Design Patterns for us

-Total 23 patterns are catalogued with all components

-They triggered to revise principles OO Modeling

-The opinion of reusable solution/experience emerged

-Reliability, flexibility and robustness of these solutions have been tested in different situations and in different domains

-Common terminology is extended by means of patterns

-Pattern based analysis of the problem makes modeler more aware about the quality

-Patterns make software more flexible and easier to change

Basic principles of GoF

All of 23 patterns suggest same 3 things:

“Design interfaces”

“Favour aggregation over inheritance”

“Find what varies and encapsulate it”

Principles Underlying Patterns

Rely on abstract classes to hide differences between subclasses from clients

- object class vs. object type
  - *class defines how an object is implemented*
  - *type defines an object's interface (protocol)*

Program to an interface, not an implementation

Principles (cont'd)

Black-box vs. white-box reuse

- black-box relies on object references, usually through instance variables
- white-box reuse by inheritance
- black-box reuse preferred for information hiding, run-time flexibility, elimination of implementation dependencies
- disadvantages: Run-time efficiency (high number of instances, and communication by message passing)

Favor composition over class inheritance

Principles (cont'd)

Delegation

powerful technique when coupled with black-box reuse

Allow delegation to different instances at run-time, as long as instances respond to similar messages

disadvantages:

sometimes code harder to read and understand

efficiency (because of black-box reuse)

Find what varies and encapsulate it

Design patterns taxonomy

Creational patterns

concern the process of object creation

Structural patterns

deal with the composition of classes or objects

Behavioral patterns

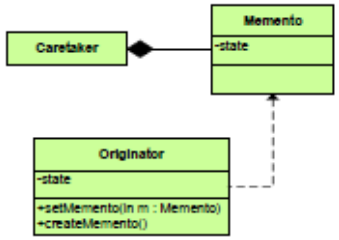
characterize the ways in which classes or objects interact and distribute responsibility.

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

Memento

Type: Behavioral

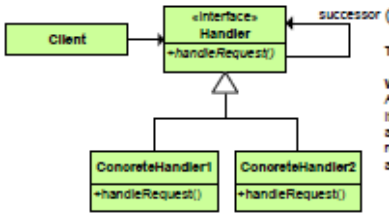
What it is:
Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.



Chain of Responsibility

Type: Behavioral

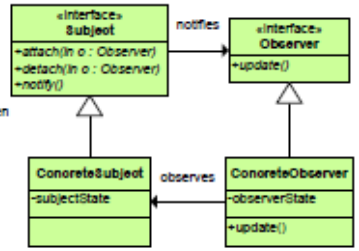
What it is:
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.



Observer

Type: Behavioral

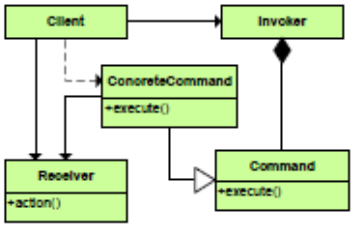
What it is:
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



Command

Type: Behavioral

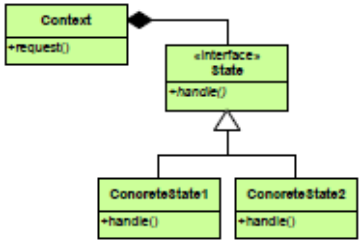
What it is:
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.



State

Type: Behavioral

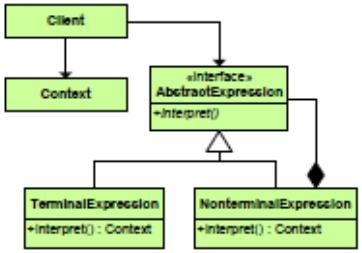
What it is:
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.



Interpreter

Type: Behavioral

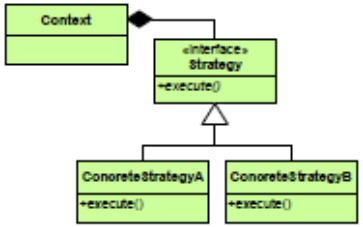
What it is:
Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.



Strategy

Type: Behavioral

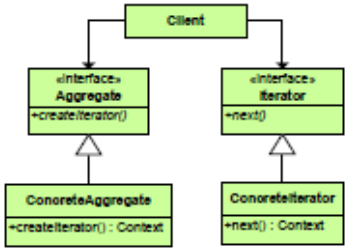
What it is:
Define a family of algorithms, encapsulate each one, and make them interchangeable. Lets the algorithm vary independently from clients that use it.



Iterator

Type: Behavioral

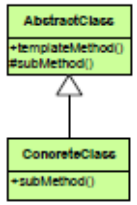
What it is:
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

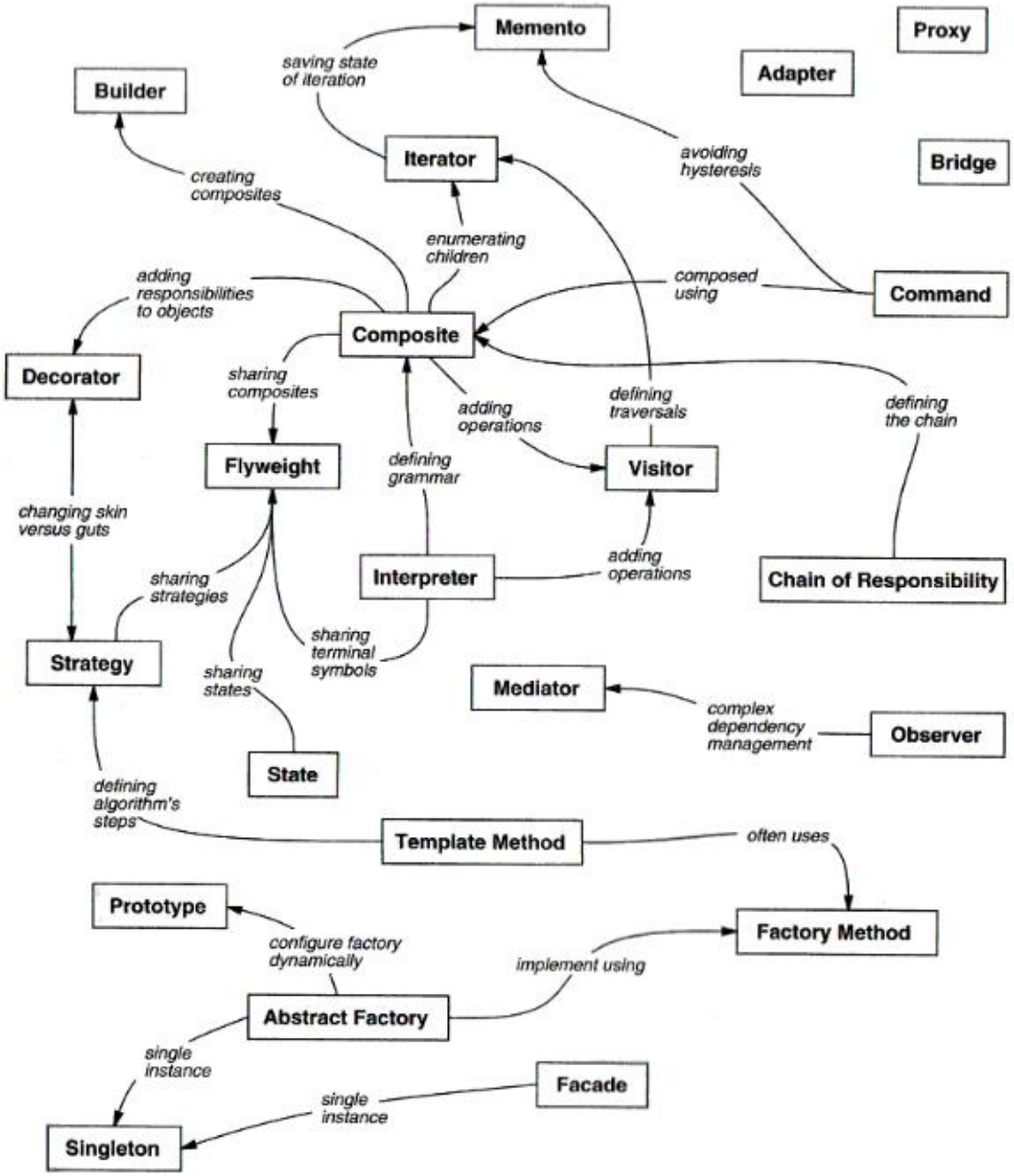


Template Method

Type: Behavioral

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




Memento
Proxy
saving state of iteration
Adapter
Builder
Iterator
avoiding hysteresis
Bridge
creating composites
enumerating children
adding responsibilities to objects
composed using
Command
Composite
Decorator
sharing composites
adding operations
defining traversals
defining the chain
Flyweight
defining grammar
Visitor
changing skin versus guts
sharing strategies
Interpreter
adding operations
Chain of Responsibility
sharing terminal symbols
Strategy
sharing states
Mediator
complex dependency management
Observer
State
defining algorithm's steps
Template Method
often uses
Prototype
Factory Method
configure factory dynamically
implement using
Abstract Factory
single instance
Facade
single instance
Singleton

When and How Design patterns are used

Whatever Software Life Cyle Model is applied
(i.e. RUP, Scrum, XP, Waterfall)

While building analysis model
and
Modeling of software requirements

Design Patterns are also known as GRASP Patterns

"GRASP is an acronym that stands for ***G***eneral ***R***esponsibility ***A***ssignment ***S***oftware ***P***atterns"

The name was chosen to suggest the importance of ***grasping*** these principles to successfully design object-oriented software

GRASP Patterns

Do not state new ideas

Name and codify widely used basic principles

Responsibilities

UML defines a responsibility as “a contract or obligation of a classifier”.

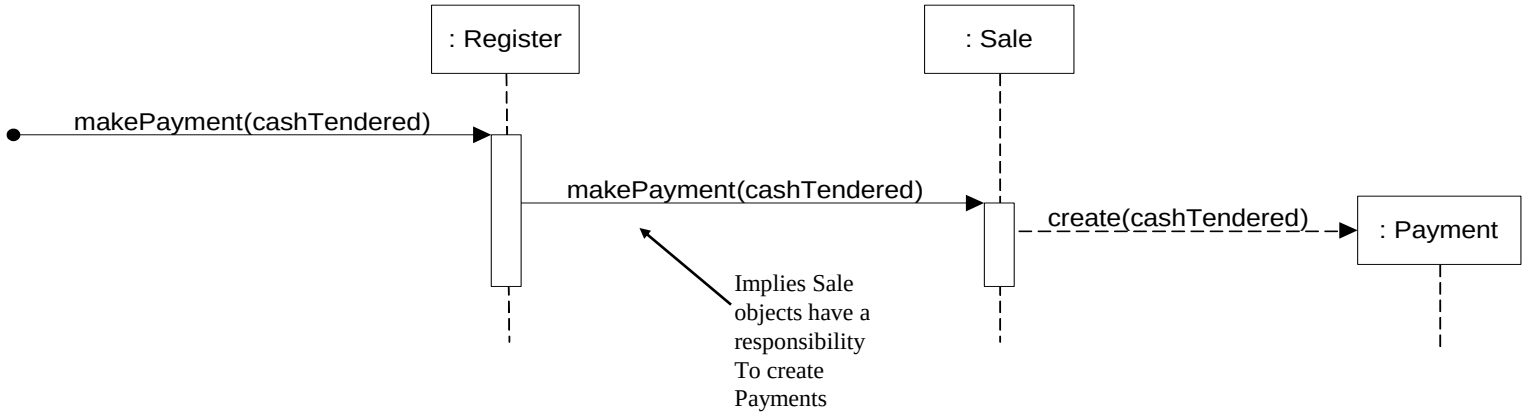
A class embodies a set of **responsibilities** that define the behaviour of the objects in the class.

Responsibilities

"A responsibility is not the same thing as a method, but methods are implemented to fulfill responsibilities."

"Responsibilities are implemented using methods that either act alone or ***collaborate*** with other methods and objects."

Responsibilities and methods are related



Responsibilities revolve around

Doing

Knowing

Collaboration

“Doing” responsibilities

Doing something itself, such as creating an object or doing a calculation

Initiating action in other objects

Controlling and co-coordinating activities in other objects

“Knowing” responsibilities

Knowing about private encapsulated data

Knowing about related objects

Knowing about things that it can derive or calculate

GRASP Patterns

Key three:

- Creator
- Controller
- Information Expert

Creator

Who should be responsible for creating an new instance of some class?

Some options:

Assign B the responsibility to create A if one or more of the following is/are true:

B "contains" A (e.g. Invoice creates InvoiceLineItem)

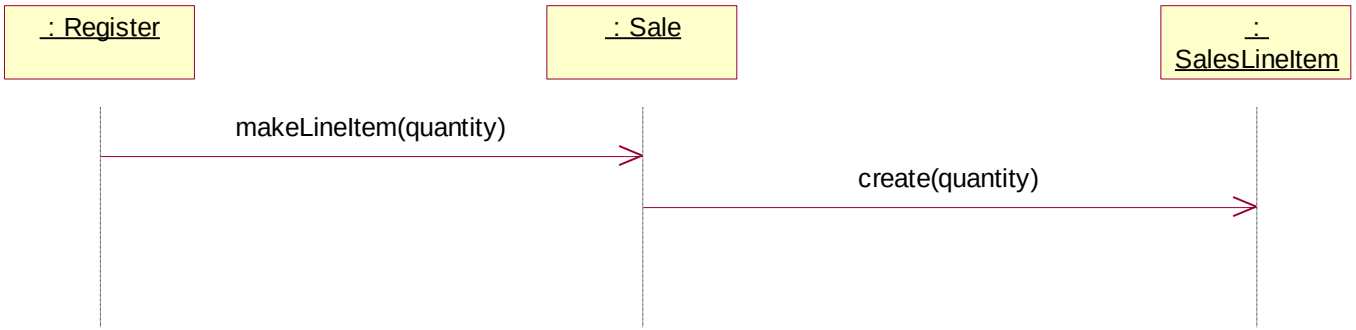
B records A

B closely uses A

B has the initializing data for A that will be passed to A when it is created (thus B is the Expert with respect to creating A). (e.g. Sale creates Payment)

Do not distribute the creation knowledge of A

Creating a SalesLineItem



Controller

What first object beyond the UI layer receives and coordinates a system operation?

- *Use Case or Session Controller*
  - *Use case/session (e.g. Register)**

Guidelines/Issues

Controller usually delegates work to other objects—it controls, coordinates, it does not do much work itself

Danger: Bloated controller

- a single controller receives all system events (and there are many)
- a controller that does the work itself
- a controller that has many attributes; maintains significant information

Among Cures for Bloat

- more controllers, use case controllers, more delegation

Information Expert

What is the general principle of assigning responsibilities to objects?

A Solution:

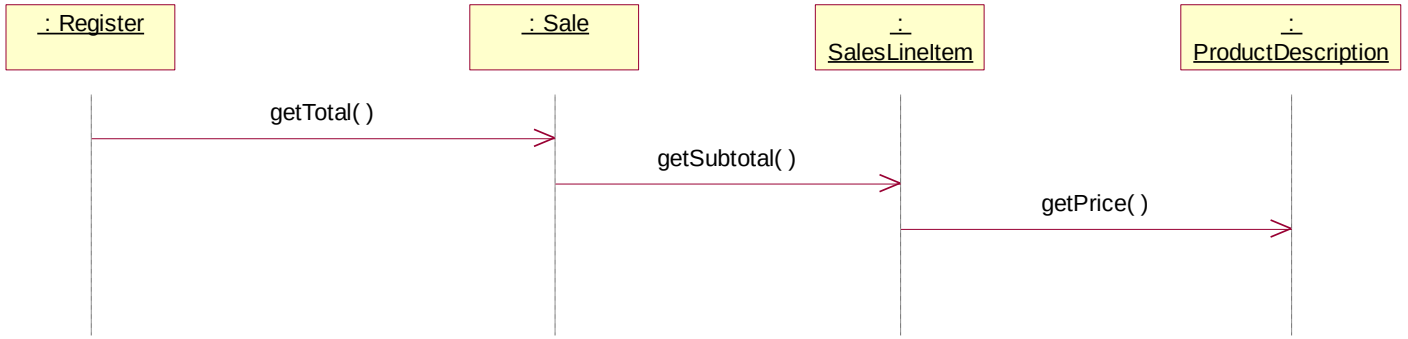
Assign a responsibility to the class that has the information necessary to fulfill it—the "information expert"

(note: start this process by clearly stating the responsibility!)

Information Expert

Example: Sale has the responsibility of knowing its total, expressed with the method named *getTotal*

Information Expert



Collaboration

Fulfillment of a responsibility often requires information from different classes of objects

Example, sales total requires the collaboration of 3 classes of objects: Sale, SalesLineItem, ProductDescription

Interact via messages*

Appendix:Analysis Model

In Analysis, we analyze and refine the requirements described in the Use Cases in order to achieve a more precise view of the requirements, without being overwhelmed with the details

Again, the Analysis Model is still focusing on WHAT we're going to do, not HOW we're going to do it (Design Model). But what we're going to do is drawn from the point of view of the developer, not from the point of view of the customer

Whereas Use Cases are described in the language of the customer, the Analysis Model is described in the language of the developer:

- Boundary Classes
- Entity Classes
- Control Classes

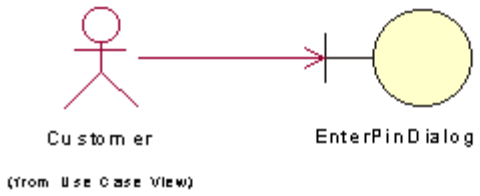
Appendix:Boundary Classes (out of DP's Scope)

Boundary classes are used in the Analysis Model to model interactions between the system and its actors (users or external systems)

Boundary classes are often implemented in some GUI format (dialogs, widgets, beans, etc.)

Boundary classes can often be abstractions of external APIs (in the case of an external system actor)

Every boundary class must be associated with at least one actor:



Appendix: Entity Classes (in the DP's Scope)

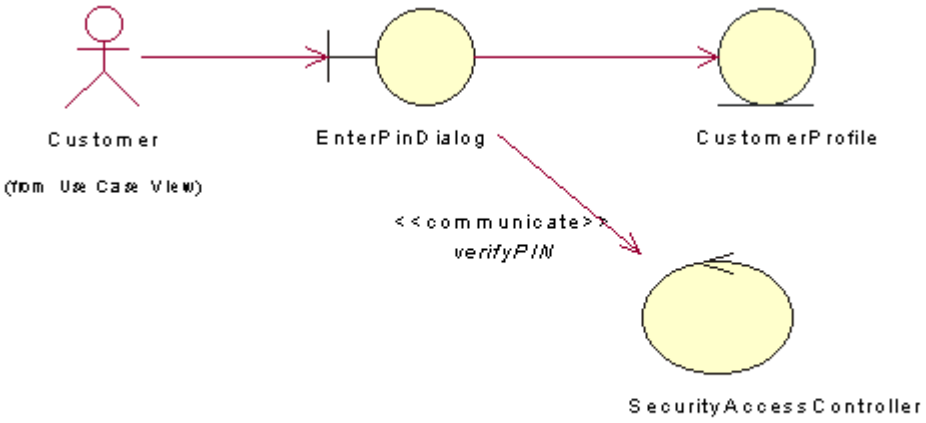
Entity classes are used within the Analysis Model to model persistent information

Often, entity classes are created from objects within the business object model or domain model

Appendix: Control Classes (in the DP's Scope)

Control classes model abstractions that coordinate, sequence, transact, and otherwise control other objects

Control classes are often encapsulated interactions between other objects, as they handle and coordinate actions and control flows.



Appendix: Requirements/Glossary

Course Registration System
Glossary

Version 2.0

Revision History

Date	Version	Description	Author
26/Dec/1998	1.0	Draft version	Bill Collings
19/Feb/1999	2.0	Moved some of the terms to the Wylie College glossary.	Bill Collings

Glossary

1.Introduction

The glossary contains the working definitions for terms and classes in the Course Registration System. This glossary will be expanded throughout the life of the project. Any definitions not included in this document may be included in the Rational Rose Model. Generic terms used outside this project should be captured in the organizational Glossary.

2. Definitions

Alternative course selection

A student might choose to register for one or more alternative courses, in case one or more of the primary selections are not available.

Billing System

Part of the university's Finance System used for processing billing information.

Appendix:Requirements/Stakeholder Requests

This artifact contains any type of requests, a stakeholder (customer, end user, marketing person, and so on) might have on the system to be developed.

It may also contain references to any type of external sources to which the system must comply.

Although the **system analyst** is responsible for this artifact, many people will contribute to it: **marketing people**, **end users**, **customers**-anyone who is considered to be a stakeholder to the result of the project.

Appendix:Stakeholder Requests

Stakeholder requests are mainly collected during the inception and elaboration phases, however you should continue to collect them throughout the project's lifecycle for planning enhancements and updates to the product.

A change request tracking tool is useful for collecting and prioritizing these requests.

Appendix:Storyboard

A **Storyboard** is a logical and conceptual description of system functionality for a specific scenario, including the interaction required between the system users and the system. A Storyboard "tells a specific story".

System Analyst

Optional. Produced in early Elaboration, during requirements elicitation.

Appendix: Requirements/Storyboard

The following people use the Storyboards:

system analysts, to explore, clarify, and capture the behavioral interaction envisioned by the user as part of requirements elicitation.

user-interface designers, to design the user interface and to build a prototype of the user interface;

designers of the classes that provide the user interface functionality; They use this information to understand the system's interactions with the user, so they can properly design the classes that will implement the user interface;

those who design the next version of the system to understand how the system carries out the flow of events;

those who test to test the system's features;

the manager to plan and follow up the analysis & design work.

Appendix: Software Requirements Specification

The Software Requirements Specification (SRS) captures the software requirements for the complete system, or a portion of that system.

The Requirements Specifier role specifies and maintains the detailed system requirements.

Considered first in the Inception phase, refined in the Elaboration and Construction phases.

Appendix: Software Requirements Specification

The following people use the Software Requirements Specification:

The system analyst creates and maintains the Vision and Supplementary Specifications, which serves as input to the SRS and are the communication medium between the system analyst, the customer, and other developers.

The requirements specifier creates and maintains the individual use case and other components of the SRS package,

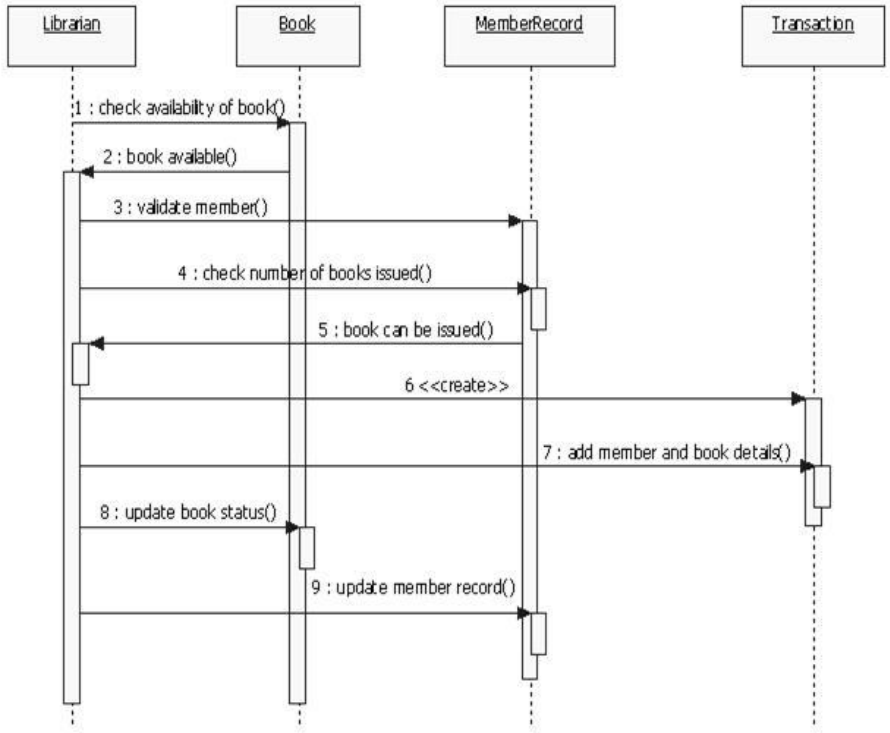
Designers use the SRS Package as a reference when defining responsibilities, operations, and attributes on classes, and when adjusting classes to the implementation environment.

Implementers refer to the SRS Package for input when implementing classes.

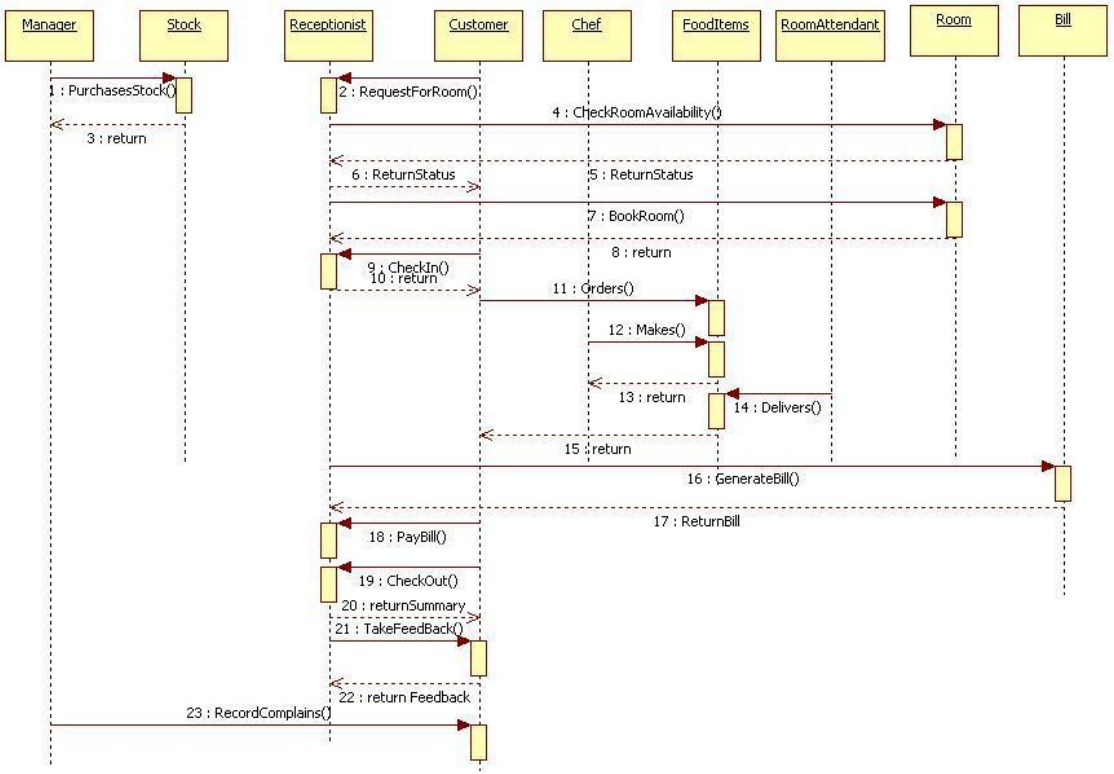
The Project Manager refers to the SRS Package for input when planning iterations.

Testers use the SRS Package as an input to considering what tests will be required.

Fast Review: Sequence Diagram



Fast Review: Sequence Diagram



Fast Review: State Diagram