Design Patterns (I)

Chung-Kil Hur

(Credit: Byung-Gon Chun &Many Slides from UCB CS169 taught by Armando Fox, David Patterson, George Necula)

SWPP, CSE, SNU

The process of preparing programs for a digital computer is especially attractive, not only because it can be economically and scientifically rewarding, but also because it can be an aesthetic experience much like composing poetry or music. -- Donald Knuth

Patterns, Antipatterns, and SOLID

Code Smell and Refactoring

- Start with code that has 1 or more problems/smells (shotgun surgery, data clump, inappropriate intimacy, repetitive boilerplate, etc.)
- Through a series of *small steps,* transform to code from which those smells are absent
- Protect each step with tests
- Minimize time during which tests are red

SOFA Principles for Good Method Design

SOFA

- Be short
- Do one thing
- Have few arguments
- Consistent level of abstraction

Approach We’re Taking

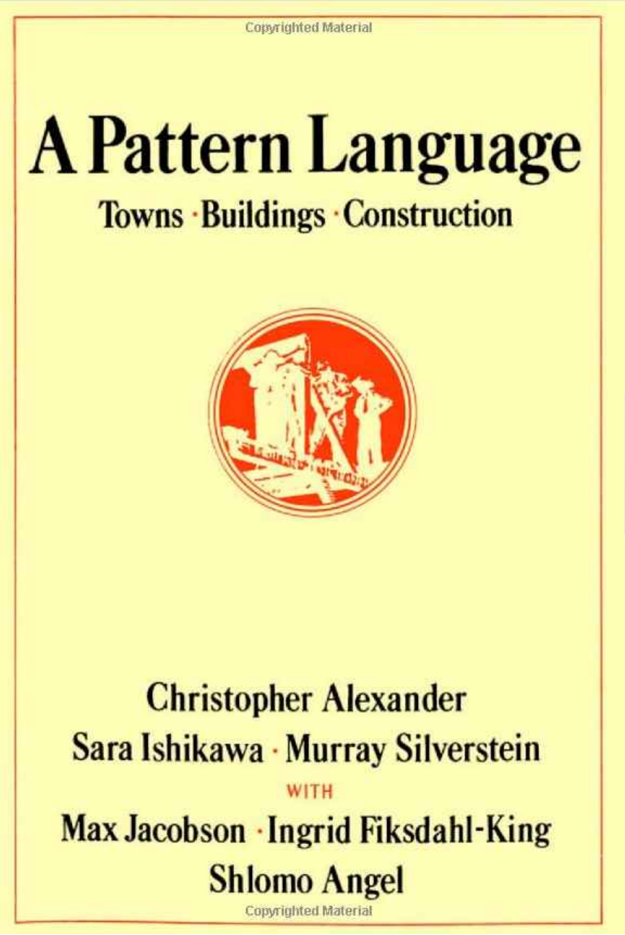
- Design smell : anti-pattern which indicates poor class design
- Refactoring a design: how refactoring the bad code can fix the violation
- Apply design patterns, which apply to classes and class architecture

- Motivate the use of design patterns by starting from some guidelines

Design Patterns Promote Reuse

“A pattern describes a problem that occurs often, along with a tried solution to the problem” - Christopher Alexander, 1977

- Christopher Alexander’ s 253 (civil) architectural patterns range from the creation of cities (2. distribution of towns) to particular building problems (232. roof cap)
- A pattern language is an organized way of tackling an architectural problem using patterns
- *Separate the things that change from those that stay the same*



A Pattern Language

Towns · Buildings · Construction

Christopher Alexander

Sara Ishikawa · Murray Silverstein

WITH

Max Jacobson · Ingrid Fiksdahl-King

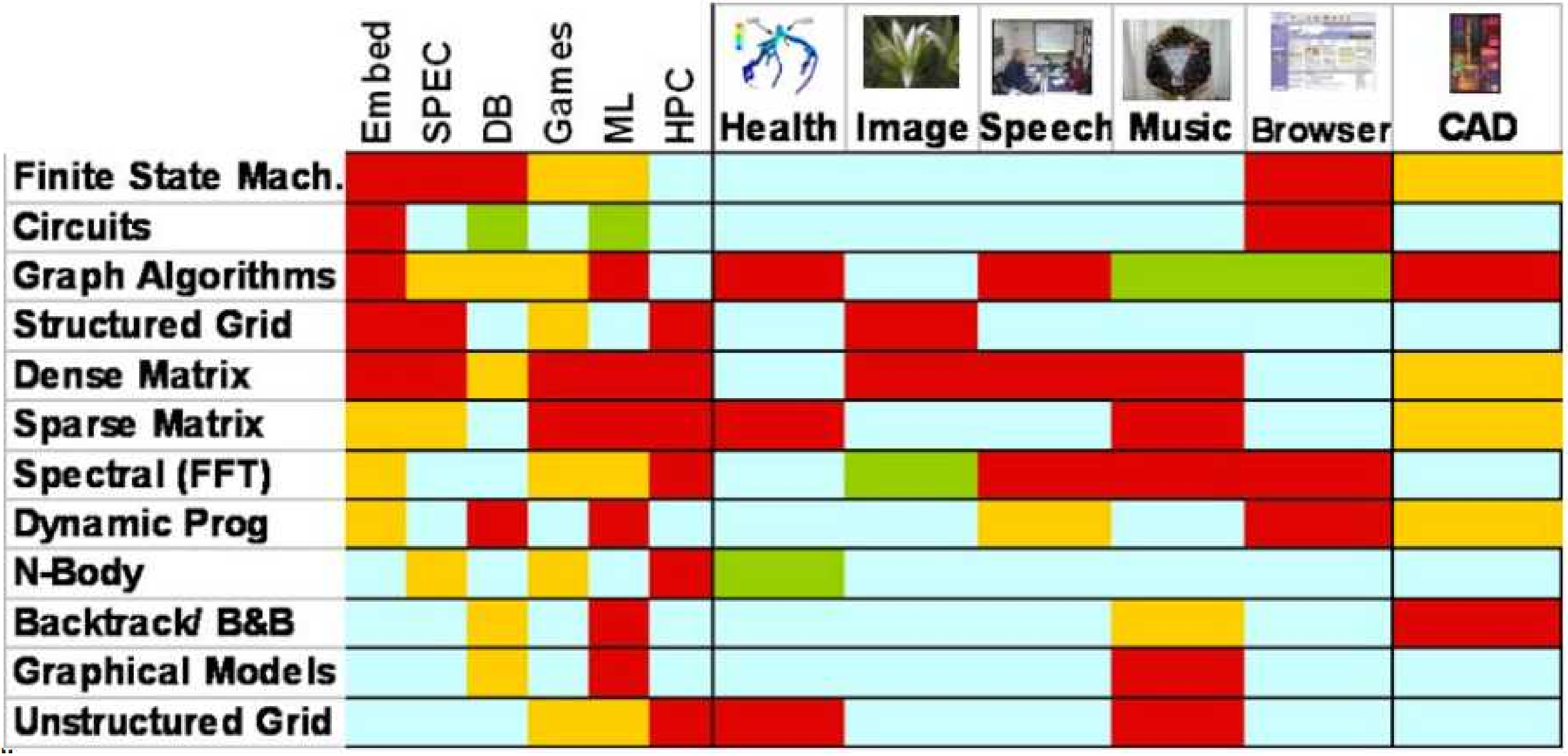
Shlomo Angel

Kinds of Patterns in Software

- Architectural (“macroscale”) patterns
  - Model-view-controller
  - Pipe & Filter (e.g. compiler, Unix pipeline)
  - Event-based (e.g. interactive game)
  - Layering (e.g. SaaS technology stack)
  - Map-Reduce
- Computation patterns
  - Fast Fourier transform
  - Structured & unstructured grids
  - Dense linear algebra
  - Sparse linear algebra

Computation Patterns

The Dwarfs from "The Berkeley View" (Asanovic et al.)
Dwarfs form our key computational patterns



	Embed	SPEC	DB	Games	ML	HPC	Health	Image	Speech	Music	Browser	CAD
Finite State Mach.	🟥	🟥	🟥	🟨	🟨	🟦	🟦	🟦	🟦	🟦	🟥	🟨
Circuits	🟥	🟦	🟩	🟦	🟩	🟦	🟦	🟦	🟦	🟦	🟥	🟦
Graph Algorithms	🟥	🟨	🟨	🟨	🟥	🟦	🟥	🟦	🟥	🟩	🟩	🟥
Structured Grid	🟥	🟥	🟦	🟨	🟦	🟥	🟦	🟥	🟦	🟦	🟦	🟦
Dense Matrix	🟥	🟥	🟨	🟥	🟥	🟥	🟦	🟥	🟥	🟥	🟦	🟨
Sparse Matrix	🟨	🟨	🟦	🟥	🟥	🟥	🟥	🟦	🟦	🟥	🟦	🟨
Spectral (FFT)	🟨	🟦	🟦	🟨	🟨	🟥	🟦	🟩	🟥	🟥	🟥	🟦
Dynamic Prog	🟨	🟦	🟥	🟦	🟥	🟦	🟦	🟦	🟨	🟦	🟥	🟨
N-Body	🟦	🟨	🟦	🟨	🟦	🟥	🟩	🟦	🟦	🟦	🟦	🟦
Backtrack/ B&B	🟦	🟦	🟨	🟦	🟥	🟦	🟦	🟦	🟦	🟨	🟦	🟥
Graphical Models	🟦	🟦	🟨	🟦	🟥	🟦	🟦	🟦	🟦	🟥	🟦	🟦
Unstructured Grid	🟦	🟦	🟦	🟨	🟨	🟥	🟥	🟦	🟦	🟥	🟦	🟦

Kinds of Patterns in Software

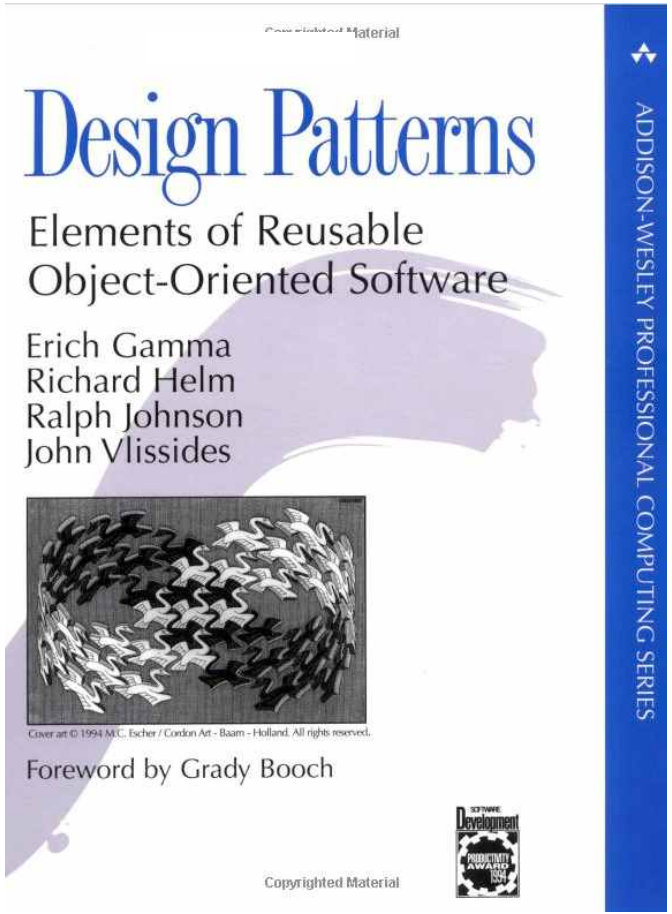
- Architectural (“macroscale”) patterns
  - Model-view-controller
  - Pipe & Filter (e.g. compiler, Unix pipeline)
  - Event-based (e.g. interactive game)
  - Layering (e.g. SaaS technology stack)
  - Map-Reduce
- Computation patterns
  - Fast Fourier transform
  - Structured & unstructured grids
  - Dense linear algebra
  - Sparse linear algebra
- *GoF (Gang of Four) Patterns: structural, creational, behavior*

Refactoring & Design Patterns

Methods within a class	Relationships among classes
Code smells	Design smells
Many catalogs of code smells & refactorings	Many catalogs of design smells & design patterns
Some refactorings are superfluous in Ruby	Some design patterns are superfluous in Ruby
Metrics: ABC & Cyclomatic Complexity	Metrics: Lack of Cohesion of Methods (LCOM)
Refactor by extracting methods and moving around code within a class	Refactor by extracting classes and moving code between classes
SOFA: methods are **S**hort, do **O**ne thing, have **F**ew arguments, single level of **A**bstraction	SOLID: **S**ingle responsibility per class, **O**pen/closed principle, **L**iskov substitutability, **I**njection of dependencies, **D**emeter principle

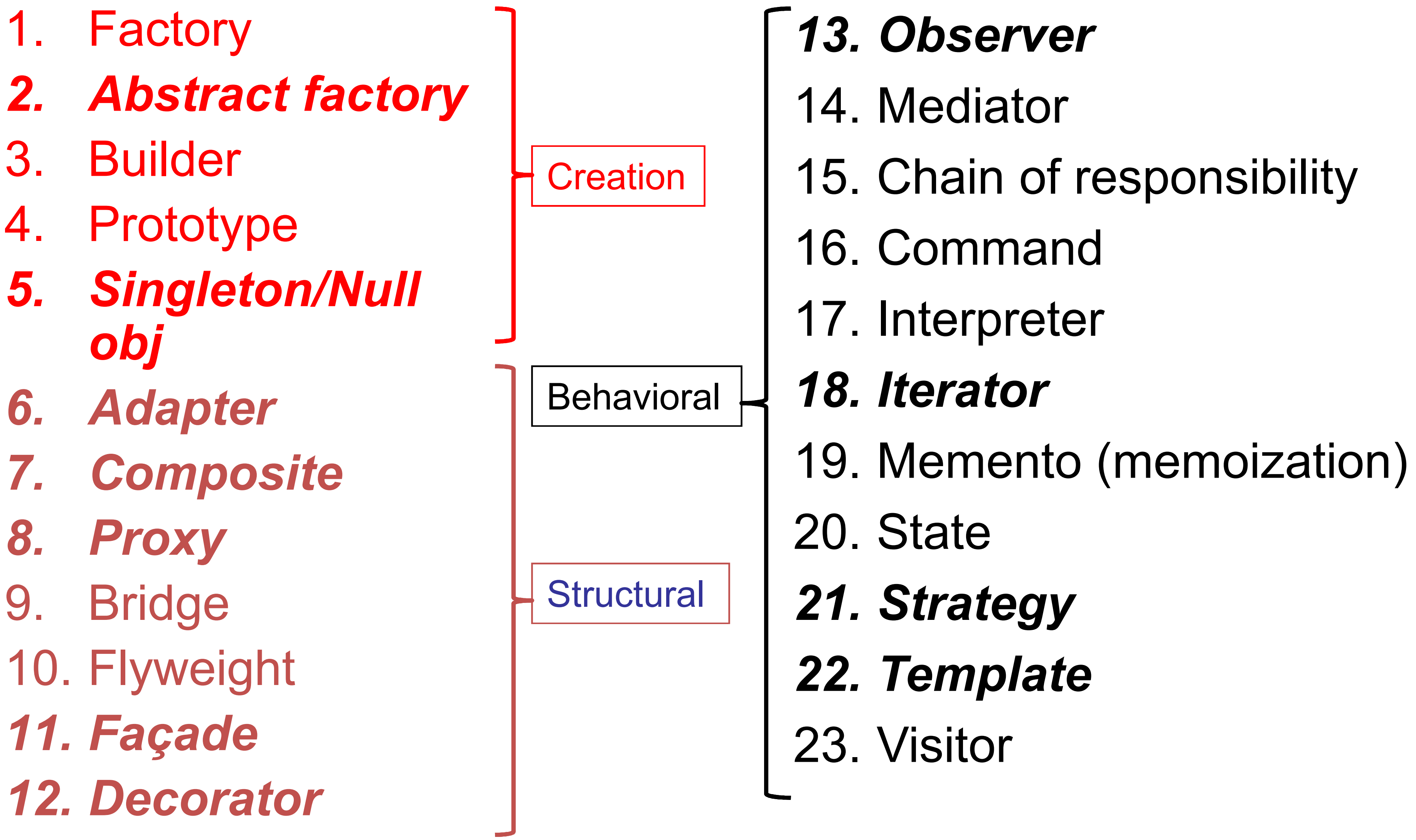
The Gang of Four (GoF)

- 23 *structural* design patterns
- description of communicating objects & classes
  - captures common (and successful) solution to a *category* of related problem instances
  - can be customized to solve a specific (new) problem in that category


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

- Pattern ≠
  - individual classes or libraries (list, hash, ...)
  - full design—more like a blueprint for a design

The GoF Pattern Zoo



1. Factory
2. ***Abstract factory***
3. Builder
4. Prototype
5. ***Singleton/Null obj***

Creation

6. ***Adapter***
7. ***Composite***
8. ***Proxy***
9. Bridge
10. Flyweight
11. ***Façade***
12. ***Decorator***

Structural

Behavioral

13. ***Observer***
14. Mediator
15. Chain of responsibility
16. Command
17. Interpreter
18. ***Iterator***
19. Memento (memoization)
20. State
21. ***Strategy***
22. ***Template***
23. Visitor

Principles of Good Object-Oriented Design that Inform Patterns

Separate out the things that change from those that stay the same

Two overarching principles cited by the GoF authors

1. Program to an Interface, not an Implementation
2. Prefer Composition and Delegation over Inheritance

Antipattern

- Code that looks like it should probably follow some design pattern, but doesn’t
- Often result of accumulated *technical debt*
- Symptoms:
  - Viscosity (easier to do hack than Right Thing)
  - Immobility (can’t DRY out functionality)
  - Needless repetition (comes from immobility)
  - Needless complexity from generality

SOLID OOP principles

(Robert C. Martin, co-author of Agile Manifesto)

Five design principles that clean code should respect

- Single Responsibility principle
- Open/Closed principle
- Liskov substitution principle
- Injection of dependencies
  - traditionally, Interface Segregation principle
- Demeter principle

Which statement is FALSE?

- □ Software that uses more design patterns isn’t necessarily better.
- □ Well-designed software can evolve to the point where patterns become antipatterns.
- □ Trying to apply design patterns too early can be just as bad as applying them too late.
- □ Most design patterns are specific to a particular subset of programming languages.

Just Enough UML

Modeling

- Describing a system at a high level of abstraction
  - A model of the system
  - Used fore requirements, specification, design

- Many notations over time
  - State machines
  - Entity-relationship diagrams
  - Dataflow diagrams

Recent History: 1980’s

- The rise of object-oriented programming language
- New class of OO modeling language
- By early 90’s, tens of modeling language

Recent History: 1990’s

- Three leading OO notations decide to combine
  - Grady Booch (BOOCH)
  - Jim Rumbaugh (OML: Object Modeling Technique)
  - Ivar Jacobsen (OOSE: OO Soft. Eng)

- Why?
  - Natural evolution towards each other
  - Effort to set an industry standard

UML

- UML stands for Unified Modeling Language

- Design by committee
  - Many interest groups participating
  - Everyone wants their favorite approach to be in

- Resulting design is huge
  - Many features
  - Many loosely unrelated styles under one roof

UML diagrams

- For example,
  - Class diagrams
  - Object diagrams

  for structural models

  - Sequence diagrams
  - Activity diagrams

  for dynamic models

- This is a subset of UML
  - But probably the most used subset

(Too Much UML)

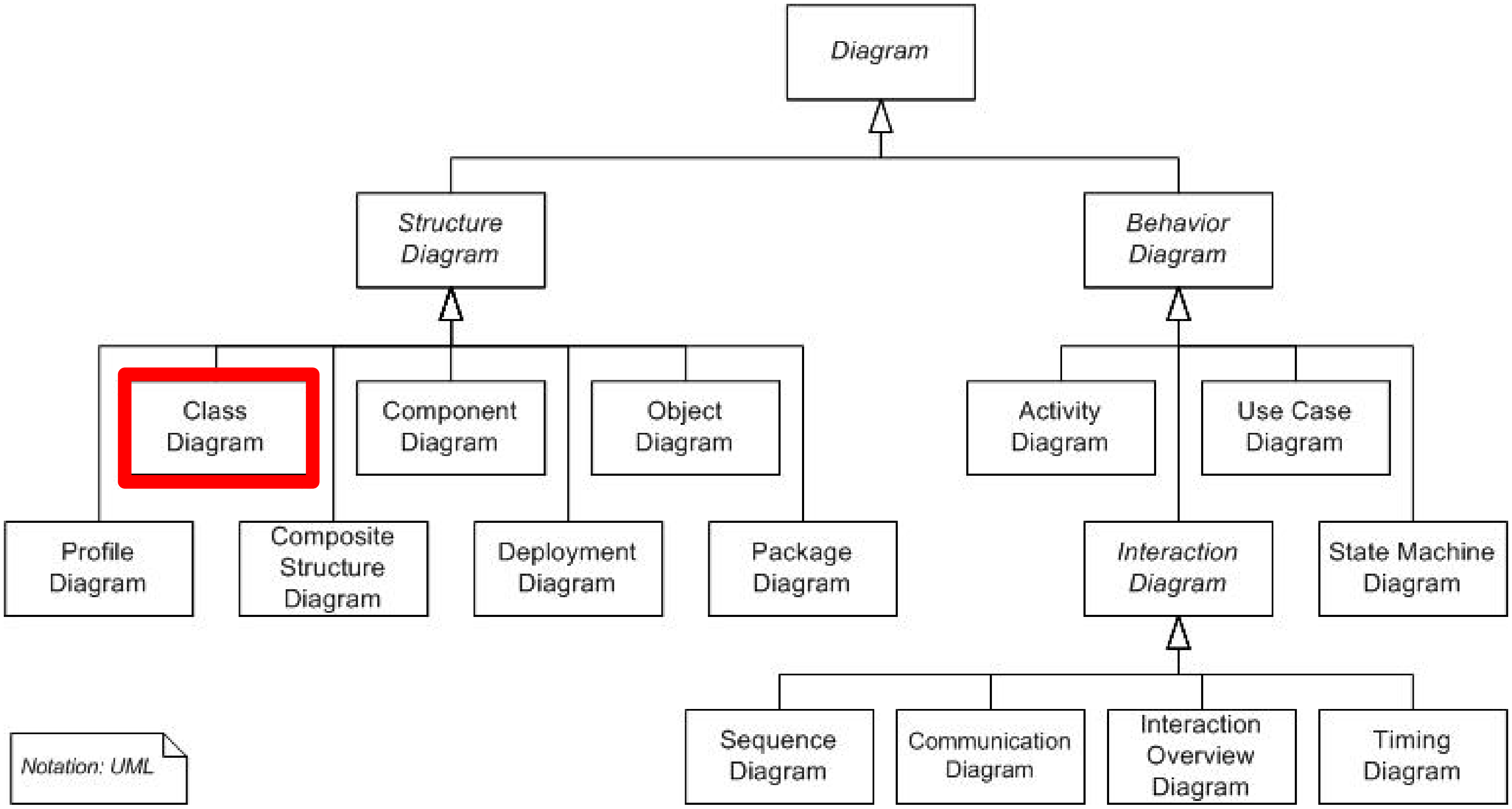
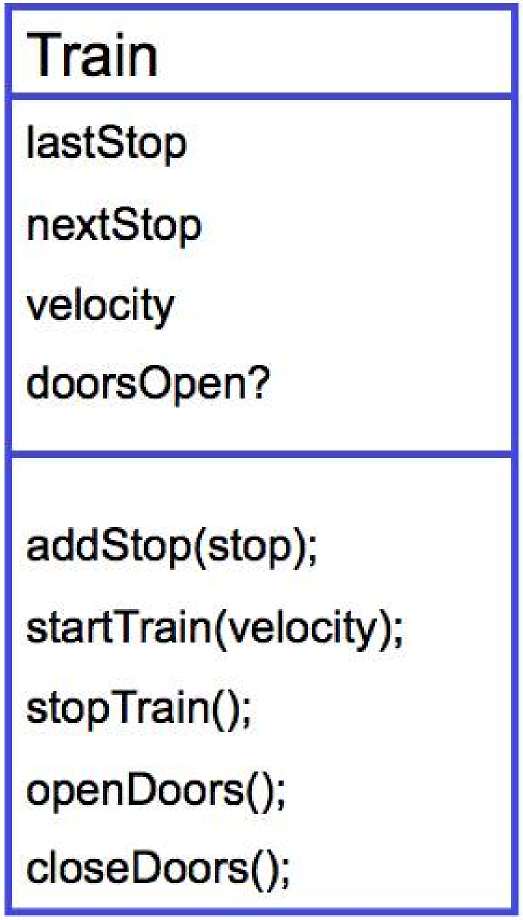


Diagram
Structure Diagram
Behavior Diagram
Class Diagram
Component Diagram
Object Diagram
Activity Diagram
Use Case Diagram
Profile Diagram
Composite Structure Diagram
Deployment Diagram
Package Diagram
Interaction Diagram
State Machine Diagram
Sequence Diagram
Communication Diagram
Interaction Overview Diagram
Timing Diagram
Notation: UML

Class Diagrams

- Describe classes
  - In the OO sense
  - Statically: what interacts with what, but not what happens
- Each box is a class
  - Name
  - (public) fields
  - (public) methods
- The more detail, the more it becomes a design



Train

lastStop

nextStop

velocity

doorsOpen?

addStop(stop);

startTrain(velocity);

stopTrain();

openDoors();

closeDoors();

Class Diagrams: Relationships

- Many different kinds of edges to show different relationships between classes

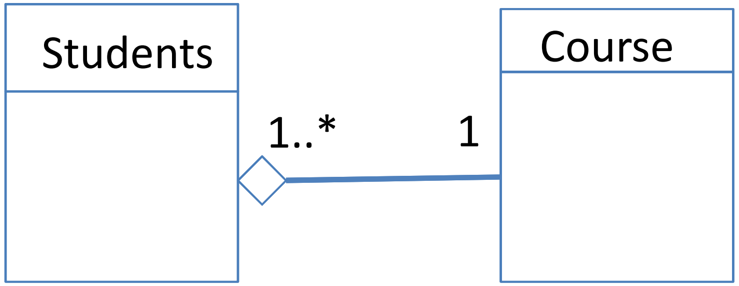
- Associations
  - Aggregation
  - Composition
- Inheritance

Associations

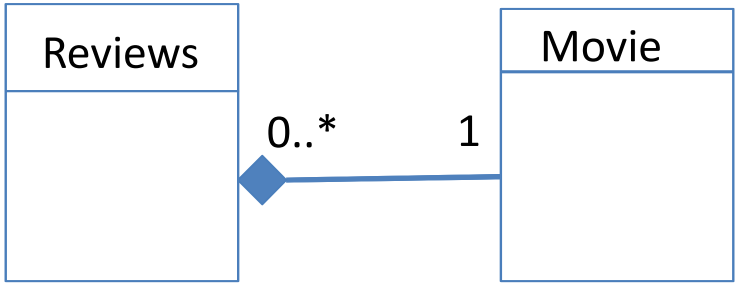
- Capture n-m relationships
  - Like entity-relationship diagrams (from databases)
  - “Connected to” relationship
- Label endpoints of edge with cardinalities
  - Use * for arbitrary

Two Kinds of Owning Associations: Aggregation and Composition

- In an **aggregation**, the owned objects survive destruction of the owning object

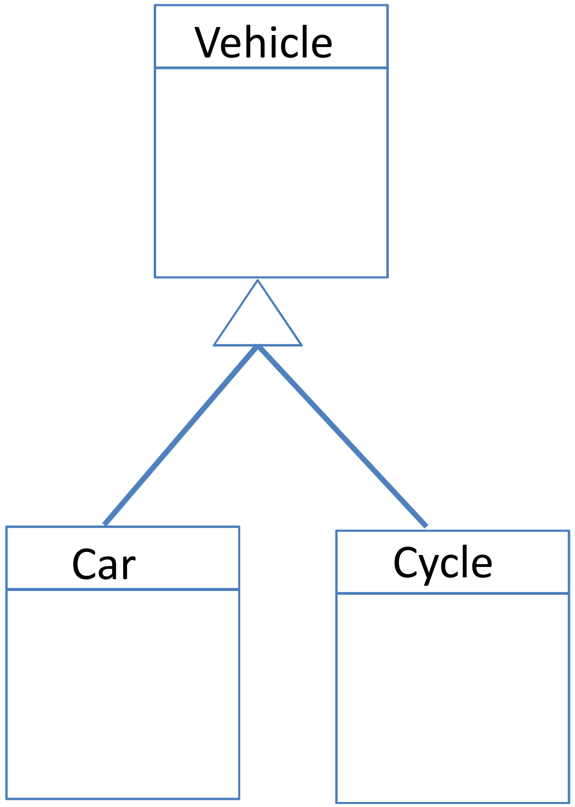


- In a **composition**, the owned objects are usually destroyed when the owning object is destroyed

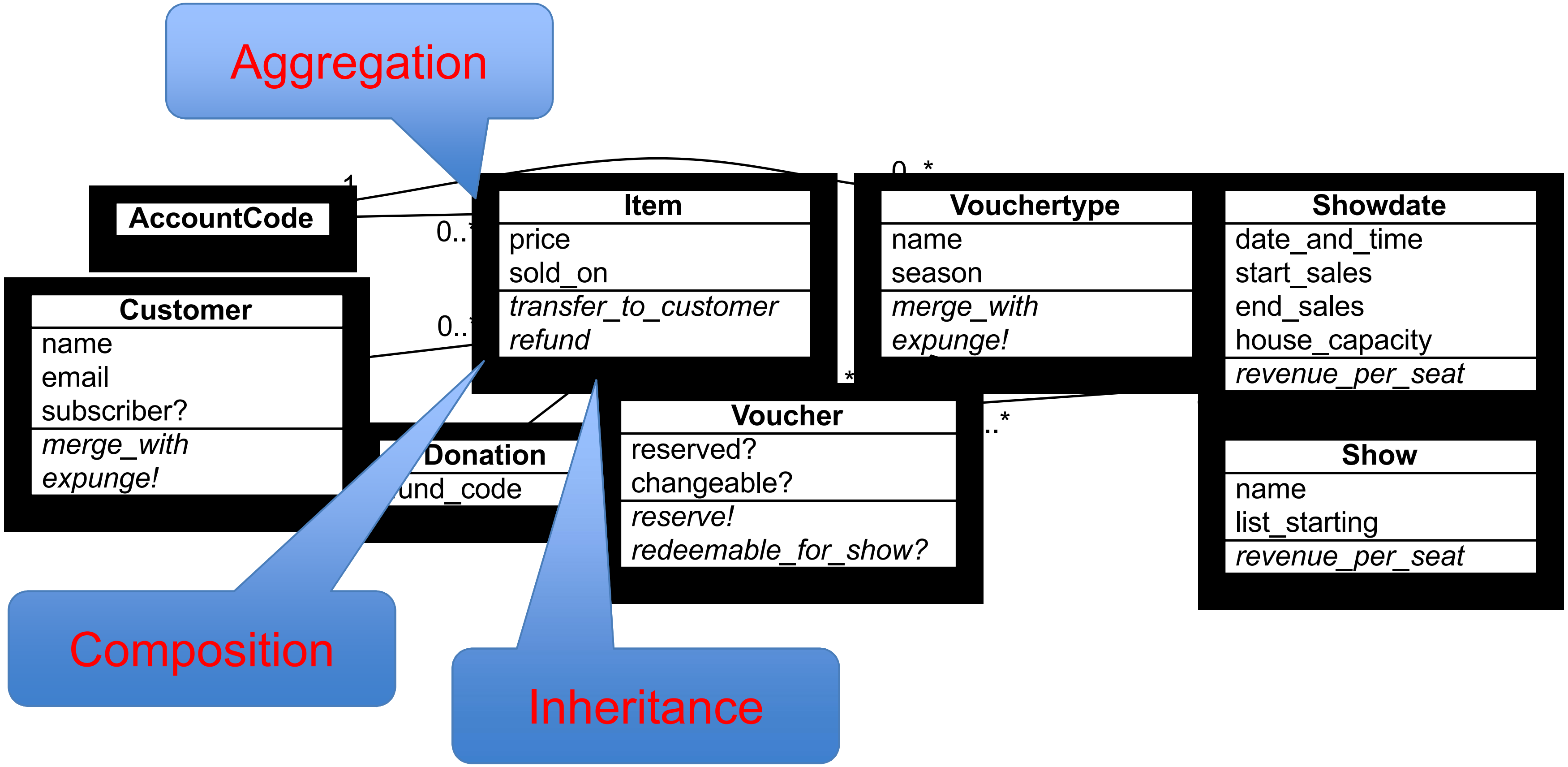


Generalization/Inheritance

- Inheritance between classes
- Denoted by open triangle on superclass
- All arrows point in the direction of code dependency



Relationships



Aggregation

AccountCode

Item
price
sold_on
transfer_to_customer
refund

Vouchertype
name
season
merge_with
expunge!

Showdate
date_and_time
start_sales
end_sales
house_capacity
revenue_per_seat

Customer
name
email
subscriber?
merge_with
expunge!

Donation
und_code

Voucher
reserved?
changeable?
reserve!
redeemable_for_show?

Show
name
list_starting
revenue_per_seat

Composition

Inheritance

Should the relationship “University has many Departments” be modeled as an aggregation or a composition?

- □ Aggregation
- □ Composition
- □ Neither