

Lecture #19 | Unified Modeling Language (UML) Design Patterns

SE271 Object-oriented Programming (2017)

Prof. Min-gyu Cho

Today's topic

- Unified Modeling Language
- Introduction to Design Patterns

Unified Modeling Language (UML)

- Unified Modeling Language
 - Object Management Group (OMG) Standard
 - Based on work from Booch, Rumbaugh, Jacobson
- UML is a modeling language to visualize and design a software system
 - Particularly useful for OO design
 - Independent of implementation language
 - Supported by several IDEs
 - Popular in industry, rather than academia

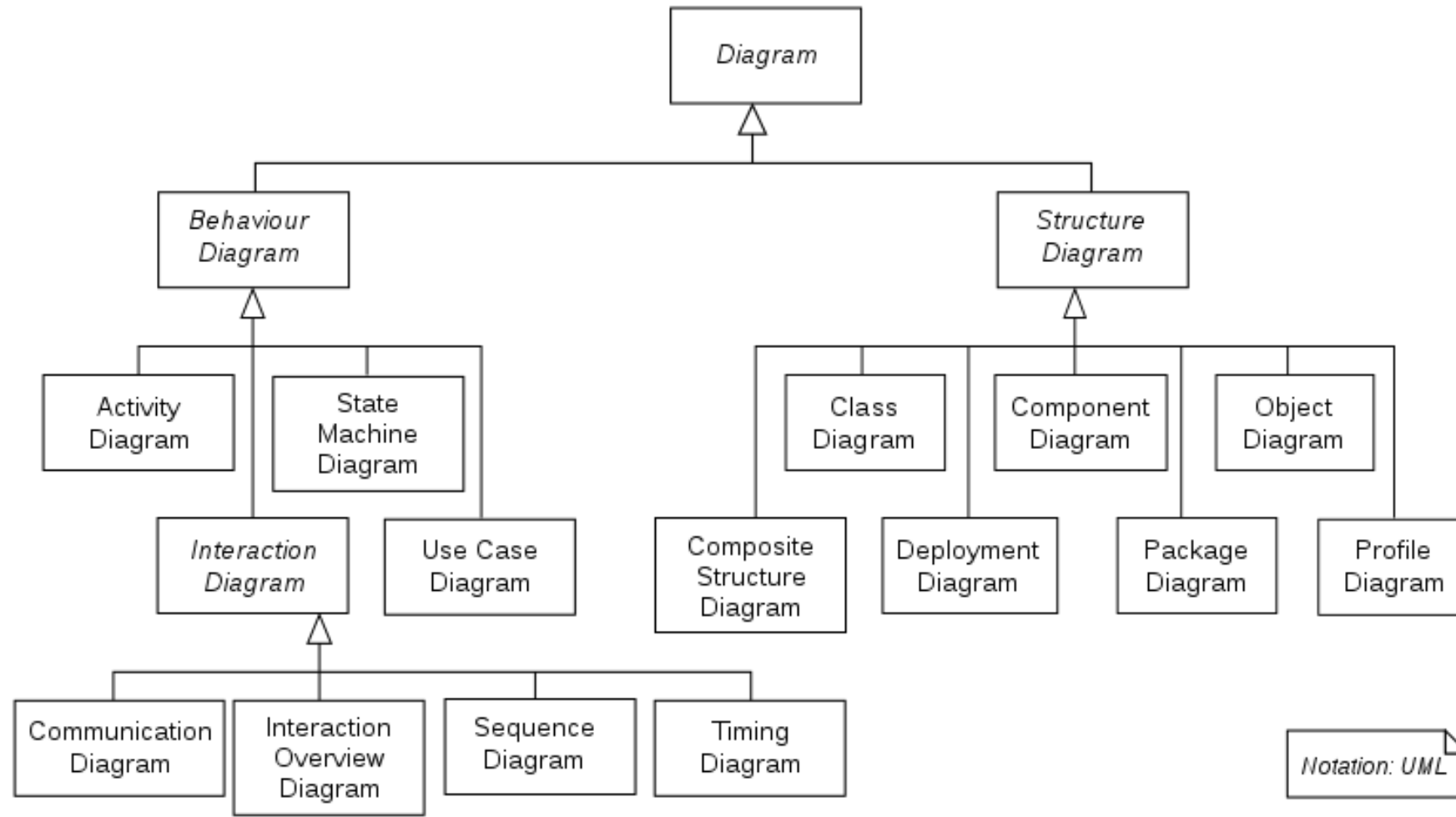
Basic Modeling Steps

- Use Cases
 - Capture requirements
- Domain Model
 - Capture process, key classes
- Design Model
 - Capture details and behaviors of use cases and domain objects
 - Add classes that do the work and define the architecture

UML Baseline

- Use Case Diagrams
- Class Diagrams
- Package Diagrams
- Interaction Diagrams
 - Sequence
 - Collaboration
- Activity Diagrams
- State Transition Diagrams
- Deployment Diagrams

UML Diagrams

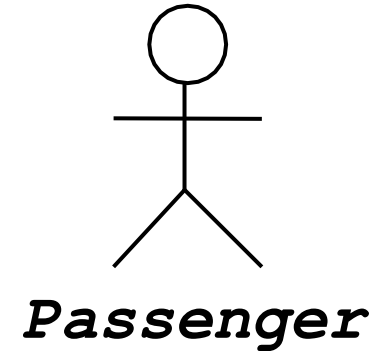


Use case diagram

- Used during requirements elicitation to represent external behavior
- **Actors** represent roles, that is, a type of user of the system
- **Use cases** represent a sequence of interaction for a type of functionality; summary of scenarios
- The use case model is the set of all use cases. It is a complete description of the functionality of the system and its environment

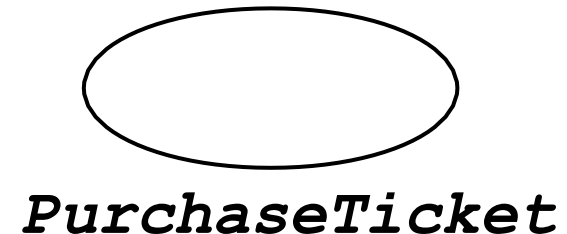
Actors

- An actor models an external entity which communicates with the system:
 - User
 - External system
 - Physical environment
- An actor has a unique name and an optional description
- Examples:
 - Passenger: A person in the train
 - GPS satellite: Provides the system with GPS coordinates



Use case

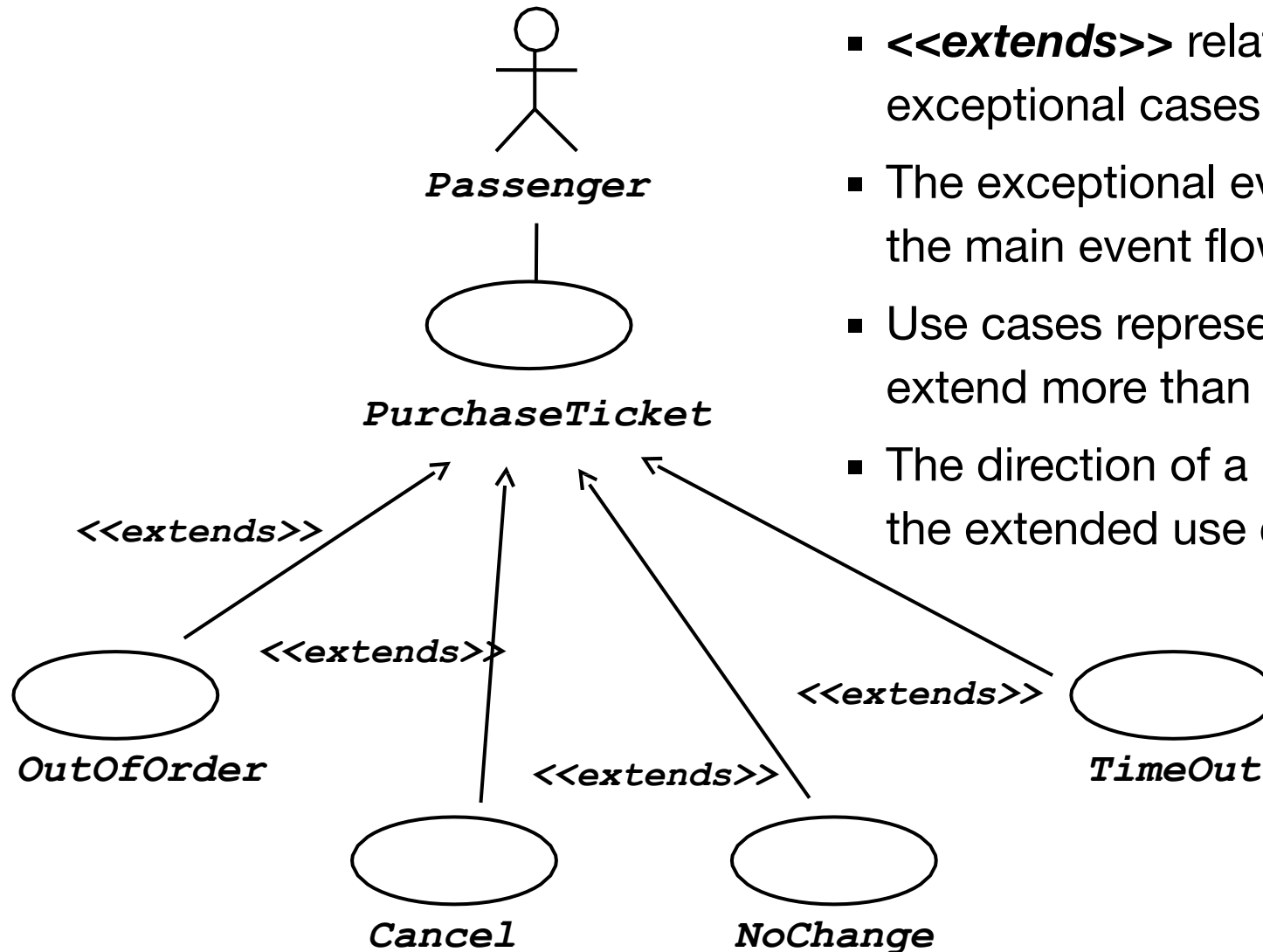
- A use case represents a class of functionality provided by the system as an event flow
- A use case consists of
 - Unique name
 - Participating actors
 - Entry conditions
 - Flow of events
 - Exit conditions
 - Special requirements



Use case diagram examples

- Name: *Purchase ticket*
- Participating actor
 - *Passenger*
- Entry condition
 - *Passenger* standing in front of ticket distributor
 - *Passenger* has sufficient money to purchase ticket
- Exit condition
 - *Passenger* has ticket
- Event flow
 1. Passenger selects the number of zones to be traveled
 2. Distributor displays the amount due
 3. Passenger inserts money, of at least the amount due
 4. Distributor returns change
 5. Distributor issues ticket
- ***Anything missing?***
- ***Exceptional cases!***

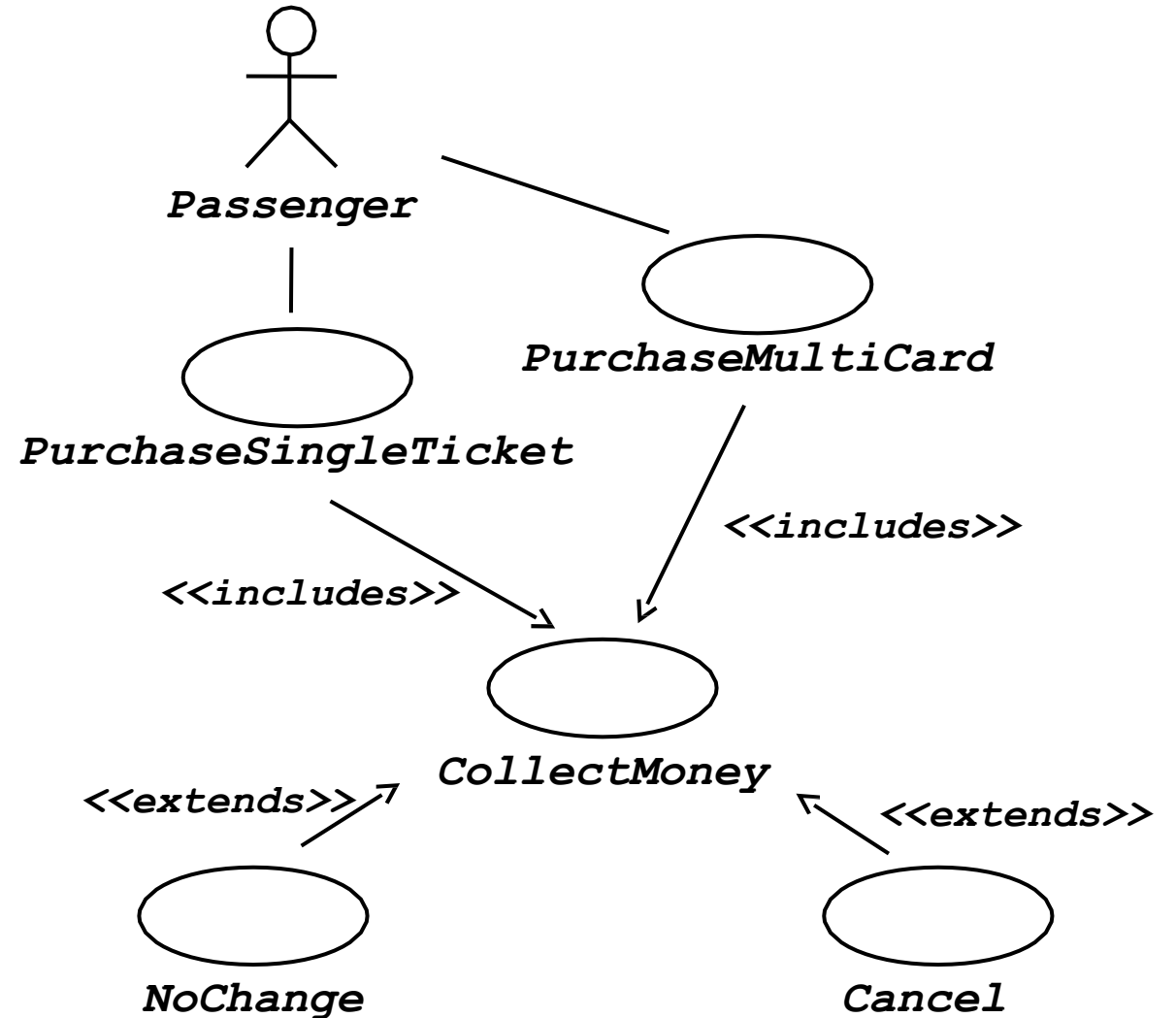
The <<extends>> relationship



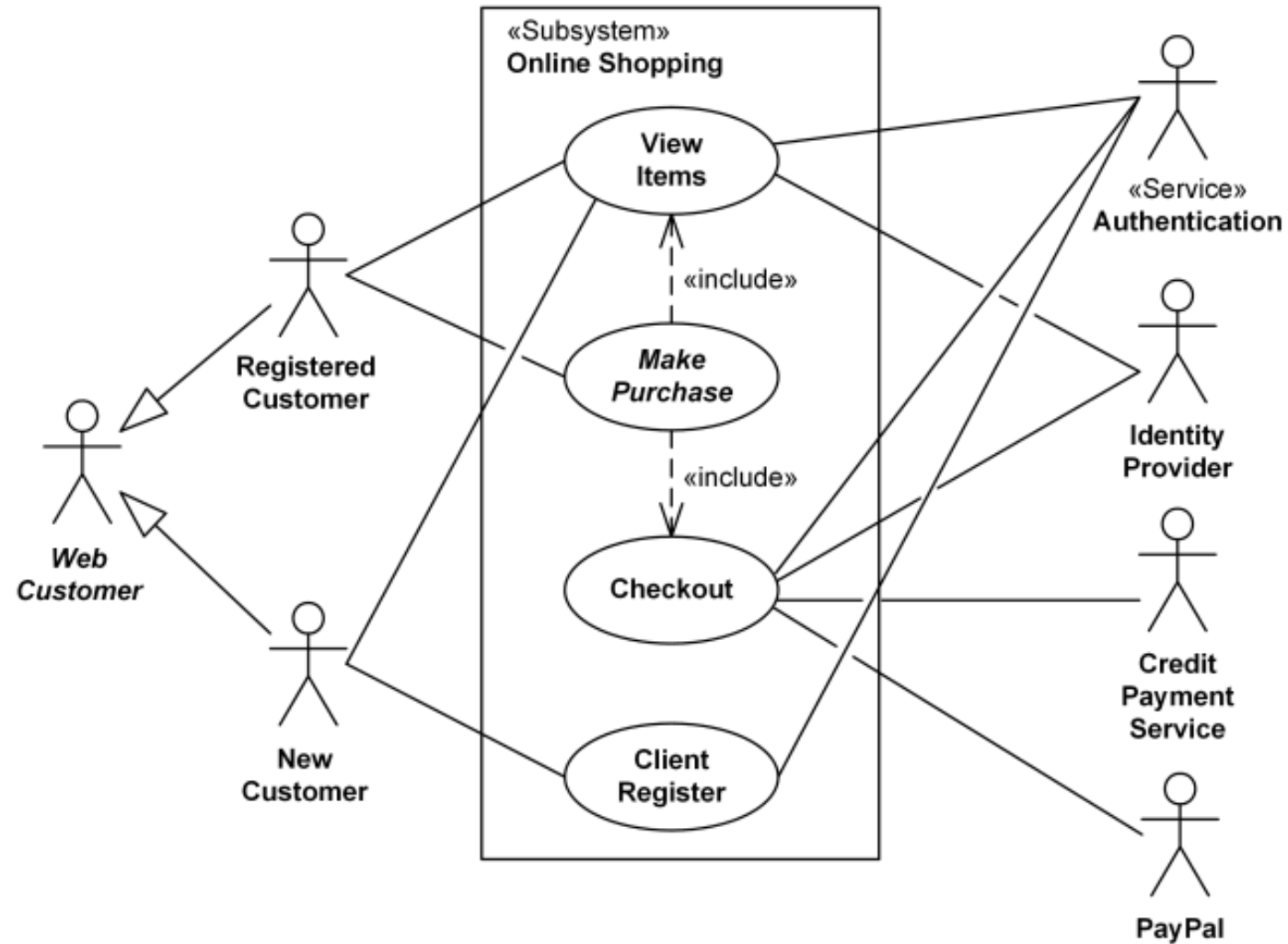
- <<**extends**>> relationships represent optional or exceptional cases
- The exceptional event flows are factored out of the main event flow for clarity
- Use cases representing exceptional flows can extend more than one use case
- The direction of a <<**extends**>> relationship is to the extended use case

The <<includes>> use case

- <<includes>> relationship represents behavior that is factored out of the use case
- <<includes>> behavior is factored out for reuse, not because it is an exception
- The direction of a <<includes>> relationship is to the using use case (unlike <<extends>> relationships)



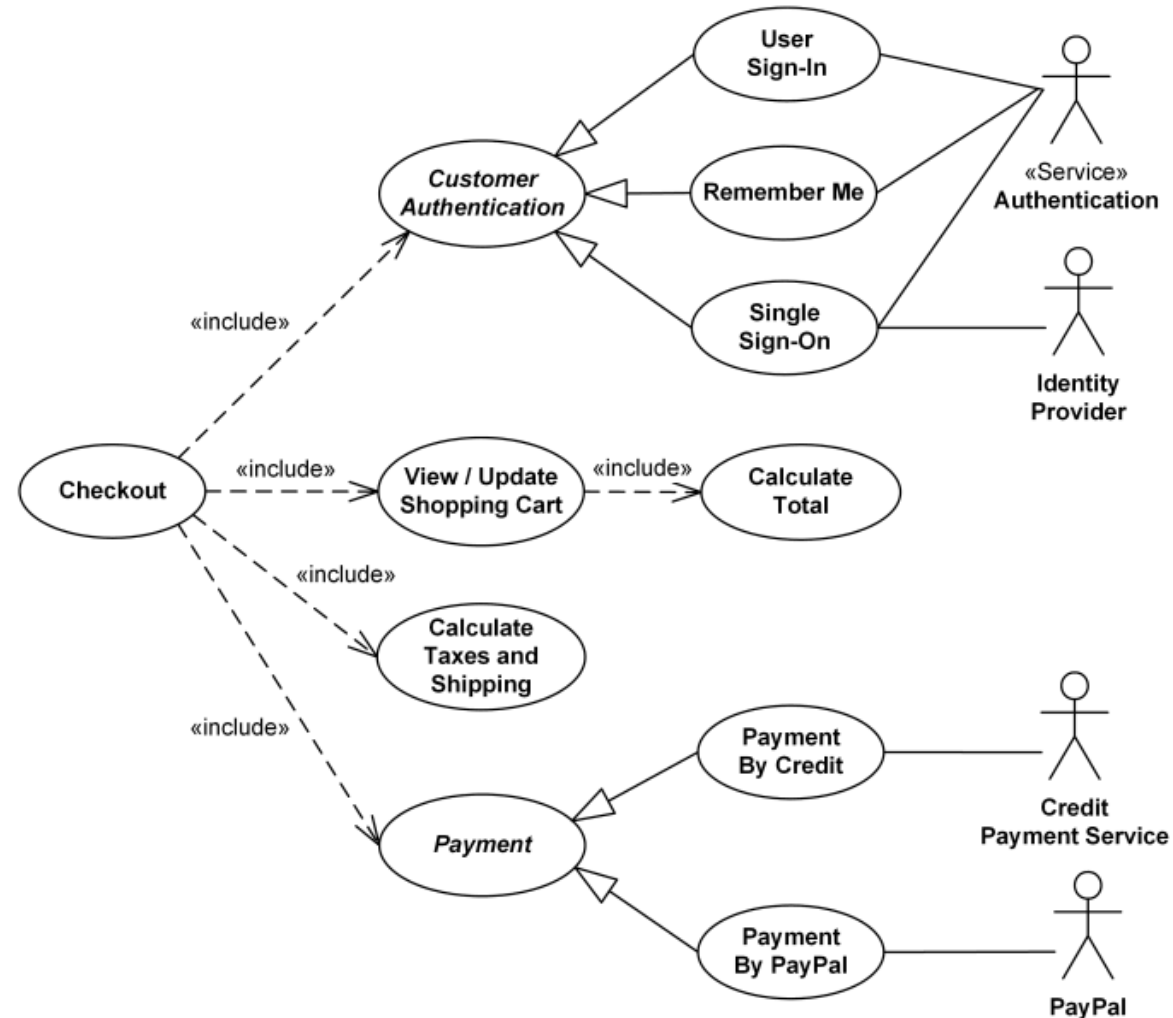
Example: use case diagram



Example: use case diagram (cont.)



Example: use case diagram (cont.)



Benefits of use cases

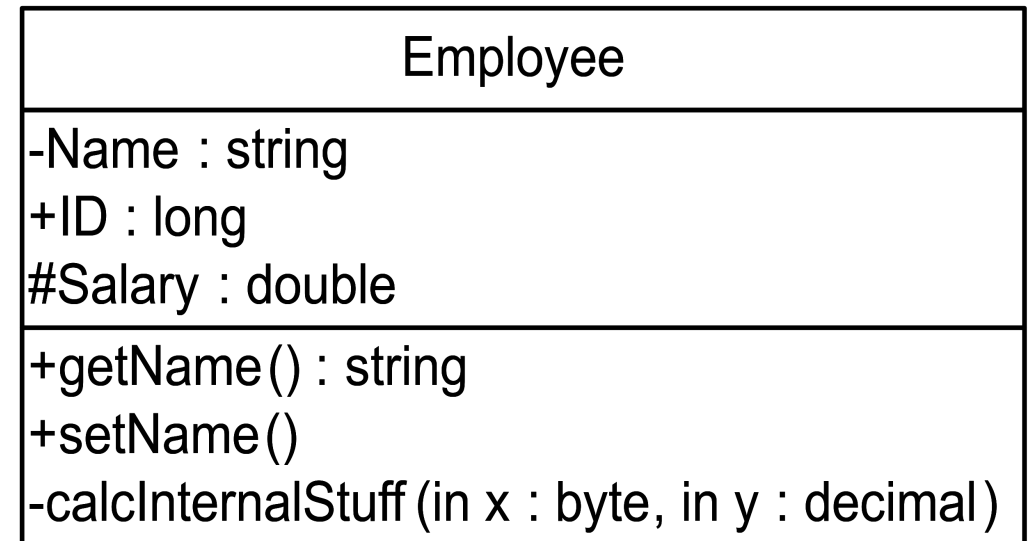
- Determining requirements
 - New use cases often generate new requirements as the system is analyzed and the design takes shape
- Communicating with clients
 - Their notational simplicity makes use case diagrams a good way for developers to communicate with clients
- Generating test cases
 - The collection of scenarios for a use case may suggest a suite of test cases for those scenarios

Class Diagrams



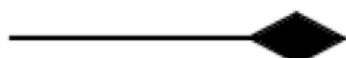

- Give an overview of a system by showing its classes and the relationships among them
 - Show what interacts but not what happens when they interact
 - Show attributes and operations of each class
 - Good way to describe the overall architecture of system components
- Perspectives
 - Conceptual: software or language independent
 - Specific: focusing on the interfaces of the software
 - Implementation: focusing on the implementation of the software

UML Class Notation

- A class is a rectangle divided into three parts
 - Class name
 - Class attributes (i.e., member variables)
 - Class operations (i.e., member functions)
- Modifiers
 - Private: -
 - Public: +
 - Protected: #
 - Static: underlined
- Abstract class: name in italics

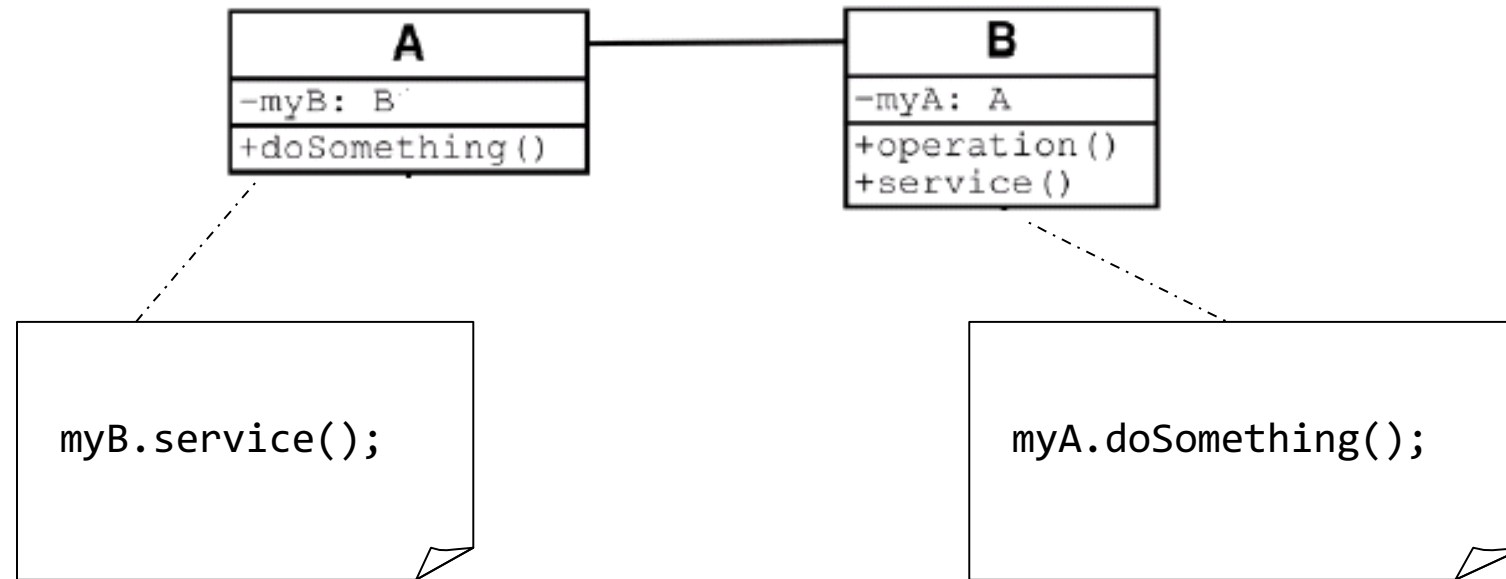


UML Class relations

- **Association:** a straight line or arrow 
 - A relationship between instances of two classes, where one class must know about the other to do its work, e.g., client communicates to server
- **Aggregation:** an empty diamond on the side of the collection 
 - An association where one class belongs to a collection, e.g., instructor part of faculty
- **Composition:** a solid diamond on the side of the collection 
 - Strong form of aggregation
 - Lifetime control; components cannot exist without the aggregate
- **Inheritance:** a triangle pointing to superclass 
 - An inheritance link indicating one class a superclass relationship, e.g., bird is part of mammal

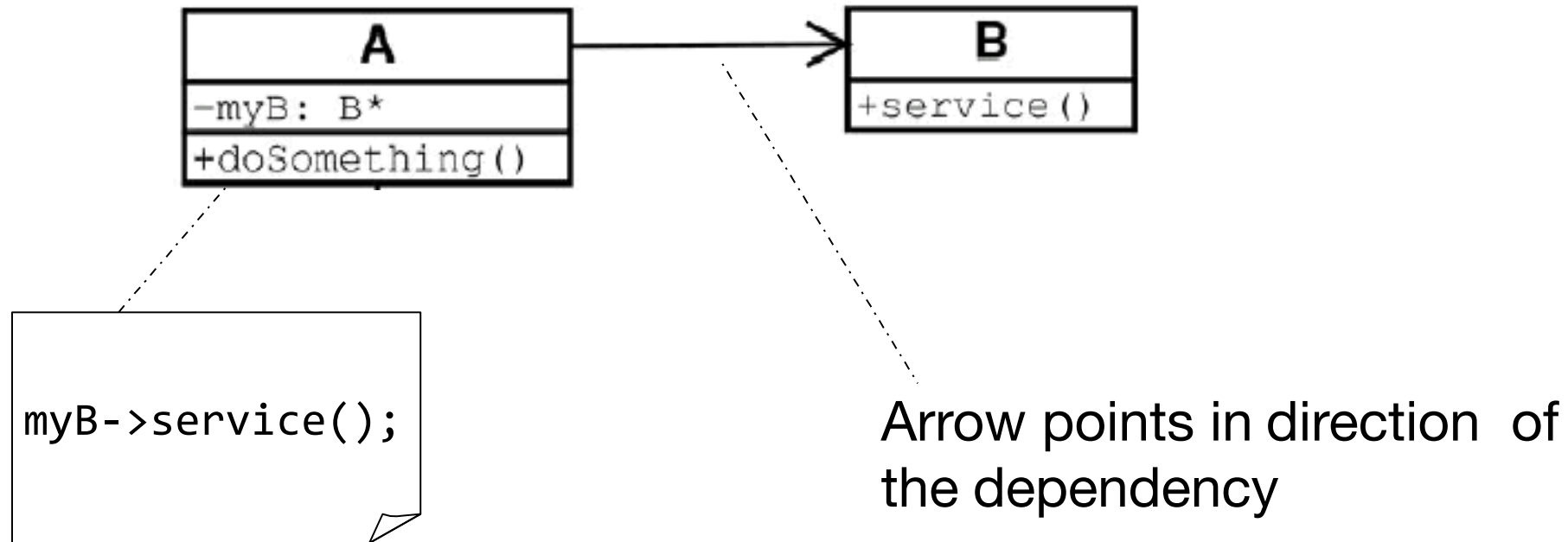
Binary Association

- Both entities “know about” each other



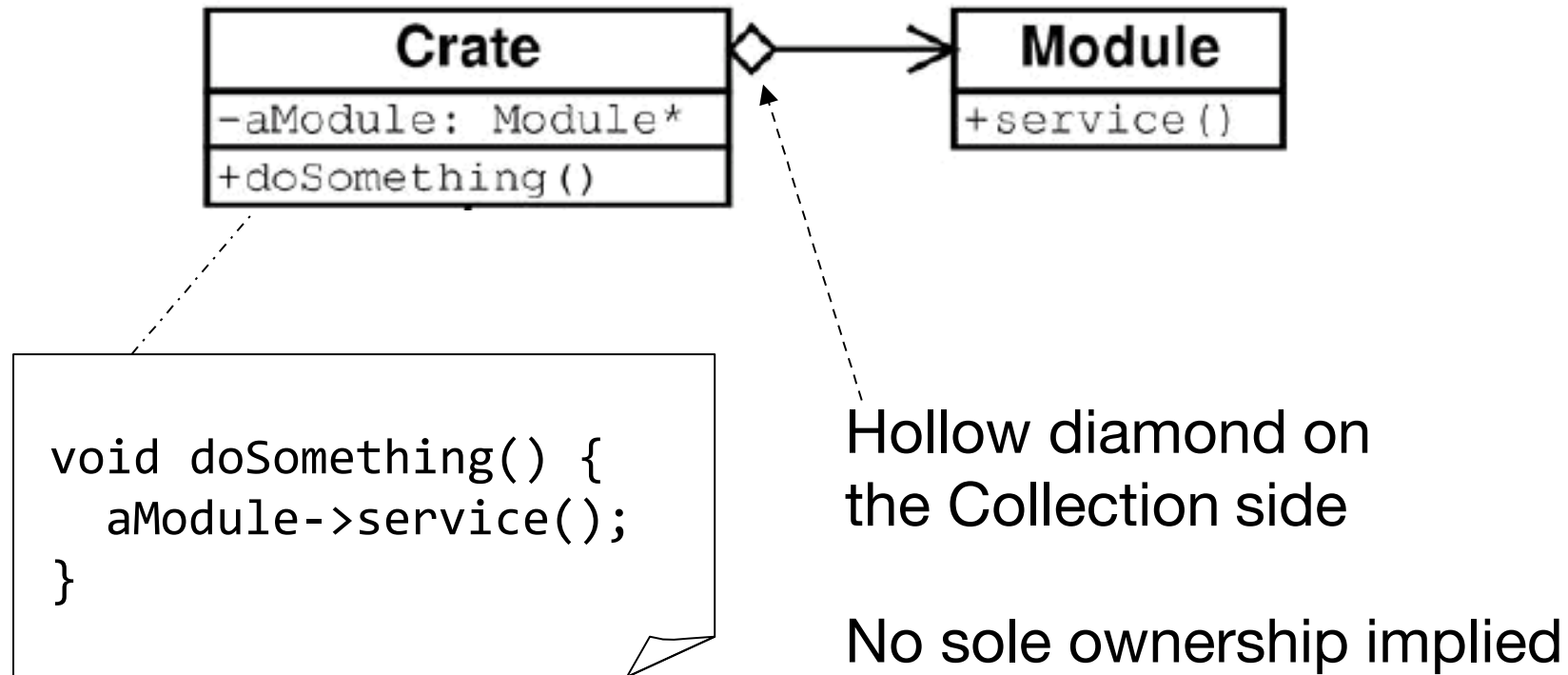
Unary Association

- A knows about B, but B knows nothing about A



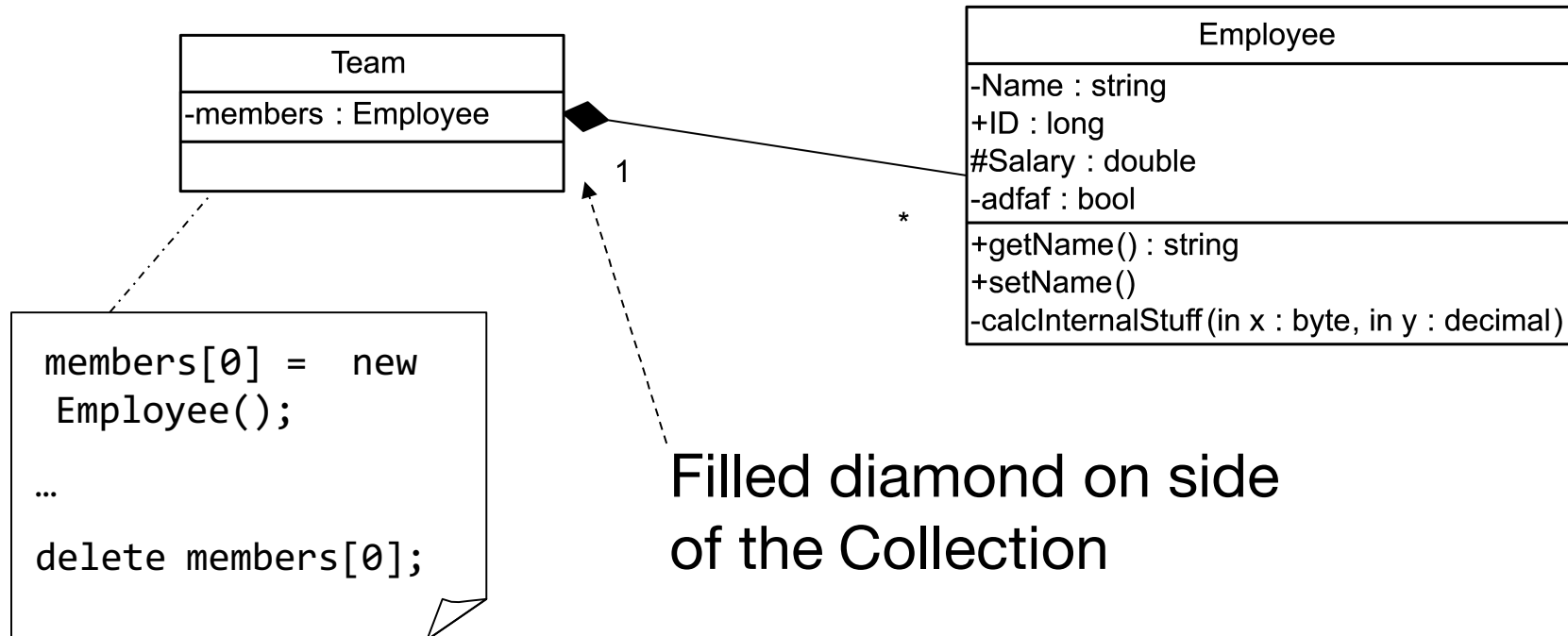
Aggregation

- Aggregation is an association with a “collection-member” relationship



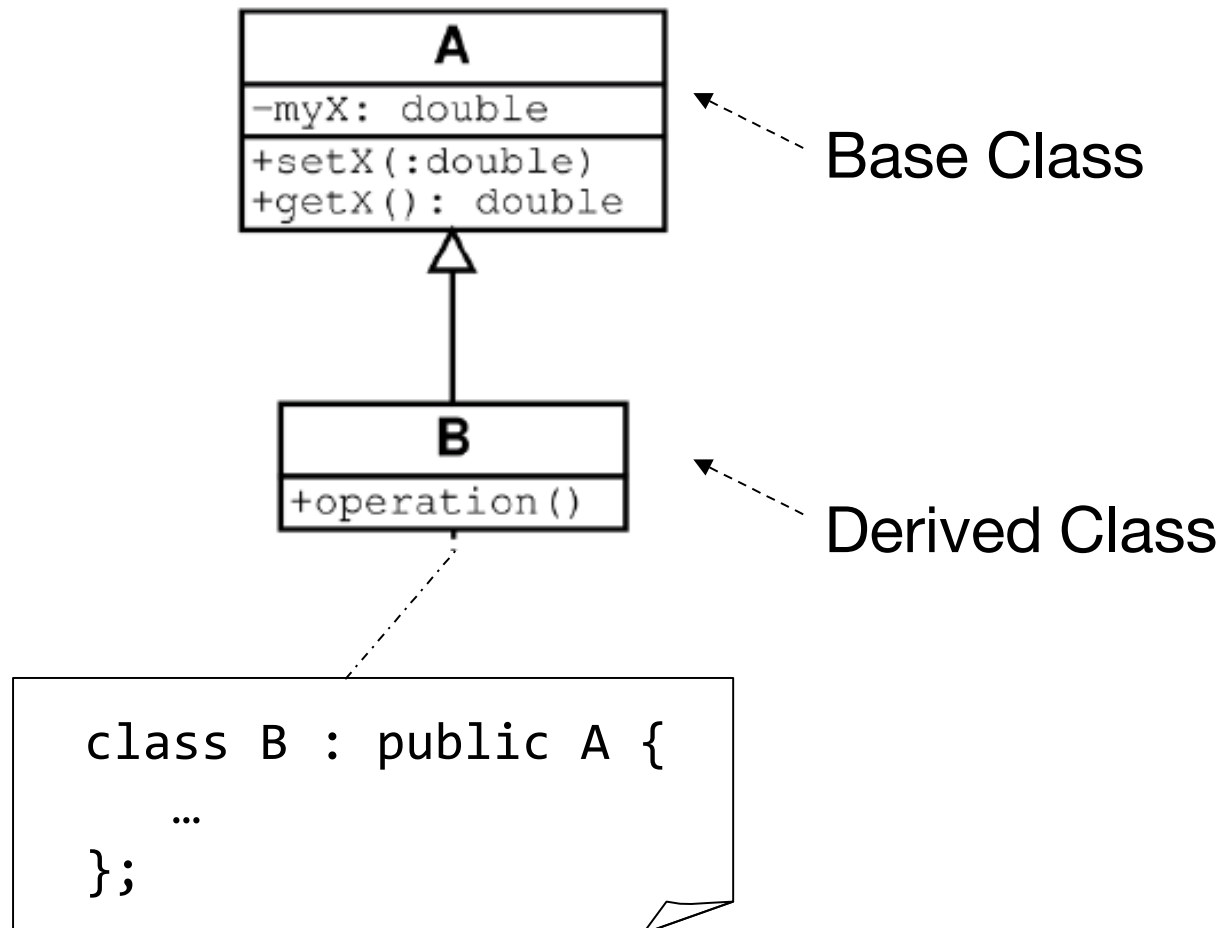
Composition

- Composition is aggregation with
 - The whole-part relationship
 - Lifetime control (owner controls construction & destruction)



Inheritance

- Standard concept of inheritance

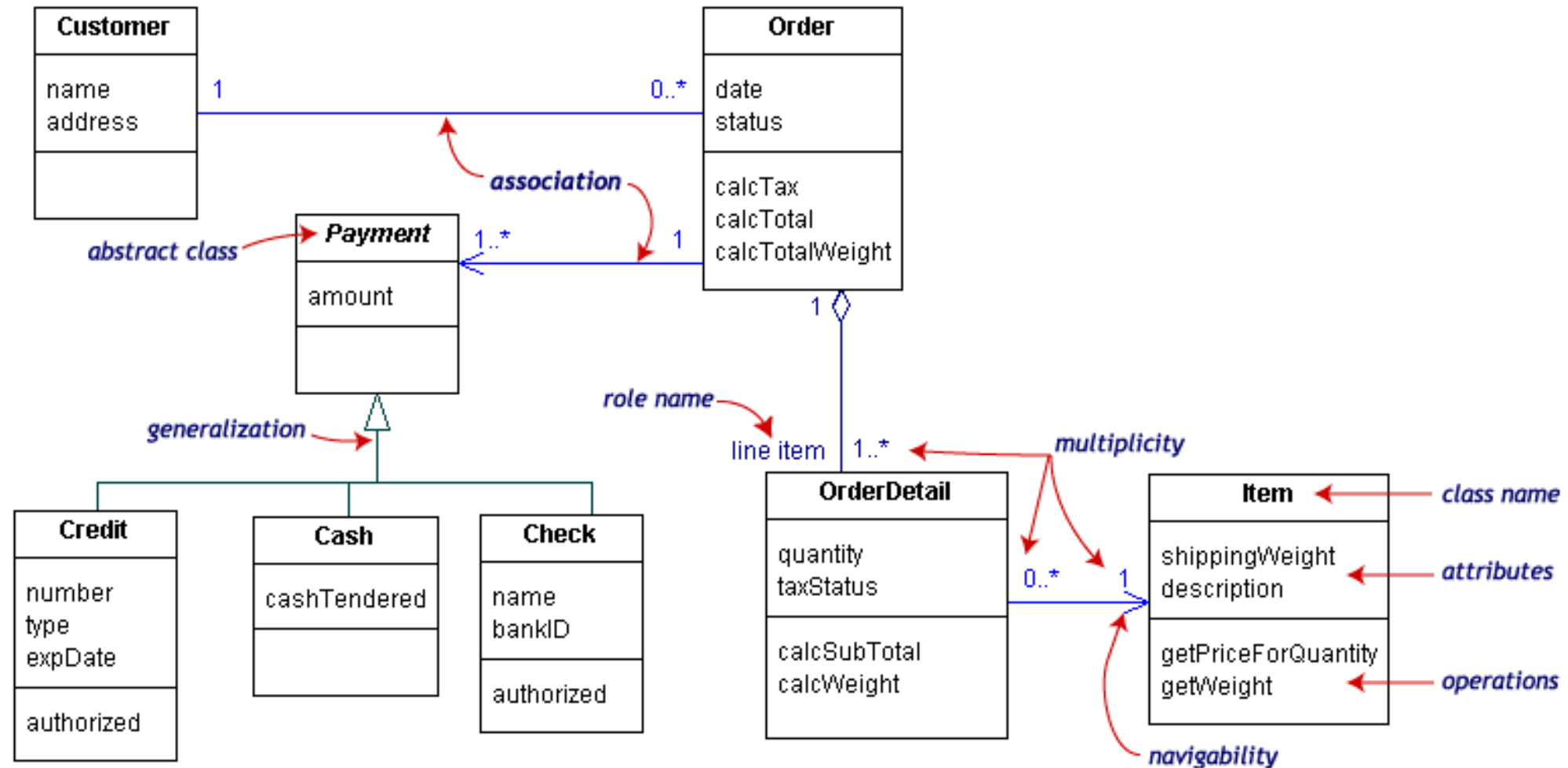


UML Multiplicities

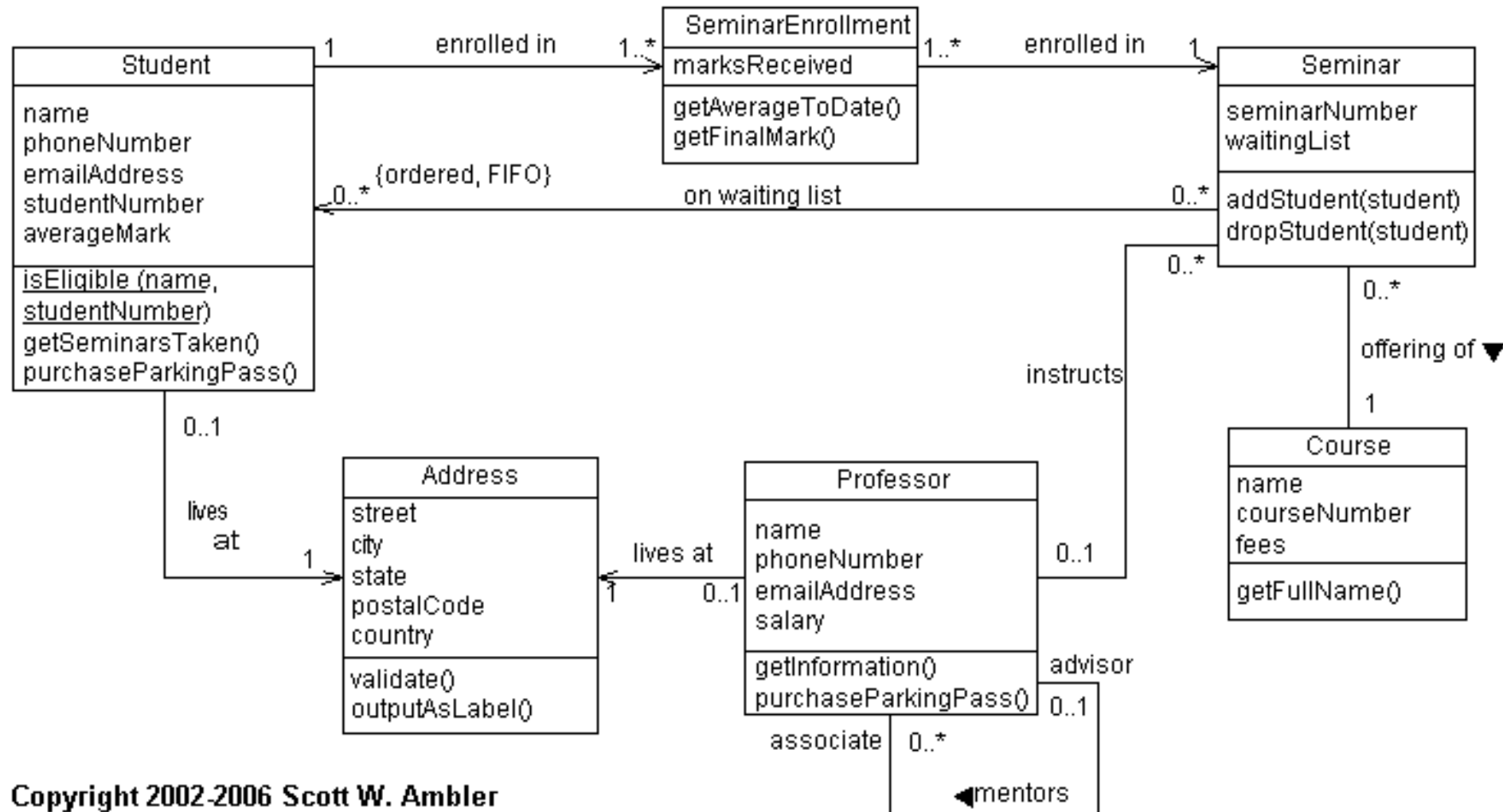
- Links on associations to specify more details about the relationship

Multiplicities		Meaning
n..m		n to m instances
*		no limit on the number of instances (including none)
1		exactly one instance
1..*		at least one instance

Example: UML class diagram



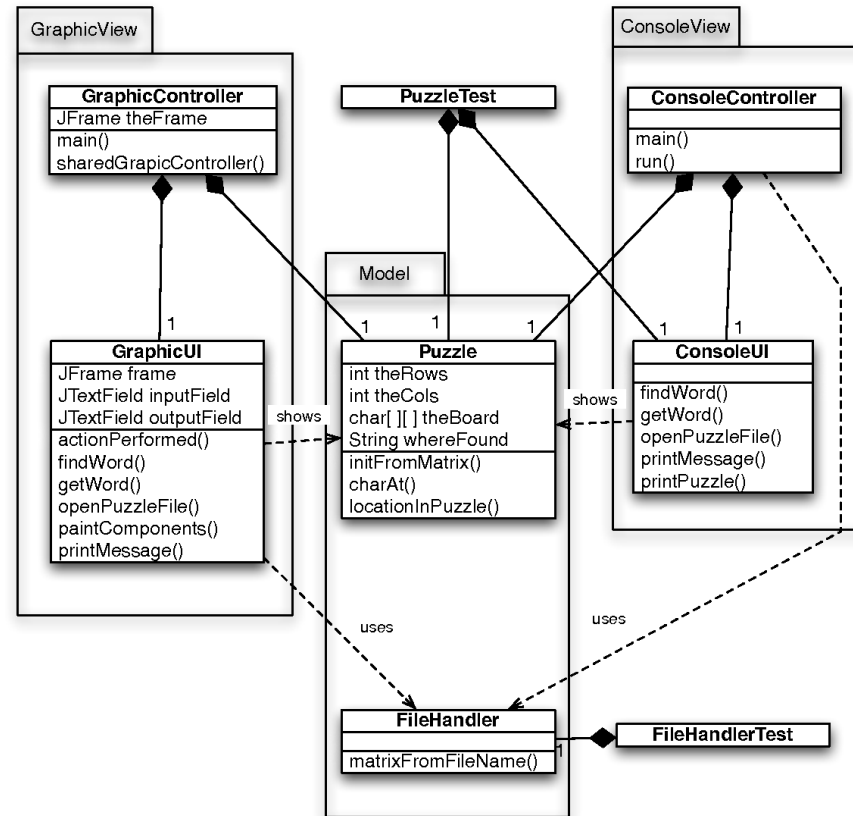
Example: UML class diagram



Copyright 2002-2006 Scott W. Ambler

Example: UML class diagram (word search)

Word Search UML Class Diagram

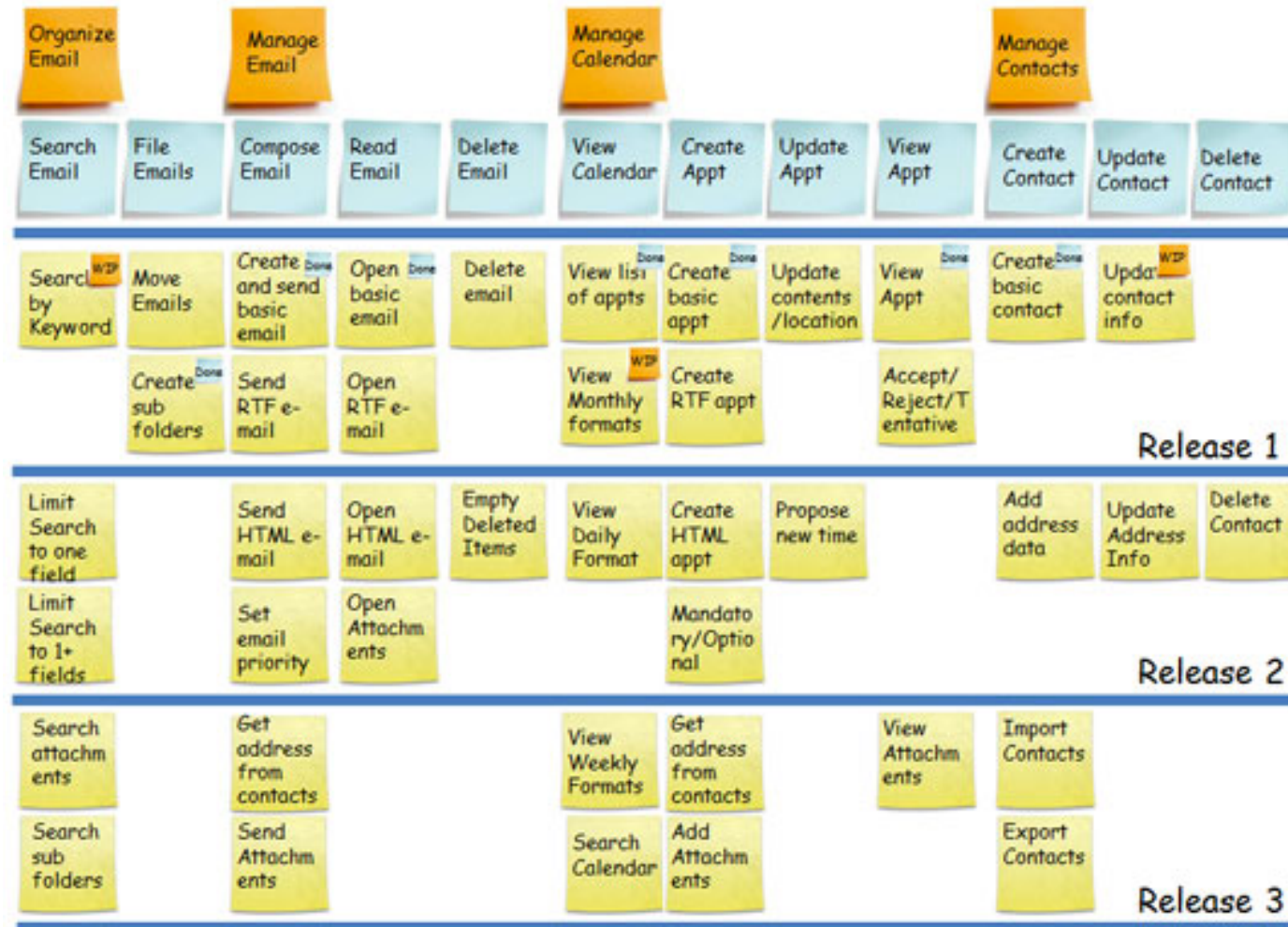


User Story (in agile development process)

- User story
 - As a *role*, I want *goal/desire* so that *benefit*
 - Acceptance criteria: condition or metric to determine goal is met
- Personas are often defined for specific contexts, e.g.,
 - Tina is a high school math teacher in 30s. She is tech-savvy and open to adopt new pedagogy
 - Sam is a high school student. His math skill is just about the average of class. He is more into literature and history, and likes biology most among science courses
- Examples
 - As Tina, I want to adjust difficulty level of exit tickets* for each student so that I can properly motivate my students

* A quiz taken at the end of a class; students are required to pass the quiz to exit the classroom

User story map



Job story (Jobs to be done)

- When *situation*, I want to *motivation* so I can *expected outcome*
- Why job story?
 - Try to add or clarify context and causality on user story
 - But some variations of user story address this issue, thus not so popular yet
- Example
 - **User Story**
 - As a moderator, I want to create a new game by entering a name and an optional description, so that I can start inviting estimators
 - **Job Story**
 - When I'm ready to have estimators bid on my game, I want to create a game in a format estimators can understand, so that the estimators can find my game and know what they are about to bid on

Design Patterns

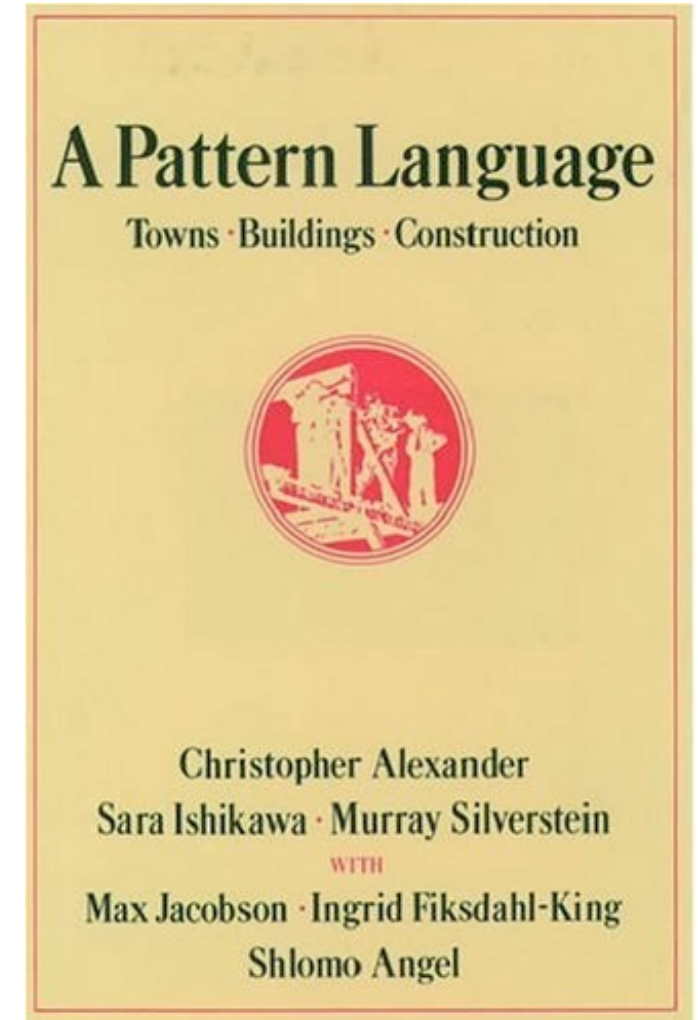
- What is design patterns?
 - A *pattern* is a recurring *solution* to a standard *problem*, in a context
 - General reusable solution to a commonly occurring problem in software design
 - Extension of OOP and object-oriented analysis and design (OOAD)
- What is *NOT* design patterns?
 - Requirement of object oriented programming
 - Requirement of good programming
 - Substitute for application code and logic
 - Best solution for every job
 - Easiest concept to grasp or use

History of design patterns

- Patterns originated as an architectural concept by Christopher Alexander - 1977
- Kent Beck and Ward Cunningham applied patterns to programming and presented their results at OOPSLA conference - 1987
- Gained popularity after the book Design Patterns: Elements of Reusable Object-Oriented Software was published by "Gang of Four" – 1994
- First Pattern Languages of Programming Conference was held – 1994
- Portland Pattern Repository was set up for documentation of design patterns – 1995

Design patterns in architecture

- Idea of *design patterns* originated by Christopher Alexander (Austrian Architect) – 1977
- Initially applied for architecture for building and towns
- “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”



Objected-Oriented Programming Features

- Abstraction
- Encapsulation
- Polymorphism
- Inheritance

Wish list and OOP-principles

- Loose coupling: 1 change = ceteris paribus*
- Code reuse (is not the same as copy/paste)
- Open for extension, closed for modification
- Encapsulate what varies
- Single responsibility principle
- Program against an interface not against an implementation. Dependency injection
- Prefer composition over inheritance

* With other conditions
remaining the same

Essential parts of design patterns

- **Name:** good, recognizable name for common vocabulary
- **A context:** descriptions the sort of situation in which the patterns occurs
- **A problem:** descriptions of the problem that the design pattern is intended to solve
- **A solution:** describes what elements are required to make up the design, their relationships and their context
- **Consequences**
 - What are the results and trade offs by applying the design pattern
 - Allows comparison between different design patterns, to see if there is a better fit for the actual problem

Types of design patterns

- **Creational:** concern the process of object creation
 - Creational patterns are ones that create objects for you, rather than having you instantiate objects directly
 - This gives your program more flexibility in deciding which objects need to be created for a given case
- **Structural:** deal with the composition of classes/objects
 - These concern class and object composition
 - They use inheritance to compose interfaces and define ways to compose objects to obtain new functionality
- **Behavioral:** characterize the ways in which classes/objects interact and distribute responsibilities
 - Most of these design patterns are specifically concerned with communication between objects

List of 23 Design Patterns

- Creational Patterns (5)
 - Singleton
 - Abstract Factory
 - Builder
 - Factory Method
 - Prototype
- Structural Patterns (7)
 - Adapter
 - Bridge
 - Composite
 - Decorator
 - Façade
 - Flyweight
 - Proxy
- Behavioral Patterns (11)
 - Chain of Responsibility
 - Command
 - Interpreter
 - Iterator
 - Mediator
 - Memento
 - Observer
 - State
 - Strategy
 - Template Method
 - Visitor

Creational patterns

- **Abstract Factory:** Groups object factories that have a common theme
- **Builder constructs:** Complex objects by separating construction and representation
- **Factory Method:** Creates objects without specifying the exact class to create
- **Prototype:** Creates objects by cloning an existing object
- **Singleton:** Restricts object creation for a class to only one instance.

Structural patterns

- **Adapter:** Allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class
- **Bridge:** Decouples an abstraction from its implementation so that the two can vary independently
- **Composite:** Composes zero-or-more similar objects so that they can be manipulated as one object
- **Decorator:** Dynamically adds/overrides behavior in an existing method of an object
- **Facade:** Provides a simplified interface to a large body of code
- **Flyweight:** Reduces the cost of creating and manipulating a large number of similar objects
- **Proxy:** Provides a placeholder for another object to control access, reduce cost, and reduce complexity.

Behavioral patterns

- **Chain of responsibility:** Delegates commands to a chain of processing objects
- **Command:** Creates objects which encapsulate actions and parameters
- **Interpreter:** Implements a specialized language
- **Iterator:** Accesses the elements of an object sequentially without exposing its underlying representation
- **Mediator:** Allows loose coupling between classes by being the only class that has detailed knowledge of their methods
- **Memento:** Provides the ability to restore an object to its previous state (undo)

Behavioral patterns (cont.)

- **Observer:** Is a publish/subscribe pattern which allows a number of observer objects to see an event
- **State:** Allows an object to alter its behavior when its internal state changes
- **Strategy:** Allows one of a family of algorithms to be selected on-the-fly at runtime
- **Template method:** Defer the exact steps of an algorithm to a subclass
- **Visitor:** Defines a new operation to a class without change

Smell of good/bad code

- What is good code?
- What is good design?
 - Maintainable/Readable/Easily understandable
 - Reusable
 - Easier to change if requirement changes
 - Performs better
 - Modular/Componentized
 - ...

Smell within Class

- Comments
 - Should only be used to clarify "why" not "what". Can quickly become verbose and reduce code clarity
- Long Method
 - The longer the method the harder it is to see what it's doing
- Long Parameter List
 - Don't pass everything the method needs. Pass enough so that the method gets everything it needs
- Duplicated Code
- Large Class
 - A class that is trying to do too much can usually be identified by looking at how many instance variables it has. When a class has too many instance variables, duplicated code cannot be far behind

Smell within Class (cont.)

- Type Embedded in Name
 - Avoid redundancy in naming. Prefer `Schedule.Add(course)` to `Schedule.AddCourse(course)`
- Uncommunicative Name
 - Choose names that communicate intent (pick the best name for the time, change it later if necessary)
- Inconsistent Names
 - Use names consistently
- Dead Code
 - A variable, parameter, method, code fragment, class is not used anywhere (perhaps other than in tests)
- Speculative Generality
 - Don't over-generalize your code in an attempt to predict future needs

Smell between Classes

- Primitive Obsession
 - Use small objects to represent data such as money (which combines quantity and currency) or a date range object
- Data Class
 - Classes with fields and getters and setters and nothing else (a.k.a., Data Transfer Objects, DTO)
- Data Clumps
 - Clumps of data items that are always found together
- Refused Bequest
 - Subclasses don't want or need everything they inherit. Liskov Substitution Principle (LSP) says that you should be able to treat any subclass of a class as an example of that class
- Inappropriate Intimacy
 - Two classes are overly entwined

Smell between Classes (cont.)

- Lazy Class
 - Classes that aren't doing enough should be refactored away
- Feature Envy
 - Often a method that seems more interested in a class other than the one it's actually in. In general, try to put a method in the class that contains most of the data the method needs
- Message Chains
 - This is the case in which a client has to use one object to get another, and then use that one to get to another, etc. Any change to the intermediate relationships causes the client to have to change
- Middle Man
 - When a class is delegating almost everything to another class, it may be time to refactor out the middle man

Smell between Classes (cont.)

- Divergent Change
 - Occurs when one class is commonly changed in different ways for different reasons. Any change to handle a variation should change a single class
- Shotgun Surgery
 - The opposite of Divergent Change. A change results in the need to make a lot of little changes in several classes
- Parallel Inheritance Hierarchies
 - Special case of Shotgun Surgery. Every time you make a subclass of a class, you also have to make a subclass of another

“First Law of Software Quality”

$$\left\{ \begin{array}{l} \text{errors} = (\text{more code})^2 \\ e = mc^2 \end{array} \right\}$$



milindaudichya

C++ Core Guidelines and Coding Style Guide

- Core Guidelines: <https://github.com/isocpp/CppCoreGuidelines>
 - Korean Translation: <https://github.com/CppKorea/CppCoreGuidelines>
 - Presentation slides at CppCon15:
<https://github.com/isocpp/CppCoreGuidelines/blob/master/talks/Stroustrup%20-%20CppCon%202015%20keynote.pdf>
- C++ Coding Style Guide
 - Coding Standards
 - <https://isocpp.org/wiki/faq/coding-standards>
 - C++ Coding Standards: 101 Rules, Guidelines, and Best Practices
 - <https://www.amazon.com/exec/obidos/ASIN/0321113586/>
 - Google C++ Style Guide
 - <https://google.github.io/styleguide/cppguide.html>
 - GCC C++ Coding Conventions
 - <https://gcc.gnu.org/wiki/CppConventions>
 - LLVM Coding Standards: <http://llvm.org/docs/CodingStandards.html>

Reading list on UML

- Examples of UML diagrams
 - <http://www.uml-diagrams.org/index-examples.html>
 - <http://creately.com/blog/diagrams/uml-diagram-types-examples/>
- User story
 - <http://www.romanpichler.com/blog/personas-epics-user-stories/>
 - <http://www.romanpichler.com/blog/10-tips-writing-good-user-stories/>
- Job story
 - <https://jtbd.info/replacing-the-user-story-with-the-job-story-af7cdee10c27#.e9766lidv>

Reference on Design Patterns

- Design Patterns: Elements of Reusable Object-Oriented Software by Gamma, Helm, Johnson and Vlissides (Gang of Four, GoF)
- Head First Design Patterns: A Brain-Friendly Guide by Freeman, Robson, Bates and Sierra
- Design Patterns: explained simply
 - https://sourcemaking.com/design_patterns
 - <https://sourcemaking.com/antipatterns>



ANY QUESTIONS?