

Agile Process Models

- Agile software engineering combines a philosophy and a set of development guidelines
- **Philosophy**
 - Encourages customer satisfaction and early incremental delivery of the software
 - Small highly motivated project teams
 - Informal methods
 - Minimal software engineering work products
 - Overall development simplicity
- **Development guidelines**
 - Stress delivery over analysis and design
 - Active and continuous communication between developers and customers

Agile Process Models (contd.)

IMPOTANT VALUES TO THE AGILE SOFTWARE DEVELOPMENT MODEL



Individuals and interactions rather than processes and tools.



Development of working software



Collaboration with the customer or client



Quickly Responding to change

Agile Process Models (contd.)

Scrum

Crystal Methodologies

DSDM (Dynamic Software Development Method)

Feature driven development (FDD)

Lean software development

Extreme Programming (XP)

Agile vs. Waterfall Method

Agile Model	Waterfall Model
Agile method proposes incremental and iterative approach to software design	Development of the software flows sequentially from start point to end point.
The agile process is broken into individual models that designers work on	The design process is not broken into an individual models
The customer has early and frequent opportunities to look at the product and make decision and changes to the project	The customer can only see the product at the end of the project
Agile model is considered unstructured compared to the waterfall model	Waterfall model are more secure because they are so plan oriented

Agile vs. Waterfall Method (contd.)

Agile Model	Waterfall Model
Small projects can be implemented very quickly. For large projects, it is difficult to estimate the development time.	All sorts of project can be estimated and completed.
Error can be fixed in the middle of the project.	Only at the end, the whole product is tested. If the requirement error is found or any changes have to be made, the project has to start from the beginning
Development process is iterative, and the project is executed in short (2-4) weeks iterations. Planning is very less.	The development process is phased, and the phase is much bigger than iteration. Every phase ends with the detailed description of the next phase.
Documentation attends less priority than software development	Documentation is a top priority and can even use for training staff and upgrade the software with another team

Agile vs. Waterfall Method (contd.)

Agile Model	Waterfall Model
In agile testing when an iteration end, shippable features of the product is delivered to the customer. New features are usable right after shipment. It is useful when you have good contact with customers.	All features developed are delivered at once after the long implementation phase.
Testers and developers work together	Testers work separately from developers
At the end of every sprint, user acceptance is performed	User acceptance is performed at the end of the project.
It requires close communication with developers and together analyze requirements and planning	Developer does not involve in requirement and planning process. Usually, time delays between tests and coding

Advantages of Agile Model

- Customer satisfaction by rapid, continuous delivery of useful software.
- People and interactions are emphasized rather than process and tools. Customers, developers and testers constantly interact with each other.
- Working software is delivered frequently (weeks rather than months).
- Face-to-face conversation is the best form of communication.
- Close, daily cooperation between business people and developers.
- Regular adaptation to changing circumstances.
- Even late changes in requirements are welcomed

Disadvantages of Agile model

- In case of some software deliverables, especially the large ones, it is difficult to assess the effort required at the beginning of the software development life cycle.
- There is lack of emphasis on necessary designing and documentation.
- The project can easily get taken off track if the customer representative is not clear what final outcome that they want.
- Only senior programmers are capable of taking the kind of decisions required during the development process. Hence it has no place for newbie programmers, unless combined with experienced resources.

When to use Agile model

- When **new changes need to be implemented**. The freedom agile gives to change is very important. New changes can be implemented at very little cost because of the frequency of new increments that are produced.
- To implement a new feature the developers need to lose only the work of a few days, or even only hours, to roll back and implement it.
- Both system developers and stakeholders alike, find they also get more freedom of time and options than if the software was developed in a more rigid sequential way.
- Having options gives them the ability to leave important decisions until more or better data or even entire hosting programs are available; meaning the project can continue to move forward without fear of reaching a sudden standstill.

Introduction to UML

Introduction to UML

- What is UML?
- Motivations for UML
- Types of UML diagrams
- UML syntax
- Descriptions of the various diagram types
- UML pitfalls

UML Diagrams

- **Unified Modeling Language (UML)** is a standard notation for object-oriented design (A standardized, graphical “modeling language” for communicating software design)
 - Used to **model** object-oriented designs
 - Shows overall design of a solution
 - Shows class specifications
 - Shows how classes interact with each other
 - Diagrams use specific icons and notations
 - It is **language independent**
- Allows implementation-independent specification of:
 - user/system interactions (required behaviors)
 - partitioning of responsibility (OO)
 - integration with larger or existing systems
 - data flow and dependency
 - operation orderings (algorithms)
 - concurrent operations
- UML is not “process”. (That is, it doesn’t tell you how to do things, only what you should do.)

Motivations for UML

- UML is a fusion of ideas from several precursor modeling languages.
- We need a modeling language to:
 - help develop efficient, effective and correct designs, particularly Object Oriented designs.
 - communicate clearly with project stakeholders (concerned parties: developers, customer, etc).
 - give us the “big picture” view of the project.

Types of UML diagrams

- There are different types of UML diagram, each with slightly different syntax rules:
 - use cases.
 - class diagrams.
 - sequence diagrams.
 - package diagrams.
 - state diagrams
 - activity diagrams
 - deployment diagrams.

UML: First Pass

- ◆ You can model 80% of most problems by using about 20 % UML
- ◆ We teach you those 20%

- ◆ Use case Diagrams

What system?

- ◆ Describe the functional behavior of the system as seen by the user.

- ◆ Class diagrams

Are class diagrams of this sort about requirements or design?

- ◆ Describe the static structure of the system: Objects, Attributes, Associations

- ◆ Sequence diagrams

Are sequence diagrams of this sort about requirements or design?

- ◆ Describe the dynamic behavior between actors and the system and between objects of the system

- ◆ Statechart diagrams

Are StateCharts of this sort about requirements or design?

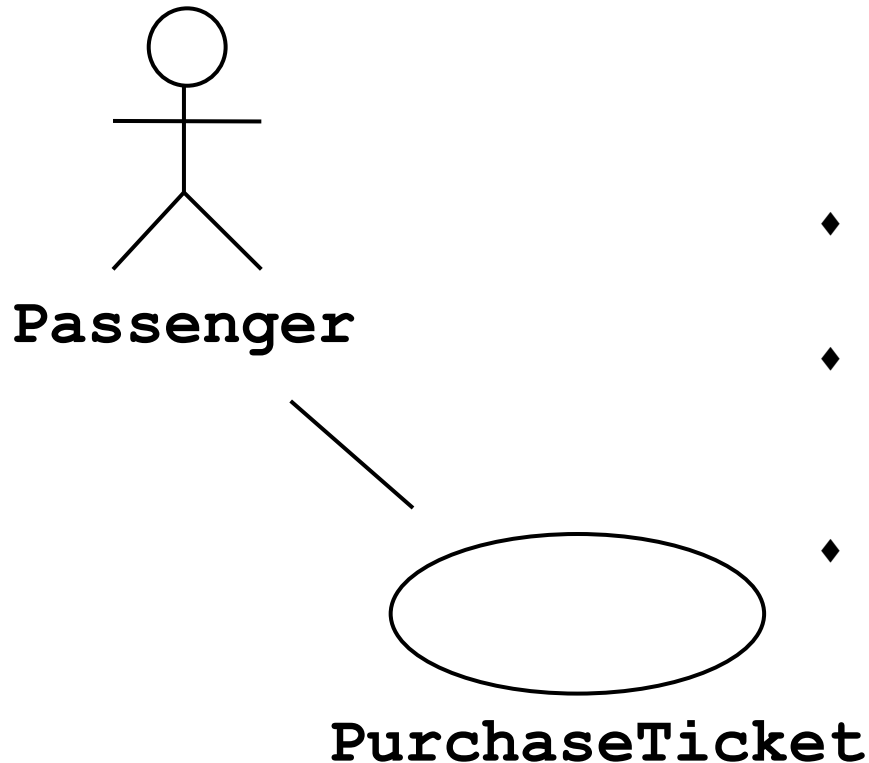
- ◆ Describe the dynamic behavior of an individual object (essentially a finite state automaton)

- ◆ Activity Diagrams

Are activity diagrams of this sort about requirements or design?

- ◆ Model the dynamic behavior of a system, in particular the workflow (essentially a flowchart)

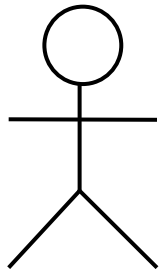
Use Case Diagrams



- ♦ 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
- ♦ 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

What would the functionality of the environment mean?

Actors



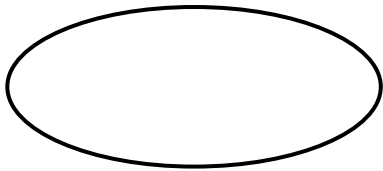
Passenger

- ◆ 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**

course project?

Use Case

A use case represents a class of functionality provided by the system as an event flow.



PurchaseTicket

A use case consists of:

- ◆ Unique name
- ◆ Participating actors
- ◆ Entry conditions
- ◆ Flow of events
- ◆ Exit conditions
- ◆ Special requirements

Use Case Diagram: Example

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.

- ◆ Use case diagrams represent external behavior
- ◆ Use case descriptions provide meat of model, not the use case diagrams.
- ◆ All use cases need to be described for the model to be useful.

UML syntax, 1

- Actors: a UML actor indicates an interface (point of interaction) with the system.
 - We use actors to group and name sets of system interactions.
 - Actors may be people, or other systems.
 - An actor is NOT part of the system you are modeling. An actor is something external that your system has to deal with.
- Boxes: boxes are used variously throughout UML to indicate discrete elements, groupings and containment.

UML syntax, 2

- Arrows: arrows indicate all manner of things, depending on which particular type of UML diagram they're in. Usually, arrows indicate flow, dependency, association or generalization.
- Cardinality: applied to arrows, cardinalities show relative numerical relationships between elements in a model: 1 to 1, 1 to many, etc.

UML syntax, 3

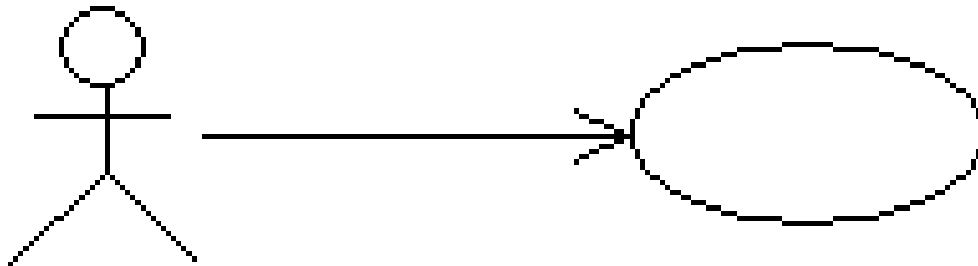
- Constraints: allow notation of arbitrary constraints on model elements. Used, for example, to constrain the value of a class attribute (a piece of data).
- Stereotypes: allow us to extend the semantics of UML with English. A stereotype is usually a word or short phrase that describes what a diagram element does. That is, we mark an element with a word that will remind us of a common (stereotypical) role for that sort of thing. Stereotypes should always be applied consistently (with the same intended meaning in all instances).

UML diagrams: use cases

- A use case encodes a typical user interaction with the system. In particular, it:
 - captures some user-visible function.
 - achieves some concrete goal for the user.
- A complete set of use cases largely defines the requirements for your system: everything the user can see, and would like to do.
- The granularity of your use cases determines the number of them (for your system). A clear design depends on showing the right level of detail.
- A use case maps actors to functions. The actors need not be people.

Use case examples, 1

(High-level use case for powerpoint.)



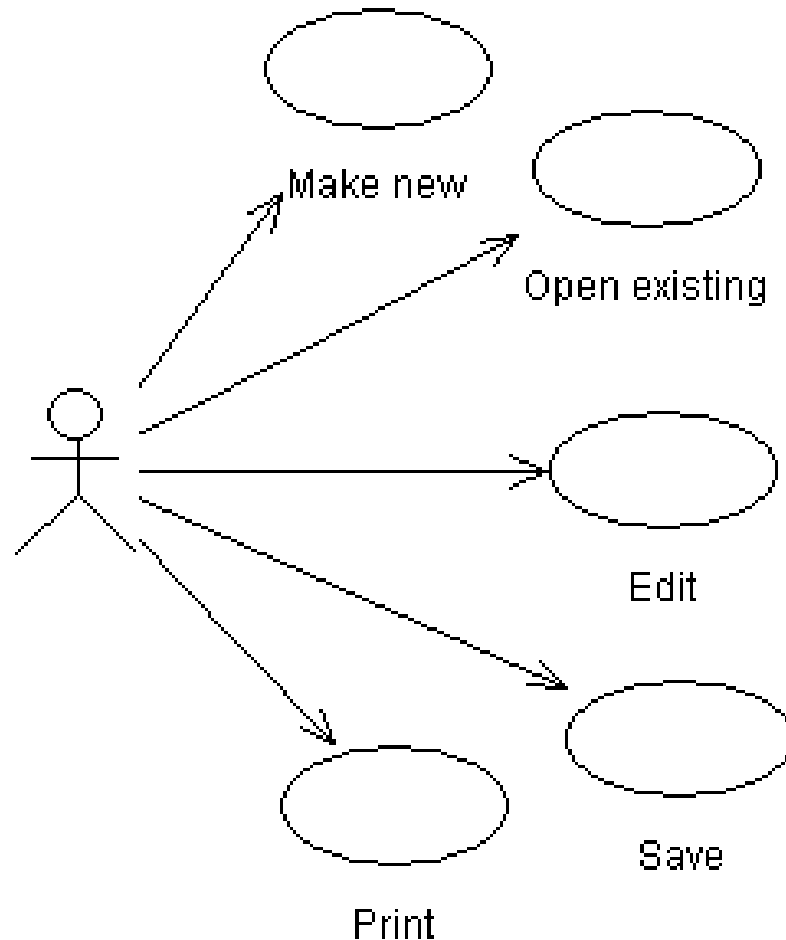
Create slide presentation

About the last example...

- Although this is a valid use case for powerpoint, and it completely captures user interaction with powerpoint, it's too vague to be useful.

Use case examples, 2

(Finer-grained use cases for powerpoint.)

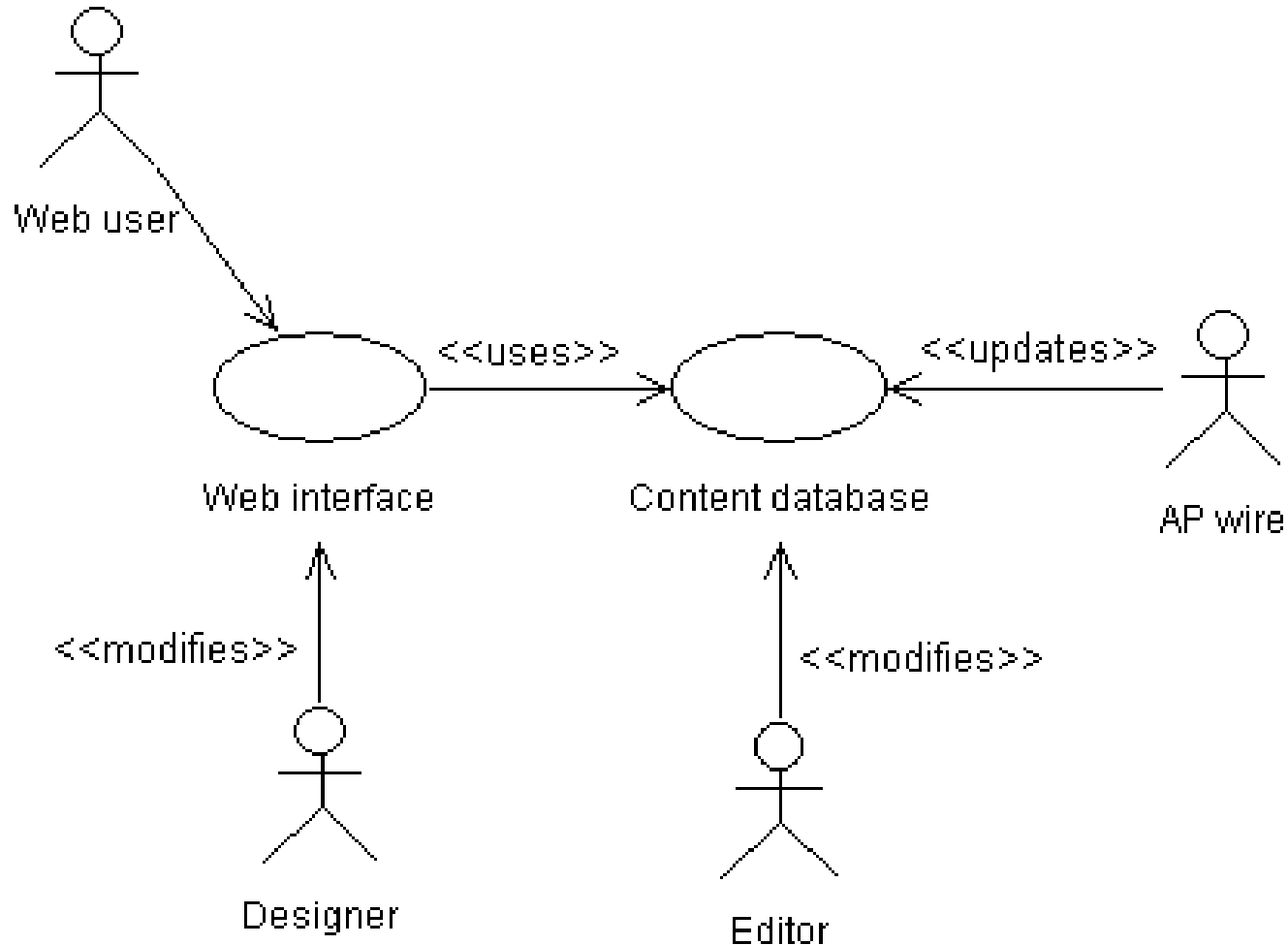


About the last example...

- The last example gives a more useful view of powerpoint (or any similar application).
- The cases are vague, but they focus your attention the key features, and would help in developing a more detailed requirements specification.
- It still doesn't give enough information to characterize powerpoint, which could be specified with tens or hundreds of use cases (though doing so might not be very useful either).

Use case examples, 3

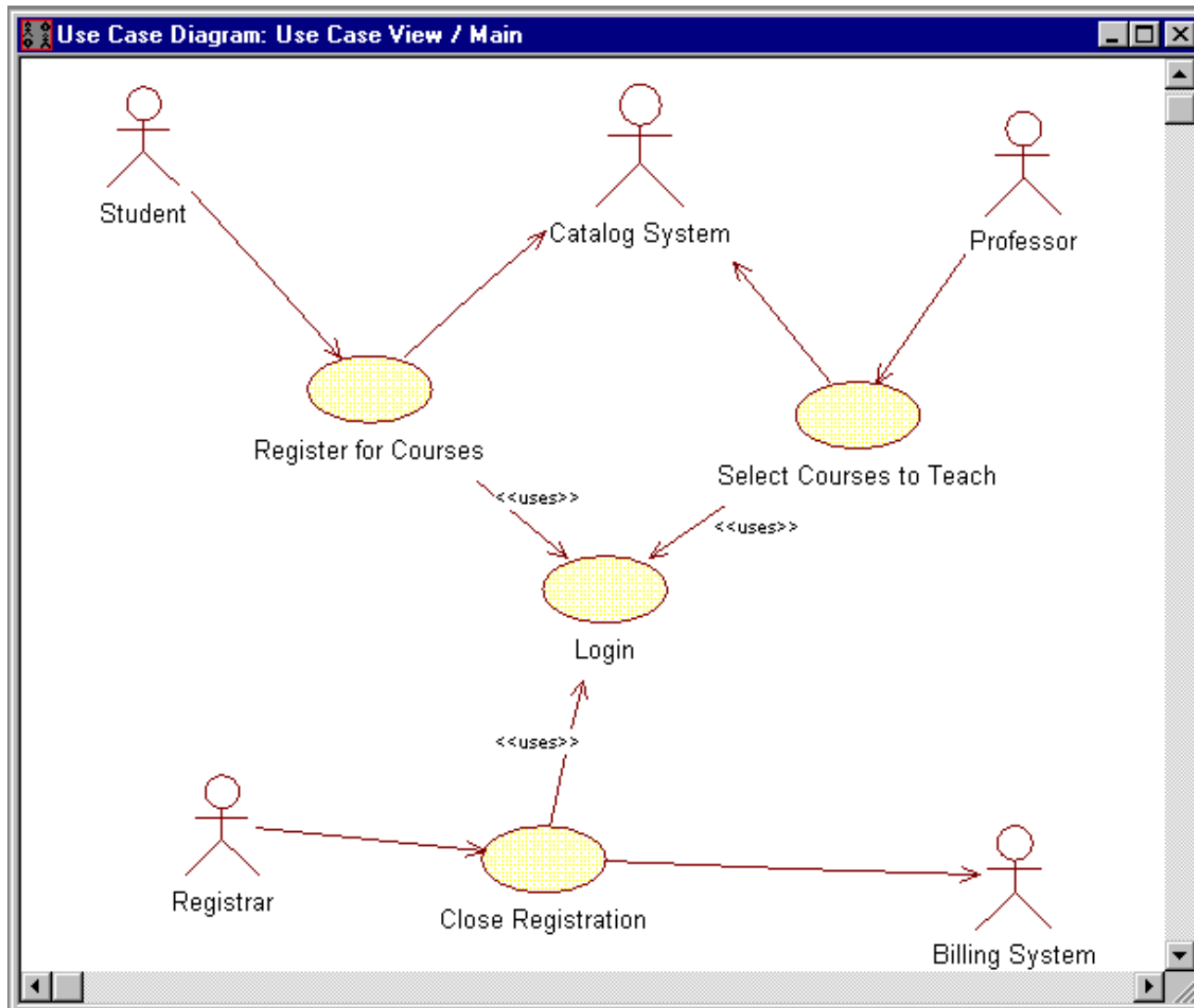
(Relationships in a news web site.)



About the last example...

- The last is more complicated and realistic use case diagram. It captures several key use cases for the system.
- Note the multiple actors. In particular, ‘AP wire’ is an actor, with an important interaction with the system, but is not a person (or even a computer system, necessarily).
- The notes between << >> marks are *stereotypes*: identifiers added to make the diagram more informative. Here they differentiate between different roles (ie, different meanings of an arrow in this diagram).

Use Case Relationships



Class Diagrams

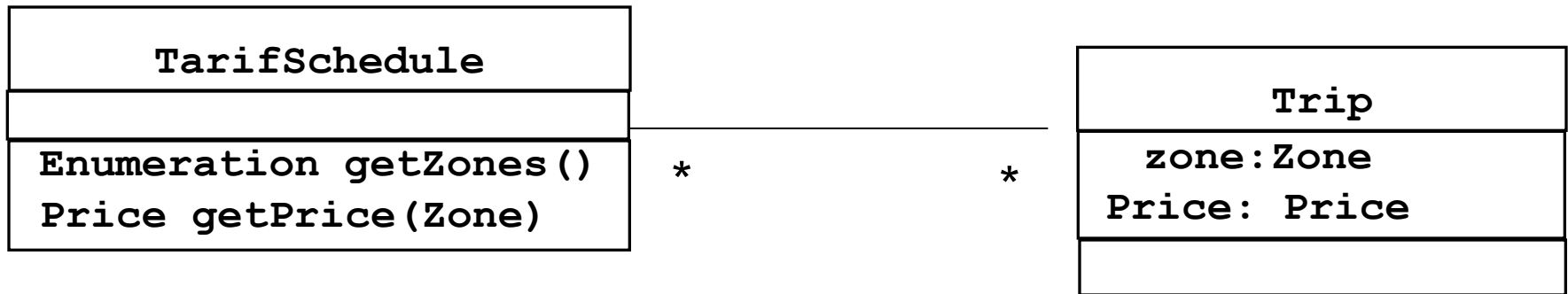
UML diagrams: class diagram

- Motivated by Object-Oriented design and programming (OOD, OOP).
- A class diagram partitions the system into areas of responsibility (classes), and shows “associations” (dependencies) between them.
- Attributes (data), operations (methods), constraints, part-of (navigability) and type-of (inheritance) relationships, access, and cardinality (1 to many) may all be noted.

Class diagram “perspective”

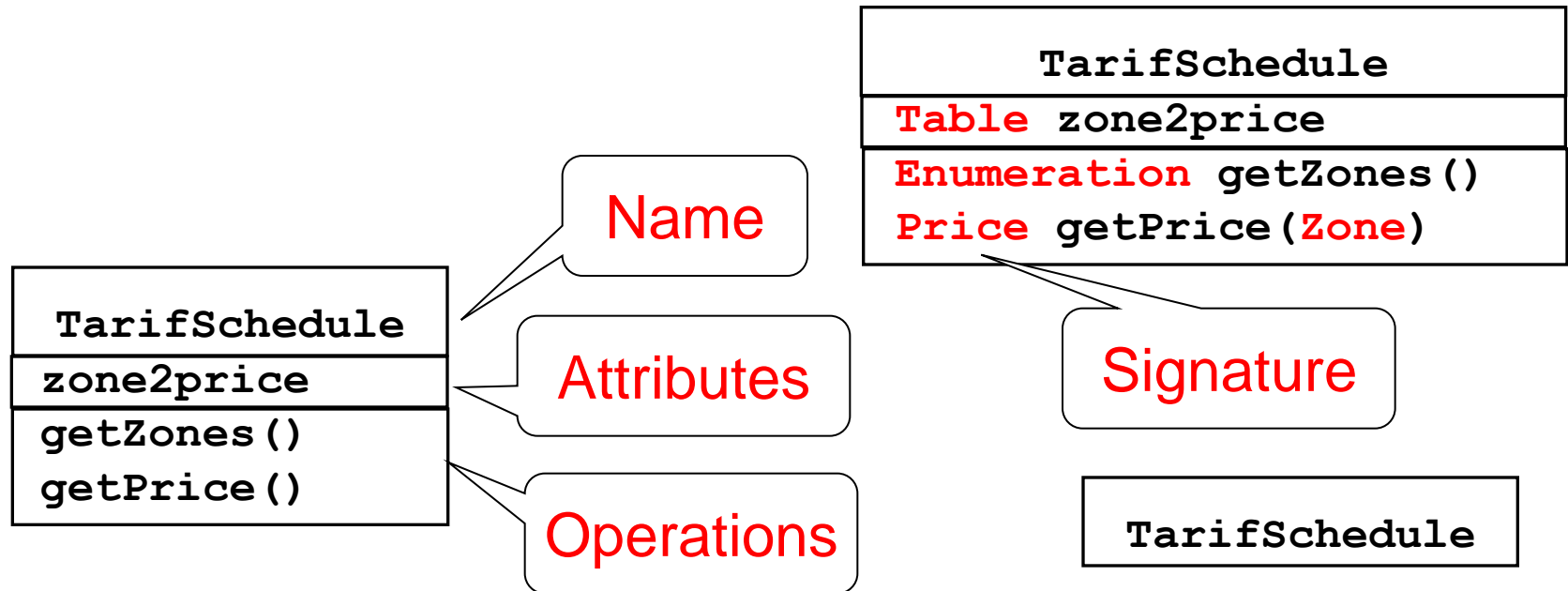
- Class diagrams can make sense at three distinct levels, or perspectives:
 - Conceptual: the diagram represents the concepts in the project domain. That is, it is a partitioning of the relevant roles and responsibilities in the domain.
 - Specification: shows interfaces between components in the software. Interfaces are independent of implementation.
 - Implementation: shows classes that correspond directly to computer code (often Java or C++ classes). Serves as a blueprint for an actual realization of the software in code.

Class Diagrams = OO?



- ♦ Class diagrams represent the structure of the system.
- ♦ Used
 - ♦ during requirements analysis to model problem domain concepts
 - ♦ during system design to model subsystems and interfaces
 - ♦ during object design to model classes.

Classes



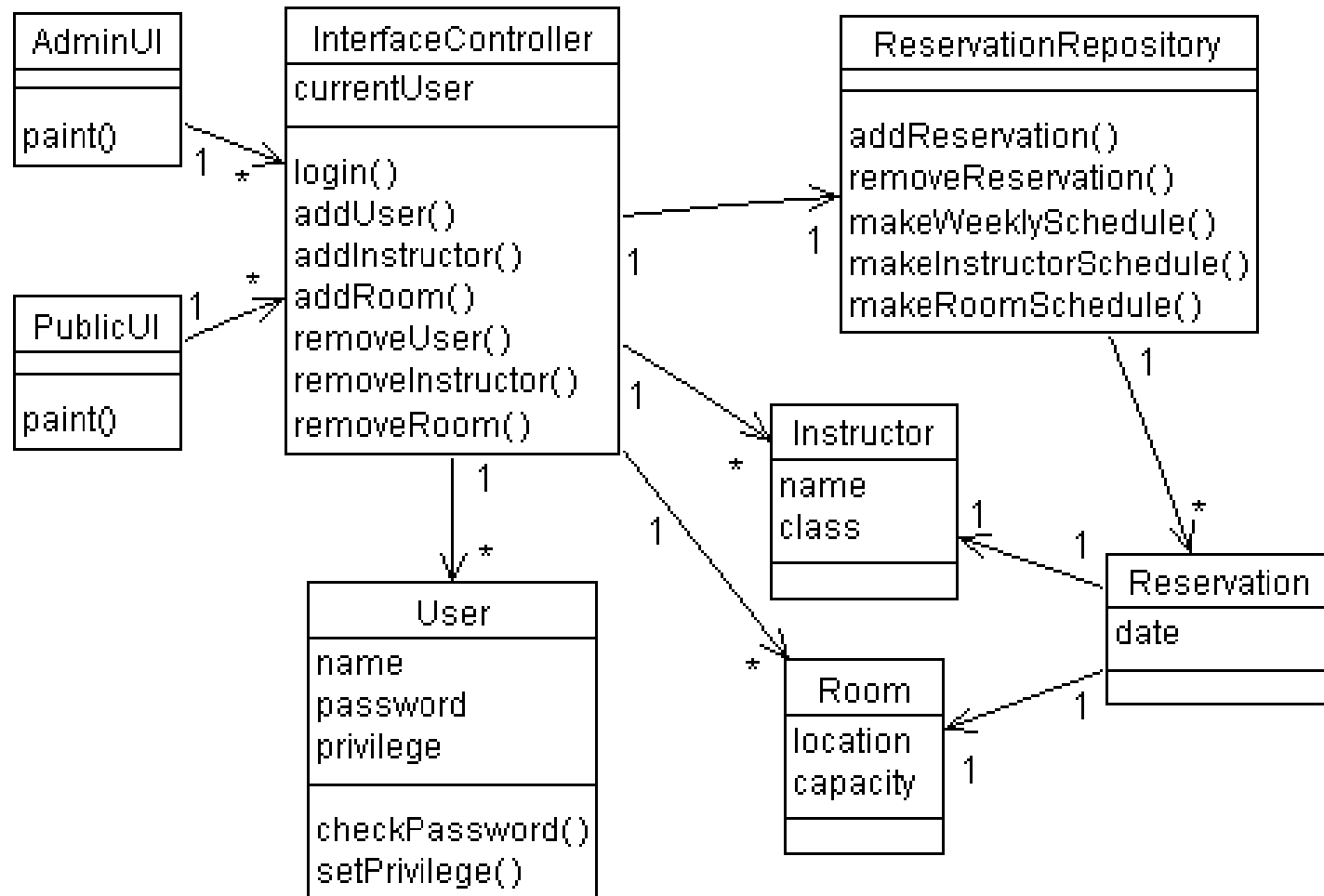
- ♦ A *class* represent a concept
- ♦ A class encapsulates state (*attributes*) and behavior (*operations*).
- ♦ Each attribute has a *type*.
- ♦ Each operation has a *signature*.
- ♦ The class name is the only mandatory information.

Actor vs Instances

- ◆ What is the difference between an *actor* , a *class* and an *instance*?
- ◆ Actor:
 - ◆ An entity outside the system to be modeled, interacting with the system (“Passenger”)
- ◆ Class:
 - ◆ An abstraction modeling an entity in the problem domain, must be modeled inside the system (“User”) ***Would you agree?***
- ◆ Object:
 - ◆ A specific instance of a class (“Joe, the passenger who is purchasing a ticket from the ticket distributor”).

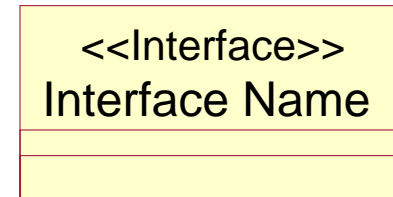
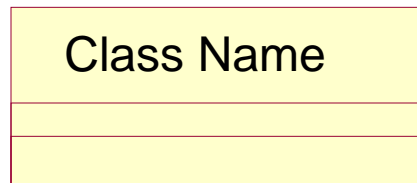
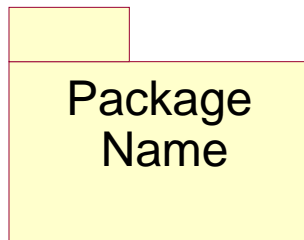
Class diagram examples

(A classroom scheduling system: specification perspective.)



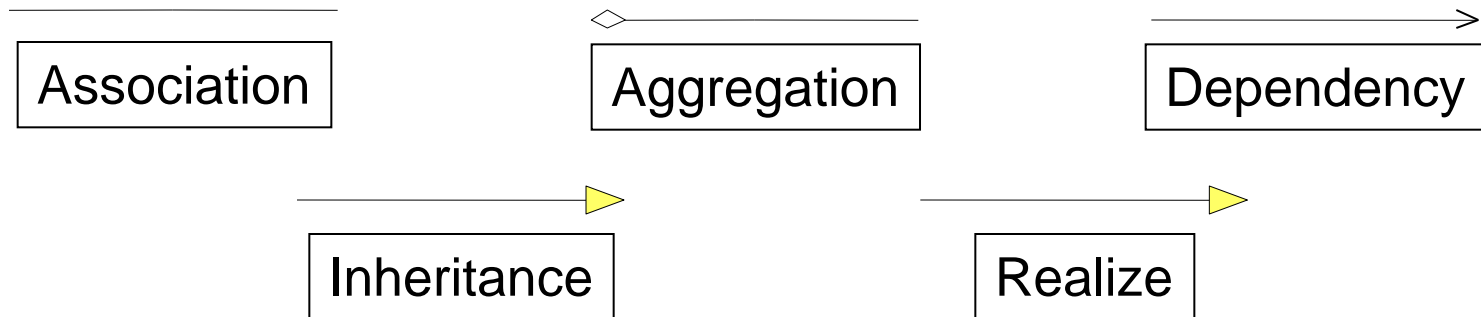
What is a Class Diagram?

- A class diagram is a view of the static structure of a system
 - Models contain many class diagrams
- Class diagrams contain:
 - Packages, classes, interfaces, and relationships
- Notation:



Relationships

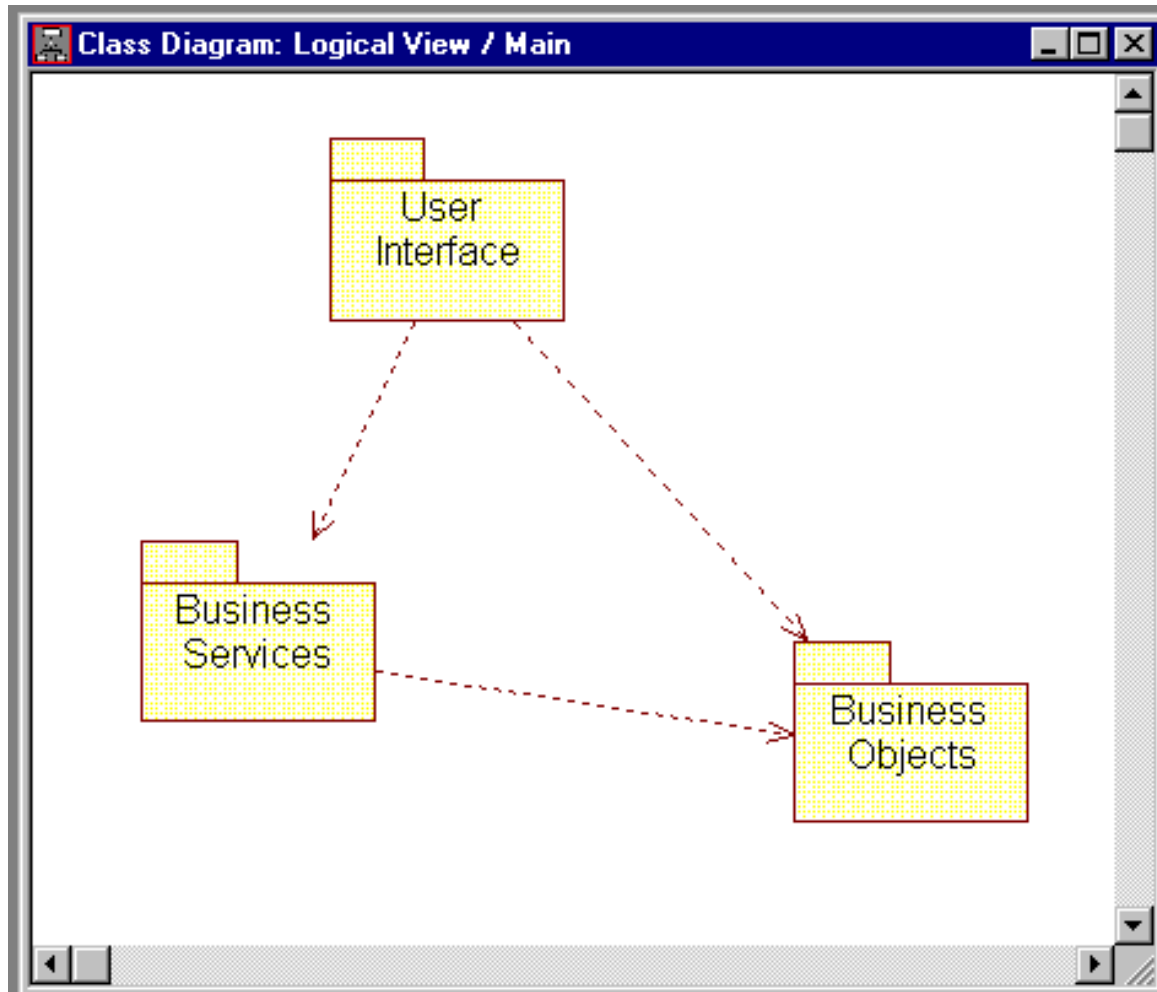
- Class diagrams may contain the following relationships:
 - Association, aggregation, dependency, realize, and inheritance
- Notation:



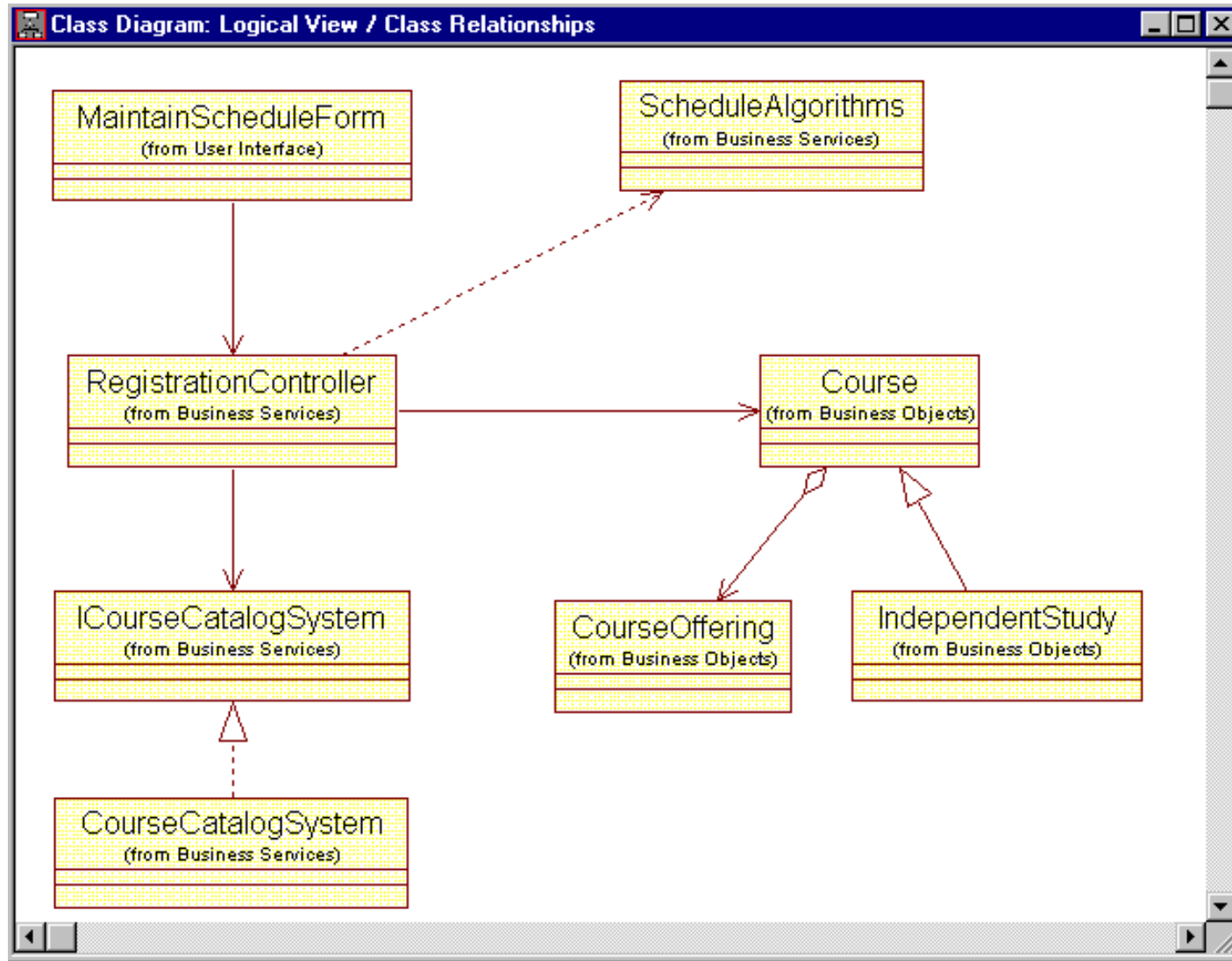
About the next 2 slides...

- The next slide shows a package diagram, with dependencies.
- The following slide shows a class diagram, with various associations between the classes.

Package Relationships



Class Relationships








About the next 2 slides...

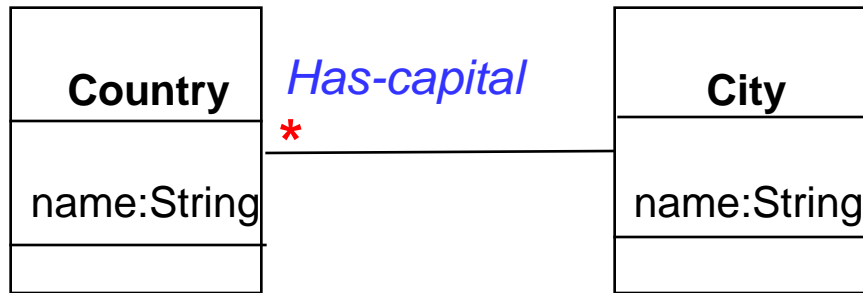
- The next slide shows how cardinalities are denoted in Rose.
- The following slide is the class diagram example from before, but this time with cardinalities marked on the associations.

Multiplicity Indicators

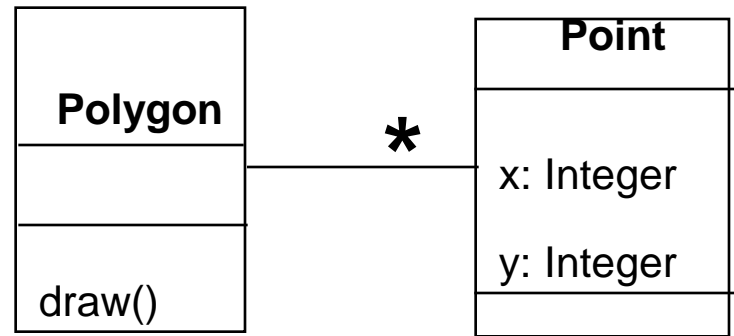
- Each end of an association or aggregation contains a multiplicity indicator
 - Indicates the number of objects participating in the relationship

 1	Exactly one
 0..*	Zero or more
 1..*	One or more
 0..1	Zero or one
 2..7	Specified range

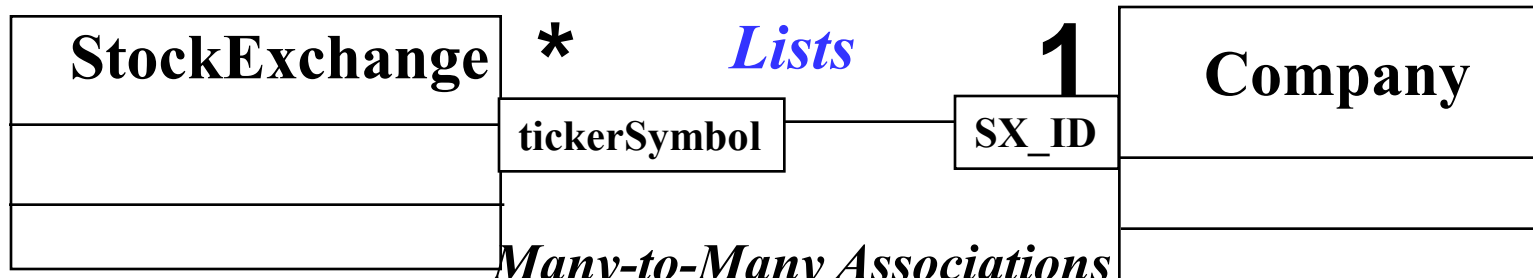
1-to-1, 1-to-many, many-to-many Associations



One-to-one association



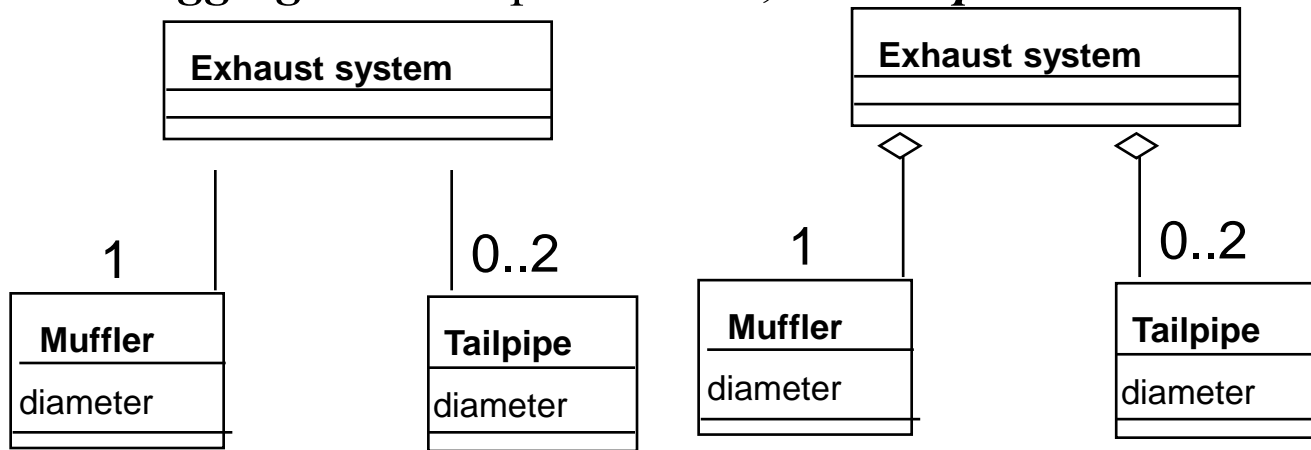
One-to-many association



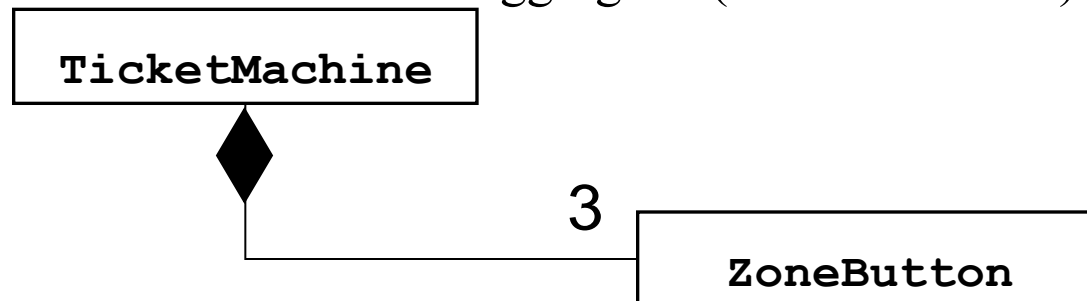
Many-to-Many Associations

Aggregation

- ♦ An **aggregation** is a special case of association denoting a “consists of” hierarchy.
- ♦ The **aggregate** is the parent class, the **components** are the children class.



- ♦ A solid diamond denotes **composition**, a strong form of aggregation where components cannot exist without the aggregate. (Bill of Material)

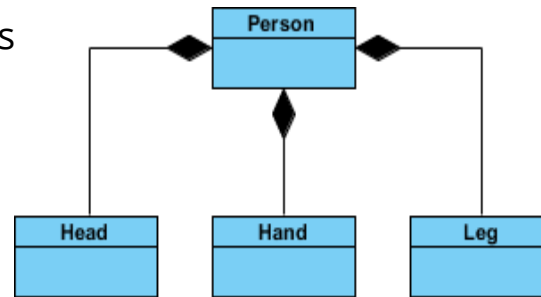


Aggregation and **Composition** are subsets of association meaning they are **specific cases of association**. In both aggregation and composition object of one class "owns" object of another class. But there is a subtle difference:

- Aggregation** implies a relationship where the child can exist independently of the parent. Example: Class (parent) and Student (child). Delete the Class and the Students still exist.
- Composition** implies a relationship where the child cannot exist independent of the parent. Example: House (parent) and Room (child). Rooms don't exist separate to a House.

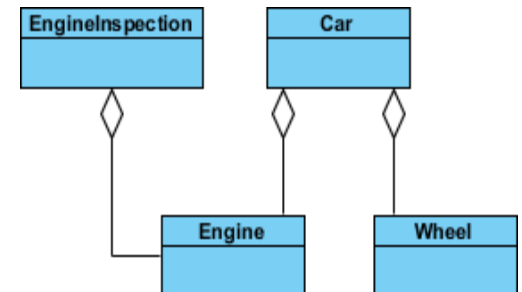
Composition Example:

We should be more specific and use the composition link in cases where in addition to the part-of relationship between Class A and Class B - there's a strong lifecycle dependency between the two, meaning that when Class A is deleted then Class B is als

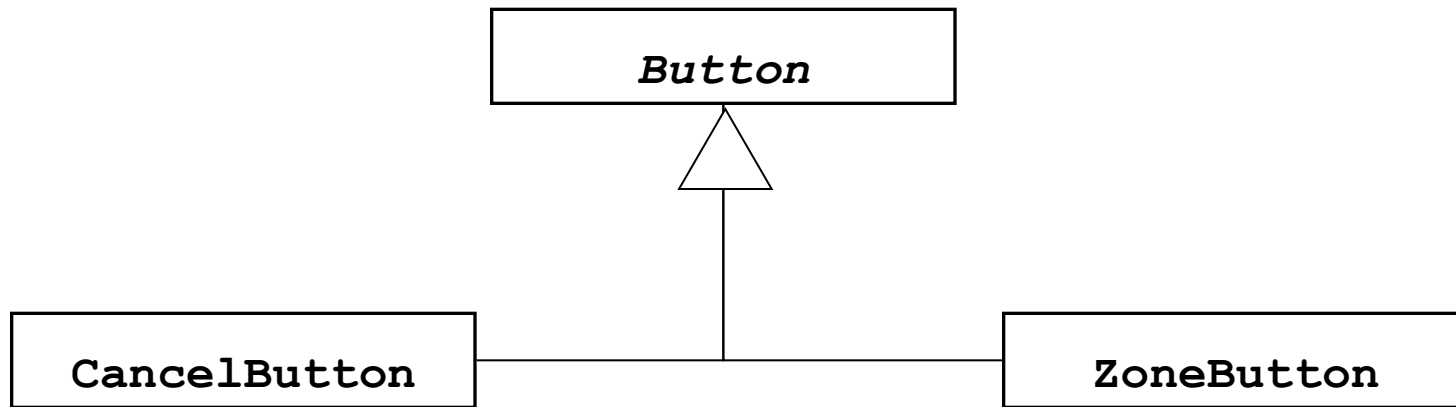


Aggregation Example:

It's important to note that the **aggregation link doesn't state in any way that Class A owns Class B nor that there's a parent-child relationship** (when parent deleted all its child's are being deleted as a result) between the two. Actually, quite the opposite! The aggregation link is usually used to stress the point that Class A instance is not the exclusive container of Class B instance, as in fact the same Class B instance has another container/s.

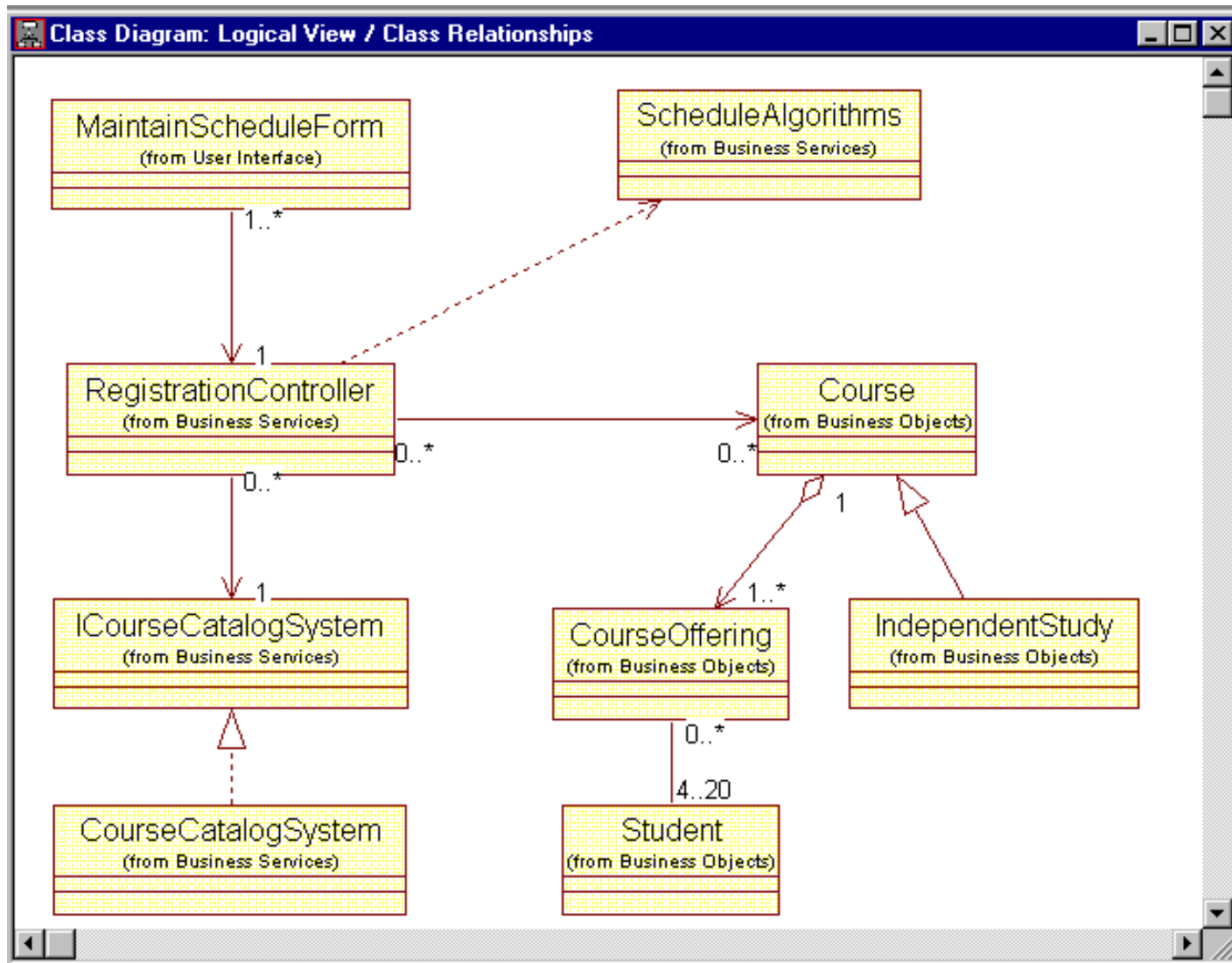


Inheritance



- ♦ The **children classes** inherit the attributes and operations of the **parent class**. **What else?**
- ♦ Inheritance simplifies the model by eliminating redundancy.

Multiplicity Indicators

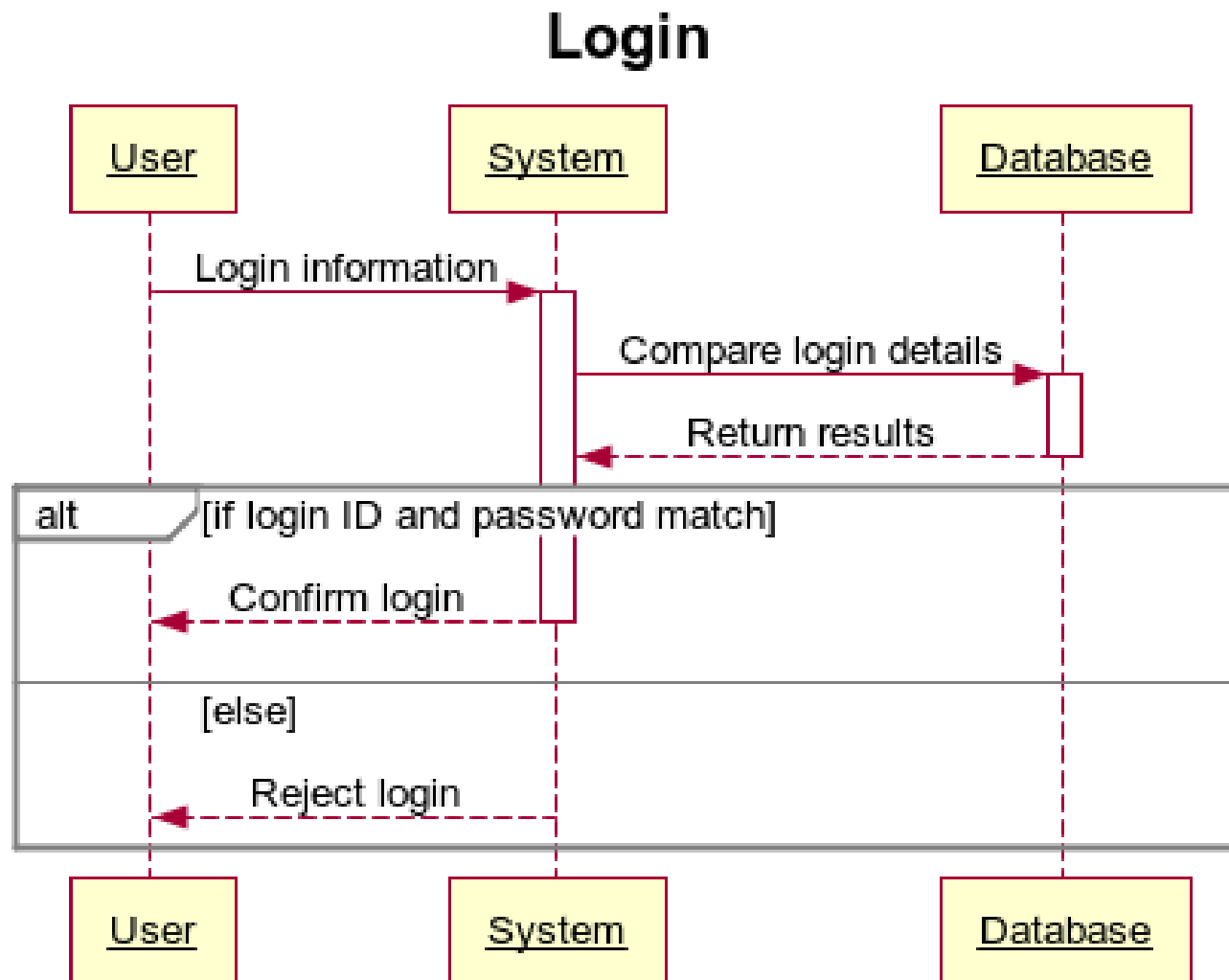


Sequence Diagrams

UML diagrams: sequence diagram

- Sequence diagram describe algorithms, though usually at a high level: the operations in a useful sequence diagram specify the “message passing” (method invocation) between objects (classes, roles) in the system.
- The notation is based on each object’s life span, with message passing marked in time-order between the objects. Iteration and conditional operations may be specified.
- May in principle be used at the same three levels as class diagrams, though the specification level will usually be most useful. (At the implementation level, you might better use pseudocode.)

Sequence diagram example



Other Diagrams

UML diagrams: Package diagram

- A type of class diagram, package diagrams show dependencies between high-level system component.
- A “package” is usually a collection of related classes, and will usually be specified by it’s own class diagram.
- The software in two distinct packages is separate; packages only interact through well-defined interfaces, there is no direct sharing of data or code.
- Not all packages in a system’s package diagram are new software; many packages (components) in a complex system are often already available as existing or off-the-shelf software.

Package diagram example

