Class Diagrams

Massimo Felici

Room 1402, JCMB, KB

0131 650 5899

mfelici@inf.ed.ac.uk

Class Diagrams

- support **architectural design**
  - Provide a structural view of systems
- Represent the basics of **Object-Oriented systems**
  - identify what **classes** there are, how they **interrelate** and how they **interact**
  - Capture the **static** structure of Object-Oriented systems - how systems are structured rather than how they behave
- Constrain interactions and collaborations that support functional requirements
  - **Link to Requirements**

Class Diagram Rationale

- Desirable to build systems **quickly** and **cheaply** (and to meet requirements)
  - All required behaviour can be realized simply from objects in the classes of the system
  - The system consists of a collection of objects in the implemented classes (e.g., there may be a GUI coordinate human interaction with the other parts of the system)
- Desirable to make the system easy to **maintain** and **modify**
  - The classes should be derived from the (user) domain – avoid abstract object
  - Classes provide limited support to capture system behaviour – avoid to capture non-functional requirements of the system as classes

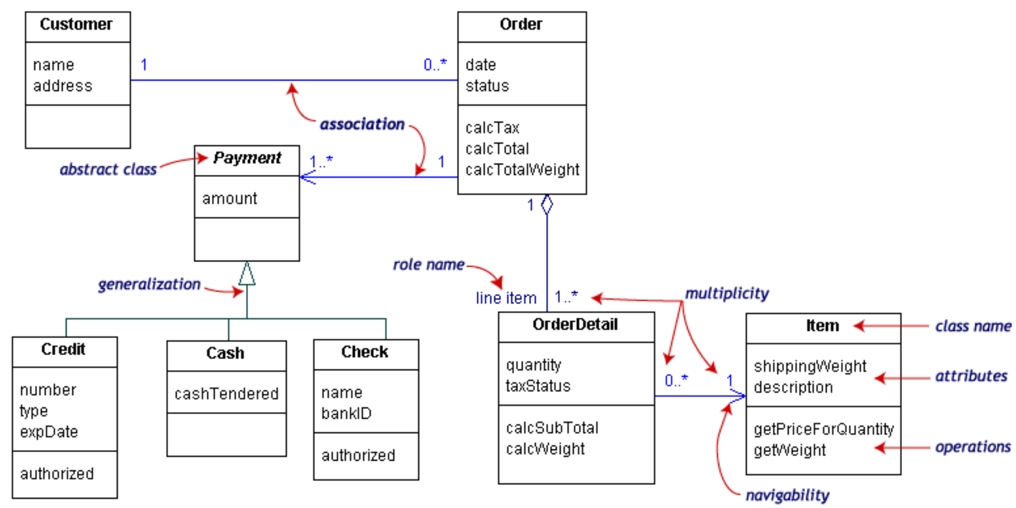
Class Diagrams in the Life Cycle

- They can be used throughout the development life cycle
- Class diagram carry different information depending on the phase of the development process and the level of detail being considered
  - The contents of a class diagram will reflect this change in emphasis during the development process
  - Initially, class diagrams reflect the **problem domain**, which is familiar to end-users
  - As development progresses, class diagrams move towards the **implementation domain**, which is familiar to software engineers

Class Diagrams at a Glance

- **Class Diagram Basics**
  - classes and associations
- **Classes**
  - Basic Class Components
  - Attributes and Operations
- Class **Relationships**
  - Associations
  - Generalizations
  - Aggregations and Compositions
- **Construction** involves:

1. Modeling **classes**
2. Modeling **associations** between classes and
3. Refining and elaborate as necessary



Objects and Object Classes

- **Objects** are entities in a software system which represent instances of real-world and system entities
- **Objects** derive from:
  - **Things**: tangible, real-world objects, etc.
  - **Roles**: classes of actors in systems, e.g., students, managers, nurses, etc.
  - **Events**: admission, registration, matriculation, etc.
  - **Interactions**: meetings, tutorials, etc.

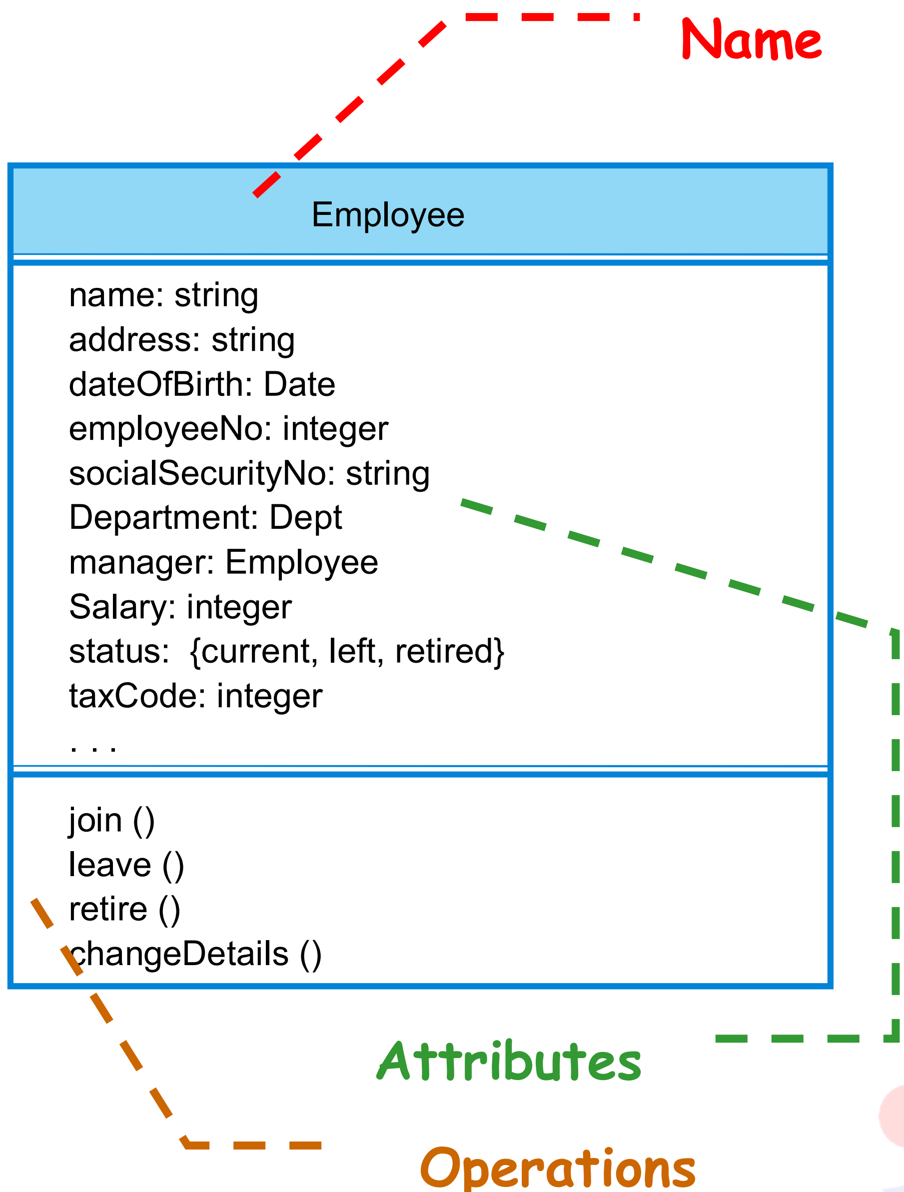
- **Object classes** are templates for objects. A description of a group of objects all with similar roles in the system
  - **Structural features** define what objects of the class know
  - **Behavioral features** define what objects of the class can do
- **Object classes** may inherit attributes and services from other object classes. They may be used to create objects

Objects and Object Classes

- An **object** is an entity that has a state and a defined set of operations which operate on that state. The state is represented as a set of object attributes. The operations associated with the object provide services to other objects (clients) which request these services when some computation is required.
- Objects are created according to some **object class** definition. An object class definition serves as a template for objects. It includes declarations of all the attributes and services which should be associated with an object of that class.

Basic Class Compartments

- Name
- Attributes
  - represent the state of an object of the class
  - Are descriptions of the structural or **static** features of a class
- Operations
  - define the way in which objects may interact
  - Operations are descriptions of behavioral or **dynamic** features of a class
- Note that the level of detail known or displayed for attributes and operations depends on the phase of the development process
- Objects are instances of classes

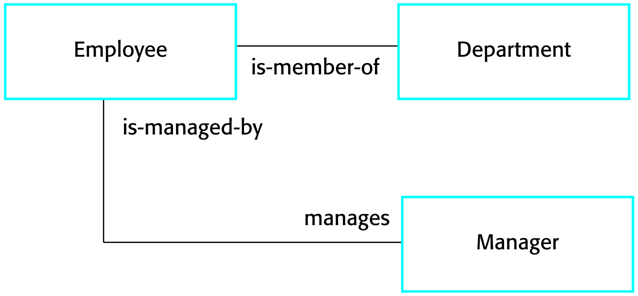


Attributes and Operations

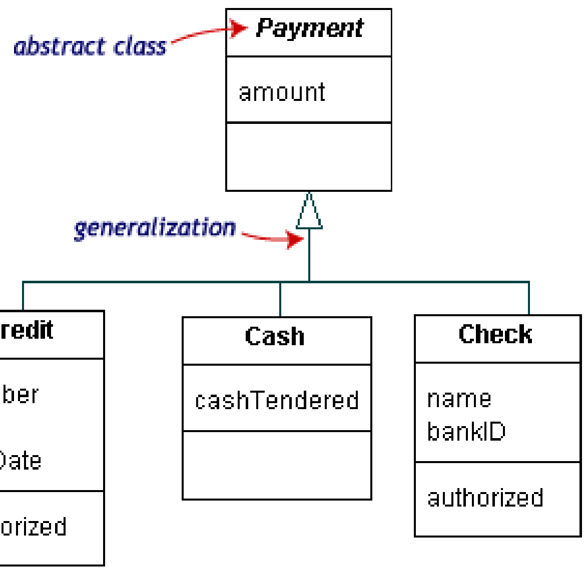
- <featureName>:<type>
- Type is the data type of the attribute or the data returned by the operation
- Visibility: private (-), public (+) or protected (#)
- **Attributes**
  - **Initial value**, **Derived Attribute**, Multiplicity [m..n]
  - Examples of **Multiplicity:** n..m - n to m instances; **0..1** - zero or one instance; **0..*** or ***** - no limit on the number of instances (including none). **1** - exactly one instance; **1..*** at least one instance
- **Operations**
  - Parameters (passed by **value** or by **reference**), Method Note, Grouping by **Stereotype**
  - A **Method Note** captures the actual implementation of operations

Associations

- Objects (classes) participate in relationships with other objects (classes)
  - (binary or n-ary) **relationships** between instances (i.e., objects) of classes
- **Associations**
  - an attribute of an **object** is an associated **object**
  - a method relies on an associated object
  - an instance of one class must know about the other in order to perform its work
  - Passing messages and receiving responses
- **Associations** may be annotated with information
  - Name, Multiplicity, Role Name, Ends, Navigation



Generalizations



- an inheritance link indicating one class is a superclass of the other, the subclass
  - An object of a subclass to be used as a member of the superclass
  - The behavior of the two specific classes on receiving the same message should be similar
- A generalization has a triangle pointing to the superclass
- Payment is a superclass of **Cash**, **Check**, and **Credit**

Generalizations continued

- **Checking Generalizations**
  - If class A is a generalization of a class B, then "Every B is an A"
- **Design by Contract**
  - A **subclass** must keep to the contract of the **superclass** by: ensuring operations observe the pre and post conditions on the methods and that the class invariant is maintained
- **Implementing Generalizations**
  - Java: creating the subclass by extending the super class
  - Inheritance increases system coupling
  - Modifying the superclass methods may require changes in many subclasses
  - Restrict inheritance to conceptual modeling
  - Avoid using inheritance when some other association is more appropriate

Aggregations and Compositions

- **Aggregations**
  - are used to indicate that, as well as having attributes of its own, an instance of one class may consist of, or include, instances of another class
  - are an association in which one class belongs to a collection.
  - have a diamond end pointing to the part containing the whole.
- **Compositions**
  - imply coincident lifetime. A coincident lifetime means that when the whole end of the association is created (deleted), the the part components are created (deleted).

Modeling by Class Diagrams

- **Class Diagrams** (models)
  - from a **conceptual viewpoint**, reflect the requirements of a problem domain
  - From a **specification (or implementation) viewpoint**, reflect the intended design or implementation, respectively, of a software system
- **Producing** class diagrams involve the following **iterative** activities:
  - Find **classes** and **associations** (directly from the **use cases**)
  - Identify **attributes** and **operations** and allocate to classes
  - Identify **generalization** structures

How to build a class diagram

- Design is driven by criterion of completeness either of data or responsibility
  - **Data Driven Design** identifies all the data and see it is covered by some collection of objects of the classes of the system
  - **Responsibility Driven Design** identifies all the responsibilities of the system and see they are covered by a collection of objects of the classes of the system
- **Noun identification**
  - **Identify noun phrases**: look at the use cases and identify a noun phrase. Do this systematically and do not eliminate possibilities
  - **Eliminate inappropriate candidates**: those which are redundant, vague, outside system scope, an attribute of the system, etc.
- Validate the model…

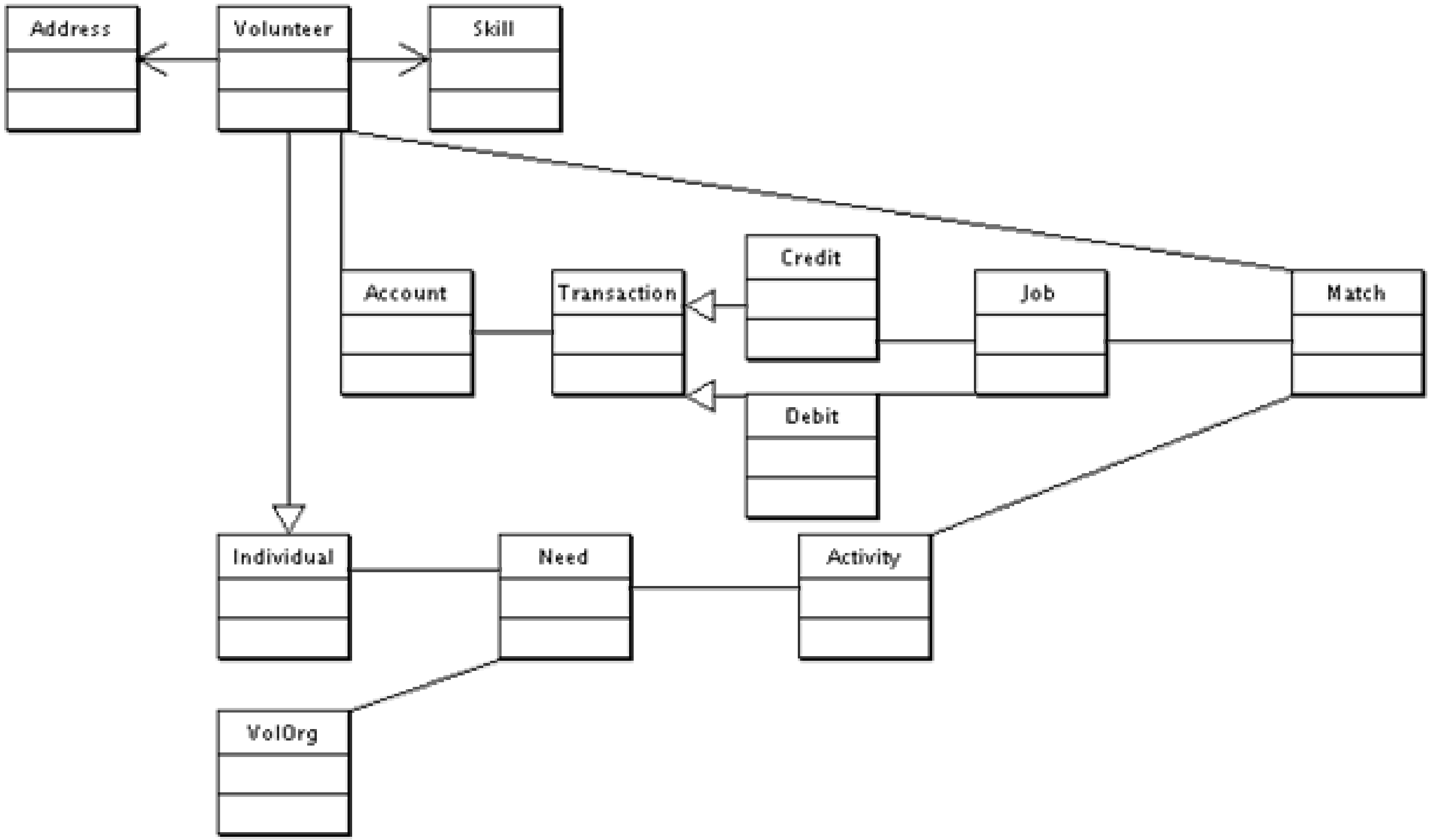
Common Domain Modeling Mistakes

- Overly specific **noun-phrase analysis**
- Counter-intuitive or incomprehensible **class** and **association names**
- Assigning **multiplicities** to associations too soon
- Addressing **implementation issues** too early:
  - Presuming a specific implementation strategy
  - Committing to implementation constructs
  - Tackling implementation issues
- Optimizing for **reuse** before checking use cases achieved

Class and Object Pitfalls

- Confusing basic **class relationships** (i.e., is-a, has-a, is-implemented-using)
- Poor use of **inheritance**
  - Violating encapsulation and/or increasing coupling
  - Base classes do too much or too little
  - Not preserving base class invariants
  - Confusing interface inheritance with implementation inheritance
  - Using multiple inheritance to invert is-a

VolBank: Early Class Diagram



Reading/Activity

- Please review the use of ArgoUML in the generation of UML diagrams
  - http://argouml.tigris.org/tours

Summary

- Class Diagrams in the life cycle
- Class Diagram Rationale
- Classes
    - Basic Class Components
    - Attributes and Operations
- Class Relationships
    - Associations
    - Generalizations
    - Aggregations and Compositions
- Modeling by Class Diagrams
    - How to build a class diagram
    - Common domain modeling mistakes
    - Class and Object Pitfalls