# Static specifications in UML: Class Models

CS3: SEOC1

Note 4

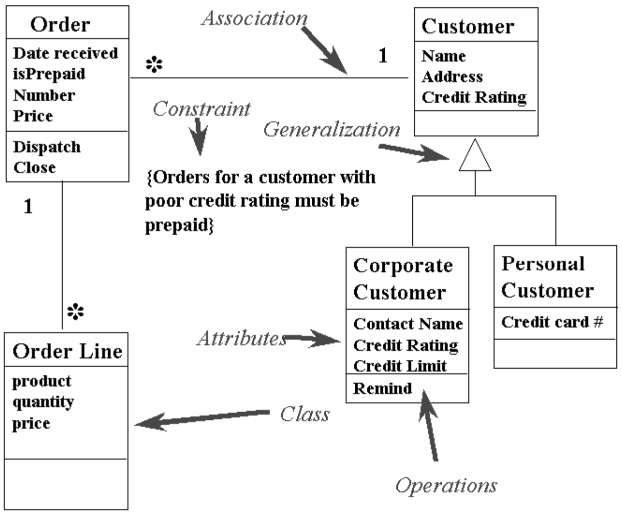


## Class Models

*Class diagrams are used to document the static structure of OO systems. They indicate what classes there are, how they interrelate, and how they interact.*



## Annotated Example Class Diagram






## Quality of Class Models

- identify the classes and their relationships
- desirable to build the system quickly and cheaply (and to meet requirements):
  - all required behaviour can be realised simply from objects in the classes of the system
  - the system *is* some collection of objects in the classes we have (there may be a GUI that coordinates human interaction with the objects)
- desirable to make the system easy to maintain and modify
  - the classes should be derived from the domain – avoid abstract objects introduced to "simplify" implementation
  - don't incorporate short lived features of the system as classes

## How to Build Class Models

**What drives design:** Driven by criterion of completeness either of data or responsibility

**Data Driven Design:** identify all the data and see it is covered by some collection of objects of the classes of the system.

**Responsibility Driven Design:** identify all the responsibilities of the system and see they are covered by a collection of objects of the classes of the system



## How to Build Class Models (cont)

**Noun identification:** As described in note 3:

**Identify noun phrases:** look at the use cases and other requirements documents and identify noun phrases. Do this systematically and do not eliminate possibilities at this stage.

**Eliminate inappropriate candidates:** those which are redundant, vague, an event or operation, in the meta-language, outside system scope, an attribute of the system.

**Validate the model:** using CRC cards – see later.



## What are Classes?

- A description of a group of objects all with similar roles in the system.
- Objects derive from:

**Things:** tangible, real-world objects, e.g. wards, beds, patients, ....

**Roles:** classes of actors in systems, e.g. nurses, managers, ....

**Events:** admission, discharge, updates, ... (likely if there is an audit trail where these are treated as objects).

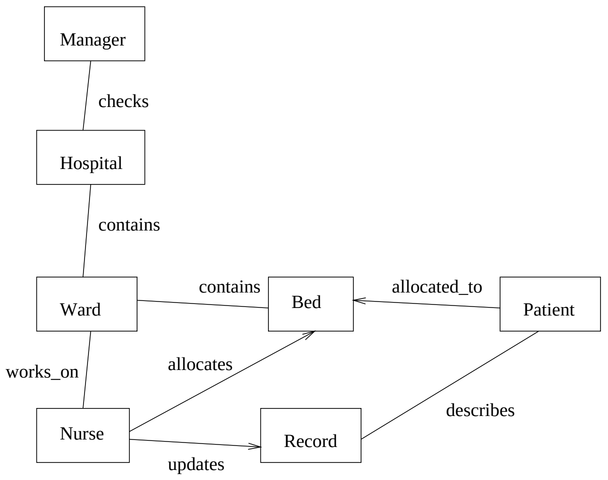
**Interactions:** meeting, handover, ... (again, likely if there is some degree of reflection on the action of the system within the system)



## Associations between Classes

- Class A and B are *associated* if:
  - an object of class A sends a message to an object of class B.
  - an object of class A creates an object of class B
  - an object of class A has attributes that are objects of class B (or collections of such objects).
  - an object of class A accepts messages having objects of class B as an argument
- design associations early – keep them conceptual initially – think about methods later
- but don't forget about implementations.

## Initial Class Diagram for HIS






## Multiplicities

- These label association links between classes and indicate then number of objects of a particular class that are related to objects of an associated class.
- A multiplicity can be:
  - a number
  - a range m..n (typically 0..n or 1..n)
  - the unspecified multiplicity: *
  - a list of multiplicities

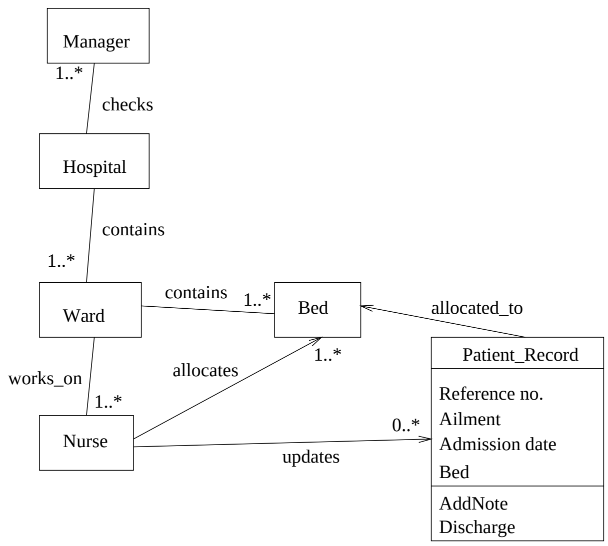


## Attributes and Operations

- after considering classes we need to think about attributes (these determine the state of an object), and the methods in a class (these define how it interacts)
- *Attributes* occupy the second compartment of a class icon. These represent the state of an object of the class (omit any that are used purely to implement the class).
- *Methods* are listed in the final compartment of the class icon, here we just specify their arguments and return values.



## Revised Class Diagram for HIS



## Generalisation

- Important relationship between classes.
- If we had a system with classes `Nurse` and `Sister` we might consider a generalisation `NursingStaff` that include the functions common to the two original classes.
- We expect:
  - An object of the more specialised class to be good for use as a member of the generalised class.
  - The behaviour of the two specific classes on receiving the same message should be similar.



## Checking for Generalisations

- Suppose we claim class `A` is a generalisation of class `B`. Then we can check by seeing if the sentence: "Every `B` is an `A`."
- *Every engineer is a worker.*
- *Engineer is a profession. Every engineer is a profession.*

## Design by Contract

- The contract is described by:
  - Pre- and post-conditions on the operations
  - Class invariants
- A specialisation of a class 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 Generalisations

- In Java this is done by creating the subclass by extending the super class.
- Inheritance increases the coupling of a system.
- Modifying the superclass methods may require changes in many subclasses of that class.
- Restrict inheritance to *conceptual* relationships.
- Avoid using inheritance when some other association is more appropriate e.g. `Engineer` might inherit from `Person` but not from `Qualification` that might be a part of an `Engineer`.



## Class Diagrams and Class Models

- Class model develops by iteration
- A class model represents the a view of the system at some level of abstraction
- A class diagram is a way of representing the model
- One model may need several diagrams to describe it, and a particular class may make several appearances in the diagrams.

## Common Domain Modelling 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 (eg, integrating legacy systems)
- Optimising for reuse before checking use cases achieved
- "Premature patern-isation"



## Class and Object Pitfalls

- Confusing basic class relationships
  - 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*
- "Object ooze":
  - Poor encapsulation
  - Bloated objects
  - Swiss army knife classes
- Object spaghetti; object hyperspaghetti

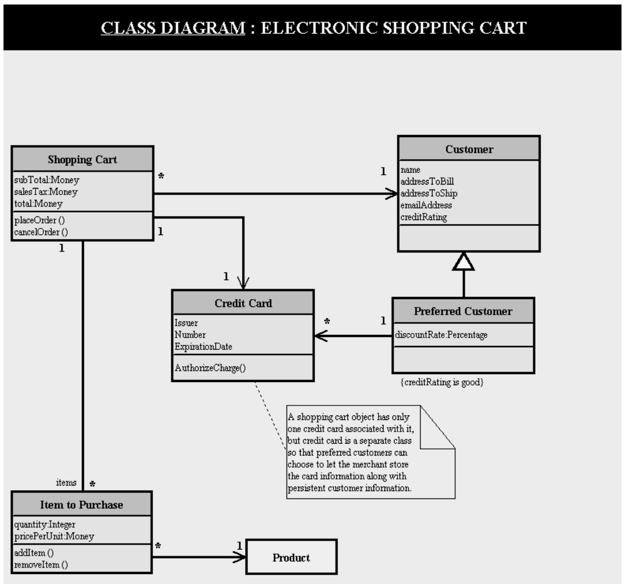


## Summary

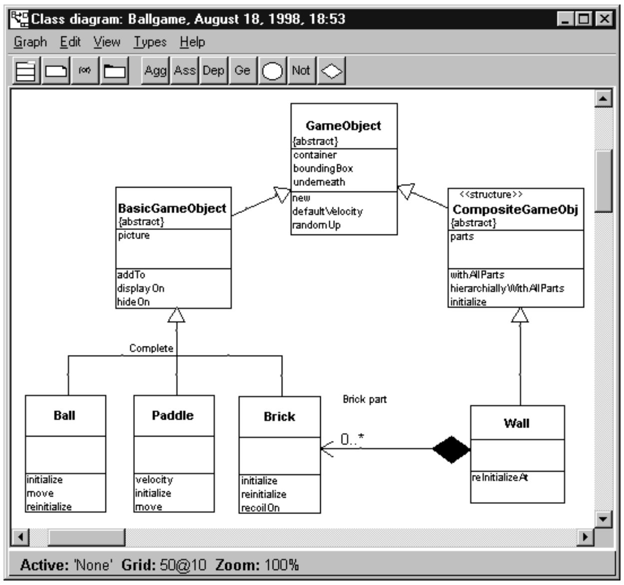
- A class model describes the static structure of an OO system
- The class models consists of class icons and associations between the icons.
- Associations may have multiplicities associated with them.
- Class icons give the name, attributes and method names and types of a class.
- Class diagrams get developed iteratively as the project progresses. It may be useful to delay decisions on attributes and methods until the main associations are sorted out.



## Further Example Class Diagrams (1)



## Further Example Class Diagrams (2)






## Further Example Class Diagrams (3)