Class diagrams

Modeling system structure
(essential concepts)

Contents

- Background
  - Object model and design principles
- Class diagram essentials
  - Classes and basic relationships
  - Attributes and operations
  - Constraints
  - Object diagram
- Method
  - Modeling techniques and advice
- See also : Class diagrams, Advanced concepts
  - Interfaces, class-associations, qualified associations, class (static) variables, composition

Background

Some background concepts

Introduction

- The class diagram is central to any object-oriented model.
- It describes the types (classes) of objects found within a system, and the static relationships between them.
- It defines the attributes and operations of each class of object.
- It imposes constraints on the ways in which objects may be connected.
- Its concepts are directly translatable into code and are form the skeleton of a program.

The object model (1)

- The object model is about as close to the real world as a software model can get:

Real world	**Object model**	***Example***
Object	Object	*man, woman*
Relationship between objects	Relationship between objects	*marriage*
Object characteristic	Object attribute	*eye colour*
Object action	Object behaviour	*kiss!*
Collaboration	Exchange of messages	*make babies*

The object model (2)

- A system is modeled as an ensemble of objects that collaborate to realise that system's functions.
- An object is defined as an independant, self-reliant and closed entity that is characterised by:
  - attributes,
  - behaviour,
  - state,
  - identity,
  - responsability,

The object model (3)

Attributes	*Behaviour*	*State*	*Identity*	*Responsability*
Dog				
biting power teeth size	bark attack	sleeping on guard	Fido	Guard the house at night
Telephone				
number	place call send SMS	on hook off hook	06.32.54.22.76	Manage incoming and outgoing calls
Bank Account				
balance transaction history	credit interest withdraw	active overdrawn	5033-12-325-7	Manage a bank account

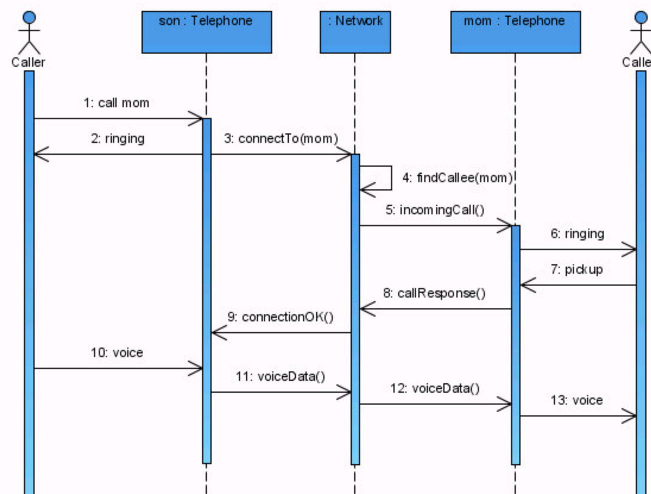
The object model (4)

- A Dog has the responsability of guarding the house ;
- Its attributes (teeth size and bark volume) and behaviour (sleep during the day, bark at night) are selected to enable it to fulfil its responsability ;
- Its state (sleeping, on guard) determines how it reacts to world events, for example a burglar ;
- Its identity allows it to be distinguished from another seemingly identical dog (Fido vs Fifi).

The object model (5)

- Objects collaborate to realise services.
- A collaboration takes the form of an exchange of messages between objects.
- A message received at an object corresponds to a request for service.
  - If the service requested falls under the responsability of the object then it renders the service;
  - otherwise it delegates the request to a component or a peer object.

The object model (6)



Object design principals

- Responsability
- Decomposition and delegation
- Encapsulation
- Abstraction

Responsability

- An object has a well-defined, limited responsability within a system.
- Ideally, an object should have exactly one responsability.
  - complex objects are more difficult to maintain and present less potential for re-use in future projects.
- An object's responsability determines the data that it will store (attributes) and the services it will offer (behaviour).

Decomposition and delegation

- A complex object should always be decomposed into simpler component objects,
- each of which is designed to be responsable for a small and specific aspect of the composed object.
- A complex object should delegate the execution of services to the most specific component object(s) possible.

Encapsulation

- Encapsulation is the separation of an object's interface from its implementation.
  - The interface defines a publicly available set of services.
  - The implementation is hidden and defines the internal realisation of those services.
- Objects therefore depend on interfaces, and not implementations.
- This allows the implementation of an object to be modified without affecting its dependent objects
  - Happens often : debugging, better algorithm, etc.

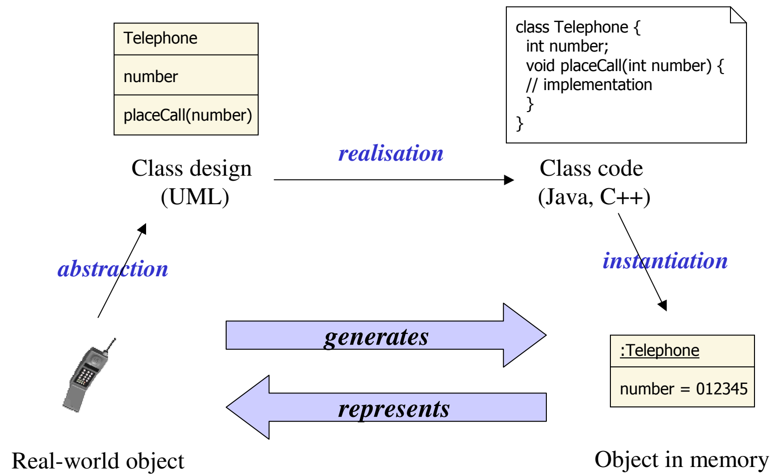
Abstraction

- Although the object model is close to the real world, the real world is too complex to model in detail.
- Abstraction is the careful omission, selection and generalisation of real world details in order to reduce model complexity.
  - Keep only the details of the real world which are pertinent to the goal of your model;
  - Generalise details of specific objects in order to reduce the number of different classes of objects in your model.

Objects and classes

- A class of objects represents an abstraction of a set of similar real world objects
  - which will have identical attributes and behaviour,
  - but individual state and
  - distinguishable identity.
- We say that an object is an instance of a class.

Objects and classes



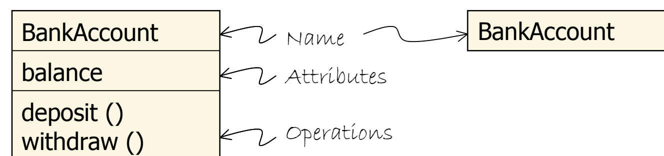
Class diagram essentials

What you will need 80% of the time

Class

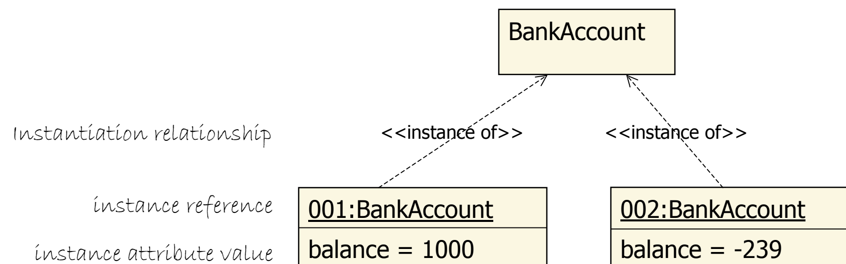
- A class is a pertinent abstraction of a set of similar real world objects found in the problem domain.
- A class defines
  - a name, describing the responsability of each instance
  - attibutes, defining the data stored by each instance
  - operations, defining the services offered by each instance
- A class is represented as a rectangle with separate "boxes" for its name, attributes and operations.

Class



•Most details can be hidden if useful
•Only the class name is obligatory

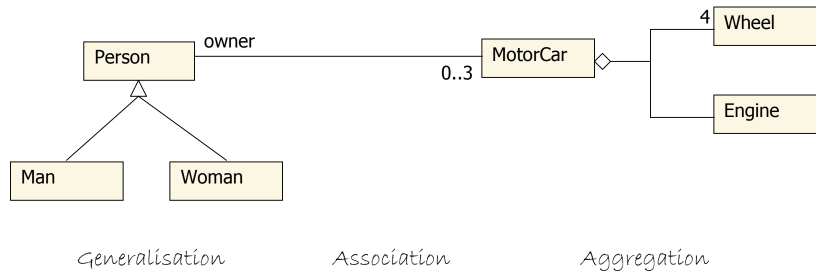
Object



Basic relationships

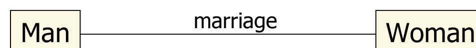
- Association
  - Defines a simple relationship between classes
- Aggregation
  - Defines an all/part relationship, used to decompose a complex class
- Generalisation
  - Defines a hierarchy of classes

Basic relationships



Association

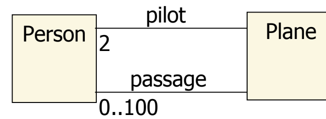
- An association defines a simple, bi-directional, named relationship between two classes of objects.
  - a person *works for* a company
  - a man is *married to* a woman
- The existence of an association between two classes enables their associated instances to communicate.



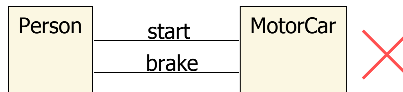
- The relationship name is optional, and is placed in the middle of the line representing the relationship.

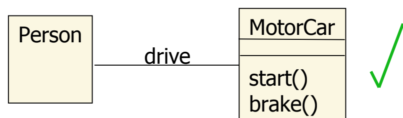
Association

- You may define multiple associations between the same two classes.



- **Danger! Common error!** Take care not to mistake the messages *allowed by* an association as new associations.

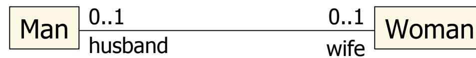




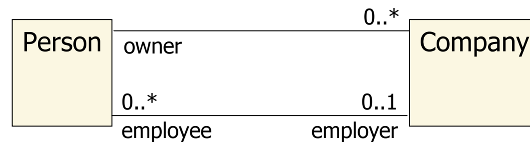
Association ends

- An association has two association ends each of which is attached to one of the classes involved in the relationship.
- An association end qualifies the class to which it is attached in two ways:
  - it **must** define a mutliplicity indicating how many instances of the class may participate in the given relationship ;
  - it **may** describe the role of the instance when this is not clear from the class's name.

Association ends



A man may be the husband of at most one woman, who is his wife; a woman may have at most one husband.



A person may be the owner of many companies; each company has exactly one owner.
A person may be the employee of at most one company; a company has many employees.

Association end multiplicity

- Defines how many instances of the class at that association end may particpate in the association.
- Default value (if not specified) is ONE (1).
- May indicate a precise value or a range of values.
- Common values:
  - *0..1* Optional (zero or one instance)
  - *N* Exactly *N* instances
  - *M..N* Between *M and N* instances
  - * Many (from zero to ∞)

Association end role

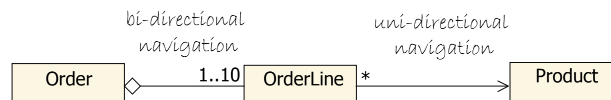
- Describes the role that an instance of the class at that association end plays in the relationship.
- If it is obvious from the class name it may be omitted.
- Useful for describing multiple relationships between the same two classes.

Association navigability

- By default, associations are bi-directional
  - Messages can be exchanged in both directions.
  - Each instance is aware of the instance(s) at the other end of the association.
  - This navigation is inverse : Navigation from one instance to another guarantees that navigation in the reverse direction always returns to the original instance.
- An association may also be defined as uni-directional
  - one of the instances will not be aware that it is part of an association

Association navigability

- Used mostly in the design phase to simplify a class's attributes, as each allowed navigation requires the storage of a reference at the source of the navigation.

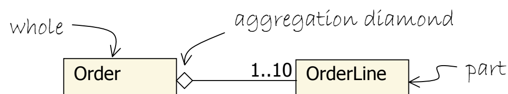


  – Navigability from Product to OrderLine would imply a reference at a Product instance for every order containing that product!

- Note that uni-directional associations are not inverse.

Aggregation

- Aggregation is the same as association,
- with the added conceptual idea that one of the classes is part of the other.
- Aggregation is useful for decomposing a complex object into smaller components.

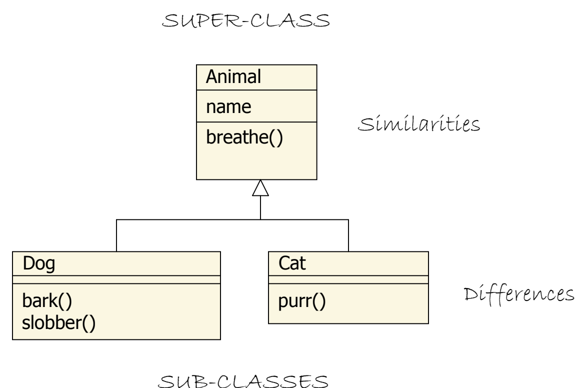


  - An Order is composed of between 1 and 10 OrderLines;
  - An OrderLine is part of exactly one Order.

Generalisation

- Generalisation allows us to define a hierarchy of classes.
- It is used when we have classes which are clearly different, but have many similarities.
  - A Dog is a type of Animal
  - A Cat is a type of Animal
- A sub-class **is a type of** its super-class.
- The similarities are placed in the super-class (Animal), and the differences in the sub-classes (Cat and Dog).

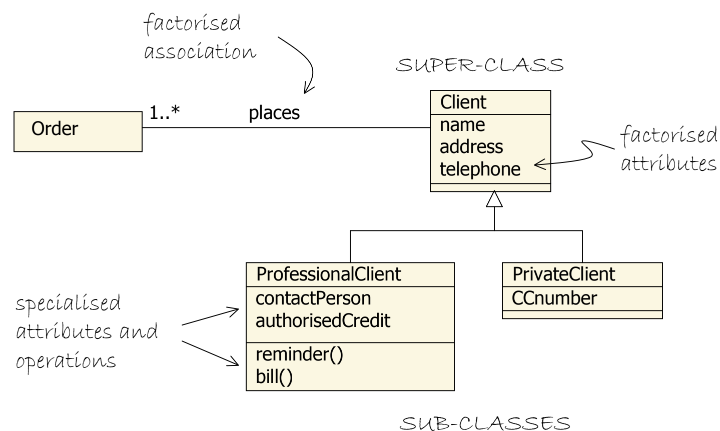
Generalisation



Generalisation for factorisation

- Generalisation allows us to factorise the common features of two classes into a super-class.
- Each of the sub-classes inherits all features of its super-class (attributes, operations and associations) and defines others of its own.

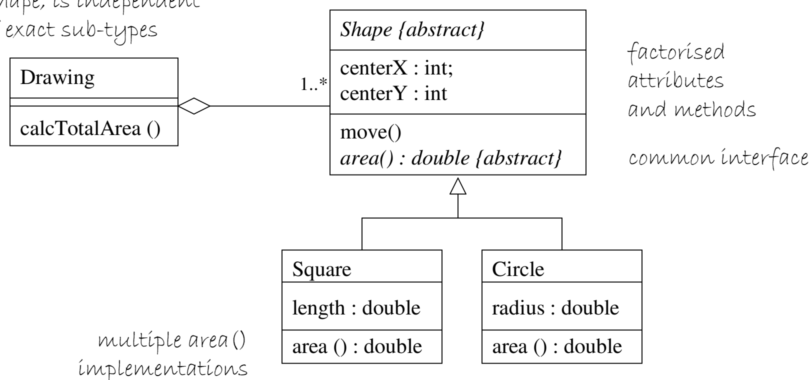
Factorisation example



Generalisation for polymorphism

- Generalisation allows us to define a common interface with multiple implementations.
- A super-class can define an interface of abstract operations which each of the sub-classes over-rides with a different implementation.
- Code which is written to work with the super-type can therefore freely use any sub-type.
  - While the implementation defined by sub-types may differ, the interface defined by the super-type is unchanging.
- The calling code is not type-dependant.

Polymorphism example



Attributes

- An attribute defines a value that qualifies an object.
  - A Car's colour attribute indicates that Cars have colours.
- An attribute defines data needed by an object to fulfil its responsability
  - A Dog's teethSize attribute is important to its job of guarding the house.
- Attributes are usually simple data structures without operations or a conceptual responsability of their own
  - Simple classes such as *String*, *Date*, …
  - or primitive (non-object) types such as *int, char,* …

Attribute syntax

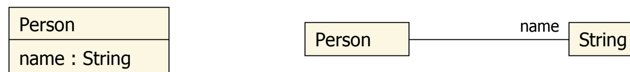
```
visibility name : type = defaultValue
```

- The amount of detail shown depends on the phase of the project.
- Visibility will be discussed later
- Examples
  - dateOfBirth
  - seconds : 0..59
  - name : String
  - quantity : int = 0

Justin Templemore - ECE - Object-oriented system design with UML - 2005/2006

Attributes and associations

- When to use attributes and when to use associations?



Attribute	*Association*
Not a distinct entity in the problem domain; simply an object qualifier.	Related, but distinct entities. The identity of each entity is important.
Value semantics	Reference semantics
No multiplicity information, cannot show an optional attribute	Multiplicities 1..5, 0..1, *, etc.
Navigable from class to attribute only.	May be bi- or uni-directional
Cannot be shared between objects	Multiple associations may point to the same object.

Operations

- An operation defines a service offered and carried out by an object.
  - SavingsAccount offers the operation creditInterest()
- Abstractly, operations describe the responsabilities of a class.
- But more obviously, operations correspond to the programmatic methods defined in a class.
- An operation queries / modifies the attributes of its instance, or invokes operations of associated objects.

Operation syntax

`visibility name (parameter-list) : return-type`

- The amount of detail shown depends on the phase of the project.
- Visibility will be discussed later
- Examples
  - withdrawal (amount : double) : boolean
  - transplantHeart (newHeart : Heart) : void
    - Passage of a reference parameter allowing future collaboration between the operation's instance and the referenced Heart instance
  - getName () : String … and … setName (name : String)
    - *Getters* and *Setters*, often not shown because implicit.

Visibility

- Visibility is used to specify encapsulation
  - Specify the *public interface* and *hidden implementation*
- UML defines three levels of visibility :
  - **+ (public)** : feature may be read or set by any object (interface)
  - **- (private)** : feature may only be read or set only by the owner object
  - **# (protected)** : feature may be read or set by the owner object or instances of sub-types of that object's type.

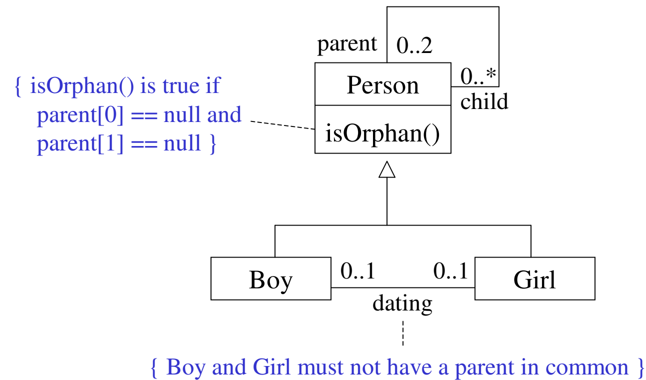
Visibility

- As a general rule, attributes are private, and operations are public.
  - Ensures that an object has total control over its own data,
  - and that other objects depend only on its interface.
- Public attributes implicitly have getters and setters.
- Private operations are "helper" operations used by the object for procedural decomposition.
- *Note that most programming languages implement visibility differently! Importance grows as design becomes more concrete.*

Constraint rules

- A class diagram is all about constraints.
- But the basic structures of association, attribute and generalisation cannot specify every constraint, particularly functional integrity constraints.
- You can add additional constraints to a class diagram by placing expressions between accolades
  - { your constraint here }
- A constraint may be expressed in any language : Informal english / french, or more formal, like the OCL (Object Constraint Language).

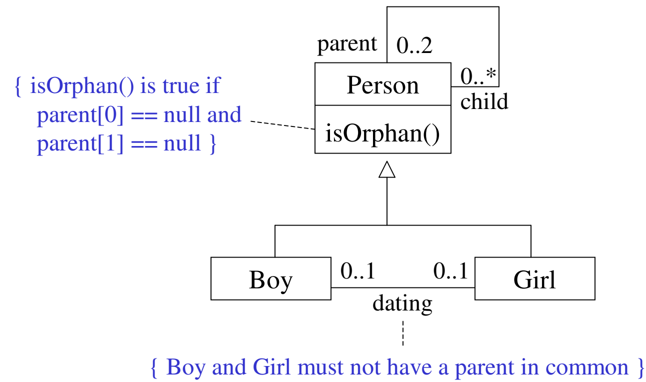
Constraint example



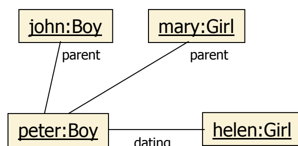
Object diagram

- An object diagram shows a snapshot of the objects and connections between them at a point in time.
- Used to test and demonstrate complex data structures
  - Find a general class structure from specific object structures
  - Test that the class structure generates all legal structures, but not illegal ones.
- Objects are shown as rectangles in which the name takes the form **instanceName:className**, and is underlined.
  - Example : peter:Person

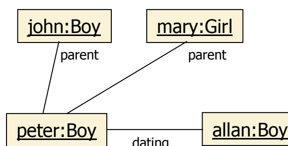
Object diagram example



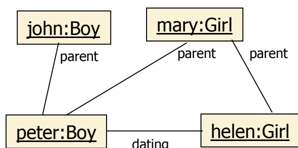
Object diagram example



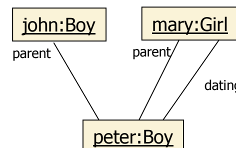
Structure allowed: Peter is dating an orphan named Helen.



Structure prohibited by Boy-Girl association "dating".



Structure prohibited by constraint : A Dating pair cannot share a parent



Structure allowed by current model, but probably unwanted; **requires new model!**

Method

Modeling techniques and advice

Modeling techniques and advice

- Class diagrams contain enormous detail. Do not get trapped into adding too much too early.
  - Ask yourself what stage you are at : analysis? conception? implementation? and limit detail accordingly.
- Try to use the vocabulary of the problem domain and avoid implementation jargon.
  - Invent as little as possible in order to avoid confusion.
- Don't feel forced to use all the possible notations.
  - Keep it simple and use only what you need.
  - For most problems, the essential modeling concepts are sufficient.
- Remember that the class diagram is not a database.
  - A class is not just data, but responsability.

Potential classes

- Real-world items
  - Entities that exist in the real world. Not necessarily electronic - *house, satellite*
- Physical devices
  - Sensors, actuators and the electronic devices they monitor or control - *robot arm, weight sensor*
- Transaction objects
  - Temporary objects which monitor/control some long-lasting action from start to end - *web shopping cart, ATM transaction*
- Clients, providers and requests
  - Messages exchanged by two objects are also objects.
  - A message must have a source object and a destination object, and probably another object that processes them.

Potential classes

- Causal objects
  - Objects which generate messages or information of importance to the system - *weight sensor, display refresh loop*
- Key concepts
  - Key concepts of the problem domain may be modelled as objects - *bank account and client for a bank*
- Transient and persistent objects
  - Certain data exists in live memory - login session
  - Other data must exist beyond the power cycling of the device - *login history*
- Visual elements
  - User interface elements that display data - *windows, buttons*
- Control / interface elements
  - Objects (software or hardware) that provide the interface for the user - *button*

Potential attributes

- Attributes qualify an object
  - they have a primitive data structure and no responsibility or operations
  - as opposed to a "part" object (shown by aggregation).
- They are the "data" portion of an information object.
- To identify an object's attributes, ask the following questions:
  - What information defines the object ?
  - What information does the object's operations modify?
  - What does the object know?
  - What is the responsibility of the object? What data does it need to store to fulfil this responsibility?

Analysis techniques

- Red-line analysis
  - A reading of the system text or use cases in which
  - nouns or the hypothetical results of actions become objects
  - adjectives become object attributes
  - natural relationships between objects imply association, aggregation or generalisation
  - (verbs will be used later to identify object behaviour)
- Walkthrough scenarios
  - Modeling of a use case scenario as a collaboration of objects
  - reveals missing model elements by the fact that the collaboration is not currently possible.