Lecture 2: Introduction to Modeling

- Why Build Models?
- What types of Models to build
- Intro to UML
- Class Diagrams
- Reverse Engineering…

Getting started

- You've just joined an ongoing project
  - Where do you start?
  - (oh, BTW, the project doesn't really have any documentation)
- Reverse Engineering:
  - Recover design information from the code
  - Create higher level views to improve understanding
- E.g. Structure of the code
  - Code Dependencies
  - Components and couplings
- E.g. Behaviour of the code
  - Execution traces
  - State machines models of complex objects
- E.g. Function of the code
  - What functions does it provide to the user?

Why build models?

- **Modelling can guide your exploration:**
  - It can help you figure out what questions to ask
  - It can help to reveal key design decisions
- **Modelling can help to uncover problems**
  - Inconsistency in the models can reveal interesting things…
    - e.g. conflicting or infeasible requirements
    - e.g. confusion over terminology, scope, etc
    - e.g. disagreements between stakeholders
- **Modelling can help us check our understanding**
  - Reason over the model to understand its consequences
    - Does it have the properties we expect?
  - Animate the model to help us visualize/validate the requirements
- **Modelling can help us communicate**
  - Provides useful abstracts that focus on the point you want to make
  - …without overwhelming people with detail

Dealing with problem complexity

- **Abstraction**
  - Ignore detail to see the big picture
  - Treat objects as the same by ignoring certain differences
  - (beware: every abstraction involves choice over what is important)
- **Decomposition**
  - Partition a problem into independent pieces, to study separately
  - (beware: the parts are rarely independent really)
- **Projection**
  - Separate different concerns (views) and describe them separately
  - Different from decomposition as it does not partition the problem space
  - (beware: different views will be inconsistent most of the time)
- **Modularization**
  - Choose structures that are stable over time, to localize change
  - (beware: any structure will make some changes easier and others harder)

the Unified Modelling Language (UML)

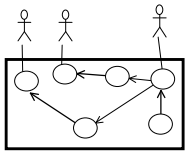
→ Third generation OO method

- Booch, Rumbaugh & Jacobson are principal authors
  - Still evolving (currently version 2.0)
  - Attempt to standardize the proliferation of OO variants
- Is purely a notation
  - No modelling method associated with it!
  - Was intended as a design notation
- Has become an industry standard
  - But is primarily promoted by IBM/Rational (who sell lots of UML tools, services)

→ Has a standardized meta-model

- Use case diagrams
- Class diagrams
- Message sequence charts
- Activity diagrams
- State Diagrams
- Module Diagrams
- Platform diagrams
- …

UML views

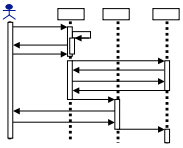


Use Cases

user's view;

Lists functions;

visual overview of the main requirements;

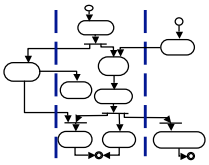


Sequence Diagrams

individual scenario;

interactions between users and system;

Sequencing of messages;

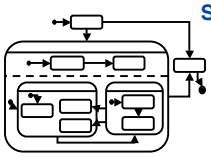


Activity diagrams

business processes;

concurrency and synchronization;

dependencies between tasks;

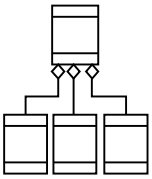


Statecharts

responses to events;

dynamic behavior;

event ordering, reachability, deadlock, etc.



Class Diagrams

information structure;

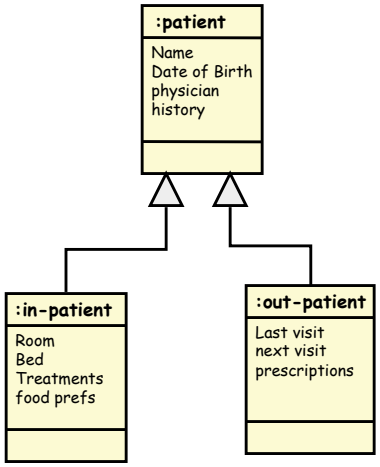
relationships between data items;

modular structure for the system;

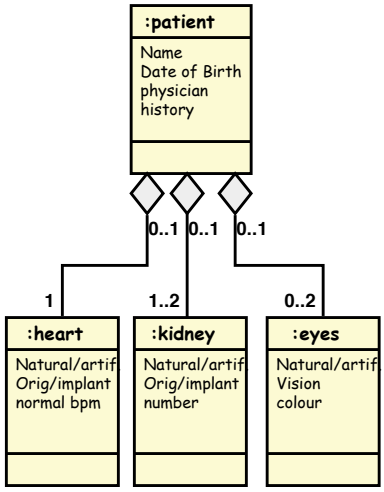
Intro: Object Classes in UML

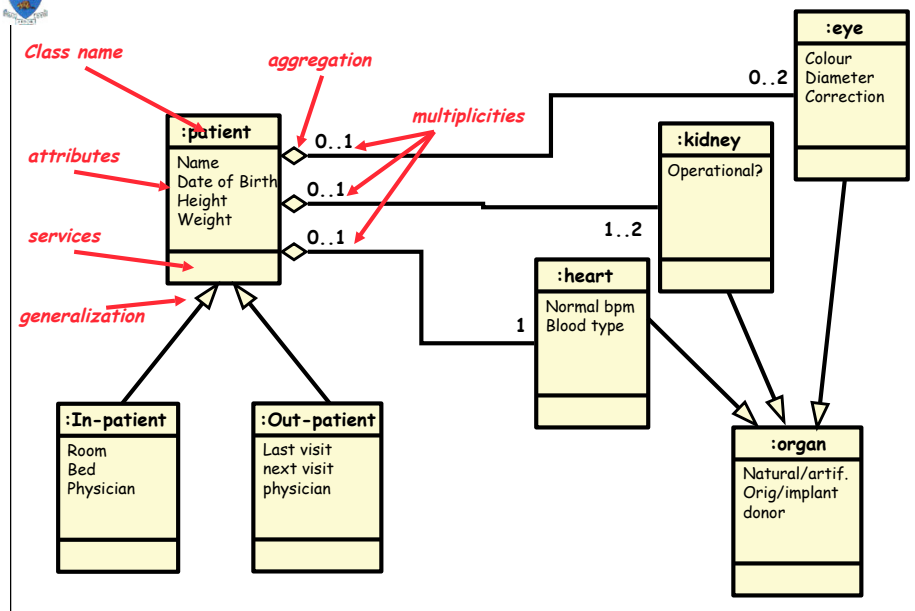
Source: Adapted from Davis, 1990, p67-68

Generalization (an abstraction hierarchy)



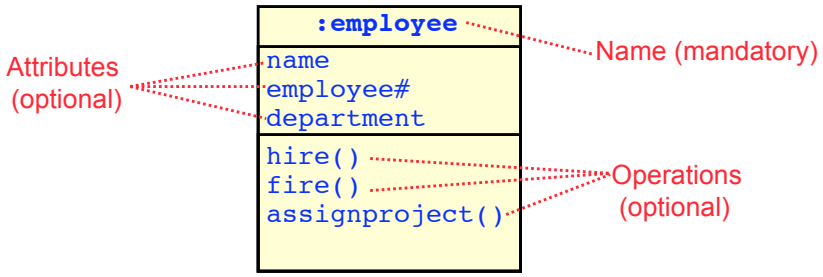
Aggregation (a partitioning hierarchy)





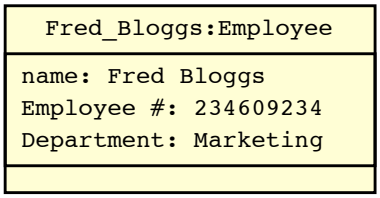
What are classes?

- A class describes a group of objects with
  - similar properties (attributes),
  - common behaviour (operations),
  - common relationships to other objects,
  - and common meaning ("semantics").
- Examples
  - employee: has a name, employee# and department; an employee is hired, and fired; an employee works in one or more projects



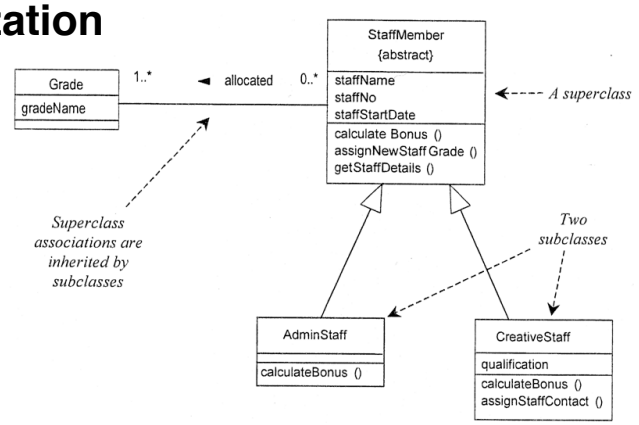
Objects vs. Classes

- The instances of a class are called objects.
  - Objects are represented as:



  - Two different objects may have identical attribute values (like two people with identical name and address)
- Objects have associations with other objects
  - E.g. Fred_Bloggs:employee is associated with the KillerApp:project object
  - But we will capture these relationships at the class level (why?)
  - Note: Make sure attributes are associated with the right class
    - E.g. you don't want both managerName and manager# as attributes of Project! (…Why??)

Generalization



→ Notes:

- Subclasses inherit attributes, associations, & operations from the superclass
- A subclass may override an inherited aspect
  - e.g. AdminStaff & CreativeStaff have different methods for calculating bonuses
- Superclasses may be declared {abstract}, meaning they have no instances
  - Implies that the subclasses cover all possibilities
  - e.g. there are no other staff than AdminStaff and CreativeStaff

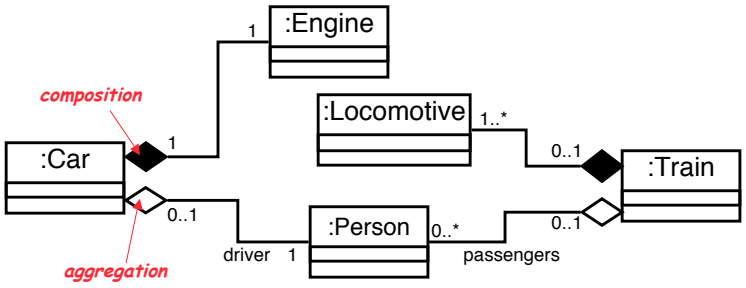
Aggregation and Composition

→ Aggregation

- This is the "Has-a" or "Whole/part" relationship

→ Composition

- Strong form of aggregation that implies ownership:
  - if the whole is removed from the model, so is the part.
  - the whole is responsible for the disposition of its parts

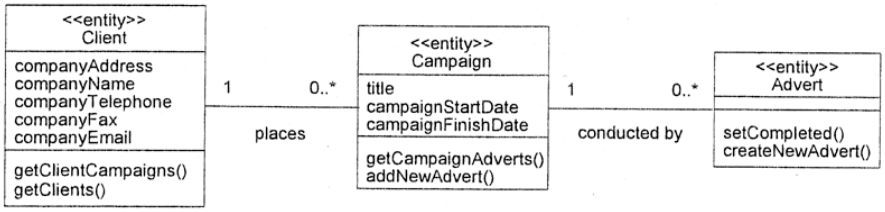


Associations

→ Objects do not exist in isolation from one another

- A relationship represents a connection among things.
- In UML, there are different types of relationships:
  - Association
  - Aggregation and Composition
  - Generalization
  - Dependency
  - Realization

→ Class diagrams show classes and their relationships



Association Multiplicity

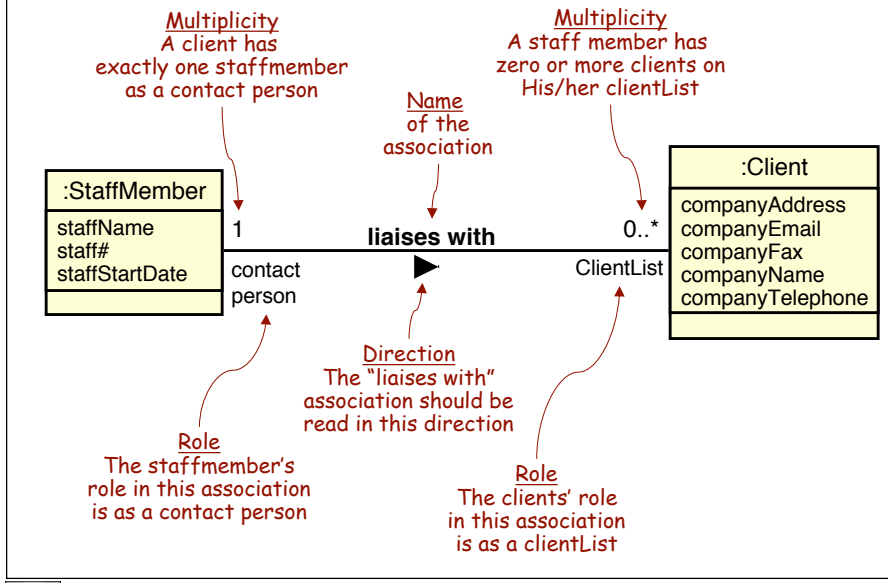
→ Ask questions about the associations:

- Can a campaign exist without a member of staff to manage it?
  - If yes, then the association is optional at the Staff end - zero or more (0..*)
  - If no, then it is not optional - one or more (1..*)
  - If it must be managed by one and only one member of staff - exactly one (1)
- What about the other end of the association?
  - Does every member of staff have to manage exactly one campaign?
  - No. So the correct multiplicity is zero or more.

→ Some examples of specifying multiplicity:

- Optional (0 or 1) 0..1
- Exactly one 1 = 1..1
- Zero or more 0..* = *
- One or more 1..*
- A range of values 2..6

Class associations



More Examples