software modeling

modeling

- one thing that we as software engineers can do to better understand software is by using models
- many choices when building models
  - multiple modeling "languages"
  - graphical/textual
  - diagrams – ER diagrams for data, class and object diagrams in OOP.
  - ad-hoc
- in this course we'll study some UML

modeling (2)

- uml as defined by wikipedia:

  "UML is a standardized general-purpose modeling language in the field of object-oriented software engineering. The UML includes a set of graphic notation techniques to create visual models of object-oriented software-intensive systems."

- caveat: how often do I use (strict) uml?

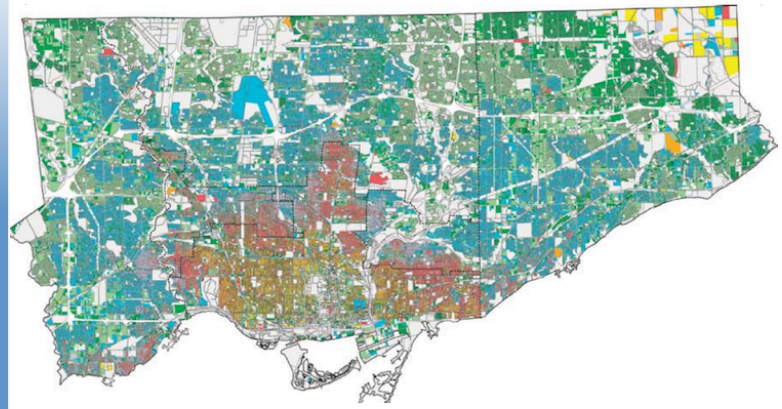
  "…in his eighteen years as a professional programmer, Wilson had only ever worked with one programmer who actually used it voluntarily ." – Two Solitudes Illustrated, Greg Wilson & Jorge Aranda, 2012

- regardless, software models are very useful

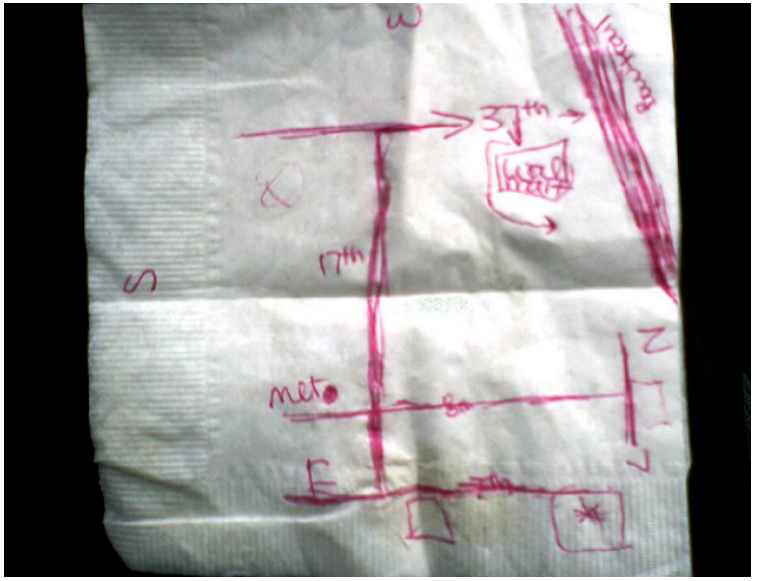
why build models?

- modeling can guide your exploration:
  - can help figure out what questions to ask
  - can help reveal key design decisions
  - can help uncover problems
- modeling can help us check our understanding:
  - reason about the model to understand its consequences
    - does it have the properties we expect?
  - animate the model to help visualize software behavior
- modeling can help us communicate:
  - provides useful abstractions that focus o the point you want to make…
  - …without overwhelming people with detail
- throw-away modeling
  - making the model is more important than the model itself
  - time spent perfecting models is probably time wasted

maps as abstractions



maps as abstractions (2)



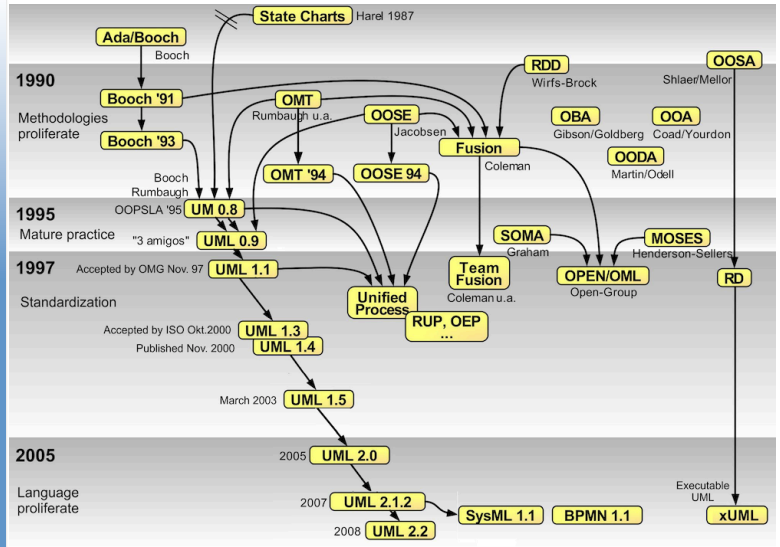
dealing with problem complexity

- abstraction
  - ignore detail to see 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 – does not partition 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 makes some changes easier & others harder)

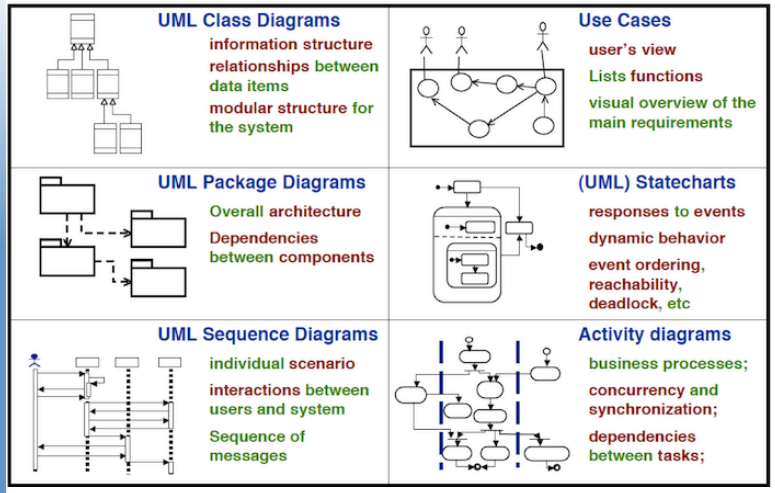
unified modeling language

- third generation OO method
  - Booch, Rumbaugh & Jacobsen are principal authors
    - still evolving (maybe) – version 2.0
    - attempt to standardize proliferation of variants
  - purely a notation
    - no modeling method associated with it
    - intended as design notation
  - has become (more or less) and industry standard
    - primarily promoted by IBM/Rational (who sell lots of UML tools/services)
- Has a standardized meta-model
  - use case diagrams, class diagrams, sequence diagrams, state chart diagrams, activity diagrams, component diagrams, package diagrams, deployment diagrams,…

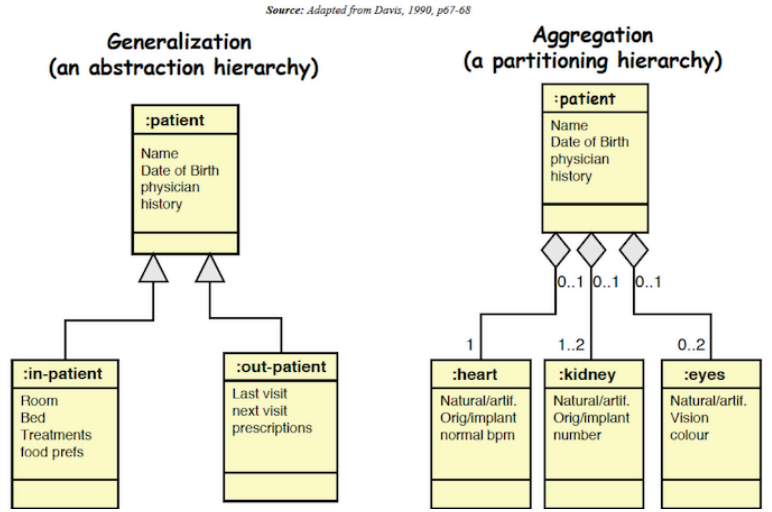
unified modeling language (2)



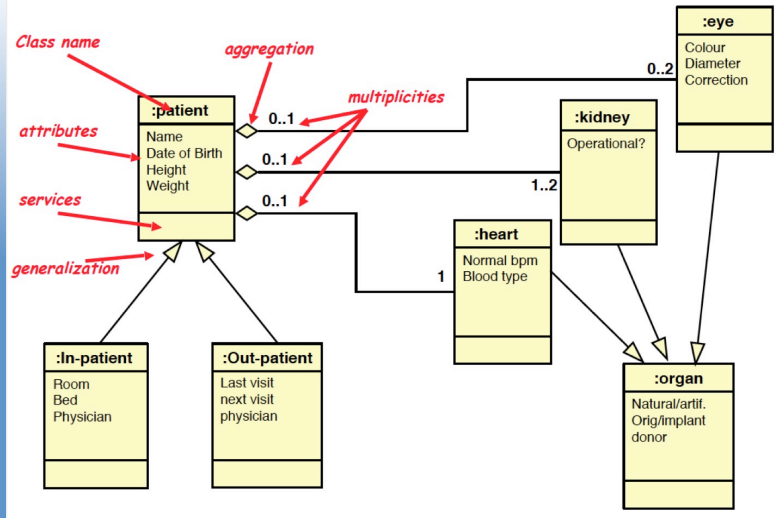
modeling notations



object classes in uml

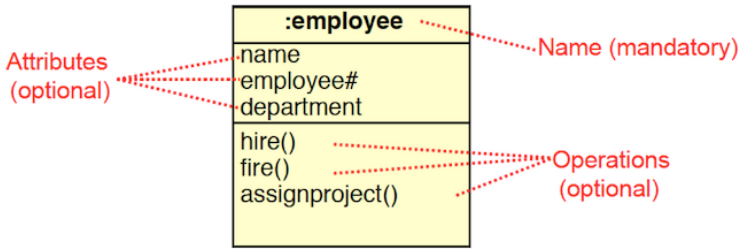


parts of a diagram



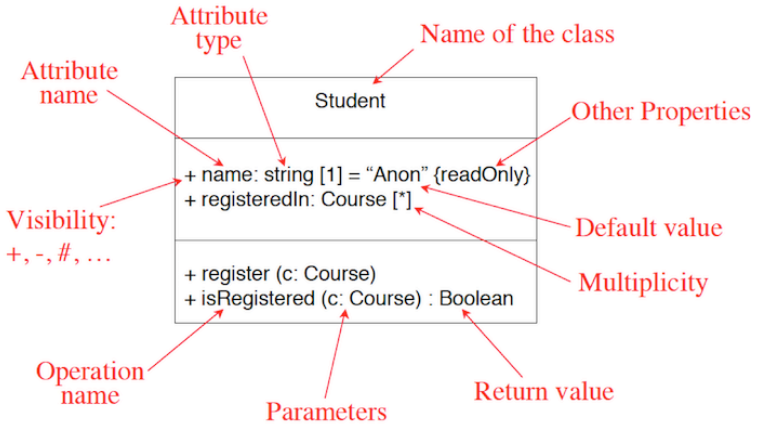
what are classes?

- a class describes a group of objects with
  - similar properties (attributes)
  - common behavior (operations)
  - common relation
  - and common meaning (semantics)
- example
  - employee: has a name, employee number and department; and employee is hired and fired (not very nice!); can work on one or more projects



The Edward S. Rogers Sr. Department
of Electrical & Computer Engineering
UNIVERSITY OF TORONTO

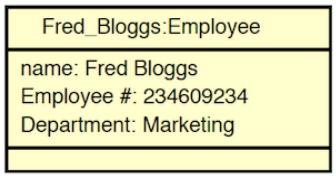
full class notation



The Edward S. Rogers Sr. Department
of Electrical & Computer Engineering
UNIVERSITY OF TORONTO

objects vs. classes

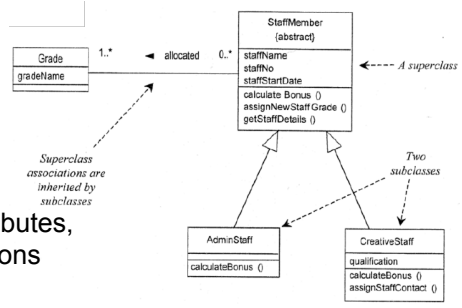
- instances of a class are called objects
  - objects are represented as:



  - two different objects may have identical attribute values (like two people with the same name and address)
- Objects have associations with other objects
  - ex. `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
    - ex. don't want `managerName` and `employee#` as attributes of a project (why?)

The Edward S. Rogers Sr. Department
of Electrical & Computer Engineering
UNIVERSITY OF TORONTO

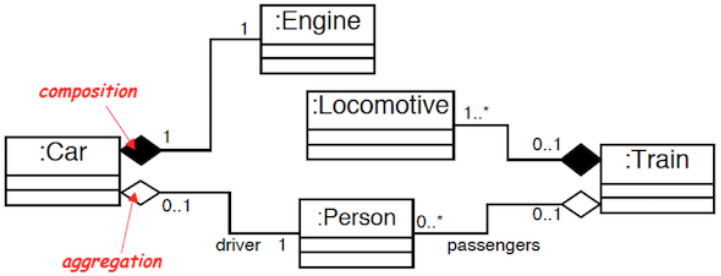
generalization



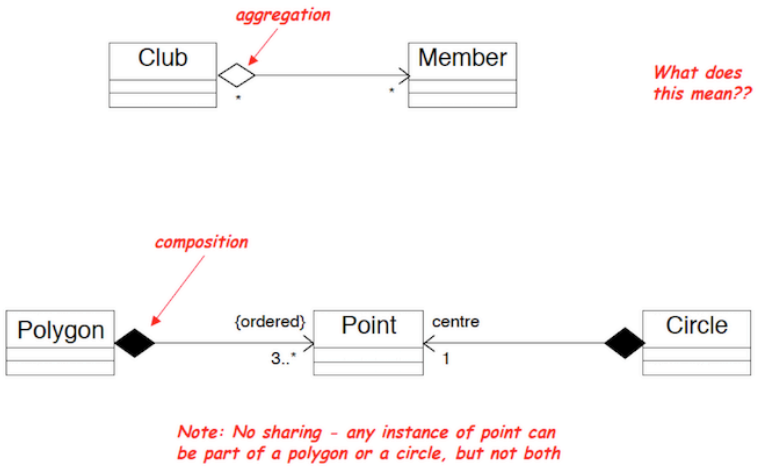
- subclasses inherit attributes, associations & operations from the superclass
- a subclass may override an inherited aspect
  - ex. `AdminStaff` & `CreativeStaff` have different methods for calculating bonuses
- superclasses may be declared `{abstract}`, meaning they have no instances
  - implies the subclasses cover all possibilities
  - ex. there are no other staff than `AdminStaff` and `CreativeStaff`

aggregation & 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

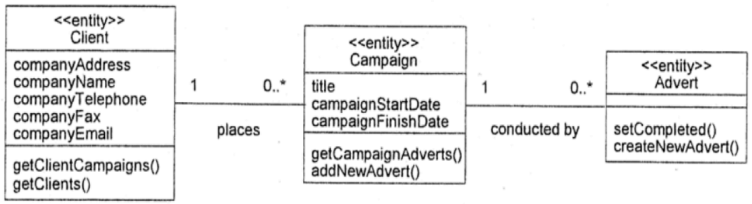


aggregation & composition (2)



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 & composition, generalization, dependency, realization
- class diagrams shoe classes and their relationships

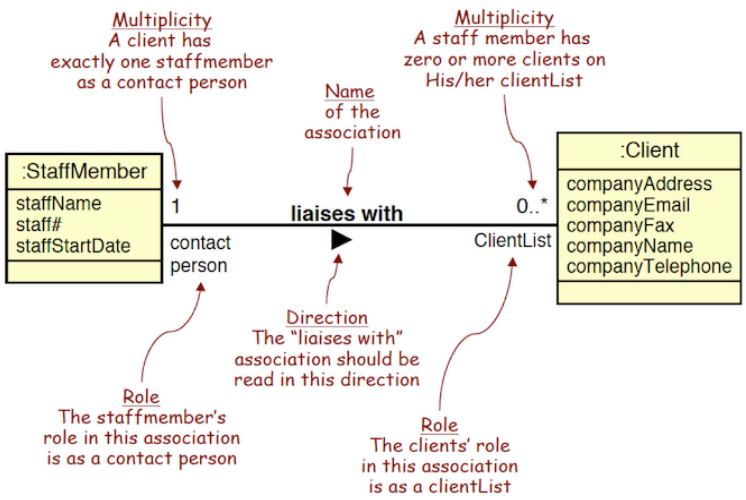


association multiplicity

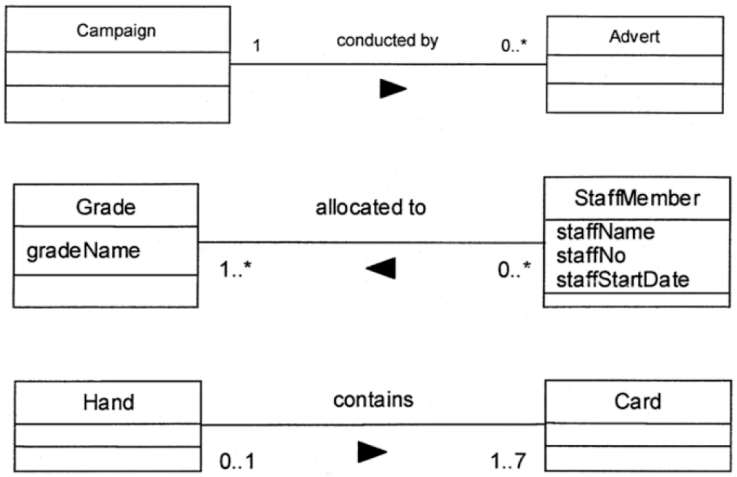
- ask questions about the associations:
  - can a campaign exist without a member of staff to manage it?
    - if yes, the association is optional at the staff end – zero or more (0..*)
    - if no, then it is not optional – one or more (1..*)
    - if it must be managed by one, and only one, member 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 0..*
- some examples:

optional	0..1	exactly one	1 (or 1..1)
zero or more	0..* (or just *)	one or more	1..*
range	2..6		

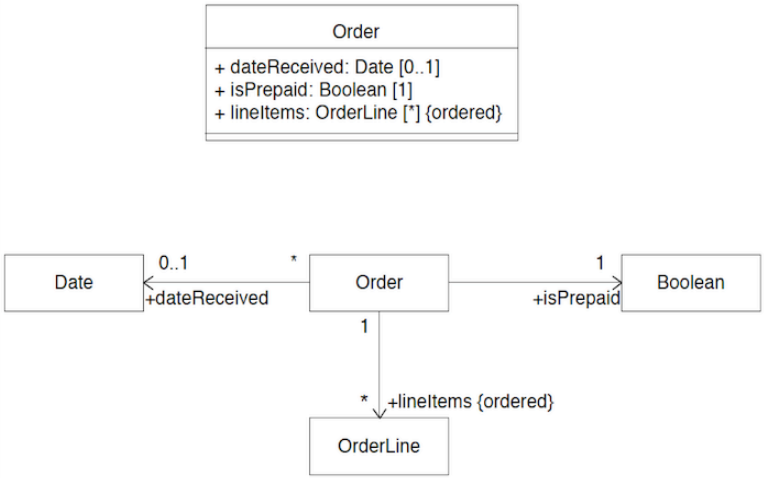
class associations



examples



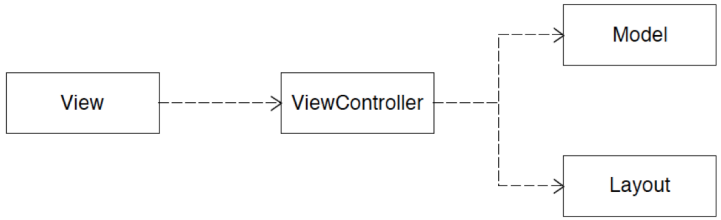
navigability / visibility



bidirectional associations

hard to implement correctly

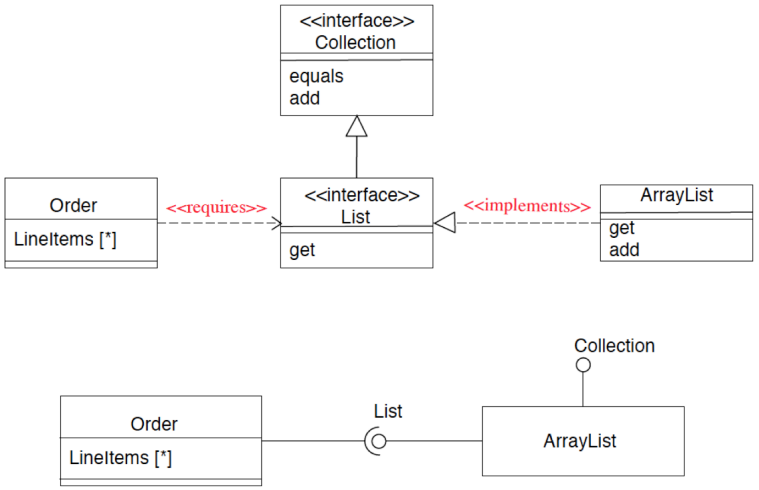
dependencies



- Examples

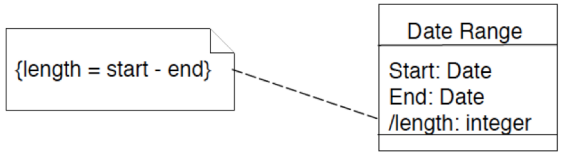
```
<<call>>          <<derive>>        <<refine>>
<<use>>           <<instantiate>>   <<substitute>>
<<create>>        <<permit>>        <<parameter>>
                  <<realize>>
```


interfaces



annotations

- comments
  - -- used to add comments within class description
- notes



- constraint rules
  - any further constraints {in curly braces}
  - ex. {time limit: length must be less than 3 months}

what class diagrams can show

- division of responsibility
  - operations that objects are responsible for providing
- subclassing
  - inheritance, generalization
- navigability / visibility
  - when objects need to know about other objects to call their operations
- aggregation / composition
  - when objects are part of other objects
- dependencies
  - when changing the design of a class will affect other classes
- interfaces
  - used to reduce coupling between objects

static vs. dynamic modeling

- static captures fixed, code-level, relationships
  - class (and package) diagrams
  - object diagrams
  - component diagrams
  - deployment diagrams
- behavioral diagrams capture dynamic, execution time, relationships
  - use case diagrams
  - sequence and interaction diagrams
  - collaboration diagrams
  - statechart diagrams
  - activity diagrams

The Edward S. Rogers Sr. Department of Electrical & Computer Engineering
UNIVERSITY OF TORONTO

summary

- summary on modeling
  - important to use modeling during design
  - modeling can be helpful to discover design and architecture (a1)
  - as with most things, it can be taken too far
  - the model should provide an easier to consume abstraction
  - strict uml is good when publishing designs for external consumption even if you don't use it yourself