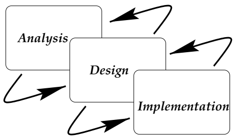
Lecture 1

We look at a simple software development process, one that might be suitable for small projects.



In software development, we build models: models of the problems domain and models of the software system. The Unified Modeling Language, UML, is a hugely popular graphical notation for expressing these models. We look in particular at Class Diagrams (which are for modeling static, structural aspects of a system) and Sequence Diagrams (which are for modeling dynamic, behavioural aspects of a system).

Software Development Process

- A *software development process* (or method or methodology) specifies who is doing what when and how to transform the stakeholders' requirements into a software system
- We describe a lightweight, low-ceremony process, suitable for small projects
- Key to communication between people involved in a process is a *modeling language* — a usually mostly graphical notation for expressing, e.g., designs
- We use UML, the Unified Modeling Language: *"a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system."*
- UML has 9 types of diagram but we look at only 2

- *Analysis:*
  - Establish business rationale, scope & purpose
  - Build a model of the problem domain: describes the world, not the software; use Class Diagrams and Sequence Diagrams
  - Identify *use cases*
- *Design:*
  - Decide on the software classes & interfaces that will do the work; use Class Diagrams, Sequence Diagrams & GUI mock-ups
- *Implementation:*
  - write code for each class and do unit testing (lecture 16), integration testing, demos, beta testing, performance tuning, user training, etc.

Analysis: Problem domain class identification

- "Underline the nouns in your requirements document"
  - easy for teaching; irrelevant in practice
- Problem domain classes will be revealed
  - in discussion with domain experts (recurring terms, terms they specially define)
  - by seeing physical manifestations (seeing customers, order forms, etc.)
  - from existing software systems (the contents of files & data structures)
  - from refining your diagrams, e.g. classes sometimes lurk behind relationships (especially many-to-many ones).

Analysis: Use case identification

- Use cases are typical interactions between users and the system that achieve user goals
- Goals include:
  - set-up
  - conducting the business
  - reporting business results
- Completeness at this stage is impossible; change is inevitable; good software development processes anticipate such change
- Use cases specify *functional requirements* (desired system functionality); it might be possible to attach *non-functional requirements* to uses cases (performance/ reliability requirements)

Design

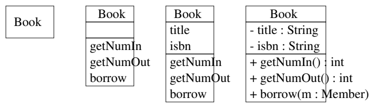
- Draw a Class Diagram, showing software classes
  - Problem domain classes might have corresponding software classes
  - Boundary classes that mediate between the software & its users/other external systems
  - Control classes that coördinate workflow between different objects (e.g. listeners)
  - Interfaces to 'hide' classes in which change is anticipated or to 'cut across' class hierarchies
  - Include navigability arrows on associations based on consideration of efficiency requirements
- Draw Sequence Diagrams for each use case
- Draw GUI mock-ups

Software class identification

- A software class is an abstraction around data
  - *state* (attributes/instance variables) and *behaviour* (operations/instance methods)
- Identify candidate classes from design patterns, code libraries (reuse!), data structures courses, experience & discussion with other programmers
- Candidate classes that aren't data abstractions should come under suspicion, e.g.:
  - Classes whose names are actions (ProcessX, ReadX), unless they're control classes
  - Classes with attributes only/operations only
- But, before throwing classes away, consider whether they might become useful as the system evolves

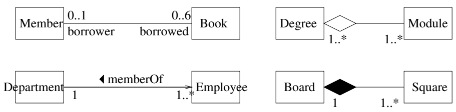
Class diagrams

- Class diagrams show classes and their relationships
- They model parts of the *structure* of your problem-domain or your software system
- Classes: name, attributes (variables), operations (methods)
- Do not have to show all or any attributes/operations; an empty compartment does not mean 'none'
- Can specify attribute type and visibility; can specify operation parameters, return type and visibility



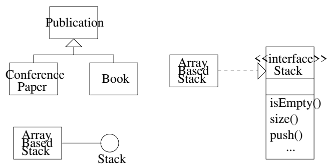
Relationships: Association

- Association: objects of class A are connected to objects of class B
  Draw as a line and optionally adorn with:
  - association name, role names & multiplicities
  - aggregation: draw a diamond to associate a whole with its parts
  - composition: draw a filled diamond to show aggregation where the part dies with the whole
  - navigability arrows to show desired efficiency of traversal



Relationships: Generalisation & 'implements'

- Generalisation: every B is a special kind of A
  The arrow points from subclass to superclass; use italics for abstract classes & abstract methods
- Interfaces and the 'implements' relationship
  Draw interfaces simply as circles, or like classes (when you want to show the operations)



- Class diagrams can also include notes (for constraints, pseudocode, etc.), packages & objects

Sequence diagrams

- Across the top, boxes denoting objects or stick figures representing the user. Beneath each box, a lifeline (dashed line). Blocks on lifelines represent a method executing; these can be nested
- Messages are represented by lines with arrows; method returns can optionally be shown by dashed lines with arrows
  - Can show conditionals or iteration, but keep this simple
  - Object creation messages point to the new object, and can optionally be labelled ⟨⟨create⟩⟩
  - Object deletion is shown by a 'X' on a lifeline; an object deletion message can optionally be labelled ⟨⟨destroy⟩⟩
  - Other options, e.g. timing constraints

Example Sequence Diagram