COMP6700/2140 Software Creation and Quality

Alexei B Khorev and Josh Milthorpe

Research School of Computer Science, ANU

May 2017

Topics

- Software Development Cycle
- Software Quality
- Elements of Software Design
- Notations for Software Design: UML
- Agile Development, *DevOps* etc

Software Development Cycle

A software project normally begins because a client (or a computer hacker, or whoever) has a problem to solve and money (or time) to spend. If not abandoned midway, the software undergoes the following life cycle:

1. Analysis
2. Design
3. Implementation
4. Integration and Testing (V&V, performance, usability etc)
5. Deployment
6. Maintenance/service/support

In the *analysis phase*, the requirements of your software are defined. You try to understand the problem that the customer has given you in as much detail as you can. Many large software projects will not have a single clearly defined problem to solve but will be required to exhibit certain behaviour for particular use cases. The output of this phase is a *software requirements specification* (SRS) document written in compliance with the industry standards (*eg*, IEEE Std 830-1998).

The most important part of the requirements are *functional requirements*, which specify the intended behaviour of the system. Analysis should not focus on *how* the program will satisfy the requirements, but it can define performance criteria such as speed and memory requirements (they represent *non-functional* requirements — performance constraints, availability, accessibility etc).

Software Design

Design is concerned with a plan about how you might implement your software system. Design phase begins with formulating the so called *Use Cases*, which is a technique for capturing functional requirements of systems. "Use cases allow description of sequences of events that, taken together, lead to a system doing something useful" (Bittner and Spence). Each use case provides one or more *scenarios* that convey how the system should interact with the users called *actors* to achieve a specific business goal or function.

The use case analysis allows the designers to establish and define (if they use OO approach) the system classes, their relationships and their data attributes and their methods (which must be described in terms of their contact and the most essential elements of implementation). The output of this phase is the *Class Diagrams*, which constitute the *static* structure of the system under development.

The *dynamic* structure is captured in the *State Machine Diagrams*. The State Machine is an abstraction of object's *life cycle*, akin to a class being an abstraction to all its instances. There are other useful diagrammatic techniques for representing the *behaviour* of objects, class collaborations and so on). Most often used are so called *Sequence and Collaboration Diagrams*. These are complimentary ways to realise the use cases using the structural and behavioural characteristics of the system, and thus to test the design against the functional requirements.

The would be informal (somewhat artistic rather then scientific or engineering) approach to design software can be made more precise and better defined (at least for communication) via the use of *design notations*. The *de facto* industry standard for the design notations is represented by the *Unified Modelling Language (UML)*.

Software Development Methodologies, 1

In the *implementation* phase you write and compile program code. The output of this phase is the completed program. The unit testing is to be done during this phase!

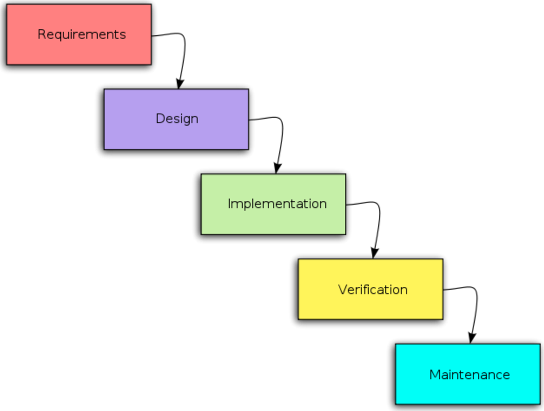
In the *integration and testing* phase, you run integration tests to verify that the program as whole works correctly and that it satisfies all of the requirements. The output of this phase is a report describing the tests that you carried out and their results.

In the *deployment* phase, the program gets installed and used in its target environment. (Some deployment characteristics can be also formalised in the design, the UML has a specialised set of notation for this, too.)

The nature of a software project is that it can get out of control very easily. Because of this, software engineers put much effort into thinking about managing *process*. When formal software processes were first introduced in the 1970s, engineers had a very simple model of these phases which they called the *waterfall model*. The output of one phase was meant to spill into the next like water falling down a set of pools as in the figure.

The waterfall model is is too **rigid**: It is very difficult to immediately come up with a perfect requirements specification (partly because customers do not actually know what they really want!). It was quite common to discover in the design phase that the requirements were inconsistent or that a small change in the requirements would result in a much better system. But the analysis phase was over, so the engineers had to build the inferior system with its errors and all! When the design was actually implemented, it was often found that the design was not perfect and so on.

Waterfall Model



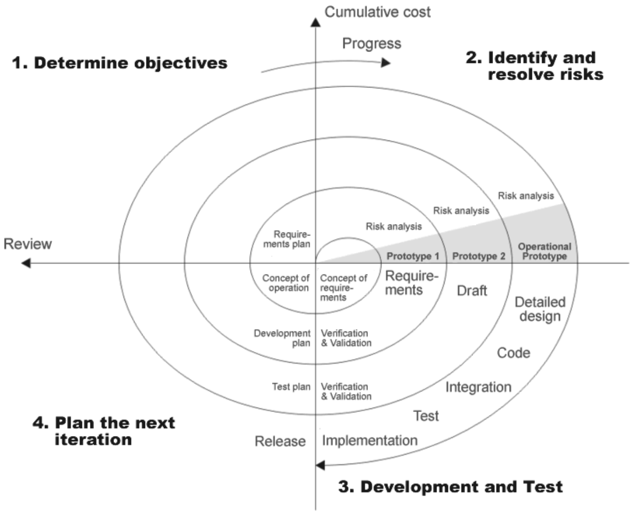
Software Development Methodologies, 2

One way to remedy the waterfall model deficiencies is the *iterative waterfall model* ("waterfall with up-flow"). This model is taught to our software engineers in their project management course. Students are encouraged to translate these phases into a timeline for their specific project.

Yet another improvement is the *spiral model* (introduced by Barry Boehm). The early phases of this model focus on the construction of prototypes. These are small systems which illustrate the behaviour of one part of the larger project. For example, a GUI prototype could be built to show the customer how it might look (without implementing any functionality). Other prototypes could be built to test out complicated algorithms, to test out integration with external systems, or to profile system performance.

One of the dangers of the spiral model is that an excessive reliance on prototypes may cause engineers to be too relaxed about actually delivering on the overall system requirements. Another alternative is the "Rational Unified Process" (RUP) — see Grady Booch, James Rumbaugh and Ivar Jacobson, "The Unified Modeling Language User Guide'', 2nd Ed., Addison Wesley, 2005,) which was developed by the inventors of the Unified Modelling Language (UML). This has a complicated set of "activity levels" as shown in Figure 3, Chapter 16 of Horstmann. A related to RUP approach is the so called *Capability Maturity Model (CMM)* (Shayne Flint is an expert). Another quite trendy methodology is "extreme programming" (see Kent Beck, "Extreme Programming Explained", Addison-Wesley, 1999) which focuses on a set of programming practices rather than a formal process. The methodology envisions a constant interaction with a representative of the customer and regular releases of useful systems which complete part of the requirements.

Spiral Model



Product Software Quality

All software methodologies aim at achieving software productivity and software quality enhancement.

Software quality is a two-face Janus, its meaning is somewhat different for users and developers. The users (in broad sense, which may include developers too) are concerned with the *product quality*, while developers are affected by the *code quality*.

As **product**, software can be assessed against:

- conformance to requirements or program specification (validation, 1st ‘v’ of V&V)
- reliability
- usability
- correctness and completeness (verification, 2nd ‘v’ of V&V, “fit for purpose”)
- absence of bugs
- fault tolerance (very important for mission critical systems)
- extensibility and maintainability
- documentation

Code Software Quality

As **code**, software can be assessed against:

- readability (adherence to coding standard)
- ease of maintenance, testing, debugging, fixing, modification and portability
- low (lowest possible) complexity
- presence of the test code (test class companions for production classes)
- code self-documentation (like Doc comments)
- optimal resource consumption: memory, CPU, GPU, bandwidth, threads *etc*
- size of the (static) code analysis tool (`lint`, and modern OO tools, like *PMD, FindBugs, Jackpot*, *Commander* in Intellij IDEA) *output*

There are well developed techniques (supported by tools) to improve the code quality, *eg*, Refactoring (code modification without changing its external behaviour, the classical exposition in Martin Fowler's book *Refactoring*, and on his web site, link above).

What Software Design Involves

Set of rules to obey and steps to make

1. Discovering classes (tangible things, roles, incidents, interactions, specifications) — they must be abstractions!
2. Defining non-functional attributes of classes — abstractions of "atomic" (non-derivable) *structural* characteristics of a class (notional slots, value holders)
3. Defining operational attributes of classes — *behavioural* characteristics of a class (actions and functions); *derived* attribute values must be computed with functions for consistency
4. Relating classes (dependencies, associations, constraints)
5. Discovering behaviour of classes (object states) — not here, not now!
6. Communicating objects (state machines, signals, events) — not here, not now!
7. Verifying the design: sequence and collaboration diagrams

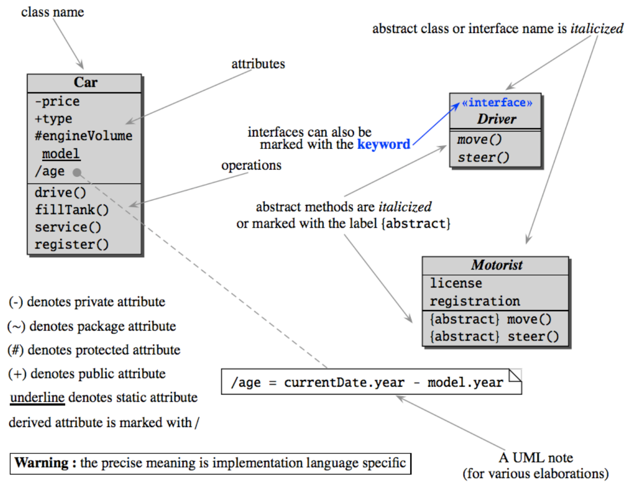
[all this if you are still "serious" about an OO-based approach, and want to use the UML and other old hats]

Unified Modelling Language

UML is a set of graphical notations (diagrams) backed by a single *meta-model* to describe and design software systems, especially SS constructed via Object Oriented technology. Actually, the UML means three different things:

- *A sketch* of a software system — for developing, reverse engineering and communicating the design, can be incomplete and crude.
- *A blueprint* — as normally practiced in the industry, often done with CASE tools, aims at to be comprehensive, but lacks precise semantics and needs humans to interpret and translate (implement) it into the working code.
- *A fully fledged programming language* — relies on precise semantics of all graphical elements (what a box representing UML class means, what does an arrow connecting two boxes mean etc — the same graphical elements may have quite different meaning when used in different UML diagrams, or even on the same UML diagram), needs a defined *meta-model* (the UML's meta-model is itself defined in terms of a UML subset); the product is a *UML model* (here it means a complete description of the system using UML diagrams and a constraint language, like *Object Constraint Language*) which requires compilers to translate it into the target code (there is lot more to this than just a code generation, take COMP3110 to learn more). *Some* of the industry greats (most notably, David L. Parnas) believe that precise definition of the design notations is unachievable ("UML are bad idea"), and urge the software designers not to delude themselves.

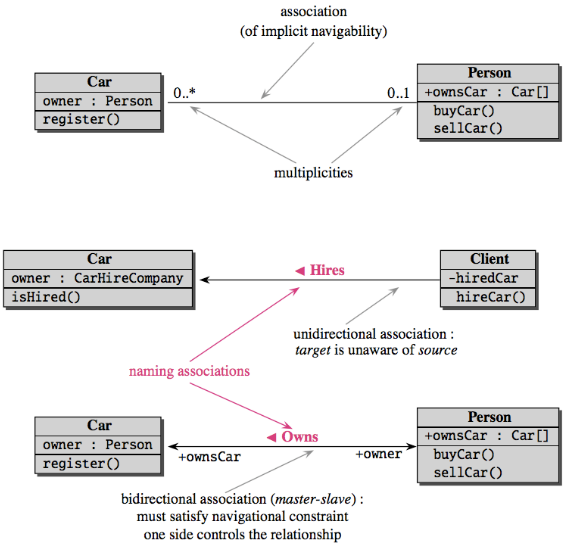
UML Class Diagrams



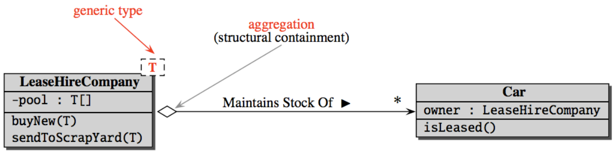
class name
attributes
abstract class or interface name is *italicized*
Car
-price
+type
#engineVolume
model
/age
drive()
fillTank()
service()
register()
«interface»
Driver
move()
steer()
interfaces can also be marked with the keyword
operations
abstract methods are *italicized* or marked with the label {abstract}
Motorist
license
registration
{abstract} move()
{abstract} steer()
(-) denotes private attribute
(~) denotes package attribute
(#) denotes protected attribute
(+) denotes public attribute
underline denotes static attribute
derived attribute is marked with /
/age = currentDate.year - model.year
A UML note (for various elaborations)

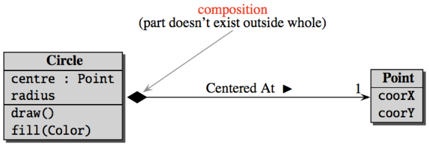
Warning : the precise meaning is implementation language specific

UML Class Associations



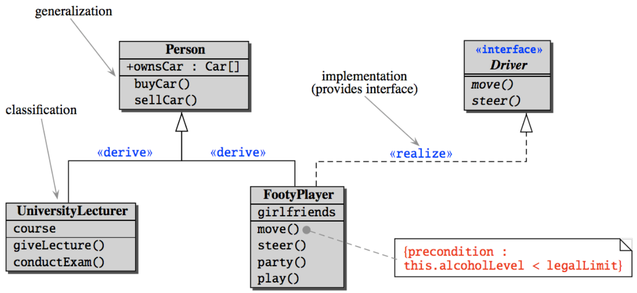
UML Aggregation and Composition





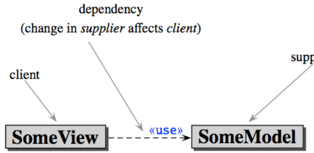
aggregation and composition are two versions of *part-whole* relationship, or structural containment ; aggregation's semantics is vague, and often association is used instead

UML Classification and Generalisation



Generalization means inheritance in OO implementation
Class on *classification* side specialises the functionality
of the class on the generalization side (*static classification*)

UML Dependencies and Constraints



Dependencies are *strictly* one-directional :
any change in the client has no influence on the supplier

Dependencies are marked by *keywords*
Some UML-2 *keywords* (*stereotypes* in UML-1) :

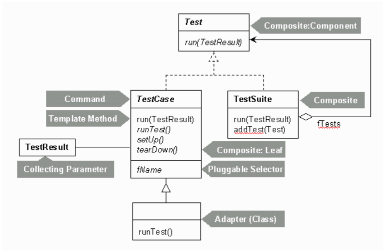
«call» — The client calls on operation in the supplier
«create» — The client creates instances of the supplier
«derive» — The client is derived from the supplier
«instantiate» — The client is an instance of the supplier (which is a metaclass)
«realize» — The source is an implementation of the interface specified in the supplier
«refine» — too tricky to comment
«substitute» — The source can replace the supplier
«use» — The source needs the supplier for its implementation

Anything (class, association, operation etc) can be a subject to constraint. The constraint can be shown inside a UML note within a pair of braces {..}. Constraints can be expressed in a natural language, pseudo-code or (better, yet) in Object Constraint Language. A simple example of an operation constraint (precondition) is shown on the previous slide "UML Classification and Generalisation" in red.

JUnit Class Design

A case study for UML use in the software design is represented by the JUnit framework.

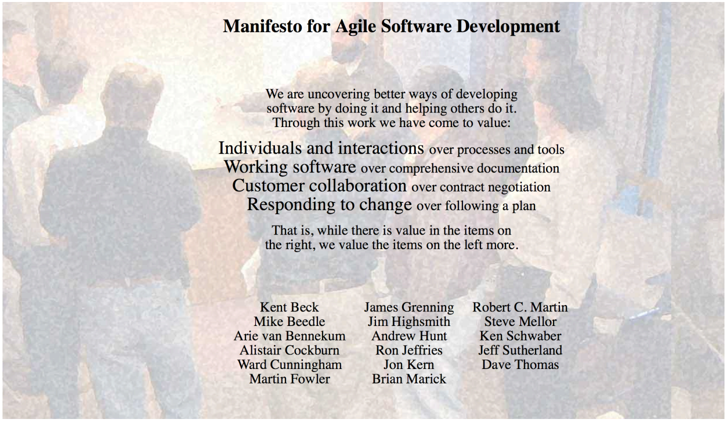
The small design for a small (yet so effective) system includes a number of classes and interfaces, various kinds of association, and *design patterns* which are codified recipes to solve recurrent design problems. The 1995 GoF catalogue offered 23 standard design patterns for the OO design of general-purpose systems. Earlier, we have encountered the *Observer* and *Visitor* design pattern.



The detailed design of JUnit (in the larger context of the *Eclipse* IDE) is discussed in the E. Gamma and K. Beck's book "Contributing to *Eclipse*. Principles, patterns, plugins".

Agile Development

Dissatisfied with software development practices, in 2001 a group of developers got together and formulated the rules of Agile Software Development, known as Agile Manifesto



Manifesto for Agile Software Development

We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on
the right, we value the items on the left more.

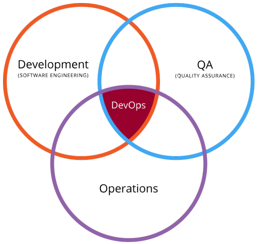
Kent Beck	James Grenning	Robert C. Martin
Mike Beedle	Jim Highsmith	Steve Mellor
Arie van Bennekum	Andrew Hunt	Ken Schwaber
Alistair Cockburn	Ron Jeffries	Jeff Sutherland
Ward Cunningham	Jon Kern	Dave Thomas
Martin Fowler	Brian Marick	

DevOps

DevOps [from **D**evelopment and IT **O**perations] is latest "big thing" in software development (and services). According to Wikipedia it is a

> *culture, movement or practice that emphasises the collaboration and communication of both software developers and other information-technology (IT) professionals while automating the process of software delivery and infrastructure changes. It aims at establishing a culture and environment where building, testing, and releasing software, can happen rapidly, frequently, and more reliably.*

The (absolutely) same principles are claimed to be the cornerstone of Agile Development and (what followed it) *Continuous Integration* (which is, given how generally and tool-agnostically the Agile principles are formulated, is a realisation of Agile Philosophy on an integrated platform of tools, like *Jenkins* for CI, *JUnit* and other testing frameworks like *Spock* for testing, *Sonar* for *Quality Assurance* and so on).



DevOps Toolchain

Understood simply as a set of practices supported by tools, DevOps is a more reasonable methodology. The processes (stages of life cycle) and associated tools are:

- Code — Code Development and Review, *continuous integration* tools:
  - Jenkins, Travis, *etc*
- Build — *Version control* tools, code merging, Build status
  - Git, Mercurial, Bazaar *etc*
  - Ant, Maven, Gradle *etc*
- Test — Test and results determine performance
  - JUnit Testing Framework
  - Spock
- Package — *Artifact repository*, Application pre-deployment staging
  - Maven
  - `npm` (for Javascript)
- Release — Change management, Release approvals, *release automation*
  - Sonar
- Configure — Infrastructure configuration and management, *Infrastructure as Code* tools
- Monitor — *Applications performance monitoring*, End user experience

Important aspects — virtualisation, containerisation, cloud-based applications

How to begin

Fear of coding and how to avoid it

- "Programming is an art of debugging a blank sheet of paper (or, in these days of on-line editing, the art of debugging an empty file)." (of unknown origin)
- "Start with the Most Difficult Part" (D. Spinellis), because:
  1. At the beginning, there is no design constraints, and hence we have maximum freedom to tackle the most difficult part. When working on easier parts at the end, the existing constrains are less restraining and give helpful guidance.
  2. Early shrinking of the project's cone of uncertainty (rapid reduction in the project's unknowns) — good for project management (decisions about budget, planning, staffing etc).
  3. Human nature: drive and enthusiasm are highest at the start, while at the end the common situation is burn-out, disillusionment, boredom.

  To understand this rather paradoxical advice, one should read the original article Diomidis Spinellis, "Start with the Most Difficult Part" (**IEEE Software**, March/April 2009, p. 70–71)
- "When faced with a problem you don't understand, do any part you do understand, then look at it again." (G. Booch in *Object-Oriented Design with Applications* citing from Robert Heinlein's *The Moon is a Harsh Mistress*)

How to carry on

All principles and methodologies of software development are rules.

No Rules are Universal
All Rules Need Context

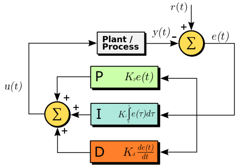
Systems nowadays are so complex...

How do you know
What to do?

PID Controller Principle

Proportional-**I**ntegral-**D**erivative stability mechanism used in industrial control systems

1. Find out where you are
2. Take a small step towards your goal
3. Adjust your understanding based on what you learnt
4. Repeat
5. [additional] When faced with two or more alternatives that deliver roughly the same value, take the path that makes future changes easier



Where to look for further study and ideas?

- Agile Manifesto and its Twelve Princiles
- Agile is Dead by Pragmatic Dave Thomas (keynote address at *GOTO Amsterdam 2015*)
- DevOps in Practice by Danilo Sato
- Diomidis Spinellis Start with the Most Difficult Part (**IEEE Software**, March/April 2009, p. 70–71)