5CCS2OSD: Object-oriented Specification and Design

Lecturer: Kevin Lano (kevin.lano@kcl.ac.uk)

Assistant: Sobhan Yassipour-Tehrani
(sobhan.yassipour_tehrani@kcl.ac.uk)

Lectures: Thursdays 11-13 in Waterloo Campus FWB B5.

Tutorials (From 5th October): Thursdays 13-14 in Waterloo Campus FWB B5.

Surgeries: Will start after coursework released (12th Oct).

Course Aims

- The module aims to introduce students to object-oriented analysis and design (OOA+D) using UML.
- We aim to describe the key *notations and techniques* used in *software specification and design*, using *object-oriented methods*, with focus on *UML and OCL*.

Learning outcomes

To be able to:

- Explain + understand key concepts of OOA+D
- Read + produce OOA+D models using UML
- Apply OOA+D, evaluating alternative solutions
- Evaluate quality of OOA+D models
- Effectively interact with stakeholders + other designers in producing models
- Use state-of-art OO modelling tools for OOA+D.

Assessment

- January exam (2 hours). 85% of course marks.
- 1 coursework, counting 15% of course marks. Carried out in small teams of 5 people.

Coursework divided into 3 parts: (i) requirements + specification, (ii) design, (iii) implementation of a system using UML and UML tools.

Deadline: 4pm on Thursday 30th November.

Submission by team leader/one representative on Keats.

Syllabus/Course Structure

1. **Software engineering and software modelling**: history, motivation, software development process, introduction to UML, software project management + quality assurance
2. **Software specification with UML**: use cases, class diagrams, OCL.
3. **UML behaviour models**: state machines, interactions, activities
4. **Software design with UML**: architecture diagrams, refactoring, design patterns, libraries + reuse
5. **Development processes**: agile development, Model-based development, professional issues.

Textbooks

UML and software modelling:

- Kevin Lano, Agile Model-Based Development using UML-RSDS, CRC Press, 2016
- Kevin Lano, Model-Driven Software Development with UML and Java, CENGAGE Learning, 2009
- Martin Fowler, UML Distilled (3rd Edition), Addison Wesley, 2003 (online pdf).

Also recommended: Roger Pressman, Software Engineering, a practitioners approach, McGraw Hill.

Lecture notes, tutorials, etc available on Keats and at: nms.kcl.ac.uk/kevin.lano/5ccs2osd.

Textbooks

Main textbook covers material in more detail + gives more examples:

- Part 2 – Chapters 3, 4, 5
- Part 3 – Chapter 19
- Part 4 – Chapters 9, 10
- Part 5 – Chapters 1, 3, 16.

Alternative textbook:

- Parts 2, 3 covered in Chapters 2, 3, 4
- Part 4 covered in Chapter 6
- Part 5 partly covered in Chapter 1.

Lecture notes are sufficient for exam.

PART 1: Software Engineering and Software Modelling

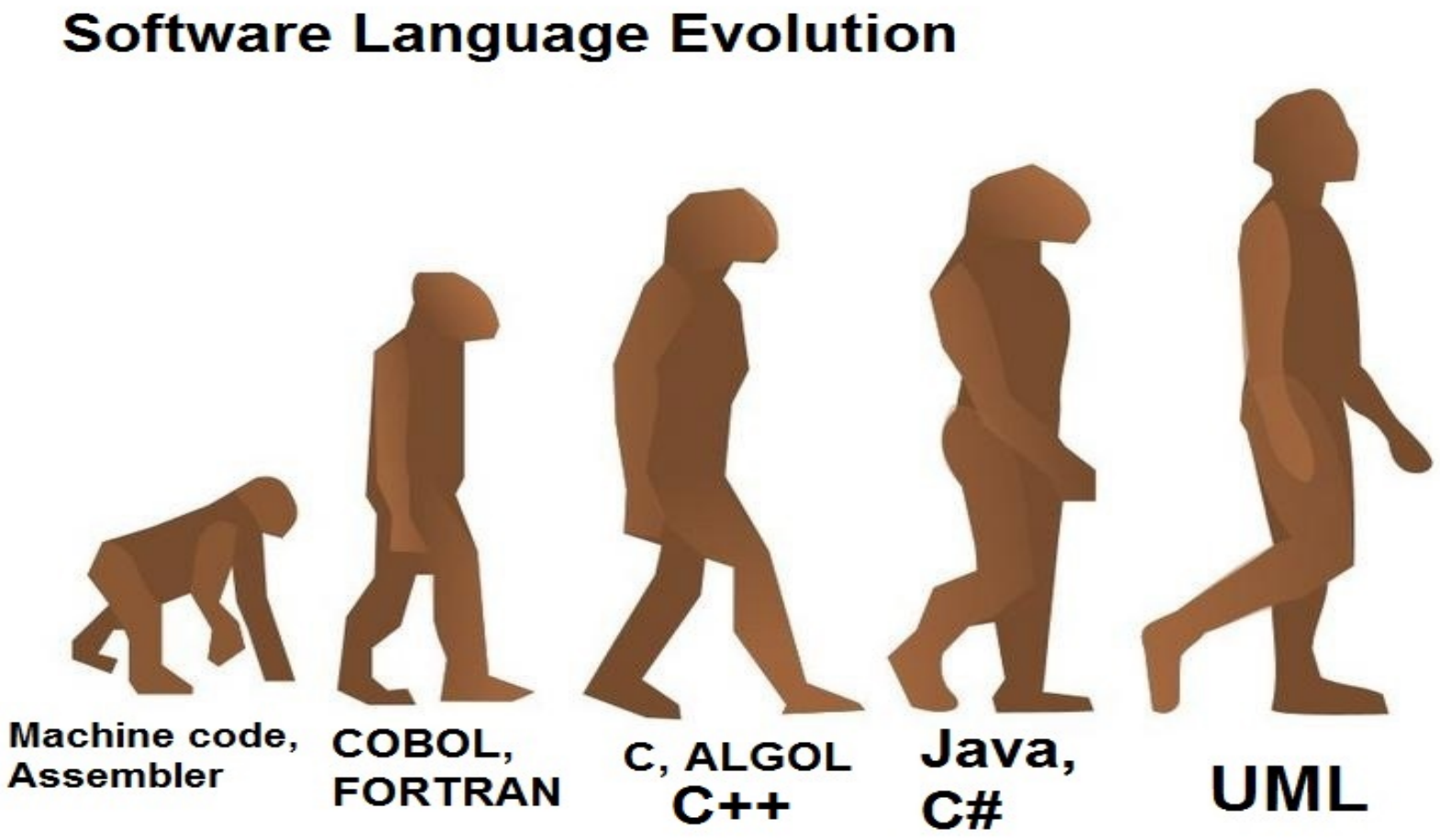
- **Software Engineering:** "The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software." (from Pressman).
- **Software Modelling:** Using graphical or textual models to describe software requirements, specifications, designs, etc., to support software development.

Some history ...

- At first, back in 'prehistoric' computing times (1940's, 1950's), programmers worked with primitive languages: machine code, assembly languages, then FORTRAN (1957), COBOL (1959). Programming effort was huge for relatively simple programs.
- Productivity improved with more advanced languages: ALGOL, BASIC (1960's), Pascal (1971), C (1972), C++ (1980's). The first modelling languages: flowcharts, entity-relationship diagrams (ERDs) enabled programmers to model and analyse algorithms + data of their programs *before* coding. Era of *structured programming*.
- C++ started the object-oriented revolution + became popular from late 80's onwards. OO modelling began, with notations such as class diagrams, use cases.
- Java (1995) simplified C++ to avoid pointers + other horrors

of C++

- 1997: OO methods unified into Unified Modelling Language (UML)
- Software continues to get larger + more complex, requiring more powerful *software engineering* methods to manage its construction.
- 00's: Agile development and Model-based development (MBD).
- Currently: integration of agile + MBD.



Evolution of software languages

Increasing programming power – simpler code for same task:

```
// old Java style iteration:
for (int i = 0; i < students.size(); i++)
{ Student st = (Student) students.get(i);
...
}


// improved Java style, C#:
foreach (Student st in students)
{ ... }


// UML, OCL style iteration:
students->forAll( st | ... )
```

Motivation: why use software engineering/modelling?

- Programming languages have become more + more powerful, and increasingly high-level: further from machine, closer to requirements/human language
- But need for software has grown even faster – and used in critical and essential applications (car + transport systems, communications, medical, robotics, finance ...)
- Catastrophic software failures fortunately rare: Therac 25 (medical); Ariane 5 (space), etc
- But massive costs from failed software projects – eg., £11 billion costs of NHS IT integration project (2011).

"The most likely way for the world to be destroyed, most experts agree, is by accident. That's where we come in; we're computer professionals. We cause accidents." (from Pressman)

Ariane 5 disaster, 1996, $370 million+ cost

The 'Software Crisis'

- Typically, only 30% of software projects completed on schedule + within budget + meet requirements.
- Skill of individual developers is key factor in project success – and this can vary greatly.
- Poor understanding of requirements (by customer + developers) is major cause of failed projects.

Pressman refers to the crisis as 'a chronic affliction' of software development.

Project management in practice ...

MODERN RESOLUTION FOR ALL PROJECTS

	2011	2012	2013	2014	2015
SUCCESSFUL	29%	27%	31%	28%	29%
CHALLENGED	49%	56%	50%	55%	52%
FAILED	22%	17%	19%	17%	19%

The Modern Resolution (OnTime, OnBudget, with a satisfactory result) of all software projects from FY2011 - 2015 within the new CHAOS database. Please note that for the rest of this report CHAOS Resolution will refer to the Modern Resolution definition not the Traditional Resolution definition.

Success rates for software projects (standishgroup.com)

The 'Software Crisis'

- Customers may only know in general terms what they want from software – process of *requirements analysis* needed to identify the true needs
- Requirements analysis also termed *requirements engineering*
- Context of actual use of software must be taken into account – eg., software for GP use in patient consultation must not interfere with GP's work
- Legal + other constraints must be taken into account, eg., data protection regulations.

The 'Software Crisis'

No one 'silver bullet' to solve software crisis, but more systematic engineering processes will help:

- Thorough requirements analysis/engineering
- Automating code production where possible
- Software reuse of trustworthy components where possible (eg., Java Collections library)
- Improved agility
- Software modelling
- Emphasis on software quality.

Motivation: why use software engineering/modelling?

- Modelling has been used for a long time by developers: *Flowcharts* show program control structure as graphs – used since 1960's to help design algorithms + to construct tests (there should be a test for each possible program execution path)
- *ERDs* used since 1970's for design of relational databases
- Diagrams act as a *tool for thought* for the developer, as a *means of communication/explanation* between developers + between team and non-technical stakeholders. Also to *support analysis and reviews* for inspection + quality improvement.

UML activity diagrams support flowcharting, class diagrams extend ERDs.

Motivation: why use software engineering/modelling?

- Imagine you start work in a software company + join team on existing project – how will you learn what project is about/organised, etc?
- Diagrams of data + architecture much faster to read/understand than 1000's of lines of code.
- If you are developing system for non-technical customer, how will you discuss requirements + concepts with them?
- Diagrams more effective to assist such discussions than code – which needs specialist expertise + skill to understand.

Motivation: why use software engineering/modelling?

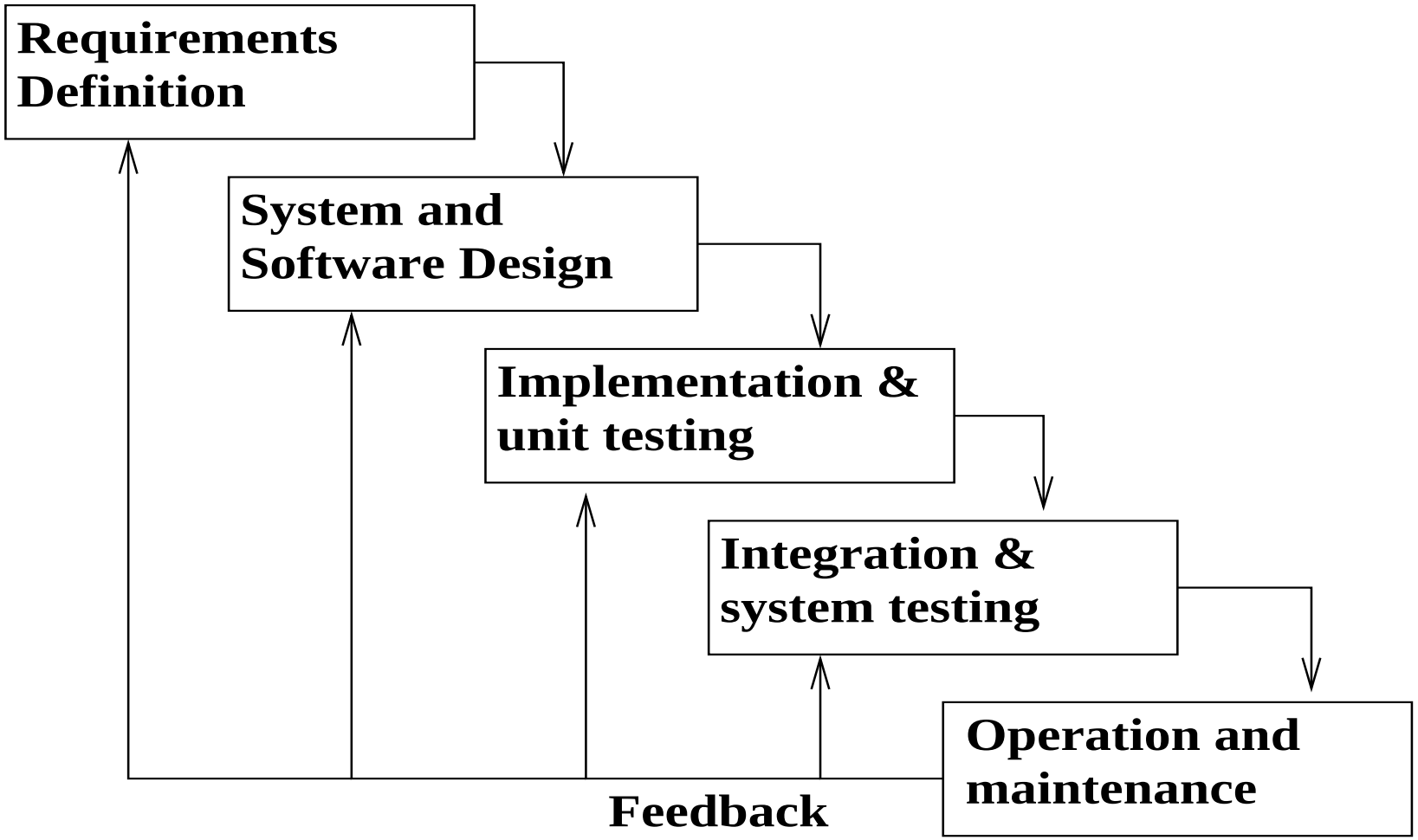
- UML is required skill for many business analyst positions, eg., www.cwjobs.co.uk/uml/in-london?radius=10&s=header
- 2013 survey of industry identified that 30% of companies use UML in projects
- UML used in SEG, PRJ, SIA, SAD courses. OCL in SAD.

The software development process

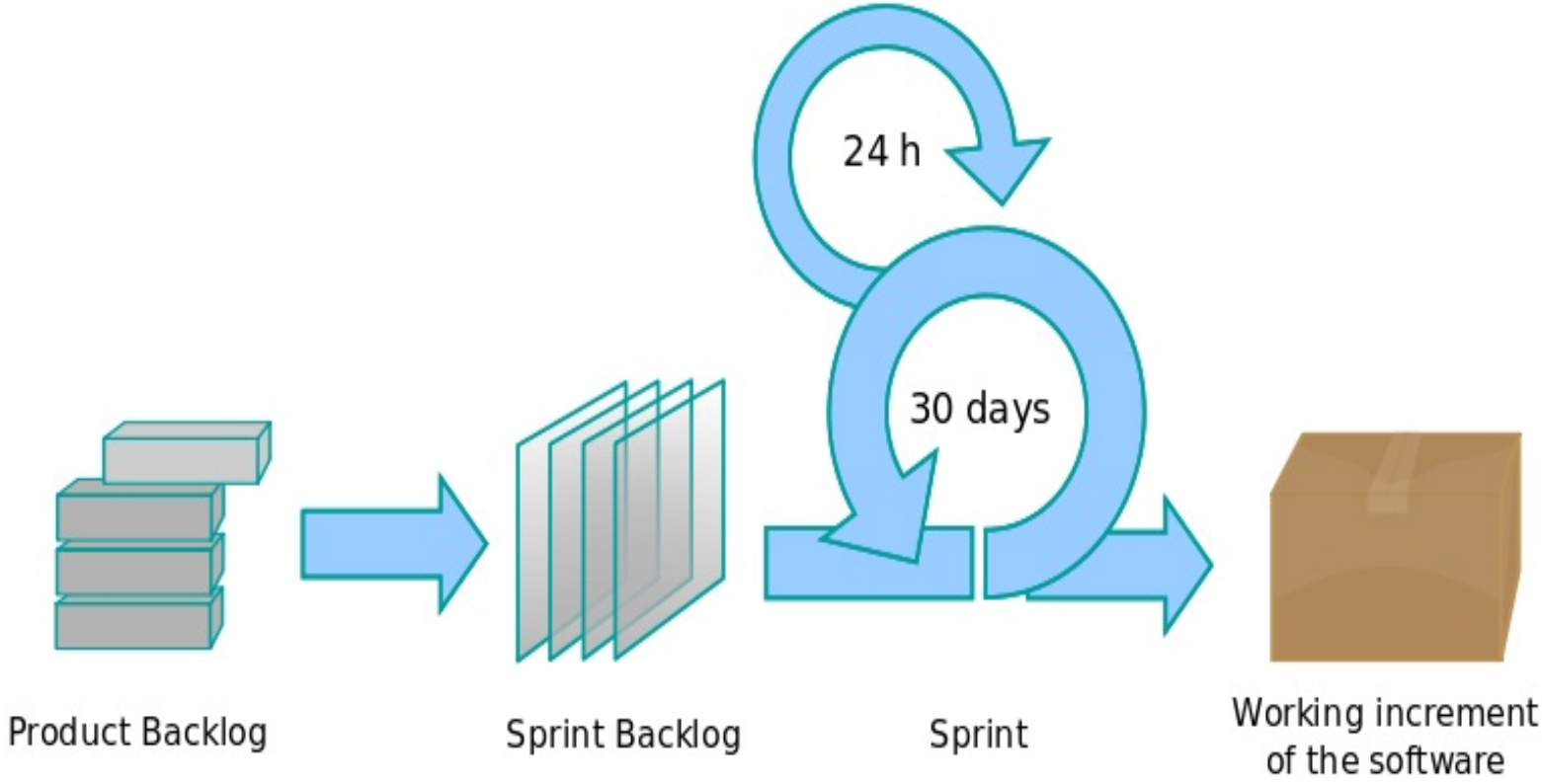
A number of stages typically present in any software development project:

- Feasibility analysis
- Requirements analysis
- Specification
- Design
- Implementation
- Testing
- Maintenance.

Stages may be repeated (in *iterative* development processes such as Scrum agile method) or carried out once to completion (eg, Waterfall model).



'Waterfall' or staged development



Scrum agile development process

Feasibility analysis

- Asks "Is there a business case for system? Will it be used?"
- Technical feasibility – is it possible with available technology?
- Financial feasibility – can be developed with available budget?
- Time – is it possible to develop in a useful time-frame?
- Resources – are necessary resources available for the development?

If a project proposal fails these tests, very high risk to attempt development!

Requirements analysis

- Systematically identifies + records requirements of stakeholders + customer(s) of system, + constraints imposed on system by existing systems it operates with/by existing work practices/regulations.
- Requirements divide into *functional requirements* (services/functions) + *non-functional requirements* (efficiency, usability, extensibility, etc).
- May be *conflicts* between different requirements & ambiguities in informal requirements. Should be resolved before specification constructed.
- Use cases can be used to document initial functional requirements.

Stages in requirements analysis process

Four main phases:

- *Domain analysis, requirements elicitation*: identify stakeholders, gather information on domain + requirements from users, customers, other stakeholders/sources.
- *Evaluation and negotiation*: identify conflicts, imprecision, omissions, redundancies in requirements; consult + negotiate with stakeholders to agree resolutions.
- *Specification and documentation*: systematically document requirements as system specification, in formal/precise notation: Agreement between developers + stakeholders on what will be delivered.
- *Validation and verification*: check formalised requirements for consistency, completeness and correctness wrt stakeholder requirements.

Requirements analysis techniques

- Interviews with stakeholders
- Brainstorming sessions
- Observation of existing processes/practices
- Scenario analysis – model specific scenarios of use of system, eg., as UML sequence diagrams
- Document mining
- Goal decomposition
- Exploratory prototyping.

Thorough requirements analysis can reduce errors + costs later in development.

Specification

- Builds precise models of system, using graphical or mathematical notations to represent required state and behaviour in abstract, platform-independent manner.
- UML class diagrams are ideal notation for this stage.
- Important to avoid implementation details. Only include sufficient detail necessary to specify logical properties of system.

Specification forms contract between developers and stakeholders – defines precisely + unambiguously what is to be developed + what capabilities developed software will have.

Design

Based on specification, defines architecture and structure for system, dividing into subsystems/modules responsible for parts of system functionality + data. Design includes:

1. *Architectural design*: define global architecture of system, as set of major subsystems, + dependencies between them.

   Eg, partitioning system into GUI, functional core, data repository. GUI depends on core, core depends on repository.

2. *Subsystem design*: decompose global subsystems into smaller subsystems. Continue until clearly identified *modules* emerge.

   Module typically consists of single class or group of closely related classes + operations on these.

3. *Module design*: define each module, in terms of:

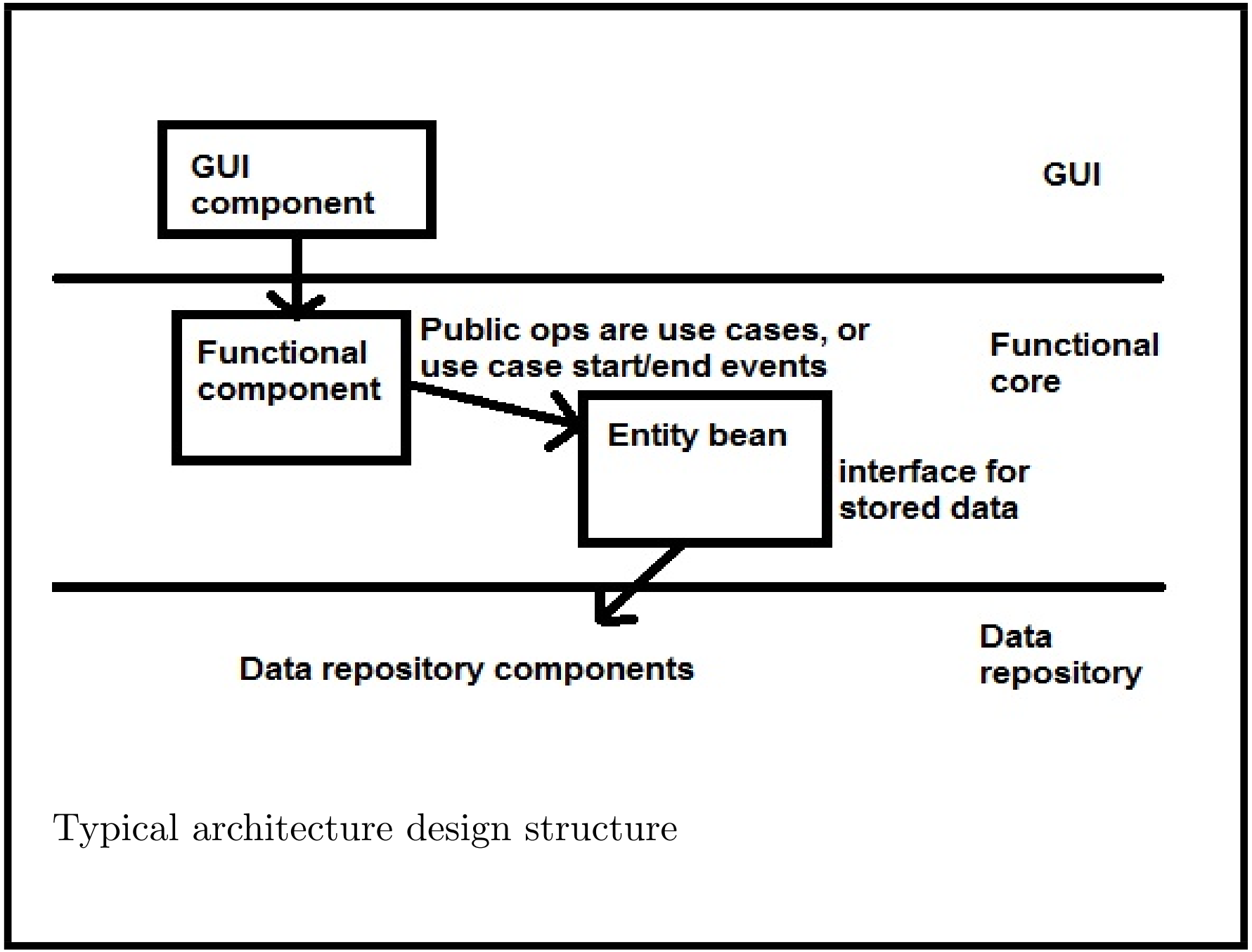
   (a) data it encapsulates – attributes/associations;

(b) properties (invariants or constraints) it is responsible for maintaining;

(c) operations it provides (external services) – eg: their names, input + output data, and specifications. This is called *interface* of the module.

4. *Detailed design*: for each operation of module, identify steps of its processing.

Structure of design may evolve as experience with prototypes of system grows, for example.



Typical architecture design structure

Architecture diagrams

Main notation used to document architecture of systems.

These consist of:

- Rectangles – show subsystems or modules. Nesting indicates that a module/subsystem belongs to a subsystem.
- Arrows – show that component at start of arrow depends on component at target, eg., it calls operations of target (client/supplier relationship).

Subsystems/modules become packages in a Java implementation.

UI design and database design

Specialised forms of design include:

- *user interface design*, to plan appearance/sequencing of dialogs + other graphical interface elements for user interaction
- *database design*, to define structure of database for system, + what queries/updates will take place on it.

We won't cover these in this course, as covered in other courses (PPA, DBS).

Implementation

- Code is produced from the design in one or more programming languages
- Traditionally, this is done manually
- In model-driven engineering (MDE)/model-based development (MBD) approaches, is automated
- Automated coding reduces time + cost, but problems arise if generated code needs to be manually adapted.

"Einstein argued that there must be a simple explanation of nature, because God is not capricious or arbitrary. No such faith comforts the software engineer" (Fred Brooks).

Testing

Aim of testing is to discover errors in system.

Can be either *white-box*: based on internal code structure, designed to test each program path; or *black-box*: based on requirements.

Testing applied at several levels:

- Code/Unit testing: test each component separately (mainly white-box)
- Integration testing: test that components interact correctly
- System testing: test entire system
- Acceptance tests: test against requirements (mainly black-box).

Maintenance

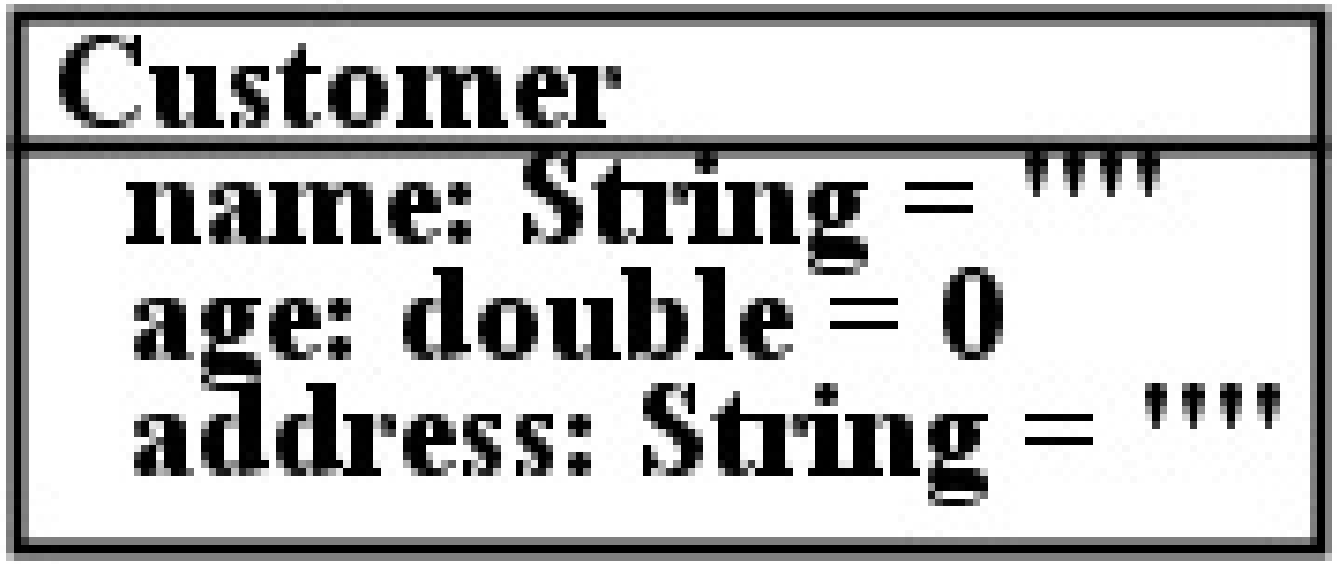
All post-delivery activities, including:

- Correction: bug-fixing and correcting defects
- Adaption: changing system to operate in new/updated environment
- Enhancement: extension to handle new requirements
- Prevention: re-engineering to improve structure of system + enhance its future evolution.

These activities can consume far more resources than development of entirely new systems.

Introduction to the UML

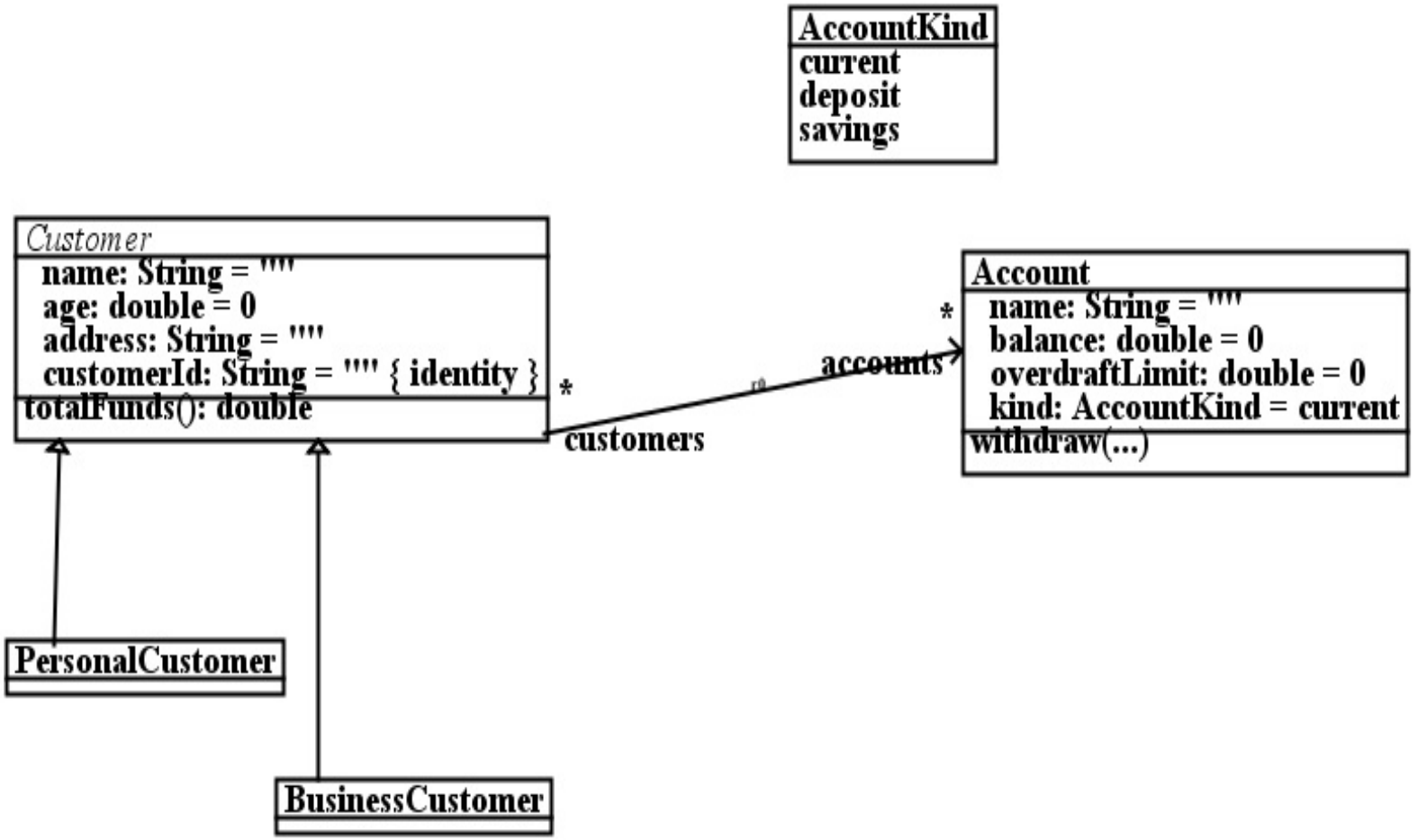
- UML, the "Unified Modelling Language", introduced in 1990's. Unification of different OO approaches – now international standard controlled by OMG industry consortium (IBM, Microsoft, Oracle, etc).
- Most widely-used modelling notation in industry; many thousands of tools and books for UML.
- Main notations: class diagrams; use case diagrams; OCL; state machines; sequence diagrams; activities.



Example of class diagram

Software modelling using UML

- Motivation for UML modelling is to (i) *Assist developers to understand and document the project*; (ii) *To precisely express project requirements*, in a System Requirements Specification (SRS); (iii) to *define reusable models* capturing domain concepts common to several projects; (iv) to *support model-based development (MBD)*, including automated code generation from models.
- Class diagrams show: entities of system (eg., *Customer*) with data (*name*, *age*, *address*), relationships (lines between entities), specialisations (inheritance arrows).



Classes with operations, association, inheritances

Class diagrams

- Central modelling notation of UML
- Similar concepts to OO programming languages: classes, members (attributes), methods (operations), inheritance, etc.
- Usually a UML class can be translated directly to a Java, C#, C++ class, or to a C `struct`.

Java:

```
class Customer
{ private String name = "";
  private double age = 0;
  private String address = "";
  private String customerId = "";  ...
}
```

C#:

```
class Customer
{ private string name = "";
  private double age = 0;
  private string address = "";
  private string customerId = "";
  ...
}
```

C++:

```
class Customer
{ private:
     string name;
     double age;
     string address;
     string customerId;
  public:
  ...
```

```
}
```

C:

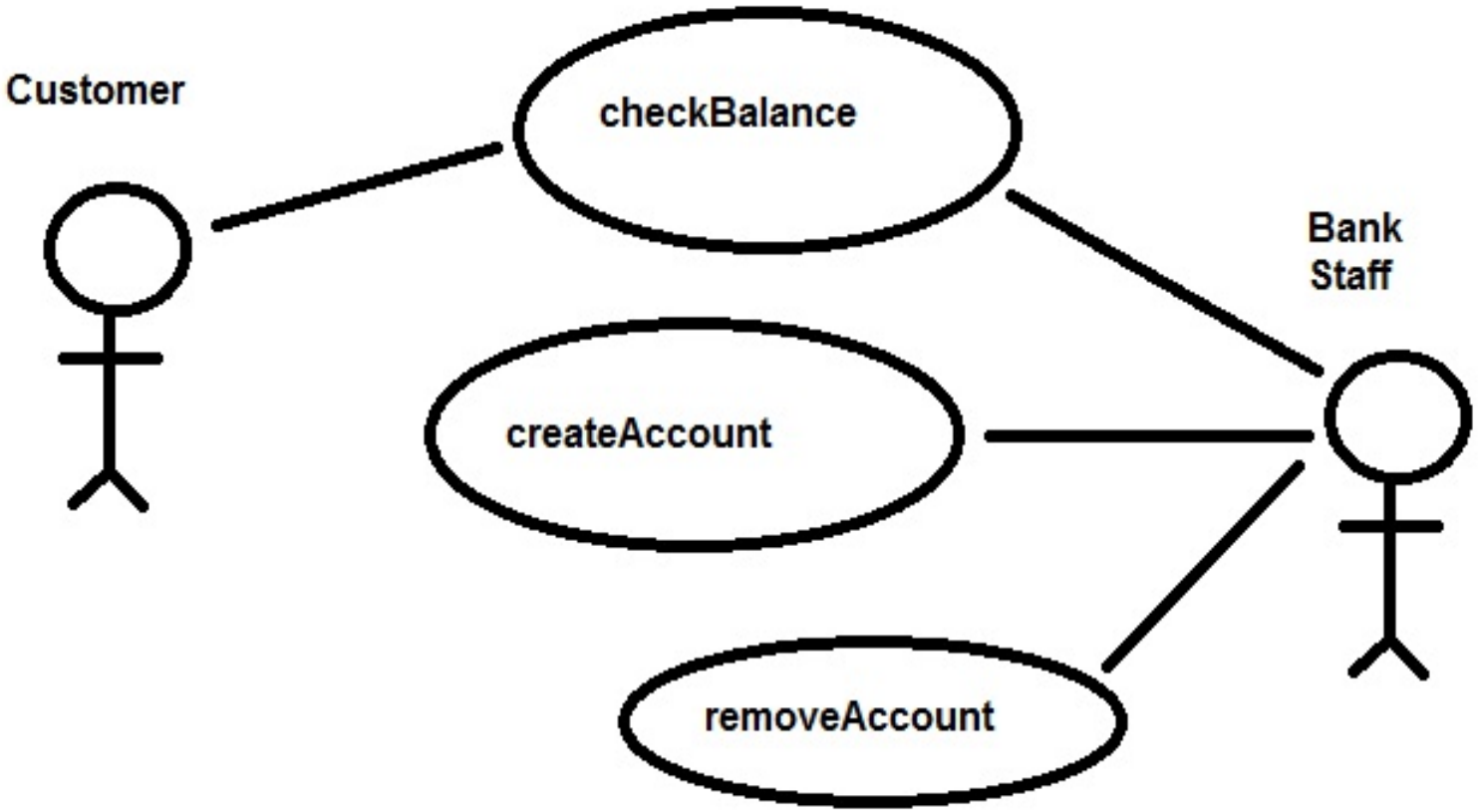
```
struct Customer
{ char* name;
  double age;
  char* address;
  char* customerId;
};

...
```

Other UML notations

- *Use case diagrams:* describe operations of system from user perspective
- *State machines:* define events + states of objects
- *Sequence diagrams:* show examples of interactions (communications) between objects + between users and system
- *OCL:* expression language, defines logical constraints
- *Activity diagrams:* flowchart/pseudocode language to express behaviour.

Although powerful, UML is complex. Some companies prefer to use lightweight modelling using XML, Excel, etc.

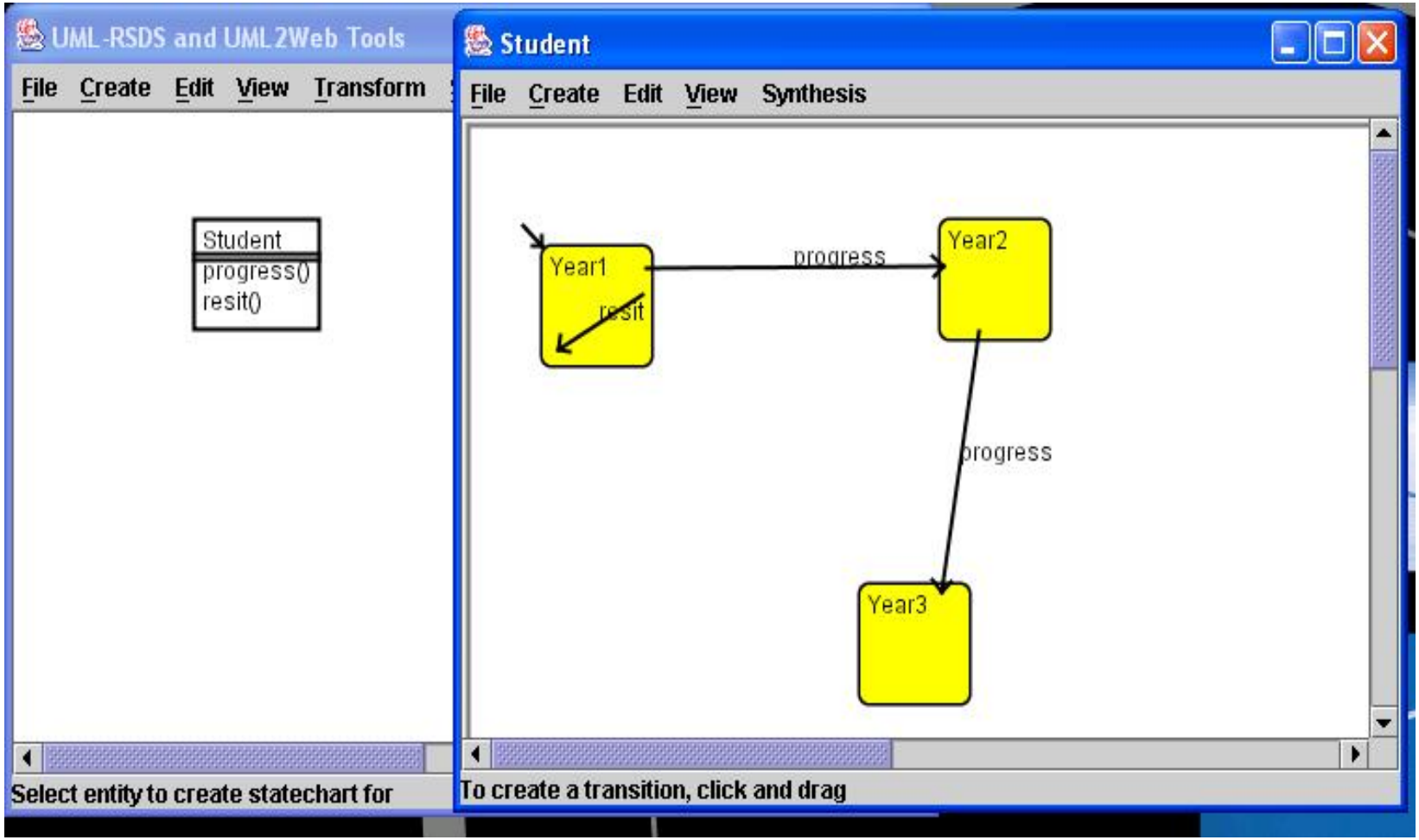


Example of use case specification

State machines

- State machines define dynamic behaviour of objects + operations.
- Can define operation processing, and life histories of objects.
- Eg., a student could have linear lifecycle of successive states $Year1,\ Year2,\ Year3$, + possibility to resit year 1.

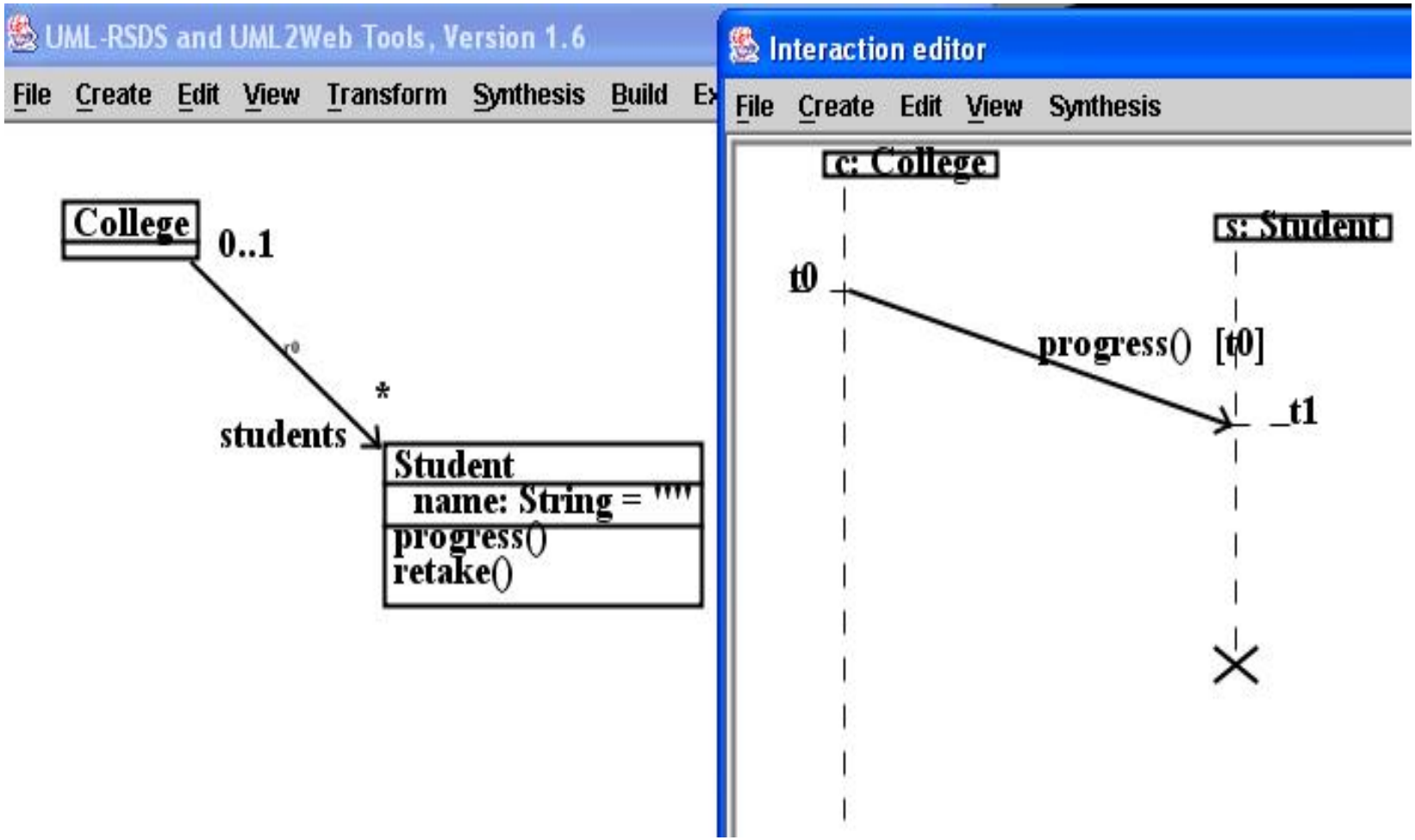
Used to describe user interaction with a UI, phases in behaviour of a robot, etc.



State machine example

Interactions

- UML *sequence diagrams* describe examples of system behaviour in terms of object communications.
- Show object instances $obj : Entity$ and messages exchanged between objects.
- Time increases from top to bottom of diagram.



Interactions example

Software Project Management

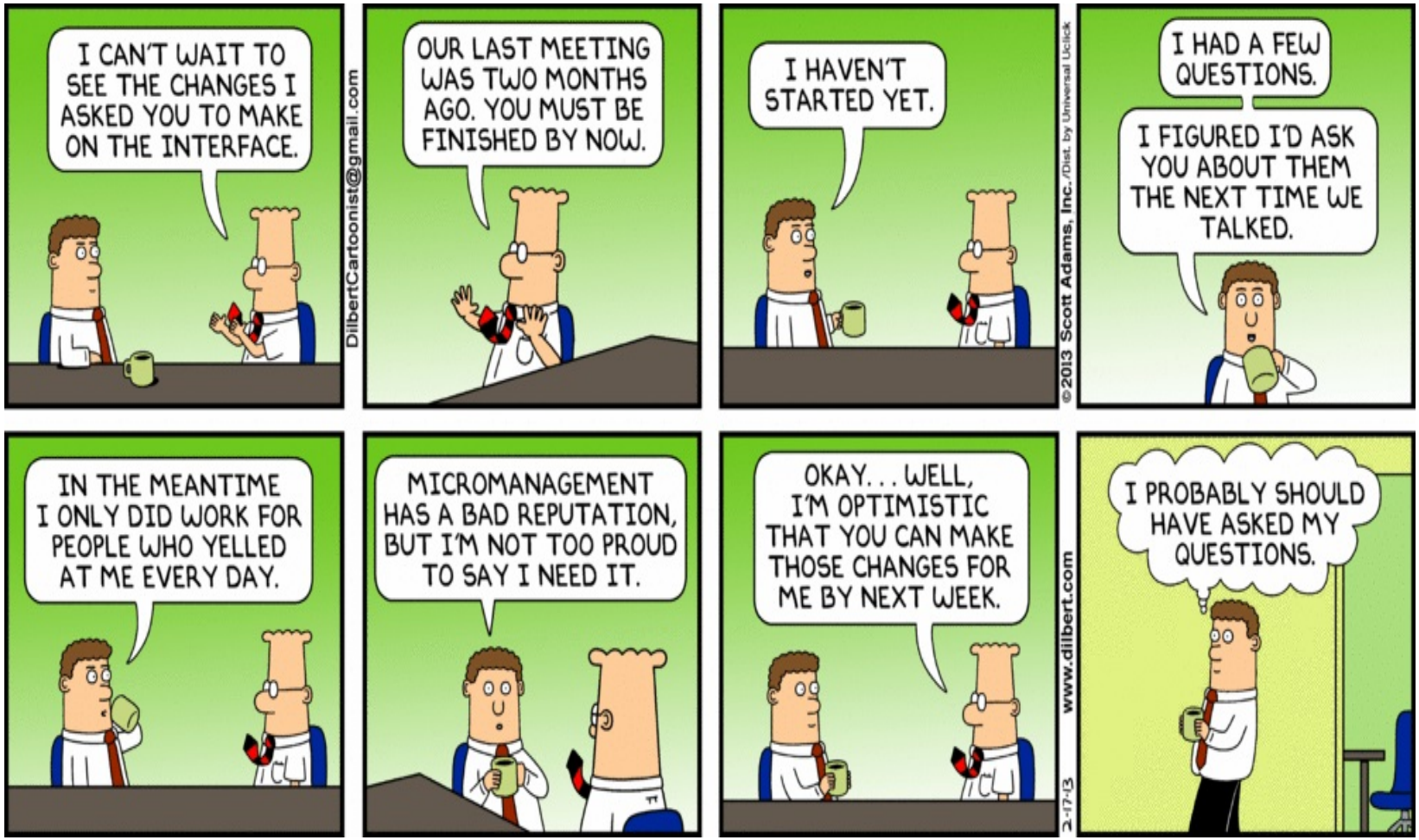
This involves:

- Management of *people* and *process* used to produce a software system.
- Scoping + objectives of project, constraints.
- Organisation/selection of development teams, roles within teams.
- Task breakdown + assignment, determining work schedule. Cost + risk estimation. Contingency planning.

Project management in practice ...

Software Project Management

- Team organisation is a key success factor.
- Team leader/organiser co-ordinates work of team, ensures schedule maintained.
- Leader needs inter-personal skills – different to technical skills.
- Clear assignment of responsibilities to team members, with agreed processes for working.
- Conflict + problem resolution procedures.



Project management in practice ...

Software Project Management

- *Scoping* – carried out during feasibility + requirements analysis stages.
- Determine what is within + outside project scope.
- By specification stage scope must be agreed between customer + developers.
- *Decomposition* – decompose problem into subtasks during requirements analysis.
- *Allocate resources* – assign developers to tasks, draft schedule, estimate costs + time.
- *Risk management* – proactively identify potential risks, avoidance strategies + contingency plans.

Software Quality Assurance

- *Software quality* can be measured in many ways – eg., code complexity, architecture modularity, absence of 'bad smells' such as excessive parameters in a method.
- Software should be *fit for purpose*: satisfy stakeholders actual needs for it + be usable + safe in the real-world context of use.

Software Quality Assurance

Elements of SQA:

1. A quality management approach
2. Effective software engineering methods + tools
3. Formal technical reviews of software
4. Multi-level testing strategy (unit, module and acceptance tests, etc)
5. Compiliance to software standards
6. Measurement + reporting mechanisms.

Although detailed inspections + testing are time-consuming, these save costs of fixing bugs in delivered code.

Formal technical reviews

- Systematic examination of part of software/documentation – by reviewer(s) independent of person/team that produced the artifact
- Aims to uncover errors
- In requirements analysis, applied in validation stage, to the requirements spec.
- During design, applied to models/architectures
- During implementation, applied to code.

Summary of Part 1

- Introduced concepts of software modelling, and historical background
- Given justifications for use of modelling
- Introduced software development process, + requirements engineering
- Introduced UML modelling notations
- Introduced software project management + quality assurance.

Subsequently, we will describe in detail UML notations + give many examples of their use.