

# 7CCSMSUF: Software Engineering and Underlying Technologies for Financial Systems

Kevin Lano

`kevin.lano@kcl.ac.uk`

## *7CCSMSUF Software Engineering for Finance: Syllabus*

- Introduction to software engineering concepts
- Financial services and markets
- Financial products and analysis
- Specification and design with UML
- Trading technologies
- Data analytics technologies.

# 7CCSMSUF: Software Engineering and Underlying Technology for Financial Systems

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

Industrial advisor: Howard Haughton

**Lecture/tutorial:** Thursdays 1pm-3pm, BH S1.01

In the event of disruption by transport strikes, etc, I will setup a Teams online LGT instead.

### *Textbooks*

- Main course book: **Financial software engineering: Kevin Lano, Howard Haughton, Springer-Verlag, 2019**
- UML: Martin Fowler, UML Distilled (3rd Edition), Addison Wesley
- Software Engineering: Roger Pressman, Software Engineering, a practitioners approach, McGraw Hill.
- Finance: John Hull, Options, Futures and other Derivatives, Pearson.

Lecture notes, tutorials, etc available on Keats.

### *Assessment*

- May exam (2 hours). 100% of course marks.
- Consists of 10 essay-type questions.
- In-person, paper-based exam. Closed-book.

Revision lecture at end of term.

# Part 1. Introduction to software engineering concepts

- Software in finance
- Software lifecycle stages
- Software modelling using UML
- Agile development methods: Scrum, XP, Kanban
- Model-driven engineering (MDE)
- Financial models.

## *Software in Finance*

Used for:

- Valuation of (assigning price to) financial products
- Investigation of properties of new products
- Financial risk analysis
- Trade management
- Data analysis
- Data storage
- Communication of financial data.

## *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.



## *Software engineering in finance*

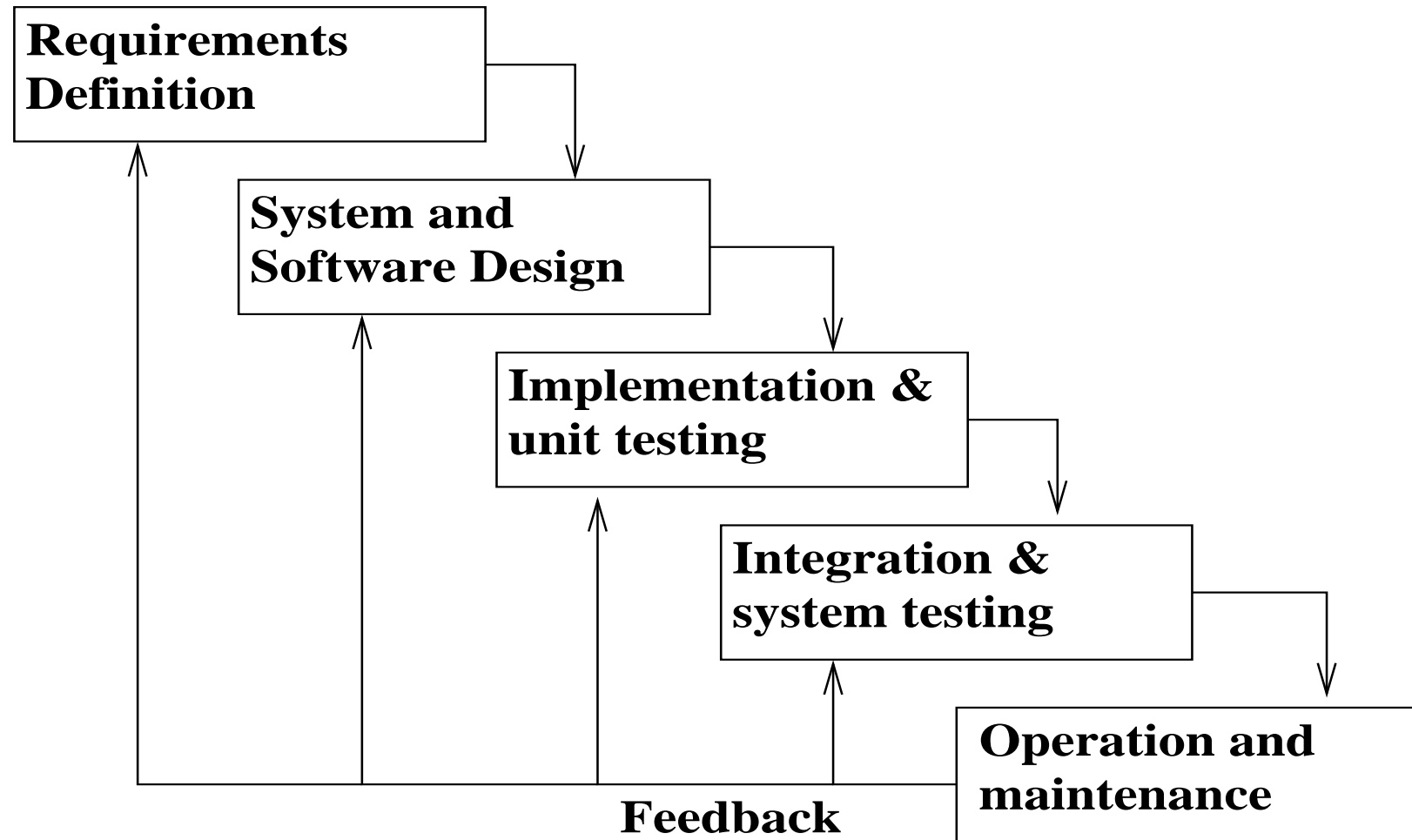
- Common practice in financial software development involves prototyping in language such as Excel, Matlab or Python,
- then either (i) retaining these prototypes as production software, or (ii) using manual coding to produce versions in C++ or C#.
- Option (i) has deficiencies in terms of software quality and efficiency
- Option (ii) is costly, and involves duplicating the software functionality.
- Alternatively, retain specification/prototype, use automated code generation to produce efficient production implementation from these.

## *Software lifecycle stages*

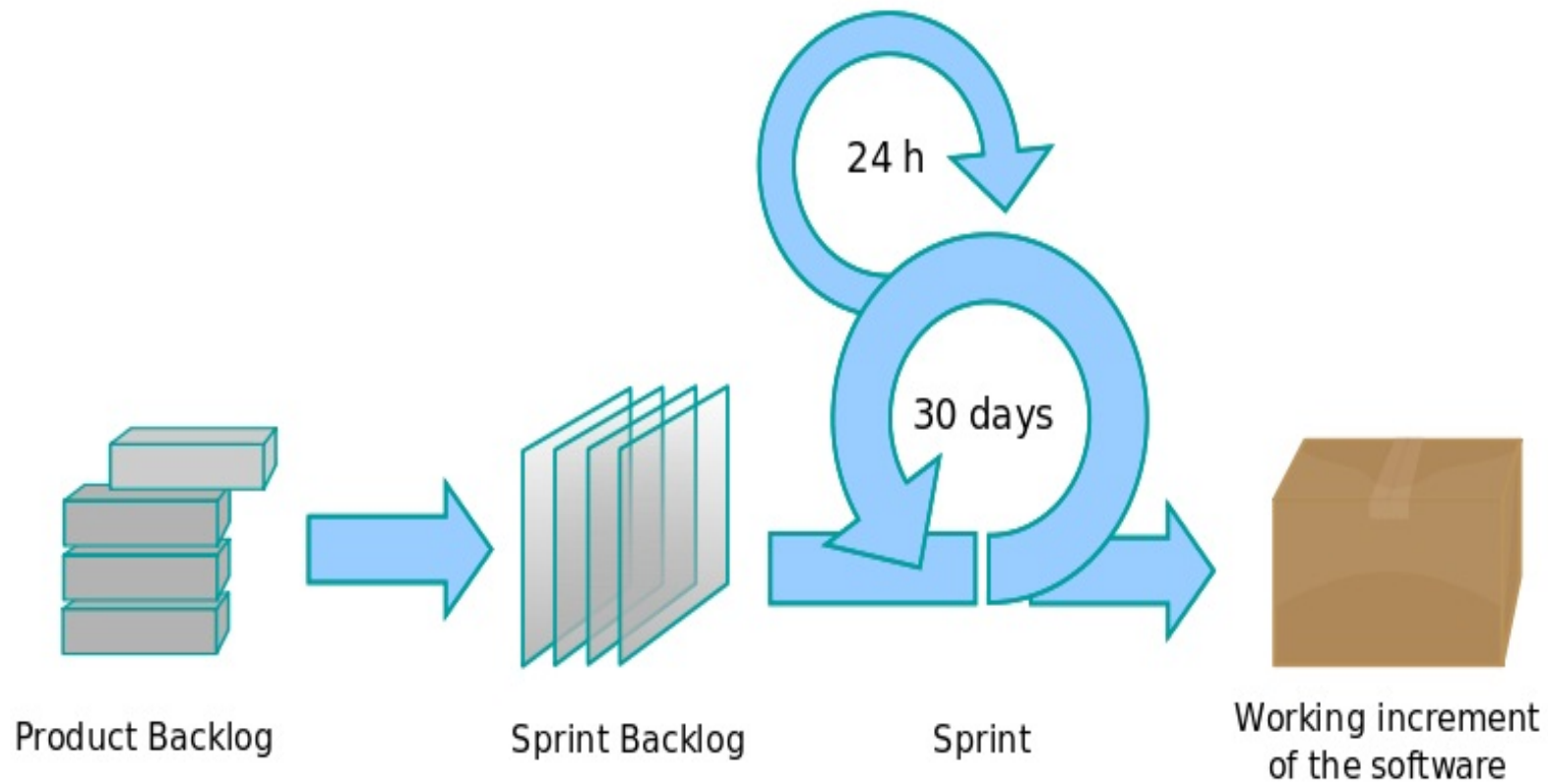
A number of stages typically present in any software development project:

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

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

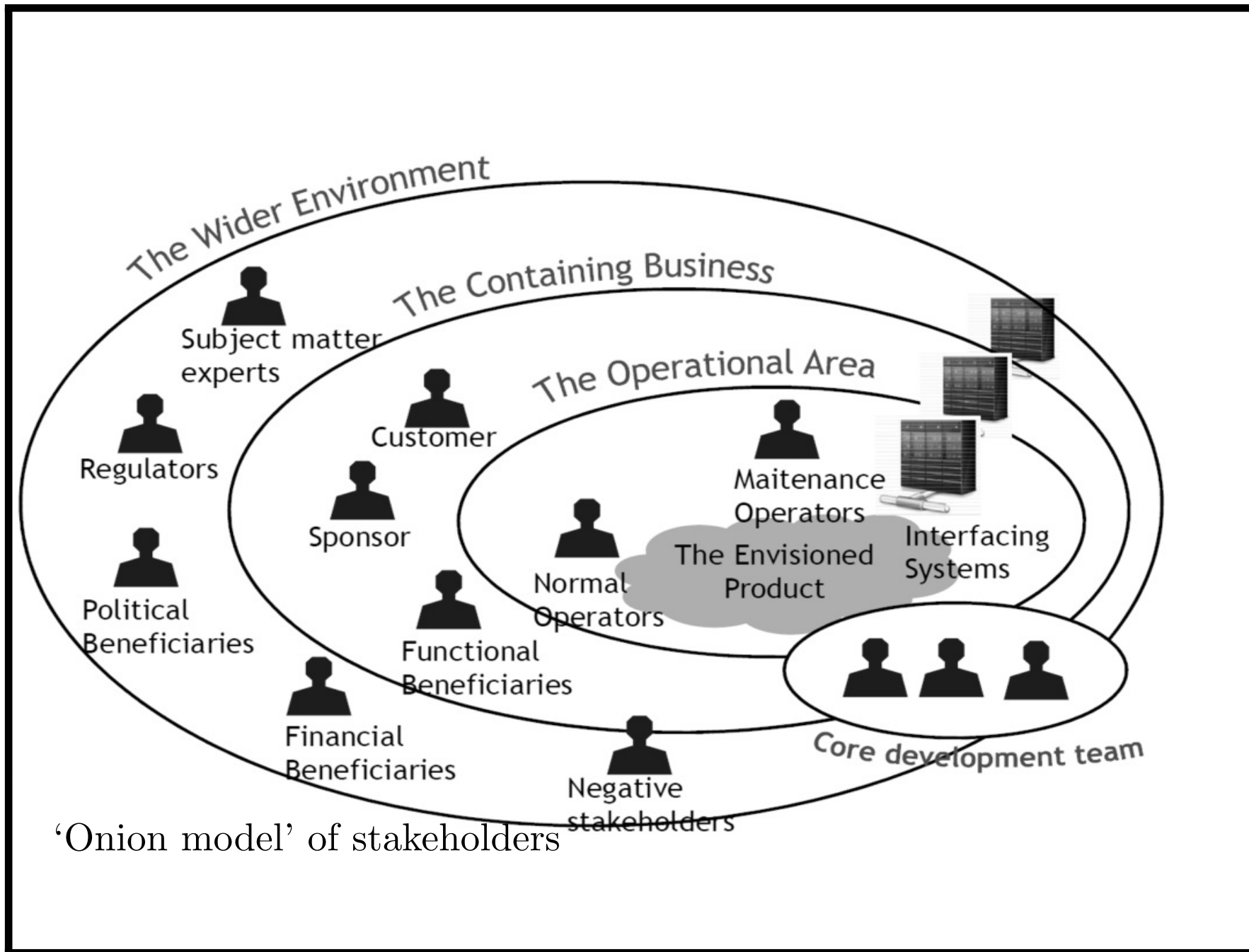
### *Feasibility analysis*

- 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 (people, tools) 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.

# Domain Understanding and Requirements Elicitation

Background Study



Interview



Workshop



Observation/User Studies



Surveys/Questionnaires



Users Forums, Feature Requests



Requirements elicitation techniques

## *User Stories*

- Agile methods express requirements as *user stories*
- “As a [user], I need to [required functionality], so that [user objective]”
- Eg: “As an online bank customer, I need to view the current balance and recent transactions of my accounts, so I can manage my finances.”
- The *required functionality* becomes a use case that system should provide.

User stories may be grouped in *epics* of related functional requirements.

## *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.

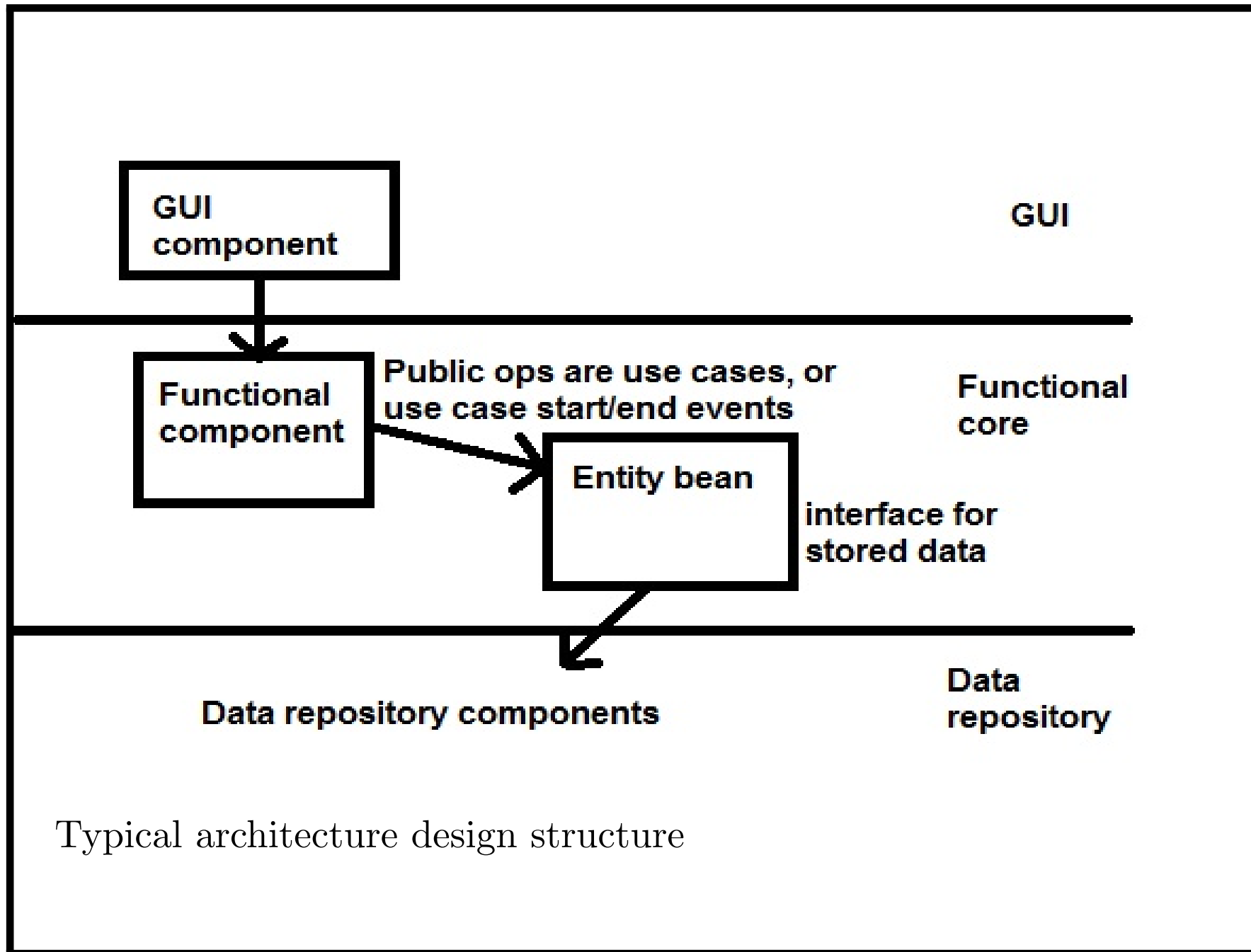
3. *Module design*: define each module, in terms of:

- (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.



## *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.



## *Implementation*

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

## *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*

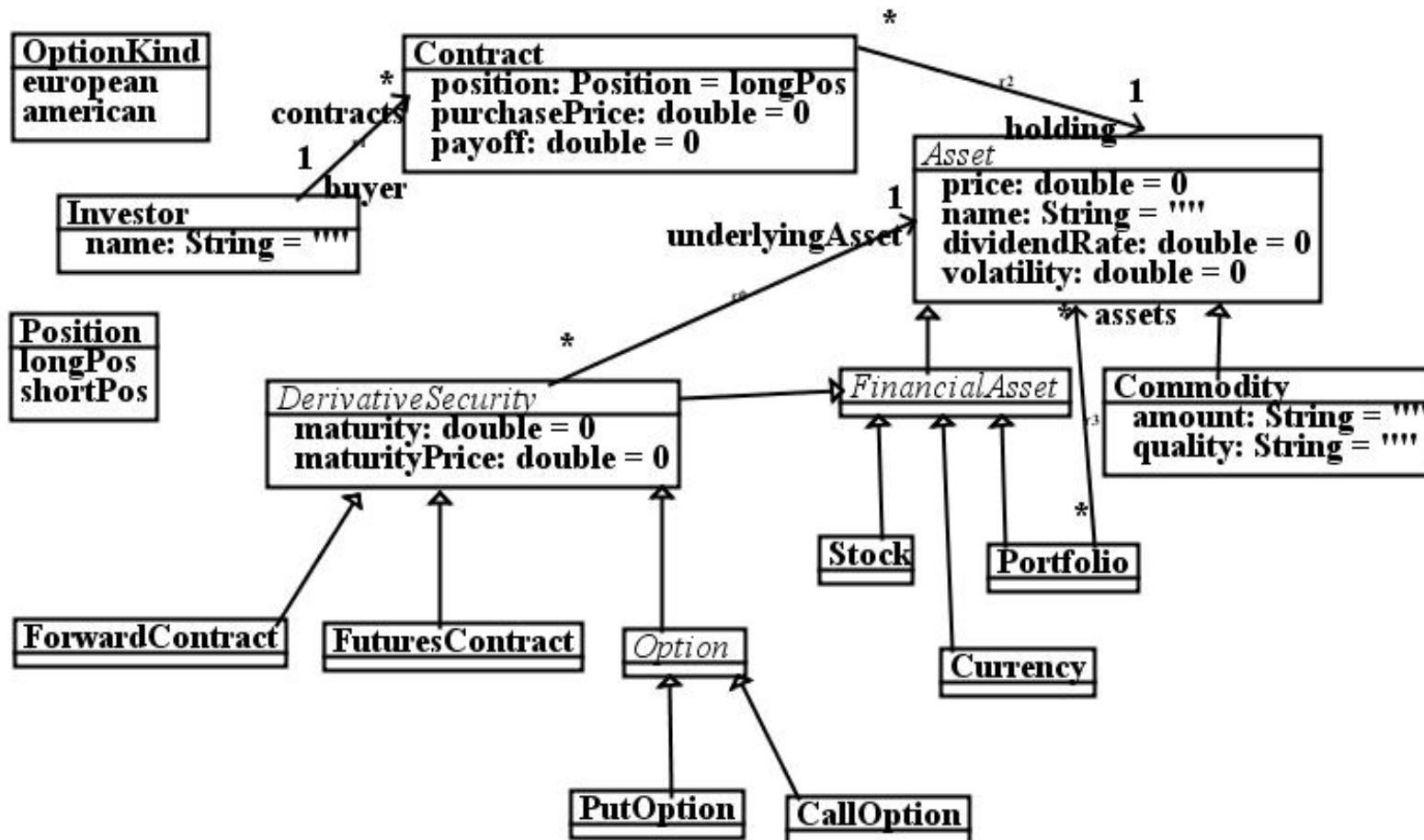
All post-delivery activities, including:

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

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

### *Software modelling using UML*

- UML, the “Unified Modelling Language”, introduced in 1990’s as unification of different OO approaches – now an international standard controlled by OMG industry consortium (IBM, Microsoft, Oracle, etc).
- The most widely-used modelling notation in industry; many thousands of tools and books for UML.
- Main notations: class diagrams; use case diagrams; state machines; sequence diagrams; activities; deployment diagrams.



Example UML class diagram

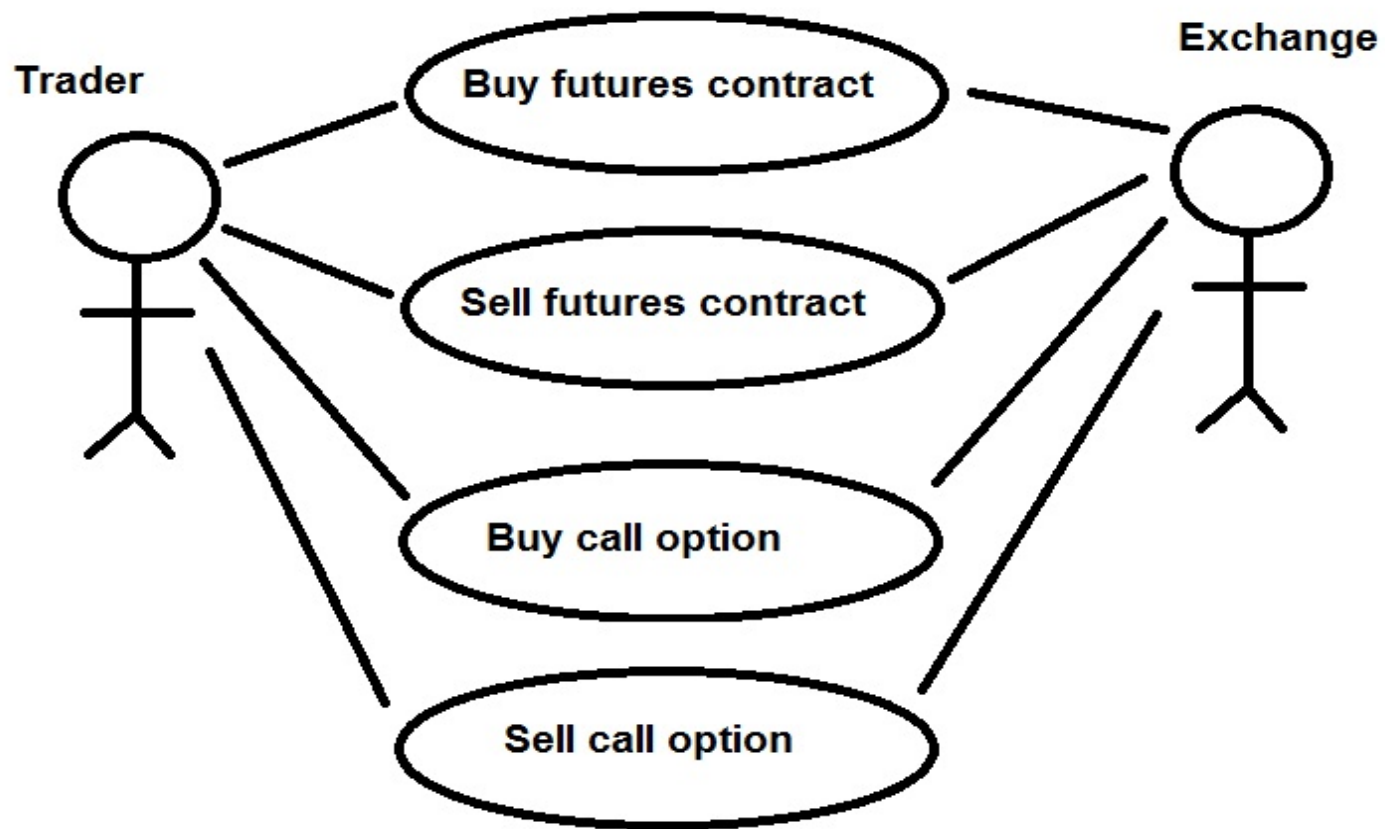
## *Software modelling using UML*

- Motivation for UML modelling is to (i) *precisely define requirements*, in a System Requirements Specification (SRS); (ii) to *define reusable models* capturing domain concepts; (iii) to *support Model-driven engineering*, including code generation from models.
- Class diagrams show: entities of system (eg., *Investor*) with data, relationships (lines between entities: a *DerivativeSecurity* has an *underlyingAsset*, which is an *Asset*) and specialisations (*Commodity* is a special case of *Asset*).
- Can also show business rules as constraints.

### *Other UML notations*

- *Use case diagrams*: describe functional capabilities of system from user perspective;
- *State machines*: define events and states of objects;
- *Sequence diagrams*: show examples of interactions (communications) between objects + between users and system;
- *Deployment diagrams*: show physical configurations of devices and distribution of software across devices.

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



Use case diagram example



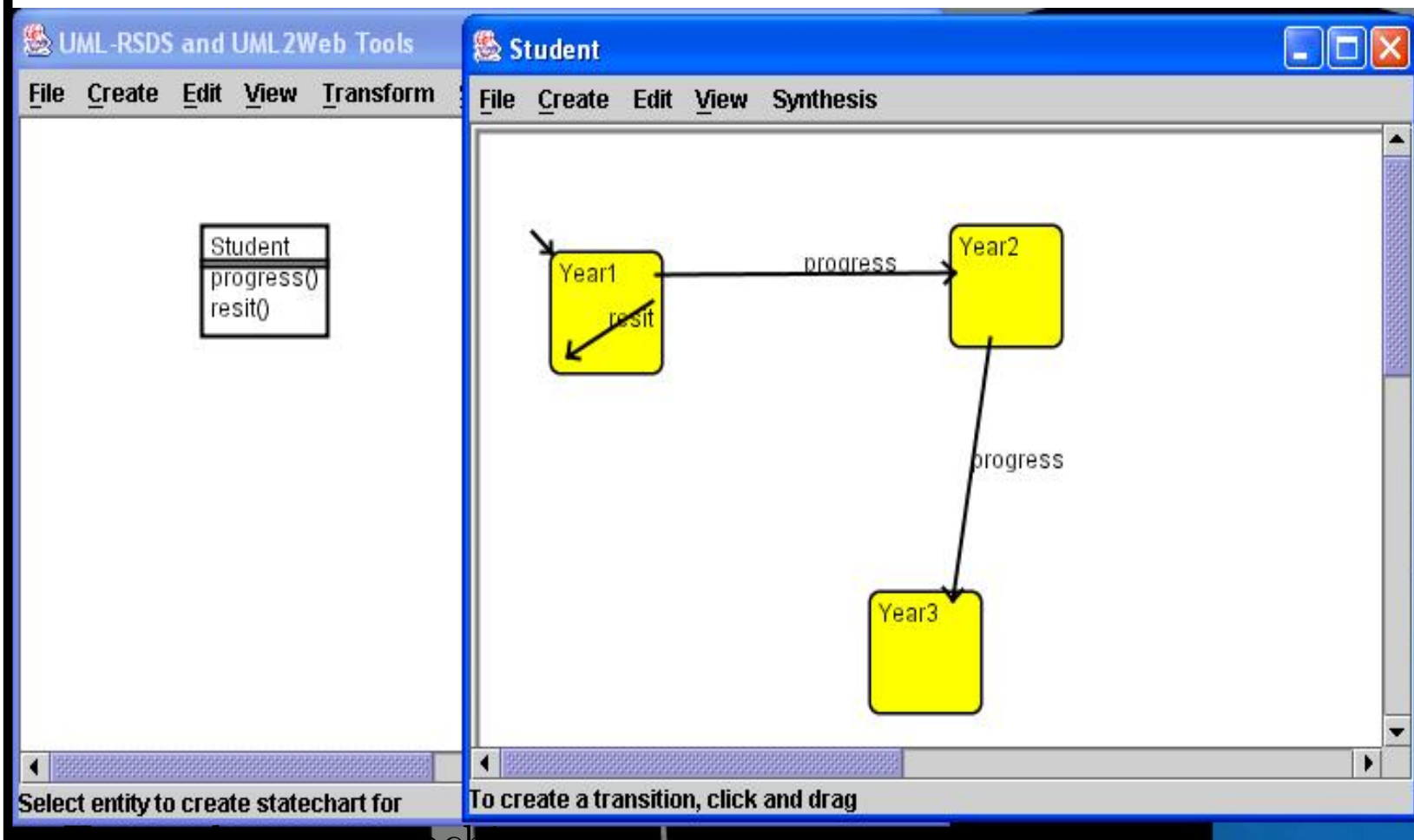
### *Use case diagrams*

- Define functionalities of system, the services it provides to users
- Each use case has name, written in oval, linked to agents/actors who interact with the case
- A textual description for each use case: its purpose, parameters, steps of its processing, result/output.

High level view of system and its services.

### *State machines*

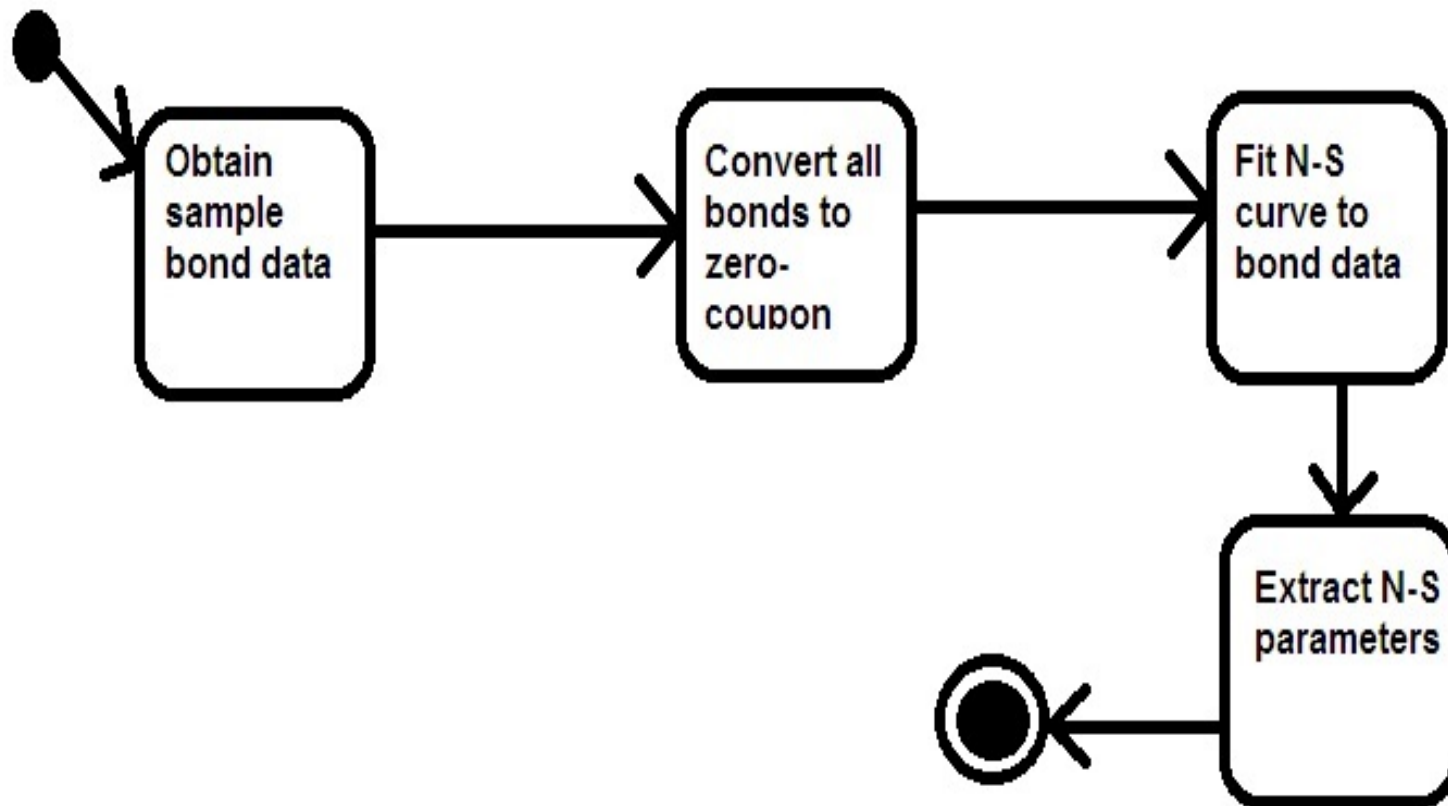
- State machines define dynamic behaviour of objects and operations.
- Can be used to define operations behaviour, and to express life histories of objects.
- For example, a student could have a linear lifecycle of successive states *Year1*, *Year2*, *Year3*.



Example state machine

### *State machines*

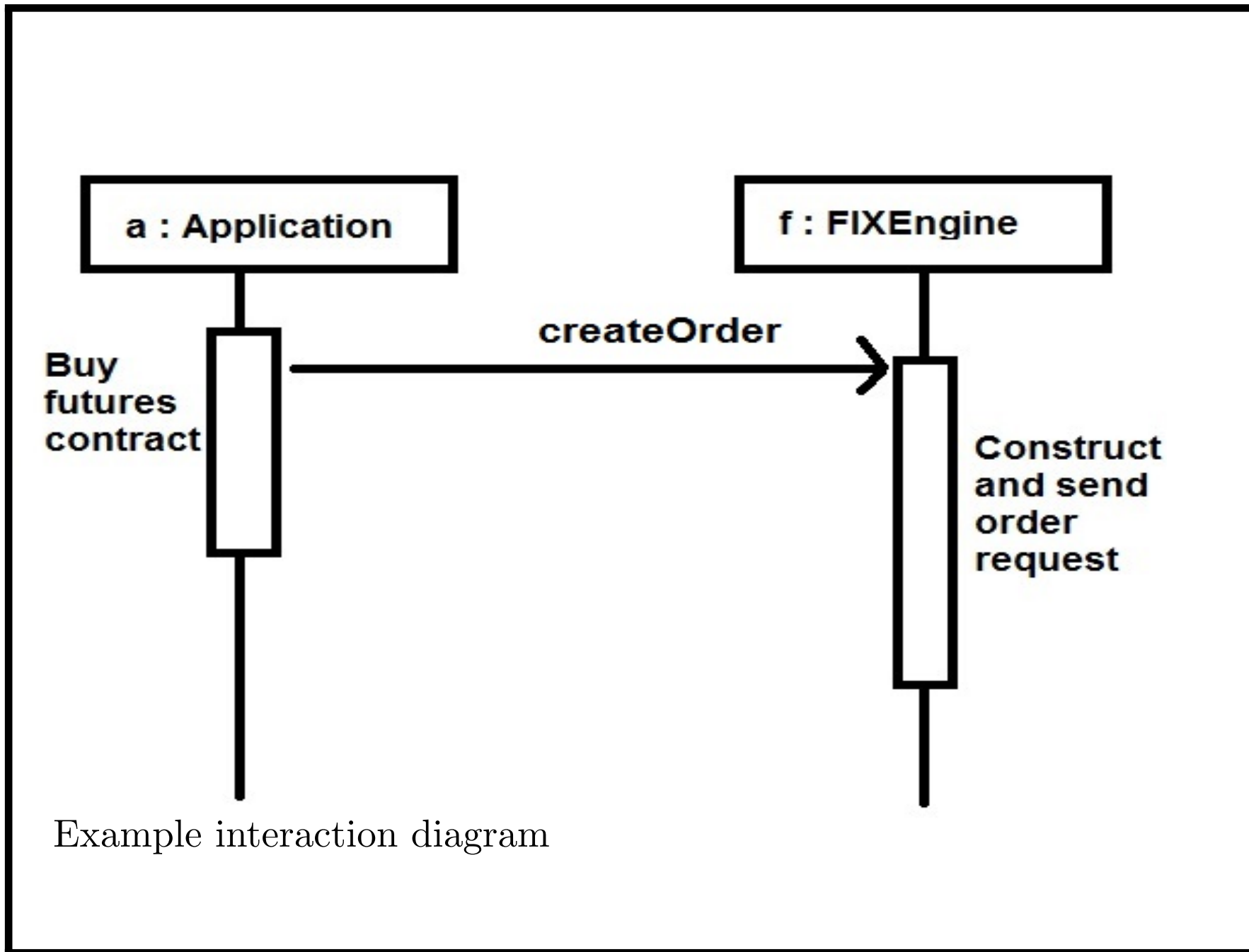
- Another example is procedure to obtain yield curve from bond market data
- From initial state (shown as black circle), process moves through states (processing stages) of obtaining market data, converting this to zero-coupon bond data, then using an optimisation procedure to fit a Nelson-Siegel curve to the data, and finally extracting parameters of this curve.
- Termination state shown as a ‘bullseye’ symbol.



Example state machine diagram

## *Interactions*

- Interactions, expressed as 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.
- Can be used to show test cases.



*Agile development methods: Scrum, XP*

- Agile software development introduced c. 2000 to correct drawbacks of conventional plan-based/heavyweight processes.
- Conventional development relies on gathering and formalising all requirements before starting coding; agile emphasises incremental work on requirements, coding, integration in short cycles. Responsive to changing requirements.
- In conventional development, team isolated from stakeholders: delays in obtaining information/feedback. Agile emphasises close collaboration between team and customer.
- Agile now widely adopted in industry; 2 main approaches: eXtreme Programming (XP) and Scrum.

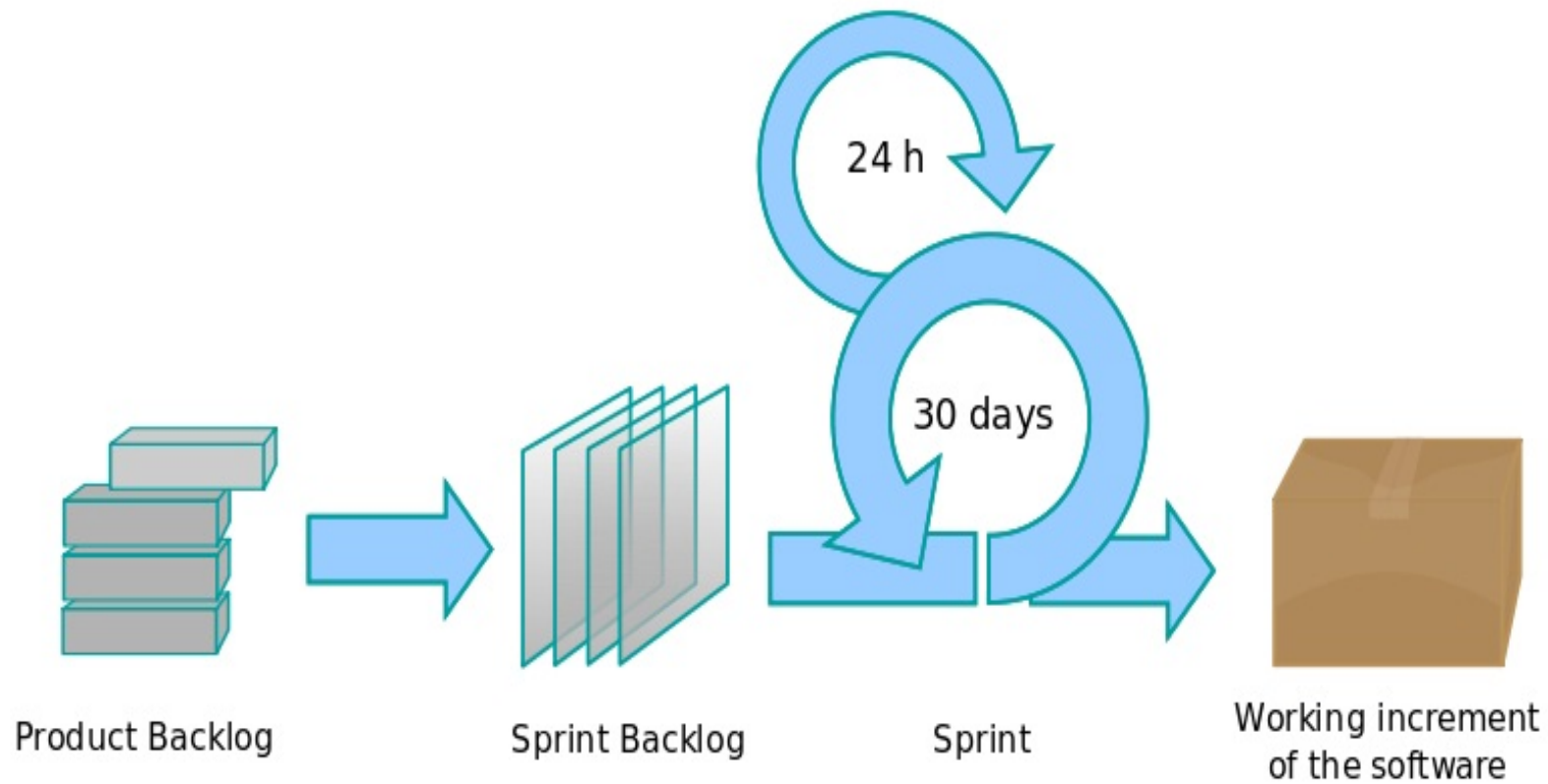


*Agile versus plan-based development*

<i>Agile development</i>	<i>Plan-based development</i>
Small/medium-scale	Large scale (10+ people)
In-house project, co-located team	Distributed/outsourced team
Experienced/self-directed developers	Varied experience levels
Requirements/environment volatile	Requirements fixed in advance
Close interaction with customer	Distant customer/ stakeholders
Rapid value, high-responsiveness required	High reliability/correctness required

### *Agile development techniques*

- *Sprints*: development work which implements specific user requirements, in short time frame to produce new release. “Deliver working software frequently”. Scrum methodology.
- *Refactoring*: Regularly restructure code to improve it, remove redundancy, etc. (ref: Fowler, “Refactoring: improving the design of existing software”). A key element of XP.



Scrum agile process

### *Agile development: Sprints*

- Sprints are regular re-occurring iterations in which project work is completed. Produce deliverables that contribute to overall project.
- Each iteration of system involves set of work items/tasks (eg., from user stories/epics) to be implemented. Tasks can be classified by business value to customer (high, medium, low), and by risk/effort by developer. High priority and high risk tasks should be dealt with first.
- Project *velocity*: amount of developer-time available per iteration.
- Taking these into account, can define *release plan*: which tasks will be delivered by which iteration and by which developers.

## *Agile methods: Scrum*

### Key events:

- *Sprint Planning*: performed by Scrum team before sprint, team agrees tasks to be worked on in sprint
- *Daily Scrum*: organises activities of team, review sprint progress + deal with issues. Time limited (eg., 15 mins). Key questions for developers: (i) what did I achieve yesterday? (ii) what will I achieve today? (iii) is there anything blocking me from achieving my work?
- *Sprint Review*: at end of sprint
- *Sprint Retrospective*: after sprint review, before next sprint planning. Analyse achievements of sprint, ideas for improvement of process.

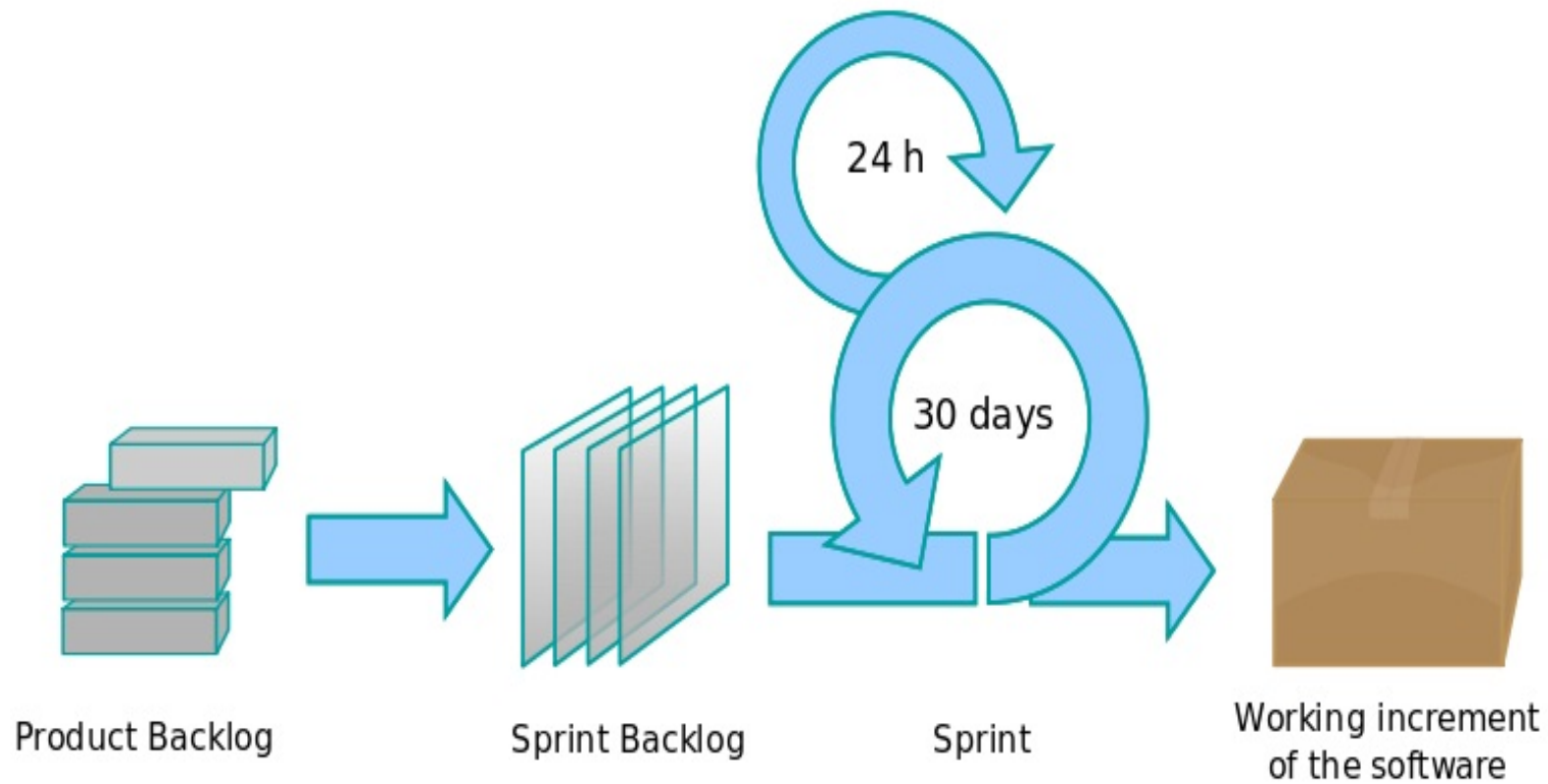
### *Agile methods: Scrum*

- *Story*: a generalised task (eg., develop/revise a system use case). Can consist of set of subtasks
- *Product Backlog*: ordered (in terms of priority) list of stories relevant to project
- *Sprint Backlog*: ordered list of stories to be completed in a sprint.

Team uses *Scrum Board* showing tasks to do, in progress and completed. *Burndown Chart* shows graph of remaining work against time.

Product Backlog	Iteration Backlog	In progress	Completed
task 9	task 6	task 2	task 1
task 10	task 7	task 4	task 3
task 12	task 8	task 5	
task 13	task 11		
task 14			
task 15			

Scrum board example



Scrum agile process



### *Agile methods: XP*

- 5 Values: *communication, simplicity, feedback, courage, respect.*
- 3 Principles: *Feedback* – via customer interaction, fine-grain testing; *Assuming simplicity* – code that's just good enough, small changes; *Embracing change.*
- 12 Practices: *pair programming; planning game; test-driven development; whole team; refactoring; small releases; system metaphor; simple design; continuous integration; collective code ownership; sustainable pace; coding standards.*

Appropriate for small teams (up to 12 people).

### *Other agile development techniques*

- *Test driven development (TDD)*: write unit tests first, then successively extend and refine code until it passes all tests.
- *Pair programming*: two developers work at same terminal, observer has role to improve programmers code.

TDD appropriate if no clear understanding of how to solve problem.

Pair programming can improve quality, and reduce time, whilst increasing cost.



Pair programming

## *Kanban*

- General manufacturing concept originating in Japanese car industry: only produce what is needed when it is needed; track demand through all production stages
- Unlike Scrum, oriented to a continuous delivery software production process
- Demand-led production, demand determines priority for work items
- Limit the work-in-progress (WIP): work only on few tasks – often just one – at a time
- When current task finished, start on next highest priority task from backlog

## *Kanban versus Scrum*

<i>Kanban</i>	<i>Scrum</i>
No prescribed roles	Roles of Scrum master; team member; product owner
Continuous delivery	Timeboxed sprints
Work pulled through system (single task flow)	Work performed in batches (sprint backlogs)
Environments with high variation in task priorities	Environments where tasks can be batched + worked on together

### *Kanban boards*

Similar idea to Scrum boards, may have columns such as *Backlog* (tasks to be done next); *In Development*; *Testing*; *Customer Acceptance*; *Done*.

Possible to have separate teams (eg, specification and design team, and coding and delivery team). Output of one team fed into the backlog of the next.

## *Benefits/disadvantages of Agile*

State of Agile Survey (versionone.com):

- Main benefits of agile: (i) management of changing priorities; (ii) team productivity; (iii) project visibility.
- Majority of respondents found agile projects were mainly successful.
- Scrum was main method used, with daily standups, short iterations, prioritised backlogs, iteration planning most common techniques.

Disadvantages of agile:

- Focussed on manual coding – resource intensive.
- Focussed on functional requirements; does not emphasise reuse.

### CHAOS RESOLUTION BY AGILE VERSUS WATERFALL

SIZE	METHOD	SUCCESSFUL	CHALLENGED	FAILED
All Size Projects	Agile	39%	52%	9%
	Waterfall	11%	60%	29%
Large Size Projects	Agile	18%	59%	23%
	Waterfall	3%	55%	42%
Medium Size Projects	Agile	27%	62%	11%
	Waterfall	7%	68%	25%
Small Size Projects	Agile	58%	38%	4%
	Waterfall	44%	45%	11%

The resolution of all software projects from FY2011–2015 within the new CHAOS database, segmented by the agile process and waterfall method. The total number of software projects is over 10,000.

Project success statistics (Standish group, 2015)



### *Barriers to adoption of Agile development*

- Inability to change organisational culture
- Lack of experience with agile methods
- Company philosophy/values conflict with agile
- Established rigid/Waterfall development approach
- Lack of management support/ management concerns about lack of up-front planning
- External requirements to follow conventional staged process
- Lack of access to users/customers.

## *Model-driven Engineering (MDE)*

- Uses UML or other modelling language to develop systems based on models, instead of code.
- MDD (Model-driven development), MDA (Model-driven architecture) are specific MDE approaches which make models primary artifact.
- Benefits: models are simpler to review + modify than code; code generation can raise productivity substantially; model repositories and libraries can be established for rapid production of systems in a common ‘product line’ family. Reduces need for outsourcing.
- Has been adopted in specific industries, with generally positive results (Whittle et al, IEEE Software, 2014).
- But training + adoption costs, and poor tool support.

### *Financial models*

- Financial models are mathematical models of financial products/processes
- Expressed using equations, including differential and stochastic differential equations
- Tools such as MathApp enable financial model specification, analysis and implementation.

MathApp: <https://agilemde.co.uk/mathapp.zip>



MathOCL Editor

File Edit Style Analysis Translate Help

$\exists$	$\forall$	$\in$	$\notin$	$\emptyset$	$\bullet$	$\approx$	$\rightarrow$	$\Sigma$	
$\mathbb{R}$	$\alpha$	$\beta$	$\gamma$	$\delta$	$\epsilon$	$\zeta$	$\theta$	$\lambda$	

specification S

Define principal = 100

Define price = 97.5

Define term = 1.0

Simplify principal/price =  $e^{\text{yield} \cdot \text{term}}$

Specification of zero-coupon bond

specification S

Define principal = 100

Define price = 97.5

Define term = 1.0

Simplify  $\text{principal}/\text{price} = e^{\text{yield} \cdot \text{term}}$

---

**specification S**

**Define principal = 100**

**Define price = 97.5**

**Define term = 1**

**Simplify  $\log(\text{principal}/\text{price}) = \text{yield} \cdot \text{term}$**

Zero-coupon bond analysis (1)

---

specification S

Define principal = 100

Define price = 97.5

Define term = 1.0

Simplify principal/price =  $e^{\text{yield} \cdot \text{term}}$

---

**specification S**

**Define principal = 100**

**Define price = 97.5**

**Define term = 1**

**Solve  $\log(\text{principal}/\text{price}) = \text{yield} \cdot \text{term}$  for yield**

Zero-coupon bond analysis (2)

specification S

Define principal = 100

Define price = 97.5

Define term = 1.0

Simplify principal/price =  $e^{\text{yield} \cdot \text{term}}$

---

**specification S Define principal = 100**

**Define price = 97.5**

**Define term = 1**

**Define yield =  $\log(\text{principal}/\text{price})/((-1.0 \cdot (-\text{term})))$**

Zero-coupon bond analysis (3)

*Case studies in the course*

- Monte-Carlo simulation
- Internal rate of return
- Yield curve estimation
- Data analytics of stock prices.



### *Summary of Part 1*

- Have introduced the concepts of software engineering, software modelling and software life cycle.
- Introduced UML notations
- Introduced MDE and Agile development.

### *Useful references*

- K. Beck, *Extreme programming explained: Embrace change*, Addison-Wesley, 2000.
- Agile manifesto: [www.agilemanifesto.org](http://www.agilemanifesto.org)
- M. Fowler et al, *Refactoring: improving the design of existing code*, Addison-Wesley, 1999.
- M. B. Nakicenovic, *An Agile Driven Architecture Modernization to a Model-Driven Development Solution*, International Journal on Advances in Software, vol 5, nos. 3, 4, 2012, pp. 308–322.