Systems Design and Security

Part 1: Software Engineering

http://staffwww.dcs.shef.ac.uk/people/A.Simons/

Home ⇒ Teaching ⇒ Lectures ⇒ COM2008/COM3008

The University Of Sheffield.

Bibliography

- Software Engineering
  - I Sommerville, Software Engineering, 10th ed., Pearson, 2016.
  - R S Pressman and B R Maxim, Software Engineering: A Practitioner's Approach, 9th ed., McGraw-Hill, 2019.

Outline

- Software engineering definitions
- History of software engineering
- Famous myths and software failures
- Software lifecycle activities
- Traditional software process models
- Radical software process models
- Software component-based industry

Reading: Sommerville chapters 1-3, 16
Pressman chapters 2-4

What's the Difference?

- Computer Science
  - develops theory, methods, limits of what is computable
  - algorithms, data structures, formal grammars, abstract machines, numerical analysis
- Software Engineering
  - applies theory, methods to solve practical problems
  - creates complex software products
  - follows a design, management process
  - uses models of systems (diagrams, specifications)
  - deals with people (stakeholders, managers)

Software Engineering…

- is the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software.

  [IEEE Standard Glossary of Software Engineering Terminology, IEEE std 610.12, 1990]

- is an engineering discipline that is concerned with all aspects of software production.

  [I Sommerville, 2016]

- is the establishment and use of sound engineering principles in order to economically obtain software that is reliable and works efficiently on real machines.

  [F L Bauer, NATO Conf. Softw. Eng., Garmisch, 1968]

Some History…

- NATO Conference, Garmisch, Germany, 1968
  - convened to discuss the "software crisis"
  - more complex systems on 3rd generation hardware
  - first coined the term "software engineering"
- The software crisis
  - delays in software delivery
  - higher costs than originally estimated
  - software unreliable, difficult to maintain
- Need for new methods
  - engineering discipline (apply theories within real constraints)
  - cover all aspects of production (technical and management)

Software Myths

- Management myths
  - Follow published standards for building software
  - Use state of the art tools to build software
  - Add more programmers if behind schedule
- Customer myths
  - General description of objectives is enough to start coding
  - Requirements may change as the software is flexible
- Practitioner myths
  - Task is accomplished when the program works
  - Quality assessment only when the program is running
  - Working program is the only project deliverable
- Continuing "software affliction" [Pressman, 1997].

Software Failures

- Therac-25 (1985-1987): six people overexposed during treatments for cancer
- Taurus (1993): London Stock Exchange automated transaction settlement system cancelled after 5 years in development
- UK Passport Office (2006): project cancelled after 10 years, wasting £12m, wrecking 500 holidays; due to lack of testing and change control

  [http://infotech.fanshawec.on.ca/gsantor/Computing/FamousBugs.htm]

Technical Failures

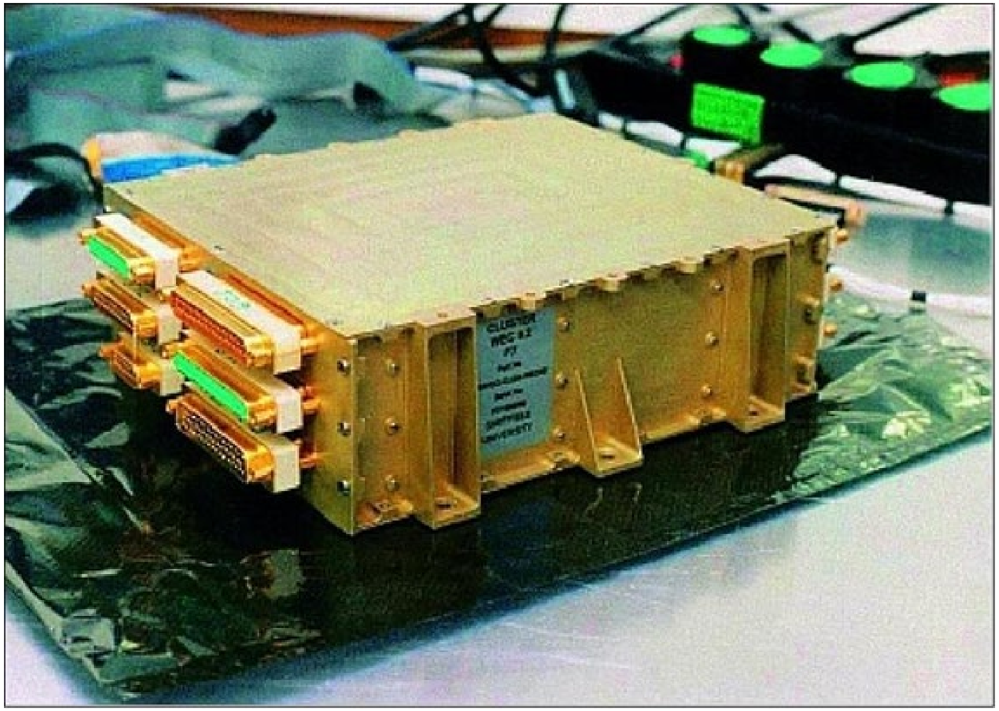
- Ariane 5 (1996): explodes 39 sec. into launch due to an error converting from floating point into 16-bit integer
- Costs $7bn and 10 years of effort – carried the original Cluster satellite mission
- Real cause: wrong assumption when reusing Ariane 4 guidance package

Click image for video

What we Lost!

Digital Wave Processor

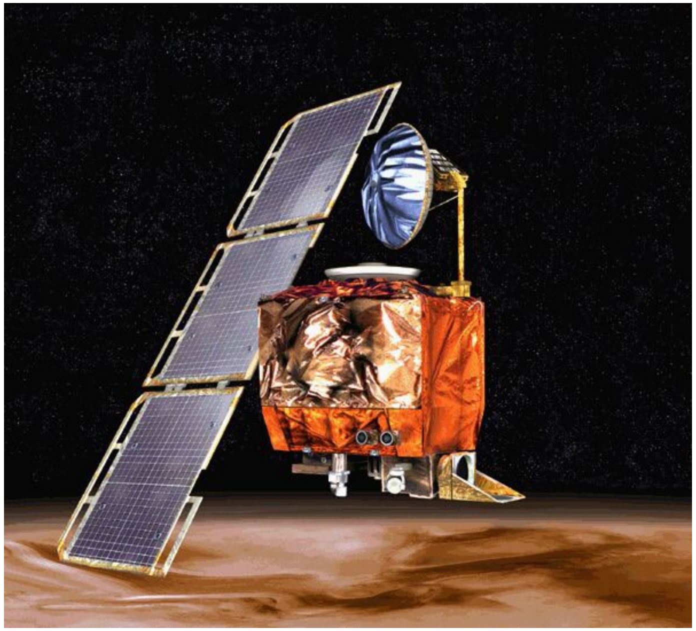
DCS built a high reliability multiprocessor unit based on the INMOS Transputer with parallel processing and re-allocatable tasks. Controlled all scientific instruments for data compaction, compression, event selection, and particle/wave correlation.



© University of Sheffield

Systemic Failures

- Mars Climate Orbiter (1999): lost by NASA flight controllers during orbital insertion
- Different measurement units (imperial and metric) for guidance and propulsion
- Real cause: lack of communication between software teams



Some Progress…

- Increased ability to produce more complex software systems (modular, component-based)
- Effective methods to specify, design, implement software have been developed (formal models, code generation)
- A better understanding of the activities involved in software development (people issues – psychology, politics)
- Novel software engineering approaches (clean room, time-boxed prototyping, test-first design)
- Standard notations and tools have been produced (UML, CASE tools)

Lab 1: Software and Risk

- Other famous failures?
  - Can you name recent interesting cases?
  - What dangers and risks are we running?
- What main cause of failure? (rank 1-3)
  - weak developer skills/coding practices
  - poor team communication
  - poor project/risk management
- What legal/social changes?
  - How will the profession of Software Engineer change?

Run a Poll

Kinds of Software

- System software
- Real-time software
- Business software
- Engineering and scientific software
- Embedded software
- Personal computer software
- Web-based software
- Artificial Intelligence software
- Research software

Software is Everywhere

- Business information systems
  - Banking, finance, purchasing, office data systems
- Embedded systems
  - Internet, telecoms, medical, industrial processes
- Lifestyle/entertainment
  - TV, video, film, games, publishing, social networks
- Science and engineering
  - Genetics, robotics, education, military ...
- Software is...
  - the most complex of any engineered human artifact

Software Process

A software process consists of a *set of activities* and associated results which lead to the production of a software product [Sommerville, 2016]

Four fundamental activities:

- Software specification – requirements, formalisation
- Software development – design, implementation
- Software validation – validation, verification, testing
- Software evolution – bug fixes, upgrades, adaptation

Software involves: Products – Processes – Models – People

Software Lifecycle

- What kinds of activity?
- Commission a software system
  - Feasibility study
  - Requirements elicitation
- Construct the software system
  - Systems analysis, formal specification and design
  - Implementation, testing and deployment
- Maintain the software system
  - Fix faults, extend the system
  - Decommission the system

Lifecycle Activities – I

- Feasibility Study
  - Outline the objectives of the business
  - Met by the software system, or by other means?
- Requirements Elicitation
  - Collect required system behaviours from stakeholders
  - Use interviews, workshops; get different viewpoints
- Systems Analysis
  - Determine the scope, interface to other systems
  - Break down into models of: data, processing, time
  - Model what the system has to achieve (not how)

Lifecycle Activities – II

- Formal Specification
  - Logical model and proof of expected behaviour
  - For safety-, mission-, business-critical systems
- Systems Design
  - Split into units, modules and subsystems
  - Choose between alternative design strategies
  - Model how the system achieves its goals
- Implementation
  - Select technology: programming language, web script
  - Construct units, modules and integrate the system

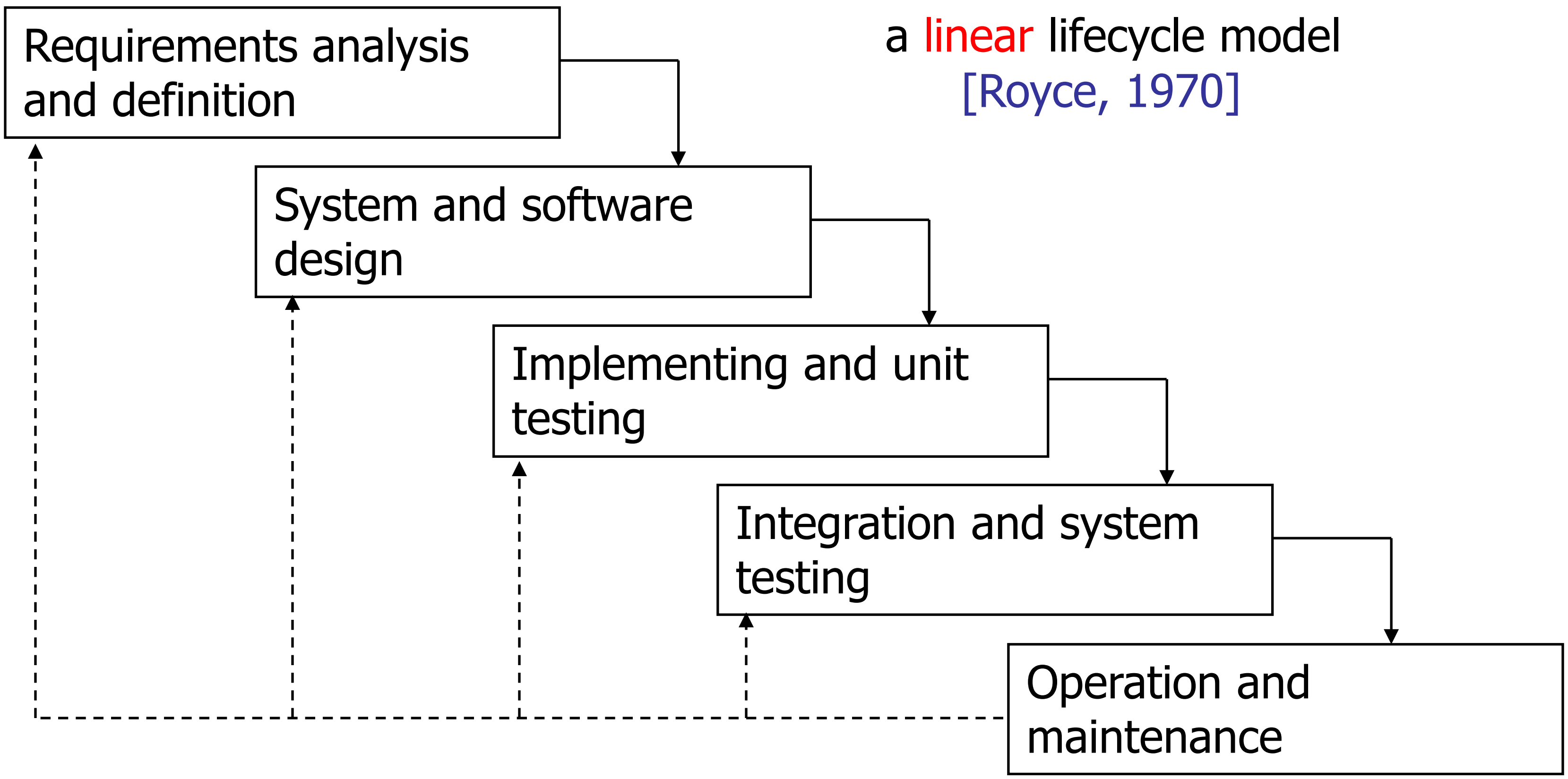
Lifecycle Activities – III

- Testing
  - Unit testing, integration testing (up to specification)
  - Path coverage, stress-, load-testing (no failures)
  - Acceptance testing by users (up to requirements)
- Deployment
  - Install, configure system (at multiple sites?)
  - Operate the system
- Maintenance
  - Fix faults discovered late in the system
  - Add new functionality (new requirements)
  - Decommission the system (outlived its usefulness)

Software Process Models

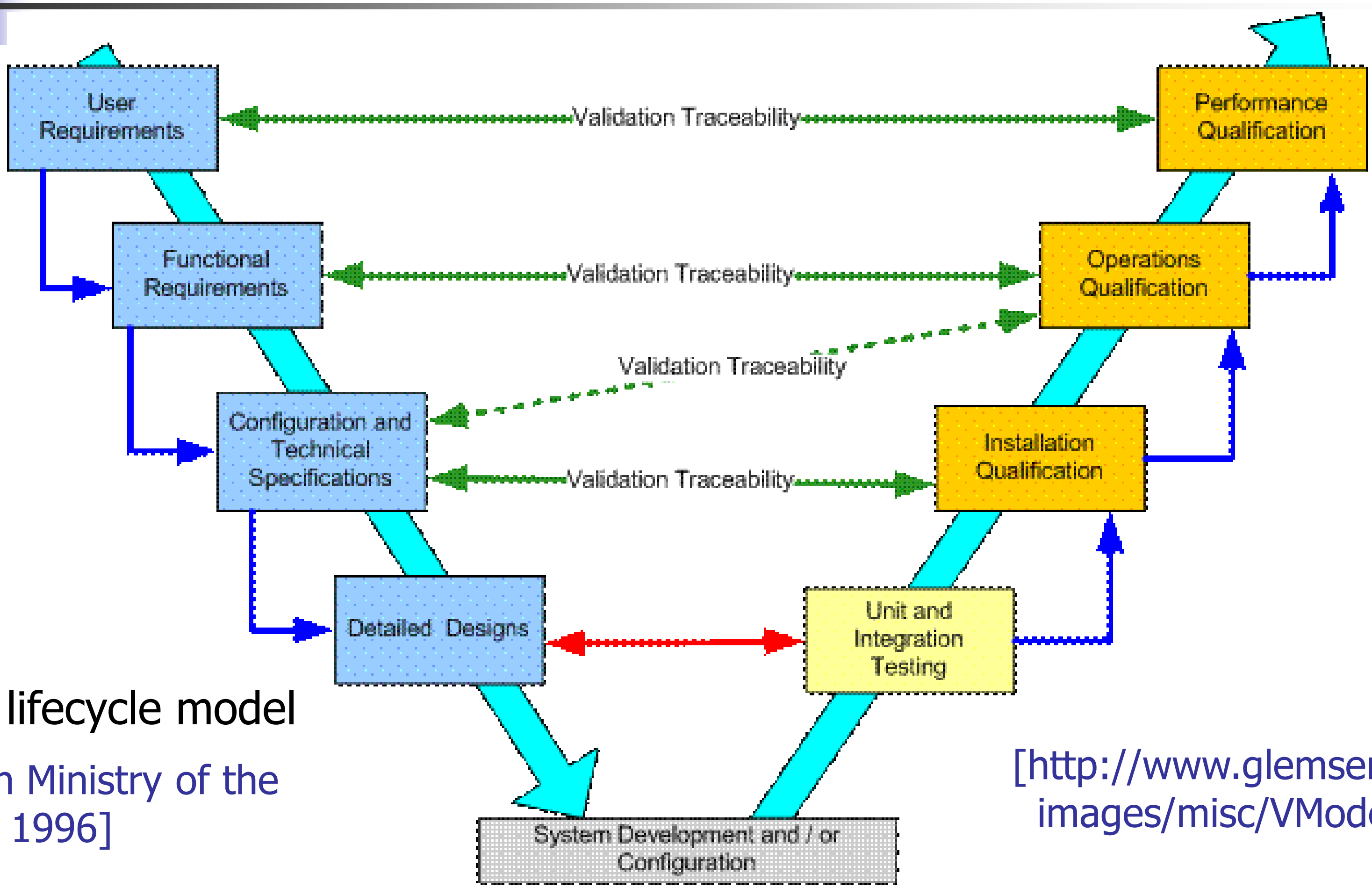
- Traditional software process models
  - Waterfall software lifecycle model [Royce, 1970]
  - Spiral software lifecycle model [Boehm, 1988]
  - V-model of the software lifecycle [Germany, 1996]
- Formal software process models
  - Cleanroom [IBM]
  - VDM or Z/B with refinement
- Radical software process models
  - Prototyping models (rapid- ; evolutionary-)
  - Time-boxed models (Rapid Appl. Dev., DSDM)
  - Agile methods (eXtreme Programming, Scrum)

Waterfall Lifecycle Model



a linear lifecycle model
[Royce, 1970]

German V-Model



a linear lifecycle model

[German Ministry of the Interior, 1996]

[http://www.glemser.com/images/misc/VModel.gif]

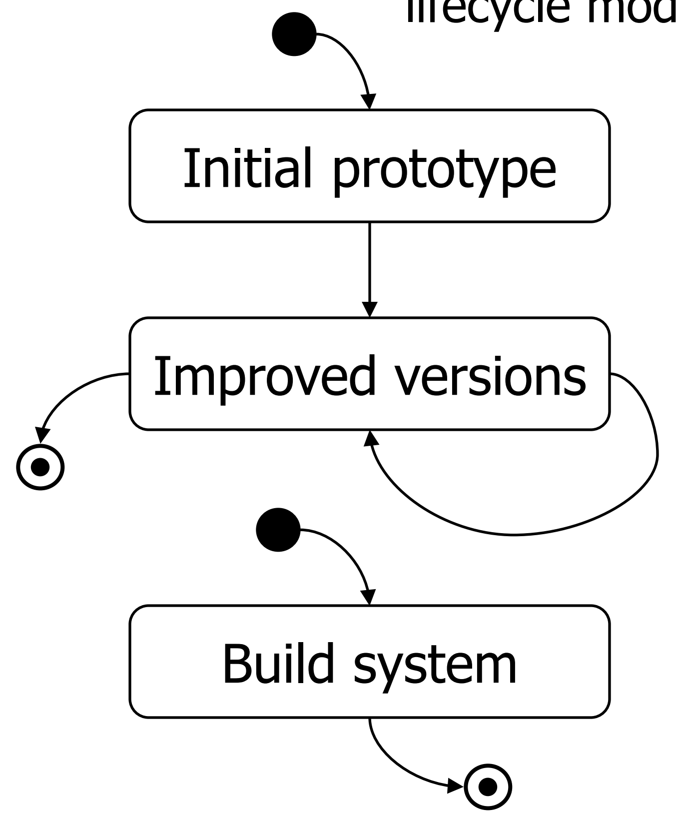
Waterfall: Evaluation

- Strong points
  - staged development reflects engineering practice
  - encourages a discipline of abstract modelling, to break down complex systems into parts, views
  - good for project management, coordination, documentation
  - V-model matches each specification stage with a validation stage, reducing cost of faults discovered early
- Weak points
  - linear process is inflexible – in practice, stages overlap
  - imposes early commitment to rigid requirements
  - difficult to deal with late changes – increasing costs
  - real systems evolve faster than one complete cycle

Rapid Prototyping

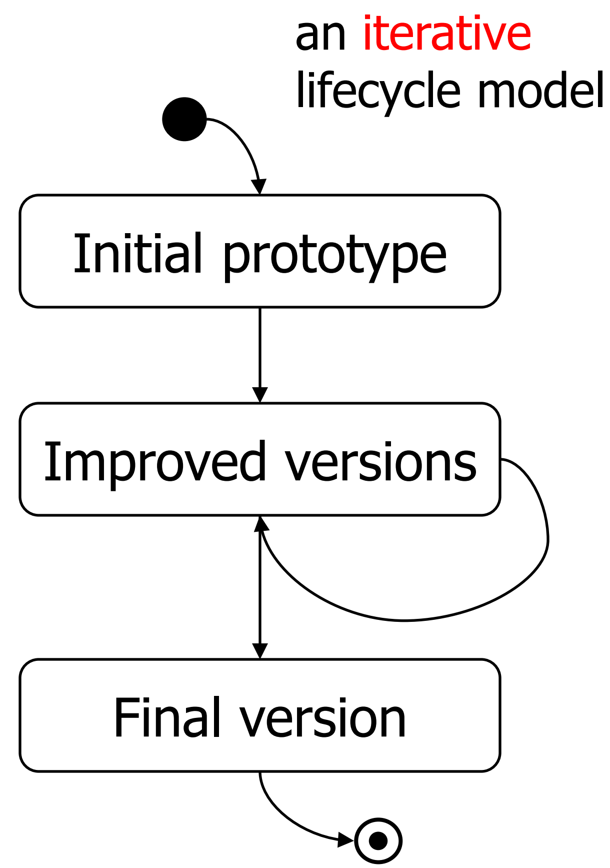
- Reasons to adopt
  - Requirements are vague or poorly understood
  - Build a prototype to help elicit requirements
  - Collect feedback
- Software lifecycle
  - Build a basic prototype
  - Iterate the prototype, add or change features
  - When acceptable, discard and build finished system

an iterative lifecycle model



Evolutionary Prototyping

- Reasons to adopt
  - Requirements are vague or misunderstood
  - Time constraints may force early termination
- Software lifecycle
  - Build a basic prototype
  - Evolve the prototype towards finished system
  - Stop when the system is ready, or the time limit is exceeded



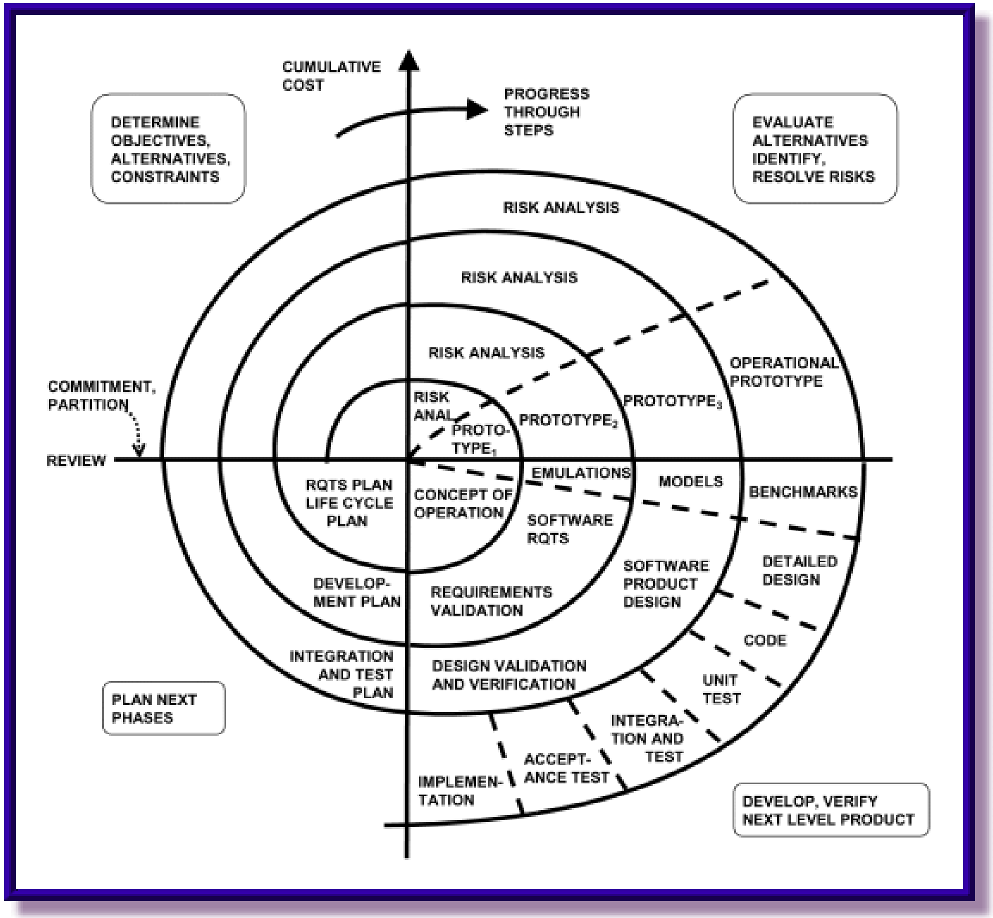
Prototyping: Evaluation

- Strong points
  - Helpful when requirements are not understood
  - Flexible, for small-to-medium sized systems
  - Short turnaround, accommodates late changes
  - Prototype can be discarded, or evolve to final version
- Weak points
  - Development process is not visible
  - Evolution creates poorly structured systems
  - Special tools, special techniques required
  - Cannot use with large-scale projects requiring coordination

Spiral Lifecycle Model

an iterative lifecycle model

[Boehm, 1988]



[http://www.stsc.hill.af.mil/crossTalk/2001/05/boehm.html]

Spiral Model: Features

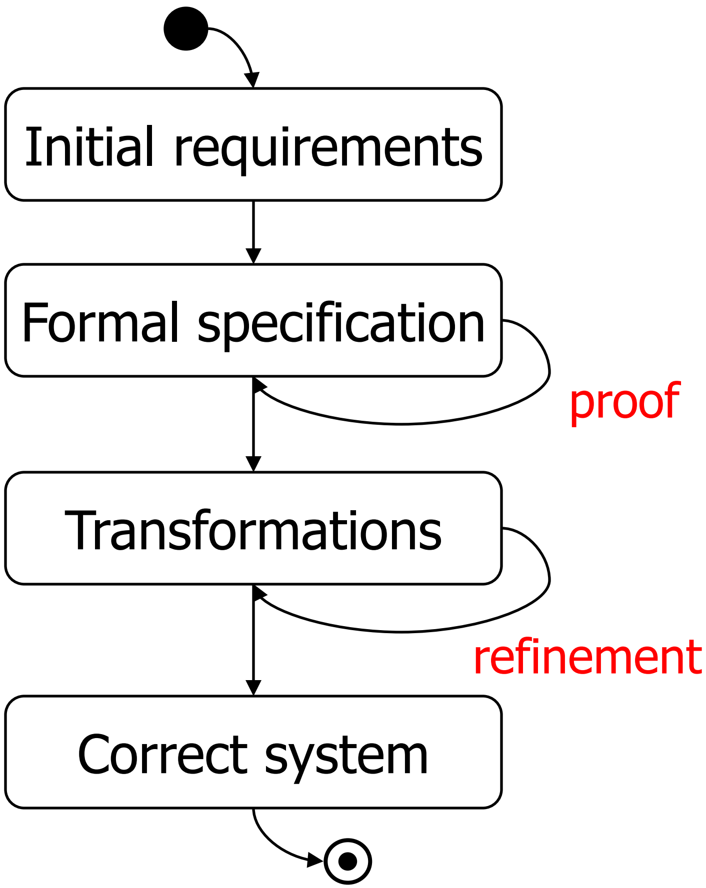
- Reasons to adopt
  - Key idea is risk evaluation and reduction
  - Each cycle is a stage in the software process (here, requirements, analysis, design, implementation)
  - Other stages possible (eg interface, database, middleware)
  - Can choose to terminate after each risk evaluation
- Four stages
  - Set specific objectives for the current stage
  - Evaluate risks, explore alternative approaches
  - Develop and validate the current stage
  - Plan the next stage, when requested

Spiral Model: Evaluation

- Strong points
  - Combination of waterfall and prototyping approaches
  - Adapts to different projects with different stages
  - Clear software process for large-scale projects, iterative cycles last 6 months – 2 years
  - Clear evaluation of each stage, determines whether to proceed, or cut losses due to high risk
- Weak points
  - Projects can still be terminated before delivery
  - Late changes can still have serious consequences

Formal Systems Development

- Reasons to adopt
  - Safety-critical, mission-critical systems
  - Medical treatment, aircraft, traffic control systems, etc.
- Software lifecycle
  - Construct formal spec. from initial requirements
  - Prove safety properties of the specification
  - Use refinement rules to transform into a correct implementation



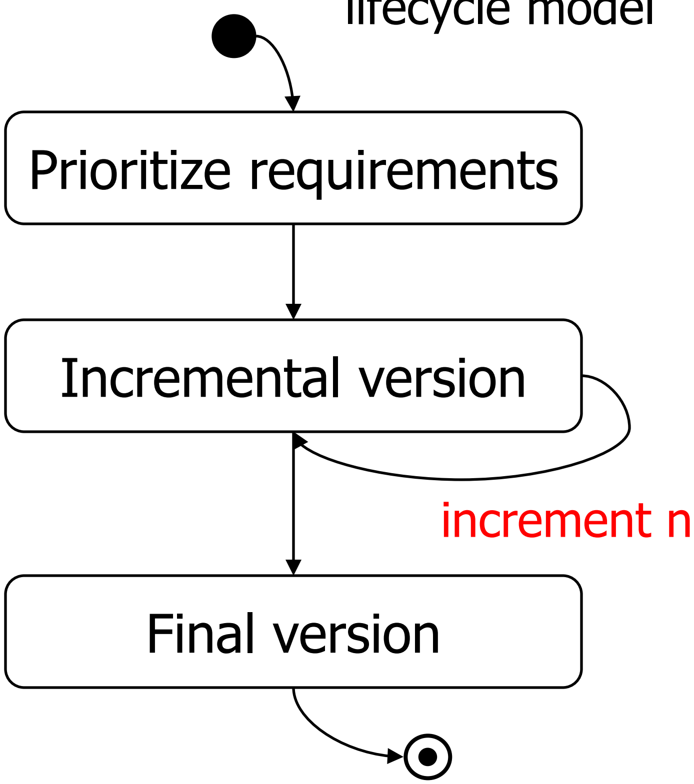
Formal Systems: Evaluation

- Strong points
  - Precise and error-free: correct by construction
  - Used for safety-critical, mission-critical systems
  - Proofs of correct specification (complete, consistent)
- Weak points
  - Requires specialised mathematical expertise
  - Perceived as difficult, or costly (but counter-examples)
  - Must still validate formal models against user requirements
  - Time-to-market is more compelling than correctness!
  - Non-formal approaches often seem adequate

Incremental Development

- Reasons to adopt
  - Time-to-market is critical
  - Must deliver something
- Software lifecycle
  - Sort requirements into prioritized increments
  - Deliver each increment within fixed time-box
  - Stop after the agreed number of iterations (often 4: DSDM)

an iterative lifecycle model



Incremental: Evaluation

- Strong points
  - Reduces gap between specification and delivery
  - Much lower risk of total project failure
  - Early increments can be prototypes
  - Highest priority services are delivered first
- Weak points
  - Increments have to be small, to deliver within the time-boxed period (avoid "feature creep")
  - Hard to map large global requirements onto increments
  - Short time-box (RAD, DSDM) works best when requirements are well-understood and modular

Lab 2: Which Lifecycle?

Run a Poll

- Your client is a healthcare SME (small company)
  - want a ground-breaking software system to learn the links between disabilities and the kinds of equipment prescribed by doctors.
- Your client is the Daimler AG automotive company
  - want to coordinate the release of their on-board engine monitoring system with matching engine maintenance systems at garages.
- Your client is Boeing Commercial Airplanes
  - want the latest fly-by-wire and on-board entertainment systems for the next-generation Dreamliner airplane.
- Your client is the UK National Health Service
  - want a standardised national healthcare record system, but don't yet know how far the government budget will stretch.

Shift to Components

- Bespoke structured systems
  - 1970s: systems made-to-measure (at any cost!)
  - older structured languages: Fortran, Cobol
  - clear top-down design and stepwise refinement
- Component-based systems
  - 1990s: mix-and-match systems (fast, cheap)
  - object-oriented, component-based development
  - clear emphasis on component reuse
  - mixture of top-down design, bottom-up assembly

The University Of Sheffield.

Component-Based Development

- Requirements phase
  - Collect system requirements as a set of tasks (top-down), or user-interactions: use cases (bottom-up)
  - Analyse requirements to map onto software components
- Construction phase
  - Build as much of the system as possible from existing components, or frameworks (component architectures)
  - Develop new components and integrate within existing framework
- Reuse/refactoring phase
  - Schedule framework for maintenance (refactoring – improve structure) and harvest any new reusable components

Changing Practices

Conan the Librarian

© Universal Pictures

- How to reward developers?
  - KLOC? – promotes code bloat!
  - code reuse? – needs a proper market
  - elegance? – how to measure effectively?
- New roles in software house?
  - developer – focus just on the application
  - facilitator – finds needed components in the library
  - scavenger – harvests components for the library
  - librarian – defends library from unguarded modifications

Components: Evaluation

- Strong points
  - Rapid development strategy, promotes reuse
  - Greatly reduced time-to-market (from 3rd project)
  - Reduces costs, and risks
- Weak points
  - High initial set-up costs (building frameworks, components)
  - Retrain personnel to fit new roles (eg: librarian)
  - Can lose control over the component set (adaptation)
  - Systems are sometimes forced to fit the framework and so don't deliver as expected

Agile Methods (XP)

- Agile methods philosophy
  - No "big design up-front" – just "user stories"
  - The production code base and saved tests are the only lasting deliverables
  - Short increments, daily builds, pass all tests
  - XP has "pair programming": coder and critic
  - Anyone can modify the code base (extreme!)
  - Embrace change (extreme!)
- Examples
  - eXtreme Programming (XP) [Beck, 2000]
  - Scrum [Sutherland and Schwaber, 1996]

Agile Methods: Evaluation

- Strong points
  - Avoids unproductive, wasted documentation
  - Emphasis on testing (= meeting specification)
  - Developers enjoy the lightweight approach
  - System is always up-to-date
- Weak points
  - Inflexible process, cannot adapt to organisations
  - Expects good design to emerge by evolution
  - In practice, may fail for large systems that require coordination – no change tracking documentation

Shift to Services

- Bespoke systems
  - 1970s: large monolithic systems serving a single purpose, each a one-off design
- Component-based systems
  - 1990s: systems broken down into mix-and-match reusable components
  - affected creation, not deployment; software still sold as shrink-wrapped releases
- Service-oriented systems
  - 2000s: early web-wrappers for ERM software
  - 2010s: components converted into service bundles, rented in the cloud

Enterprise Resource Management

Services – Evaluation

- Strong points
  - Purchase only those services you need, rather than a whole shrink-wrapped package
  - No need to re-purchase the latest upgrades, since rented services are always up-to-date
  - Converts capital expenditure (CAPEX) into operating expenditure (OPEX)
- Weak points
  - Web layer adds further complexity, many interfaces between service components
  - Additional Internet security issues, especially with JavaScript
  - Cloud platforms with large ecosystem of providers; very hard to assure quality control

Summary

- Software engineering covers all aspects of software production: technical, management process
- Software engineering includes software products, processes, models and people
- The software lifecycle consists of: requirements elicitation, analysis, design, implementation, testing, maintenance…
- Traditional process models are linear, good at coordinating large teams, but weak when handling change
- Radical process models are iterative, good at adapting to change and delivering, but weak at coordination
- The shift to software component technology requires a shift in the software development strategy