Software engineering Facts

- **Fact**: The economies of ALL developed nations are dependent on software.
- **Fact**: More and more systems are software controlled
- **Fact**: Expenditure on software represents a significant fraction of GNP in all developed countries.
- **Fact**: Software often costs more than the computer it runs on.
- **Fact**: Software costs more to maintain than to develop

What is software?

Software is:
- Computer programs
  - Source code
  - Executables, binaries, runtimes
- Associated documentation
  - Requirements
  - Design models
  - User manuals

What is software engineering?

- Software engineering (SE) is the **design**, **development**, and **documentation** of software by applying technologies and practices from computer science, project management, engineering, application domains, interface design, digital asset management and other fields.
- Term was invented in 1968
- Used to be called “programmer” or “systems analyst”

Why do we need Software Engineering?

- Software is big business
- Bad software is expensive to a company
- Stakes are very high
- Having a good plan and good process improves chances for success
- Lots of high paying jobs in software

Software Development Processes
The software process

- A structured set of activities required to develop a software system
  - Specification;
  - Design;
  - Validation;
  - Evolution.
- A software process model is an abstract representation of a process. It presents a description of a process from some particular perspective.

The old way

- The way software is engineered has evolved over the years
- The “new” ways of software engineering resemble the “old” ways in a lot of ways
- See if you can make out the resemblance

Waterfall Model

- Inflexible partitioning into distinct stages makes it hard to deal with changing customer requirements.
- Only works when requirements are well-known and few changes are expected… which is rare!
- The waterfall model is still used for some large, multi-site projects, but used less and less

Evolutionary development

- Problems
  - Throw-away prototyping might waste work
  - Lack of process visibility
  - Systems are often poorly structured, evolve
  - Special skills (e.g. in languages for rapid prototyping) may be required
- Applicability
  - For small or medium-size interactive systems;
  - For parts of large systems (e.g. the user interface);
  - For short-lifetime systems.
Incremental development

- System requirements ALWAYS evolve in the course of development
- Iteration can be applied to any of the generic process models
- Break down software into "releases", deliver incrementally (version 1.0, version 2.0, etc.)
- "Freeze" requirements during a release

Process Iteration & Incremental Delivery

Spiral model of the software process

- Process is represented as a spiral rather than a sequence of activities with backtracking.
- Each loop in the spiral represents a phase in the process.
- No fixed phases such as specification or design - loops in the spiral are chosen depending on what is required.
- Risks are explicitly assessed and resolved throughout the process.

Spiral development

The new way

- Agile – it’s Spiderman at the keyboard
- Extreme – it’s like totally radical
- Scrum – what’s rugby got to do with it?

The Agile Approach (1)

- Continuous delivery of software
- Cycle is weeks rather than months
- Working software is the principal measure of progress
- Even late changes in requirements are welcomed
- Close, daily, cooperation between business people and developers
- Face-to-face conversation is the best form of communication.
The Agile Approach (2)

- Projects are built around motivated individuals, who should be trusted
- Continuous attention to technical excellence and good design
- Simplicity
- Self-organizing teams
- Regular adaptation to changing circumstances
- From the Agile Manifesto (Google it) (HANDOUT)

Extreme programming

- Perhaps the best-known and most widely used agile method.
- Extreme Programming (XP) takes an 'extreme' approach to iterative development.
  - New versions may be built several times per day
  - New version delivered every 2 weeks
  - All tests run with each build, all must pass
  - Doesn't reinvent the wheel – use COTS whenever possible and affordable

Extreme programming practices 1

- Incremental planning: Requirements are rounded as story cards and the stories to be included in a release are determined by the time available and the relative priority. The developers break these stories into development tasks.
- Small releases: The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.
- Simple design: Enough description is carried out to meet the current requirements and no more.
- Test first development: An automated test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.
- Refactoring: All developers are expected to refactor the code immediately as soon as possible and improvements are found. This keeps the code simple and maintainable.

The XP release cycle

![XP Release Cycle Diagram]

- Select user stories for this release
- Break down stories into tasks
- Plan release
- Evaluate system
- Developing more/test software
- Release software

Extreme programming practices 2

- Pair programming: Developers work in pairs, checking each other's work and providing the support to always do a good job.
- Collective ownership: The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers own all the code. Anyone can change any thing.
- Continuous integration: As soon as work on a task is completed it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
- Sustainable pace: Large amounts of overtime are not considered acceptable so the net effect is often to reduce code quality and reduce team productivity.
- On-site customer: A representative of the end-use of the system (the Customer) should be available full-time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for integration.

Problems with agile methods

- It can be difficult to keep the interest of customers who are involved in the process.
- Team members may be unsuited to the intense involvement that characterises agile methods.
- Prioritizing changes can be difficult where there are multiple stakeholders.
- Maintaining simplicity requires extra work.
- Contracts may be a problem as with other approaches to iterative development.
Testing in XP

- Test-first or Test-driven development (TDD)
- For each and every component (class, module, whatever) you develop, add one or more tests at the same time
- Building means compiling the code and running all the tests, automatically
- Keeps software working all the time

Pair programming

- In XP, programmers work in pairs, sitting together to develop code.
- This helps develop common ownership of code and spreads knowledge across the team.
- It serves as an informal review process as each line of code is looked at by more than 1 person.
- It encourages refactoring as the whole team can benefit from this.
- Measurements suggest that development productivity with pair programming is similar to that of two people working independently.

SCRUM Approach

- Backlog – list of all of the tasks to get done
- Sprint – short iteration, get current backlog items done in this time
- Scrum – short, daily stand-up meeting
- Planning session – start of each sprint, plan which backlog items will be done
- Heartbeat retrospective – end of sprint, reflect about the past sprint

SCRUM stuff

- Scrum Master - removes impediments to the ability of the team to deliver the sprint goal, not the team leader
- Self organizing teams – magically everybody gets organized
- Easily adapt to change – major benefit

Software Requirements

Requirements engineering

- The process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed.
- The requirements themselves are the descriptions of the system services and constraints that are generated during the requirements engineering process.
What is a requirement?

- It may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification.
- This is inevitable as requirements may serve a dual function
  - May be the basis for a bid for a contract - therefore must be open to interpretation;
  - May be the basis for the contract itself - therefore must be defined in detail;
- Both these statements may be called requirements.

Types of requirement

- User requirements
  - Statements in natural language plus diagrams of the services the system provides and its operational constraints. Written for customers.
- System requirements
  - A structured document setting out detailed descriptions of the system’s functions, services and operational constraints. Defines what should be implemented so may be part of a contract between client and contractor.

Functional and non-functional requirements

- Functional requirements
  - Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.
- Non-functional requirements
  - Constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.
- Domain requirements
  - Requirements that come from the application domain of the system and that reflect characteristics of that domain.

Functional requirements

- Describe functionality or system services.
- Depend on the type of software, expected users and the type of system where the software is used.
- Functional user requirements may be high-level statements of what the system should do but functional system requirements should describe the system services in detail.

Requirements imprecision

- Problems arise when requirements are not precisely stated.
- Ambiguous requirements may be interpreted in different ways by developers and users.
- Consider the term ‘appropriate viewers’
  - User intention - special purpose viewer for each different document type;
  - Developer interpretation - Provide a text viewer that shows the contents of the document.

Requirements completeness and consistency

- In principle, requirements should be both complete and consistent.
  - Complete
    - They should include descriptions of all facilities required.
  - Consistent
    - There should be no conflicts or contradictions in the descriptions of the system facilities.
  - In practice, it is impossible to produce a complete and consistent requirements document.
Non-functional requirements

- System properties
  - Reliability
  - Response time
  - Storage requirements
- Constraints
  - I/O device capability
  - System representations, etc.
- CASE system, programming language or development method
- Important – project could fail otherwise

Non-functional classifications

- Product requirements
  - Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.
- Organizational requirements
  - Requirements which are a consequence of organizational policies and procedures e.g. process standards used, implementation requirements, etc.
- External requirements
  - Requirements which arise from factors which are external to the system and its development process e.g. interoperability requirements, legislative requirements, etc.

Domain requirements

- Derived from the application domain and describe system characteristics and features that reflect the domain.
- Domain requirements can be new functional requirements, constraints on existing requirements or define specific computations.
- If domain requirements are not satisfied, the system may be unworkable.

Problems with natural language

- Lack of clarity
  - Precision is difficult without making the document difficult to read.
- Requirements confusion
  - Functional and non-functional requirements tend to be mixed-up.
- Requirements amalgamation
  - Several different requirements may be expressed together.

Guidelines for writing requirements

- Invent a standard format and use it for all requirements.
- Use language in a consistent way. Use shall for mandatory requirements, should for desirable requirements.
- Use text highlighting to identify key parts of the requirement.
- Avoid the use of computer jargon.

Verification and Validation
Verification vs validation

- Verification: "Are we building the product right".
- The software should conform to its specification.
- Validation: "Are we building the right product".
- The software should do what the user really requires.

The V & V process

- Is a whole life-cycle process - V & V must be applied at each stage in the software process.
- Has two principal objectives
  - The discovery of defects in a system;
  - The assessment of whether or not the system is useful and usable in an operational situation.

V & V goals

- Verification and validation should establish confidence that the software is fit for purpose.
- This does NOT mean completely free of defects.
- Rather, it must be good enough for its intended use and the type of use will determine the degree of confidence that is needed.

V & V confidence

- Depends on system's purpose, user expectations and marketing environment
  - Software function
    - The level of confidence depends on how critical the software is to an organization.
  - User expectations
    - Users may have low expectations of certain kinds of software.
  - Marketing environment
    - Getting a product to market early may be more important than finding defects in the program.

Static and dynamic verification

- Software inspections. Concerned with analysis of the static system representation to discover problems (static verification)
  - May be supplement by tool-based document and code analysis
- Software testing. Concerned with exercising and observing product behavior (dynamic verification)
  - The system is executed with test data and its operational behavior is observed

Program testing

- Can reveal the presence of errors NOT their absence.
- The only validation technique for non-functional requirements as the software has to be executed to see how it behaves.
- Should be used in conjunction with static verification to provide full V&V coverage.
Types of testing

- Defect testing
  - Tests designed to discover system defects.
  - A successful defect test is one which reveals the presence of defects in a system.
- Validation testing
  - Intended to show that the software meets its requirements.
  - A successful test is one that shows that a requirements has been properly implemented.

Software inspections

- These involve people examining the source representation with the aim of discovering anomalies and defects.
- Inspections not require execution of a system so may be used before implementation.
- They may be applied to any representation of the system (requirements, design, configuration data, test data, etc.).
- They have been shown to be an effective technique for discovering program errors.

Defect testing

- The goal of defect testing is to discover defects in programs
- A successful defect test is a test which causes a program to behave in an anomalous way
- Tests show the presence not the absence of defects

System testing

- Involves integrating components to create a system or sub-system.
- May involve testing an increment to be delivered to the customer.
- Two phases:
  - Integration testing - the test team have access to the system source code. The system is tested as components are integrated.
  - Release testing - the test team test the complete system to be delivered as a black-box.

Integration testing

- Involves building a system from its components and testing it for problems that arise from component interactions.
- Top-down integration
  - Develop the skeleton of the system and populate it with components.
- Bottom-up integration
  - Integrate infrastructure components then add functional components.
- To simplify error localization, systems should be incrementally integrated.

Black-box testing

- Involves examining the input-output behavior of a system.
- The system is treated as a black box, with no knowledge of its internal workings.
- Tests are designed to cover all possible input combinations and expected outputs.
- Used to verify that the system behaves as expected under various conditions.
- Useful for detecting functional failures and ensuring the system meets its specified requirements.
Testing guidelines

- Testing guidelines are hints for the testing team to help them choose tests that will reveal defects in the system.
  - Choose inputs that force the system to generate all error messages;
  - Design inputs that cause buffers to overflow;
  - Repeat the same input or input series several times;
  - Force invalid outputs to be generated;
  - Force computation results to be too large or too small.

Use cases

- Use cases can be a basis for deriving the tests for a system. They help identify operations to be tested and help design the required test cases.
  - From an associated sequence diagram, the inputs and outputs to be created for the tests can be identified.

Stress testing

- Exercises the system beyond its maximum design load. Stressing the system often causes defects to come to light.
- Stressing the system test failure behavior. Systems should not fail catastrophically. Stress testing checks for unacceptable loss of service or data.
- Stress testing is particularly relevant to distributed systems that can exhibit severe degradation as a network becomes overloaded.

Test automation

- Testing is an expensive process phase. Testing workbenches provide a range of tools to reduce the time required and total testing costs.
- Systems such as Junit support the automatic execution of tests.
- Most testing workbenches are open systems because testing needs are organization-specific.
- They are sometimes difficult to integrate with closed design and analysis workbenches.

Configuration management

- New versions of software systems are created as they change:
  - For different machines/OS;
  - Offering different functionality;
  - Tailored for particular user requirements.
- Configuration management is concerned with managing evolving software systems:
  - System change is a team activity;
  - CM aims to control the costs and effort involved in making changes to a system.
System families

- HP version
- Desktop version
- Windows XP version
- Server version
- Mac version

Frequent system building

- It is easier to find problems that stem from component interactions early in the process.
- This encourages thorough unit testing - developers are under pressure not to 'break the build'.
- A stringent change management process is required to keep track of problems that have been discovered and repaired.

Service-centric Software Engineering

Service-oriented architectures

- A means of developing distributed systems where the components are stand-alone services
- Services may execute on different computers from different service providers
- Standard protocols have been developed to support service communication and information exchange

Benefits of SOA

- Services can be provided locally or outsourced to external providers
- Services are language-independent
- Investment in legacy systems can be preserved
- Inter-organisational computing is facilitated through simplified information exchange
Dr. Tom Way
CSC 4181
Slide 67

Project Management

Dr. Tom Way
CSC 4181
Slide 68

- Concerned with activities involved in ensuring that software is delivered on time and on schedule and in accordance with the requirements of the organisations developing and procuring the software.
- Project management is needed because software development is always subject to budget and schedule constraints that are set by the organisation developing the software.

Dr. Tom Way
CSC 4181
Slide 69

Software project management

- The product is intangible.
- The product is uniquely flexible.
- Software engineering is not recognized as an engineering discipline with the same status as mechanical, electrical engineering, etc.
- The software development process is not standardised.
- Many software projects are 'one-off' projects.

Dr. Tom Way
CSC 4181
Slide 70

Software management distinctions

- Proposal writing.
- Project planning and scheduling.
- Project costing.
- Project monitoring and reviews.
- Personnel selection and evaluation.
- Report writing and presentations.

Dr. Tom Way
CSC 4181
Slide 71

Management activities

- These activities are not peculiar to software management.
- Many techniques of engineering project management are equally applicable to software project management.
- Technically complex engineering systems tend to suffer from the same problems as software systems.

Dr. Tom Way
CSC 4181
Slide 72

Management commonalities

- May not be possible to appoint the ideal people to work on a project
  - Project budget may not allow for the use of highly-paid staff;
  - Staff with the appropriate experience may not be available;
  - An organisation may wish to develop employee skills on a software project.
- Managers have to work within these constraints especially when there are shortages of trained staff.

Dr. Tom Way
CSC 4181
Slide 73

Project staffing
**Project planning**

- Probably the most time-consuming project management activity.
- Continuous activity from initial concept through to system delivery. Plans must be regularly revised as new information becomes available.
- Various different types of plan may be developed to support the main software project plan that is concerned with schedule and budget.

**Refactoring**

**Fowler’s definition**

“A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior”

(Martin Fowler, Refactoring, page 53)

**Composite Definition**

Changes made to the system that
- Do not change observable behavior (all the tests still pass)
- Remove duplication or needless complexity
- Enhance software quality
- Make the code simpler and easier to understand
- Make the code more flexible
- Make the code easier to change

**Why Refactor?**

- Prevent “design decay”
- Clean up messes in the code
- Simplify the code
- Increase readability and understandability
- Find bugs
- Reduce debugging time
- Build in learning we do about the application
- Redoing things is fundamental to every creative process

**Refactoring Flow**

1. Ensure all tests pass
2. Find code that smells
3. Determine simplification
4. Ensure all tests still pass
5. Make simplification
When to refactor

- "All the time"
- Rule of Three (2’s company, 3’s a crowd)
- When you add functionality
- When you learn something about the code
- When you fix a bug
- When the code smells

When not to refactor

- When the tests aren’t passing
- When you should just rewrite the code
- When you have impending deadlines

Problems with refactoring

- Taken too far, refactoring can lead to incessant tinkering with the code, trying to make it perfect
- Refactoring code when the tests don’t work or tests when the application doesn’t work leads to potentially dangerous situations
- Databases can be difficult to refactor
- Refactoring published interfaces can cause problems for the code that uses those interfaces