Table of Contents

COMP1531 Notes

Week 1 – Software Development

Software Eng Life-Cycle Development Phases

1. **Analysis and Specification**
   - Understand problem definition
   - Determine **functional** (in/output) and **non-functional** requirements (performance, security, quality, maintainability, extensibility)
   - Techniques : User stories/case modelling
2. **Design**
   - Designing a solution blueprint to implement customer requirements
   - Techniques : UML, CRC, ER Modelling, Conceptual Class Diagrams
3. **Implementation**
   - Encoding design to deliver a software product
4. **Testing**
   - Process of verification that system works and realises goals
   - Techniques : Unit Tests, User Acceptance Tests, Integration Tests

Methodologies

Iterative – every iteration encompasses all phases of software development

- Short iterations
- Heavy customer involvement
- Continuous feedback

Incremental – once a phase is finished, we do not go back

- Documentation heavy

Waterfall

Linear 'plan driven' development model with detailed planning. Suitable for simple, risk free projects with unchanging product statements or mission critical applications (e.g. NASA); heavy documentation.

- Linear sequential of all 4 dev phases

Model variation – quality gate system that ensures quality is maintained throughout the lifecycle.

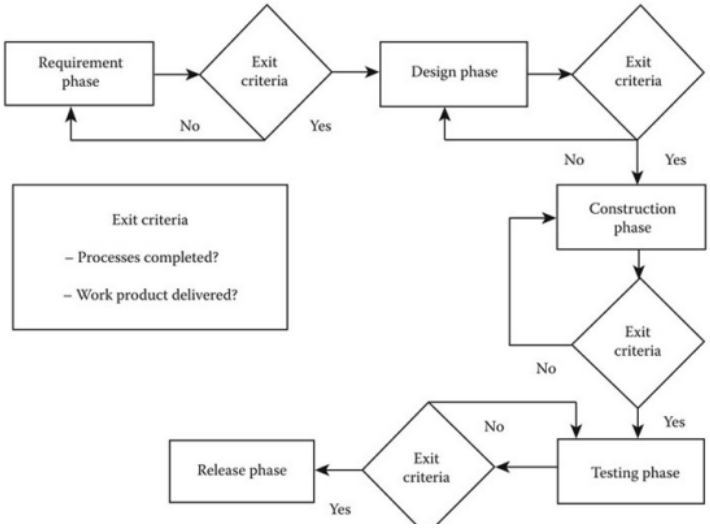

Figure 1 Waterfall model variation

Rational Unified Process (RUP)

Iterative software dev process that has a series of four phases that all possess their own software eng life-cycles:

1. Inception – scope the project and identify major players, resources, architecture and risks, and cost.
2. Elaboration – understanding problem domain and analysis.
3. Construction – design/build/test software
4. Transition – release into production

Agile

Software is built in iterations; all four dev phases are implemented in each iteration.

- Open to change at every stage of the process.
- Continuous customer involvement/feedback

Week 2 - XP, Scrum, Agile

Extreme Programming (XP)

'Design and code for today and not for tmr'

XP intends to improve software quality and responsiveness to changing customer requirements through **short dev cycles**; high focus on **adaptability** (changing requirements) rather than **predictability** (laying out all the requirements at the beginning).

Boehm's Cost Curve – as software dev proceeds through its lifecycle, the cost of making changes will grow; warning to be prepared for changes. Look out for:

- **Improper design** that increases complexity
- **Code duplication**
- Budgeting resources "cost of making vital changes exceeds resources"

Values of XP Programming

1. **Communication** – increased collab btw customers and developers; and among team members
2. **Simplest Solution** – focus on design and coding needs for today to avoid spending resources on uncertain future requirements
3. **Feedback** – from system (**periodic unit/integration** test), customer (**user acceptance** test), and team
4. **Courage** – be comfortable with refactoring code for better solutions and transparent team communication
5. **Respect** – never commit changes that fail tests; strive for quality

XP Principles

1. **Pair programming**
2. **Continuous Integration Checks**
3. **Open Workspace**
4. **Test-Driven Development** – periodic unit /user acceptance testing to ensure low coupling and high cohesion
5. **Refactoring** – reconstructing the code without changing external behaviour to improve non-functional attributes
6. **Sustainable Pace** – program for today not for tomorrow

Extras:

- Simple design – focus on the stories of the current iteration to keep design simple and expressive
- Constant feedback – from team and customers
- System Metaphor – figure of speech that summarises the system into layman's terms for everyone to understand

XP Planning Game

Initial Exploration

Conversation between customer and devs to identify all significant features (more will be discovered later).

- Break down epic stories into user stories
- Estimate velocity via approx number of stories doable in each iteration
- Cost of each user story is extrapolated from project velocity

Release Plan

Plan a release date (6/12/24 months in the future) with customer and which stories are critical for the planned date. Can be planned by:

- **Time –** how many user stories can be implemented before given date
- **Scope** – how long a set of stories will take to finish

Negotiate with all plan devs/customer/ and manager to not underestimate scope of work.

Iteration Planning

Create iteration plans (typically 1~ 2 weeks).

- Customer to prioritise user stories; must fit current velocity
- Stories cannot change once iteration begins
- Iteration ends even if not all stories are done
- **Story is not done until all acceptance tests pass**

Task Planning

User stories are broken down into tasks with estimated time for completion and implementation order is determined. Time is summed up and if it is too much, customer must choose stories to cull.

When should it be used?

- When problem domains are uncertain and open to change
- Customer do not have firm idea of product
- Ideal project group sizes of 2 – 12

Agile

Requirements engineering is formulating a well defined problem to solve. Traditional phases of requirements engineering:

- Requirements gathering – get understanding of problem through customers and stakeholders
- Requirements analysis – defining the problem through gathered info, negotiate with customer to determine priorities, ensure devs and customer problem matches.
- Requirements specification – documentation to ensure clarity (e.g. UML to model use cases)

Agile Requirements Engineering

1. **Visioning** – identify:

- Theme or epic stories of the project.
- Target users
- Key selling points of the product (3-5 objectives)

2. **Brainstorm** the features of the epics
3. **Breakdown** into smaller user stories
4. **Detail** user stories to yield iteration deliverables
5. Finalise **product backlog** (list of US that needs to be done)

User Stories

Key to user stories is to focus on **who**, **what**, and **why**:

'As a <type of user>, I want <some goal> so that <some reason>'

Capturing the Components of a US

Card – a post-it-note as a physical token giving tangible form of what would otherwise be an abstraction

Conversation – Taking place at different time and places during the project with various people that is largely verbal but is supplemented by documentation

Confirmation – devs getting confirmation regarding acceptance criteria from product owner to get a clear understand of how the feature will work under different situations

I.N.V.E.S.T – US Quality Control

Independent – US should be developed independently and delivered separately

Negotiable – US should be discussable further

Valuable – product owner should be clear on 'why'

Estimable – should be understandable enough that it can be broken up into tasks to be estimated and delivered