10 Software Life Cycle Models

Yan Shi

SE 273 Lecture Notes

Part of this note is adopted from Sommerville’s textbook slides
Software Life Cycle

- A **structured set of activities** required to develop a software system
- AKA **software process**
  - **Specification** - what the system should do and its development constraints
  - **Development** - production of the software system
  - **Validation** - checking that the software is what the customer wants
  - **Evolution** - changing the software in response to changing demands.
A software life cycle model is an abstract representation of a life cycle. It presents a description of a process from some particular perspective.

Generic software life cycle model examples:
- Waterfall model
- Evolutionary development
- Component-based software engineering
The Waterfall
The Waterfall Model

1. Requirements definition
2. System and software design
3. Implementation and unit testing
4. Integration and system testing
5. Operation and maintenance
Waterfall Model Features

- **In principle:**
  - The result of each phase is document(s) that are approved ("signed off").
  - The following phase will not start until the previous phase is finished.
  - Freeze output document(s) after one phase is finished.
  - No going back.

- **In reality:**
  - Phases may overlap.
  - May rewind back to the previous phase:
    - design identifies requirement problems
    - coding reveals design issues
    - testing detects all kinds of failures
Waterfall Model Advantages

- Simple and easy to follow.
- Documentations are produced at each phase.
- Similar to other engineering process models.
  - Often used when developing software which is part of a larger system engineering project.
Waterfall Model Problems

- The main drawback of the waterfall model is the difficulty of accommodating change after the process is underway.

**Why?**
- In the waterfall model, one phase has to be complete before moving onto the next phase.
- Inflexible partitioning of the project into distinct stages makes it difficult to respond to changing customer requirements.

- Therefore, this model is only appropriate when the requirements are well-understood and changes will be fairly limited during the design process.
Evolutionary Development

Software development life cycles. (a) Traditional waterfall model. (b) Evolutionary (EVO) development model.
Evolutionary Development Features

- Develop the system in many small but complete iterations.
- Each iteration results in a working deliverable.
- Present each deliverable to the customer for feedback.
- Allow intermediate corrections of requirements, specifications, and plans to make sure that the project is targeting the right direction.

- Two examples:
  - Exploratory development
  - Throwaway prototyping
Exploratory Development

- Objective:
  - to work with customers
  - to evolve a final system from an initial outline specification.

- Start with well-understood part of the requirements
- Add new features as proposed by the customer.
Often, it is necessary to experiment with a system to determine what exactly is needed.

Prototypes are quick (we ignore FURPS) solutions that are intended to be thrown away when done.

It is important not to let a prototype be released into production.

- They were not designed or implemented with the same care that we would exercise for production code.
When to use prototyping?

- Prototyping is useful when:
  - Customers don’t know exactly what they want
  - We want to analyze the usability of an HCI
  - We are not sure if a particular approach to solving a problem will work
  - We have multiple approaches to solving a problem and we want to experiment to see which is better.
Evolutionary Development Problems

- **Lack of process visibility:**
  - It is not cost effective to produce documentations for each version because the system is developed too quickly.
  - Hard for traditional project management.

- **Systems are often poorly structured:**
  - Continual change tends to corrupt the structure.
  - Incorporating changes will become more and more difficult and costly.

- **Special skills (e.g. in languages for rapid prototyping; special process activities) may be required.**
Component-Based Development

http://www.oig.co.uk/development.jsp
Component-Based Development

- Based on **systematic reuse** where systems are integrated from existing components or COTS (Commercial-off-the-shelf) systems.

- **Process stages**
  - Requirement analysis ➔
  - Component assessment and specification ➔
  - Requirements modification ➔
  - System design with reuse ➔
  - Development, integration and testing.

- This approach is becoming increasingly used as component standards have emerged.
Component-Based Development

- **Benefits:**
  - Reduce the amount of software to be developed
  - Reduce costs and risks
  - Usually faster delivery of the software

- **Problems:**
  - Requirement compromises are inevitable.
  - Developed system may not meet the real need of the customer.
Other Life Cycle Models

- **Incremental Delivery**
  - Specification, design and implementation are broken into a series of increments.

- **Spiral Model**
  - Spirals outwards from an initial outline to the final system.
  - Focus on risk assessment and reduction

- **Rational Unified Process**
  - A modern model derived from UML
  - Separation of phases and workflows
Costs of software engineering

- Roughly 60% of costs are development costs, 40% are testing costs.
- For custom software, evolution costs often exceed development costs.
- Costs vary depending on the type of system being developed and the requirements of system attributes such as performance and system reliability.
- Distribution of costs depends on the process model that is used.
Activity cost distribution

Waterfall model

<table>
<thead>
<tr>
<th>Phase</th>
<th>Specification</th>
<th>Design</th>
<th>Development</th>
<th>Integration and testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>25</td>
<td>50</td>
<td>75</td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Iterative development

<table>
<thead>
<tr>
<th>Phase</th>
<th>Specification</th>
<th>Iterative development</th>
<th>System testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Component-based software engineering

<table>
<thead>
<tr>
<th>Phase</th>
<th>Specification</th>
<th>Development</th>
<th>Integration and testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>25</td>
<td>50</td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Development and evolution costs for long-lifetime systems

<table>
<thead>
<tr>
<th>Phase</th>
<th>System development</th>
<th>System evolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>200</td>
</tr>
</tbody>
</table>
Summary

- Software life cycle activities
  - Specification
  - Development
  - Validation
  - Evolution
- Waterfall model
- Evolutionary development
- Component-based software engineering
- Other life cycle models
Constraints for software process

A process must satisfy many constraints:
  - The software product type
  - Size of the software
  - Resources available
  - Expected lifespan of the software
  - Economics
Characteristics of a good process

- **Easy to understand** by those needing to perform the process
- Recorded in a **formal** manner
- **Continually evolving**, as we find things that don’t work we remove these practices and replace them with something that we hope is better.
- **Tailored to fit the project needs.**
- **Necessary and useful** in producing and maintaining a software product.