Software Life-Cycle Models

Xiaojun Qi

Life-Cycle Model

- It specifies the various phases/workflows of the software process, such as the requirements, analysis (specification), design, implementation, and postdelivery maintenance, and the order in which they are to be carried out.

Software Development in Theory

- Ideally, software is developed as described in Chapter 1
  - Linear
  - Starting from scratch

Software Development in Practice

- In the real world, software development is totally different and is more chaotic
  - Software professionals make mistakes
  - The client’s requirements change while the software product is being developed
  - A software product is a model of the real world, and the real world is continually changing.

1. Code-and-Fix Life-Cycle Model

- No design
- No specifications

The easiest way to develop software
The most expensive way for maintenance (i.e., maintenance nightmare)

Code-and-Fix Life-Cycle Model (Cont.)

- The product is implemented without requirements or specifications, or any attempt at design.

- The developers simply throw code together and rework it as many times as necessary to satisfy the client.

- It is used in small project and is totally unsatisfactory for products of any reasonable size.
2. Waterfall Life-Cycle Model

- The linear life cycle model with feedback loops
  - The waterfall model cannot show the order of events

Waterfall Life-Cycle Model (Cont.)

- No phase is complete until the documentation for that phase has been completed and the products of that phase have been approved by the software quality assurance (SQA) group.
- If the products of an earlier phase have to be changed as a consequence of following a feedback loop, that earlier phase is deemed to be complete only when the documentation for the phase has been modified and the modifications have been checked by the SQA group.

Waterfall Life-Cycle Model

- Advantages:
  - Documentation is provided at each phase
  - All the products of each phase (including the documentation) are meticulously checked by SQA. ➔ Maintenance is easier

- Disadvantages:
  - Specification documents are long, detailed, and boring to read.

3. Rapid-Prototyping Life-Cycle Model

- Linear model
- “Rapid”

Rapid-Prototyping Life-Cycle Model (Cont.)

- A rapid prototype is a working model that is functionally equivalent to a subset of the product.
- The first step is to build a rapid prototype and let the client and future users interact and experiment with the rapid prototype.
- Strength:
  - The development of the product is essentially linear, proceeding from the rapid prototype to the delivered product.
  - The feedback loops of the waterfall model are less likely to be needed in the rapid prototyping model.
  - It is built rapidly and modified rapidly to reflect the client’s needs. ➔ Speed is of the essence.

Rapid-Prototyping Life-Cycle Model (Cont.)

- Weakness:
  - Once the client’s real needs have been determined, the rapid prototype implementation is discarded.
- The lessons learned from the rapid prototype implementation are retained and used in subsequent development phases.
4. Open-Source Life-Cycle Model

• The first informal phase
  – One individual builds an initial version and makes it available via the Internet (e.g., SourceForge.net)
  – If there is sufficient interest in the project, the initial version is widely downloaded; users become co-developers; and the product is extended

• Key point: Individuals generally work voluntarily on an open-source project in their spare time

Open-Source Life-Cycle Model (Cont.)

• The second informal phase
  – Reporting and correcting defects
    • Corrective maintenance
  – Adding additional functionality
    • Perfective maintenance
  – Porting the program to a new environment
    • Adaptive maintenance

• The second informal phase consists solely of postdelivery maintenance
  – The word “co-developers” on the previous slide should rather be “co-maintainers”

Open-Source Life-Cycle Model (Cont.)

• A point will be reached when the open-source product is no longer maintainable
  • The open-source life-cycle model is restricted in its applicability
    – It can be extremely successful for infrastructure projects, such as: Operating systems (Linux, OpenBSD, Mach, Darwin), Web browsers (Firefox, Netscape), Compilers (gcc), Web servers (Apache), and Database management systems (MySQL)
    – There cannot be open-source development of a software product to be used in just one commercial organization
    – The open-source life-cycle model is inapplicable unless the target product is viewed by a wide range of users as useful

Open-Source vs. Closed-Source

• Closed-source software is maintained and tested by employees
  – Users can submit failure reports but never fault reports

• Open-source software is generally maintained by unpaid volunteers
  – Users are strongly encouraged to submit defect reports, both failure reports and fault reports
    • Core group: Small number of dedicated maintainers with the inclination, the time, and the necessary skills to submit fault reports (“fixes”); They take responsibility for managing the project; They have the authority to install fixes
    • Peripheral group: Users who choose to submit defect reports from time to time
Open-Source vs. Closed-Source (Cont.)

- New versions of closed-source software are typically released roughly once a year
  - After careful testing by the SQA group

- The core group releases a new version of an open-source product as soon as it is ready
  - Perhaps a month or even a day after the previous version was released
  - The core group performs minimal testing
  - Extensive testing is performed by the members of the peripheral group in the course of utilizing the software
  - “Release early and often”

Winburg Mini Case Study

- **Episode 1**: The first version is implemented

- **Episode 2**: A fault is found
  - The product is too slow because of an implementation fault
  - Changes to the implementation are begun

- **Episode 3**: The requirements change
  - A faster algorithm is used

- **Episode 4**: A new design is adopted
  - Development is complete

- **Epilogue**: A few years later, these problems recur

Teal Tractors Mini Case Study

- While the Teal Tractors software product is being constructed, the requirements change

- The company is expanding into Canada

- Changes needed include:
  - Additional sales regions must be added
  - The product must be able to handle Canadian taxes and other Canadian business aspects
  - The product must be extended to handle two different currencies, USD and CAD

- These changes may be
  - Great for the company; but
  - Disastrous for the software product

Evolution-Tree Life-Cycle Model

- The explicit order of events is shown

- At the end of each episode
  - We have a baseline, a complete set of artifacts (a constitute component of a software product)

- Example:
  - Baseline at the end of Episode 4:
    - Requirements₄, Analysis₄, Design₄, Implementation₄

Moving Target Problem

- **Moving Target Problem**: The requirements change while the software product is being developed. This change is inevitable
  - Growing companies are always going to change
  - If the individual calling for changes has sufficient clout, nothing can be done to prevent the changes being implemented.

- The software product can be adversely impacted
  - Numerous changes can induce dependencies within the code.

- Any change made to a software product can potentially cause a regression fault. That is, a change to one part of the software induces a fault in an apparently unrelated part of the software

- If there are too many changes, the entire product may have to be redesigned and reimplemented

- **There is no solution to the moving target problem!!**
6. Iterative and Incremental Life-Cycle Model: Iteration

• In the real life, we cannot speak about “the analysis phase”
  – Instead, the operations of the analysis phase are spread out over the life cycle as a consequence of both the moving target problem and the need to correct the inevitable mistakes

• The basic software development process is iterative
  – Each successive version is intended to be closer to its target than its predecessor

Iterative and Incremental Life-Cycle Model: Incrementation

• Miller’s Law: At any one time, we can concentrate on only approximately seven chunks (units of information)

• To handle larger amounts of information, use stepwise refinement
  – Concentrate on the aspects that are currently the most important
  – Postpone aspects that are currently less critical
  – Every aspect is eventually handled, but in the order of current importance

• This is an incremental process

Iterative-and-Incremental Model

Iteration and incrementation are used in conjunction with one another

1) There is no single “requirements phase/workflow” or “design phase/workflow”
2) Instead, there are multiple instances of each phase/workflow. However, only one phase/workflow predominates at most times.

Workflows

• All five core workflows are performed over the entire life cycle

• However, at most times one workflow predominates

• Examples:
  – At the beginning of the life cycle
    • The requirements workflow predominates
  – At the end of the life cycle
    • The implementation and test workflows predominate

• Planning and documentation activities are performed throughout the life cycle

Combine the Evolution-Tree and the Iterative-and-Incremental Models
Combine the Evolution-Tree and the Iterative-and-Incremental Models (Cont.)

- Each episode corresponds to an increment
- Not every increment includes every workflow
- Increment B was not completed
- Dashed lines denote maintenance
  - Episodes 2, 3: Corrective maintenance
  - Episode 4: Perfective maintenance

Risks and Other Aspects of Iteration and Incrementation

- We can consider the project as a whole as a set of mini projects (increments)
- Each mini project extends the
  - Requirements artifacts, Analysis artifacts, Design artifacts, Implementation artifacts, Testing artifacts
- During each mini project, we
  - Extend the artifacts (incrementation);
  - Check the artifacts (test workflow); and
  - If necessary, change the relevant artifacts (iteration)
- The final set of artifacts is the complete product

Risks and Other Aspects of Iteration and Incrementation (Cont.)

- Each iteration can be viewed as a small but complete waterfall life-cycle model
- During each iteration we select a portion of the software product
- On that portion we perform the
  - Classical requirements phase
  - Classical analysis phase
  - Classical design phase
  - Classical implementation phase

Strength of the Iterative-and-Incremental Model

- There are multiple opportunities for checking that the software product is correct
  - Every iteration incorporates the test workflow
  - Faults can be detected and corrected early
- The robustness of the architecture can be determined early in the life cycle
  - Architecture — the various component modules and how they fit together
  - Robustness — the property of being able to handle extensions and changes without falling apart

Strength of the Iterative-and-Incremental Model (Cont.)

- We can mitigate (resolve) risks early
  - Risks are invariably involved in software development and maintenance
- We always have a working version of the software product
  - Variation: Deliver partial versions to smooth the introduction of the new product in the client organization
- There is empirical evidence that the life-cycle model works

Managing Iteration and Incrementation

- The iterative-and-incremental life-cycle model is as regimented as the waterfall model since developing a software product using the iterative-and-incremental model is equivalent to developing a series of smaller software products, all using the waterfall model.
  - The project as a whole is broken up into a series of waterfall mini projects.
  - During each mini-project, iteration is performed as needed.
  - For each increment, the requirements, analysis, design, and implementation phases are repeatedly performed until it is clear that no further iteration is needed. That is: each increment is a waterfall mini project.
- The iterative-and-incremental life-cycle model is the waterfall model, applied successively
7. Agile Processes: Extreme Programming

- Extreme programming is a somewhat controversial new approach to software development based on the iterative-and-incremental model.
- The software development team determines the various features (stories) the client would like the product to support.

Agile Processes: Extreme Programming (Cont.)

- Based on the features (stories) the client wants:
  - The development team estimates the duration and cost of each feature
  - The client selects the features for next build using cost-benefit analysis
  - The proposed build is broken down into smaller pieces termed tasks
  - The programmer draws up test cases for a task → Test-driven development (TDD)

Unusual Features of Extreme Programming (XP)

- Pair Programming: The programmer works together with a partner on one screen to implement the task and ensure that all the test cases work correctly.
- Continuous integration of tasks since a number of pairs implement tasks in parallel
- The TDD test cases used for the task are retained and utilized in all further integration testing.

Agile Processes

- Extreme programming is one of a number of new paradigms that are collectively referred to as agile processes.
- Agile processes are characterized by
  - Less emphasis on analysis and design
  - Earlier implementation (working software is considered more important than detailed documentation)
  - Responsiveness to change in requirements
  - Close collaboration with the client

Agile Processes (Cont.)

- A principle in the Manifesto is
  - Deliver working software frequently
    - Ideally every 2 or 3 weeks
- One way of achieving this is to use timeboxing
  - Used for many years as a time-management technique
- A specific amount of time is set aside for a task
  - Typically 3 weeks for each iteration
  - The team members then do the best job they can during that time
- Agile processes demand fixed time, not fixed features
Agile Processes (Cont.)

- Another common feature of agile processes is **stand-up meetings**
  - Short meetings held at a regular time each day
  - Attendance is required
- Participants stand in a circle
  - They do not sit around a table
  - To ensure the meeting lasts no more than 15 minutes
- The aim of a stand-up meeting is
  - To raise problems, not solve them
- Solutions are found at follow-up meetings, preferably held directly after the stand-up meeting

Evaluating Agile Processes

- Agile processes have had some successes with small-scale software development
  - However, medium- and large-scale software development is very different
- The key decider: **The impact of agile processes on postdelivery maintenance**
  - Refactoring is an essential component of agile processes
  - Refactoring continues during maintenance
  - Will refactoring increase the cost of postdelivery maintenance?

8. Synchronize-and-Stabilize Life-Cycle Model

- Microsoft’s life cycle model
- Requirements: Interview numerous potential clients for the package and extract a list of features of highest priority to the clients.
- Draw up specifications
- Divide the work into three or four builds.
  - The 1st build: Most critical features.
  - The 2nd build: The next most critical features.
  - Each build is carried out by a number of small teams working in parallel.

Synchronize-and-Stabilize Life-Cycle Model (Cont.)

- **Synchronize at the end of each day:** Put the partially completed components together and test and debug the resulting product.
- **Stabilize at the end of each build:** Fix remaining faults and no further changes will be made to the specifications
Synchronize-and-Stabilize Life-Cycle Model (Cont.)

- Advantages:
  - The repeated synchronization step ensures that the various components always work together.
  - The regular execution of the partially constructed product makes the developers gain early insight into the operation of the product and modify the requirements if necessary during the course of a build.

9. Spiral Life-Cycle Model

- Simplified form
  - Rapid prototyping model plus risk analysis preceding each phase

- If all risks cannot be mitigated, the project is immediately terminated

Full Spiral Life-Cycle Model

- Precede each phase by
  - Alternatives
  - Risk analysis

- Follow each phase by
  - Evaluation
  - Planning of the next phase

Radial dimension: cumulative costs to date
Angular dimension: progress through the spiral.

Spiral Life-Cycle Model (Cont.)

- Minimize risk via the use of prototypes and other means.

- Two types of risk:
  - Analyzable Risk: Time and cost
  - Un-analyzable Risk:
    - Personnel turnover
    - Difference between small-scale and large-scale software
    - Evaluate the delivery promises of a hardware supplier

Spiral Life-Cycle Model (Cont.)

- Precede each phase by determining:
  - Objective of the phase;
  - Alternatives for achieving the objectives;
  - Constraints imposed on those alternatives.

- Strategy is analyzed from the viewpoint of risk. Attempts are made to resolve every potential risk.

- Follow each phase by:
  - The results of the phase are evaluated.
  - The next phase is planned.

Spiral Life-Cycle Model (Cont.)

- Strengths
  - The emphasis on alternatives and constraints supports the reuse of existing software and the incorporation of software quality as a specific objective.
  - There is essentially no distinction between development and maintenance since maintenance is another cycle of the spiral.

- Weaknesses
  - For large-scale software only
  - For internal (in-house) software only
Comparison of Life-Cycle Models

• Different life-cycle models have been presented
  – Each with its own strengths and weaknesses

• Criteria for deciding on a model include:
  – The organization
  – Its management
  – The skills of the employees
  – The nature of the product

• Best suggestion
  – “Mix-and-match” life-cycle model

<table>
<thead>
<tr>
<th>Life-Cycle Model</th>
<th>Strengths</th>
<th>Weaknesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evolutionary model (Section 2.2)</td>
<td>Clearly models real-world software production Equivalent to the waterfall and incremental model</td>
<td>Clumsy models real-world software production Underestimates the development process</td>
</tr>
<tr>
<td>Spiral life-cycle model (Section 2.3)</td>
<td>Fully defined models, thorough process planning</td>
<td>Not yet proven beyond all doubt</td>
</tr>
<tr>
<td>Rapid prototyping (Section 2.4)</td>
<td>Ensures that the delivered product meets the client’s needs</td>
<td>Difficult product may not meet client’s needs</td>
</tr>
<tr>
<td>Agile processes (Section 2.5)</td>
<td>Works well when the client’s requirements are vague</td>
<td>Limited applicability Usually does not work</td>
</tr>
<tr>
<td>Synthesis and stability life-cycle model (Section 2.6)</td>
<td>Ensures that components can be successfully integrated</td>
<td>Can be used for only large-scale, in-house projects</td>
</tr>
</tbody>
</table>

Developers have to be competent in risk analysis and risk resolution.