CHAPTER 02
SOFTWARE LIFE-CYCLE MODELS

Software Development in Theory

Reference

In an ideal world, a software product is developed as described theoretically:
- Developed from scratch
- First the client’s requirements are determined
- Then the analysis is performed
- When the analysis artifacts are complete, the design is produced
- This is followed by the implementation of the complete product
- It is then installed on the client’s computer

Figure 2.1
Software Development in Theory

- Software development is considerably different in practice
  - Software professionals are human and therefore make mistakes
  - The client's requirements can change while the software is being developed
- Life-cycle model — Series of steps to be performed while the software product is developed and maintained

Software Development in Theory

- Evolution-tree life-cycle model
  - Different episodes are depicted
  - At the end of each episode, there is a baseline (a complete set of artifacts)
- Figure 2.2

![Evolution-tree life-cycle model diagram]

Software Development in Theory

- Waterfall life-cycle model
  - A linear model with feedback loops
- Figure 2.3

![Waterfall life-cycle model diagram]

Lessons of Case Studies

- Why are so many changes to a software product needed
  - Software professionals make mistakes
  - A software product is a model of the real world, and the real world is continually changing
- Moving-target problem — While software is being developed, the requirements change
  - A manager's changing mind regarding the functionality
  - Feature creep — Succession of small additions to requirements
- Frequent changes harmful to the health of software product
- Changes can induce regression faults
  - A change inducing other faults
Iteration and Incrementation

- Life cycle of actual software products resemble the evolution tree model or the waterfall model
  - Rather than the idealized chain
  - Consequence of both the moving-target problem and the need to correct the inevitable mistakes
- The basic process for an artifact is iterative
  - First version produced
  - Revised to produce second
  - And so on
- Iteration is an intrinsic aspect of software engineering
- Iterative life-cycle models used for over 30 years
- Another aspect of developing real-world software is the restriction imposed on us by Miller's Law
  - At any one time, humans are capable of concentrating on only approximately seven chunks (units of information)

Iteration and Incrementation

- A typical software artifact has far more than seven chunks
  - Code artifacts with more than seven variables
  - Requirements documents with many more than seven requirements
- One way humans handle this restriction is to use stepwise refinement
  - Concentrate on aspects currently most important
  - Postpone those aspects that are currently less critical
- E.g., first considering the seven most important requirements, and then the next few
- This is an incremental process
- Incrementation is also an intrinsic aspect of software engineering
  - Incremental software development is over 45 years old

Iteration and Incrementation

- In practice, iteration and incrementation are used in conjunction with one another
  - An artifact is constructed piece by piece (incrementation)
  - And each increment goes through multiple versions (iterations)
- Iterative-and-incremental life-cycle model
  - Based on the iteration and incrementation concepts
  - Different products will have different increments
- Shows how the emphasis changes from iteration to iteration

Iteration and Incrementation

- Figure 2.4
Iteration and Incrementation

- Five core workflows (activities) are performed over the entire life cycle
  - Requirements, analysis, design, implementation, and testing
- All performed over the life cycle of a software product
- There are times when one workflow predominates over the other four
- Planning and documentation activities are performed throughout the iterative-and-incremental life cycle
- Testing is a major activity during each iteration
- Total effort
  - 1/5 devoted to requirements and analysis workflows
  - 1/5 devoted to design workflow
  - 3/5 devoted to implementation workflow

Iteration and Incrementation

- There is iteration during each increment
  - Number of iterations varies from increment to increment
- Figure 2.5

Risks and Other Aspects of Iteration and Incrementation

- The iterative and incremental model has many strengths
  1. Multiple opportunities for checking that the software product is correct
     - Every iteration incorporates the test workflow
  2. The robustness of the underlying architecture can be determined relatively early in the life cycle
     - Architecture of a software product consists of various component artifacts and how they fit together
     - Extended continually to incorporate the next increment
     - Robustness — Being able to handle such extensions and changes without falling apart
  3. Enables the early mitigation of risks
     - Risks are invariably involved in software development and maintenance
     - Risks mitigated early during incremental development
  4. There is always a working version of the software
     - At the end of each iteration, there is a working version of part of the overall software product
     - Client and intended users can experiment with that version
     - Benefits of introducing a software product gradually: Understandable fear of being replaced by a computer is diminished, and it is easier to learn the functionality of a complex product if it is introduced stepwise
  5. Empirical evidence that it works
     - Standish Group report on projects
     - Results for 1994 through 2006
     - Iterative process a factor for successful projects
Risks and Other Aspects of Iteration and Incrementation

- 2002 to 2004 decrease in successful projects
  - More large projects, use of waterfall model, lack of user involvement, and lack of support from senior executives
- 2006 increase in successful projects
  - Better project management, emerging Web infrastructure, and iterative development

Figure 2.7

Managing Iteration and Incrementation

- The iterative-and-incremental model is as regimented as the waterfall model
  - Similar to developing a series of smaller software products, all using the waterfall model
  - The waterfall model applied successively

Other Life-Cycle Models

Code-and-Fix Life-Cycle Model

- The product is implemented without requirements or specifications nor any attempt at design
- Developers throw code together and rework it as many times as necessary
- Unsatisfactory for products of any reasonable size
  - May work on 100 or 200 LOC
- Regrettably, all too many projects use this model

Other Life-Cycle Models

Figure 2.8
Other Life-Cycle Models

Waterfall Life-Cycle Model

- No phase is complete until
  - Documentation for that phase has been completed and
  - Products of that phase have been approved by the software quality assurance (SQA) group
- Inherent in every phase is testing
- The strength is enforced disciplined approach
  - Documentation to be provided at each phase
  - All products of each phase to be checked by SQA
- The weakness is being documentation driven
  - Specification documents can be long, detailed, and boring for the client to read
  - Written in a style with which the client is unfamiliar
  - Could be written in a formal specification language

Other Life-Cycle Models

Rapid Prototyping Life-Cycle Model

- Rapid prototype — A working model that is functionally equivalent to a subset of the product
- First step: Build a rapid prototype and let the client and future users interact and experiment
- Once the client is satisfied that the rapid prototype does most of what is required: Developers can draw up the specification document
- Major strength
  - Development of the product is essentially linear
  - Feedback loops of the waterfall model less likely to be needed
- Essential aspect
  - Rapid prototype is constructed as rapidly as possible
  - Internal structure of the rapid prototype is not relevant
Open-Source Life-Cycle Model

- Two informal phases of almost all successful open-source projects
  1. Single individual has an idea for a program and an initial version is built and made available for distribution free of charge
     - Operating system (Linux), net browser (Firefox), Web server (Apache)
     - Sites such as SourceForge.net and FreshMeat.net
  2. If sufficient interest, users become co-developers
     - Reporting defects and fixing defects (corrective maintenance)
     - Ideas for extending functionality and implementing the ideas (perfective maintenance)
     - Porting the program to new environments (adaptive maintenance)

Key aspect: Individuals work in their spare time on a voluntary basis and not paid to participate

Differences between closed-source and open-source life-cycle models

- Closed-source: Users submit failure reports (observed incorrect behavior), not fault reports (where the source code is incorrect)
- Open-source: Users encouraged to submit defect reports, and have access to code
  - Core group — Dedicated maintainers managing the project
  - Peripheral group — Users who sometimes submit reports

When a fault report is submitted
- Core group member checks that the fix indeed solves the problem

When a failure report is submitted
- Core group member determines the fix or assigns the task
- Installs the fix

Closed-source: New versions typically released roughly once a year
- Carefully checked by SQA

Open-source: "Release early. Release often"
- Minimal testing
- More extensive testing by peripheral group

Open-source model has features in common with code-and-fix and rapid-prototyping models
- An initial working version is produced
- Reworked until it becomes the target product
- No specification or design

How have some open-source projects been so successful, bearing in mind great importance of specification and design
- Attracted finest software experts
- Model restricted in its applicability
- Extremely successful for certain infrastructure projects
  - Operating system: Linux, OpenBSD, Mach, Darwin
  - Web browsers: Firefox, Netscape and Web servers: Apache
  - Compilers: gcc
  - Database management systems: MySQL
Other Life-Cycle Models

**Open-Source Life-Cycle Model**
- More than 350,000 open-source projects
  - SourceForge.net and FreshMeat.net
  - About half never attracted a team to work on them
  - Many of those started unlikely to progress
  - When it has worked, incredibly successful
- Figure 2.11

**Agile Processes**
- Extreme programming (XP) is a somewhat controversial new approach to software development based on the iterative-and-incremental model
- Steps of extreme programming
  - Development team determines the features (stories) the client would like the product to support
  - For each feature, the team informs the client of the time for implementation and the cost
  - Previous steps correspond to requirements and analysis workflow of the iterative-and-incremental model
  - Client selects features to be included in each successive build using cost-benefit analysis (team estimates and potential benefits)
  - The proposed build broken down into smaller pieces termed tasks

---

**Other Life-Cycle Models**

**Agile Processes**
- A programmer first draws up test cases for a task
  - This is termed task-driven development (TDD)
  - Two programmers work together on one computer (pair programming) implementing the task
  - Ensuring that all test cases work correctly
  - Task is then integrated into the current version of the product
- Ideally, implementing and integrating a task should take no more than a few hours
  - Two programmers alternate typing every 15 to 20 minutes
    - One not typing carefully checks the code being entered
  - In general, a number of pairs will implement tasks in parallel
    - Integration can take place essentially continually
  - If possible, team members change coding partners daily
  - Learning from other team member

---

**Other Life-Cycle Models**

**Agile Processes**
- Some drawbacks have been observed in practice
  - Requires large blocks of uninterrupted time
  - Difficulty in finding 3 to 4 hour blocks of time
  - Does not work well with shy or overbearing individuals or two inexperienced programmers
- Features of extreme programming
  - Computers of the team set up in the center of a large room lined with small cubicles
  - A client representative works with the team at all times
  - No individual can work overtime for two successive weeks
  - No specialization: All members work on analysis, design, code, and testing
  - No overall design step before the various builds are constructed: Design is modified while the product is being built (refactoring)
Other Life-Cycle Models

Agile Processes

- Whenever a test case does not run, code is reorganized until team is satisfied that design is simple and runs all test cases
- Two acronyms associated with extreme programming
  - YAGNI — You aren't gonna need it
  - DTSTTCPW — Do the simplest thing that could possibly work
    Minimize the number of features
- Extreme programming is one of a number of new paradigms that are collectively referred to as agile processes
- Agile Alliance
  - 17 software developers (Agile Alliance)
  - Authors of extreme programming, Crystal, Scrum
  - Meeting in 2001
  - Produced Manifesto for Agile Software Development

- Do not prescribe a specific software model
- Common underlying principles
- Agile processes
  - Considerably less emphasis on analysis and design than in almost all other life-cycle models
  - Implementation starting much earlier in the life cycle
  - Major goals: Responsive to change and collaborating with client
- One of the Manifesto principles: Deliver working software frequently, ideally every two or three weeks
- Two basic principles based on successful management techniques
  - Timeboxing
  - Stand-up meeting

- Five questions
  1. What have I done since yesterday's meeting?
  2. What am I working on today?
  3. What problems are preventing me from achieving this?
  4. What have we forgotten?
  5. What did I learn that I would like to share with the team?
- Results of agile processing
  - Has been successfully used on a number of small-scale projects
  - Has not been used widely enough to determine whether it will fulfill its early promise
- Many software professionals have expressed doubts about agile processes within the context of medium-scale and specially large-scale software products
Other Life-Cycle Models

Agile Processes

- The analogy of successfully hammering together a few planks to build a doghouse, versus building a three-bedroom home without detailed plans
- Skills in plumbing, wiring, and roofing are needed
- A skyscraper that is the height of 1000 doghouses cannot be built by piling 1000 doghouses on top of one another
- A key determinant in deciding whether agile processes are a major breakthrough in software engineering will be the cost of future postdelivery maintenance
  - If agile processes result in a reduction in the cost of postdelivery maintenance, they will be widely adopted
  - If design of a product is open-ended and flexible: Perfective maintenance should be easy and low cost

Other Life-Cycle Models

Synchronize-and-Stabilize Life-Cycle Model

- The model is a version of the iterative-and-incremental model
- Used by Microsoft, Inc. (world's largest manufacturer of COTS software) to build the majority of its packages
- Steps of synchronize-and-stabilize
  - The requirements analysis phase is conducted by interviewing numerous potential customers
  - Extracting a list of features of highest priority to the customers
  - A specification document is drawn up
  - Work divided into three or four builds: first build (most critical features), second build (next most critical features), so on
  - Each build carried out by a small number of teams working in parallel
Other Life-Cycle Models

**Synchronize-and-Stabilize Life-Cycle Model**
- At the end of each day all teams synchronize — Put the partially completed components together and test and debug the resulting product
- Stabilization is performed at the end of each of the builds
- Any remaining fault that have been detected so far are fixed
- Freeze the build (no further changes made to the specifications)
- **Features**
  - Repeated synchronization step: Various components always work together
  - Regular execution of the partially constructed product: Developers obtain an early insight into the product’s operation and they can modify the requirements if necessary during the course of a build
  - Model can even be used if the initial specification is incomplete

Other Life-Cycle Models

**Spiral Life-Cycle Model**
- An element of risk almost always is involved in development of software
  - Key personnel can resign before the product has been adequately documented
  - Manufacturer of hardware on which the product is critically dependent can go bankrupt
  - Too much or too little invested in testing and quality assurance
  - Technological breakthroughs can render the entire product worthless
  - Lower-priced, functionally equivalent package by a competitor
  - Components may not fit together during integration
- Software developers try to minimize such risks whenever possible

Other Life-Cycle Models

**Spiral Life-Cycle Model**
- Constructing a prototype
  - A proof-of-concept prototype — An engineering prototype, a scale model constructed to test the feasibility of construction
  - Not a rapid prototype
  - E.g., a prototype to test the timing of certain computations
- The model is based on the idea of minimizing risk via the use of prototypes and other means
- Figure 2.12

Other Life-Cycle Models

**Spiral Life-Cycle Model**
- Model is waterfall model with each phase preceded by risk analysis
- Before each phase, an attempt is made to mitigate (control) the risks
- If impossible to mitigate all significant risks: Project is terminated
Other Life-Cycle Models

Spiral Life-Cycle Model
- The full spiral model
  - Radial dimension: Cumulative cost to date
  - Angular dimension: Progress through the spiral
- Each cycle corresponds to a phase, beginning in the top left quadrant
- Figure 2.13

- Has been used successfully to develop a variety of products
  - In cases, the productivity of projects increased by 50% to 100%

- Strengths of the spiral model
  - Supports the reuse of existing software
  - Incorporation of software quality as a specific objective
  - Risk analysis to determine amount of testing
  - Too much testing: Waste of money
  - Too little testing: Residual faults
  - Postdelivery maintenance is another cycle of the spiral
  - No distinction between postdelivery maintenance and development

- Restrictions on the applicability of the model
  - Intended exclusively for internal development of large-scale software
    - Risk analysis can lead to the termination of project
    - In-house personnel can be reassigned
    - With contract software, termination can lead to a breach-of-contract
    - Contract software requires risk analysis before start of contract by both client and developers
  - Applicable to only large-scale software
    - Cost of performing risk analysis should be justified
    - Being risk driven can also be a weakness
    - Software developers have to be skilled at pinpointing the possible risks and analyzing the risks accurately
Other Life-Cycle Models

Object-Oriented Life-Cycle Model
- In object-oriented paradigm, the need for iteration between phases is more common than the structured paradigm
- Life-cycle models explicitly reflect the need for iteration
- Fountain model
  - Overlapping circles represent overlap
  - Arrows within a phase represent iteration
  - Smaller maintenance circle represents reduced effort
  - Not commonly used
- Figure

Comparison of Life-Cycle Models

- Different life-cycle models examined
  - Special attention to strengths and weaknesses
- Each software development organization should decide on a life-cycle model
  - Appropriate for that organization, its management, its employees, and its software process
  - Can vary the model depending on the features of the specific product currently under development
  - Incorporating appropriate aspects of the various life-cycle models: Utilizing their strengths and minimizing their weaknesses

<table>
<thead>
<tr>
<th>Life-Cycle Model</th>
<th>Strengths</th>
<th>Weaknesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evolution-tree model (Section 2.2)</td>
<td>Closely models real-world software production</td>
<td>Totally unsatisfactory for minitual programs</td>
</tr>
<tr>
<td>Iterative-and-incremental life-cycle model (Section 2.5)</td>
<td>Equivocally to the Iterative-and-incremental model</td>
<td>Delivered product may not meet client’s needs</td>
</tr>
<tr>
<td>Code-and-fix life-cycle model (Section 2.9.1)</td>
<td>Fine for short programs that require no maintenance</td>
<td>Not yet proven beyond all doubt</td>
</tr>
<tr>
<td>Waterfall life-cycle model (Section 2.9.2)</td>
<td>Disciplined approach</td>
<td>Limited applicability. Usually does not work</td>
</tr>
<tr>
<td>Rapid-prototyping life-cycle model (Section 2.9.4)</td>
<td>Ensures that the delivered product meets the client’s needs</td>
<td>Appears to work on only small-scale projects</td>
</tr>
<tr>
<td>Open-source life-cycle model (Section 2.9.5)</td>
<td>Has worked extremely well in a small number of instances</td>
<td>Has not been widely used other than at Microsoft</td>
</tr>
<tr>
<td>Agile process (Section 2.9.3)</td>
<td>Work well when the client’s requirements are vague</td>
<td>Can be used for only large-scale, in-house products</td>
</tr>
<tr>
<td>Synchronize-and-stabilize life-cycle model (Section 2.9.6)</td>
<td>Future users’ needs are met</td>
<td>Developers have to be competent in risk analysis</td>
</tr>
<tr>
<td>Spiral life-cycle model (Section 2.9.7)</td>
<td>Ensures that components can be successfully integrated</td>
<td>and risk resolution</td>
</tr>
</tbody>
</table>