Other Software Life Cycle Models

- Code-and-fix life-cycle model
- Waterfall life-cycle model
- Rapid prototyping life-cycle model
- Open-source life-cycle model
- Agile processes
- Synchronize-and-stabilize life-cycle model
- Spiral life-cycle model

Code-and-Fix Model (Figure 2.8)

- No design
- No specifications
- Maintenance nightmare
- The easiest way to develop software
- The most expensive way

Full Waterfall Model (Figure 2.9)

- Characterized by feedback loops
- Documentation-driven
- Advantages: documentation maintenance is easier
- Disadvantages: specification document

Rapid Prototyping Model (Figure 2.10)

- Linear model
- Analysis after prototype
- "Rapid"

Open-Source Life-Cycle Model

- Two informal phases. First, one individual builds an initial version
- Made available via the Internet (e.g., SourceForge.net)
- Then, if there is sufficient interest in the project
  - The initial version is widely downloaded
  - Users become co-developers
  - The product is extended
- Key point: Individuals generally work voluntarily on an open-source project mostly in their spare time

Open-Source Life-Cycle Model (2)

- In the second informal phase, report and correct defects
- Corrective maintenance
- Add additional functionality
- Perfective maintenance
- Port the program to a new environment
- Adaptive maintenance
- This phase consists solely of postdelivery maintenance
- The word "co-developers" on the previous slide really should be "co-maintainers"
Open-Source Life-Cycle Model (3)

- Could be called the postdelivery maintenance life-cycle model (Figure 2.11)

Compare Open-Source Life-Cycle with Closed-Source Software

- Closed-source software is maintained and tested by employees
  - Users can submit failure reports but never fault reports (the source code is not available)
  - Open-source software is generally maintained by unpaid volunteers
  - Users are strongly encouraged to submit defect reports, both failure reports and fault reports

Compare Open-Source Life-Cycle with Closed-Source Software (2)

- Two types of maintainers. 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

Compare Open-Source Life-Cycle with Closed-Source Software (3)

- 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"

Open-Source Life-Cycle Model (4)

- An initial working version is produced when using
  - The rapid-prototyping model;
  - The code-and-fix model; and
  - The open-source life-cycle model
- Then:
  - Rapid-prototyping model
  - The initial version is discarded
  - Code-and-fix model and open-source life-cycle model
  - The initial version becomes the target product

Open-Source Life-Cycle Model (5)

- Consequently, in an open-source project, there are generally no specifications and no design
- How have some open-source projects been so successful without specifications or designs?
Open-Source Life-Cycle Model (6)
- Open-source software production has attracted some of the world’s finest software experts
  - They can function effectively without specifications or designs
  - However, eventually a point will be reached when the open-source product is no longer maintainable

Open-Source Life-Cycle Model (7)
- 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)
    - Database management systems (MySQL)

Open-Source Life-Cycle Model (8)
- There cannot be open-source development of a software product to be used in just one commercial organization
  - Members of both the core group and the periphery are invariably users of the software being developed
  - The open-source life-cycle model is inapplicable unless the target product is viewed by a wide range of users as useful to them

Open-Source Life-Cycle Model (9)
- About half of the open-source projects on the Web have not attracted a team to work on the project
  - Even where work has started, the overwhelming preponderance will never be completed
  - But when the open-source model has worked, it has sometimes been incredibly successful
    - The open-source products previously listed have been utilized on a regular basis by millions of users

Extreme Programming (XP)
- Somewhat controversial new approach
- Stories (features client wants)
  - Estimate duration and cost of each story
  - Client selects stories for next build
  - Each build is divided into tasks
  - Test cases for a task are drawn up first
- Pair programming
- Continuous integration of tasks

Unusual Features of XP
- The computers are put in the center of a large room lined with cubicles
- A client representative is always present
- Software professionals cannot work overtime for 2 successive weeks
- No specialization
- Refactoring (design modification)
Agile Processes

- XP is one of a number of new paradigms collectively referred to as agile processes
- Seventeen software developers (later dubbed the "Agile Alliance") met at a Utah ski resort for two days in February 2001 and produced the Manifesto for Agile Software Development
- The Agile Alliance did not prescribe a specific life-cycle model
  - Instead, they laid out a group of underlying principles

Agile Processes (2)

- Agile processes are a collection of new paradigms characterized by
  - Less emphasis on analysis and design
  - Earlier implementation (working software is considered more important than documentation)
  - Responsiveness to change
  - Close collaboration with the client

Agile Processes (3)

- 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 (4)

- It gives the client confidence to know that a new version with additional functionality will arrive every 3 weeks
- The developers know that they will have 3 weeks (but no more) to deliver a new iteration
  - Without client interference of any kind
  - If it is impossible to complete the entire task in the timebox, the work may be reduced ("descoped")
  - Agile processes demand fixed time, not fixed features

Agile Processes (5)

- 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

Agile Processes (6)

- At a stand-up meeting, each team member in turn answers five questions:
  - What have I done since yesterday's meeting?
  - What am I working on today?
  - What problems are preventing me from achieving this?
  - What have we forgotten?
  - What did I learn that I would like to share with the team?
Agile Processes (7)
- 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

Agile Processes (8)
- Stand-up meetings and timeboxing are both
  - Successful management techniques
  - Now utilized within the context of agile processes
- Both techniques are instances of two basic principles that underlie all agile methods:
  - Communication; and
  - Satisfying the client's needs as quickly as possible

Evaluating Agile Processes
- Agile processes have had some successes with small-scale software development
  - However, medium- and large-scale software development could be very different
- The key decider: the impact of agile processes on post-delivery maintenance
  - Refactoring is an essential component of agile processes
  - Refactoring continues during maintenance
  - Will refactoring increase the cost of post-delivery maintenance, as indicated by preliminary research?

Evaluating Agile Processes (2)
- Agile processes are good when requirements are vague or changing
- It is too soon to evaluate agile processes
  - There are not enough data yet
  - Even if agile processes prove to be disappointing
    - Some features (such as pair programming) may be adopted as mainstream software engineering practices

Synchronize-and-Stabilize Model
- Microsoft’s life-cycle model
- Requirements analysis — interview potential customers
- Draw up specifications
- Divide project into 3 or 4 builds
- Each build is carried out by small teams working in parallel

Synchronize-and-Stabilize Model (2)
- At the end of the day — synchronize (test and debug)
- At the end of the build — stabilize (freeze the build)
- Components always work together
  - Get early insights into the operation of the product
- However, read WSJ article for Friday’s class (link on course webpage)
Spiral Model
- Simplified form (Figure 2.12)
- Rapid prototyping model plus risk analysis preceding each phase
- Key point: If all risks cannot be mitigated, the project is immediately terminated

Full Spiral Model
- Precede each phase by
  - Alternatives
  - Risk analysis
- Follow each phase by
  - Evaluation
  - Planning of the next phase
- Radial dimension: cumulative cost to date
- Angular dimension: progress through the spiral

Full Spiral Model (Figure 2.13)

Analysis of the Spiral Model
- Strengths
  - It is easy to judge how much to test
  - No distinction is made between development and maintenance
- Weaknesses
  - For large-scale software only
  - Risk analysis can be expensive
  - For internal (in-house) software only
  - Cannot arbitrarily decide to terminate project

Comparison of Life-Cycle Models
- Each model has 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
CS 390 – Lecture 4
Other Software Life Cycle Models

- Code-and-fix life-cycle model
- Waterfall life-cycle model
- Rapid prototyping life-cycle model
- Open-source life-cycle model
- Agile processes
- Synchronize-and-stabilize life-cycle model
- Spiral life-cycle model
Code-and-Fix Model (Figure 2.8)

- No design
- No specifications
  - Maintenance nightmare
- The easiest way to develop software
- The most expensive way

Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.
Full Waterfall Model (Figure 2.9)

- Characterized by
  - Feedback loops
  - Documentation-driven

- Advantages
  - Documentation
  - Maintenance is easier

- Disadvantages
  - Specification document
Rapid Prototyping Model (Figure 2.10)

- **Linear model**
  - Analysis after prototype
- "Rapid"
Open-Source Life-Cycle Model

- Two informal phases. First, one individual builds an initial version
  - Made available via the Internet (e.g., SourceForge.net)
- Then, if there is sufficient interest in the project
  - The initial version is widely downloaded
  - Users become co-developers
  - The product is extended
- Key point: Individuals generally work voluntarily on an open-source project mostly in their spare time
Open-Source Life-Cycle Model (2)

- In the second informal phase, report and correct defects
  - Corrective maintenance
- Add additional functionality
  - Perfective maintenance
- Port the program to a new environment
  - Adaptive maintenance
- This phase consists *solely* of postdelivery maintenance
  - The word “co-developers” on the previous slide really should be “co-maintainers”
Open-Source Life-Cycle Model (3)

- Could be called the postdelivery maintenance life-cycle model (Figure 2.11)
Compare Open-Source Life-Cycle with Closed-Source Software

- Closed-source software is maintained and tested by employees
  - Users can submit failure reports but never fault reports (the source code is not available)
- Open-source software is generally maintained by unpaid volunteers
  - Users are strongly encouraged to submit defect reports, both failure reports and fault reports
Compare Open-Source Life-Cycle with Closed-Source Software (2)

- Two types of maintainers. 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
Compare Open-Source Life-Cycle with Closed-Source Software (3)

- 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”
Open-Source Life-Cycle Model (4)

- An initial working version is produced when using
  - The rapid-prototyping model;
  - The code-and-fix model; and
  - The open-source life-cycle model

- Then:
  - Rapid-prototyping model
    - The initial version is discarded
  - Code-and-fix model and open-source life-cycle model
    - The initial version becomes the target product
Open-Source Life-Cycle Model (5)

- Consequently, in an open-source project, there are generally no specifications and no design.
- How have some open-source projects been so successful without specifications or designs?
Open-source software production has attracted some of the world’s finest software experts

- They can function effectively without specifications or designs

However, eventually a point will be reached when the open-source product is no longer maintainable
Open-Source Life-Cycle Model (7)

- 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)
  - Database management systems (MySQL)
Open-Source Life-Cycle Model (8)

- There cannot be open-source development of a software product to be used in just one commercial organization
  - Members of both the core group and the periphery are invariably users of the software being developed
- The open-source life-cycle model is inapplicable unless the target product is viewed by a wide range of users as useful to them
Open-Source Life-Cycle Model (9)

- About half of the open-source projects on the Web have not attracted a team to work on the project.
- Even where work has started, the overwhelming preponderance will never be completed.
- But when the open-source model has worked, it has sometimes been incredibly successful:
  - The open-source products previously listed have been utilized on a regular basis by millions of users.
Extreme Programming (XP)

- Somewhat controversial new approach
- *Stories* (features client wants)
  - Estimate duration and cost of each story
  - Client selects stories for next build
  - Each build is divided into tasks
  - Test cases for a task are drawn up first
- Pair programming
- Continuous integration of tasks
Unusual Features of XP

○ The computers are put in the center of a large room lined with cubicles
○ A client representative is always present
○ Software professionals cannot work overtime for 2 successive weeks
○ No specialization
○ *Refactoring* (design modification)
Agile Processes

- XP is one of a number of new paradigms collectively referred to as *agile processes*
- Seventeen software developers (later dubbed the “Agile Alliance”) met at a Utah ski resort for two days in February 2001 and produced the *Manifesto for Agile Software Development*
- The Agile Alliance did not prescribe a specific life-cycle model
  - Instead, they laid out a group of underlying principles
Agile Processes (2)

- Agile processes are a collection of new paradigms characterized by
  - Less emphasis on analysis and design
  - Earlier implementation (working software is considered more important than documentation)
  - Responsiveness to change
  - Close collaboration with the client
Agile Processes (3)

- 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 (4)

- It gives the client confidence to know that a new version with additional functionality will arrive every 3 weeks.
- The developers know that they will have 3 weeks (but no more) to deliver a new iteration
  - Without client interference of any kind.
- If it is impossible to complete the entire task in the timebox, the work may be reduced (“descoped”)
  - Agile processes demand fixed time, not fixed features.
Agile Processes (5)

○ 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
Agile Processes (6)

- At a stand-up meeting, each team member in turn answers five questions:
  - What have I done since yesterday’s meeting?
  - What am I working on today?
  - What problems are preventing me from achieving this?
  - What have we forgotten?
  - What did I learn that I would like to share with the team?
Agile Processes (7)

- 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
Agile Processes (8)

- Stand-up meetings and timeboxing are both
  - Successful management techniques
  - Now utilized within the context of agile processes
- Both techniques are instances of two basic principles that underlie all agile methods:
  - Communication; and
  - Satisfying the client’s needs as quickly as possible
Evaluating Agile Processes

- Agile processes have had some successes with small-scale software development
  - However, medium- and large-scale software development could be 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 post-delivery maintenance, as indicated by preliminary research?
Evaluating Agile Processes (2)

- Agile processes are good when requirements are vague or changing
- It is too soon to evaluate agile processes
  - There are not enough data yet
- Even if agile processes prove to be disappointing
  - Some features (such as pair programming) may be adopted as mainstream software engineering practices
Synchronize-and-Stabilize Model

- Microsoft’s life-cycle model
- Requirements analysis — interview potential customers
- Draw up specifications
- Divide project into 3 or 4 builds
- Each build is carried out by small teams working in parallel
Synchronize-and-Stabilize Model (2)

- At the end of the day — *synchronize* (test and debug)
- At the end of the build — *stabilize* (freeze the build)
- Components always work together
  - Get early insights into the operation of the product
- However, read WSJ article for Friday’s class (link on course webpage)
Spiral Model

- **Simplified form (Figure 2.12)**
  - Rapid prototyping model plus risk analysis preceding each phase
- **Key point:** If all risks cannot be mitigated, the project is immediately terminated
Full Spiral Model

- Precede each phase by
  - Alternatives
  - Risk analysis
- Follow each phase by
  - Evaluation
  - Planning of the next phase
- Radial dimension: cumulative cost to date
- Angular dimension: progress through the spiral
Full Spiral Model (Figure 2.13)
Analysis of the Spiral Model

- **Strengths**
  - It is easy to judge how much to test
  - No distinction is made between development and maintenance

- **Weaknesses**
  - For large-scale software only
    - Risk analysis can be expensive
  - For internal (in-house) software only
    - Cannot arbitrarily decide to terminate project
Comparison of Life-Cycle Models

- Each model has 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