Chapter 3
Agile Software Development
Topics covered

- Rapid software development
- Agile methods
- Plan-driven vs. agile development
- Extreme programming (XP)
- Agile project management
- Scaling agile methods
Rapid software development
Rapid software development

- **Rapid development and delivery** is often the most critical requirement for software systems.
  
  • Businesses must be responsive to new opportunities and competition.
  
  • Software has to evolve quickly to facilitate this responsiveness.
  
  • Evolving quickly requires being able to cope with unstable requirements.
General characteristics of a rapid development processes

- Processes of **specification, design and implementation are interleaved** (= incremental development).
- System is developed in a series of versions with stakeholders involved in version evaluation and feedback.
- User interfaces are often developed using an interactive development system that supports quick GUI development.
Agile methods
Agile methods background and aim

- Dissatisfaction with the overhead of heavyweight, plan-driven development approaches of the 1980s and 1990s led to the creation of agile methods.
  - Careful project planning, formalized QA, CASE-supported analysis and design, and a controlled, rigorous process are appropriate for large, long-lifetime systems developed by large, geographically dispersed teams over many years, but...
  - When applied to small and medium-sized business systems, the overhead dominates and interferes with rapid development and delivery.

(cont’d)
Agile methods background and aim

The newly proposed *agile methods* allowed:

- A focus on the *software* itself rather than its *design and documentation*.
- A reliance on *iterative development*.
- An ability to *deliver working software quickly* which can *evolve quickly* to meet rapidly changing requirements.
- A *reduction in bureaucracy* by avoiding unnecessary work and overhead.

See: [www.agilealliance.org](http://www.agilealliance.org)
“Manifesto for Agile Software Developers”

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

That is, while there is value in the items on the right, we value the items on the left more.
# Principles of agile methods

<table>
<thead>
<tr>
<th>Principle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer involvement</td>
<td>Customers <strong>should be closely involved</strong> throughout the development process. Their role is <strong>provide and prioritize new system requirements</strong> and to <strong>evaluate the iterations of the system</strong>.</td>
</tr>
<tr>
<td>incremental development (not “I. Delivery”)</td>
<td>The software is <strong>developed</strong> in increments with the customer specifying the requirements to be included in each increment.</td>
</tr>
<tr>
<td>People not process</td>
<td>The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive processes.</td>
</tr>
<tr>
<td>Embrace change</td>
<td>Expect the system <strong>requirements to change</strong> and so design the system to accommodate these changes.</td>
</tr>
<tr>
<td>Maintain simplicity</td>
<td>Focus on simplicity in both the <strong>software being developed</strong> and in the development <strong>process</strong>. Wherever possible, <strong>actively work to eliminate complexity from the system</strong>.</td>
</tr>
</tbody>
</table>
Agile methods applicability

- Software companies developing small or medium-sized generic products for sale.
- Small or medium-sized custom system development where there is clear commitment from the customer to be involved in the process and where there are few external rules and regulations that affect the software.

(Because of their focus on small, tightly-integrated teams, there are problems in scaling agile methods to large systems.)
Problems with agile methods

- Can be **difficult to keep the interest of customers** who are involved in the process.
- **Team members may be unsuited to the intense involvement** that characterizes agile methods.
- **Prioritizing changes can be difficult** where there are multiple stakeholders.
- **Maintaining (code) simplicity requires extra work.**
- **Contracts may be a problem** as with other iterative development approaches.
Agile methods and software maintenance

- Since most organizations spend more on maintaining existing software than they do on new software development, *if agile methods are to be successful, they have to support maintenance as well as original development.*

- Two key issues:
  - The emphasis on *minimizing formal documentation*...
  - Evolving a system in response to customer change requests *after delivery*...

- In particular, problems may arise if the original development team cannot be retained.
Plan-driven vs. agile development
Stereotypical attributes

- **Plan-driven development**: based around separate development stages with the outputs produced at each stage (including formal documents) used to **plan the following activity**.
  - Iteration only occurs *within* activities.

- **Agile development**: design and implementation are the central activities; they *incorporate* other activities such as requirements elicitation and testing.
  - Iteration occurs both *within* and *across* activities.

(cont’d)
Stereotypical attributes (cont’d)

- However, *most* software projects include practices from **both** plan-driven AND agile approaches.
- For example, a plan-driven process CAN support *incremental development and delivery*; likewise an agile process is not inevitably code-focused and MAY produce some *design documentation*. 
Finding the right balance between plan-driven and agile development

- **Is it important to have a detailed specification and design before moving to implementation?** If so, you probably need to use a (more) plan-driven approach.

- **Is an incremental development strategy, whereby you deliver the software to customers and get rapid feedback from them, realistic?** If so, consider using a (more) agile methods-based approach.

- **How large is the system that is being developed?** Agile methods are most effective when the system can be developed with a small co-located team whose members can communicate informally. This may not be possible for large systems so a (more) plan-driven approach may have to be used.

(cont’d)
Finding the right balance between plan-driven and agile development (cont’d)

- **What type of system is being developed?** Plan-driven approaches may be required for systems that require *a lot of analysis before implementation* (e.g., real-time systems with complex timing requirements).

- **What is the expected system lifetime?** Long-lifetime systems may require *more design documentation* to communicate the original intentions of the system developers to the support team.

- **What technologies are available to support system development?** Agile methods rely on *good tools* to keep track of an evolving design.

(cont’d)
Finding the right balance between plan-driven and agile development (cont’d)

- **How is the development team organized?** If the development team is distributed or if part of the development is being outsourced, then you may need to develop design documents to communicate across the development teams.

- **Are there cultural or organizational issues that may affect system development?** Traditional engineering organizations have a culture of plan-based development, as this is the norm in engineering.
Finding the right balance between plan-driven and agile development (cont’d)

- **How skilled are the designers and programmers in the development team?** It is sometimes argued that agile methods require *higher skill levels* than plan-based approaches in which programmers simply translate a detailed design into code.

- **Is the system subject to *external regulation***? If a system has to be *approved by an external regulator* (e.g., the FAA approves software that is critical to the operation of an aircraft) then you will probably be required to produce *detailed documentation* as part of the system safety case.
XP
Extreme programming (XP)

- Perhaps the **best-known and most widely used agile method.**
- Takes an “extreme” approach to iterative development:
  - New versions may be built **several times per day.**
  - Increments are delivered to customers **every 2 weeks.**
  - **All tests** must run successfully for every build.

“XP = iterative development on steroids”
XP and agile principles

- Incremental development is supported through small, frequent system releases.
- Customer involvement means full-time customer engagement with the team.
- Focus is on people – not process – through pair programming, collective ownership, and a process that avoids long working hours.

(cont’d)
XP and agile principles (cont'd)

- Responsiveness to change is supported through regular system releases.
- Simplicity (and therefore maintainability) is preserved through constant refactoring of code.
The XP release cycle

1. Select user stories for this release
2. Break down stories to tasks
3. Plan release
4. Develop/integrate/test software
5. Release software
6. Evaluate system

Develop/integrate/test software

Plan release

Break down stories to tasks

Select user stories for this release

Evaluate system
## Extreme programming practices

<table>
<thead>
<tr>
<th>Principle or practice</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incremental planning</td>
<td>Requirements are recorded on story cards and the stories to be included in a release are determined by the time available and their relative priority. The developers break these stories into development ‘Tasks’. See Figures 3.5 and 3.6.</td>
</tr>
<tr>
<td>Small releases</td>
<td>The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.</td>
</tr>
<tr>
<td>Simple design</td>
<td>Enough design is carried out to meet the current requirements and no more.</td>
</tr>
<tr>
<td>Test-First Development</td>
<td>An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.</td>
</tr>
<tr>
<td>Refactoring</td>
<td>All developers are expected to refactor the code continuously as soon as possible code improvements are found. This keeps the code simple and maintainable.</td>
</tr>
</tbody>
</table>
## Extreme programming practices 2

<table>
<thead>
<tr>
<th>Practice</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pair Programming</td>
<td>Developers work in pairs, checking each other’s work and providing the support to always do a good job.</td>
</tr>
<tr>
<td>Collective ownership “egoless programming”</td>
<td>The pairs of developers work on all areas of the system, so that no “islands of expertise” develop and all the developers take responsibility for all of the code. Anyone can change anything.</td>
</tr>
<tr>
<td>Continuous integration</td>
<td>As soon as the work on a task is complete, it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.</td>
</tr>
<tr>
<td>Sustainable pace</td>
<td>Large amounts of overtime are not considered acceptable as the net effect is often to reduce code quality and medium term productivity.</td>
</tr>
<tr>
<td>On-site customer</td>
<td>A representative of the end-user of the system (the customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.</td>
</tr>
</tbody>
</table>
XP Requirements scenarios

- Requirements are expressed as *scenarios* or *user stories* written on index cards.
- **Customer** chooses stories for inclusion in the next release based on priorities and schedule estimates.
- Development team breaks them down into implementation *tasks*.
- Tasks are the *basis of schedule and cost estimates*. 
**Story card for document downloading**

<table>
<thead>
<tr>
<th><strong>Downloading and printing an article</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>First, you select the article that you want from a displayed list. You then have to tell the system how you will pay for it - this can either be through a subscription, through a company account or by credit card.</td>
</tr>
<tr>
<td>After this, you get a copyright form from the system to fill in and, when you have submitted this, the article you want is downloaded onto your computer.</td>
</tr>
<tr>
<td>You then choose a printer and a copy of the article is printed. You tell the system if printing has been successful.</td>
</tr>
<tr>
<td>If the article is a print-only article you can’t keep the PDF version so it is automatically deleted from your computer.</td>
</tr>
</tbody>
</table>
Task cards for document downloading

Task 1: Implement principal workflow

Task 2: Implement article catalog and selection

Task 3: Implement payment collection

Payment may be made in 3 different ways. The user selects which way they wish to pay. If the user has a library subscription, then they can input the subscriber key which should be checked by the system. Alternatively, they can input an organizational account number. If this is valid, a debit of the cost of the article is posted to this account. Finally, they may input a 16 digit credit card number and expiry date. This should be checked for validity and, if valid, a debit is posted to that credit card account.
XP and change

- Conventional SE wisdom is *design for change* (e.g., via “information hiding”) to reduce maintenance costs.

- XP maintains that this is *not worthwhile* since changes cannot be reliably anticipated.

- (Instead,) XP proposes *constant code improvement* ("refactoring") to make changes easier when they have to be implemented.
Refactoring

- Team looks for possible software improvements and makes them even if there is no immediate need for this.
- Improves understandability of software and so reduces the need for documentation.
- Changes are easier to make because the code is well-structured and clear.
- However, some changes requires *architecture refactoring* which is much more expensive. (An obvious *disadvantage* of: “Simple design – enough design is carried out to meet the current requirements and no more.”)
Examples of refactoring

- Reorganizing class hierarchies to remove duplicate code.
- Tidying-up and renaming attributes and methods to make them easier to understand.
- Replacing in-line code with calls to methods that have been included in a program library. (= information hiding!)
Testing in XP

- Testing is *central* to XP: the program is tested after every change has been made.
  - **Test-First Development:** tests are written from scenarios *before coding* to clarify requirements.
  - **Users are involved** in test development and validation.
  - **An automated testing framework** (e.g., JUnit) is used to run all component tests before each new release.

* **Test-Driven Development** is a generalization of this idea that will be considered later.
Test-First Development

- Tests are written as **programs** rather than data so they can be executed automatically. (Each test includes a check that it has executed correctly.)
- All previous and new tests are run automatically when new functionality is added, thus ensuring that the new functionality has not introduced errors.

“automated regression testing”
Customer involvement in testing

- The role of the customer is to help develop **acceptance tests** for the stories that are to be implemented in the next release.

- In theory, then, **all new code is validated** to ensure that it does what the customer requires.

- However, customers may be reluctant to accept this role – either because of limited time availability or because they feel that providing the requirements was a sufficient contribution to the process.
Pair Programming in XP

- Programmers **work in pairs**, sitting together at the same workstation to develop software.
- Pairs are created dynamically so that all team members work with each other during the development process.
- Supports idea of **collective ownership and responsibility for the system** ("egoless programming").
- Resultant sharing of knowledge is very important as it reduces the overall risks to a project when team members leave.

(cont’d)
Pair Programming in XP (cont'd)

- Serves as an informal, continuous review process, as each line of code is looked at by at least two people.
- This, in turn, encourages refactoring, from which the whole team benefits.
- Some (but not all) evidence suggests that a pair working together may be more productive than two programmers working separately.
Agile project management
Plan-driven and agile project management

- The principal responsibility of software project management is to ensure (quality) software is delivered on time and within the planned budget.

- The standard approach is plan-driven. Managers determine what should be delivered, when it should be delivered, and who will work on the deliverables.

- Agile project management requires a different approach – one that is adapted to incremental development and the particular strengths of agile methods.
The Scrum* management approach

- **Scrum** is a general agile method but its focus is on managing iterative development rather than specific agile practices.

- There are three phases in Scrum...
  - **Planning phase**: establish general objectives for the project and design the software architecture.
  - **Sprint or development phase**: a series of “sprint cycles,” where each cycle yields an increment of the system.
  - **Closure phase**: wraps-up the project, completes required documentation such as help frames and user manuals, and assesses the lessons learned from the project.

*Scrum* is a rugby term for the close-knit, shoulder-to-shoulder formation a rugby team forms to jointly move the ball forward.
The Sprint cycle

- **Sprints** are fixed length (normally 2-4 weeks) and correspond to the development of a new system release in XP.

- The starting point for each sprint is the *product backlog*, which is the list of work to be done on the project.

- The project team works with the customer to select the *(stable)* features and functionality to be developed during the sprint.

(cont’d)
The Sprint cycle (cont’d)

- Once these are agreed upon, the team is isolated from the customer, with all communications channelled through the so-called “Scrum master.”

- The role of the Scrum master is to protect the development team from external distractions.

- At the end of the sprint, the development work is reviewed and presented to stakeholders. The next sprint cycle then begins.
Teamwork in Scrum

- The **Scrum master** is a facilitator who arranges daily meetings, tracks the backlog of work to be done, records decisions, measures progress, and communicates with customers and management.

- The whole team attends **short daily meetings** to share information, describe their progress and problems, and plan their work for the following day.

- Thus, everyone on the team knows what is going on and, if problems arise, can re-plan their short-term work to cope with them.

*See, for example, “The Daily Scrum Meeting.”*
Scrum benefits

- The product is broken down into a set of manageable and understandable pieces.
- Unstable requirements do not, in theory, hold-up progress.
- The whole team has visibility of everything, so team communication is improved.
- Customers see regular, on-time delivery of increments and learn how the product works.
- Trust between customers and developers is established and a positive culture is created in which everyone expects the project to succeed.
Scaling agile methods
Small and medium systems development

- Agile methods work well for small and medium sized projects that can be developed by a small, co-located team.
- Some argue this is because of the improved communications that naturally take place when small groups work together in one location.
- To scale effectively, agile methods must be modified to cope with larger, longer projects utilizing multiple development teams working in different locations.
Large systems development

- Large systems are often comprised of communicating sub-systems with separate teams developing each.
  - Frequently, these teams work in different places, and even in different time zones.
  - A significant fraction of the development may be concerned with sub-system configuration (to support integration) rather than original code development. (⇒ high communication overhead)

- Most large systems are required to interact with other, existing systems, and so don’t lend themselves to requirements flexibility and incremental development.

(cont’d)
Large systems development (cont’d)

- Large systems and their development processes are often constrained by external rules and regulations that limit the way that they can be developed.

- Large systems have a long procurement and development time. It is difficult to maintain coherent teams that know about the system over long periods.

- Large systems usually have a diverse set of stakeholders. It is practically impossible to involve all of them in the development process.
Two perspectives on “scaling” agile methods

- “Scaling-up:” using agile methods for developing large software systems that cannot be developed by a small team.

- “Scaling-out:” how agile methods can be introduced across a large organization with years of (traditional) software development experience. classic “technology transfer” problem

- In either case, it is essential to maintain agile fundamentals (i.e., flexible planning, frequent system releases, continuous integration, test-driven development, and good team communications).
Scaling-up to large systems

- For large systems development, it is impractical to focus solely on code; more up-front design and system documentation is required.

- Cross-team communication mechanisms must be used (e.g., phone/video/electronic conferences and meetings where teams update each other on their progress/problems).

- Continuous integration (the whole system is built every time a developer checks-in a change) is practically impossible. But it is essential to maintain frequent system builds and regular releases.
Scaling-out in large companies: some impediments…

- Project managers (and procurers) inexperienced with agile methods may be reluctant to accept the risk of a new approach.

- Large organizations often have bureaucratic quality procedures and standards that all projects are expected to follow. These are likely to be incompatible with agile methods.

(cont’d)
Scaling-out in large companies: some impediments… (cont’d)

- Agile methods seem to work best when team members have **high skill levels**. But there are likely to be a **wide range of skill levels and abilities within large organizations**.

- There may be **cultural resistance** to agile methods, especially in those organizations that have a long history of using conventional systems engineering processes.

(See *Grady, Van Slack*, "Key Lessons in Achieving Widespread Inspection Use," *IEEE Software*, July, 1994, pp. 46-57, for general insights about successful technology transfer.)
Key points

- **Agile methods** are incremental development methods that focus on rapid development, frequent software releases, reducing process overhead, and producing high-quality code. They involve the customer directly in the development process.

- The decision on whether to use an agile or a plan-driven development approach should depend on the type of software being developed, the capabilities of the development team, and the culture of the company developing the system.

(cont’d)
Key points (cont’d)

- **Extreme programming** is a well-known agile method that **integrates a range of good programming practices** such as frequent releases of the software, continuous software improvement, and customer participation in the development team.

- A particular strength of XP is **the development of automated tests before code is written**. All tests must successfully execute when an increment is integrated into a system.

(cont’d)
Key points (cont’d)

- **Scrum** is an **iterative development management approach** that provides a **project management framework for agile methods**. Its focus is a set of **sprints**, which are fixed time periods during which a system increment is developed.

- **Scaling agile methods is difficult**. Large systems require up-front design and some documentation. Introducing and sustaining the use of agile methods across a large organization is a **lengthy process of cultural change**.
Chapter 3

Agile Software Development