

Chapter 3

Agile Software Development

(57p.)

Because these businesses are operating in a changing environment, it is often practically impossible to derive a complete set of stable software requirements. The initial requirements inevitably change because customers find it impossible to predict how a system will affect working practices, how it will interact with other systems and what user operations should be automated. It may only be after a system has been delivered and users gain experience with it that the real requirements become clear. Even then, the requirements are likely to change quickly and unpredictably due to external factors. The software may then be out-of date when it is delivered.

Software development processes that plan on completely specifying the requirements then designing, building and testing the system are not geared to rapid software development. As the requirements change or as requirements problems are discovered, the system design or implementation has to be reworked and retested. As a consequence, a conventional waterfall or specification-based process is usually prolonged and the final software is delivered to the customer long after it was originally specified.

For some types of software, such as safety-critical control systems, where a complete analysis of the system is essential, a plan-driven approach is the right one. However, in a fast-moving business environment, this can cause real problems. By the time the software is available for use, the original reason for its procurement may have changed so radically that the software is effectively useless. Therefore, for business systems in particular, development processes that focus on rapid software development and delivery are essential.

Rapid software development processes are designed to produce useful software quickly. The software is not developed as a single unit but as a series of increments, with each increment including new system functionality. Although there are many approaches to rapid software development, they share some fundamental characteristics:

1. The processes of specification, design and implementation are inter-leaved. There is no detailed system specification, and design documentation is minimized or generated automatically by the programming environment used to implement the system. The user requirements document only defines the most important characteristics of the system.

2. The system is developed in a series of versions. End-users and other system stakeholders are involved in specifying and evaluating each version. They may propose changes to the software and new requirements that should be implemented in a later version of the system.

3. System user interfaces are often developed using an interactive development system that allows the interface design to be quickly created by drawing and placing icons on the interface. The system may then generate a web-based interface for a browser or an interface for a specific platform such as Microsoft Windows.

Agile methods are incremental development methods in which the increments are small and, typically, new releases of the system are created and made available to customers every two or three weeks. They involve customers in the development process to get rapid feedback on changing requirements. They minimize documentation by using informal communications rather than formal meetings with written documents.

Topics covered

- ❑ Agile methods
- ❑ Plan-driven and agile development
- ❑ Extreme programming
- ❑ Agile project management
- ❑ Scaling agile methods

Rapid software development

- ❑ **Rapid development and delivery** 가 가장 중요한 요구사항!
 - Businesses operate in a fast – changing requirement
 - stable requirements 작성은 불가능.
 - Software has to evolve quickly
- ❑ Rapid software development
 - Specification, design and implementation are **inter-leaved**
 - System is developed as a series of versions with stakeholders involved in version evaluation
 - UI는 주로 IDE, graphical toolset을 이용하여 개발됨.

3.1 Agile methods

Agile methods

- ❑ 기존 SW 설계기법에 대한 불만에서 나옴:
 - Focus on the code rather than the design
 - based on an iterative approach to software development
 - intended to deliver working software quickly and evolve this quickly to meet changing requirements.
- ❑ Agile methods의 목적
 - to reduce overheads in the software process (e.g. by limiting documentation)
 - to be able to respond quickly to changing requirements without excessive rework.

Agile manifesto

- ❑ *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.*

The principles of agile methods

Principle	Description
Customer involvement	Customers should be closely involved throughout the development process. Their role is provide and prioritize new system requirements and to evaluate the iterations of the system.
Incremental delivery	The software is developed in increments with the customer specifying the requirements to be included in each increment.
People not process	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.
Embrace change	Expect the system requirements to change and so design the system to accommodate these changes.
Maintain simplicity	Focus on simplicity in both the software being developed and in the development process. Wherever possible, actively work to eliminate complexity from the system.

Agile method applicability

- ❑ **a small or medium-sized** product for sale.
- ❑ Custom system development within an organization, where there is a clear commitment from the **customer** to become involved in the development process and where there are **not a lot of 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

- ❑ customer 개입이 어려움.
- ❑ team member 문제
- ❑ prioritizing changes 문제
- ❑ simplicity 유지가 extra work을 초래
- ❑ contracts 문제

(61p.)

Formal documentation is supposed to describe the system and so make it easier for people changing the system to understand. In practice, however, formal documentation is often not kept up-to-date and so does not accurately reflect the program code. For this reason, agile methods enthusiasts argue that it is a waste of time to write this documentation and that the key to implementing maintainable software is to produce high-quality, readable code. Agile practices therefore emphasize the importance of writing well-structured code and investing effort in code improvement. Therefore, the lack of documentation should not be a problem in maintaining systems developed using an agile approach.

However, my experience of system maintenance suggests that the key document is the system requirements document, which tells the software engineer what the system is supposed to do. Without such knowledge, it is difficult to assess the impact of proposed system changes. Many agile methods collect requirements informally and incrementally and do not create a coherent requirements document. In this respect, the use of agile methods is likely to make subsequent system maintenance more difficult and expensive.

Agile methods and software maintenance

- ❑ agile methods가 성공하려면...
 - they have to support maintenance as well as original development.
- ❑ Two key issues:
 - 유지보수성
 - Are systems that are developed using an agile approach maintainable, given the emphasis in the development process of minimizing formal documentation?
 - 변경 요청 반영
 - Can agile methods be used effectively for evolving a system in response to customer change requests?
- ❑ original 개발 팀이 지속되지 않을 경우 문제.

Supporters of agile methods have been evangelical in promoting their use and have tended to overlook their shortcomings. This has prompted an equally extreme response, which, in my view, exaggerates the problems with this approach (Stephens and Rosenberg, 2003). More reasoned critics such as DeMarco and Boehm (DeMarco and Boehm, 2002) highlight both the advantages and disadvantages of agile methods. They propose a hybrid approach where agile methods incorporate some techniques from plan-driven development may be the best way forward. (62p.)

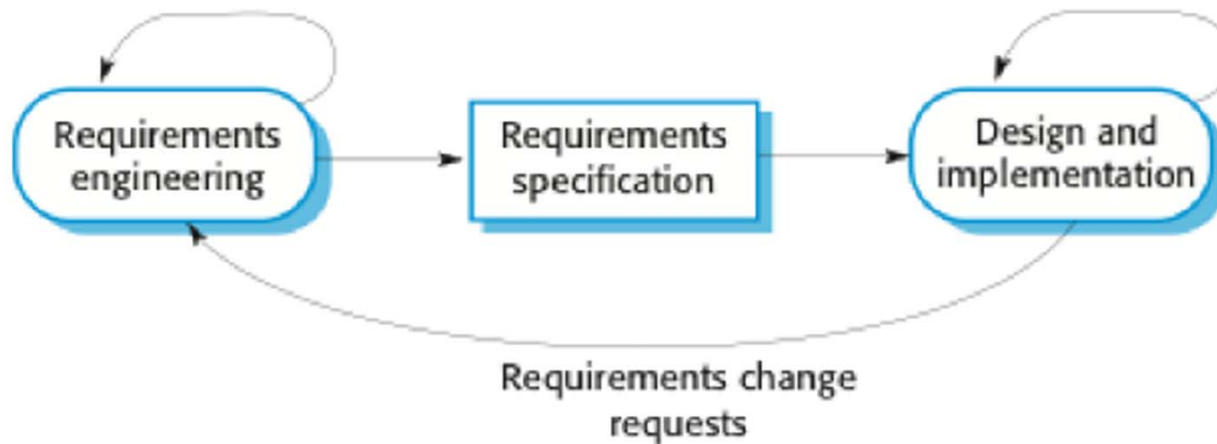
3.2 Plan-driven and agile development

Plan-driven and agile development

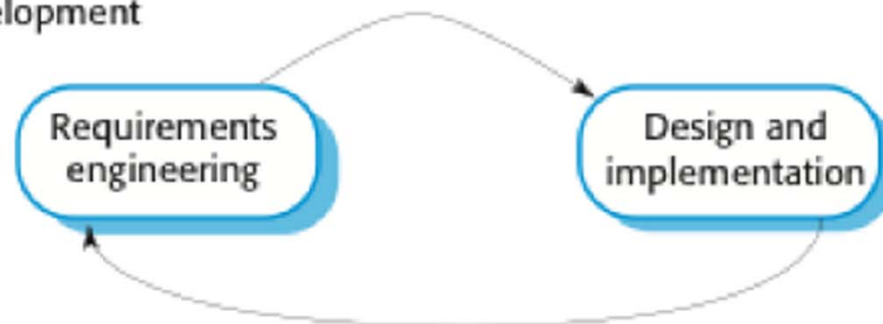
- ❑ Plan-driven development
 - based around separate development stages with the outputs to be produced at each of these stages planned in advance.
 - 반드시 waterfall model일 필요 없음
 - plan-driven, incremental development is possible
 - Iteration occurs **within activities**.
- ❑ Agile development
 - Specification, design, implementation and testing are inter-leaved and the outputs from the development process are decided through a process of negotiation during the software development process.
 - Iteration occurs **across activities**.

Plan-driven and agile specification

Plan-based development



Agile development



Technical, human, organizational issues


- ❑ 대부분의 프로젝트는 plan-driven과 agile 요소를 다 가지고 있음...
 - Is it important to have a very detailed specification and design before moving to implementation? If so, you probably need to use a plan-driven approach.
 - Is an incremental delivery strategy, where you deliver the software to customers and get rapid feedback from them, realistic? If so, consider using agile methods.
 - 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 who can communicate informally. This may not be possible for large systems that require larger development teams so a plan-driven approach may have to be used.

Technical, human, organizational issues

- 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 system 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
- 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.

Technical, human, organizational issues

- Are there cultural or organizational issues that may affect the system development?
 - Traditional engineering organizations have a culture of plan-based development, as this is the norm in engineering.
- How good 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 approve 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.



In reality, the issue of whether a project can be labelled as plan-driven or agile is not very important. Ultimately, the primary concern of buyers of a software system is whether or not they have an executable software system that meets their needs and does useful things for the individual user or the organization. In practice, many companies who claim to have used agile methods have adopted some agile practices and have integrated these with their plan-driven processes.

3.3 Extreme programming

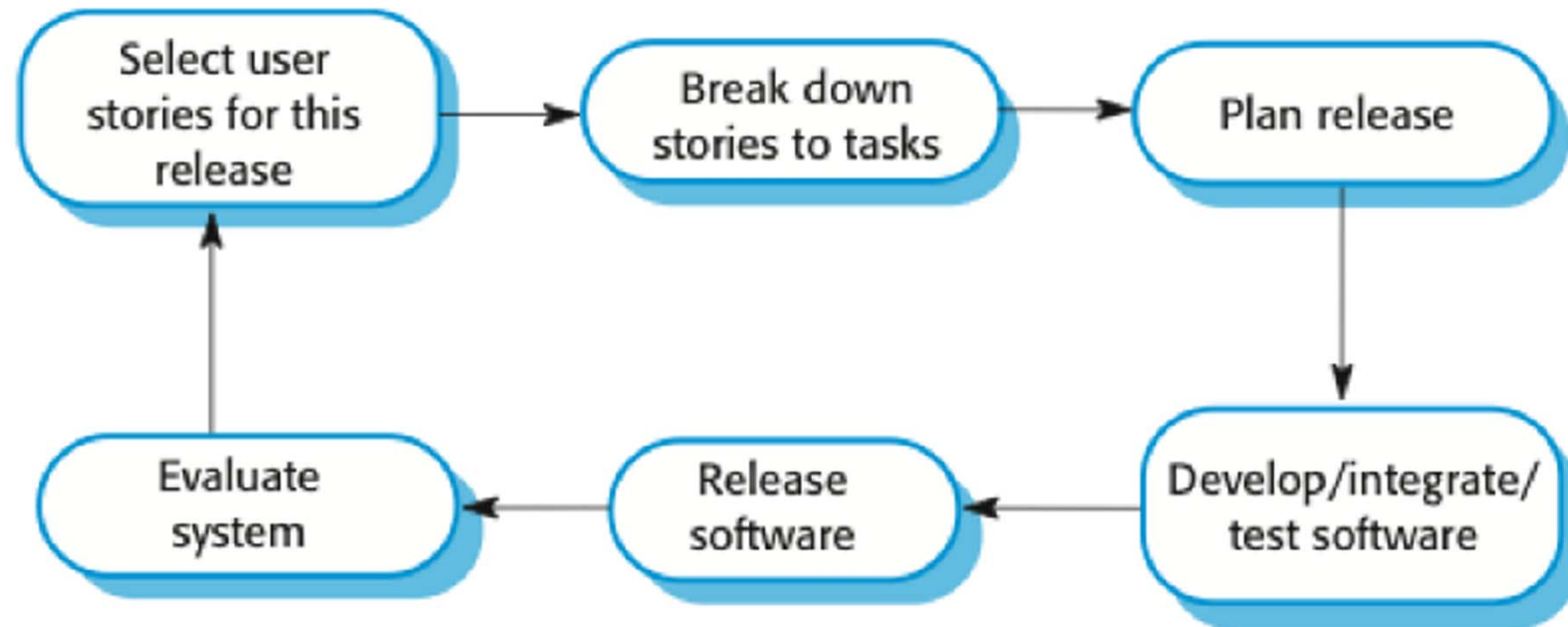
Extreme programming

- ❑ 가장 널리 사용되는 agile method by Kent Beck (2000).
- ❑ Extreme Programming (XP) 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 be run for every build and the build is only accepted if tests run successfully.

XP and agile principles

- ❑ Incremental development is supported through small, frequent system releases.
- ❑ Customer involvement means full-time customer engagement with the team.
- ❑ People not process through pair programming, collective ownership and a process that avoids long working hours.
- ❑ Change supported through regular system releases.
- ❑ Maintaining simplicity through constant refactoring of code.

The extreme programming release cycle



Extreme programming practices (a)

Principle or practice	Description
Incremental planning	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.
Small releases	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.
Simple design	Enough design is carried out to meet the current requirements and no more.
Test-first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.
Refactoring	All developers are expected to refactor the code continuously as soon as possible code improvements are found. This keeps the code simple and maintainable.

Extreme programming practices (b)

Pair programming	Developers work in pairs, checking each other's work and providing the support to always do a good job.
Collective ownership	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.
Continuous integration	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.
Sustainable pace	Large amounts of overtime are not considered acceptable as the net effect is often to reduce code quality and medium term productivity
On-site customer	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.

Requirements scenarios

- ❑ customer나 user가 XP team의 일원
- ❑ user requirement가 시나리오나 user story로 표현
- ❑ story card에 작성되며, 개발팀이 implementation task로 나눔
 - task는 일정 및 비용 추정의 기반이 됨.
- ❑ customer는 story들의 우선순위나 일정 추정에 따라 참여할 story를 결정.

A 'prescribing medication' story

Prescribing medication

The record of the patient must be open for input. Click on the medication field and select either 'current medication', 'new medication' or 'formulary'.

If you select 'current medication', you will be asked to check the dose; If you wish to change the dose, enter the new dose then confirm the prescription.

If you choose, 'new medication', the system assumes that you know which medication you wish to prescribe. Type the first few letters of the drug name. You will then see a list of possible drugs starting with these letters. Choose the required medication. You will then be asked to check that the medication you have selected is correct. Enter the dose then confirm the prescription.

If you choose 'formulary', you will be presented with a search box for the approved formulary. Search for the drug required then select it. You will then be asked to check that the medication you have selected is correct. Enter the dose then confirm the prescription.

In all cases, the system will check that the dose is within the approved range and will ask you to change it if it is outside the range of recommended doses.

After you have confirmed the prescription, it will be displayed for checking. Either click 'OK' or 'Change'. If you click 'OK', your prescription will be recorded on the audit database. If you click 'Change', you reenter the 'Prescribing medication' process.

Examples of task cards for prescribing medication

Task 1: Change dose of prescribed drug

Task 2: Formulary selection

Task 3: Dose checking

Dose checking is a safety precaution to check that the doctor has not prescribed a dangerously small or large dose.

Using the formulary id for the generic drug name, lookup the formulary and retrieve the recommended maximum and minimum dose.

Check the prescribed dose against the minimum and maximum. If outside the range, issue an error message saying that the dose is too high or too low. If within the range, enable the 'Confirm' button.

XP and change

- ❑ Conventional wisdom in software engineering is to design for change. It is worth spending time and effort anticipating changes as this reduces costs later in the life cycle.
- ❑ XP, however, maintains that this is not worthwhile as changes cannot be reliably anticipated.
- ❑ Rather, it proposes constant code improvement (refactoring) to make changes easier when they have to be implemented.

Refactoring

- ❑ Programming team look for possible software improvements and make these improvements even where there is no immediate need for them.
- ❑ This improves the understandability of the 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 and this is much more expensive.

Examples of refactoring

- ❑ Re-organization of a class hierarchy to remove duplicate code.
- ❑ Tidying up and renaming attributes and methods to make them easier to understand.
- ❑ The replacement of inline code with calls to methods that have been included in a program library.

Testing in XP

- ❑ Testing은 XP의 중심
 - 매 변경마다 프로그램이 test됨.
- ❑ XP testing features:
 - Test-first development.
 - Incremental test development from scenarios.
 - User involvement in test development and validation.
 - Automated test harnesses are used to run all component tests each time that a new release is built.

Test-first development

- ❑ Writing tests before code clarifies the requirements to be implemented.
- ❑ Tests are written as programs rather than data
 - executed automatically.
 - the test includes a check that it has executed correctly.
 - Usually relies on a testing framework such as JUnit.
- ❑ All previous and new tests are run automatically when new functionality is added, thus checking that the new functionality has not introduced errors.

Customer involvement

- ❑ Testing에서 customer의 역할
 - to help develop acceptance tests for the stories that are to be implemented in the next release of the system.
- ❑ 팀 일원인 customer는 개발이 진행되면 test작성
 - All new code is therefore validated to ensure that it is what the customer needs.
- ❑ Customer역할을 하는 사람은 full time으로 일하기 어려움
 - They may feel that providing the requirements was enough of a contribution and so may be reluctant to get involved in the testing process.

Test case description for dose checking

Test 4: Dose checking

Input:

1. A number in mg representing a single dose of the drug.
2. A number representing the number of single doses per day.

Tests:

1. Test for inputs where the single dose is correct but the frequency is too high.
2. Test for inputs where the single dose is too high and too low.
3. Test for inputs where the single dose * frequency is too high and too low.
4. Test for inputs where single dose * frequency is in the permitted range.

Output:

OK or error message indicating that the dose is outside the safe range.

Test automation

- ❑ Tests are written as executable components before the task is implemented
 - These testing components should be stand-alone, should simulate the submission of input to be tested and should check that the result meets the output specification. An automated test framework (e.g. Junit) is a system that makes it easy to write executable tests and submit a set of tests for execution.
- ❑ As testing is automated, there is always a set of tests that can be quickly and easily executed
 - Whenever any functionality is added to the system, the tests can be run and problems that the new code has introduced can be caught immediately.

XP testing difficulties

- ❑ Programmers prefer programming to testing and sometimes they take short cuts when writing tests.
 - For example, they may write incomplete tests that do not check for all possible exceptions that may occur.
- ❑ Some tests can be very difficult to write incrementally.
 - For example, in a complex user interface, it is often difficult to write unit tests for the code that implements the 'display logic' and workflow between screens.
- ❑ It difficult to judge the completeness of a set of tests.
 - Although you may have a lot of system tests, your test set may not provide complete coverage.

Pair programming

- ❑ In XP, programmers work in pairs, sitting together to develop code.
 - develop common ownership of code
 - spreads knowledge across the team.
 - serves as an informal review process as each line of code is looked at by more than 1 person.
 - encourages refactoring as the whole team can benefit from this.

- ❑ Measurements suggest that development productivity with pair programming is similar to that of two people working independently.

Pair programming

- ❑ Programmers sit together at the same workstation to develop the software.
- ❑ Pairs are created dynamically
 - all team members work with each other during the development process.
- ❑ The sharing of knowledge
 - it reduces the overall risks to a project when team members leave.

Advantages of pair programming

- ❑ collective ownership and responsibility for the system.
 - Individuals are not held responsible for problems with the code. Instead, the team has collective responsibility for resolving these problems.
- ❑ an informal review process
 - each line of code is looked at by at least two people.
- ❑ support refactoring, which is a process of software improvement.
 - Where pair programming and collective ownership are used, others benefit immediately from the refactoring so they are likely to support the process.

Pair Programming의 효율성 관련

(71-72p.) You might think that pair programming would be less efficient than individual programming. In a given time, a pair of developers would produce half as much code as two individuals working alone. There have been various studies of the productivity of paid programmers with mixed results. Using student volunteers, Williams and her collaborators (Cockburn and Williams, 2001, Williams, et al., 2000) found that productivity with pair programming seems to be comparable with that of two people working independently. The reasons suggested are that pairs discuss the software before development so probably have fewer false starts and less rework. Furthermore, the number of errors avoided by the informal inspection is such that less time is spent repairing bugs discovered during the testing process.

However, studies with more experienced programmers (Arisholm, et al., 2007, Parrish, et al., 2004) did not replicate these results. They found that there was a significant loss of productivity compared with 2 programmers working alone. There were some quality benefits but these did not fully compensate for the pair programming overhead. Nevertheless, the sharing of knowledge that happens during pair programming is very important as it reduces the overall risks to a project when team members leave. In itself, this may make pair programming worthwhile.