Course: Software Engineering

LECTURE 3: AGILE SOFTWARE DEVELOPMENT

Yutaka Watanobe

Topics covered

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

Agile methods

Rapid software development

- Rapid development and delivery is now often the most important requirement for software systems
  - Businesses operate in a fast –changing requirement and it is practically impossible to produce a set of stable software requirements
  - Software has to evolve quickly to reflect changing business needs.
- Rapid software development
  - Specification, design and implementation are inter-leaved
  - System is developed as a series of versions with stakeholders involved in version evaluation

Agile methods

- Dissatisfaction with the overheads involved in software design methods of the 1980s and 1990s led to the creation of agile methods. These methods:
  - Focus on the **code** rather than the design
  - Are based on an **iterative approach** to software development
  - Are intended **to deliver working software quickly** and evolve this quickly to meet changing requirements.
- The aim of agile methods is to reduce overheads in the software process and to be able to respond quickly to changing requirements without excessive rework.
  - e.g. by limiting documentation

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	>	*processes and tools*
Working software	>	*comprehensive documentation*
Customer collaboration	>	*contract negotiation*
Responding to change	>	*following a plan*

Philosophies of Agile

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

Agile method applicability

1. Product development where a software company is developing **a small or medium-sized product** for sale.
2. 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

- It 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 customers.
- Maintaining simplicity requires **extra work**.
- **Contracts** may be a problem as with other approaches to iterative development.

Agile methods and software maintenance

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

Two key issues:

1. Are systems that are developed using an agile approach **maintainable**, given the emphasis in the development process of minimizing formal documentation?
2. Can agile methods be used effectively for **evolving** a system in response to customer change requests?

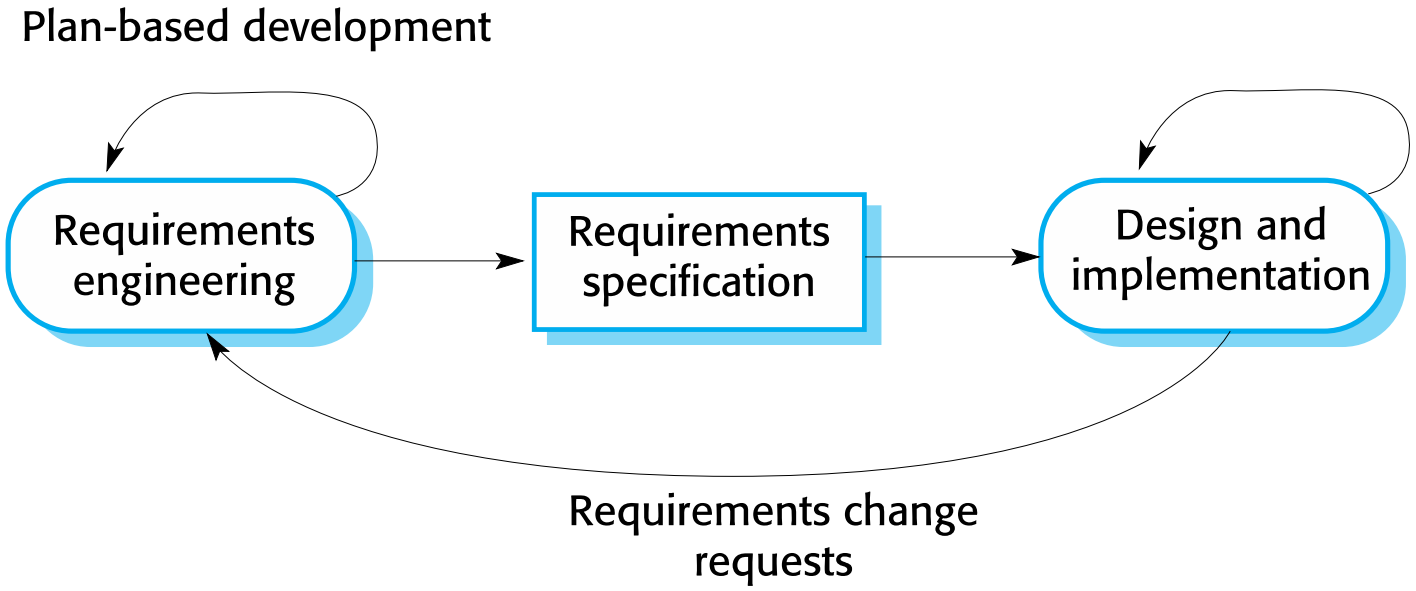
Problems may arise if original development team cannot be maintained.

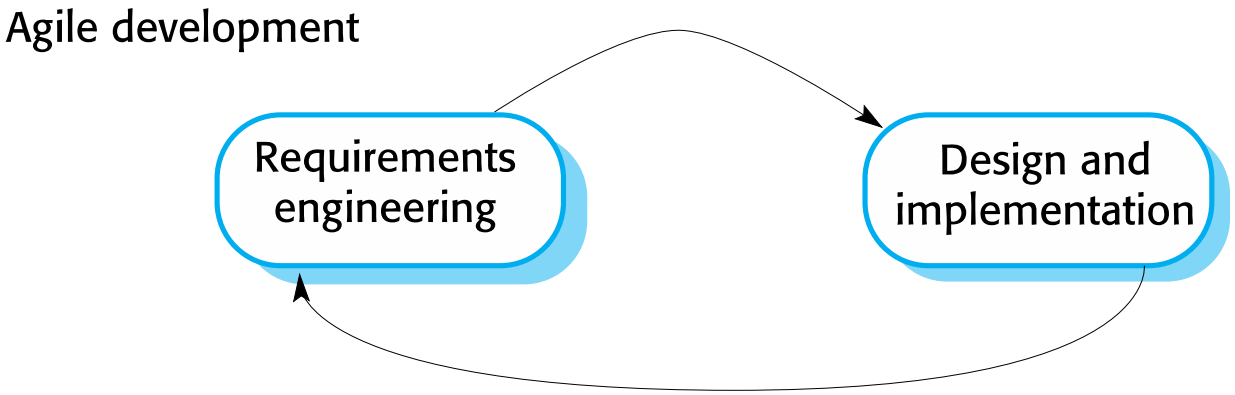
Plan-driven and agile development

Plan-driven and agile development

- **Plan-driven development**
  - A plan-driven approach to software engineering is based around **separate development stages** with the outputs to be produced at each of these stages planned in advance.
  - Not necessarily waterfall model – 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.

Plan-driven and agile specification





Technical, human, organizational Questions (1/3)

- Most projects include elements of plan-driven and agile processes. Deciding on the balance depends on:
  1. 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.
  2. 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.
  3. 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 Questions (2/3)

5. 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).
6. 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.
6. What **technologies** are available to support system development?
   - Agile methods rely on good tools to keep track of an evolving design
7. 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 Questions (3/3)

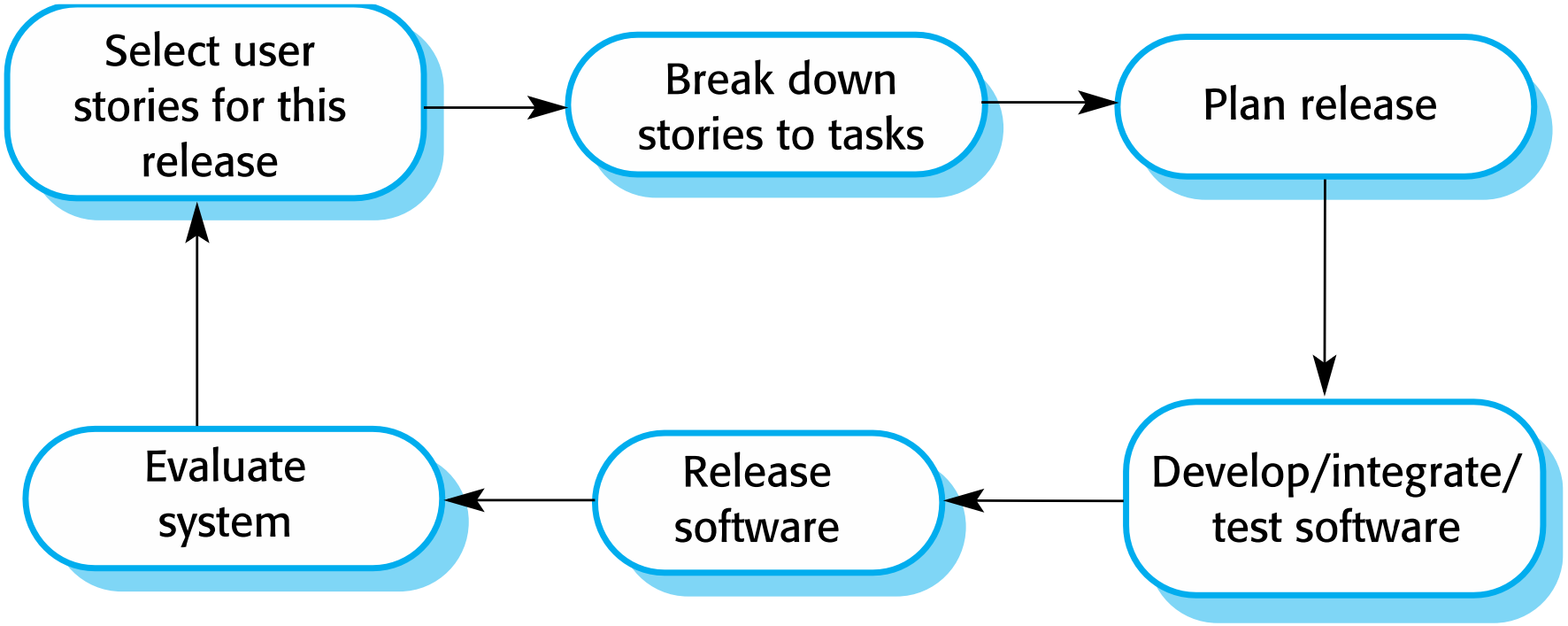
8. Are there **cultural or organizational issues** that may affect the system development?
   - Traditional engineering organizations have a culture of plan-based development, as this is the norm in engineering.
9. How good (**capability**) are the designers and programmers in the development team?
   - 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
10. Is the system subject to **external regulation**?
    - If a system has to be approved by an external regulator then you will probably be required to produce detailed documentation as part of the system safety case.

Extreme programming

Extreme programming

- Perhaps the best-known and most widely used agile method.
- **Extreme Programming (XP)** takes an ‘extreme’ approach to iterative development. Some examples:
  - 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.
  - etc.

The extreme programming release cycle



Principles of agile methods

1. Incremental development is supported through small, **frequent system releases**.
2. Customer involvement means **full-time customer engagement** with the team.
3. People, not process, are supported through **pair programming**, collective ownership, and a process that avoids long working hours.
4. **Change** supported through **regular system releases**, **test-first development,** refactoring, and **continuous integration**.
5. Maintaining **simplicity** through constant **refactoring** of code.

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'.
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

- In XP, a customer or user is part of the XP team and is responsible for making decisions on requirements.
- User requirements are expressed as **scenarios** or user **stories**.
- These are written on **cards** and the development team break them down into implementation **tasks**. These tasks are the basis of schedule and cost estimates.
- The customer chooses the stories for inclusion in the next release based on their priorities and the schedule estimates.

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 (restructuring)

- 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 require 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.

Test-first development

- Writing tests before code clarifies the requirements to be implemented.
- Tests are written as programs rather than data so that they can be executed automatically.

- You can run the test as the code is being written and discover problems during development.

Customer involvement

- The role of the customer in the testing process is to help develop acceptance tests for the stories that are to be implemented in the next release of the system.
- The customer who is part of the team writes tests as development proceeds. All new code is therefore validated to ensure that it is what the customer needs.
- However, people adopting the customer role have limited time available and so cannot work full-time with the development team. 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 automation

- Test automation means that 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

1. **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.
2. 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.
3. It is 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 pair programming, programmers sit together at the same workstation to develop the software.
- Pairs are created dynamically so that all team members work with each other during the development process.
- Pair programming is not necessarily inefficient and there is evidence that a pair working together is more efficient than 2 programmers working separately.

  * There are another research which did not replicate such result

Advantages of pair programming

1. It supports the idea of **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.
2. It acts as an informal **review process** because each line of code is looked at by at least two people.
3. It helps 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.

Productivity of pair programming (Pros.)

- There have been various studies of the productivity of paired programmers with mixed results.
- A research found that productivity with pair programming seems to be comparable with that of two people working independently. Why?
  - pairs discuss the software before development so probably have fewer false starts and less rework.
  - the number of errors avoided by the informal inspection is such that less time is spent repairing bugs discovered during the testing process.

Productivity of pair programming (Cons.)

- However, studies with more experienced programmers did not replicate these results
  - They found that there was a significant loss of productivity compared with two 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.

Agile project management

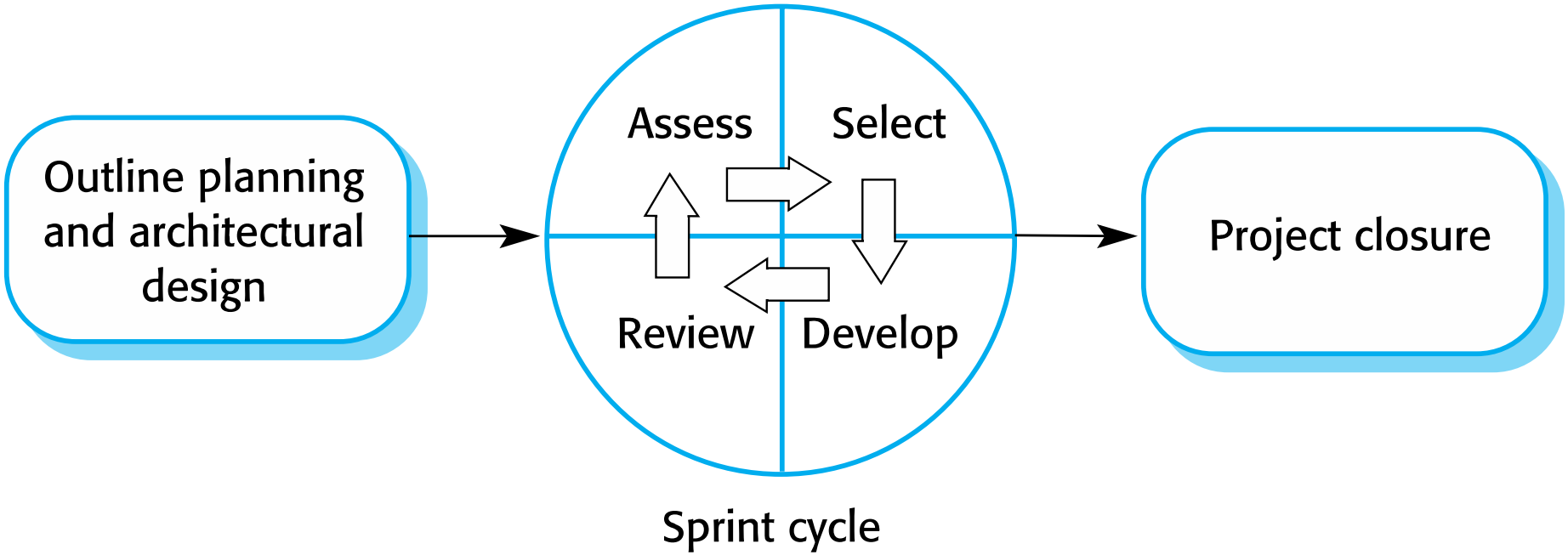
Agile project management

- The principal responsibility of software project managers is to manage the project so that the software is delivered on time and within the planned budget for the project.
- The standard approach to project management is plan-driven. Managers draw up a plan for the project showing
  - what should be delivered,
  - when it should be delivered and
  - who will work on the development of the project deliverables.
- Agile project management requires a different approach, which is adapted to **incremental development** and the particular strengths of agile methods.

Scrum

- The Scrum approach is a general agile method but its focus is on managing **iterative development** rather than specific agile practices.
- There are three phases in Scrum.
    1. The initial phase is an outline planning phase where you establish the general objectives for the project and design the software architecture.
    2. This is followed by a series of sprint cycles, where each cycle develops an increment of the system.
    3. The project closure phase wraps up the project, completes required documentation such as system help frames and user manuals and assesses the lessons learned from the project.

The Scrum process



The Sprint cycle (1)

1. Sprints are **fixed length**, normally 2–4 weeks. They correspond to the development of a release of the system in XP.
2. The starting point for planning is the product **backlog**, which is the list of work to be done on the project.
3. The selection phase involves all of the project team who work with the customer to select the features and functionality to be developed during the sprint.

The Sprint cycle (2)

4. Once these are agreed, the team organize themselves to develop the software. During this stage the team is isolated from the customer and the organization, 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.
5. At the end of the sprint, the work done 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 against the backlog and communicates with customers and management outside of the team.
- The whole team attends short daily meetings where all team members share information, describe their progress since the last meeting, problems that have arisen and what is planned for the following day.
  - This means that everyone on the team knows what is going on and, if problems arise, can re-plan short-term work to cope with them.

Scrum benefits

1. The product is broken down into a set of manageable and understandable chunks.
2. Unstable requirements do not hold up progress.
3. The whole team have visibility of everything and consequently team communication is improved.
4. Customers see on-time delivery of increments and gain feedback on how the product works.
5. Trust between customers and developers is established and a positive culture is created in which everyone expects the project to succeed.

Scaling agile methods

Scaling agile methods

- Agile methods were developed for use by small programming teams who could work together in the same room and communicate informally.
- Agile methods have therefore been mostly used for the development of small and medium-sized systems.

- The need for faster delivery of software, which is more suited to customer needs, also applies to larger systems.
- There has been a great deal of interest in scaling agile methods to cope with larger systems, developed by large organizations.

Large system development

1. Large systems are usually collections of separate, communicating systems, where separate **teams** develop each system. Frequently, these teams are working in different places, sometimes in different time zones.
2. Large systems are ‘brownfield systems’, that is they include and interact with a number of existing systems. Many of the system requirements are concerned with this **interaction** and so don’t really lend themselves to flexibility and incremental development.
3. Where several systems are integrated to create a system, a significant fraction of the development is concerned with **system configuration** rather than original code development.

Large system development

4. Large systems and their development processes are often constrained by **external rules and regulations** limiting the way that they can be developed.
5. Large systems have a long procurement and **development time**. It is difficult to maintain coherent teams who know about the system over that period as, inevitably, **people move** on to other jobs and projects.
6. Large systems usually have a diverse set of **stakeholders**. It is practically impossible to involve all of these different stakeholders in the development process.

Scaling out and scaling up

- '**Scaling up**' is concerned with using agile methods for developing large **software systems** that cannot be developed by a small team.
- '**Scaling out**' is concerned with how agile methods can be introduced across a large **organization** with many years of software development experience.

Scaling up to large systems

1. For large systems development, it is not possible to focus only on the code of the system. You need to do more up-front design and system **documentation**
2. **Cross-team communication** mechanisms have to be designed and used. This should involve regular phone and video conferences between team members and frequent, short electronic meetings where teams update each other on progress.
3. **Continuous integration**, where the whole system is built every time any developer checks in a change, is practically impossible. However, it is essential to maintain **frequent system builds** and **regular releases** of the system.

Scaling out to large companies

1. **Project managers** who do not have experience of agile methods may be reluctant to accept the risk of a new approach.
2. Large organizations often have quality procedures and **standards** that all projects are expected to follow and, because of their **bureaucratic nature**, these are likely to be incompatible with agile methods.
3. Agile methods seem to work best when team members have a relatively high skill level. However, within large organizations, there are likely to be a wide range of **skills and abilities**.
4. There may be **cultural resistance** to agile methods, especially in those organizations that have a long history of using conventional systems engineering processes.

Key points

- **Agile** methods are incremental development methods that focus on rapid development, **frequent releases** of the software, reducing process overheads 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 approach to development should depend on the type of software being developed, the capabilities of the development team and the culture of the company developing the system.
- **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.

Key points

- A particular strength of extreme programming is the development of **automated tests** before a program feature is created. All tests must successfully execute when an increment is integrated into a system.
- The **Scrum** method is an agile method that provides a project management framework. It is centred round a set of sprints, which are fixed time periods when a system increment is developed.
- Scaling agile methods for large systems is difficult. Large systems need up-front design and some documentation.

References

SOMMERVILLE, IAN. “SOFTWARE ENGINEERING 10TH EDITION.