

Software Engineering

Lecture 5 – Agile Software Development



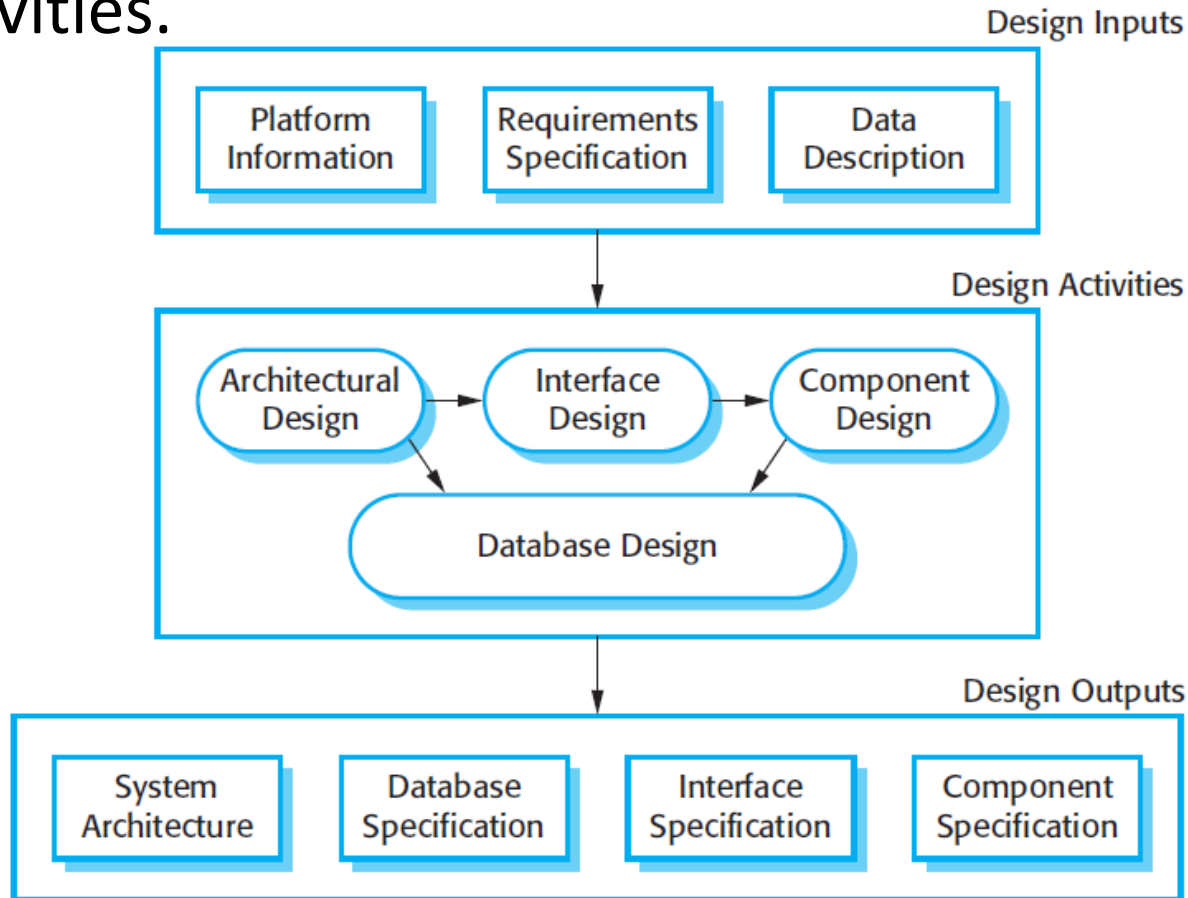
A word cloud visualization of agile software development concepts. The words are arranged in a circular pattern, with the most prominent words in the center and smaller words towards the edges. The words are in various shades of red and grey. The central words include 'change', 'software', 'Individuals', 'Working', 'and', 'collaboration', 'Responding', 'interactions', and 'to'. Other words include 'contract', 'processes', 'negotiation', 'plan', 'following', 'documentation', 'comprehensive', and 'tools'.

change software
Individuals Working
and collaboration
Responding
interactions to
contract processes
negotiation plan following
documentation comprehensive tools

JJCAO

Exercise

Describe the main activities in the software design process and the outputs of these activities. Using a diagram, show possible relationships between the outputs of these activities.

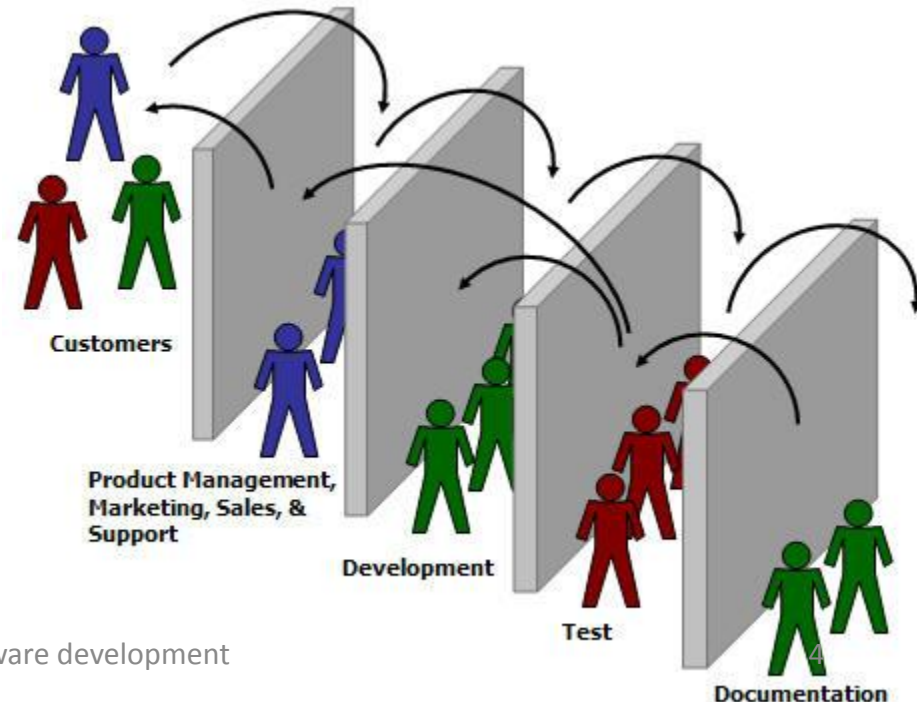


Explain why systems developed as prototypes should not normally be used as production systems

- Prototypes should be **discarded** after development as they are not a good basis for a production system:
 1. It may be **impossible** to tune the system to meet **non-functional** requirements;
 2. Prototypes are normally **undocumented**;
 3. The prototype **structure** is usually **degraded** through rapid change;
 4. The prototype probably will not meet normal organizational **quality** standards.

Topics covered

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



Rapid software development

- **Rapid development & 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**
 - User interfaces are often developed using an **IDE and graphical toolset**.

Agile methods

- **Dissatisfaction** with the **overheads** involved in software design methods of the 1980s & 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 (e.g. by **limiting documentation**) and to be able to **respond quickly** to changing requirements without excessive rework.

The Agile Manifesto

– a statement of values

Individuals and
interactions

over

Process and tools

Working software

over

Comprehensive
documentation

Customer
collaboration

over

Contract negotiation

Responding to
change

over

Following a plan

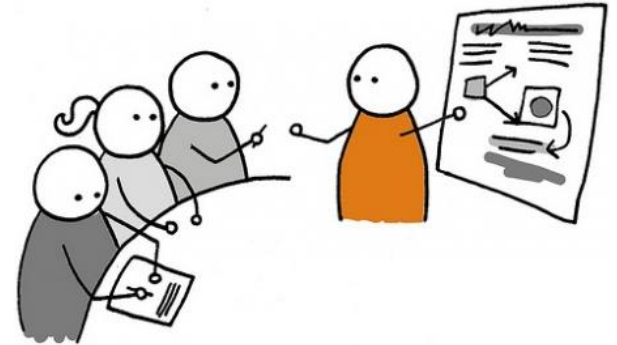
The principles of agile methods



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

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

1. It can be **difficult to keep the interest of customers** who are involved in the process.



2. Team members may be **unsuited to the intense involvement** that characterises agile methods.
3. **Prioritising** changes can be difficult where there are **multiple stakeholders**.
4. Maintaining simplicity requires extra work.
5. **Contracts may be a problem** as with other approaches to iterative development.

Agile methods and software maintenance

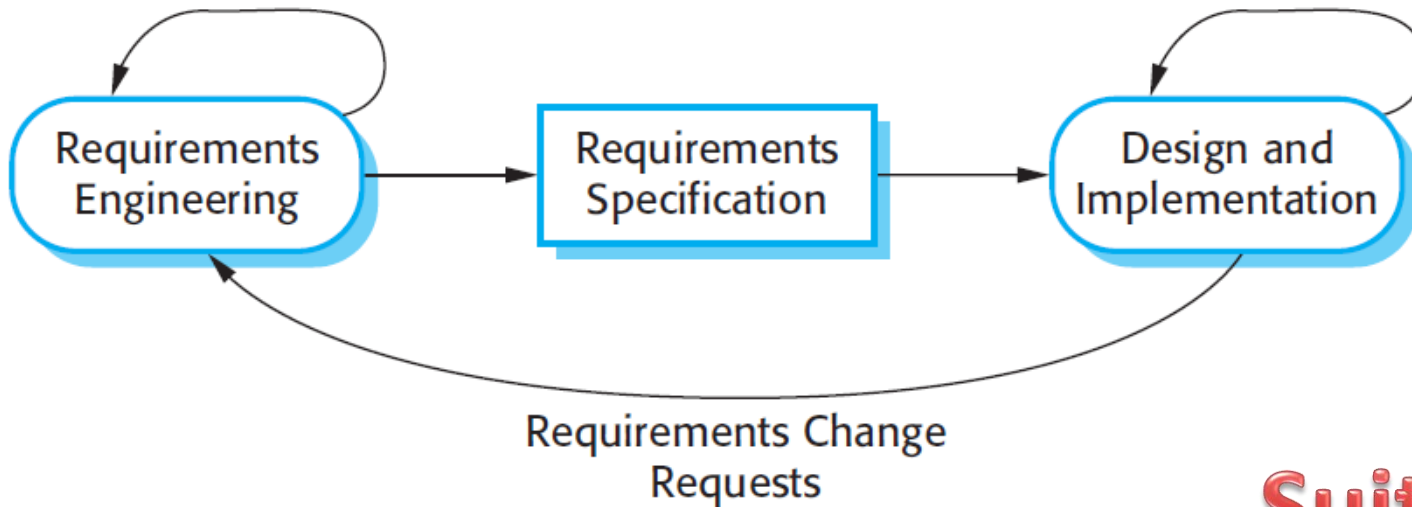
- Most organizations **spend more** on maintaining existing software than they do on new software development. So, if agile methods are to be successful, they have to **support maintenance** as well as original development.
- Two key issues:
 - Are systems that are developed using an agile approach **maintainable**, given the emphasis in the development process of **minimizing** formal **documentation**?
 - 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

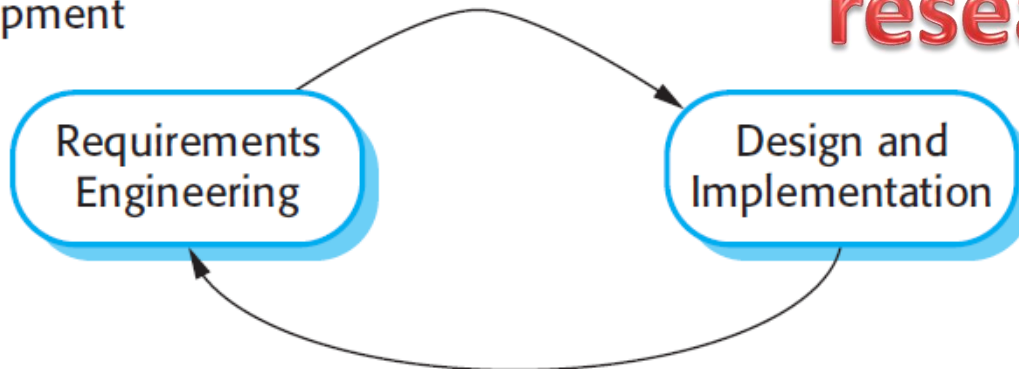
- Plan-driven development
 - A plan-driven approach to software engineering is based around separate development stages with the **outputs** to be produced at each of these stages **planned in advance**.
 - Not necessarily waterfall model – **plan-driven, incremental** development is possible
 - Iteration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

Plan-Based Development



Agile Development



**Suit our
research project**

Technical, human, organizational issues

- Most projects include elements of plan-driven and agile processes. Deciding on the **balance** depends on:
 - 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.



Technical, human, organizational issues

- 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?

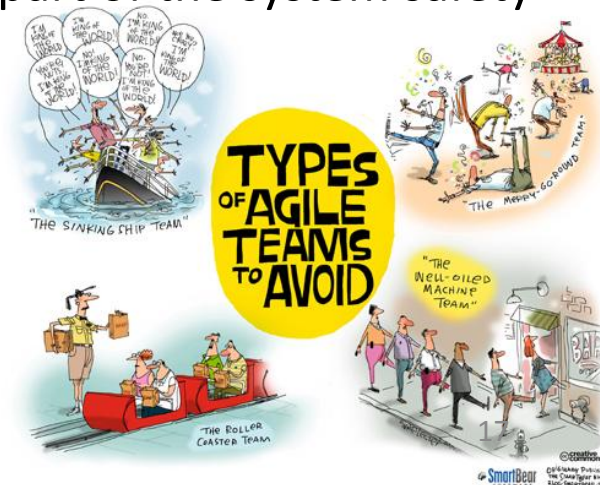
- Agile methods rely on **good tools** to keep track of an evolving design

– 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.
- 



Lean

- Complements existing practices
- Focuses on project ROI
- Eliminates project waste

Scrum

- Focuses on project management
- Self-organizing teams and feedback
- Priorities based on business value

Extreme Programming (XP)

- Strong technical practices
- Developer ownership of estimates
- Low ceremony

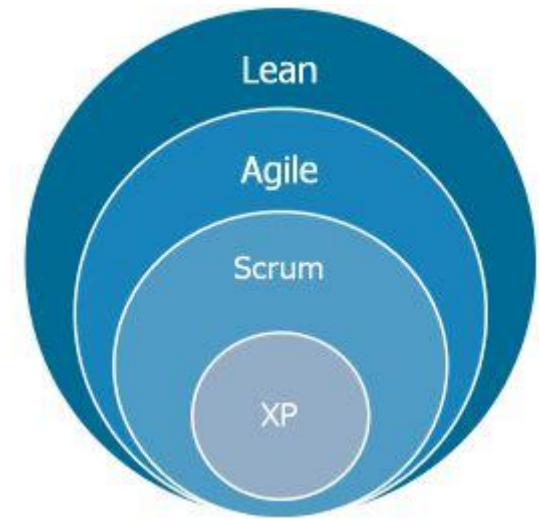
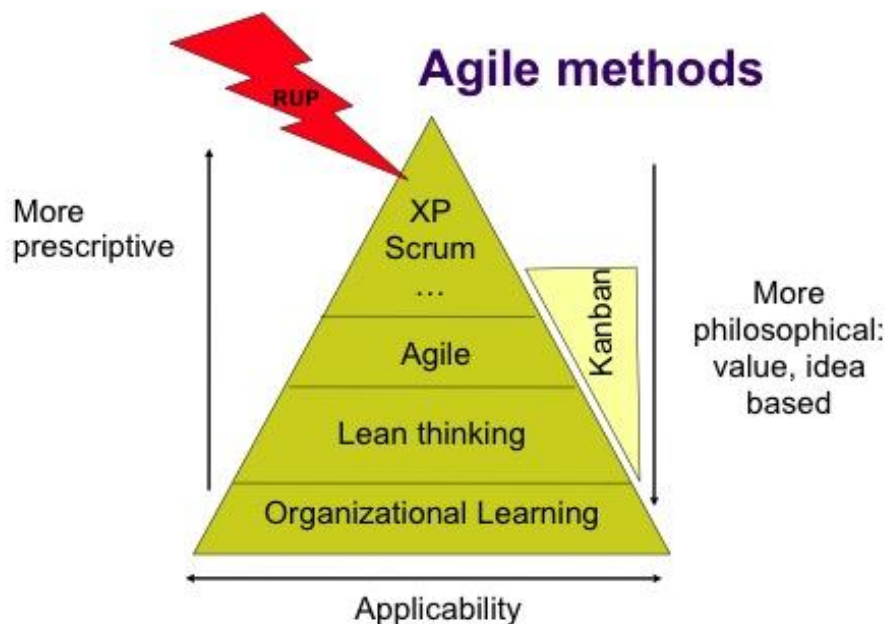


Diagram 1.2 – Lean, Scrum and XP Key Features



(c) 2006-2008 Allan Kelly, Software Strategy Ltd.

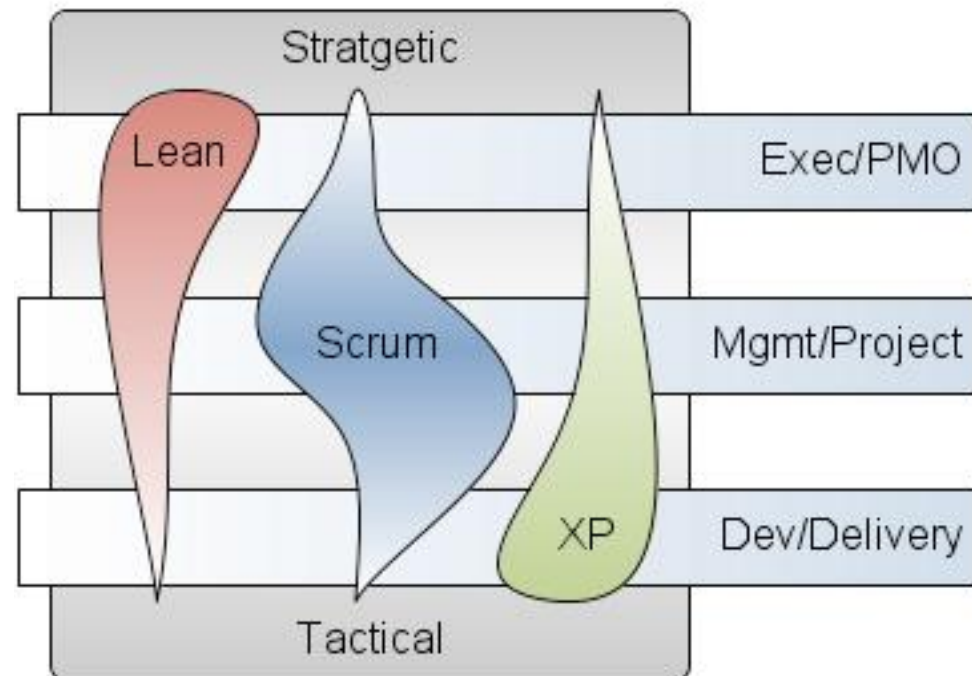


Diagram 1.5 – Applying Agile to Organizational Levels

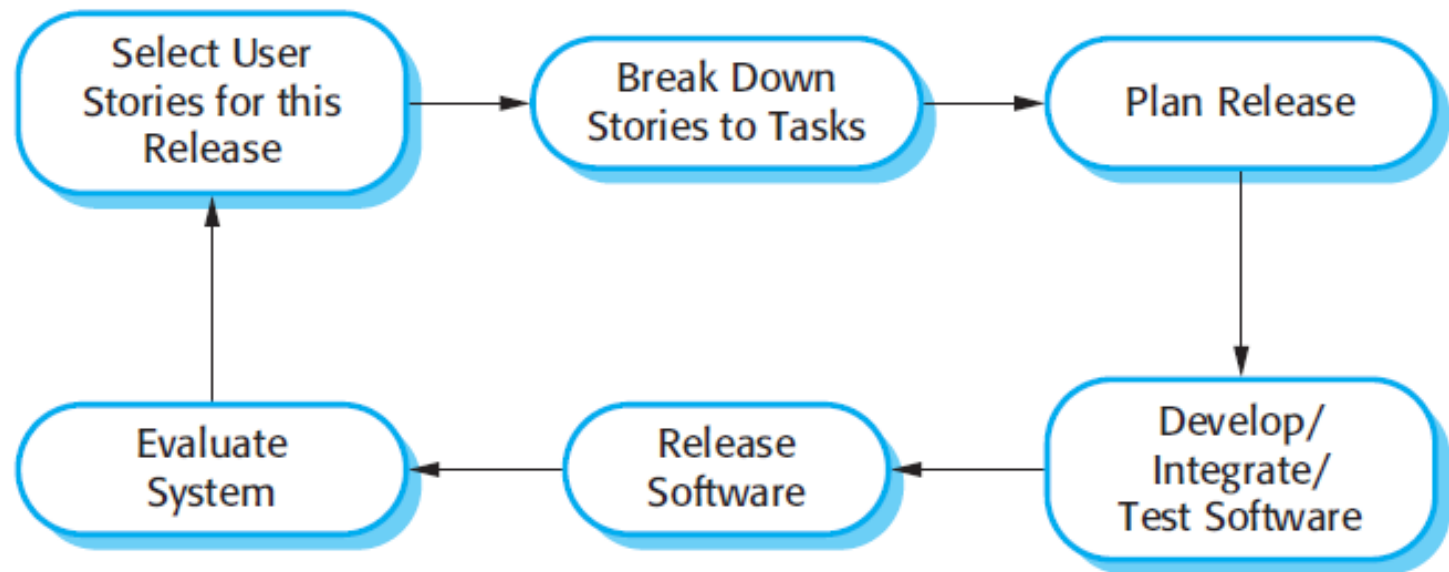
Extreme programming

- Perhaps the **best-known** and most widely used agile method.
- 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 are supported 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

- 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.

A 'prescribing medication' story

Prescribing Medication

Kate is a doctor who wishes to prescribe medication for a patient attending a clinic. The **patient record** is already **displayed** on her computer so she clicks on the **medication field** and can **select** '**current medication**', '**new medication**' or '**formulary**'.

If she selects '**current medication**', the system asks her to **check** the **dose**. If she wants to **change** the dose, she enters the dose and then **confirms** the prescription.

If she chooses '**new medication**', the system assumes that she knows which medication to prescribe. She **types the first few letters** of the drug name. The system **displays a list of possible drugs** starting with these letters. She **chooses** the required medication and the system responds by asking her to **check** that the medication selected is correct. She enters the **dose** and then **confirms** the prescription.

If she chooses '**formulary**', the system displays a **search box** for the approved formulary. She can then search for the drug required. She **selects** a drug and is asked to **check** that the medication is correct. She enters the **dose** and then **confirms** the prescription.

The system always **checks that the dose is within the approved range**. If it isn't, Kate is asked to change the dose.

After Kate has confirmed the prescription, it will be displayed for **checking**. She either clicks 'OK' or 'Change'. If she clicks 'OK', the prescription is recorded on the audit database. If she clicks on 'Change', she reenters 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, look up 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.

- Each task generates one or more **unit tests** that check the implementation described in that task.

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 \times frequency is too high and too low.
4. Test for inputs where single dose \times frequency is in the permitted range.

Output:

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

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.

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.

Chapter 3 – Agile Software Development

Lecture 6

Testing in XP

- **Testing is central** to XP and XP has developed an approach where the program is tested after every change has been made.
- XP testing features:
 1. **Test-first** development.
 2. **Incremental test** development from scenarios.
 3. **User involvement** in test development and validation.
 4. **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 so that they can be **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

- 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 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.

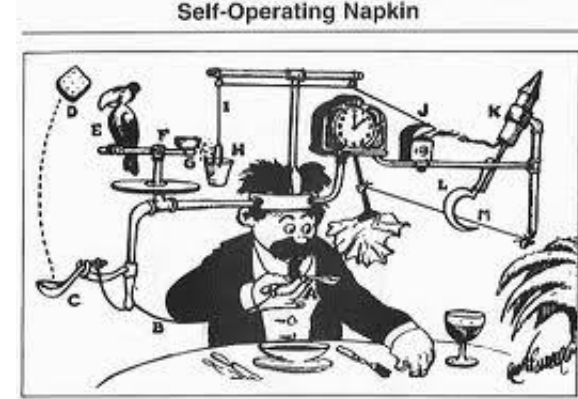
Tests:

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 \times frequency is too high and too low.
4. Test for inputs where single dose \times frequency is in the permitted range.

Output:

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

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

- 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.



- This helps develop common ownership of code and **spreads knowledge** across the team.
- It serves as an informal **review** process as each line of code is looked at by more than 1 person.
- It encourages **refactoring** as the whole team can benefit from this.

Pair programming

- Pairs are created **dynamically** so that all team members work with each other during the development process.



- 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**.
- Pair programming is not necessarily inefficient and there is evidence that a pair working together is **more efficient** than 2 programmers working separately.



Advantages of pair programming

- 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.
- It acts as an informal **review** process because each line of code is looked at by at least two people.
- 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.

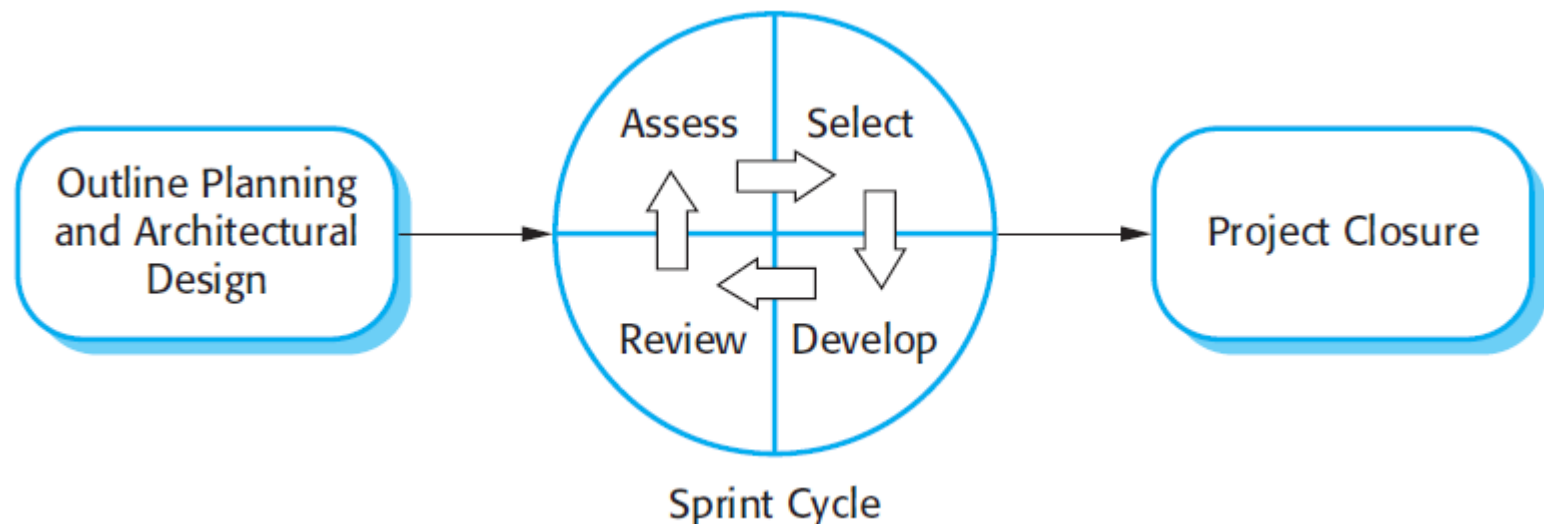
Agile project management

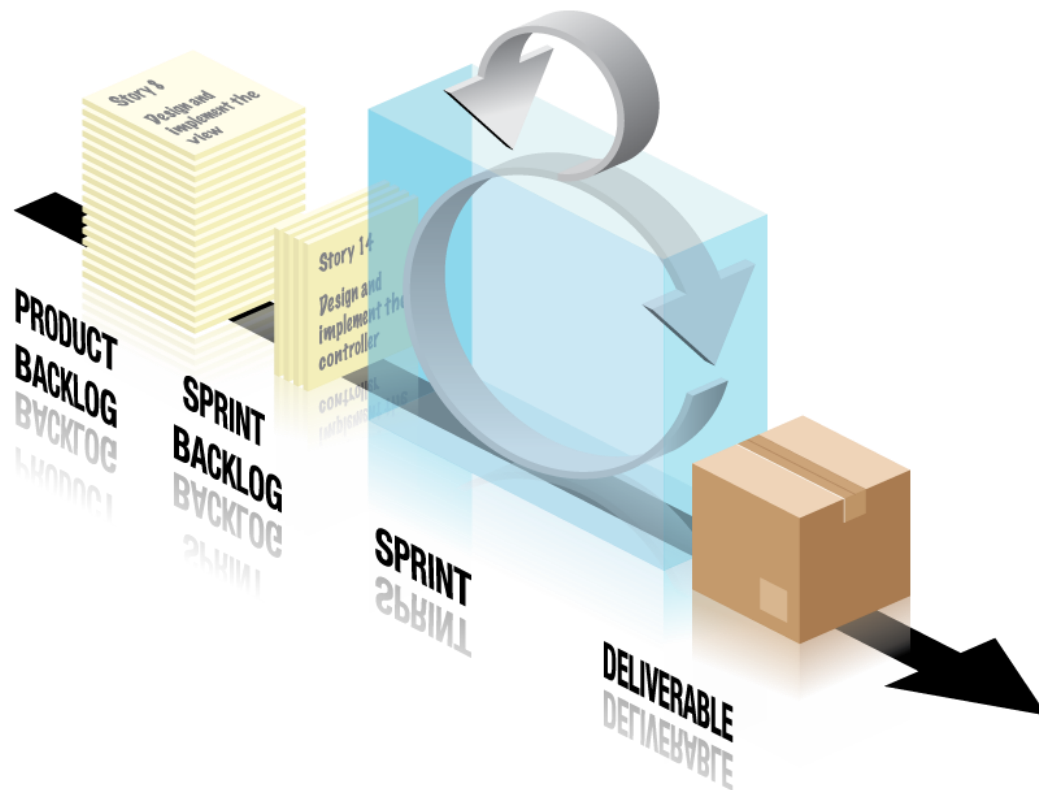
- The principal responsibility of software project managers is to manage the project so that the software is delivered on **time** and within the planned **budget** for the project.
- The standard approach to project management is plan-driven. Managers draw up a **plan** for the project showing what 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**



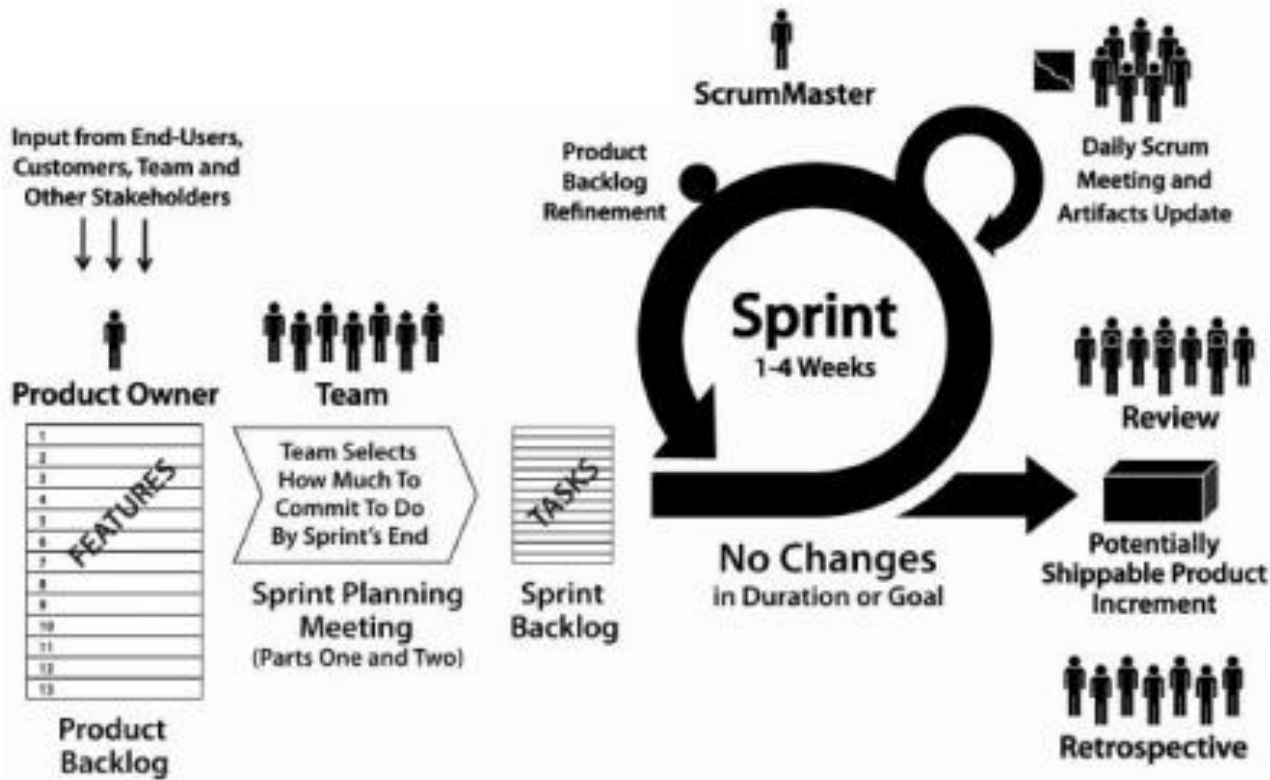
Scrum

- 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
 - The **initial** phase is an outline planning phase where you establish the **general objectives** for the project and design the software **architecture**.
 - This is followed by a series of **sprint cycles**, where each cycle develops an increment of the system.
 - 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.





- Sprints are fixed length, normally **1–4 weeks**. They correspond to the development of a release of the system in XP.
- The starting point for planning is the **product backlog**, which is the list of work to be done on the project.
- 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.



- 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**.
- 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

- The product is broken down into a set of manageable and understandable chunks.
- **Unstable requirements do not hold up** progress.
- The **whole team have visibility** of everything and consequently team communication is improved.
- Customers see on-time **delivery of increments and gain feedback** on 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

- Agile methods have proved to be successful for small and **medium sized** projects that can be developed by a small **co-located team**.
- It is sometimes argued that the **success** of these methods comes **because** of improved communications which is possible when everyone is working together.
- Scaling up agile methods involves changing these to cope with larger, longer projects where there are multiple development teams, perhaps working in different locations.

Large systems development

- 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**.
- 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**.
- 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

- Large systems and their development processes are often **constrained** by external rules and regulations limiting the way that they can be developed.
- 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.
- 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.
- When scaling agile methods it is essential to **maintain agile fundamentals**
 - 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 not possible to focus only on the code of the system. You need to do **more up-front design and system documentation**
- **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.
- 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

- Project managers who do not have experience of agile methods may be reluctant to accept the **risk of a new** approach.
- 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**.
- 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**.
- 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

- 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.