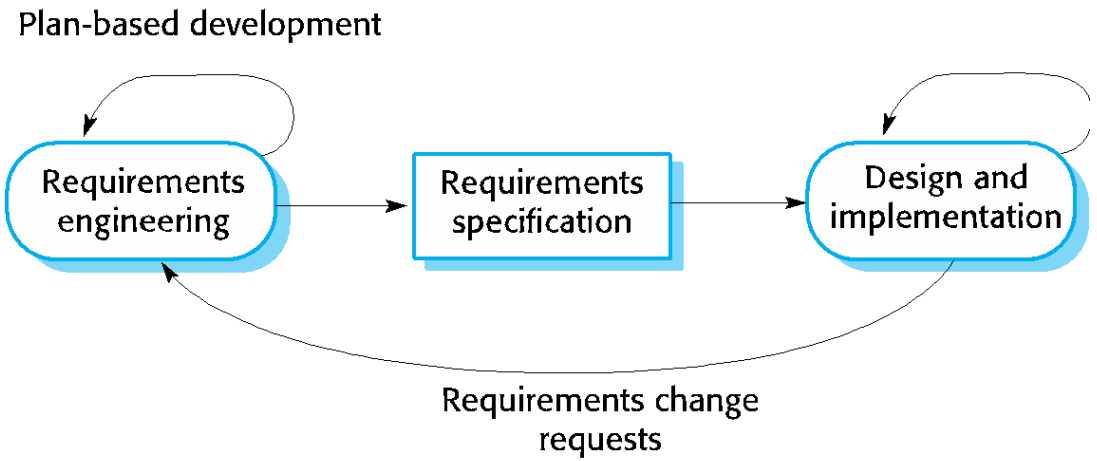
Rapid software development

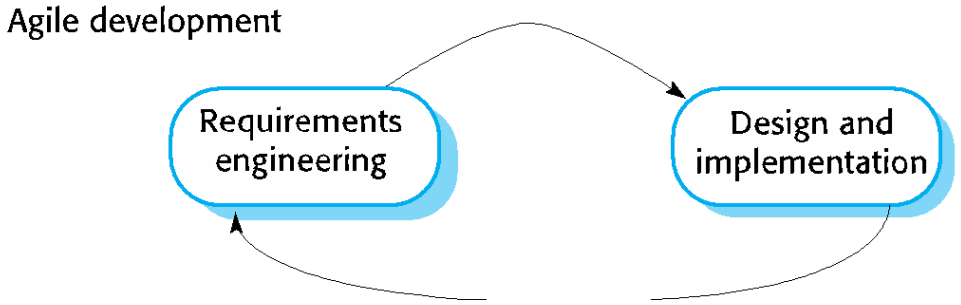
- Rapid development and delivery is now often the most important requirement for software systems
  - Businesses operate in a fast changing requirement and it is practically impossible to produce a set of stable software requirements
  - Software has to evolve quickly to reflect changing business needs.
- Plan-driven development is essential for some types of system but does not meet these business needs.
- Agile development methods emerged in the late 1990s whose aim was to radically reduce the delivery time for working software systems

Agile development

- Program specification, design and implementation are inter-leaved
- The system is developed as a series of versions or increments with stakeholders involved in version specification and evaluation
- Extensive tool support (e.g. automated testing tools) used to support development.
- Minimal documentation – focus on working code
- For progress in a project, software should be developed and delivered rapidly in small increments.
- Even late changes in the requirements should be entertained (small-increment model of development helps in accommodating them).
- Face-to-face communication is preferred over documentation.
- Continuous feedback and involvement of customer is necessary for developing good-quality software.
- Simple design which evolves and improves with time is a better approach than doing an elaborate design up front for handling all possible scenarios.

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

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.

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 (e.g. by limiting documentation) and 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.

Ref: https://agilemanifesto.org/

The principles of agile methods

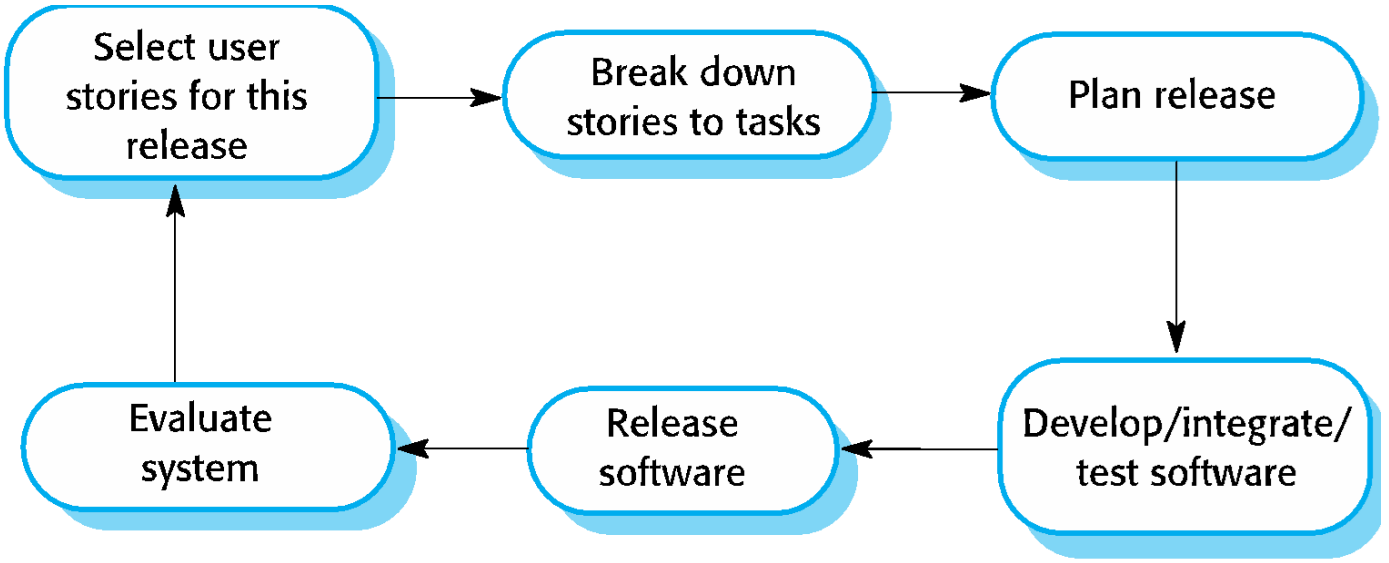
Principle	Description
Customer involvement	Customers should be closely involved throughout the development process. Their role is provide and prioritize new system requirements and to evaluate the iterations of the system.
Incremental delivery	The software is developed in increments with the customer 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

- Product development where a software company is developing a small or medium-sized product for sale.
  - Virtually all software products and apps are now developed using an agile approach
- 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 few external rules and regulations that affect the software.

Agile development technique: Extreme programming

- A very influential agile method, developed in the late 1990s, that introduced a range of agile development techniques.
- 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.



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'.
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.
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.

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.

Influential XP practices

- Extreme programming has a technical focus and is not easy to integrate with management practice in most organizations.
- Consequently, while agile development uses practices from XP, the method as originally defined is not widely used.
- Key practices
  - ✓ User stories for specification
  - ✓ Refactoring
  - ✓ Test-first development
  - ✓ Pair programming

User stories for requirements

- 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 user stories or scenarios.
- 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

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.

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.

Refactoring

- 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.
- 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 code is well-structured and clear.

Examples:

1) Re-organization of a class hierarchy to remove duplicate and dead code.
2) Tidying up and renaming attributes and methods to make them easier to understand.
3) The replacement of code with calls to methods that have been included in a program library.
4) Indenting and Formatting
5) Extracting long functions into short functions

Examples of Refactoring

```
function isUserBanned(userId) {
  let result;
  if (doesUserExist(userId)) {
    if (getUser(userId).isBanned) {
      result = true;
    } else {
      result = false;
    }
  } else {
    result = null;
  }
  return result;
}
```

```
void printOwing() {
  printBanner();
  //print details
  System.out.println ("name: " + _name);
  System.out.println ("amount      " + getOutstanding());
  }
```

```
void printOwing() {
  printBanner();
  printDetails(getOutstanding());
}
void printDetails (double outstanding) {
  System.out.println ("name: " + _name);
  System.out.println ("amount      " + outstanding);
}
```

```
function isUserBanned(userId) {
  if (!doesUserExist(userId)) return null;
  if (getUser(userId).isBanned) return true;
  return false;
}
```

Examples of Refactoring

```java
private boolean mychecker(int m, int mX, int x, int t) {
    if (m>0 && mX<100 && m<mX)
        if (m<x+t)
            if(x+t<mX) return true;
        else return false;
        else return false;
        return false;}
```

```java
/**
* Function checks if sum of two integers in between two values, which sh
* @param minValue - lower bound of the sum, should be greater than 0 and
* @param maxValue - upper bound of the sum, should be less than 100 and
* @param first - an integer value
* @param second - an integer value
* @return true if sum of first and second is between minValue and maxVal
*         false otherwise
*/

private boolean isSumInBounds(int minValue, int maxValue, int first, int
    if (minValue > 0 && maxValue < 100) {
        int sum = first + second;
        if (minValue < sum && sum < maxValue) {
            return true;
        }
    }
    return false;
}
```

Refactoring

“Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written.” - Martin Fowler

The purposes of refactoring according to M. Fowler are stated in the following:

- Refactoring Improves the Design of Software
- Refactoring Makes Software Easier to Understand
- Refactoring Helps Finding Bugs
- Refactoring Helps Programming Faster

Benefits of Code Refactoring

- Save Money: Better cope up with changes and reduce maintance load
- Improve Performance: Example most developers know which SQL queries are called together and can be combined easily.
- Reduce Risks: Easy re-use and better cope up with changes

Test-driven development

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:

- 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.

Why it is helpful?

- Writing tests before code clarifies the requirements to be implemented.

Why and how it is automated?

- 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.
- All previous and new tests are run automatically when new functionality is added, thus checking that the new functionality has not introduced errors.
- 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.

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 * 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.

Problems with test-first development

- 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

- Pair programming involves programmers working in pairs, developing code together.
- 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 improving the system code.
- 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 some evidence that suggests that a pair working together is more efficient than 2 programmers working separately.

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 practices used in agile methods.

Scrum

- Scrum is an agile method that focuses 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.

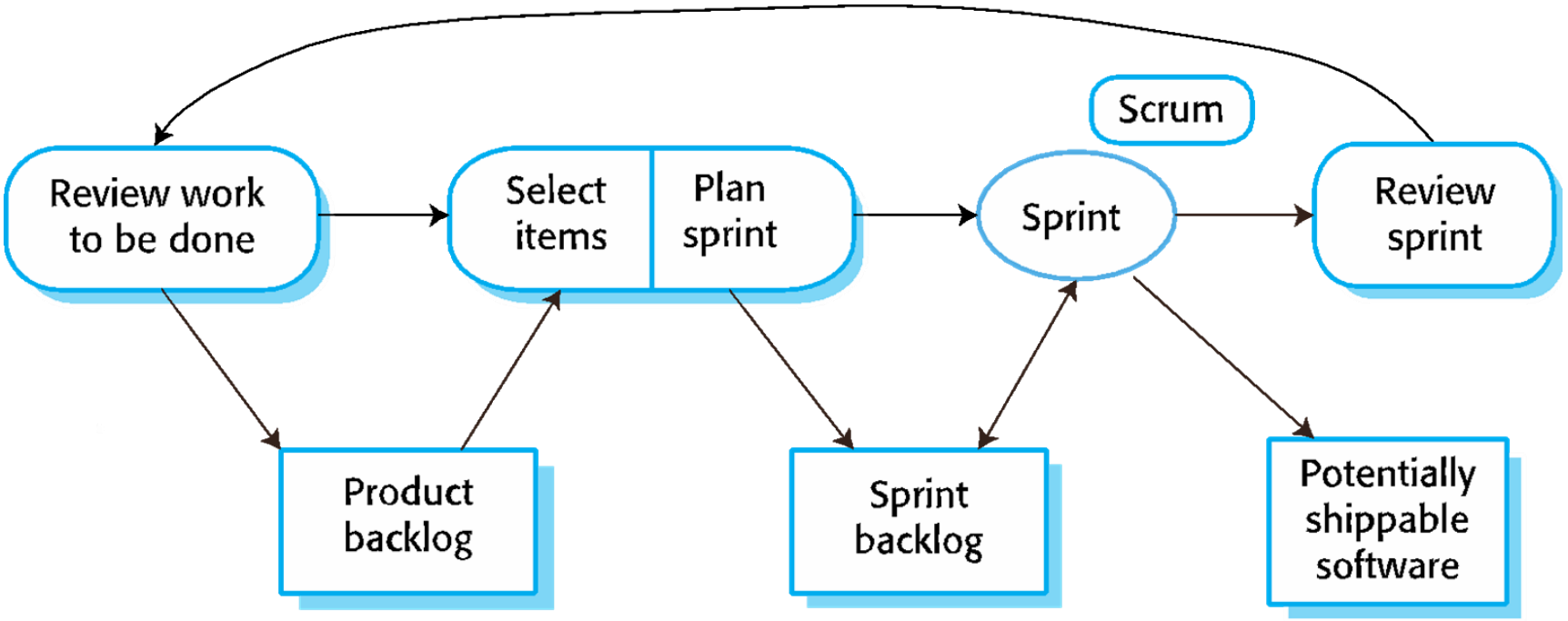
Scrum

Scrum term	Definition
Development team	A self-organizing group of software developers, which should be no more than 7 people. They are responsible for developing the software and other essential project documents.
Potentially shippable product increment	The software increment that is delivered from a sprint. The idea is that this should be ‘potentially shippable’ which means that it is in a finished state and no further work, such as testing, is needed to incorporate it into the final product. In practice, this is not always achievable.
Product backlog	This is a list of ‘to do’ items which the Scrum team must tackle. They may be feature definitions for the software, software requirements, user stories or descriptions of supplementary tasks that are needed, such as architecture definition or user documentation.
Product owner	An individual (or possibly a small group) whose job is to identify product features or requirements, prioritize these for development and continuously review the product backlog to ensure that the project continues to meet critical business needs. The Product Owner can be a customer but might also be a product manager in a software company or other stakeholder representative.

Scrum

Scrum term	Definition
Scrum	A daily meeting of the Scrum team that reviews progress and prioritizes work to be done that day. Ideally, this should be a short face-to-face meeting that includes the whole team.
ScrumMaster	The ScrumMaster is responsible for ensuring that the Scrum process is followed and guides the team in the effective use of Scrum. He or she is responsible for interfacing with the rest of the company and for ensuring that the Scrum team is not diverted by outside interference. The Scrum developers are adamant that the ScrumMaster should not be thought of as a project manager. Others, however, may not always find it easy to see the difference.
Sprint	A development iteration. Sprints are usually 2-4 weeks long.
Velocity	An estimate of how much product backlog effort that a team can cover in a single sprint. Understanding a team's velocity helps them estimate what can be covered in a sprint and provides a basis for measuring improving performance.

Scrum sprint cycle



- Sprints are fixed length, normally 2–4 weeks.
- 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 from the product backlog to be developed during the sprint.

The Sprint cycle

- 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 (Scrums) 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.

Practical problems with agile methods

- The informality of agile development is incompatible with the legal approach to contract definition that is commonly used in large companies.
- Agile methods are most appropriate for new software development rather than software maintenance. Yet the majority of software costs in large companies come from maintaining their existing software systems.
- Agile methods are designed for small co-located teams yet much software development now involves worldwide distributed teams.

Contractual issues

- Most software contracts for custom systems are based around a specification, which sets out what has to be implemented by the system developer for the system customer.
- However, this precludes interleaving specification and development as is the norm in agile development.
- A contract that pays for developer time rather than functionality is required.
  - However, this is seen as a high risk in many legal departments because what has to be delivered cannot be guaranteed.

Agile methods and software maintenance

- Most organizations spend more on maintaining existing software than they do on new software development. Agile methods have to support maintenance, to be successful.
- 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 arise if original development team is not retained.
- Key problems are:
  - Lack of product documentation
  - Keeping customers involved in the development process
  - Maintaining the continuity of the development team
- Agile development relies on the development team knowing and understanding what has to be done.
- For long-lifetime systems, this is a real problem as the original developers will not always work on the system.

Agile and plan-driven methods

- 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.
  - 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.

System issues

- How large is the system being developed?
  - Agile methods are most effective a relatively small co-located team who can communicate informally.
- What type of system is being developed?
  - Systems that require a lot of analysis before implementation need a fairly detailed design to carry out this analysis.
- What is the expected system lifetime?
  - Long-lifetime systems require documentation to communicate the intentions of the system developers to the support team.
- Is the system subject to external regulation?
  - If a system is regulated you will probably be required to produce detailed documentation as part of the system safety case.

People and teams

- 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.
- How is the development team organized?
  - Design documents may be required if the team is dsitributed.

Problems of Agile methods in large systems

- 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.

Problems of Agile methods in large systems

- 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.