# Agile Software Development

# A bit of history

- Standish Group CHAOS* study:
  - IT projects **fail** to fulfill **schedule** and **cost** forecasts, and often fail to deliver the **benefits predicted**.
  - These issues have been confirmed by various organizations, including the Department of Defense (DoD).
- The DoD noted that, of the $35.7 billion spent by the organization in 1995 for software, only **2 percent** of the software was usable as delivered. The DoD found that **75 percent** of the software developed was either **never used** or was cancelled prior to delivery.

* http://www.standishgroup.com

# Requirements Uncertainty

Watts Humphrey -- IBM researcher introduced **Requirements Uncertainty Principle**

*“for a new software system, the requirements will not be completely known until after the users have used it.”*

# Software Engineering Uncertainty

- Hadar Ziv of the University of California introduced **Uncertainty Principle in Software Engineering**,

*"Uncertainty is **inherent** and **inevitable** in software development **processes** and **products**."*

# So?

**If it is impractical**

- for users can’t foretell what they’ll want until they see it
- to predicting and planning substantial IT projects is not possible
- to protecting projects against changes that arise during the development process

**the ideas behind existing “waterfall” methods are flawed**

**So, an incremental, prototype-based methodology could offer substantial benefits.**

# Rapid Software Development

- Interleaved
  - Specification
  - Design
  - Implementation
- System is developed as a series of versions
- Stakeholders involved in version evaluation
- Tools
  - User interfaces are often developed using an IDE and graphical toolset.

# Agile Methodology

- Focus on the code rather than the design
- Iterative software development
- Intended to deliver **working software** quickly
- Meet **changing** requirements.
- Reduce overheads in the software process
  - limit documentation
- respond to change
  - quickly
  - efficiently

# 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*

# Principles of Agile Development

- Customer involvement
  - Provide and proritize requirements
  - Evaluate
- Incremental delivery
- People not process
  - Recognize team skills
- Embrace change
- Maintain simplicity
  - Software design and development

# When is Agile Applicable?

- Development of a small or medium-sized product
- Custom system development
- Clear commitment from the customer to become involved in the development process
- Few external rules & regulations that affect software.
- Scaling problem:
  - Small tightly-integrated teams

# Challenges with Agile

- Maintain customer interest in the process.
- Team members may be unsuitable
- Prioritising changes can be difficult where there are multiple stakeholders.
- Maintaining simplicity requires extra work.
- Contracts may be a problem

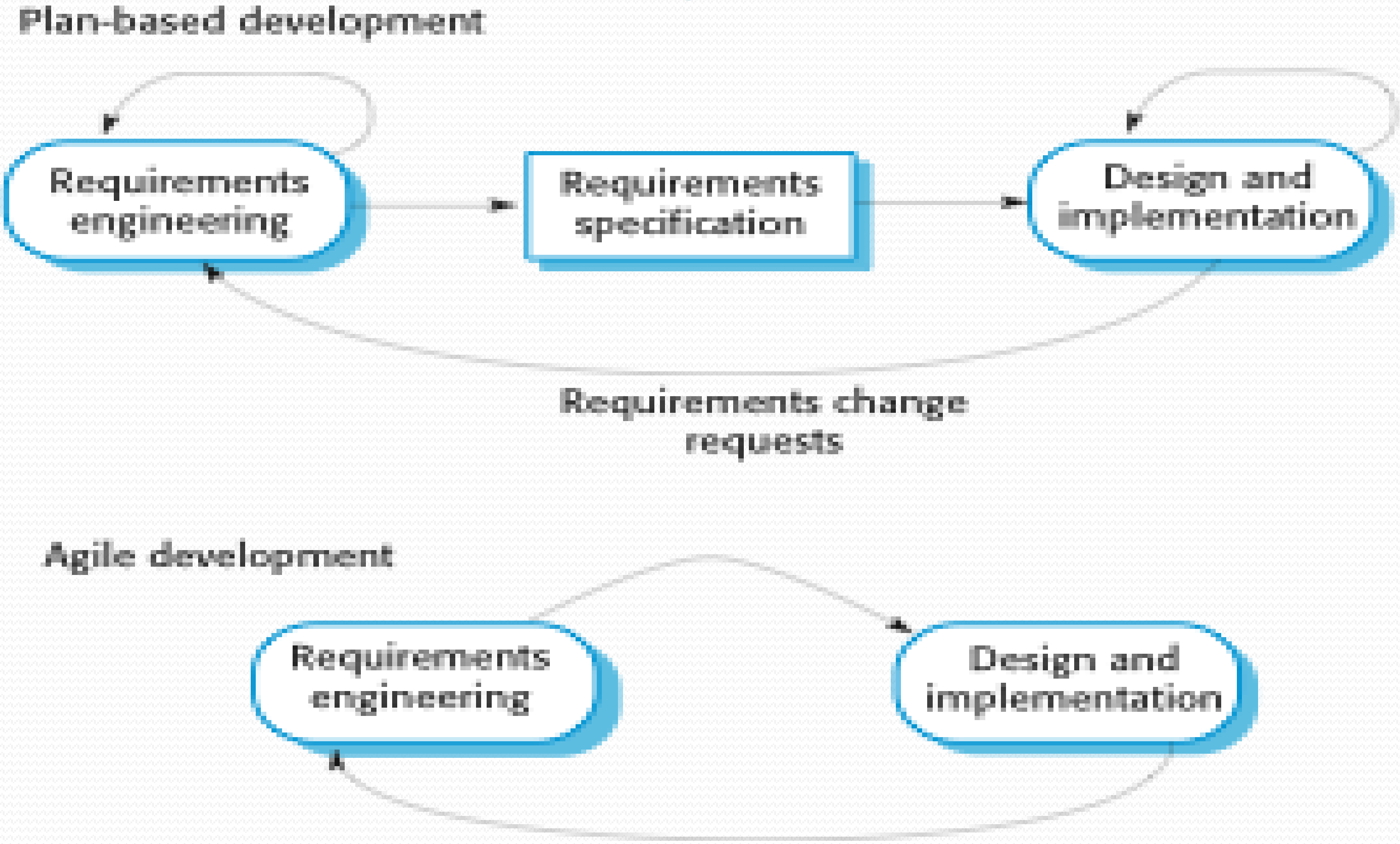
# Maintenance

- Most organizations spend more on **maintaining** existing software than they do on new software development.
- Agile must support maintenance as well as original development.

# Plan driven development

- Plan-driven development
  - based around separate development stages
  - outputs to be produced at each of these stages
  - planned in advance
- Agile development
  - Specification, design, implementation and testing are inter-leaved
  - 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
Requirements engineering
Requirements specification
Design and implementation
Requirements change requests
Agile development
Requirements engineering
Design and implementation

# Technical, human, organizational issues

- Balance between plan-driven and agile processes
- plan-driven approach if:
  - Deciding on method based on:
  - Is it important to have a **very detailed specification** and design **before** moving to implementation?
  - i.e. Government contracts
- Agile if:
  - Incremental delivery strategy, where you deliver the software to customers and get rapid feedback
  - Small co-located team who can communicate informally
  - i.e. automation for small companies

# Technical, human, organizational issues

- Type of system
  - 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).
- 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

- Cultural or organizational issues
  - Traditional engineering organizations have a culture of plan-based development
- Designers and programmers in the development team
  - Agile methods require higher skill levels than plan-based approaches in which programmers 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.

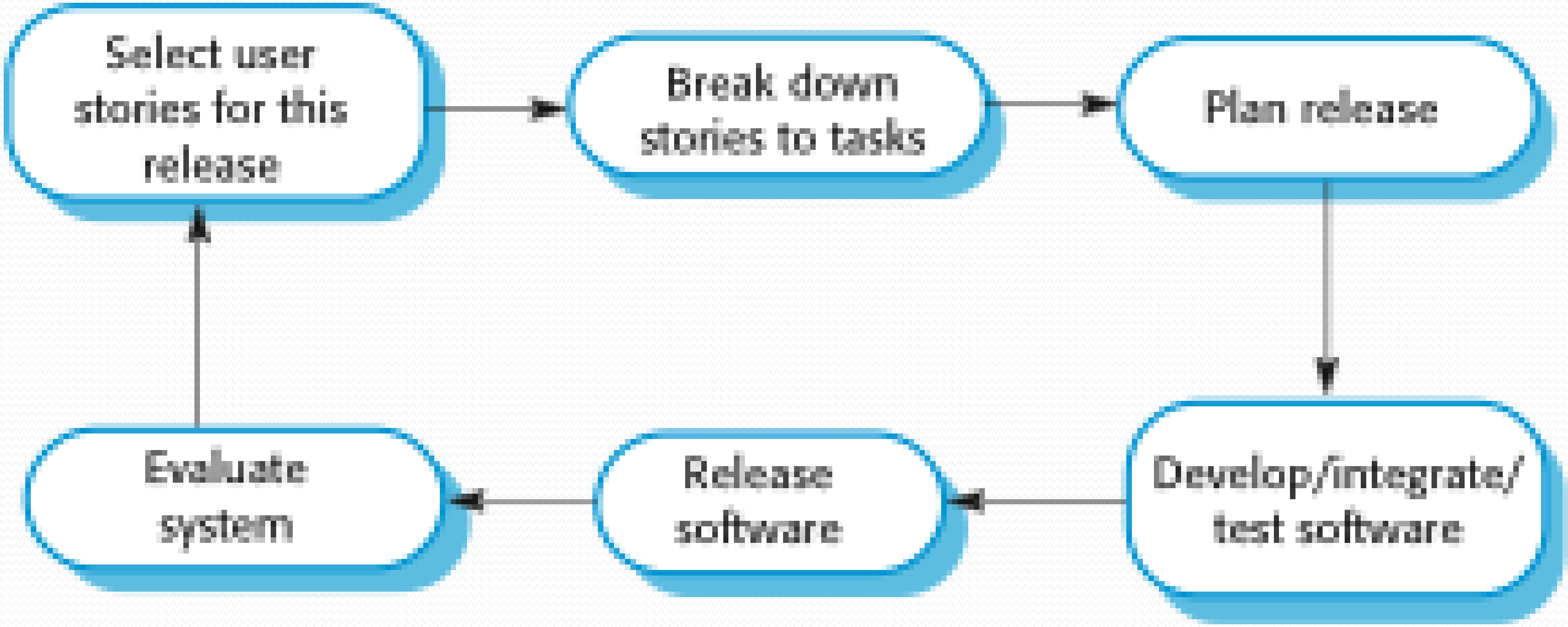
# Extreme programming

- Best-known and 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**
  - Build is accepted only 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 in team
  - pair programming
  - collective ownership
  - process that avoids long working hours.
- Change supported through regular system releases.
- Maintaining simplicity through constant refactoring of code.

# The XP release cycle


Select user stories for this release
Break down stories to tasks
Plan release
Develop/integrate/ test software
Release software
Evaluate system

# Extreme programming practices (a)

| Principle or practice | Description |
|---|---|
| **Incremental planning** | Requirements on story cards<br>Stories to be included in a release are determined by the **time available** and their **relative priority**.<br>The developers break these stories into development 'Tasks'. |
| **Small releases** | The **minimal useful set of functionality** that provides business value is **developed first**.<br>Frequent releases that **incrementally** add functionality to the first release. |
| **Simple design** | **Only enough** design to meet the current requirements |
| **Test-first development** | An **automated unit test** framework is used to write tests **before** a functionality is **implemented**. |
| **Refactoring** | Code is continuously refactored asap.<br>This keeps the code simple and maintainable. |

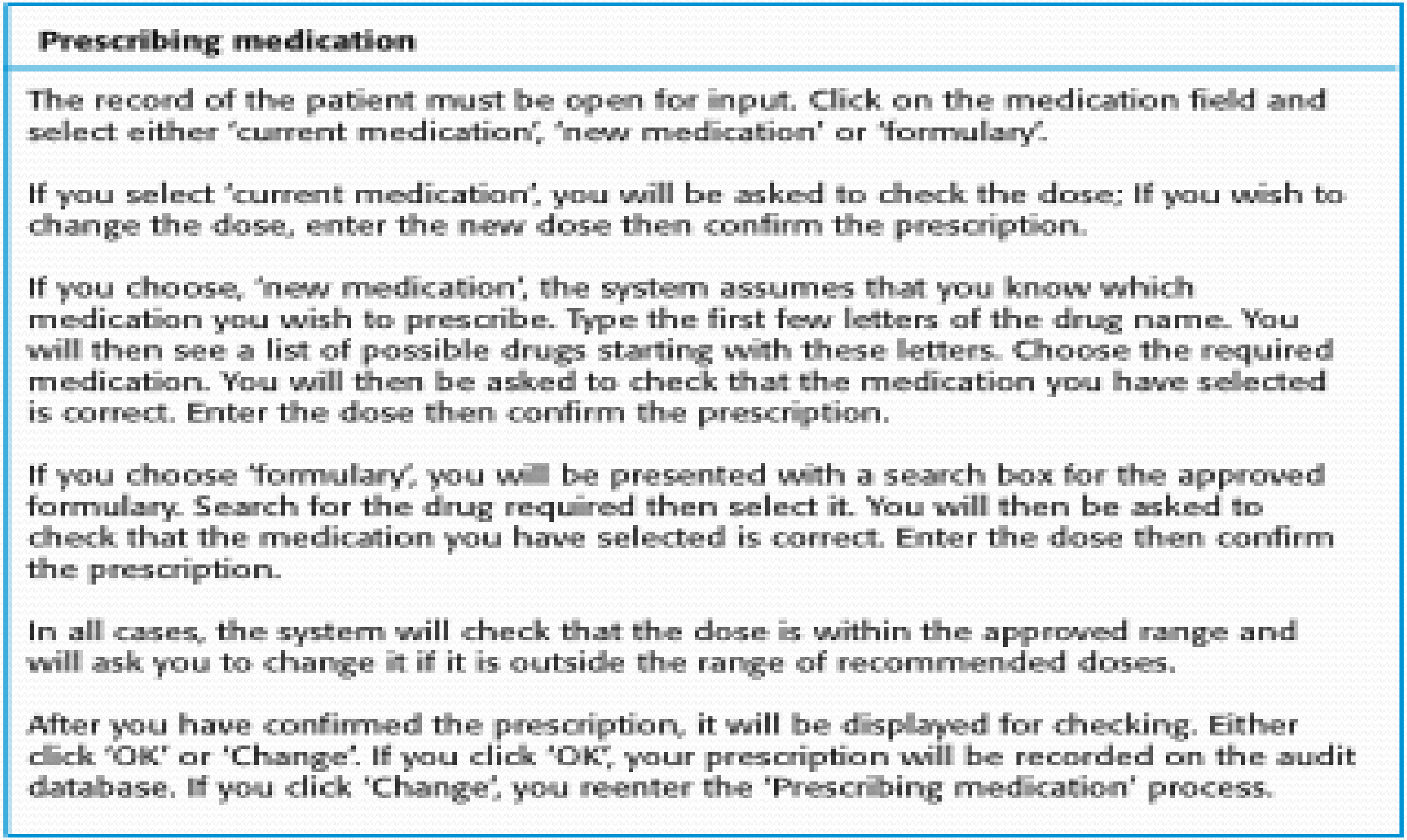
# Extreme programming practices (b)

| | |
|---|---|
| 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<br>All the developers take responsibility for all of the code.<br>Anyone can change anything. |
| Continuous integration | As soon as the work on a task is complete, it is integrated into the whole system.<br>After each integration **all the unit tests must pass**. |
| Sustainable pace | Overtime not desired since it tends to reduce **code quality** and medium term **productivity** |
| On-site customer | End-user (the customer) should be available full time for the use of the XP team.<br>Customer is part the development team<br>Responsible for bringing system requirements to the team |

# XP Requirements **scenarios**

- In XP, a customer or user is part of the XP team
  - responsible for making decisions on requirements.
- User requirements are expressed as **scenarios** or **user stories.**
  - written on **cards**
  - team breaks them down into implementation **tasks**.
- 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.

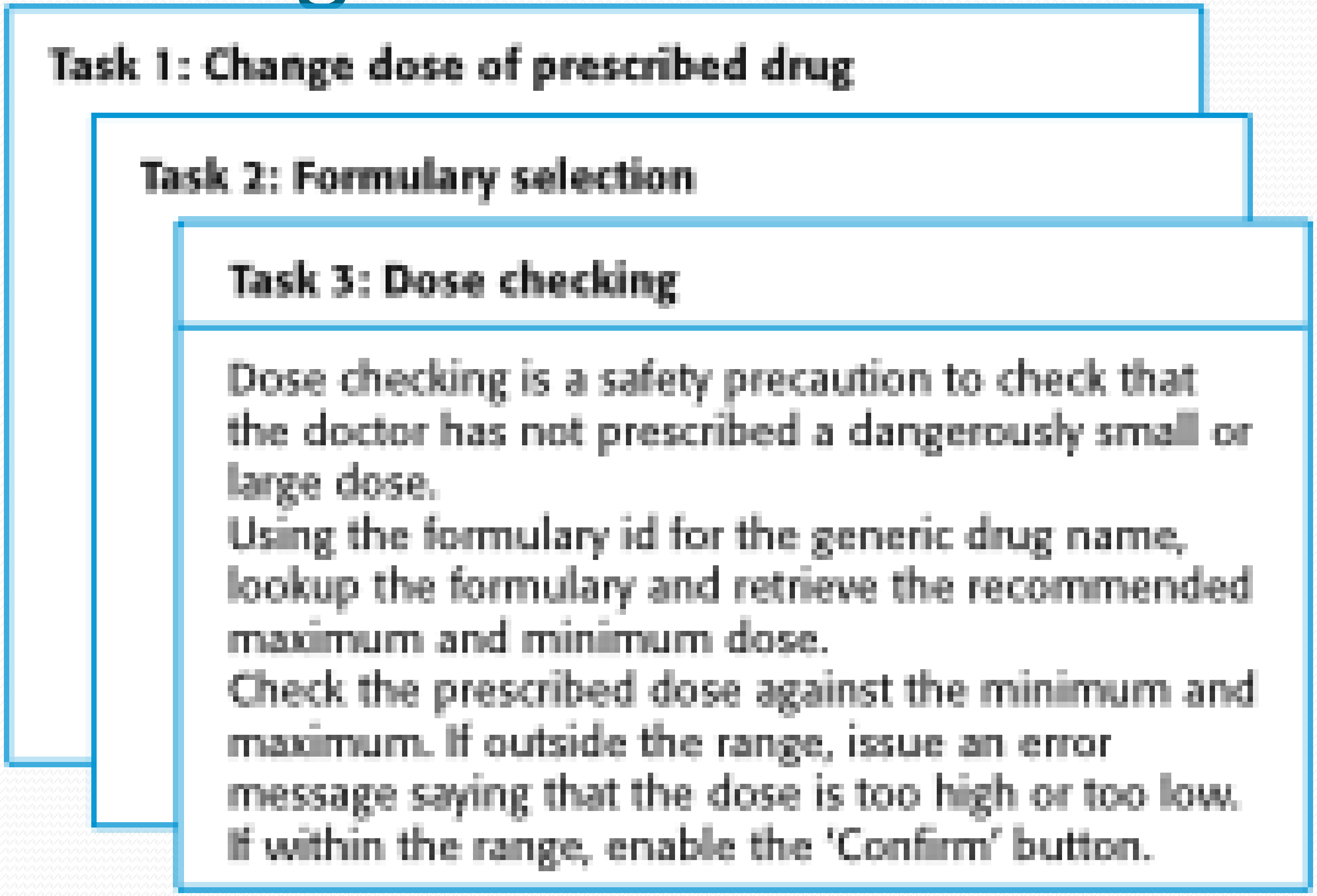
If you choose, ‘new medication’, the system assumes that you know which medication you wish to prescribe. Type the first few letters of the drug name. You will then see a list of possible drugs starting with these letters. Choose the required medication. You will then be asked to check that the medication 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.

# 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

- Testing is central to XP
- The program is **tested** after **every change**
- XP testing features:
  - Test-first development.
  - Incremental test development from scenarios.
  - User involvement in test development and validation.
  - Automated test are used
    - run all component tests
    - each time that a new release is built

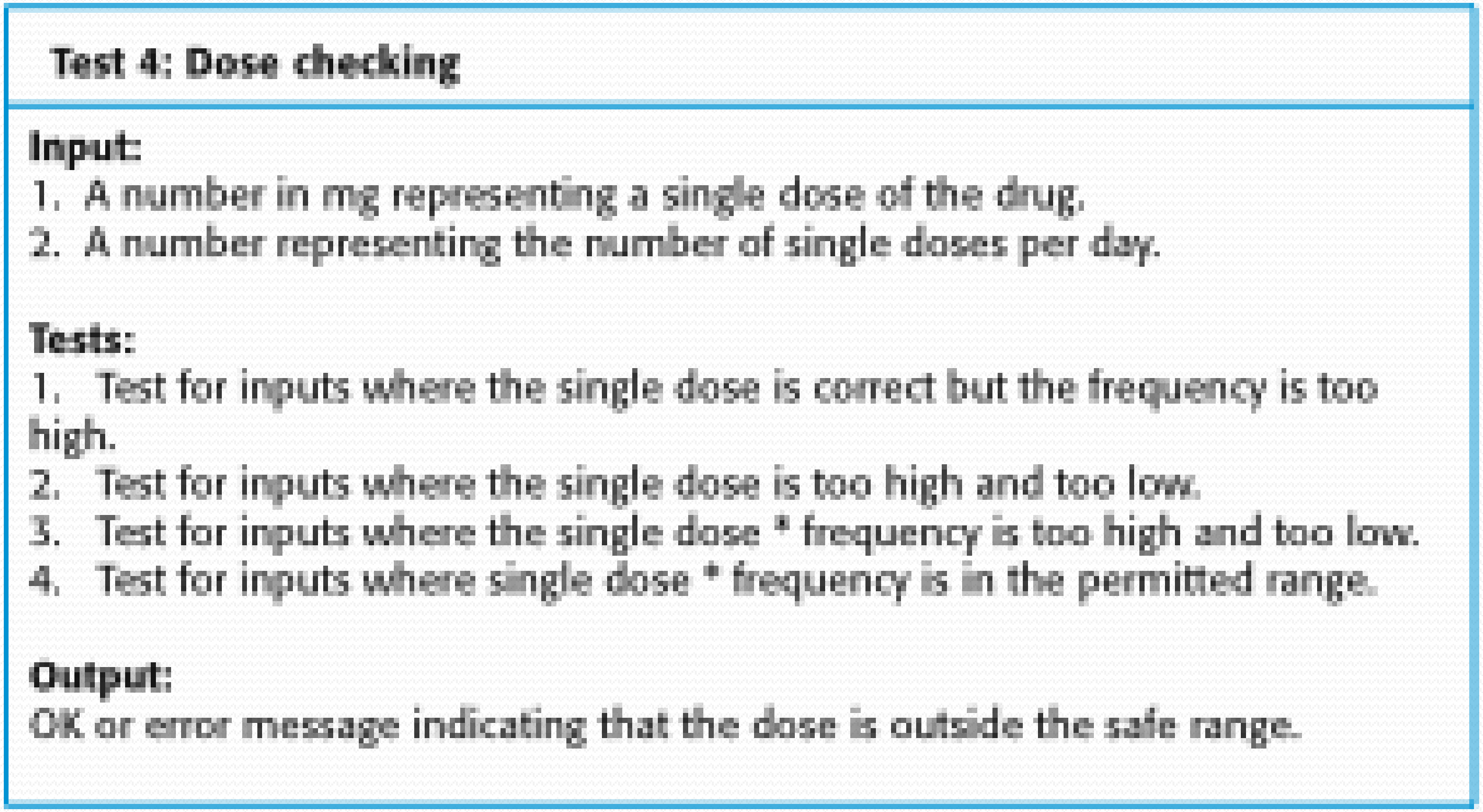
# Test-first development

- 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.
- Tesst include 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

- In the **testing process** is to help develop **acceptance tests** for the stories to be implemented in the next release .
- 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.
- In practice may be reluctant to get involved in the testing process – time constraints.

# 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 and** must be
  - stand-alone
  - simulate the submission of input to be tested
  - 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 may not like to write tests
- 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.

# 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 -- whole team benefits
- Measurements suggest that **development productivity** with pair programming is similar to that of two people working independently.

# 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.
- Knowledge sharing during pair programming is very important – reduces risks to a project
- 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

- Collective **ownership** and **responsibility** for the system.
  - Team has collective responsibility for resolving problems.
- 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.
  - Everyone benefit immediately from refactoring so they are likely to support the process.

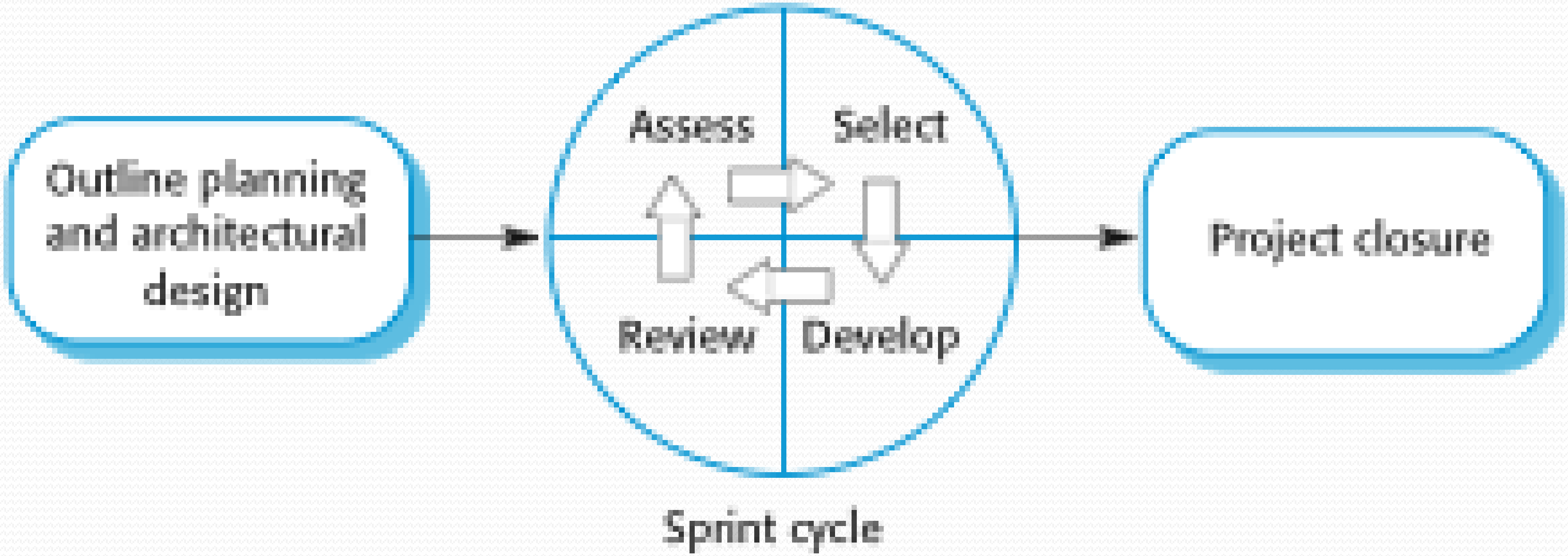
# Agile project management

- Reminder: The principal responsibility of software project managers is to manage the project so that the **correct software** is delivered **on time** and **within the planned budget** for the project.
- 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 an agile method focuses on **managing** iterative development rather than specific agile practices.
- Three phases
  - **Initial phase** is an **outline planning** phase establish
    - general objectives for the project
    - design the software architecture.
  - Series of **sprint cycles** -- each cycle develops an increment of the system.
  - **Project closure**
    - required documentation i.e. user help frames, user manuals
    - assesses **the lessons learned** from the project.

# The Scrum process



# The Sprint cycle

- Sprints are fixed length, normally 2–4 weeks.
- Corresponds to a release of the system in XP.
- The starting point for planning is the **product backlog** -- 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.

# The Sprint cycle

- The team organizes themselves to develop the software.
- At 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**.
- Then 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
  - communicates with customers and management outside of the team.

# Communication

- The **whole team** attends **short daily meetings**
- All team members
  - share information
  - describe their progress since the last meeting
  - problems that have arisen
  - what is planned for the following day.
- Everyone on the team knows what is going on
- If problems arise, team can re-plan short-term work to cope with them.

# Scrum benefits

- Product is broken down into a set of chunks that are
    - Understandable
    - Manageable
- The whole team has 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 proven to be successful for
  - small and medium sized projects
  - 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 (sub)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.