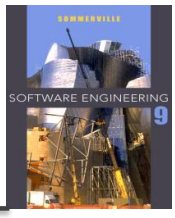
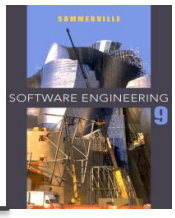


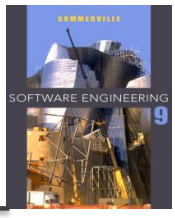

# Chapter 3 – Agile Software Development

# Topics covered

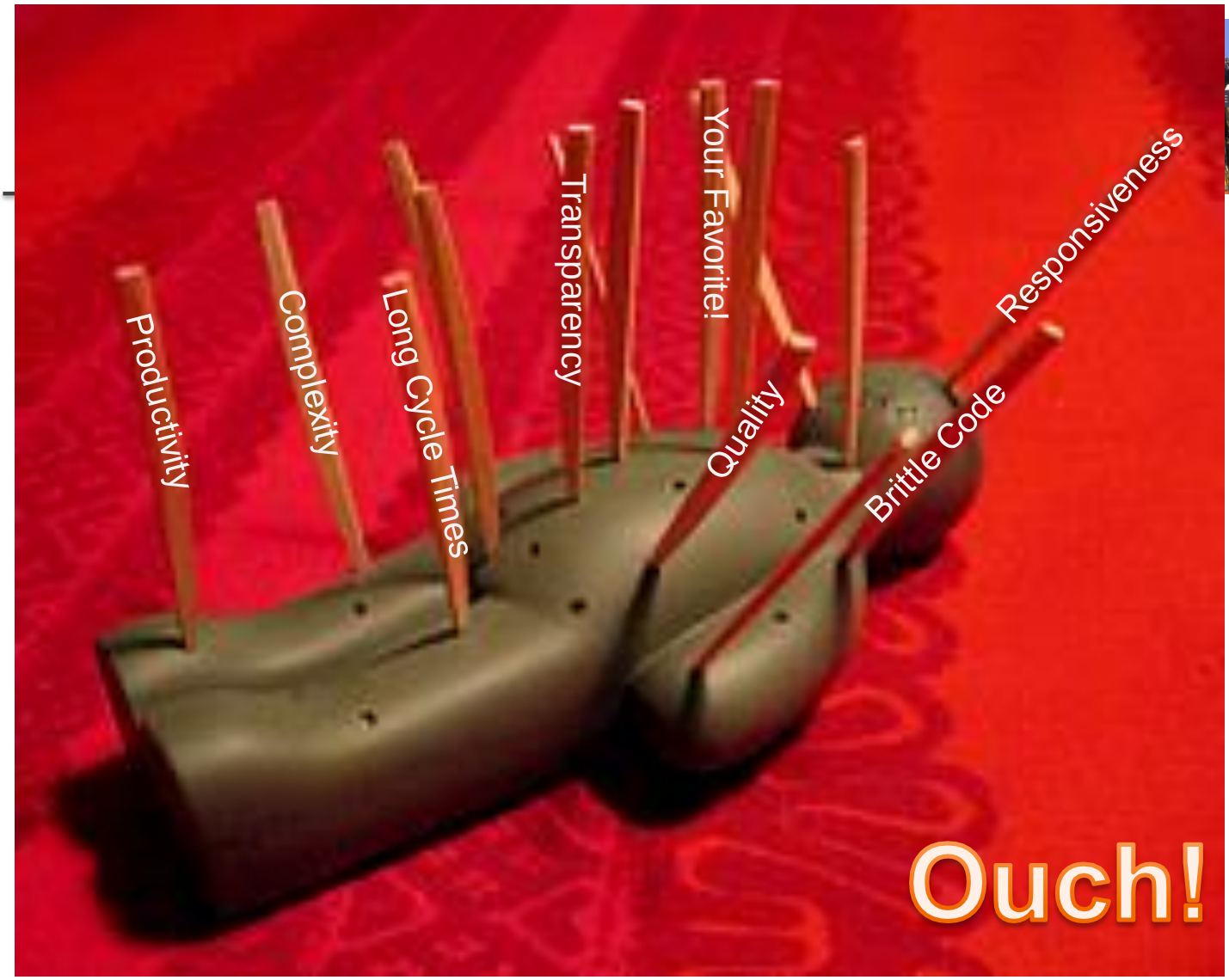



- ✧ Agile methods
- ✧ Plan-driven and agile development
- ✧ Extreme programming
- ✧ Agile project management
- ✧ Scaling agile methods

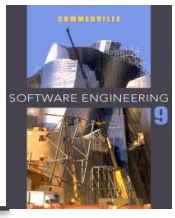
# Rapid software development




- Why?
  - Need to react to changes more quickly than 2 year long waterfall projects
  - 2 years and then you got the design wrong anyway! Small deliveries aren't abstract
- How?
  - Goal - Deliver working software quickly
    - Compromise - less functionality in a delivery, not lower quality
    - Less documentation
  - Focus on the code rather than the design
  - Interleave
    - Specification, design and implementation are inter-leaved
  - Deliver small versions and get user (stakeholder) input

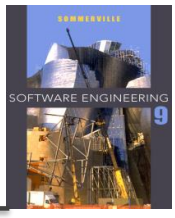


# Agile manifesto




- *Our values:*
  - *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.*

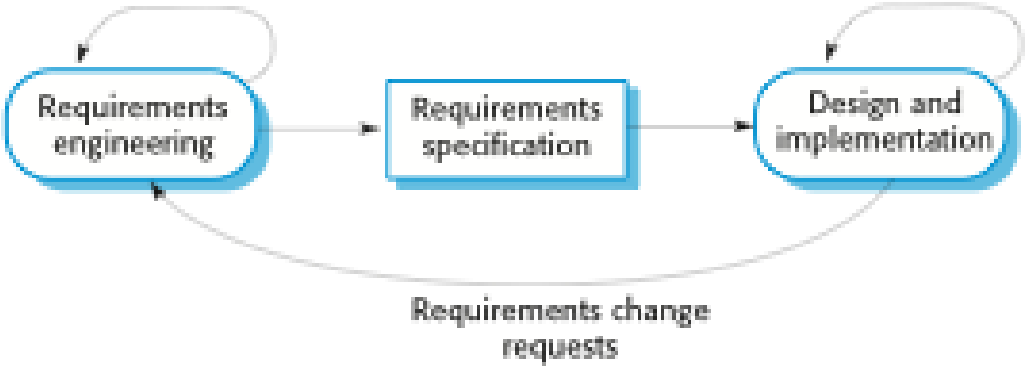
# Plan-driven and agile specification




separate development stages with the outputs to be produced at each of these stages planned in advance.

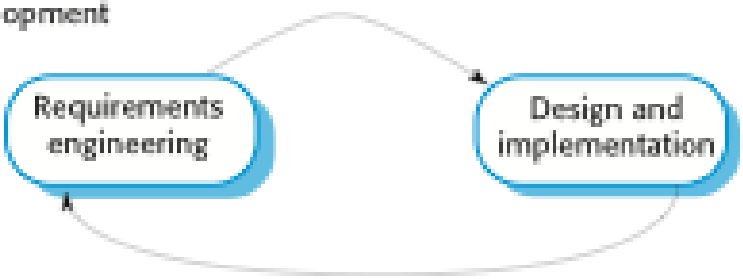
Plan-based development

Iteration within stage



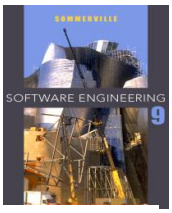

Not necessarily waterfall model – plan-driven, incremental development is possible

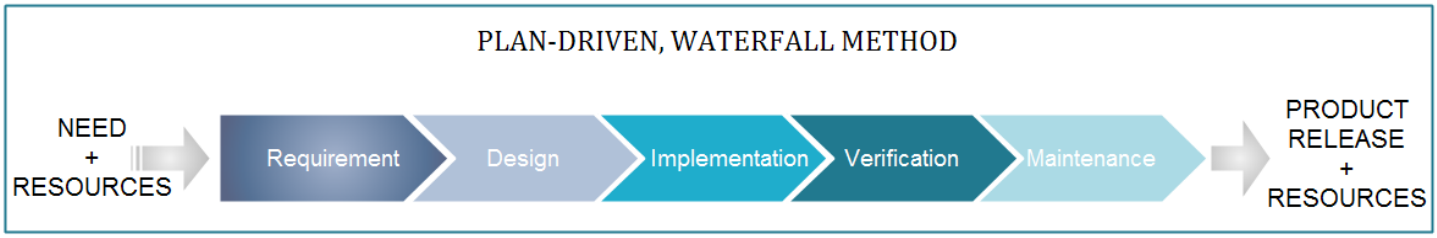
Agile development

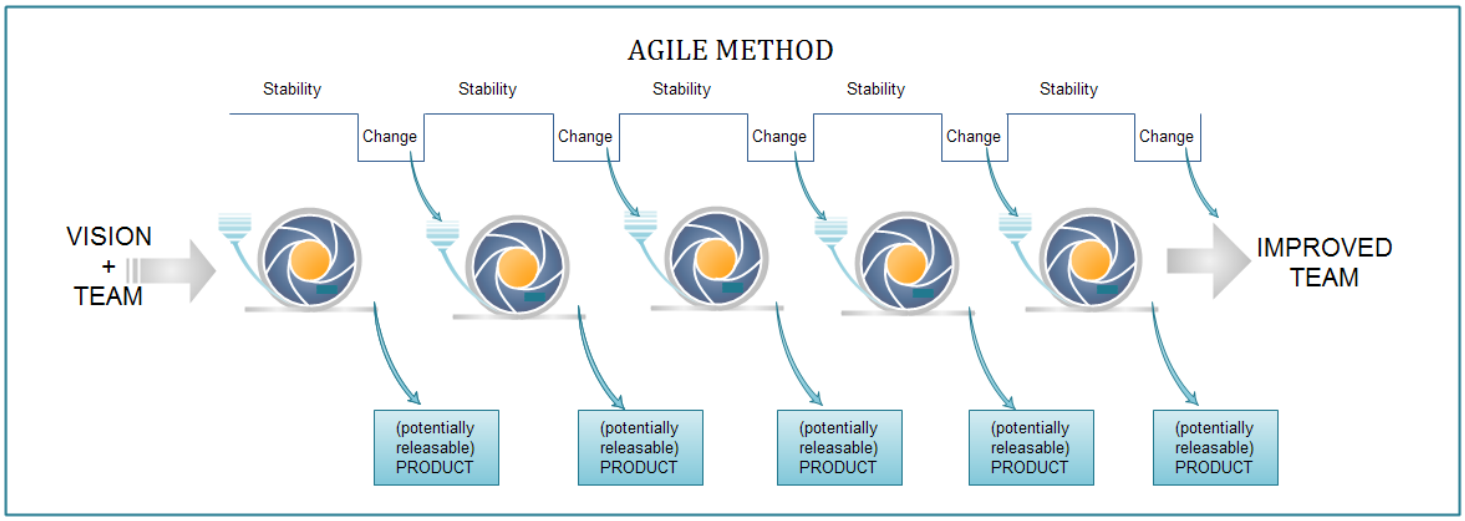



Iteration of stage

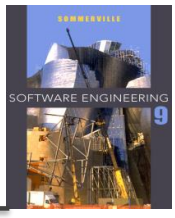
User's full agreement at end, not before code



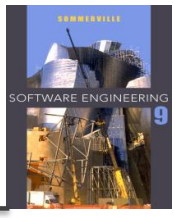



# Problems with agile methods



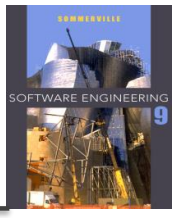

- It can be difficult to keep the interest of customers / users who are involved in the process.
- Team members may be unsuited to the intense involvement that characterizes agile methods.
- Prioritizing changes can be difficult where there are multiple stakeholders.
- Maintaining simplicity requires extra work.
- Contracts may be a problem as with other approaches to iterative development.
- Because of their focus on small, tightly-integrated teams, there are problems in scaling agile methods to large systems.
- Less emphasis on documentation - harder to maintain when you get a new team for maintenance

# Balance plan driven and agile



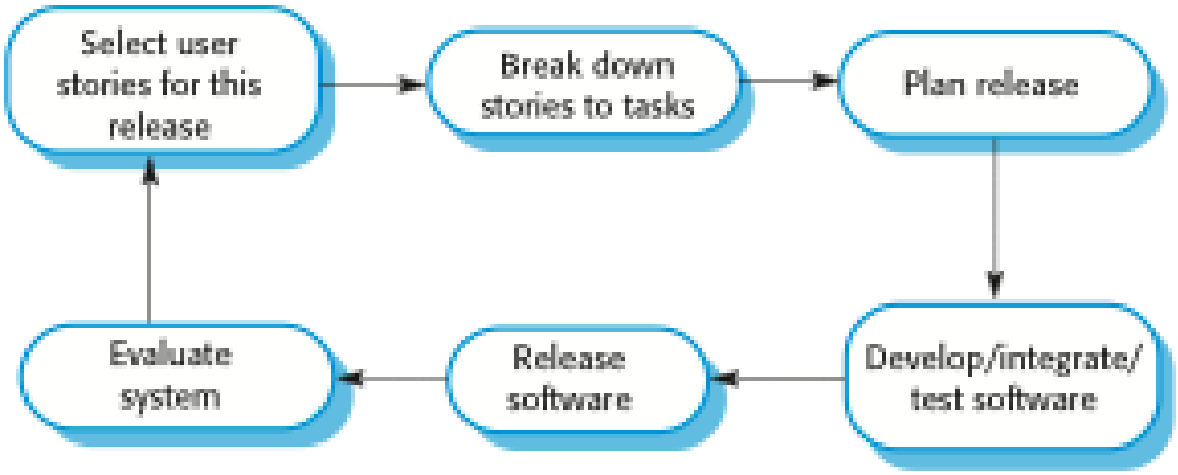

- Not great for Agile:
  - What **type of system** is being developed?
    - Plan-driven approaches may be required for systems that require **a lot of analysis before implementation** (e.g. real-time system with complex timing requirements).
  - What is the expected **system lifetime**?
    - Long-lifetime 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?
    - **Many teams; Outsourcing** ---> need design documents to control borders
  - Culture or contract needs **detailed specification**
  - Is **rapid feedback from users realistic**?
  - **Large scale, not co-located** may require more formal communication methods
  - Need high level programming skills - refactoring, work with little spec
  - Outside regulation documentation requirements

# Extreme programming

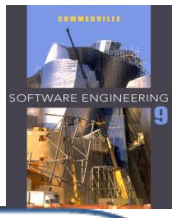


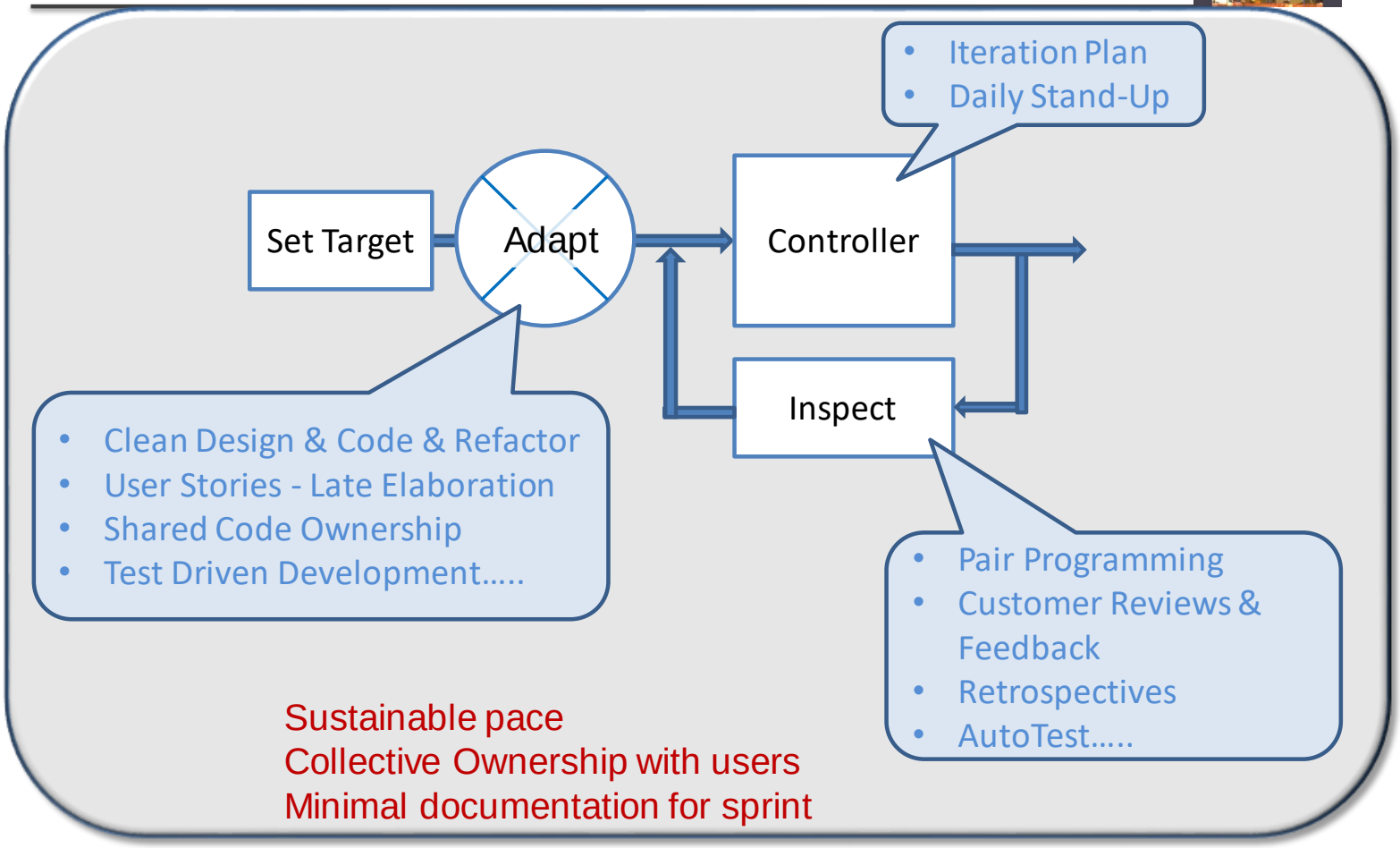

- A popular form of Agile
- Extreme Programming (XP) takes an ‘extreme’ approach to iterative development.
  - New versions may be built several times per day;
  - Increments are delivered to customers every 2 weeks;
  - All tests must be run for every build and the build is only accepted if tests run successfully.
- Customer involvement means full-time customer engagement with the team. - Specifications through user stories broken into tasks
- People not process : pair programming, collective ownership and a process that avoids long working hours.
- Regular system releases. - release set of user stories
- Maintaining simplicity through constant refactoring of code.

# The extreme programming release cycle

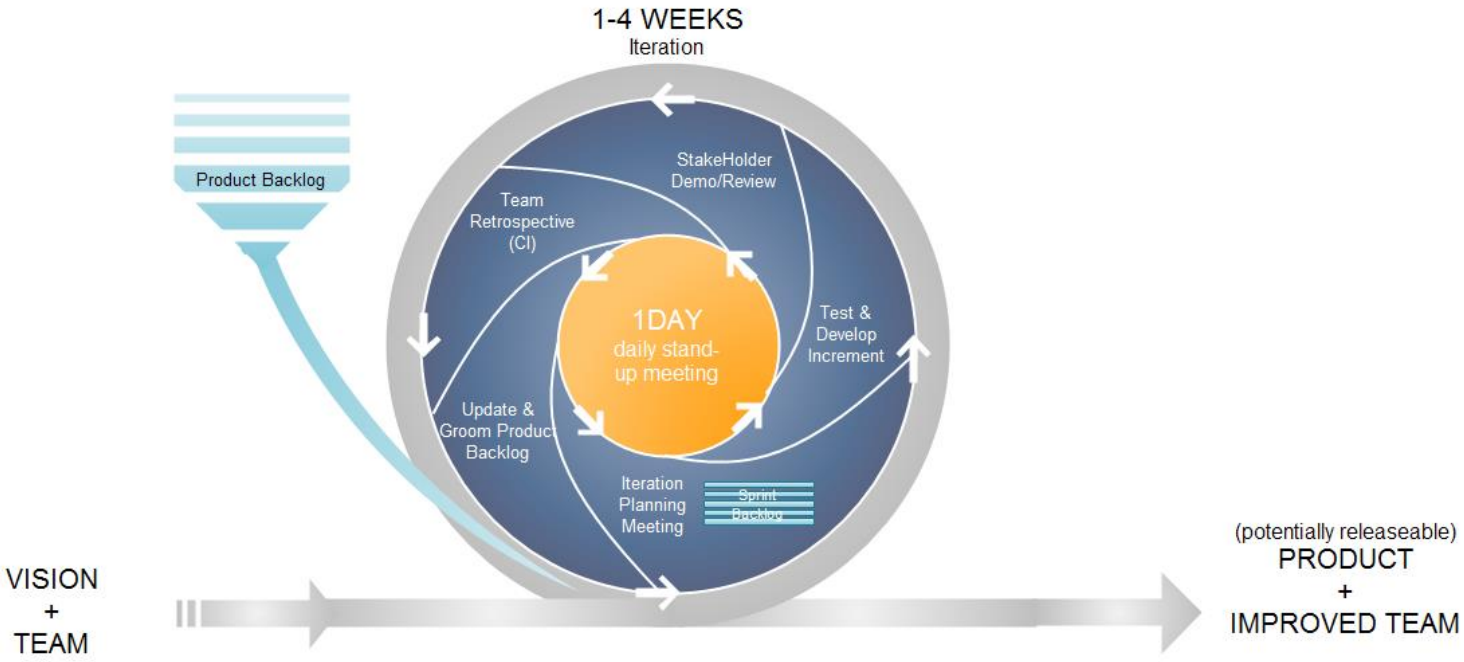



# The Parts and the Whole

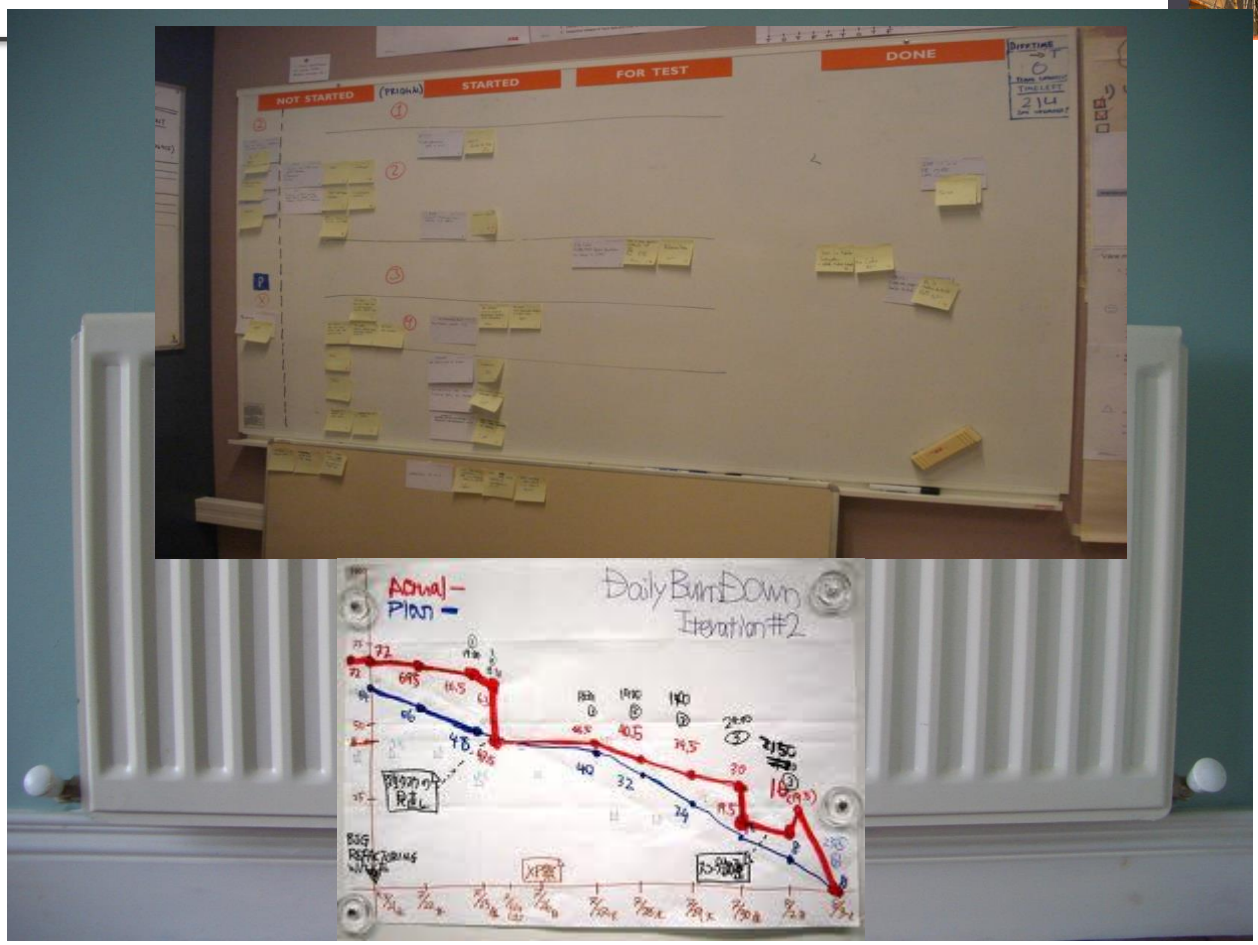


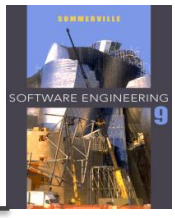
# The Life of an Iteration



# Transparency



# 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.

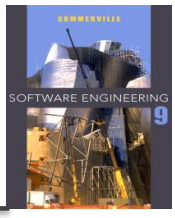
If you choose, ‘new medication’, the system assumes that you know which medication you wish to prescribe. Type the first few letters of the drug name. You will then see a list of possible drugs starting with these letters. Choose the required medication. You will then be asked to check that the medication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



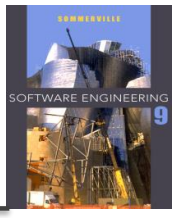

**Task 1: Change dose of prescribed drug**

**Task 2: Formulary selection**

**Task 3: Dose checking**

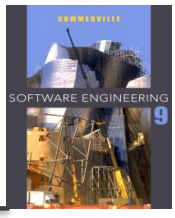
Dose checking is a safety precaution to check that the doctor has not prescribed a dangerously small or large dose.
Using the formulary id for the generic drug name, lookup the formulary and retrieve the recommended maximum and minimum dose.
Check the prescribed dose against the minimum and maximum. 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.

# Refactoring



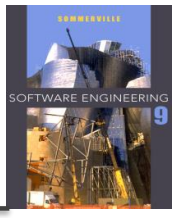

- Programming team look for possible software improvements and make these improvements even where there is no immediate need for them.
- This improves the understandability of the software and so reduces the need for documentation.
- Changes are easier to make because the code is well-structured and clear.
- However, some changes requires architecture refactoring and this is much more expensive.
- RISK:
  - Changes the user does not test
  - Changes to working software break it

## 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 case description for dose checking



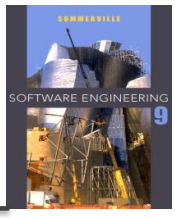

**Test 4: Dose checking**

**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.

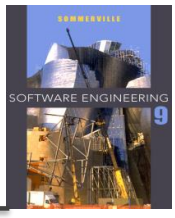
**Output:**
OK or error message indicating that the dose is outside the safe range.

# Test automation



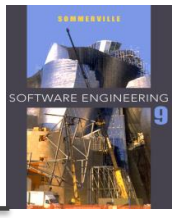

- Automate tests (junit)
- Run upon checkin
- Difficulties:
  - Time constraints
  - Programmer preferences to not test
  - Test coverage

## Pair programming



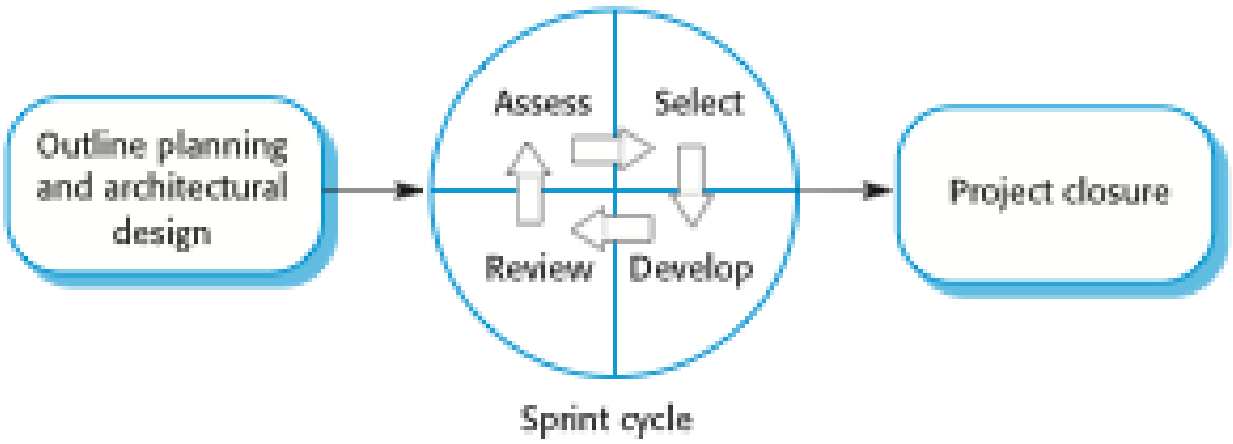

- In XP, programmers work in pairs, sitting together to develop code.
- Common ownership
- Knowledge spread
- Informal review
- Refactoring
- Similar output to two people coding

## Scrum

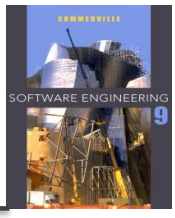



- Project Manager's job: - Deliver needed system on time within budget
- The Scrum approach - manage the iterations
- There are three phases in Scrum.
  - outline planning phase - general picture and architecture
  - Sprint cycles releasing increments of the system.
  - The project closure phase - final delivery, documentation and review of lessons learned.

# The Scrum process

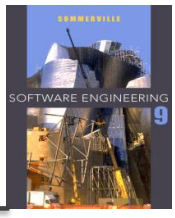


# The Sprint cycle



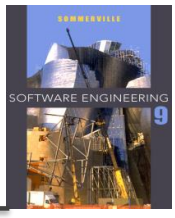

- Every 2–4 weeks (a fixed length).
- 1) Project team with customer: Look at product backlog - select stories to implement
- 2) implement with all customer communication through scrum master (protecting pgmr at this point)
  - Scrum master has project manager role during sprint
  - Daily 15 min meetings
    - Stand up often
    - Team presents progress and impediments
    - Scrum master tasked with removing impediments
- 3) Review system release with user

## 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.

# Summary




- Plan Driven (Ex: Waterfall) vs Incremental (Ex: Agile )
  - Structure and benefits and downfalls
- XP - an implementation of Agile - Power to the Programmer
  - User story requirements
  - Test driven design with continual retest and integration
  - Pair Programming
  - Refactoring encouraged
- Scrum - project management of Agile using sprints
  - Iterations of full team contact / Scrum master protection of programmers / full team release review