FRESNO STATE
Discovery. Diversity. Distinction.

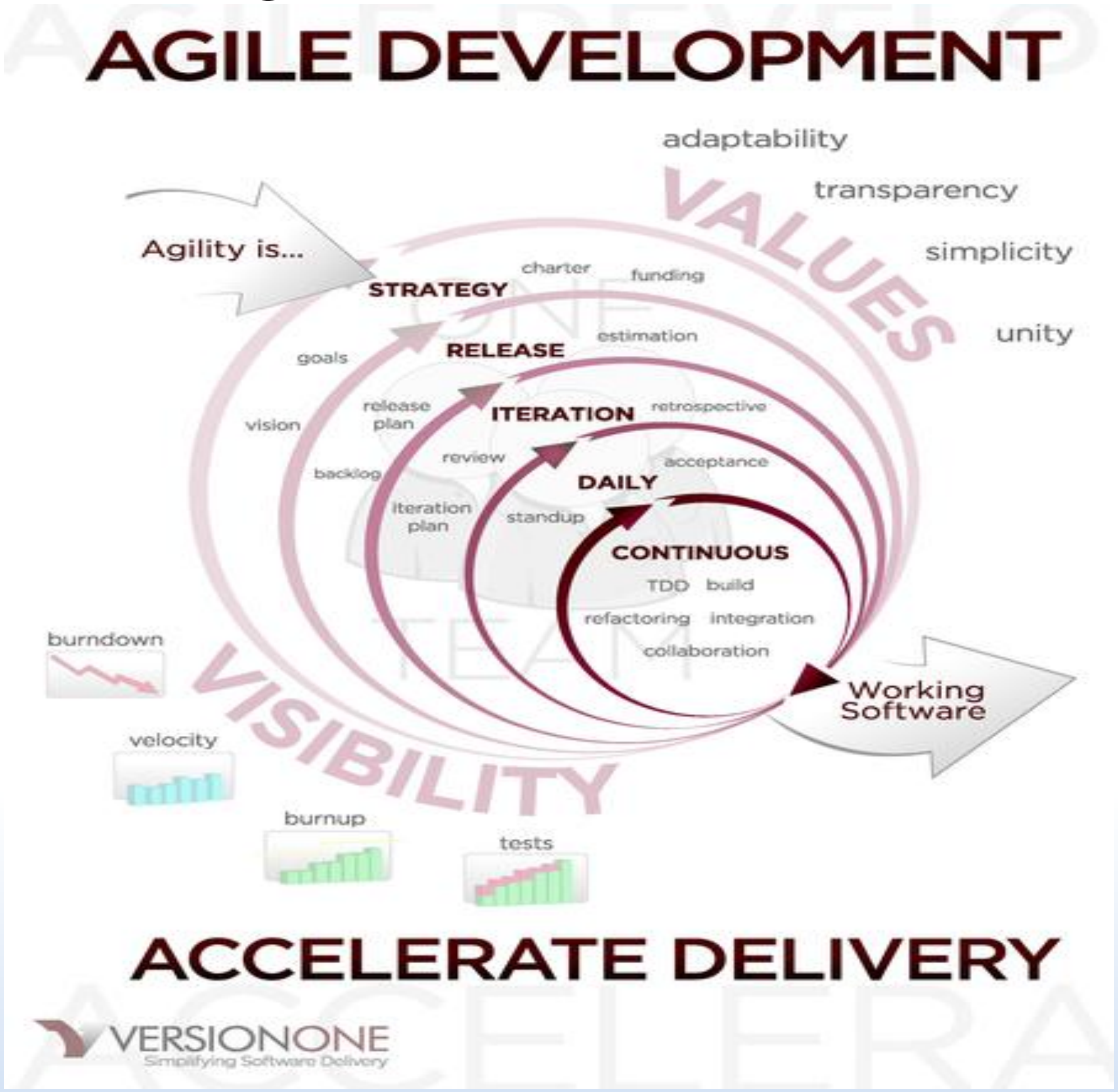
CSCI 150
Intro to Software Engineering

Oct. 2, 2018

Shih-Hsi "Alex" Liu

Chapter 3 – Agile Software Development

Lecture 1



California State University, Fresno – Department of Computer Science

Topics covered

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

Rapid software development

- **Rapid** development and delivery is now often the most important requirement for software systems
  - Businesses operate in a fast–changing requirement and it is practically impossible to produce a set of stable software requirements
  - Software has to evolve quickly to reflect changing business needs.
    - Otherwise, SW may be outdated when it is delivered.

Rapid software development

- Rapid software development characteristics:
  - Specification, design and implementation are inter-leaved. No detailed system specifications, and design documentation is minimized. User requirements document only defines the most important characteristics
  - System is developed as a series of versions with stakeholders involved in version specification and 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 and 1990s led to the creation of agile methods. Overheads are:
  - ✧ Plan-driven
  - ✧ Teams work geographically dispersed and worked on the SW for long periods of time
  - ✧ Significant overhead in planning, designing, documenting
  - ✧ Good for multiple development teams, for critical systems, different people involve in maintenance.
- ✧ 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 – targeting on smaller projects

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; and*
  - *Responding to change over following a plan*

Agile Methods (features)

- Incremental development methods
  - Small increments, which are released every 2-3 weeks
- Involve **customers** in the development process to **get rapid feedback** on changing requirements
- Minimize documentation by using informal communications rather than *formal* meetings with written *documents*.

Well known models of agile methods

- eXtreme Programming
- Scrum
- Crystal
- Adaptive Software Development
- DSDM (Dynamic Systems Development Method)
- Feature Driven Development
- Agile Modeling
- Agile instantiation of RUP

The principles of agile methods (shared by the models of previous slide)

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.
- **Custom** system development within an organization, where there is a clear commitment from the customer to become involved in the development process and where there are not a lot of external rules and regulations that affect the software.
- Because of their focus on small, tightly-integrated teams, there are **problems in scaling** agile methods to large systems.

Problems with agile methods

- It can be difficult to keep the interest of customers who are involved in the process.
  - It is not client's primary focus!
- Team members may be unsuited to the intense involvement that characterizes agile methods.
  - Some people may not like stressful environment
- Prioritizing changes can be difficult where there are multiple *stakeholders*. Different stakeholders get different priorities.
- Maintaining simplicity requires extra work.
  - Simplicity may not be done because of deadlines.
- Culture issues
  - Large companies have difficulty to move to working models in which processes are informal and defined by development team

Problems with agile methods

- General to incremental development and delivery – occurs when the system customer uses an outside org for system development.
  - Contracts may be a problem as with other approaches to iterative development.
  - Agile Methods need to rely on contracts in which the customer pays for the time required for system development rather than the development of a specific set of requirements.
    - Potential disputes may occur

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 (problems):
  - 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 (personnel turnover).**

Agile methods and software maintenance

- Agile methods enthusiasts argue that it is a waste of time to write this documentation and that the key to implementing maintainable software is to produce high-quality, **readable** code.
- However, without key documents (system requirements – tell software engineers what the system is supposed to do), it is **difficult to assess the impact of changes** (lacking bigger pictures).
  - Agile methods collect requirements piece by piece: *Non-coherent* requirements documents make system maintenance difficult and expensive.

Agile methods and software maintenance

- Client loses interests in providing feedback and evaluating software **after** delivery, even though they commit to do it.

- Agile Methods rely on team skills and intensive interconnections. If team is broken, without sufficient documentation, implicit knowledge may be lost. It's not easy for new team members to grasp the same level of understanding of the software in short time.

Agile Methods (a software process) vs. Incremental Development (a software process model)

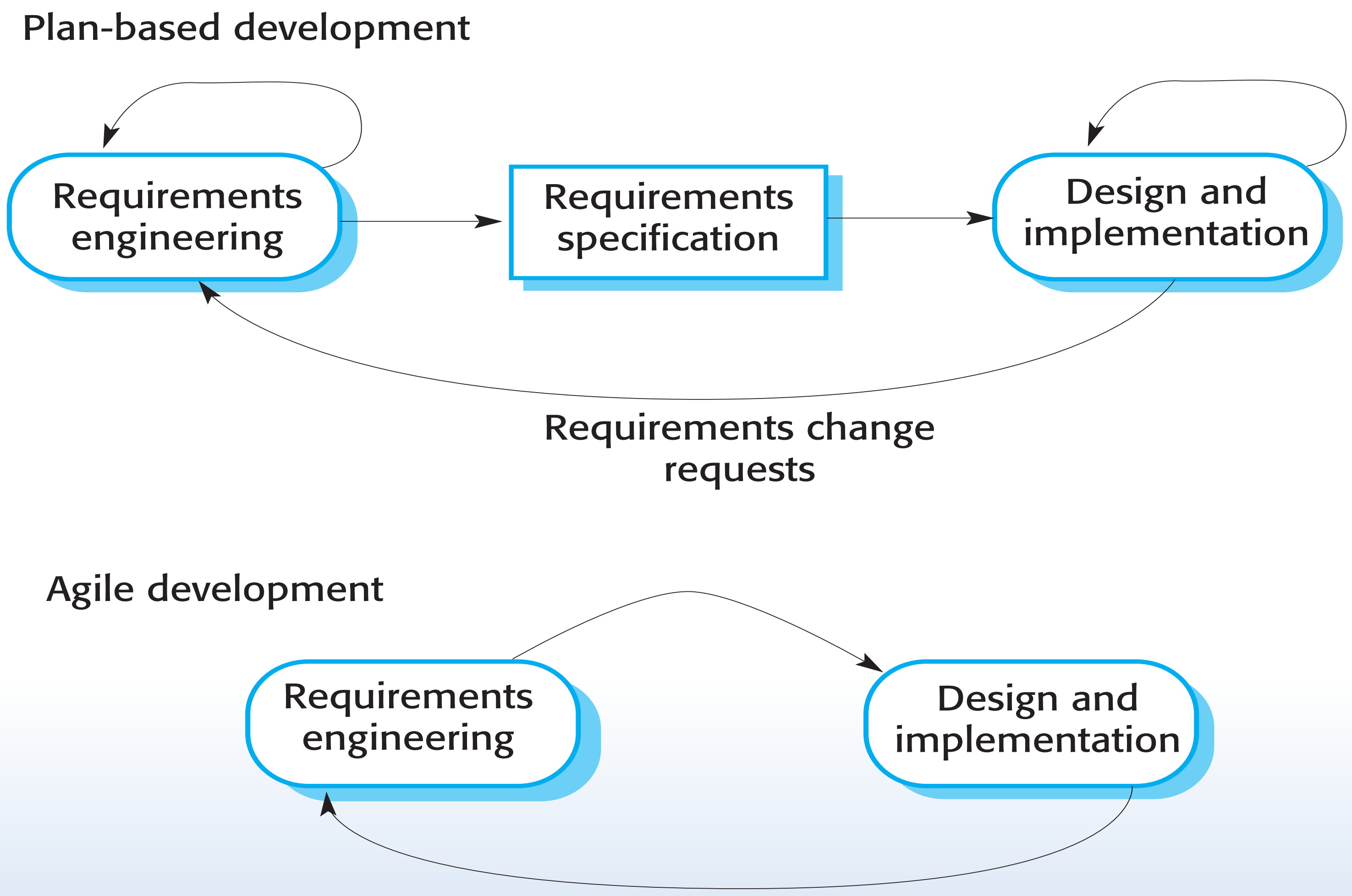
- A lot in common
  - – Incremental approach
  - – Embrace changes
  - – Client involvement
- Some differences (improvements)
  - – Eliminate complexity
  - – People not process

Waterfall, incremental development, and reuse-oriented are software process *models* (partial representation of a SW process). Plan-driven and agile methods are two different *classifications* for software process.

Plan-driven and agile development (two kinds of software process)

- 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.

Figure 3.2 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

Plan-driven and agile development

- Plan-driven development
  - Iterations occur **within** activities, with **formal** documents used to communicate between stages of the process.
  - It can support incremental development (see previous slide)
- Agile development
  - Iterations **across** activities, therefore, requirements and design are developed together (interleaved).
  - Could also generate documents if needed (called “spike”)

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.
  - 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 (e.g., IDEs with program visualization and analysis)
- 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.

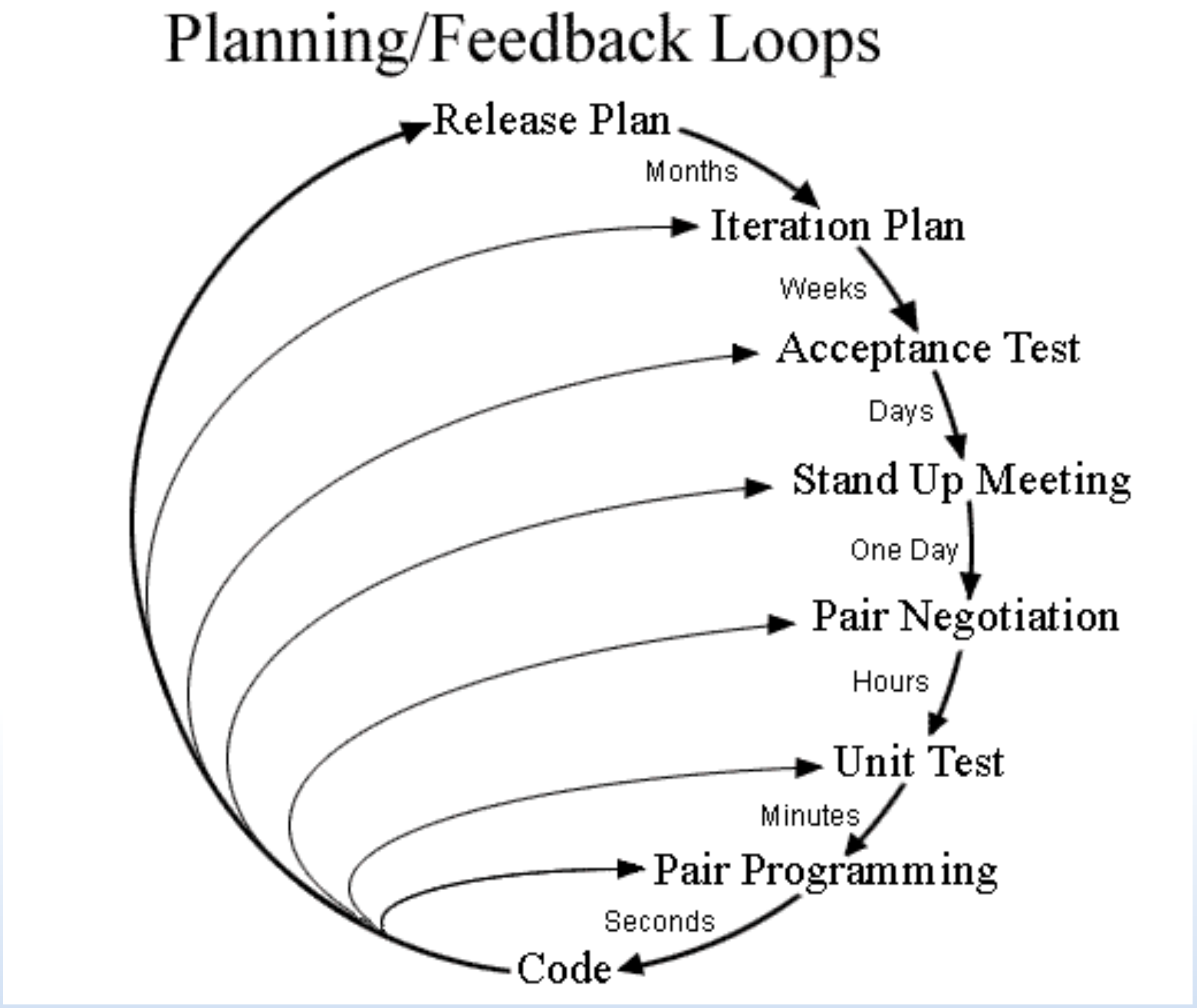
Technical, human, organizational issues

- In reality, the issue of whether a project can be labelled as plan-driven or agile is not very important.
- Practically, many companies who claim to have used agile methods have adopted some agile practices and have integrated these with their plan-driven processes.

Agile Process Models

- eXtreme programming
- Scrum

Extreme programming



Extreme programming

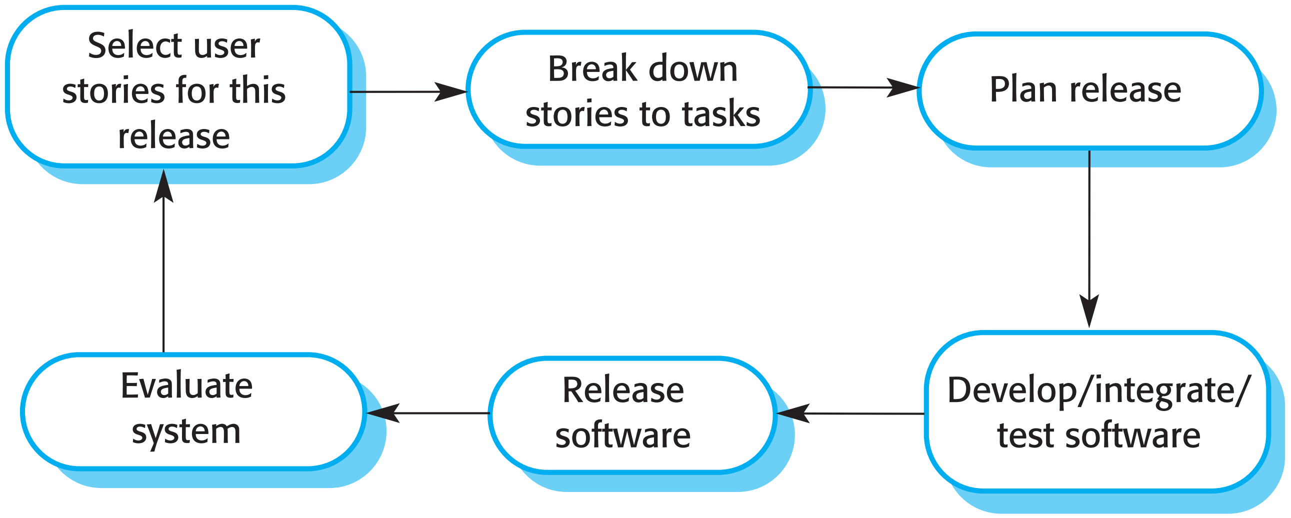
- Perhaps the best-known and most widely used agile method.
- Extreme Programming (XP) takes an 'extreme' approach to iterative development.
  - New versions may be built several times **per day**;
  - Increments are delivered to customers every 2 weeks;
  - Requirements are expressed as **scenarios** (called user **stories**), which are implemented directly as a series of tasks.
  - All tests must be run for every build and the build is only accepted if tests run successfully (TEST DRIVEN Approach, test cases first and then coding).

XP and agile principles

- Incremental development is supported through small, frequent system releases.
- Customer involvement means **full-time customer** engagement with the team (Customers define acceptance tests).
- Change supported through regular system releases, test-first development, refactoring to avoid code degeneration, and continuous integration
- Time management using timeboxing -- If it is impossible to complete the entire task in the timebox, the work may be reduced ("descoped")
  - Agile processes demand fixed time, not fixed features
- People (not process) are supported through pair programming*, collective ownership (similar to egoless programming) and a process that avoids long working hours.
- Maintaining simplicity through constant refactoring of code*.

*only in XP

Figure 3.3 The extreme programming release cycle


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

Requirements scenarios

- In XP, a **customer** or user is part of the XP team and is responsible for making decisions on requirements.
  - Specifying and prioritizing system 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.

Figure 3.5 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.

Planning Game

- After store cards are available, they are broken into implementation tasks. Then estimate on cost, time, resource and customer prioritization is carried out.
- If there are still **doubts** during planning game, prototype may be introduced. This is also called “spike”.
- Release deadlines are never slipped.
  - If there are development problems, the customer is consulted and functionality is removed from the planned release.

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
  - We will cover **how to design for change** using OOP concepts some in Csci 152 and most in CSCI 250 .
- XP considers that "design for change" is not worthwhile as changes cannot be reliably anticipated.
  - Again, CSCI 152/250 will cover how to make design for change worthwhile
- Rather, it proposes constant code improvement (**refactoring**) to make changes easier when they have to be implemented.
- XP people also claim that this may also solve the problem of structure degrading that most incremental development possesses.

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 and structure 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 <u>remove duplicate</u> code.
- ✧ Tidying up and <u>renaming</u> attributes and methods to make them easier to understand.
- ✧ The <u>replacement of inline code</u> with calls to methods that have been included in a program library.

Practices of XP

- Not all XP principles are adapted
  - Some don't use pair programming; others don't use refactoring too much
  - Yet, most XP people adapt small release, test-first development, and continuous integration.

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 (regression test).**
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 as well as defining acceptance testing**.

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
Part 2

Chapter 3 – Agile Software Development

Lecture 2

Testing in XP

- Agile methods may not have sufficient documentation, so testing may become informal. XP tackles this problem by Test-driven concepts

  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.

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**.
  - Behavior spec. and interfaces are usually defined in test cases. It reduces potential misunderstanding.
  - **Traceability** issues may be solved
    - Links between system req. and code implementation can be easily seen
  - Testers/programmers need to understand spec before writing test cases – avoid ambiguities and omissions
  - Avoid test-lag and skip of testing

Test-first development

- All previous and new tests are run automatically (more later) when new functionality is added, thus checking that the **new functionality has not introduced errors**. (more about JUnit later)

```
Test public void simpleAdd()
   {  Money m12CHF= new
      Money(12, "CHF");
      Money m14CHF= new
      Money(14, "CHF");
      Money expected= new
      Money(26, "CHF");
      Money result=
    m12CHF.add(m14CHF);

   assertTrue(expected.equals
   (result));
}
```

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 helps define/write tests as development proceeds (as seen in next slide). All new code is therefore validated to ensure that it is what the customer needs.
- Major difficulty of XP: 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.

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.
  - Question: When new functionality is added, do we still need to run existing test cases that are run before current increment?

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 is difficult to judge the completeness of a set of tests. Although you may have a lot of system tests, your test set may not provide complete coverage.
  - Some tools may help to see the code coverage in terms of logical conditions (but not req. coverage).
  - We will cover this in CSCI 152

Timeout
JUnit

jUnit

- Test framework
  - Initially developed to support Extreme Programming
  - Focused on unit testing
- Features and Goals
  - Trivial to add unit tests
  - Trivial to run tests
    - Tests must be complete automatic (no user input)
  - Trivial presentation of test results
    - Green = good
    - Red = one or more tests failed
  - Rerun tests without exiting the testing interface
  - Large assertion API to aid condition checking
- Idea: Unit tests are good, but a pain to write -> jUnit alleviates much of the pain

Writing a jUnit Test Case

1. Define a subclass of TestCase
2. Override the setUp() method to initialize object(s) under test (optional)
3. Override the tearDown() method to release any permanent resources you allocated in setUp() (optional)
4. Define one or more testXxx() methods that exercise the objects under test
5. Define a suite() method that creates a TestSuite containing all the testXxxx() methods of a TestCase
6. Define a main() method that runs the TestCase (not always necessary)

Basic jUnit Test Case

```
import junit.framework.*;

public class CubeTest extends TestCase {
  public CubeTest(String name) {
    super(name);
  }
  protected void setUp() {
    /* Any pre-test setup */
  }
  protected void tearDown() {
    /* Any post-test teardown */
  }
  public void testCubePositive() {
    assertEquals(Cube.calcCube(5), 125);
  }
  public void testCubeNegative() {
    assertEquals(Cube.calcCube(-5), -125);
  }
  public void testCubeZero() {
    assertEquals(Cube.calcCube(0), 0);
  }
}
```

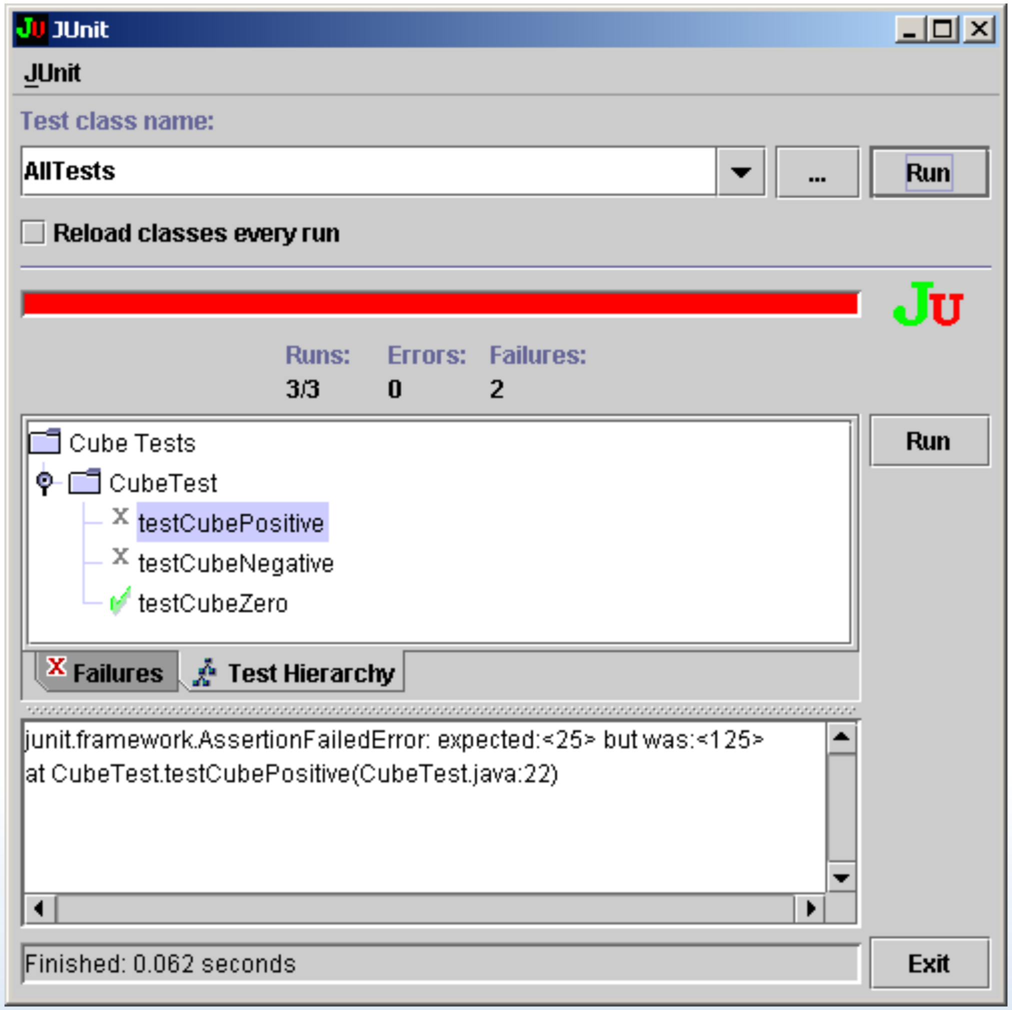
Basic jUnit Test Suite

```
import junit.framework.*;
import junit.runner.BaseTestRunner;

public class AllTests {

    private TestSuite suite;

  public static Test suite() {
    suite= new TestSuite("Cube Tests");
    suite.addTestSuite(CubeTest.class);
    return suite;
  }

  public static void main(String[] args) {
    junit.textui.TestRunner.run(suite());
  }
}
```

Running the jUnit Tests

- Three ways:
  - Text UI – java junit.textui.TestRunner <Suite>
  - AWT UI - java junit.awtui.TestRunner <Suite>
  - Swing UI - java junit.swingui.TestRunner <Suite>
  - Eclipse - preferred
- GUI features
  - Reload classes each run
  - Browse test cases
  - Run individual tests

Screenshot of Cube Test Run



Test class name: AllTests

☐ Reload classes every run

```
Runs:   Errors:   Failures:
3/3     0         2
```

Cube Tests
- CubeTest
  - testCubePositive
  - testCubeNegative
  - testCubeZero

```
junit.framework.AssertionFailedError: expected:<25> but was:<125>
at CubeTest.testCubePositive(CubeTest.java:22)
```

Finished: 0.062 seconds

Assertion API

```
assertEquals(T expected, T actual);

assertEquals(double expected, double actual, double delta);

assertNull(Object object);

assertNotNull(Object object);

assertSame(Object expected,Object actual);

assertTrue(boolean condition);

fail(String message);
```

- **If any exceptions are thrown, jUnit will catch and log them too.**

Pair programming

- In pair programming, programmers sit together at the same workstation to develop the software.
  - Or using Skype (or other software)'s desktop sharing
- 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.
  - But with a pair of very experienced programmers, the above statement may not be true.

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 (recall: democratic team).
- ✧ 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.

Chapter 3
Part 3

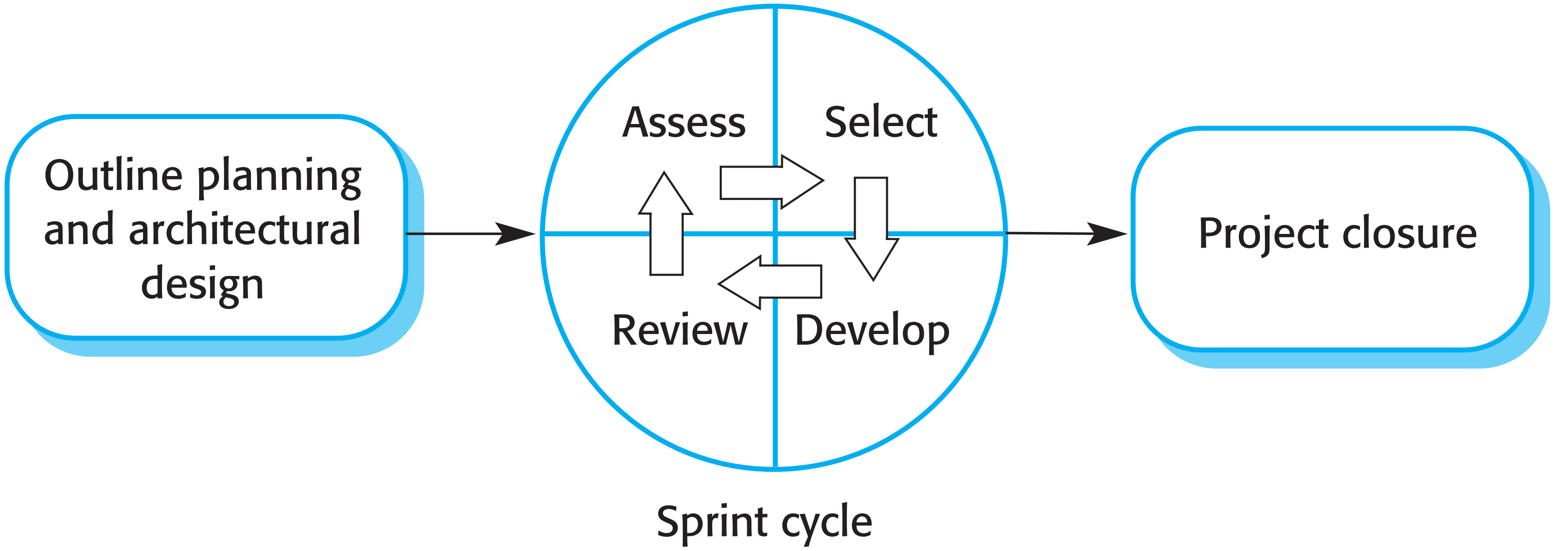
Agile project management

- ✧ 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 (because its req. are adapted incrementally) and the particular strengths of agile methods.

Scrum (Used in Pelco, Decade Software…)

- 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.

The Scrum process



The Sprint cycle

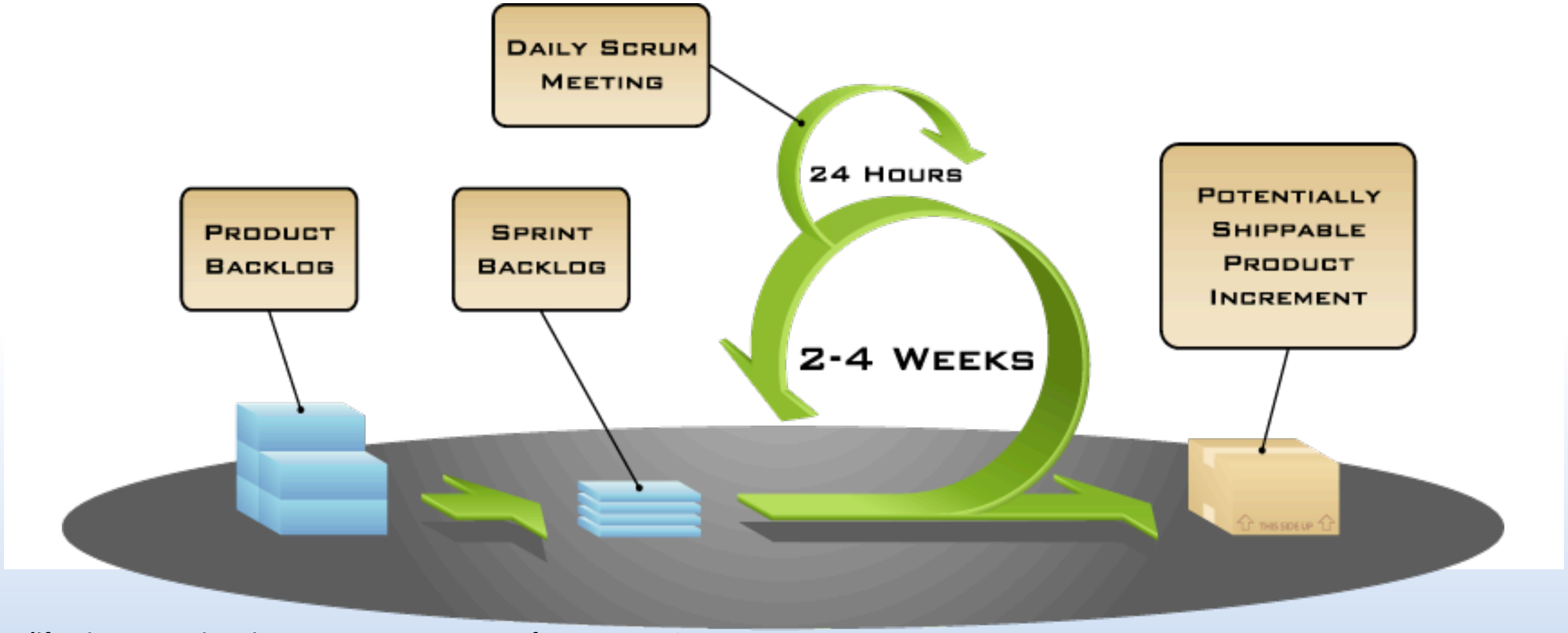
- ✧ Sprints are fixed length, normally 2–4 weeks. They correspond to the development of a release of the system in XP.
- ✧ The starting point for planning is the product backlog, which is the list of work to be done on the project.
- ✧ The selection phase involves all of the project team who work with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

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.

The Skeleton and Heart of Scrum

- Iterative and incremental skeleton.
- Daily inspections.
- Iterate until there is no longer funds.



The Skeleton and Heart of Scrum

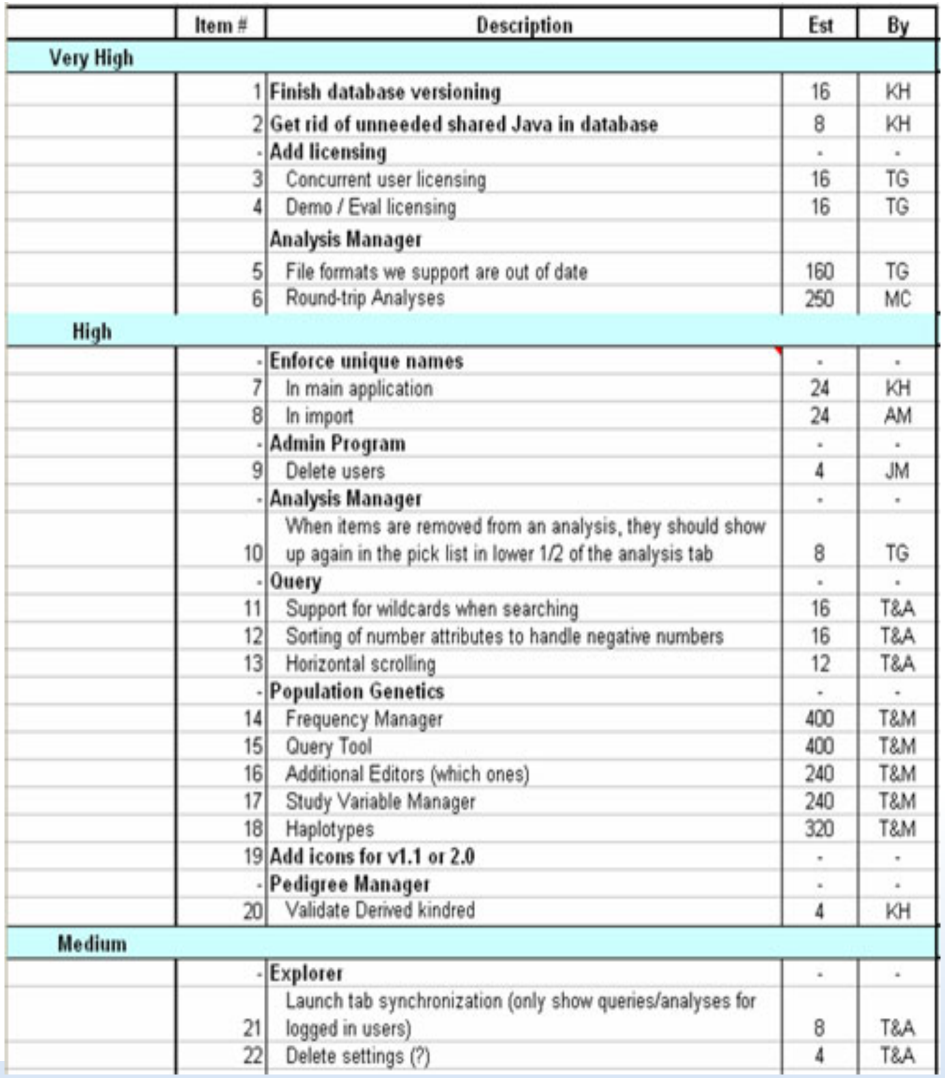
- Heart of Scrum lies in iteration.
  - Look at requirements
  - Considers available technology
  - Evaluates its own skills and capabilities.
  - Determine how to build the functionalities
    - Modify it daily as it encounters new complexities, difficulties, and surprises.
    - This creative process is the heart of Scrum's productivity.

Scrum Roles

- Product Owner
- The Team
- Scrum Master

Product Owner

- Responsible for representing the interest of everyone in the project.
- Initial and ongoing funding
- Initial requirements with prioritize product backlog, objectives and release plans.
  - Most valuable functionality is produced first and built upon.



	Item #	Description	Est	By
Very High				
	1	**Finish database versioning**	16	KH
	2	**Get rid of unneeded shared Java in database**	8	KH
	-	**Add licensing**	-	-
	3	Concurrent user licensing	16	TG
	4	Demo / Eval licensing	16	TG
		Analysis Manager		
	5	File formats we support are out of date	160	TG
	6	Round-trip Analyses	250	MC
High				
	-	**Enforce unique names**	-	-
	7	In main application	24	KH
	8	In import	24	AM
	-	**Admin Program**	-	-
	9	Delete users	4	JM
	-	**Analysis Manager**	-	-
	10	When items are removed from an analysis, they should show up again in the pick list in lower 1/2 of the analysis tab	8	TG
	-	**Query**	-	-
	11	Support for wildcards when searching	16	T&A
	12	Sorting of number attributes to handle negative numbers	16	T&A
	13	Horizontal scrolling	12	T&A
	-	**Population Genetics**	-	-
	14	Frequency Manager	400	T&M
	15	Query Tool	400	T&M
	16	Additional Editors (which ones)	240	T&M
	17	Study Variable Manager	240	T&M
	18	Haplotypes	320	T&M
	19	**Add icons for v1.1 or 2.0**	-	-
	-	**Pedigree Manager**	-	-
	20	Validate Derived kindred	4	KH
Medium				
	-	**Explorer**	-	-
	21	Launch tab synchronization (only show queries/analyses for logged in users)	8	T&A
	22	Delete settings (?)	4	T&A

The Team

- Developing the functionality
- Collective responsible for the success of each iteration and the project as a whole.

✧In summary, 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 Master

- Responsible for the Scrum process
- Teaching Scrum to everyone
- Implementing Scrum
  - Ensuring that everyone follows Scrum’s rules and practices
- In summary, 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.

Sprints

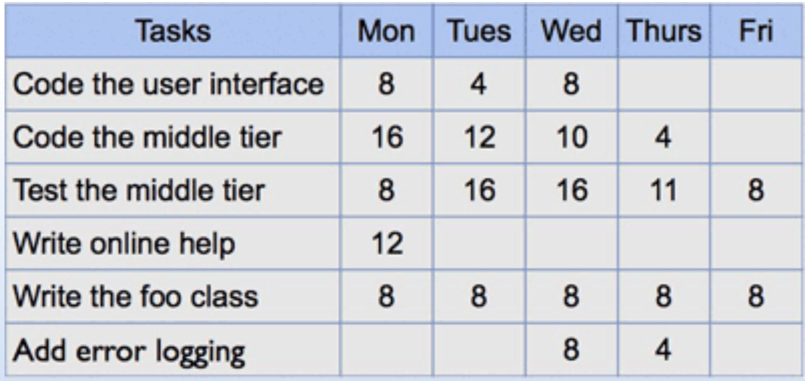
- All works is done in sprints.
- Each sprint is an iteration of 30 consecutive calendar days (some says 2-4 weeks).
- Each sprint is initiated with a **Sprint planning meeting**, where the *Product Owner* and *Team* meet to collaborate about what will be done next sprint
- Product owner tells the team what is desired by referring to the highest priority Product Backlog

Sprints cont.

- The team tells the Product Owner how much of what is desired it believes it can turn into functionality over the next Sprint.
- Sprint planning meetings: **No longer than 8 hours**
- **Goal:** To get to work, not think about working
- Sprint Meetings: Two Parts

Sprints cont.

- Sprints Meeting: Two parts
  - First Four Hours: Spent with *product owner* presenting highest priority *Product Backlog* to the team. Team then questions him/her about the Product Backlog content or ideas (similar to interview)
  - The teams selects as much Product Backlog as it believes it can turn into a completed increment of potentially shippable product functionality by the end of the Sprint
    - *Sprint backlog* is introduced as commitment of tasks to be completed during current sprint
  - Second Four Hours:
  - *Teams* plans out the Sprint



Tasks	Mon	Tues	Wed	Thurs	Fri
Code the user interface	8	4	8		
Code the middle tier	16	12	10	4	
Test the middle tier	8	16	16	11	8
Write online help	12				
Write the foo class	8	8	8	8	8
Add error logging			8	4	

Daily Scrum

- **Daily Scrum:** Every day the team gets together for a 15-minute meeting (STAND UP MEETING)
- At the daily scrum each team member answers three questions:
  - *What have you done* on the project since the last Daily Scrum meeting?
  - *What do you plan* on doing on this project between now and the next Daily Scrum meeting?
  - *What impediments* stand in the way of you meeting your commitments to this Sprint and the project?
- Purpose: Synchronize the work of all team members daily and to schedule any meetings that the Team need to forward progress.

Sprint Review Meeting

- Held at the end of the Sprint
- 4 hour time-boxed meeting
  - *Team* presents what was developed during the Sprint (Functionality is presented)
  - Meeting is used to collaboratively determine what the Team should do next (based on how much have accomplished against its earlier sprint planning meeting)
  - After the Sprint Review and prior to the next Sprint planning meeting, the *Scrum Master* holds a **Sprint retrospective** meeting with the Team the *Product Owner* and any other stakeholders who want to attend

Sprint Retrospective

- 3-hour, time-boxed meeting
- Scrum Master encourages the Team to revise, within the Scrum process framework and practices, its development process to make it more <u>effective</u> and <u>enjoyable</u> for the next Sprint.
- Together, the Sprint planning meeting, the Daily Scrum, the Sprint review, and the Sprint retrospective constitute the empirical inspection and adaptation practices of Scrum

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.

How to implement Scrum in Academia?

- Can scrum planning meeting be implemented?
- Can daily scrum be implemented?
- Can scrum review meeting be implemented
- Can scrum retrospective meeting be implemented?

XP vs. Scrum

- XP’s each sprint is1-2 weeks; Scrum is about 2- 4 weeks.
- Scurm master is responsible not letting customers interfere during each spring. XP allows customers request changes in the middle.
- XP’s priority is determined and fixed by customers. Scurm’s product owner (customers) suggest priority but team can revise it based on implementation needs.
- XP prescribes engineering practices (e.g., refactoring, test-driven); Scrum doesn’t.

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.