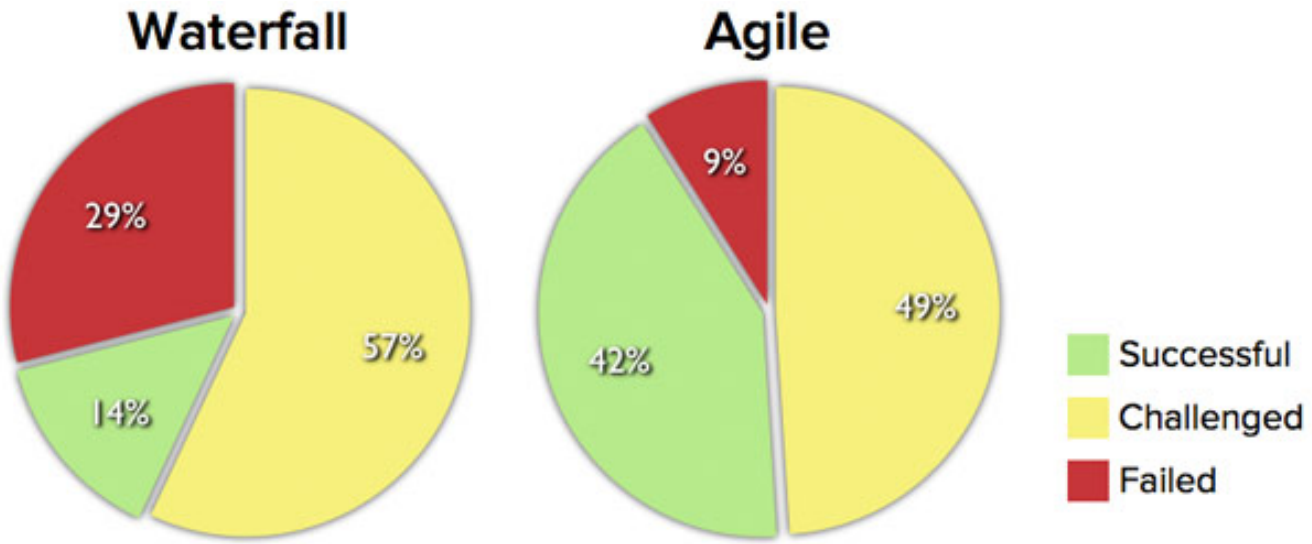
Agile Processes and Extreme Programming

In the past two decades there has been a move towards lightweight software development processes, collectively known as **Agile Processes**.

Traditional (waterfall) approaches try to **prevent** error by adding more administration and process.

Agile process **accept** that error and new requirements occur, and remain flexible (agile) so as to be able to deal with them.



Source: The CHAOS Manifesto, The Standish Group, 2012.

Agile Manifesto

The so-called **Agile Manifesto** was proposed in 2001 by a number of people, including well-known names like Kent Beck, Robert Martin, Ward Cunningham, Dave Thomas and others. It read:

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.

Extreme Programming

One well-known Agile process is Extreme Programming, or **XP** for short.

The name comes from viewing the development process as being controlled by a dashboard.

On the dashboard are controls that represent good things like "early testing," "rapid prototyping," "customer involvement" etc.

"Extreme" refers turning all these dials to maximum to see if a stable software development process results.

Footnote: **Scrum** is a more popular Agile methodology. We cover XP as it more focussed on engineering practices. The two can be combined as well.

Main Features of Extreme Programming

Three main features of XP:

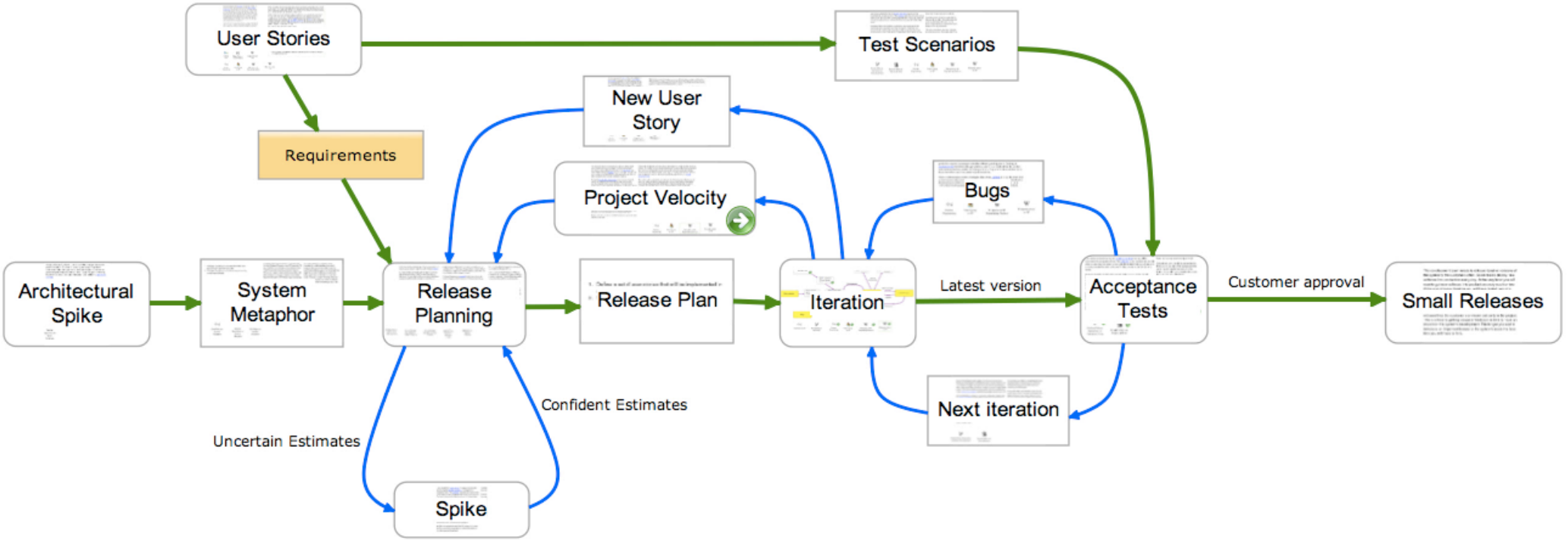
Continual Customer Involvement: Ideally there is a customer on-site for the duration of the project (or a product manager playing that role).

Testing: Testing is central to the process as we shall see.

"*Pay as you go design*": The design is only as complicated as it needs to be for the current iteration; no more.

In this overview, I'll focus on the **engineering aspects** of XP, especially those dealing with testing and design.

Extreme Programming Overview



Eliciting the Requirements

The requirements are elicited in a process called the **Planning Game**.

A number of **user stories** are obtained. A story is like a UML Use Case, but is expressed in the customer's terms.

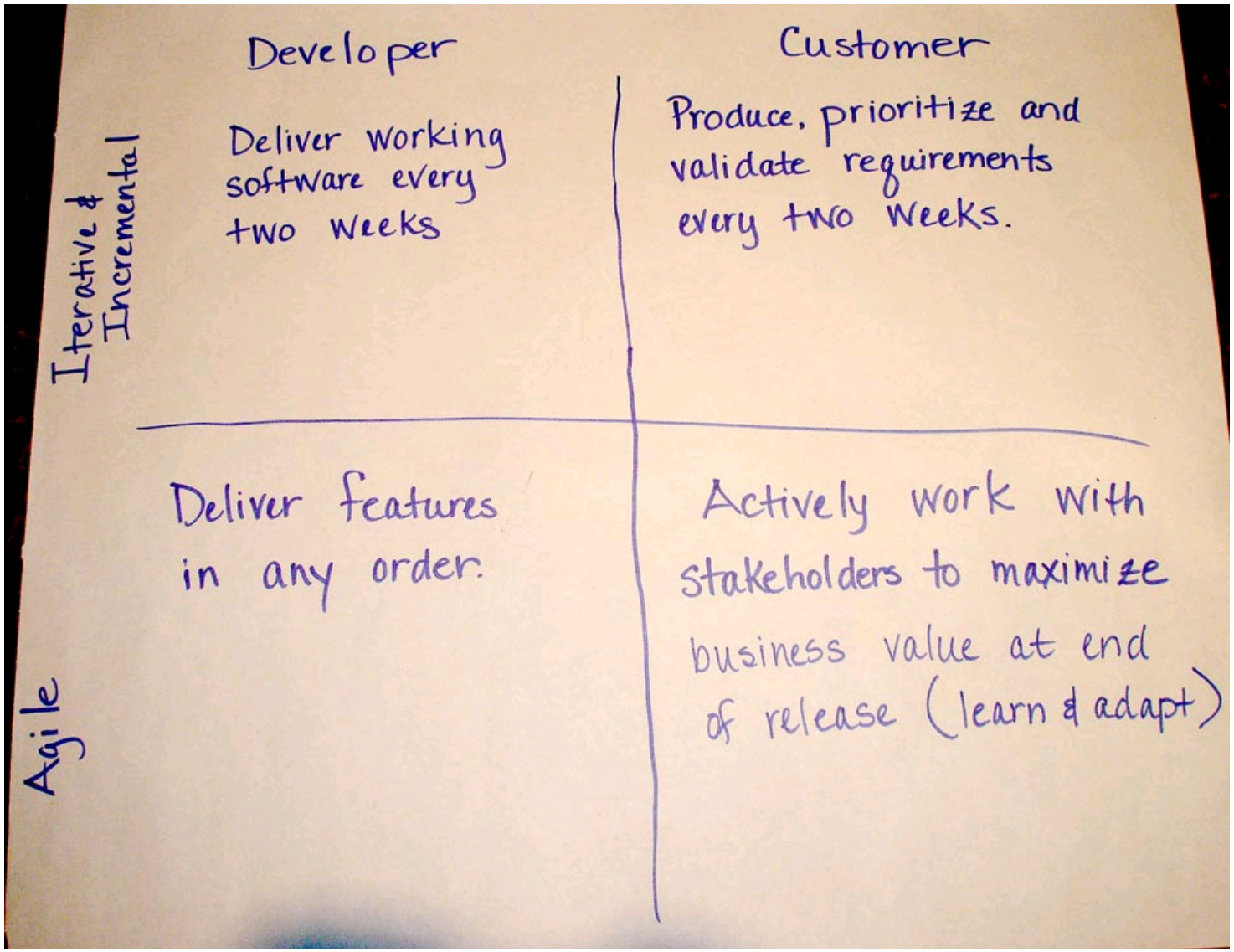
There is no assumption that these will remain fixed during the project (**embrace change**).

The user must be able to search for a book by Title, and display the results as a list.

Risk: Med. **Cost**: 1 point

Customer and Developer Roles

Summarised as follows:



Frequent Releases

The system is built in a number of releases. On each iteration, the customer selects a number of high-priority user stories to be implemented.

An iteration is typically 2 or 3 weeks in length, so the customer sees a new version of the system regularly.

So what happens on each iteration?

Little Upfront Design

There is little or no upfront design done in XP. UML diagrams may be drawn, but principally for the purpose of immediate communication. Coding starts immediately.

Everything is kept as simple as can be. Flat files are used rather than a full-featured database, REST preferred over SOAP, etc. Advanced technology is used only where necessary.

YAGNI
You
Ain't
Gonna
Need
It.

So where does the design come from? Isn't this just code-and-fix?

There are a number of XP practices that are vital that prevent the process from degenerating into code-and-fix. Two key processes we examine are:

- **Testing**
- **Refactoring**

Testing happens first

Testing is a good thing, so in XP it is the very first thing that is done.

Before a developer writes code to implement any functionality, they write a test case for it.

Now they have a concrete aim. They must write code so that the test case runs without error.

The testing is automated, so they can run the test suite at any time. This reassures them that they code they have already written is still correct.

A dyed-in-the-wool XP developer will never write any program code unless there is a test case that is failing. Why should they?

Test cases check correctness

The test suite is actually used as an **operational specification** of what the system is to do.

Of course, if the test suite is wrong or omits important cases, the code developed will be wrong or insufficient as well.

At least a test suite is a **usable** specification. Contrast with a complex UML model where it's impossible to assess if a given program is a correct implementation.

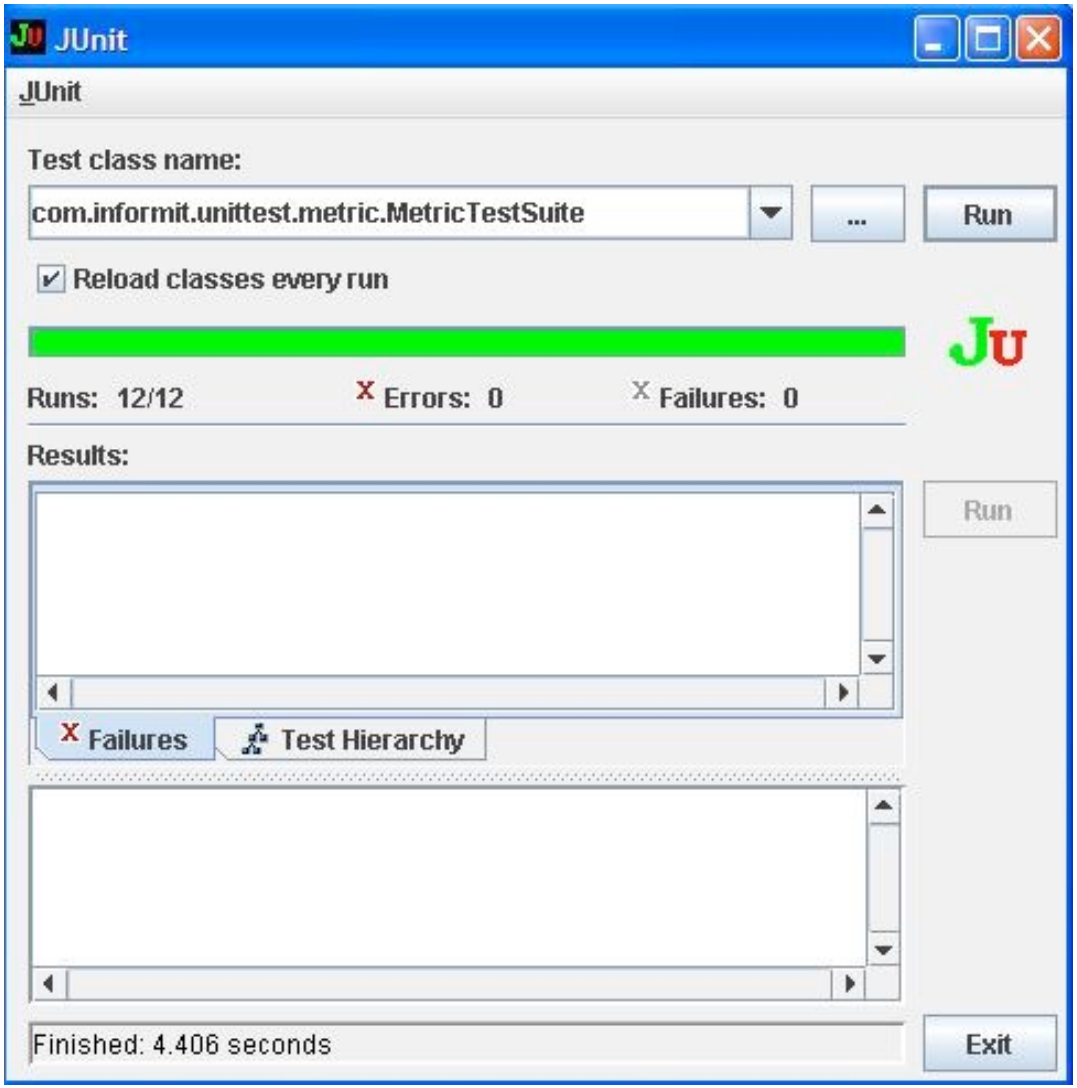
The promotion of testing to this level of importance has a vital characteristic that can be exploited:

> Whenever the code is updated, it is a simple matter to test if a new bug has been introduced.

i.e. Regression Testing

Sample JUnit Screenshot

For example, here's a JUnit screenshot after running the unit tests for a class named MetricTestSuite:



Emergent Design through Refactoring

How can a program be designed well if there is no proper design done in the first place?

In XP, the design of the program **evolves** during the project. The developer does not aim for a certain design; they regard the design as a flexible entity that is likely to change.

The design evolves through the process of **refactoring**.

Refactoring is the process of improving the design of a program, without changing its external behaviour.

Refactoring goes on throughout an XP project, both in terms of continual low-level improvements to the code performed by one developer, to large-scale, project-wide design overhauls.

Refactor to make the Design more Flexible

An XP developer is very involved with the code. As they extend it to deal with a new user story, they sense whether or not the new user story fits in easily.

A user story is never forced in where it doesn't fit easily. If the design doesn't support the change elegantly, it is **refactored** in order to accommodate the new user story.

Here's where testing comes into its own. A large-scale refactoring that changes many classes in the system could introduce many errors.

However, by running the automated test suite after a refactoring, it is likely that any new errors can be detected.

Floss Refactoring, Root Canal Refactoring

A refactoring may involve a small change to a program, such as renaming a field or a method.

It may be more significant, such as splitting a large, overly-complex class into two classes. This may involve updating a lot of code to ensure program behaviour doesn't change.

Radical refactoring occurs when the whole program design is reviewed and overhauled. This is a process that may take a few weeks and involve the whole programming team.

"Root Canal" Refactoring

The XP View of Code

Consider how an XP developer feels about the code:

The code is ***it***. There is little or no documented design, except what is in the code.

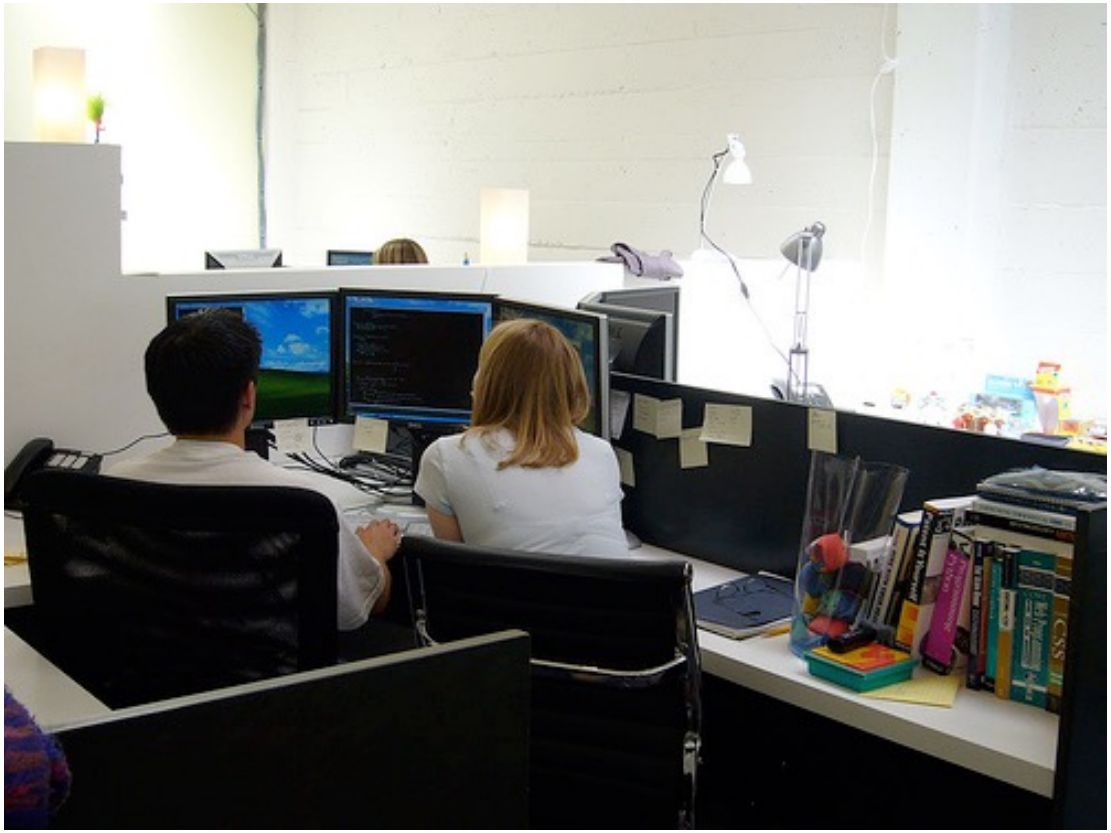
The **code is elegant and clear**. It has to be because colleagues will have to work with it as well. Refactoring is as important as new development.

The **design is simple**. The design is only for the current system, not for possible futures.

The code works. This can be easily verified at any time by running the automated test suite.

Pair Programming

Pair programming: No-one writes code alone. Code is written by two people; one writes the code, the other watches and considers the effect of the code on the larger system.



Most controversial XP practice.

Jan 2017: 41 jobs on jobs.ie mention pair programming explicitly. Jan 2018: 3 jobs

Pair programmers are peers!

Other XP Practices

Collective Ownership: Traditionally, a developer works on work part of the system and becomes expert in that area. In XP, the code is collectively owned.

* Anyone can modify it
* Single coding style
* Original authorship irrelevant

40-hour week: The massive efforts typically expended by a software team before a deadline are frowned upon in XP, as the system design may never recover from the damage done. **Design debt** is permitted, but must be repaid!

Continuous Integration: 'Big bang' integration is famously problematic. XP favours **continuous integration**, with even hourly builds being employed in some companies.

Web Resources

There are many web resources in the Agile area. See what you find most useful.

http://www.agilelearninglabs.com/resources/scrum-introduction is a clear introduction to Scrum, the best known Agile process. Much is similar to XP (XP is more engineering based). For the purposes of this module, understanding Agility is important; not the details of e.g. Scrum vs. XP.

Summary

XP is one of a number of Agile development methodologies. Others include Scrum, DSDM…

It is most suited to small teams (<=12 people) on projects where the system requirements are likely to change frequently.

In a way, it is the complete antithesis of traditional development methodologies.

It has worked well in many companies, but is no doubt not the last word in this complex area.