# CMPT 276 Class 07: The Agile Manifesto and Extreme Programming (XP)

Dr. Jack Thomas

Simon Fraser University

Fall 2020

# Topics

1. What is **Agile** trying to do?
2. How to **choose Plan-Driven vs Agile?**
3. What practices go into **Extreme Programming (XP)** development?
4. How to **write tests** while **writing new code**?

# Rapid Software Development

- Rapid development and delivery is often the most important requirement for a software system.
  - **Businesses change fast** - it’s practically impossible to have stable software requirements.
  - Software has to evolve quickly to keep up.
- **Agile** aims at **rapid software development**:
  - Interleave **specification**, **design** and **implementation**.
  - **Incrementally** developed with **users evaluating each version**.

## Agile Methods

- Inspired by dissatisfaction with overheads in plan-driven software methods.
- **Agile Methods are:**
  - **Focused on the code** rather than the plan or design.
  - Based on **iterative approach** to software development;
  - Intended to **deliver working software quickly,** and evolve it quickly to meet changing requirements.
- **The Aim:**
  1. To **reduce overheads** in the software process (e.g. limit documentation)
  2. To **respond quickly** to changing requirements without excessive rework.

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.

The Agile Manifesto

Signed by Kent Beck, Robert Martin, Martin Fowler, and 14 other founders of Agile development.

# Scrum and Agile Manifesto Values

- Agile Manifesto **Values** can be summarized as **Inspect and Adapt**
  1. *Inspect*: Perceive current state of the project.
  2. *Adapt*: Adjusting the product and process to respond.

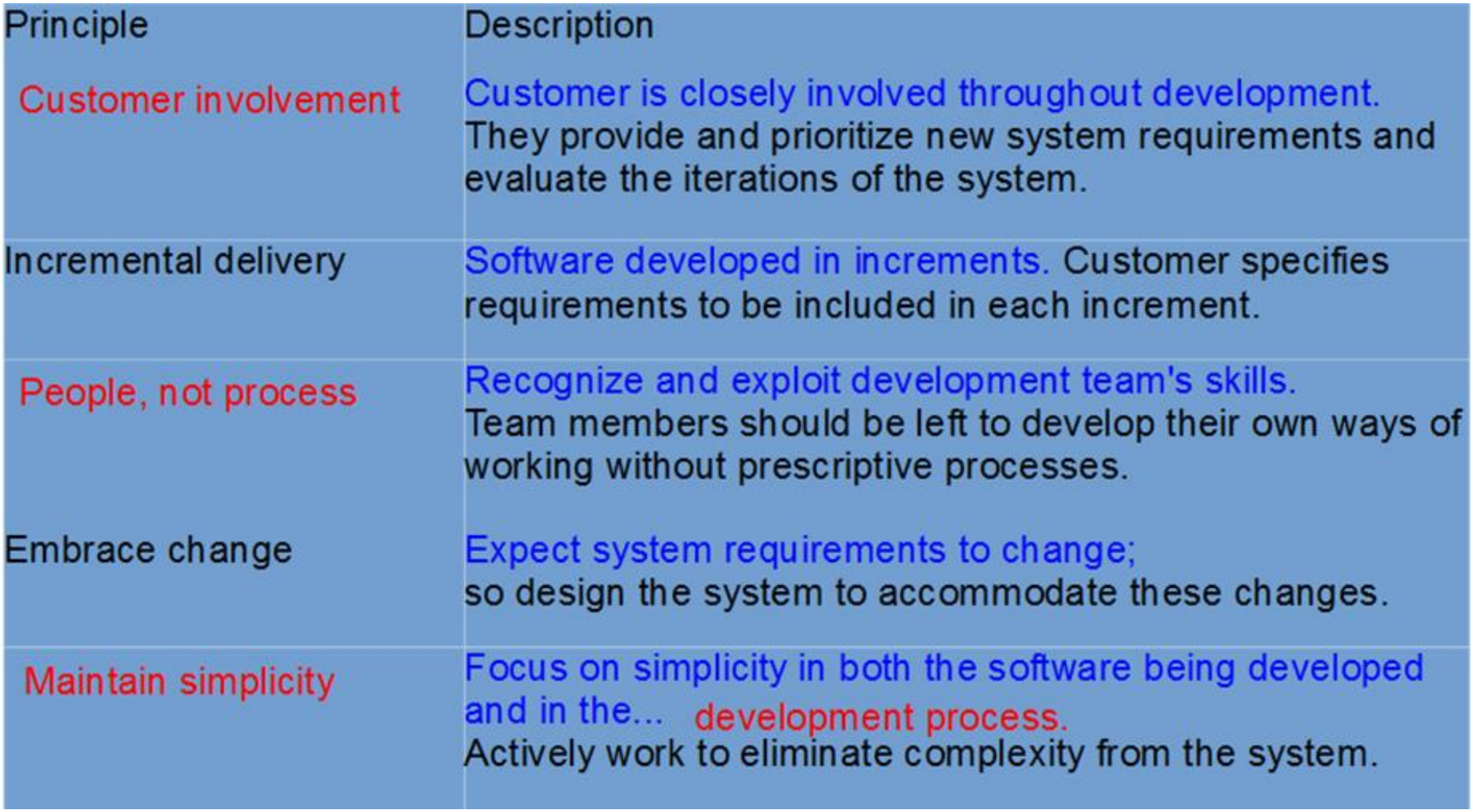
- For each Agile Manifesto Value:
  – Explain how it is “Inspect and Adapt”.
  – Explain how Scrum achieves the value.

# Inspect & Adapt in Scrum

- Scrum is about learning through inspecting and adapting:

1. **Daily stand-up**: For the sprint, keep on track.
2. **Sprint demo**: For the product, ensure most valuable features being added.
3. **Retrospective**: For the team, continuous improvement to the group.

# Principles of Agile Methods



| Principle | Description |
|---|---|
| Customer involvement | Customer is closely involved throughout development. They provide and prioritize new system requirements and evaluate the iterations of the system. |
| Incremental delivery | Software developed in increments. Customer specifies requirements to be included in each increment. |
| People, not process | Recognize and exploit development team's skills. Team members should be left to develop their own ways of working without prescriptive processes. |
| Embrace change | Expect system requirements to change; so design the system to accommodate these changes. |
| Maintain simplicity | Focus on simplicity in both the software being developed and in the... development process. Actively work to eliminate complexity from the system. |

# Agile Applicability

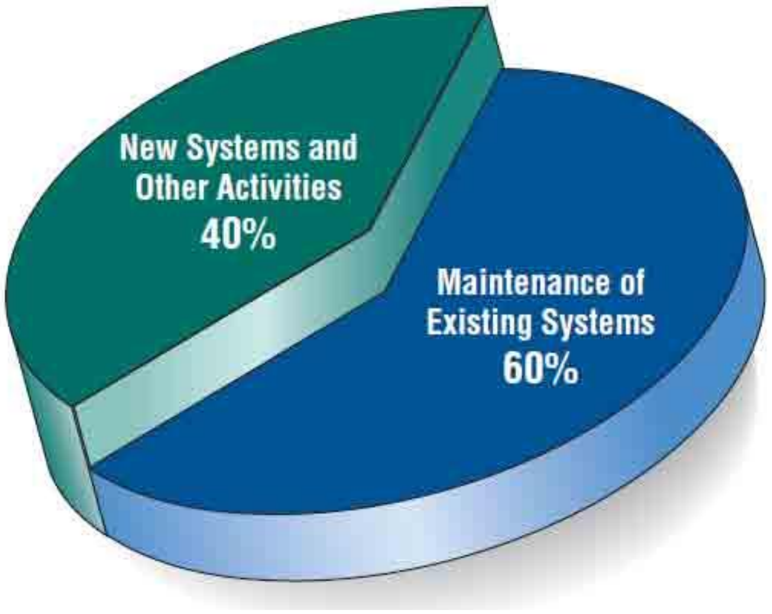
- Agile is applicable to developing **small or medium-sized products**
- It requires:
  1. Customers **willing to be involved** in development process
  2. Few **extreme regulations** affecting the software.
- **Problems with Agile**
  - **May not scale** to large systems, best with small, tightly-integrated teams.
  - **Intense interactions** may not suit development team.
  - Prioritizing change is **hard with multiple stakeholders**
  - Maintaining **simplicity requires extra work**.

# Agile & Maintenance

- **Maintenance**
  - More $ on **maintenance** than on **initial development**.
  - To succeed, Agile has to support both!

**The 60/60 Rule**

- Proposed by Robert L. Glass in a 2001 IEEE article.
- Roughly 60% of software's life cycle is spent in maintenance.
- Of that, roughly 60% of maintenance time is spent on enhancements.



# Agile & Maintenance

- **Lacking Formal Documentation**
  - Good documentation supports maintenance.
    - BDUF has docs, but **docs get out of date fast**
  - Agile focuses on **clear code expressing the design** which is better maintained.
  - Agile **can create required documents** as needed.
- **Changing needs in Maintenance**
  - Agile good at accommodating change.
  - Challenging if **original developers leave team**.
    - Process is less formal, so may be poorly understood by new group of devs.

# Choosing Agile or Plan-Driven

Most projects use elements of **both** plan-driven and agile processes. Try sorting the following:

- Need a detailed specification and design before moving to implementation? PLAN
- Is an incremental delivery strategy with rapid feedback realistic? AGILE
- Is it a small-medium size system being developed? AGILE
  - Agile is most effective with a small team who can communicate informally.
- Does it require a lot of analysis before implementation? PLAN
  - e.g. Real-time systems with complex timing requirements.

# Choosing Agile or Plan-Driven

- Is it expected to be a long lived system where maintainers need design docs? **PLAN** **AGILE**
- Are good tools available to support development? **AGILE**
- Is the team spread out/outsourced and needs design docs to communicate? **PLAN**
- Does the team have a plan-based development culture (engineering)? **PLAN**
- Is detailed documentation required for external regulatory approval? **PLAN**

## AGILE IS THE PRODUCT OF ALIENATED WORKERS

- **Marx's Theory of Alienation**: Workers become alienated (detached, dissatisfied, uninvested) from their work because they don't own or control it, don't profit from it, don't even benefit from it in their daily lives.
- **Software developers are also workers**, who emerged at a low-point in class consciousness in the 90s.
- Agile, open-source, hacker culture, etc., are **reactions to the corporatization of the digital landscape**.
- Yet Agile is one of the most dominant software development paradigms, including major corporations.

# It’s time to get… *XTREEEEEME!!!!*

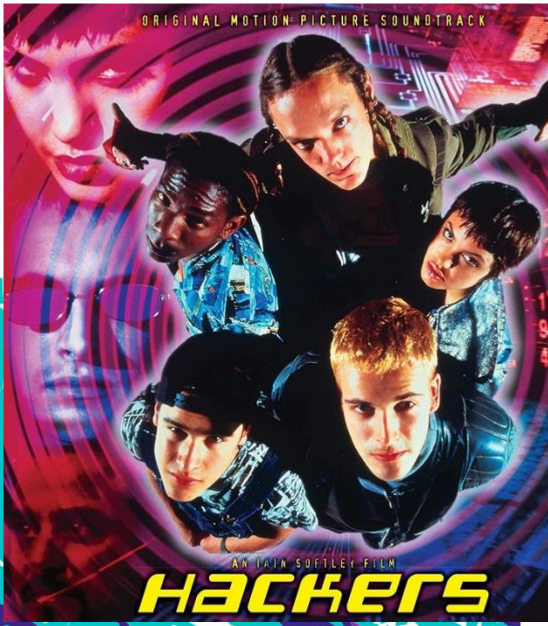


Image credits: https://knowyourmeme.com/memes/solo-jazz-pattern
http://iruntheinternet.com/08832
https://www.amazon.ca/Game-Color-Console-Atomic-Purple/dp/B00000J97G
https://www.monstersinmotion.com/cart/soundtrack-cd-item-list-h-c-12_80/hackers-25th-anniversary-original-soundtrack-2cd-p-27826.html

## Extreme Programming (XP)

- Created by Kent Beck in 1999
- Extreme Programming (XP) takes best software dev. practices to an ~~extreme~~ ***XTREEEEME!***
  - **Short iterations**, deliver working software often (2 weeks)
  - Frequent **communication** with customer
  - **All tests automated and passed** for each change
  - Based on **12 XP Practices**
- Extreme Example
  - **Normal**: Code reviews are good.
  - **Extreme**: Continuous code review via pair programming.

# Select XP Practices

| Practice | Description |
|---|---|
| Small releases | Start small: First develop minimal useful set of functions which deliver business value. Release often: Releases are frequent and incrementally add functionality. |
| Simple design | Design is only done to support current requirements, not any possible future ones. |
| Test-driven development | Automated unit tests are written for a class before the class is written. |
| Refactoring | Code kept simple and maintainable by continuous refactoring by all developers. |

## More XP Practices

| Practice | Description |
|---|---|
| Pair programming | Developers work in pairs, always checking each other's work and providing support. |
| Collective ownership | All developers work on all parts of the code. Shared responsibility for the code, and no one developer has all knowledge about an area. |
| Continuous integration | Changes integrated into system as soon as the are completed. |
| Sustainable pace | Large amounts of overtime are discouraged: would compromise productivity and code quality. |
| On-site customer | Customer representative (user) is a full time member of the development team: brings the team requirements and priorities. |

# Pair Programming

- Developers **work in pairs**, sitting together to develop code.
  - **Pairs change** so everyone works together.
- Fosters **common ownership of code** and spreads knowledge across the team.
  - Reduce problem when key developers leave.
  - No one person blamed for bugs.
- **Informal review process**, as each line of code is instantly reviewed.
- **Encourages refactoring**, as the whole team gets benefit of clean code.
- Productivity with P.P. ≈ two people working independently.
- **Fringe benefit**: reduced wrist stress!

# XP and Change

- Conventional wisdom: **Design for Change**
  - It is worth spending time and effort anticipating changes as this reduces costs later in the life cycle.

- XP's view: **Change cannot be reliably anticipated**.
  - XP uses refactoring to constantly improve code.
  - This makes changes easier when they have to be implemented.

# Refactoring

- Developers look for possible code improvements and make these improvements **even when there is no immediate need for them**.
  - This improves the **understandability** of the code and reduces the need for documentation.
- Changes are easier to make because code is well-structured and clear.
  - However, some changes require **architecture refactoring** and this is **much more expensive**.

## Examples of Refactoring

- **Renaming methods** to make its purpose clearer.
- Extract a method to **make a long function shorter** or reduce duplicate code.
- Extract a class to **split a class** which does two things into two classes.
- **Single Responsibility Principle (SRP):**
  - Each class should only do one thing.
  - It should have only one reason to change.

## Overview of Testing in XP

- **Testing is Central to XP**
  - Program is tested after every change.
- **Features of XP Testing**:
  - Test-driven development (TDD).
  - User involvement in test development and validation.

# Test Automation

- Tests are written as executable components, **before the task is implemented**.
  - Automated testing frameworks (**JUnit**) run stand-alone tests which **simulate input and check results**.
- Tests run whenever new functionality is added.
  - Tests can be run quickly and easily.
  - Catch problems immediately.
- **Automated Verification**
  - Gives developers the confidence and security of knowing nothing broke.
  - Needed to support aggressive refactoring.

# Test-Driven Development (TDD)

- **Interleave** writing tests and writing code.
- The idea is that **a red bar is your only excuse to add more code**.
  - Write a test that fails, add code to make it pass.

Image credit: [https://www.gettyimages.ca/detail/photo/an-airbag-deploying-during-a-crash-test-royalty-free-image/104574303](https://www.gettyimages.ca/detail/photo/an-airbag-deploying-during-a-crash-test-royalty-free-image/104574303)

## Process

1. Find some small functionality to implement (2-3 lines of code).
2. Write the JUnit test for it.
3. Run all tests
   – Will give you a red bar.
4. Write the code to fix the bug.
5. Run all tests, hopefully giving you a green bar.
   – Refactor as required.
6. Goto 1.

# Benefits of TDD

- **Code Coverage**
  - Each line of code written to satisfy a test, so all lines are tested.
- **Regression Testing**
  - Run old tests after new changes to prove you have not broken previous features.
  - A regression test suite is continually developed as a program is developed.
- **Simplified Debugging**
  - When a test fails, it should be obvious where the problem lies (the new stuff!).
- **System Documentation**
  - The tests themselves are a form of documentation that describe what the code should be doing.

# Summary: Taking programming… *TO THE XTREME!*

- **Agile**: incremental development focused on
  - Rapid development
  - Frequent releases
  - Reducing process overheads
  - Producing high-quality code
- **Agile vs plan-driven** depends on the type of software
- **Extreme Programming**: Good practices done to the extreme.
  - TDD to ensure well tested code