Agile Processes: eXtreme Programming

XP

"XP is a lightweight methodology for small to medium sized teams developing software in the face of vague or rapidly changing requirements "

Kent Beck

Values

Principles

Practices

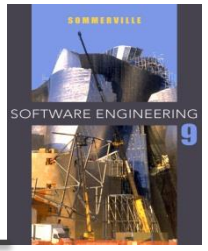
My House is always clean

Self / Hire Some

XP's Four Values

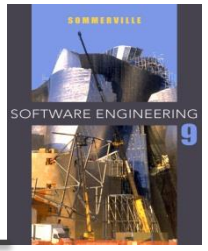
- **Communication**. Most projects fail because of poor communication. So implement practices that force communication in a positive way.
- **Simplicity.** Develop the **simplest product** that meets the customer's needs
- **Feedback**. Developers must obtain and value feedback from the customer, from the system, and from each other.
  - The same as standard Agile values: *value customer collaboration over contract negotiation.*
- **Courage.** Be prepared to make hard decisions that support the other principles and practices.

Extreme Programming



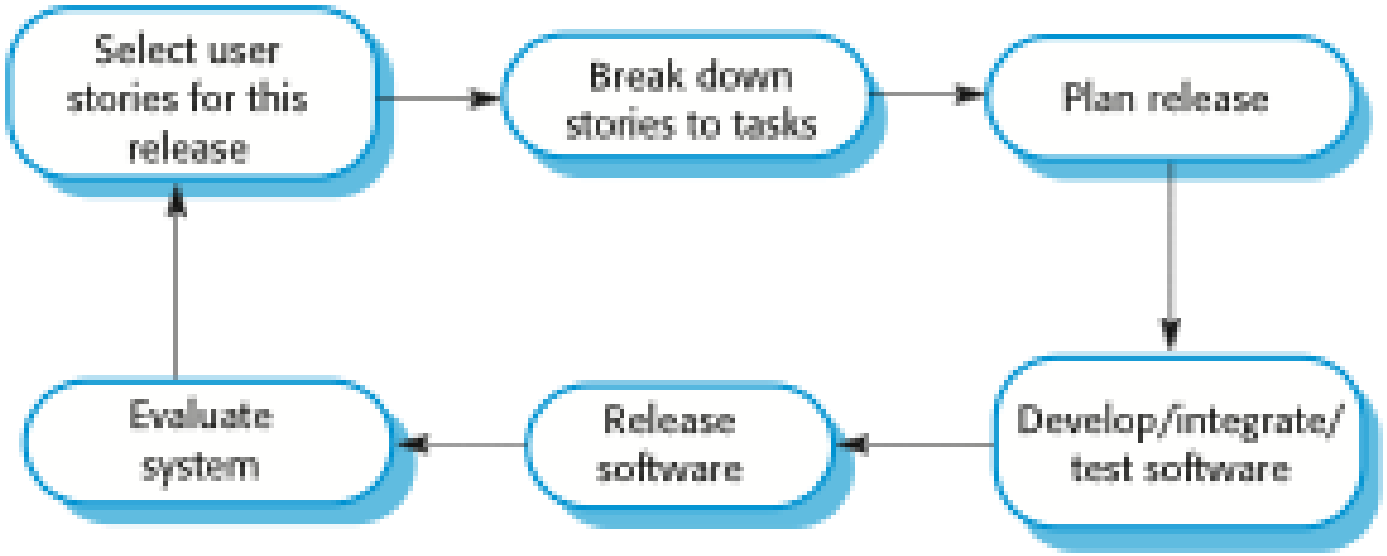
- XP is based on these
  - four values and
  - twelve practices
  - have been extended various ways since XP's introduction
- Extreme Programming (XP) takes an 'extreme' approach to iterative development.
  - New versions may be built several times per day;
  - Increments are delivered to customers approx. every 2 weeks;
  - **All** tests must be run for every build and the build is only accepted if tests run successfully.

XP and Agile Principles



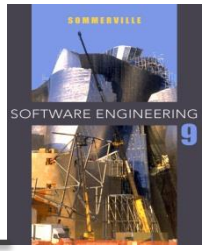
- **Incremental development** is supported through small, frequent system releases.
- Customer involvement means **full-time** customer engagement with the team.
- People not process through **pair programming**, **collective ownership** and a **process** that avoids long working hours.
- **Change** supported through regular system releases.
- Maintaining simplicity through constant **refactoring** of code.

The Extreme Programming Release Cycle



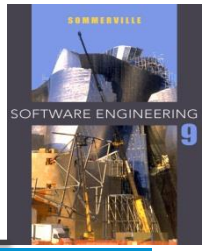
User Stories – coming…

Requirements Scenarios



- In XP, a customer or user is **part** of the XP team and is responsible for making decisions on requirements.
- User requirements are expressed as **scenarios (via use cases) or user stories.**
- These (User Stories) are often 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.

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.

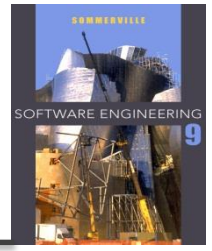
If you choose ‘formulary’, you will be presented with a search box for the approved formulary. Search for the drug required then select it. You will then be asked to check that the medication you have selected is correct. Enter the dose then confirm the prescription.

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.

This is a rather long user story

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.

Comments on XP

- XP: a strong track record for achieving **success in small applications**.

- **Scaling:** an issue w/20-person team often stated as absolute upper limit.
- XP: been tried **w/multiple teams**, but there's little experience this.

- XP is a **lightweight methodology**

- Challenges many conventional tenets,
  - including long held assumption: cost of **changing a piece of software necessarily rises dramatically over the course of time** and **YAGNI** (see next slide) are often inappropriate for stable systems with predictable evolutions.

- Beck identified several other **barriers** to using XP
  - teams not collocated,
  - long feedback cycles, and
  - long integration processes.

YAGNI – Essential to XP

- According to those who advocate the YAGNI approach, the temptation to write code that is not necessary at the moment, but might be in the future, has the following **disadvantages**:
- time spent taken from adding, testing, improving functionality.
  - New features must be debugged, documented, and supported.
  - Any new feature imposes **constraints** on what can be done in the future: an unnecessary feature **now** opens possibility of conflict with necessary feature later.

- Until feature is **actually needed**, it is difficult to **fully define** what it **should** do and to **test** it.
- If the new feature is **not properly defined and tested**, it may not work correctly, even if it eventually is needed.

- This leads to **code bloat**; software becomes larger and more complicated.
- Adding the new feature **may suggest other new features**.
  - If these new features are implemented as well, this may result in
    - **snowball effect towards feature creep**.

XP Fundamentals by Kent Beck

- Write unit tests **before** programming; keeping all tests running all times.
- Integrating and testing the whole system--**several times a day**.
- Producing all software in **pairs**, two programmers at one screen.
- Starting projects with **simple design.** Simple design can evolve.
- Putting a **minimal system** into production quickly and growing it in whatever directions prove most valuable.

XP Controversies

- **Why? Some sacred cows don't make the cut in XP:**

- **Don't force team members to specialize:** analysts, architects, programmers, testers, and integrators—
- every XP programmer **participates in all** of these critical activities every day.

- **Don't conduct complete up-front analysis and design—**
  - XP project starts with a **quick analysis** of the entire system, and
  - XP programmers continue to make analysis and design decisions throughout development.

- Develop infrastructure and frameworks **as you develop your application**, not up-front--**delivering business value** is **heartbeat** driving XP projects.

- **Don't write and maintain implementation documentation-** **c**ommunication in XP projects occurs **face-to-face**,
  - or through **efficient tests** and **carefully written code**

XP Core Practice #1- The Planning Game

- Business and development cooperate to produce **max business value** as quickly as possible.
- The planning game:
  - Business comes up with a list of desired **features**.
  - Each feature is written out as a **User Story**,
    - feature has a name, and is described in broad strokes what is required.
  - **User stories** are typically written on 4x6 cards. (You saw a variation in your book)
  - **Development estimates** how much effort each story will take, and **how much effort the team** can produce in a given time interval.
  - **Business then decides**
    - order of stories to implement,
    - And when and how often to produce a **production release** of the system.

XP – Core Practice #2: Simple Design

- Simplest possible design to get job done.
- Requirements will change tomorrow, do what's needed to meet today's requirements

- Design in XP is not a one-time; it is an "all-the-time" activity. Have design steps in
  - release planning
  - iteration planning,
  - teams engage in quick design sessions and design revisions through refactoring,
- through the course of the entire project.

XP – Core Practice #3: Metaphor

- Extreme Programming teams develop a common vision of how the program works, which we call the "metaphor".

- At its best, the **metaphor** is a simple evocative description of how the program works.

- XP teams use
- common system of **names** to be sure that everyone understands how the system works
- and **where to look to find the functionality** you're looking for,
- or to find the right place to **put the functionality** you're about to add.

Metaphor

Metaphor is something you start using when your mother asks what you are working on and you try to explain her the details. How you find it is very project-specific. Use your common sense or find the guy on your team who is good at explaining technical things to customers in a way that is easy to understand.

What XP suggests in my opinion are the following:

- Try to design a system that is easy to explain using real-life analogies. Your systems are complex, try to use a design, where the relationship and interactions between sub-components are clear and resemble something that people with common sense have already seen.

- Use the analogies in all communications: source-code, planning meetings, speaking to users, or God forsake, writing documentation. If you find that the concepts you use do not fit to some area, try to find a better metaphor. (Wiki)

XP – Core Practice #4: Simple Design

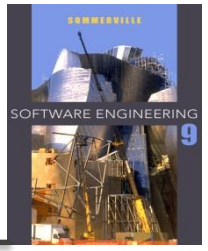
- Always use the simplest possible design that gets the job done.

- The requirements will change tomorrow, so only do what's needed to meet today's requirements.

XP – Core Practice #5: Continuous Testing

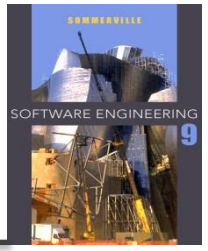
- XP teams focus on **validation** of the software at all times
- Programmers develop software by **writing tests first**, and **then code** that fulfills the requirements reflected in the tests.
- Customers provide **acceptance tests** that enable them to be **certain** that the features they need are provided.

Testing in XP



- 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

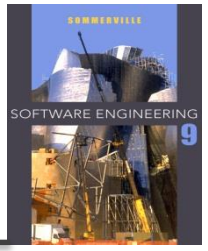


- Writing tests before code clarifies the requirements to be implemented.

- 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.

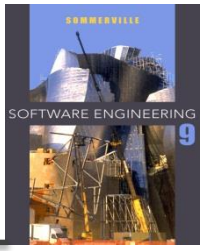
- All previous and new tests **are run automatically** when new functionality is added, thus checking that the new functionality has not introduced errors.

Customer Involvement



- 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** writes tests **as development proceeds**. All new code is **therefore** validated to ensure that it is what the customer needs.
- However, 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



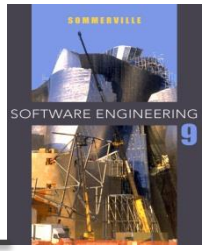
Test 4: Dose checking

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.

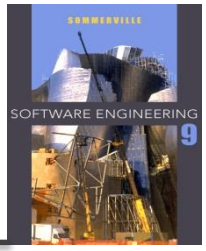
Output:
OK or error message indicating that the dose is outside the safe range.

Test Automation



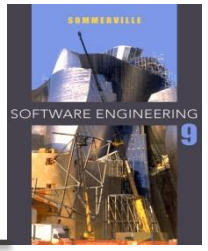
- **Test automation means that tests are written as executable components before the task is implemented**
  - These testing components should be stand-alone, should simulate the submission of input to be tested and should check that the result meets the output specification. An automated test framework (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.

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 **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.

XP and Change

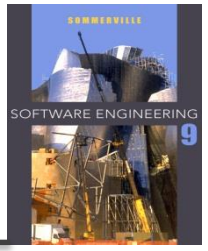


- **Conventional wisdom in software engineering is to design for change**.
- It is worth spending time and effort anticipating changes as this reduces costs later in the life cycle.
- **XP, however, maintains that this is not worthwhile** as changes cannot be reliably anticipated.
- **Rather**, it proposes constant code improvement (refactoring) to make changes easier when they have to be implemented.

XP – Core Practice #6: Refactoring

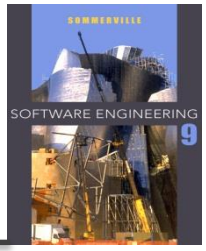
- XP Team **Refactor** out any duplicate code generated in a coding session.
- Refactoring is simplified due to extensive use of **automated** test cases.

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

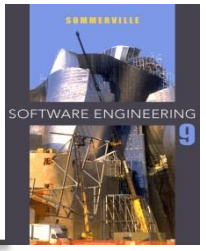


- Re-organization of a class hierarchy to remove duplicate code.
- Tidying up and renaming attributes and methods to make them easier to understand.
- The replacement of inline code with calls to methods that have been included in a program library.

XP – Core Practice #7: Pair Programming

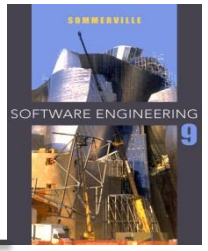
- All production code is written by two programmers sitting at one machine.
  - This practice ensures that all code is reviewed as it is written and results in better Design, testing and better code.
- Some programmers object to pair programming without ever trying it.
  - It does take some practice to do well, and you need to do it well for a few weeks to see the results.
  - Ninety percent of programmers who learn pair programming prefer it, so it is recommended to all teams.
- Pairing, in addition to providing better code and tests, also serves to **communicate knowledge** throughout the team.

Pair Programming



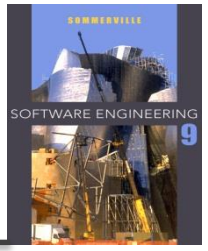
- In XP, programmers work in pairs, sitting together to develop code.
- This helps develop common ownership of code and spreads knowledge across the team.
- It serves as an informal review process as each line of code is looked at by more than 1 person.
- It encourages refactoring as the whole team can benefit from this.
- 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.
- 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.

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.
- 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.

XP – Core Practice #8: Collective Code Ownership

- No single person "owns" a module.
- Any developer is expected to be able to work on any part of the codebase at any time.

XP – Core Practice #9: Continuous Integration

- All changes are integrated into the codebase at least daily.
- Unit tests have to run 100% both before and after integration.
  - Infrequent integration leads to serious problems on a project.

- Although integration is critical to shipping good working code, the team is not practiced at it, and often it is delegated to people not familiar with the whole system.

- Problems creep in at integration time that are not detected by any of the testing that takes place on an un-integrated system.

- **Code freezes** mean that you have long time periods when the programmers could be working on important shippable features, but that those features must be held back.

XP – Core Practice #10: 40-hour Week

- Programmers go home on time.
  - In crunch mode, up to one week of overtime is allowed.
- Multiple consecutive weeks of overtime are treated as a sign that something is very wrong with the process and/or schedule.

XP – Core Practice #11: On-Site Customer

- Development team has **continuous access** to the customer who will actually be using the system.

- For initiatives with lots of customers, a **customer representative** (i.e. Product Manager) will be designated for Development team access.

XP – Core Practice #12: Coding Standards

- Everyone codes to the same standards.
- The specifics of the standard are not important: what is important is that **all** of the code looks familiar, in support of **collective ownership**.

XP on your own – Supplemental.

XP Values – Summarized.

- XP is a values-based methodology. The **values** are Simplicity, Communication, Feedback and Courage.

- XP's core values:best summarized in the following statement by Kent Beck: **Do more of what works and do less of what doesn't**.

Highlights of the four values itemized:

- **Simplicity** encourages:
  - Delivering the simplest functionality that meets business needs
  - Designing the simplest software that supports the needed functionality
  - Building for today and not for tomorrow
  - Writing code that is easy to read, understand, maintain and modify

Highlights of the four values itemized:

- **Communication** is accomplished by:
  - Collaborative workspaces
  - Co-location of development and business space
  - Paired development
  - Frequently changing pair partners
  - Frequently changing assignments
  - Public status displays
  - Short standup meetings
  - Unit tests, demos and oral communication, not documentation

Highlights of the four values itemized:

- **Feedback** is provided by:
  - Aggressive iterative and incremental releases
  - Frequent releases to end users
  - Co-location with end users
  - Automated unit tests
  - Automated functional tests
  - Courage is required to:
    - Do the right thing in the face of opposition
    - Do the practices required to succeed

Conclusion

- Extreme Programming is not a complete template for the entire delivery organization.
- Rather, XP is a set of best practices for managing the development team and its interface to the customer.

- As a process it gives the team the ability to grow, change and adapt as they encounter different applications and business needs.
- And more than any other process we have encountered Extreme Programming has the power to **transform the entire delivery organization**, not just the development team.

Extreme Programming Overview

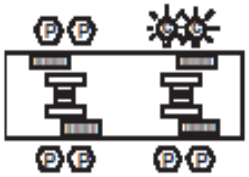
team interaction

Team Practices

- Whole team sits together in one room
- Work at a sustainable pace
- Integrate many times per day
- Share a common vision and vocabulary
- Reflect regularly
- Converge on a coding standard

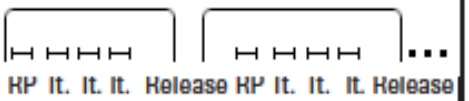
An XP Room

Dynamic pairs write all production code



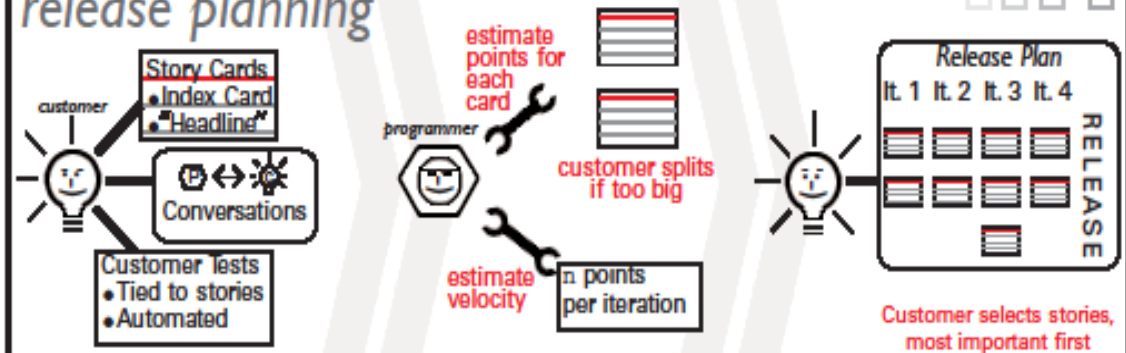
Any pair can change any code

overall schedule



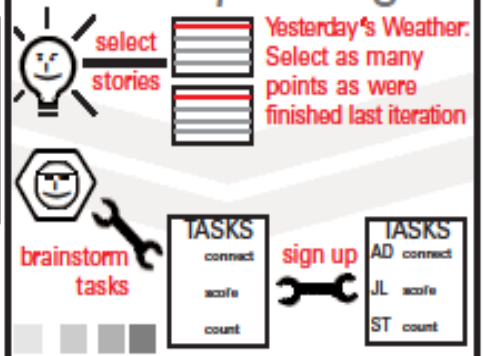
RP=Release Planning (1-3 weeks)
It.= Iteration (fixed length, 1-3 weeks)
Release to users every 1-3 months

release planning



Customer selects stories, most important first

iteration planning

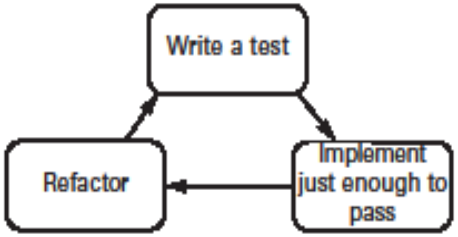


programming

Design Philosophy

- Design is evolutionary and emergent
- Pay as you go: Build just enough to meet today's requirements
- Keep design as simple as possible (but no simpler)
- High quality is both a side effect and an enabling factor
- The code says everything "once and only once"

Incremental Test-First Programming



Cycle takes 5-15 minutes

Refactoring
Stepwise design improvement via safe transformations

Example: Move Method

William.Wake@acm.org http://www.xp123.com