Based on presentation by Nurit Gal-oz, Department of Computer Science Ben-Gurion University
Based on presentation by Mira Balaban Department of Computer Science Ben-Gurion university
E. M. Burke and B.M. Coyner: Java Extreme Programming Cookbook.
L. Crispin and T. House: Testing Extreme Programming
http://www.extremeprogramming.org
And slides of: Kent Beck and Ward Cunningham,
Laurie Williams, Vera Peeters and Pascal Van Cauwenberghe,
Ian Sommerville.
http://www.comp.lancs.ac.uk/computing/resources/IanS/SE7/Presentations/index.html
Extreme Programming (XP)
## Principles of agile methods

<table>
<thead>
<tr>
<th>Principle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer involvement</td>
<td>The customer should be closely involved throughout the development process. Their role is provide and prioritise new system requirements and to evaluate the iterations of the system.</td>
</tr>
<tr>
<td>Incremental delivery</td>
<td>The software is developed in increments with the customer specifying the requirements to be included in each increment.</td>
</tr>
<tr>
<td>People not process</td>
<td>The skills of the development team should be recognised and exploited. The team should be left to develop their own ways of working without prescriptive processes.</td>
</tr>
<tr>
<td>Embrace change</td>
<td>Expect the system requirements to change and design the system so that it can accommodate these changes.</td>
</tr>
<tr>
<td>Maintain simplicity</td>
<td>Focus on simplicity in both the software being developed and in the development process used. Wherever possible, actively work to eliminate complexity from the system.</td>
</tr>
</tbody>
</table>
From Waterfall to Extreme Programming
Extreme Programming

A system of practices that a community of software developers is evolving to address the problems of quickly delivering quality software, and then evolving it to meet changing business needs.

Kent Beck & Martin Fowler, Planning extreme programming, 2000
Motivation

Why do we need XP?

Common problems of software development:

- Schedule slips
- Business misunderstood
- Defect rate
- Management
- Motivation of developers

XP solutions:

- Short iterations, fast delivery
- Whole team
- Test driven development
- Shared understanding
- Humanity and productivity
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;
  - All tests must be run for every build and the build is only accepted if tests run successfully.
Extreme?

Source: Joint Advanced Student School (JASS) 2006: Extreme Programming
Some eXtreme Practices

- **Is testing good?**
  - Then write the **unit test before the code, automate testing, and run all tests all the time**. (Coding)

- **Are code inspections good?**
  - Then **code in pairs**—all code is inspected all the time! (Team)

- **Is customer contact good?**
  - Then **locate a customer representative in the team**, so that you have access to them all the time. (Processes, Costumer)

- **If design is good**
  - refactor all the time

- **If integration testing is good,**
  - integrate all the time

- **If simplicity is good,**
  - do the simplest thing that could possibly work

- **If short iterations are good,**
  - make them really, really short
Next Step: XP

- Extreme Programming takes the idea of incremental development to the next level.

**Plan, analyze, and design a little at a time.**

- XP is a software development “culture”.
- It is based on 4 basic values (“commandments”) that are achieved by employing 12 (or 13) simple rules (practices).
Purpose of XP – Reduce the cost of change

- “Under certain circumstances, the exponential rise in the cost of changing software over time can be flattened. If we can flatten the curve, old assumptions about the best way to develop software no longer hold”.
  (kent Beck)
Purpose of XP – Reduce the cost of change:
The cost of change may not rise dramatically over time

This is the technical premise of XP.
- If the cost of change rose slowly over time, you would act completely differently from how you do under the assumption that costs rise exponentially.
  - You would make big decisions as late in the process as possible, to defer the cost of making the decisions and to have the greatest possible chance that they would be right.
  - You would only implement what you had to, in hopes that the needs you anticipate for tomorrow wouldn't come true.
  - You would introduce elements to the design only as they simplified existing code or made writing the next bit of code simpler.

- If change is ruinously expensive, you would be crazy to charge ahead without careful forethought.
- But if change stays cheap, the additional value and reduced risk of early concrete feedback outweighs the additional cost of early change.
- Keeping the cost of change low doesn't just happen magically. -There are technologies and practices that keep software pliable.

(Kent Beck)
What is Extreme programming

- Extreme programming is a discipline of software development based on values of

  *simplicity, communication, feedback, courage.*

- It works by bringing the **whole team** together in the presence of **simple practices**, with enough **feedback** to tune practices to their **unique situation**.
  - **Whole team**: the best teams have no **specialists** only general contributors with special skills.

- **Planning**: XP addresses two key questions in software development:
  - **predicting** what will be accomplished by the due date, and
  - **determining what to do next**.
The 4 basic values of XP

- We will be successful when we have a style that celebrates a consistent set of values that serve both human and commercial needs:

  Communication

  Courage

  Feedback

  Simplicity

  Kent Beck
The 4 basic values of XP

- **Communication**: is a two way thing. It’s about *talking* and *listening*
  - Use person to person communication instead of written documents where possible.
- **Simplicity**: is about simple solutions that do what’s required, no more, no less.
  - Eliminate unnecessary elements of building software.
- **Feedback**: has three axes: *Giving feedback*, *receiving feedback* and *taking action* as a result of the feedback
  - Use constant software testing as a major source of quality feedback.
- **Courage**:
  - It’s about taking *calculated risks*.
  - It’s about facing and voicing the brutal facts.
  - It’s also about creating an environment where people can be courageous.
  - Possible when you take the simplest possible approach and employ a process high in communication.
The 12 Key Practices

- The Planning Game
- Small Releases
- Metaphor
- Simple Design
- Test-Driven Development
- Refactoring
- Pair Programming
- Collective Ownership
- Continuous Integration
- 40-Hour Workweek
- On-site Customer
- Coding Standards
The Rules (Practices)

Communication Rules
Simplicity Rules
Feedback Rules
Courage Rules
Communication Rules (practices)

- Onsite customer.
- Pair programming.
- Coding standards.
Communication Rules

1. On Site Customer

- At least one customer is always present.

- This customer is available full-time to:
  - Answer questions about the system.
  - Negotiate the timing and scheduling of releases.
  - Make all decisions that affect business goals.

- The customer writes functional tests (with the help of Development).
Communication Rules

2. Pair Programming

Source: http://www.carlandsteve.com/?p=30
Communication Rules
2. Pair Programming

- All programming is done with **two coders at the same machine**.
- The **programmers must share** one mouse, keyboard, screen, etc.

⇒ At least two people are always intimately familiar with every part of the system, and every line of code is reviewed as it's written.
Communication Rules
Here is how pair programming works:

• You pick out a user story for your next task.

  *A user story is a requirement from the customer.*
  *Stories are typically written on index cards, and the customer decides which stories are the most important.*

• You ask for help from another programmer.

• The two of you work together on a small piece of functionality.

  • Try to work on small tasks that take a few hours.
  • After the immediate task is complete, pick a different partner or offer to help someone else.

  Eric Burke and Brian Coyner
Pair Programming
http://www.pairprogramming.com/

With pair-programming:

• Two software engineers work on one task at one computer
• One engineer, the driver, has control of the keyboard and mouse and creates the implementation
• The other engineer, the navigator, watches the driver’s implementation to identify defects and participates in on-demand brainstorming
• The roles of driver and observer are periodically rotated between the two software engineers
• Shortcomings?

Pair programming to please your boss is just frustrating.
Pair programming to communicate, get feedback, simplify the system, catch errors, and bolster your courage makes a lot of sense. (Kent Beck)
Pair programming
Ways to Improve Your Pairing Experience

- Do not centralize driving
- Avoid working alone
- Alternate moments of concentration and relaxation
- Celebrate your achievements!
- Learn to deal with disagreements

https://www.thoughtworks.com/insights/blog/ecneirepxe-gniriap-ruoy-evorpmi-syaw-10
Communication Rules
Research Findings to Date

• Strong anecdotal evidence from industry
  • “We can produce near defect-free code in less than half the time.”

• Empirical Study
  • Pairs produced higher quality code
    • 15% less defects (difference statistically significant)
  • Pairs completed their tasks in about half the time
    • 58% of elapsed time (difference not statistically significant)
  • Most programmers reluctantly embark on pair programming
    • Pairs enjoy their work more (92%)
    • Pairs feel more confident in their work products (96%)

• India Technology Company
  • 24% increase in productivity (KLOC/Person-Month)
  • 10-fold reduction in defects.
Communication Rules
3. Coding Standards

• Agree upon standards for coding styles.
• Promotes ease of understanding and uniformity.
• No idiosyncratic quirks that could complicate understanding and refactoring by the entire team.

• Do it My Way or Get Out.. ?
  • No - The XP rules are rules that the Team chose to live by, and the Team says "Do it OUR way, or get out".
Simplicity Rules (practices)

- Metaphor.
- Simple design.
- Refactoring.
4. Metaphor

• Dictionary definition: “Metaphor: a figure of speech which makes an *implied* comparison between things which are not *literally* alike.”

• XP definition: “The system metaphor is a story that everyone--customers, programmers, and managers--can tell about how the system works.” -- Kent Beck, *Extreme Programming Explained*
4. Metaphor

**Metaphor:** A story that everyone-customers, programmers, and managers - can tell about how the system works

- Use metaphors to **describe how the system should work**.
- These **analogies** express the functionality of the system.
- Provides a simple way to remember naming conventions.
- The metaphor in XP replaces much of what other people call "architecture."
- The problem with calling the 10,000-meter view of the system an architecture is that architectures don't necessarily push the system into any sense of cohesion.

- [http://www.xpexchange.net/english/intro/metaphor.html](http://www.xpexchange.net/english/intro/metaphor.html)
Why metaphors work in XP

• You couldn't possibly start development with just a metaphor. There isn't enough detail there, and besides, what if you're wrong? Unless:
  • You quickly have concrete feedback from real code and tests about whether the metaphor is working in practice.
  • Your customer is comfortable talking about the system in terms of the metaphor.
  • You refactor to continually refine your understanding of what the metaphor means in practice.

• Then perhaps you could start development with just a metaphor.
Why Do You Want a Metaphor?

- Shared vision
- Generativity
- Vocabulary and Naming
- Architecture
5. Simple Design

- The code should **pass all tests** and fulfill certain functionality while maintaining:
  - Best communicate the intention (cohesion).
  - No duplicate code.
  - Fewest possible classes and methods.
  - “Say everything once and only once.”

- XP developers:
  - Do the simplest thing that could possibly work:
  - Never solve a more general problem than the specific problem at hand.
  - Never add functionality sooner than needed.
6. Refactoring

- The code may be changed at any time to provide:
  - Simplification.
  - Flexibility.
  - Reduced redundancy.

- Automated unit tests are used to verify every change.
- There is no "canonical form" for code after which point refactoring is completely unnecessary.
- Code is refactored until it satisfies the **teams notions of good XP code.**
What is refactoring

- **Refactoring** is the practice of:
  Improving the design of code without breaking its functionality.
- Simplicity requires **constant refactoring**: small changes.

Goals:

- **Clarity**: the meaning of the code should be obvious to anyone reading it.
- **Eliminate redundant code** - Similar and redundant classes and code blocks should be combined, and unnecessary functions and variables eliminated.
- **Simplicity**: Complexity should be added only if absolutely necessary to improve readability or modularity.
Refactoring example: Eliminate redundant comments

```java
/** *
 * Sets the value of x.
 * @param x the horizontal position in pixels.
 */
public void setX(int x) {
    this.x = x;
}
```

After renaming:
```
public void setXPixelPosition(int xPixelPosition) {
    this.xPixelPosition = xPixelPosition;
}
```

→ No need for comments.

But: Requires changing all references to the method throughout the application.
Refactoring example: Rename variable

```java
public class Person {
    private String firstName;
    public void setFirst(String n) {
        this.firstName = n;
    }
}

Rename variable:
public class Person {
    private String firstName;
    public void setFirst(String firstName) {
        this.firstName = firstName;
    }
}
```
Refactoring example: Rename method

```java
public class Person {
    private String firstName;
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
}
```

- The method has been refactored and is now more easily understandable.
- Changing the method name requires you to change all references to the method throughout your application.
- This is where a good IDE can help out, because it can identify all usages and update the calls automatically.
When to refactor?

- **Refactor constantly**, throughout the lifetime of a project.

- Each time you fix a bug or add a new feature, look for overly complex code. Look for:
  - Chunks of logic that are duplicated and refactor them into a shared method.
  - Try to rename methods and arguments so they make sense.
  - Try to migrate poorly designed code towards better usage of design patterns.

- Writing unit tests is a great way to identify portions of code that need refactoring.
- When you write tests for a class, your test is a client of that class.
How to refactor?

1. Make sure you have a working unit test for the feature you are about to refactor.
2. Do the refactoring, or a portion of the refactoring.
3. Run the test again to ensure you did not break anything.
4. Repeat steps 2-4 until you are finished with the refactoring.

Refactoring...
Feedback Rules (practices)

- Testing.
- Continuous integration.
- Small releases.
7. Testing

• Tests are continuously written with the system.
• All tests are run together at every step.
• Customers write tests that will convince them the system works.
• Don’t proceed until current system passes ALL tests.
Testing

- Every piece of code has a set of automated unit tests, which are released into the code repository along with the code.

- The programmers write the unit tests before they write the code, then add unit tests whenever one is found to be missing.

- No modification or refactoring of code is complete until 100% of the unit tests have run successfully.

- Acceptance tests validate larger blocks of system functionality, such as user stories.

When all the acceptance tests pass for a given user story, that story is considered complete.
Unit tests

- A unit test is a programmer-written test for a single piece of functionality in an application.

- Unit tests should be fine grained, testing small numbers of closely-related methods and classes.

- Unit tests should not test high-level application functionality.

- Testing application functionality is called acceptance testing, and acceptance tests should be designed by: people who understand the business problem better than the programmers.
Testing new features – the test-driven process (1)

1. Run the suite of unit tests for the entire project, ensuring that they all pass.

2. Write a unit test for the new feature.

3. Run the test and observe its failure.

4. Implement the new feature.

5. Run the test again and observe its success.
Testing new features – the test-driven process (2)

Think of another test, and follow this process:

1. Write another test for some aspect of the new function that might break, such as an illegal method argument.
2. Run all of your tests.
3. Fix the code if necessary, and repeat until you cannot think of any more tests.
4. Once your new feature is fully tested, it is time to run the entire suite of unit tests for the entire project.
5. **Regression testing** ensures that your new code did not inadvertently break someone else's code.
6. If some other test fails, you immediately know that you just broke it. You must fix all of the tests before you can commit your changes to the repository.
How to write tests?

All tests must be pass/fail style tests.

A bad test output:
Now Testing Person.java:
First Name: Tanner
Last Name: Burke
Age: 1

A good test output:
Now Testing Person.java:

  Failure: Expected Age 2, but was 1 instead.
Writing tests

Grouping tests into test suites:

Now Testing Person.java:
  Failure: Expected Age 2, but was 1 instead
Now Testing Account.java:
  Passed!
Now Testing Deposit.java:
  Passed!

Summary: 2 tests passed, 1 failed.

- The entire suite of unit tests must always pass at 100% before any code is integrated into the source repository.
- Acceptance tests do not have to pass at 100%.
8. Continuous Integration

- Newly finished code is **integrated immediately**. Unit tests must run 100% successfully, both before and after each integration.

- System is **rebuilt from scratch** for every addition.

- New system must **pass all tests** or new code is discarded.

- Additions and modifications to the code are integrated into the system on at least a daily basis.
9. Small Releases

- A **functional system is produced** after a few months.
- System is released **before the whole problem is solved**.
- New **releases regularly** (daily to monthly).
- The **smallest useful feature set** is identified for the first release.
- Releases are performed as early and often as possible.
- Each release:
  - a few new features added each time.
Courage Rules (practices)

- The planning game.
- Collective code ownership.
- Sustainable pace.
10. The Planning Game

- **Schedule small tasks** to be completed during the current completed **iteration**.
- Programmers focus their attention on the tasks at hand.
- List of tasks is updated regularly.
More info: http://www.xp.be/
11. Collective Ownership

• All workers can access any of the code.
• Any programmer can change any part of the system if an opportunity for improvement exists.
• The TEAM makes the product.
• It works…
  … in disciplined XP teams.
• Requires Ego-less team members.
12. Sustainable pace: 40 Hour Weeks

- Consecutive weeks of overtime is not allowed.
- The need for overtime is a symptom of a deeper problem.
Team
13. Open Workspace

- Work on computers set up in the **middle of a large room** with cubicles around the edges.
- Question: With how many people do you want to work in one room?
14. Daily Standup Meeting

• Stand up to keep it short.
• Developers report:
  • what was accomplished yesterday
  • what will be attempted today
  • what problems are causing delays.
• Everybody
  • Agrees what they will work on
  • Raises problems & difficulties
  • Knows what’s going on
• Initial pairing.
More info: http://www.xp.be/
More info: http://www.xp.be/
Just Rules

• These rules are just rules.
• XP teammates agree to follow all of the rules.
• An agreement can be made to change the rules.
  – Must address side effects of rule change.
XP Practices Loops

- **Release loop:** Small releases, planning game, customer acceptance tests, onsite customer.

- **Team loop:** Collective code ownership, metaphor, sustainable pace, coding standards, open space, daily stand-up meetings..

- **Code loop:** Test driven development, refactoring, simple design, pair programming, continuous integration.
XP Planning/Feedback times

Planning/Feedback Loops

- Release Plan
  - Months
- Iteration Plan
  - Weeks
- Acceptance Test
  - Days
- Stand Up Meeting
  - One Day
- Pair Negotiation
  - Hours
- Unit Test
  - Minutes
- Pair Programming
  - Seconds
- Code

Copyright 2001 J. Donovan Wells
The Game:

XP: How it works
Why Plan?

- We plan because
  - We need to ensure that we are always **working** on the **most important thing** we need to do.
  - We need to **coordinate** with other people.
  - When unexpected events occur we need to **understand the consequences** for the first two.
Customer-Developer Relationships

*A well-known experience in Software Development:*
The customer and the developer sit in a small boat in the ocean and *are afraid of each other.*

<table>
<thead>
<tr>
<th>Customer fears</th>
<th>Developer fears</th>
</tr>
</thead>
<tbody>
<tr>
<td>They won't get what they asked for</td>
<td>They won't be given clear definitions of what needs to be done</td>
</tr>
<tr>
<td>They must surrender the control of their careers to techies who don't care</td>
<td>They will be given responsibility without authority</td>
</tr>
<tr>
<td>They'll pay too much for too little</td>
<td>They will be told to do things that don't make sense</td>
</tr>
<tr>
<td>They won't know what is going on (the plans they see will be fairy tales)</td>
<td>They'll have to sacrifice quality for deadlines</td>
</tr>
</tbody>
</table>

**Result:** A lot of energy goes into protective measures and politics instead of success

**Source:** Prof. O. Nierstrasz, *The Planning Game*  
© Oscar Nierstrasz
# The Customer Bill of Rights

<table>
<thead>
<tr>
<th>You have the right to an overall plan</th>
<th>To steer a project, you need to know what can be accomplished within time and budget</th>
</tr>
</thead>
<tbody>
<tr>
<td>You have the right to get the most possible value out of every programming week</td>
<td>The most valuable things are worked on first.</td>
</tr>
<tr>
<td>You have the right to see progress in a running system.</td>
<td>Only a running system can give exact information about project state</td>
</tr>
<tr>
<td>You have the right to change your mind, to substitute functionality and to change priorities without exorbitant costs.</td>
<td>Market and business requirements change. We have to allow change.</td>
</tr>
<tr>
<td>You have the right to be informed about schedule changes, in time to choose how to reduce the scope to restore the original date.</td>
<td>XP works to be sure everyone knows just what is really happening.</td>
</tr>
</tbody>
</table>

**Source:** Prof. O. Nierstrasz, *The Planning Game* © Oscar Nierstrasz
# The Developer Bill of Rights

<table>
<thead>
<tr>
<th>Rights</th>
<th>Rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>You have the right to know what is needed, with clear declarations of</td>
<td>Tight communication with the customer. Customer directs by value.</td>
</tr>
<tr>
<td>priority.</td>
<td></td>
</tr>
<tr>
<td>You have the right to produce quality work all the time.</td>
<td>Unit Tests and Refactoring help to keep the code clean</td>
</tr>
<tr>
<td>You have the right to ask for and receive help from peers, managers,</td>
<td>No one can ever refuse help to a team member</td>
</tr>
<tr>
<td>and customers</td>
<td></td>
</tr>
<tr>
<td>You have the right to make and update your own estimates.</td>
<td>Programmers know best how long it is going to take them</td>
</tr>
<tr>
<td>You have the right to accept your responsibilities instead having them</td>
<td>We work most effectively when we have accepted our responsibilities</td>
</tr>
<tr>
<td>assigned to you</td>
<td>instead of having them thrust upon us</td>
</tr>
</tbody>
</table>

**Source:** Prof. O. Nierstrasz, *The Planning Game*

© Oscar Nierstrasz
The Planning Game Rationale

- Planning is an emotional minefield.
  - **Development** would like to program faster
  - **Project manager** would like to be able to say exactly how fast **Development** can go.
  - **Business** would like to be able to say exactly what they want
  - **Business** would rather not change its mind.
  - When any of the participants in planning begin acting these wishes (or rather in accordance with the fears that lie behind each wish), then planning doesn't work well.
- **The Planning Game:** Create a little emotional distance from planning by treating it as a game (hence the name). The game has a **goal**, **playing pieces**, **players**, and **rules** for allowable moves.
The Planning Game (1)

• **Pieces:**
  - The basic playing piece is the **UserStory**.
  - Each Story is written on an **index card**.
  - Stories have a **value** and a **cost**
    - although this is a little tricky because the value of some Stories depends on the presence or absence of other Stories and the values and costs change over time

• **Goal:** The goal of the game is to **put the greatest possible value of stories into production** over the life of the game.

• **Players:** The players are **Business** and **Development**.

• **Moves:**
  - **Write Story:** **Business** can write a new Story at any time.
    - For purpose of the Planning Game, writing a Story just means assigning it a value (in practice, it has to have enough information for **Development** to assign it a cost).
The Planning Game (2)

Moves:

- *Write Story…*
- *Estimate Story:*
  - **Development** Takes every story and assigns it a cost of 1, 2, or 3 weeks of **IdealProgrammingTime**
    - Ideal time is time without interruption where you can concentrate on your work and you feel fully productive
  - If the estimate is higher, **Business** splits the story.
    - This may result in the story being implemented over more than one Iteration
  - If the estimate is lower, **Business** merges it with another story.
  - Measure the **velocity** of the team and estimate the amount of effort required for each story
    - **Velocity of the team** = # of ideal weeks per iteration
The Planning Game (3)

**Moves:**

- *Write Story*...
- *Estimate Story* ...
- *Make Commitment*: **Business** and **Development** decide together
  - *what* stories constitute the next release and
  - *when* it will be ready to put into production.
  - There are two ways to drive the **commitment**,
    - **Story Driven**
    - **Date Driven**.
The Planning Game (3)

- **Story Driven Commitment:**
  - **Business** starts putting the Stories for the next release on the table.
  - As each Story is introduced, **Development** calculates and announces the release date.
  - This move stops when **Business** is satisfied that the Stories on the table make sense as the next release.

- **Date Driven Commitment:**
  - **Business** picks a release date.
  - **Development** calculates and announces the cumulative cost of Stories they can accomplish between now and the date.
  - **Business** picks Stories whose cost adds up to that number.
The Planning Game (4): Ordering

- **Value and Risk First**: Development orders the Stories in a commitment so:
  1. **BusinessValueFirst**: More valuable Stories are moved earlier in the schedule
  2. **WorstThingsFirst**: Riskier Stories are moved earlier in the schedule
Overcommitment Recovery

- **Development** had predicted they could do 150 units of stories between now and the deadline.
- Based on measuring **Project Velocity**, they find and immediately announce that they can only do 100.
- **Business** selects the 100 units of Stories to retain, deferring the other Stories to a future release. (Or highly unlikely: **Business** decides to defer the deadline to get the extra 50 units done.)
The Planning Game (5)

- **Change Value:**
  - **Business** changes the value of a Story.
  - In response, **Development** may change the order of Stories not yet completed.

- **Introduce New Story:**
  - **Business** writes a new Story.
  - **Development** estimates it.
  - **Business** defers Stories in the current Commitment whose cumulative cost is the cost of the new Story.
  - **Development** re-evaluates Value and Risk First.
The Planning Game (6)

- **Split Story:**
  - **Business** splits a Story into two or more.
  - **Business** assigns a value to each, and
  - **Development** assigns a cost to each.
  - Typically this is done because resources do not permit the whole story to be done soon enough.

- **Spike:**
  - **Business** can divert Project resources to do a throwaway Spike to fight a fire or prove a concept.
  - If this Spike is anything more than a temporary fix, **Business** makes a User Story to account for it.
  - That Story is scheduled according to **Value And Risk First**.
  - Regular spikes, especially fire-fighting ones, will affect the Load Factor.

- **Re-estimate:** **Development** estimates the remaining stories in the Commitment again.
A Story Card

Customer Story and Task Card

DATE: 3/19/91

TYPE OF ACTIVITY: NEW: X  FIX:  ___  ENHANCE:  ___  FUNC. TEST

STORY NUMBER: 1275

PRIORITY: USER:  ___  TECH:  ___

PRIOR REFERENCE:  ___  

RISK:  ___  TECH ESTIMATE:

TASK DESCRIPTION:

SPLIT COLA: When the COLA rate chaps in the middle of the BLW Pay Period, we will want to pay the 1st week of the pay period at the OLD COLA rate and the 2nd week of the Pay Period at the NEW COLA rate. Should occur automatically, but

NOTES:

For the OT, we’ll run a Milframe program that will pay or calc the COLA on the 2nd week of OT. The plant currently retroactively hours data for the 2nd week exclusively so that we can calc COLA. This will come into the Model as a “2144-COLA”

TASK TRACKING:

Date  Status  To Do  Comments

Date  Status  To Do  Comments

Date  Status  To Do  Comments

Date  Status  To Do  Comments
Story Cards for a Coffee Maker

Brew some coffee. When the brew button is pressed boil the water until empty.

Indicator light. Turn on the indicator light when the coffee is done brewing. Turn off the indicator light the first time the coffeepot is picked up.

Keep the coffee warm. When the pot has coffee in it turn on the warmer. When the coffeepot is empty turn off the warmer. When the coffeepot is removed turn off the warmer.

Interrupt brewing if the coffeepot is removed. Opening the relief valve will stop the water flow. If the coffeepot is replaced continue.
# Story card for document downloading

## Downloading and printing an article

First, you select the article that you want from a displayed list. You then have to tell the system how you will pay for it - this can either be through a subscription, through a company account or by credit card.

After this, you get a copyright form from the system to fill in and, when you have submitted this, the article you want is downloaded onto your computer.

You then choose a printer and a copy of the article is printed. You tell the system if printing has been successful.

If the article is a print-only article, you cannot keep the PDF version so it is automatically deleted from your computer.
The Planning Game

A game with a set of rules that ensures that Customer and Developers don’t become mortal enemies

Goal:

- Maximize the value of the software produced by Developers.

Overview:

1. Release Planning
2. Iteration Planning
Release Planning
Release Planning

- Release planning meeting
- Release plan
  - Lays out overall project
  - specifies which user stories are going to be implemented for each system release (and dates for those releases)
  - release plan used to create iteration plans
- In release planning the customer chooses a few months' worth of stories, typically focusing on a public release
- A release round includes 3 phases:
  - Exploration phase
  - Commitment Phase
  - Steering Phase
Release Planning

- **Exploration phase:**
  - **Goal**: Next release planned that maximizes value/effort
  - **Result**: list of stories (and tasks) to be included in next release
  - **Moves**: Write a story, estimate a story, split a story.

- **Commitment phase:**
  - **Goal**: Customer sorts stories by value;
    Programmers sort stories by risk.
  - **Moves**: sort by value, sort by risk, set velocity, choose scope.

- **Steering phase:**
  - **Goal**: Update the plan.
  - **Moves**: iteration, recovery, new story, re-estimate.
Planning Game: Exploration Phase

**Purpose:**

Get an appreciation for what the system should eventually do.

**The Moves:**

- **Customer:** *Write a story*. Discuss it until everybody understands it.
- **Developers:** *Estimate a story* in terms of effort.
- **Customer:** *Split a story*, if Developers don’t understand or can’t estimate it.
- **Developers:** Do a *spike solution* to enable estimation.
- **Customer:** *Toss stories* that are no longer wanted or are covered by a split story.

*Source: Prof. O. Nierstrasz, The Planning Game*  
© Oscar Nierstrasz
User Stories

*Principles of good stories:*

- *Customers* write stories.
  - *Developers* do *not* write stories.
- Stories must be *understandable* to the customer
- The *shorter* the better. No detailed specification!
  - Write stories on *index cards*
- Each story must provide *something of value* to the customer
- A story must be *testable*
  - then we can know *when it is done*

*Writing stories is an iterative process, requiring interaction between Customer and Developers.*

Source: Prof. O. Nierstrasz, *The Planning Game*  
© Oscar Nierstrasz
Stories

A story contains:

- a name
- the story itself
- an estimate

Example:

- When the GPS has contact with two or fewer satellites for more than 60 seconds, it should display the message “Poor satellite contact”, and wait for confirmation from the user. If contact improves before confirmation, clear the message automatically.

Source: Prof. O. Nierstrasz, The Planning Game
© Oscar Nierstrasz
Splitting Stories

*Developers ask the Customer to split a story if*

- They cannot estimate a story because of its complexity
- Their estimate is longer than two or three weeks of effort

*Why?*

- Estimates get fuzzy for bigger stories
- The smaller the story, the better the control (tight feedback loop)

*Source:* Prof. O. Nierstrasz, *The Planning Game*
© Oscar Nierstrasz
Planning Game: Commitment Phase

Purpose:

• **Customer:** to choose scope and date of next delivery
• **Developers:** to confidently commit to deliver the next release

The Moves:

• **Customer:** Sort by stories by **value**
  1. Stories without which the system will not function
  2. Less essential stories, but still providing significant business value
  3. Nice-to-have stories
• Customer wants the release to be as **valuable** as possible

Source: Prof. O. Nierstrasz, *The Planning Game*  
© Oscar Nierstrasz
Commitment Phase ...

- **Developers:** Sort stories by risk
  1. Stories that can be estimated precisely (*low risk*)
  2. Stories that can be estimated reasonably well
  3. Stories that cannot be estimated (*high risk*)
  - Developers want to tackle *high-risk first*, or at least make risk visible

- **Developers:** Set team *velocity*
  How much ideal engineering time per calendar month/week can the team offer?
  - this is the *budget* that is available to Customer

- **Customer:** Choose *scope* of the release, by either
  - fixing the date and choosing stories based on estimates and velocity
  - fixing the stories and calculating the delivery date
### Table 13.1. Time Capsule Release Plan

<table>
<thead>
<tr>
<th>Story</th>
<th>Time Estimate (Ideal Weeks)</th>
<th>Assigned Iteration #</th>
<th>Assigned Release #</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find lowest fare.</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Show available flights.</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Sort available flights by convenience.</td>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Purchase ticket.</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Do customer profile.</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Do simple customer profile.</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Do full customer profile.</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Review itineraries.</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Cancel itinerary.</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Print immigration paperwork.</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Show hotels.</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Show hotel availability.</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Offer sophisticated hotel search.</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Book a hotel.</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Provide hotel/spaceline programs.</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Offer airplane hire.</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Planning Game: Steering Phase

**Purpose:** Update the plan based on what is learned.

**The Moves:**

- **Iteration:** see Iteration Planning
- **Get stories done:** *Customer* should only accept stories that are 100% done.
- **Recovery:** *Developers* realize velocity is wrong
  - *Developers* re-estimate velocity.
  - *Customer* can defer (or split) stories to maintain release date.

*Source:* Prof. O. Nierstrasz, *The Planning Game*  
© Oscar Nierstrasz
Iteration Planning

- An iteration takes from 1-3 weeks.
- Stories (and failed tests) are split into tasks.
- Same game as in the release round.
- Tasks are written down on index cards like user stories.
- While user stories are in the customer's language, tasks are in the developer's language.
- Duplicate tasks can be removed.

When we sign up for tasks for a **three-week iteration**, there are typically **15 real work days**. Divide by 3, there are **5 perfect engineering days** per developer. Sign up for 5 days of work.
Task cards for document downloading

Task 1: Implement principal workflow

Task 2: Implement article catalog and selection

Task 3: Implement payment collection

Payment may be made in 3 different ways. The user selects which way they wish to pay. If the user has a library subscription, then they can input the subscriber key which should be checked by the system. Alternatively, they can input an organisational account number. If this is valid, a debit of the cost of the article is posted to this account. Finally, they may input a 16 digit credit card number and expiry date. This should be checked for validity and, if valid a debit is posted to that credit card account.
Project parameters and evaluation

- 4 control variables:
  - Cost
  - Time
  - Quality
  - Scope

- 3 control variables – selected by Customers, Managers.
  The remaining variable – selected by the Development team.

- Suggestion: Fix Cost, Time, Quality
  → Tune Scope accordingly.
The Planning Game Work Flow

by Michael Karneim
& Richard Karneim
& Marcel Heling

Exploration Phase
- Write stories [B]
  all 3 weeks
- Estimate stories [D]
  can't estimate
  Split a story [B]

Commitment Phase
- Sort by value [B]
- Sort by risk [D]
- Set velocity [D]
- Choose scope [B]

Steering Phase
- Recovery [D]
- Reestimate [D]

Iteration
- Pick stories to impl. [B]
- Plan is not accurate
  run Functional Tests [D]
- Planning Game [D]

New story [B]

B = Business
D = Development
XP Project Map (1)
XP Project Map (2)
The XP release cycle

Select user stories for this release → Break down stories to tasks → Plan release → Develop/integrate/test software → Release software → Evaluate system
XP and Visual Design

• **Advantages of visual design:** Provides
  • Clues on a design problem:
    • Too many elements in the picture.
    • Obvious asymmetry.
    • Many more lines than boxes (high coupling).
  • Speed.

• **Problems of visual design:** No feedback about
  • Test passing.
  • Simple code.

• **Strategy:**
  • Draw a few pictures at a time.
  • Implement in testing + code (+ refactoring).
  • Do not save implemented pictures, since the decisions will probably change.
  • Draw pictures on a whiteboard.

• **Possibly:** Use a reverse engineering tool for getting a visual description of the system, If needed.
XP Roles and responsibilities

**Programmer** - writes tests and then code.

**Customer** - writes stories and functional tests.

**Tester** - helps customer write tests and runs them.

**Tracker** - gives feedback on estimates and process on iterations.

**Coach** - person responsible for whole process.
- watches the process as a whole and calls
- the team's attention to impending problems or opportunities for
- improvement

**Consultant** - supplies specific technical knowledge needed.

**Manager** – allocates resources.
Handling Problems

- **Under-estimation**
  - Sometimes too great a commitment will be made.
  - Check to see if rules are being followed.
  - If stories cannot be completed, ask the user to choose a subset.
    - Other stories will be finished later.

- **Uncooperative Customers**
  - Some customers won’t play the game.
  - XP relies on trust.
  - Don’t move on based on guesses.
  - If customer never makes an effort, perhaps the system isn’t worth being built.
Handling Problems

- **Turnover**
  - If programmers leave, they don’t take any information that only they have.
  - Tests exist for every feature, so nothing can be broken by ignorance.
  - New people can be trained by pairing with experienced programmers.

- **Changing Requirements**
  - This isn’t a problem for XP as it is for other development models.
  - Have only planned for today, won’t have to change our plans.
  - New features will just be added to the stories.
Success Stories
Chrysler's Payroll System

- Short name C3
- A payroll project at Chrysler which has since become famous as the 'birth project' of Extreme Programming.
- 10 programmers, 4 years.
- Seen as a monument of success for XP.
- They used XP to rework the internals of a large payroll system.
Ford Motor’s Cost Analysis System

• 12 programmers, 6 years.
• Had constantly changing requirements.
• Small releases were beneficial.
• Customers and Managers noticed greater system stability.
Critics

- Steve McConnell said that different processes are needed for different projects.
  - Also said that few of the ideas are new.

- Doug Rosenberg points out that the Chrysler System was actually a failure.
  - It was cancelled before completion.
  - It proved capable of less than a quarter of what was expected of it.

- Alistair Cockburn:
  - Extreme programming says, Do it My Way or Get Out. My philosophical stance is, Permit the Maximum Personal Variation
Conclusions

- XP appears to be a good approach to building a system that may have changing requirements.
- May not apply to a system where planning ahead is necessary.
- Beck recommends applying it to small to medium sized projects rather than large ones.
- Also recommends using it on sub-problems first to get the hang of it.
Industrial XP (23 practices)

- Industrial XP (IXP) evolved in response to challenges often associated with implementing Extreme Programming (XP) in organizations with 500 or more employees.
At this time, it seems that very few organizations are using eXtreme Programming as a methodology. However, the ideas within it, including test first and pair programming, short iterations, customer contact, no anticipation, and collective ownership have gone mainstream, and are widely practiced by many organizations.