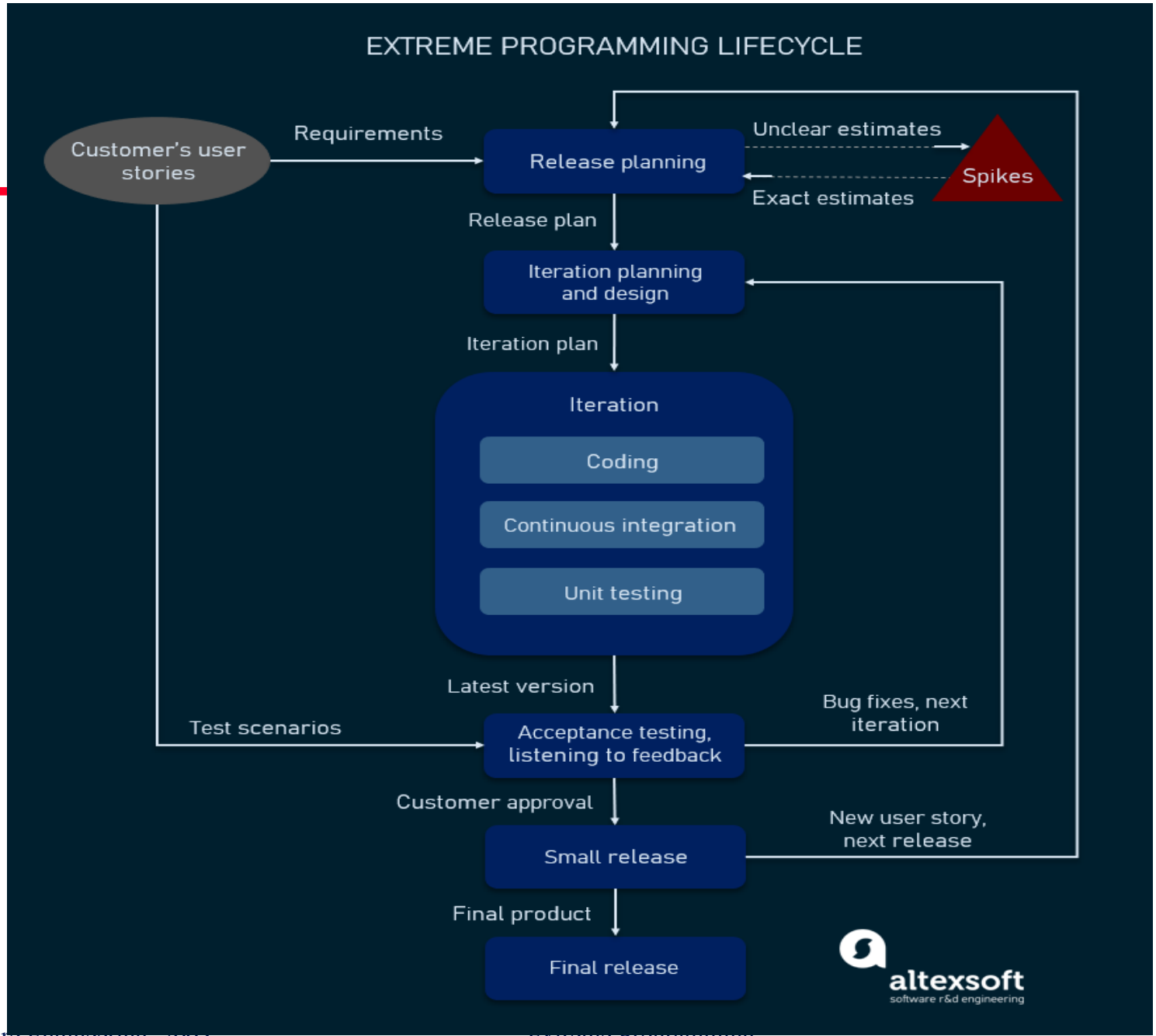
*E*xtreme Programming


EXTREME PROGRAMMING LIFECYCLE
Customer's user stories
Requirements
Release planning
Unclear estimates
Spikes
Exact estimates
Release plan
Iteration planning and design
Iteration plan
Iteration
Coding
Continuous integration
Unit testing
Latest version
Test scenarios
Acceptance testing, listening to feedback
Bug fixes, next iteration
Customer approval
Small release
New user story, next release
Final product
Final release
altexsoft
software r&d engineering

The 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)**.

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.

Pair Programming

Pair-programming has been popularized by the eXtreme Programming (XP) methodology

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**

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.

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.

4. Metaphor

- 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.

5. Simple Design

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

Simplest thing:

- XP developers always do the simplest thing that could possibly work.
- DTSTTCPW (do the simplest thing that could possibly work)
- They never solve a more general problem than the specific problem at hand.
- They 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.

Example: Eliminate redundant comments

```
/** * 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.

What is refactoring

- **Refactoring** is the practice of:

  Improving the design of code without breaking its functionality.

- Simplicity requires **constant refactoring**: small changes.
- **Goals:**
  - By constantly striving to keep code as concise and as simple as possible the cost of making changes to an application does not rise so dramatically over time.
  - Remove duplication: Duplicated logic is almost always harder to maintain because changes must be made to more than one part of the system as it evolves.
  - Without refactoring, complexity inevitably increases as more and more features are tacked onto a system.

Refactoring example: Rename variable

```
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

```
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.

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.

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).

Small releases

- 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.

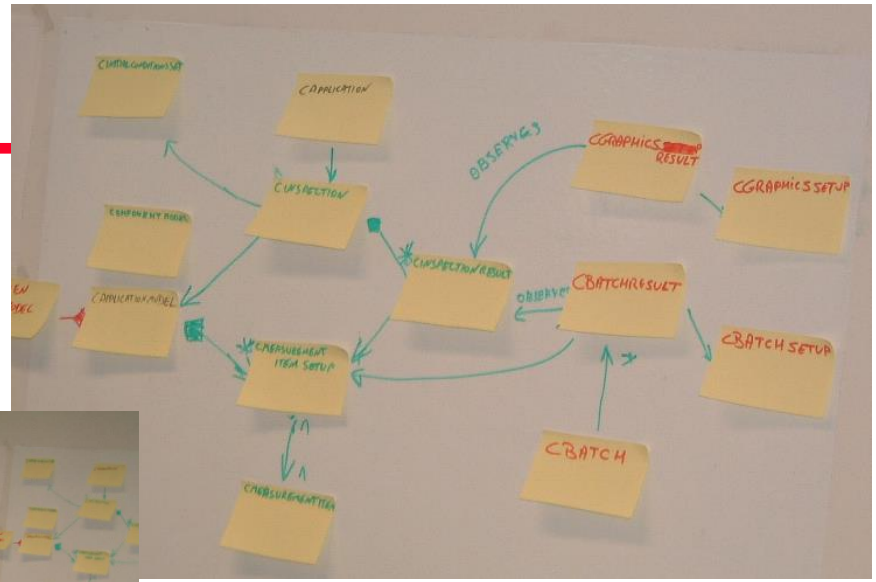
10. The Planning Game

- Schedule small tasks to be completed during the current completed iteration.
- Programmers will focus their attention on the tasks at hand.
- List of tasks is updated regularly.

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.

More info: http://www.xp.be/



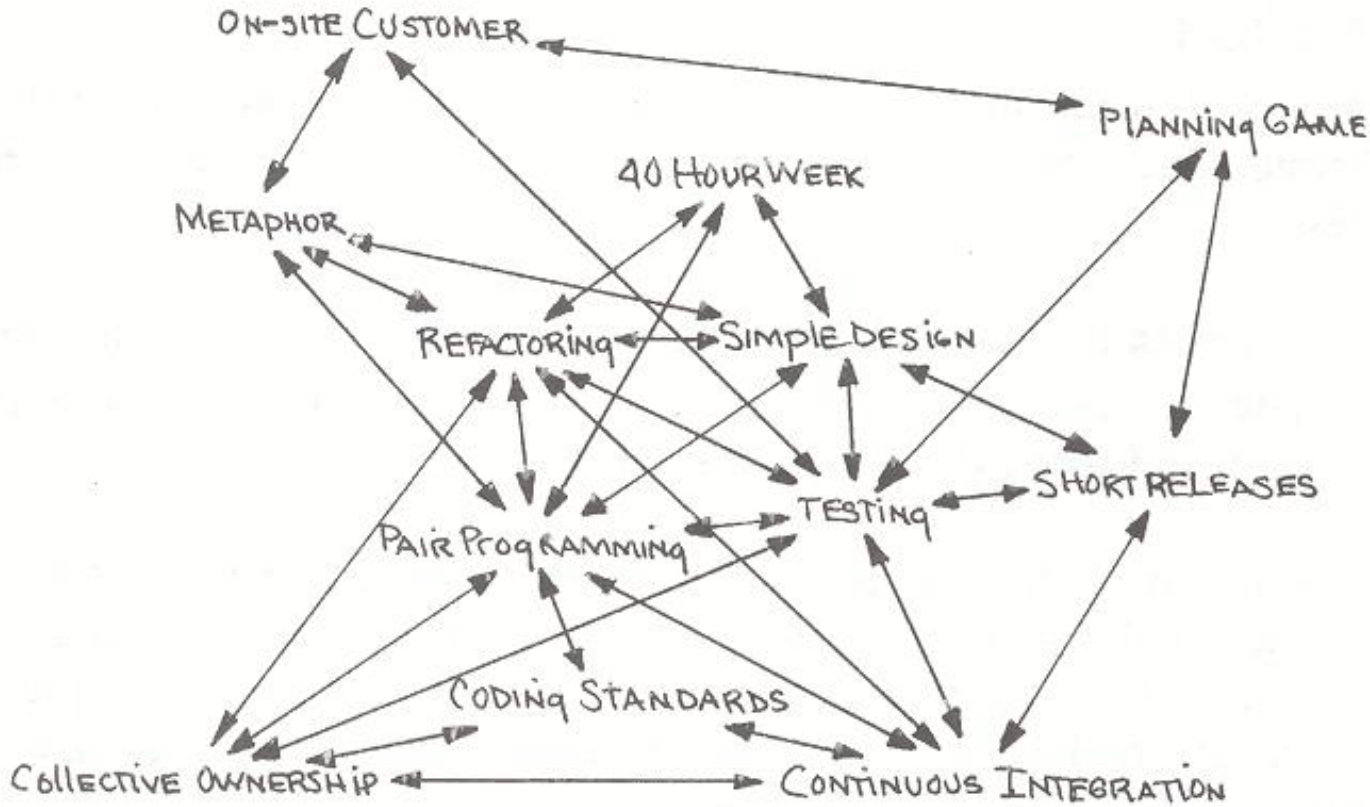
12. Sustainable pace: 40 Hour Weeks

- **Consecutive weeks of overtime is not allowed**.
- The need for overtime is a **symptom of a deeper problem**.

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.

Dependency of Practices



Source: Beck, K. (2000). *eXtreme Programming explained*, Addison Wesley.

The Planning Game:

How it works

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?

Daily Standup Meeting

- Stand up to keep it short.
- Everybody
  - Agrees what they will work on
  - Raises problems & difficulties
  - Knows what's going on
- Initial pairing.

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)

- ***Estimate Story**:* **Development** takes every story and assigns it a cost of 1, 2, or 3 weeks of IdealProgrammingTime . 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. (Sometimes we just batch small stories willy-nilly until they add up to at least a week, for estimation purposes.

- ***Make Commitment:*** **Business** and **Development** work together to decide 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** and **Date Driven**.

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 can't keep the PDF version so it is automatically deleted from your computer.

Release Planning

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.

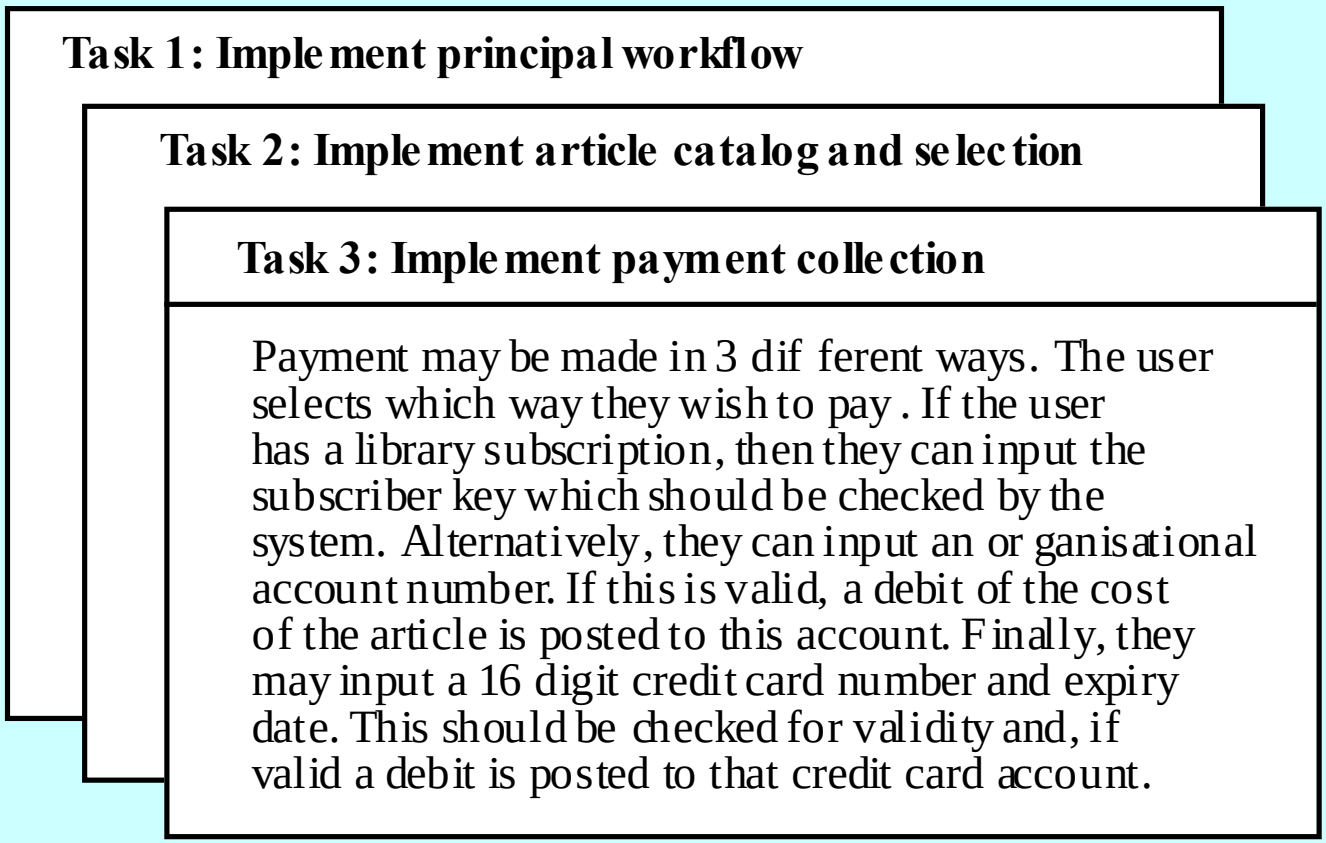
Steering phase:

- **Goal:** Update the plan.
- **Moves:** iteration, recovery, new story, re-estimate.

Iteration Planning

- An iteration takes from 1-3 weeks.
- Stories are split into **tasks**.
- Same game as in the release round.

Task cards for document downloading



XP and Fixed Price

- How can you do a fixed price / fixed date / fixed scope contract if you play the Planning Game?

  → You will end up with a

  fixed price / fixed date / roughly variable scope

  contract.

- Beck says: “Every project I've worked on that had fixed price and scope ended with both parties saying, "The requirements weren't clear." “
- Instead of fixed price/date/scope, the XP team offers something more like a subscription.
- A 12-month contract might put the system into production after three or four months, with monthly or bimonthly releases thereafter.

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.

Consultant - supplies specific technical knowledge needed.

Manager - makes decisions.

Handling Problems

Underestimation

- 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.

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.

XP Planning/Feedback times