

Extreme Programming (XP)

Pedro Cabalar

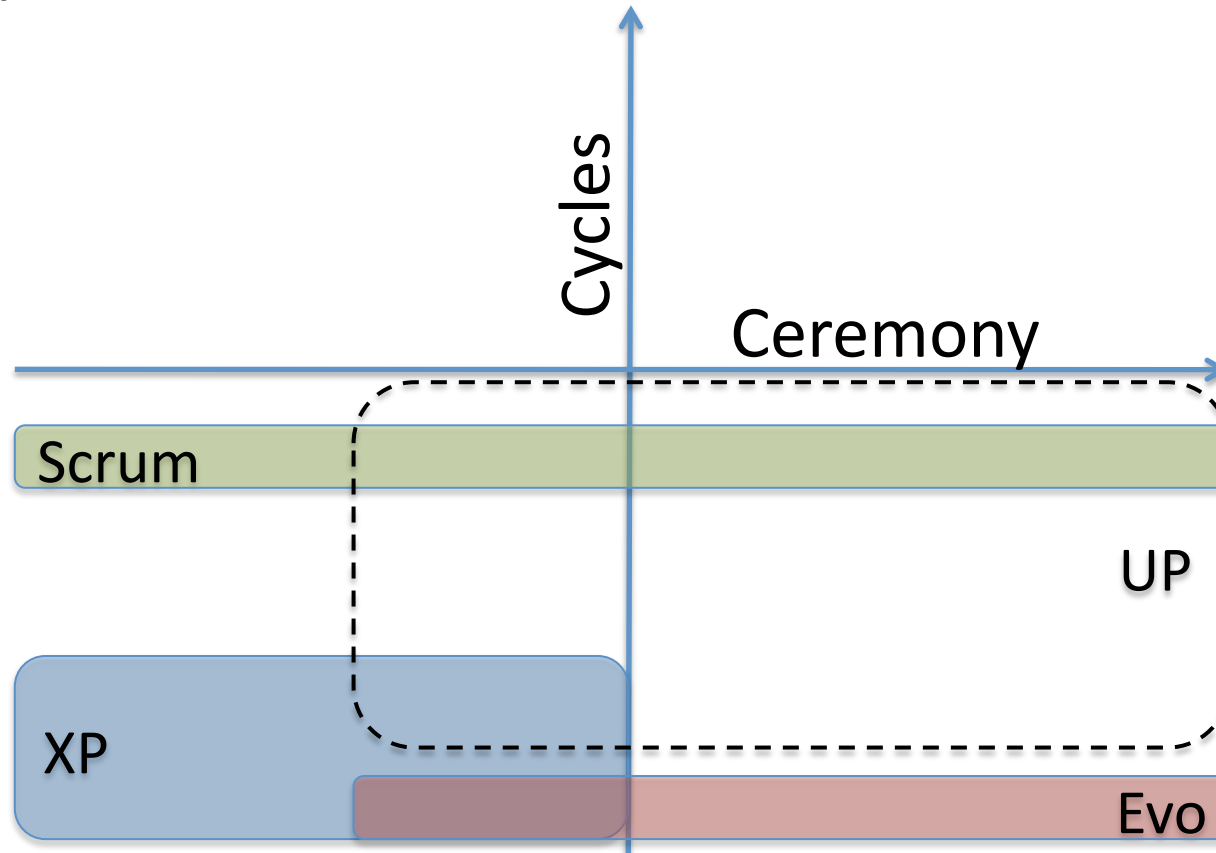
Departamento de Computación
Facultad de Informática
University of Corunna, SPAIN

Extreme Programming (XP)

- XP is an agile method emphasizing collaboration, quick/early SW creation and skillful development
- Four values: communication, simplicity, feedback and courage.
- It recommends 12 practices we will see later. They include pair programming, test-driven development, refactoring or continuous integration.

Classification of methods

- Few documents, informal (*story cards*)
- Frequent iterations, short (1-3 weeks)



Classification of methods

Criticality
(defects cause loss of ...)

Number of people
1-6 ≤20 ≤40 ≤100

Life (L)	L6	L20	L40	L100	...
Essential Money (E)	E6	E20	E40	E100	
Discretionary Money (D)	D6	D20	D40	D100	
Comfort (C)	C6	C20	C40	C100	

XP

- Typically: ≤10 developers.
- Not proved for safety-critical systems!

XP principles

- **Communication** and **team** work:
customers, developers, managers form a team and work in a **common room**
- XP doesn't detail workproducts: only **code** and **tests**.
- Other workproducts ok (story cards, task lists, ...),
but **oral communication is preferred**
- **Warning!** XP ≠ hacking
XP ≠ code-and-fix programming
XP = **disciplined** development - doc. overhead

XP practices

1. Whole Team

- Team consists of programmers and customers
- Def. **customer** = the one that defines and prioritizes **features**
- Possible customers
 - Business analysts or marketing specialists in the same company than SW developers
 - A representative commissioned by users
 - The paying customer him/herself

XP practices

1. Whole Team

- The customer should work in the **same room** than developers
- If not, (s)he should be **as close as possible**
- If not, **find someone** who can be close and stand in for the true customer
- Recent versions of XP consider a **group** of customers

XP practices

1. Whole Team

- The team uses a common **open workspace**, a “war room” with
 - Tables with workstations, each ws has **2 chairs**
 - Walls covered with calendars, diagrams, tasks lists, status charts, ... (**self-organizing team**)
- The sound is a **low buzz** of conversation
- Potential increase of distraction, but experience shows a **productivity increase of ×2** compared to isolated workplaces

XP practices

2. User stories

- To estimate a requirement we don't need all its details. **Details will surely change**
- The customer will talk about different **stories** or features.
- Each story is written on a **story card**, a paper index card with a few words. Ex: "find lowest fare"
- The story card is a **reminder of the conversation**.
- The developers write **an estimation** on the card

XP practices

3. Short cycles

- **Evolutionary delivery.** An XP project delivers working SW every two weeks
- **Release** = major delivery usually put into production. Frequency \approx 3 months or 6 iterations
- **Release planning game:** $\frac{1}{2}$ - 1 day where customer writes (prioritized) story cards for the next release and developers write estimations (budget + time). They fix a date.
- A release plan can be changed at any moment.

XP practices

3. Short cycles

- **Iteration** = minor delivery (\approx 2 weeks) that may or may not be put into production.
- **Iteration planning game**
 - customer decides the stories to implement subject to budget, but once the iteration starts, **no customer's changes are allowed**
 - developers fix the budget using experience from previous iterations
 - Developers split each story fixing a **list of tasks** that are assigned by a **volunteering round**

XP practices

4. Acceptance tests

- Written by the customer, they are examples of **what the system should do**
- Test principle: a test is useful if it can be passed **easily and repeatedly**
- In this way, each modification can be rechecked automatically, and acceptance criteria are never broken
- **Keypoint:** acceptance tests must be **automated**. Use some simple, tailored **script language**

XP practices

4. Acceptance tests

- An example

```
AddEmp 310 "John Smith" 1510.36  
Payday  
Verify Paycheck EmpId=310 GrossPay=1510.36
```

- The script language may be **enriched** along the project evolution

XP practices

5. Pair programming

- Two developers at each computer:
 - **Driver**: holds the keyboard/mouse and types the code
 - **Watcher**: looks for errors and improvements
- **Roles change** periodically
- **Pairs switch** at least once a day. At the end of an iteration, all possible pairs should have occurred
→ Everybody **has worked on everything**
- Pairs do not decrease efficiency while they significantly **reduce defect rate**

XP practices

6. Test-driven development

- Traditional test policy: first code, then test
- Test-driven = **first write the tests**, then make the program to pass those tests
- Tests & code evolve, but tests always go **ahead**
- Advantage 1: we gain a growing **corpus of executable tests**
- Advantage 2: small changes are automatically tested **not to break anything**. This facilitates refactoring

XP practices

6. Test-driven development

- Typical use of **unit test tools** (e.g. JUnit)
- Advantage 3: unit tests + object oriented design encourage **module decoupling**
- **Mock objects** = they **replace/simulate** a real object whose behaviour is difficult to predict or is **just not implemented** yet. Example
 - Real object = alarm that sends a message at a given clock time
 - Mock object = replaces the alarm and sends the message during the test

XP practices

7. Collective ownership

- Any pair of programmers can improve **any code**
- There is **no individual responsibility or authority** for any module or technology
- The entire team is **collectively responsible**
- Keypoint: replace
“it’s your code, so your problem”
by
“I spotted the problem, so I fix it”

XP practices

7. Collective ownership

- Nice but, what if it's not my speciality and I break something made by a real expert?
- Remember that tests won't let you break things
- Besides, we should be helped by coding standards (all code should look the same)

XP practices

8. Continuous integration

- Code integration is usually managed using **SW configuration management**: check-in (commit), merge
- Integration is done **several times per day**
- Example: a pair of programmers have been working 2 hours on a given task. They decide to check-in their tests and code. Steps:
 1. They **run their new tests** on their new code
 2. They **check-in** (or **merge** if another pair did it before)
 3. They integrate the code, **build the whole system and run all its tests** (unit + acceptance)
- Automated integration **tools**. Ex: AntHill, CruiseControl

XP practices

9. Sustainable pace

- A SW project is not a sprint: it is a **marathon**
- The team must preserve their energy and alertness, running a **steady, moderate pace**
- Working **overtime is generally not allowed**.
Frequent overtime = symptom of deeper problems
Only exception: last week for a release, but handle with care

XP practices

10. Simple design

- Choose the **simplest way first**
- Avoid speculative design for hypothetical future changes.
- Keep design **expressive** and **comprehensible**
- If you think “*I know I’m going to need X*” but the story/task doesn’t require X, **reject it**
- Example: store a list of users’ suggestions
 - “*ok, let’s begin choosing a database*”
 - “*hey, would a simple flat file work?*”

XP practices

10. Simple design

- Avoid top-down abstraction: avoid creating generalized components that are **not immediately needed**
- On the contrary, bottom-up: create abstractions to **remove duplication** detected in existing code
- Adopt **coding standards** agreed by all the team members. They are **crucial** for success when we have collective ownership, rotation of pairs and refactoring!

XP practices

11. Refactoring

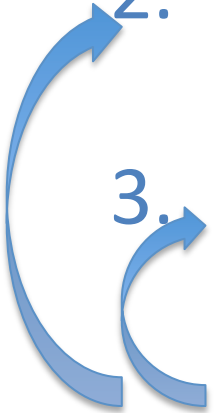
- With iterative development SW quality tends to degrade into a mess, if we don't look back on what was done before
- **Refactoring** = tiny transformations that improve the system structure without affecting its behaviour
- Continuously (each ½ -1 hour) make small changes to keep code **clean, simple and expressive**. Then run tests.

XP practices

12. Metaphors

- To help seeing the big picture, capture concepts using **memorable metaphors**
- Example: periodic chunks of data to be processed → *“putting slices in the toaster”*
- May sound ridiculous or useless, but people understand things better using metaphors. Example: directory with files vs *folder with documents*

XP lifecycle

1. **Exploration**: write initial story cards, estimates, check feasibility
 2. **Planning**: release planning game, detail story cards and estimations, fix next release date
 3. **Iterations for release**: iteration planning game, task writing, testing and programming.
I1, I2, I3 ...
 4. **Productionizing**: operational deployment, documentation, training, marketing
 5. **Maintenance**: enhance, fix, may start again
- 
- A diagram illustrating the XP lifecycle. It consists of five numbered steps. Step 3, 'Iterations for release', includes a sub-list of iterations 'I1, I2, I3 ...'. A large blue curved arrow starts from the left side of step 3 and points back to step 2, 'Planning', indicating a feedback loop. A smaller blue curved arrow starts from the left side of step 3 and points to the right, indicating the flow of the process.

Other XP practices

- Embrace change rather than fighting change
- Visible wall graphs: metrics, tasks, diagrams
- Tracker: regular collection of task and story progress
- Daily standup meeting
- Ideal Engineering Hours (IEH) is the measure for estimates