Conduite de Projet
Cours 10 — Extreme Programming

Stefano Zacchiroli
zack@irif.fr

Laboratoire IRIF, Université Paris Diderot

2017–2018

URL https://upsilon.cc/zack/teaching/1718/cproj/

Disclaimer

- lot of best practices
  - 50% technical practices
  - 50% project management practices

- lot of "philosophy" (as it often happens in software engineering. . .)
  - think it through and compare with different "philosophies"
  - make up your own mind about it
  - mix and match

Outline

Outline

Reminder: software methodology

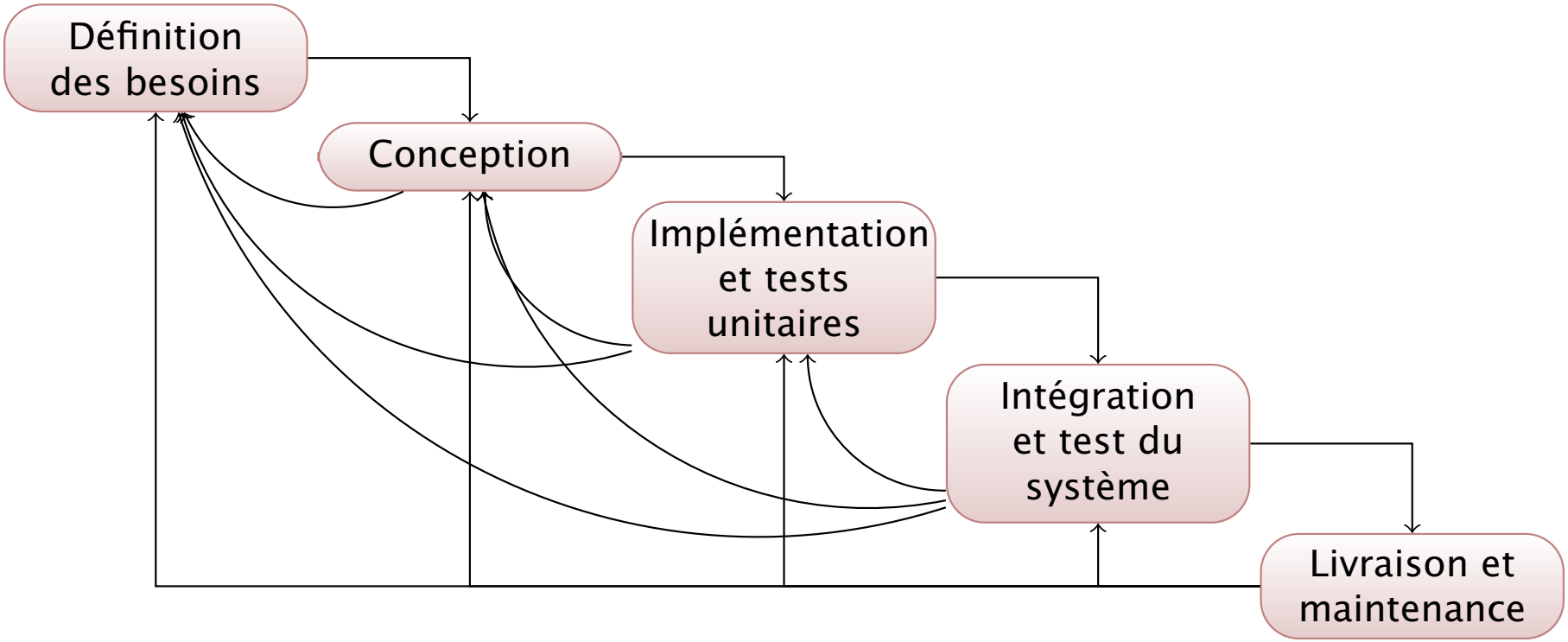
A software methodology is a formalized process or set of practices for creating software

- a set of rules you have to follow
- a set of conventions the organization decides to follow
- a systematical, engineering approach for organizing software projects

What is Extreme Programming?

- Extreme Programming (XP) is a methodology of software development based on well laid out values, principles, and practices.
- Goal of XP is outstanding software development at lower cost, with fewer defects, high productivity and much higher return on investment.

Reminder: waterfall



What is XP?

Why yet another methodology?

- Social Change — Giving up defences
- Based on Excellent programming techniques, clear communication and teamwork
- Lightweight — only do whatever adds value to the customer
- Addresses constraints in software development
- Adapts to rapidly changing requirements

What is XP?

XP addresses risks at all levels of development process

- schedule slips
- defect rate
- business misunderstood
- business changes
- false feature list
- staff turnover

Outline

History

- Early Influences
  - Incremental, stakeholder-driven design process from Alexander
  - Programming as learning from Papert, Kay
- Kent Beck & Ward Cunningham
  - Mid-80s — Pair programming at Tektronix
  - 80s, 90s — Smalltalk culture produces refactoring, continuous integration, constant testing, close customer involvement
  - Generalized to other environments
  - Early 90s — Core values developed within patterns community, Hillside Group

History

- Scrum practices incorporated and adapted as planning game
- 1999 — Extreme Programming Explained
- 1999 — Fowler publishes Refactoring
- 1999 — XP Immersion held, e-group formed
- 2000 — more books, first conferences
- Evolution continues through today

Agile

We are uncovering better ways of developing software by doing it and helping others do it.

Through this work we have come to value:

Individuals and interactions over *processes and tools*
Working software over *comprehensive documentation*
Customer collaboration over *contract negotiation*
Responding to change over *following a plan*

That is, while there is value in the items on the right, we value the items on the left more.

Manifesto for Agile Software Development
http://agilemanifesto.org/

XP and Agile

Agile methodologies:

- eXtreme Programming
- Scrum
- Crystal family
- Feature-Driven Development (FDD)
- Adaptive Software Development (ASD)
- Dynamic System Development Model (DSDM)
- Agile Unified Process (AUP)
- ...

XP and Agile

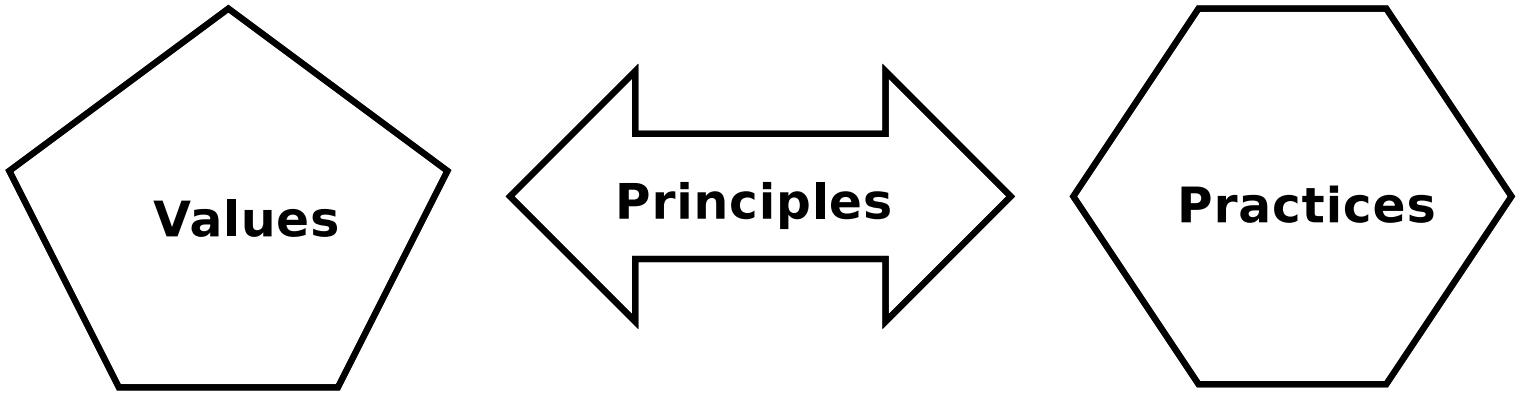
Agile methodologies:

- eXtreme Programming

XP is just one among several others Agile methodologies.
It's likely the most popular and the only discussed in this class.

Outline

Core components



- values bring purpose to practices
- practices are evidence of values
- principles are domain specific guidelines

Core values

1. communication
2. simplicity
3. feedback
4. courage
5. respect

#1 — Communication

- Often problems that arise in a SW project can be tracked back to lack of communication.
- XP enforces the Communication Value by employing many practices that could not be carried out without communicating
  - e.g., pair programming, unit testing, etc.
- XP employs a Coach whose job is that of noticing when people are not communicating and reinforce communication.

#2 — Simplicity

- “Do the simplest thing that could possibly work” (DTSTTCPW) principle (elsewhere known as KISS).
  - XP coachs may say DTSTTCPW when they see an XP developer doing something that is needlessly complicated.
- YAGNI principle (“You ain’t gonna need it”)
- Simplicity and Communication support each other mutually.

#3 — Feedback

Feedback works in XP at different time scales.

- Programmers have feedback on a minutes time scale on the status of the system thanks to unit tests.
- When customers write new stories the programmers estimate those immediately to give prompt feedback to the customer about the quality of the stories.
- The customer review the schedule every 2–3 weeks and provide prompt feedback to the developer.

#4 — Courage

- XP team should have the courage of throwing code away.
- XP team should have the courage of massively refactor the architecture of the system, if architectural flaw are detected.
- Courage without counterbalancing values is dangerous. Doing something without regard for consequences is not effective teamwork.

#5 — Respect

- respect for team members
- respect for the project

Principles

Humanity People, What do people need to become good developers?

Economics Every action should have business value

Mutual Benefit Most important and most difficult to adhere to. Extensive internal documentation

Self Similarity You can copy structure of one solution to a new context. Theme, story, tests

Principles (cont.)

Improvement In software development “perfect” is a verb not an adjective

Diversity Teams need diversity

Reflection How and Why of working

Flow Steady flow of valuable software

Opportunities Problems are opportunities

Principles (cont.)

Redundancy Do not remove redundancy that serves a valid purpose

Failure Is failure a waste?

Quality Cost of quality? Quality $\approx$ Productivity

Baby Steps Rapid small steps = leap

Accepted Responsibility Responsibility cannot be assigned

Outline

Primary Practices

1. Metaphor
2. Whole team
3. Informative workspace
4. Cycles
5. Release planning
6. Small releases
7. Testing
8. Pair programming
9. Refactoring
10. Simple, incremental design
11. Collective code ownership
12. Continuous integration
13. Customer involvement
14. Energized work
15. Coding standards
16. Stand up meeting

Practice: Metaphor

- Guide all development and conversations with a simple shared story of how the whole system works
  - Gives the team a whole picture of describing the system, where new parts fit, etc.
- Words used to identify technical entities should be chosen from the metaphor
- The default metaphor is the business domain, and it's usually just fine

Metaphor — discussion

- good idea in general
- comes naturally (but should be informed)
- people should know the business needs and how their work fits in the project
- non-default metaphors must be correct

Practice: Sit together / whole team

- the whole team sits together in the same room
  - ideally around the same table or several tables
  - whole team: programmers, testers, interaction designers, architects, customers, project managers, product managers, executives, technical writers, ...
- emphasizes that the goal is shared and reduce conflicts
  - i.e., "whole team" goal is actually bigger than just "sit together"

Sit together — discussion

- require team to be co-located in the same geographical area
  - not always a possibility
- poses an upper bound to the size of XP teams
  - one of the most common criticism of XP

Practice: Informative workspace

- make your workspace about your work

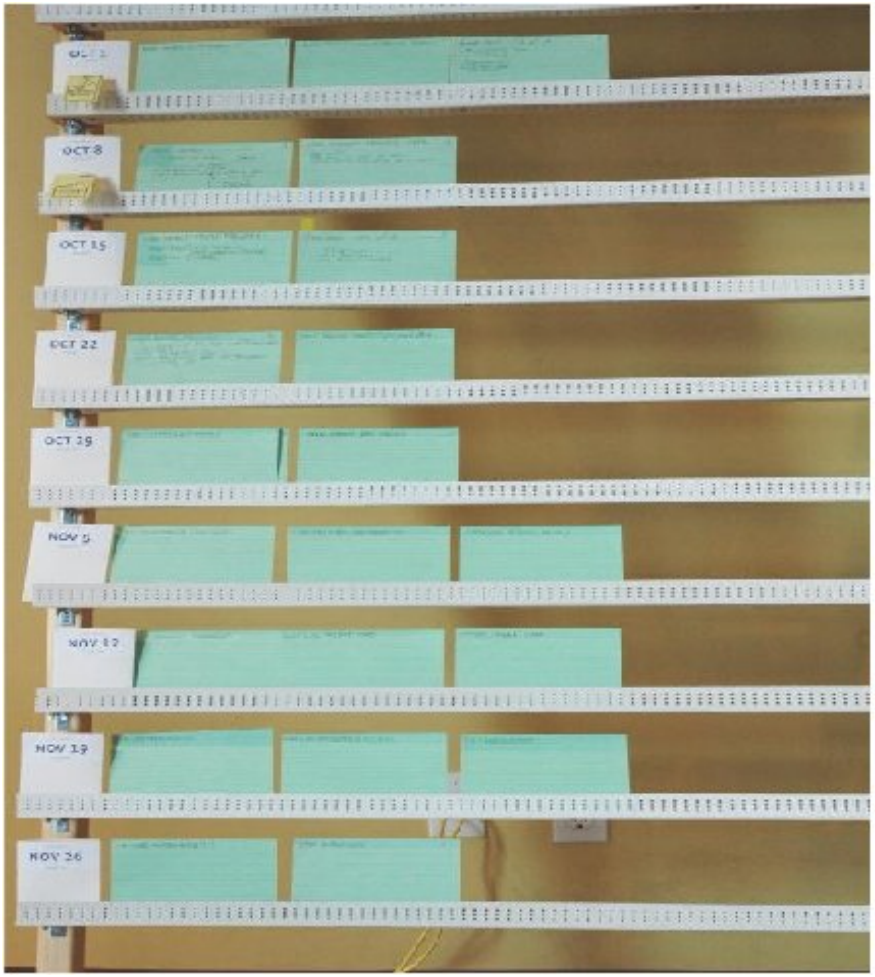
  an observer should be able to walk into team space and get a general idea of how the project is going in 15 seconds

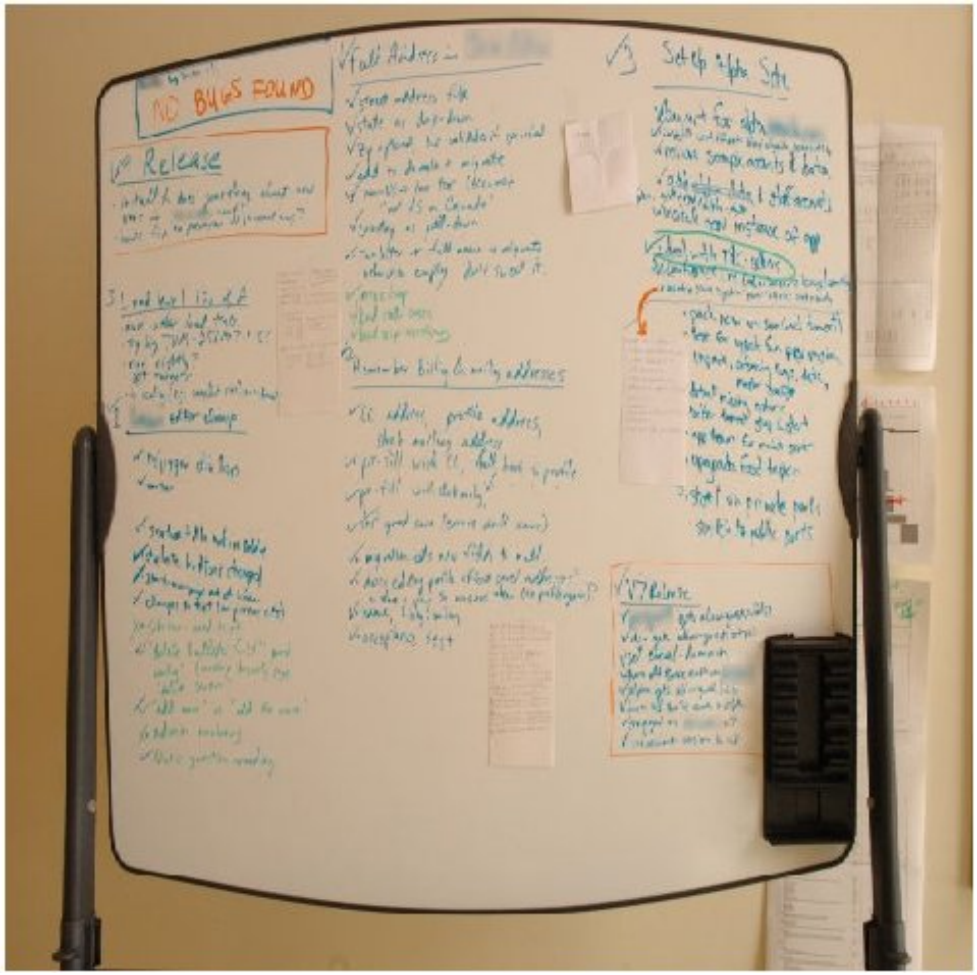
  — *Kent Beck*

- common sub-practices
  - user stories (more on this later) on the walls
  - monitor progress (e.g., using burn down charts)
  - take care of human needs (snacks, beverages, etc.)
    - they encourage social interaction (good both to get the job done and for whole team integration)

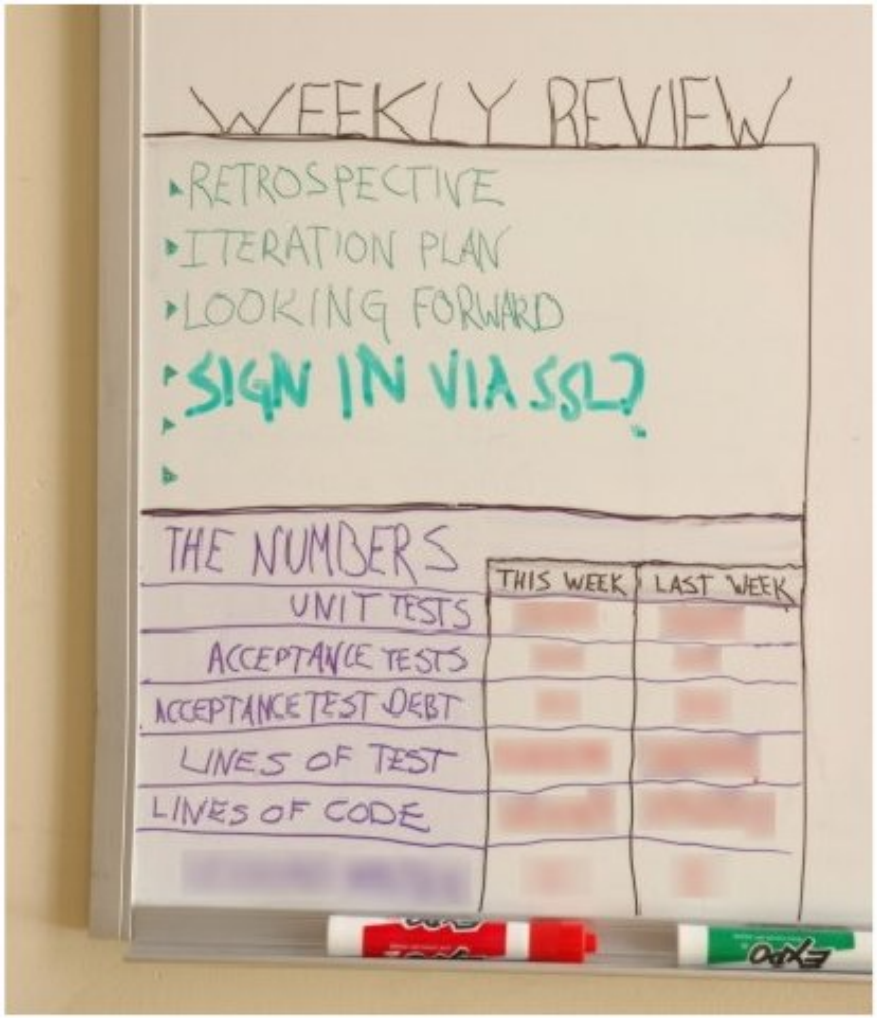
Informative workspace (cont.)

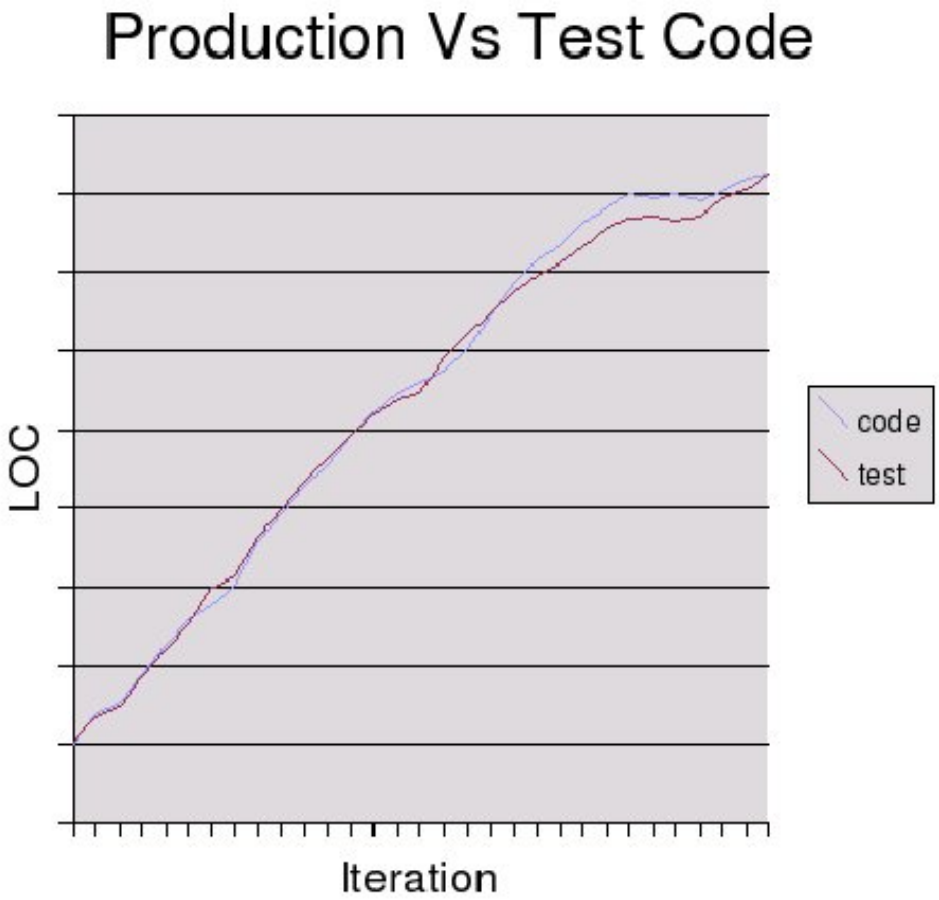
Informative workspace (cont.)



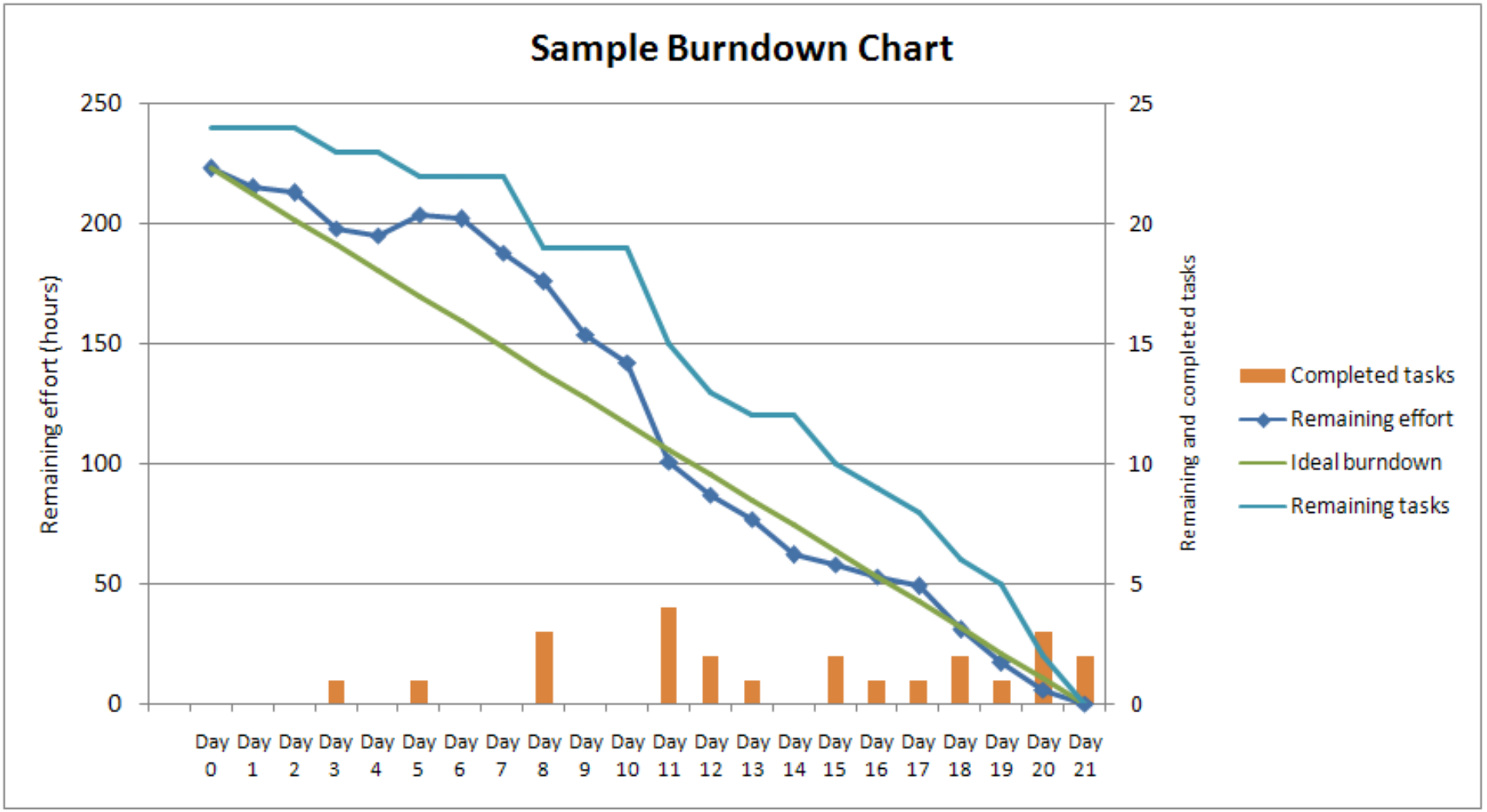


Informative workspace (cont.)





Informative workspace (cont.)



Practice: Cycles

Weekly cycles Start week by writing automated tests and then spend the week implementing them. Mainly about stories and tasks.

Quarterly cycles plan for releases which cover them. Themes can be broken down into stories for weekly cycles.

Practice: Release planning

- Requirements via User Stories
  - Short cards with natural language description of what a customer wants
  - Prioritized by customer
- Resources and risk estimated by developers
- Via “The Planning Game”
- Play the Planning Game after each increment

User stories

- units of customer visible functionality
- should be on the story wall for everyone to look at, always

Example

Search and Replace

A user realizes she mis-capitalized a word everywhere in her document, so she tells the word processor to search for all occurrences of it and replace them with the correct word.

User stories ≠ use cases

- A use case defines functional requirements in total
- Set of use cases define breadth and depth of system behaviour, augmented with non-functional requirements

- A story points to a portion of breadth and depth
- Stories incrementally build up full scope

- Usually 2–3 days of work
- Further broken down into tasks

The planning game

- planning is emotional, thanks to conflicting requirements
  - developers would like to program faster
  - project managers want guaranteed deadlines
  - customers would like to be able to say exactly what they want
  - business would like not to change their mind
- the planning game tries to reduce the emotional tension turning planning into a game
- in the planning game there are pieces, goals, players, and moves

The planning game (cont.)

- pieces: user stories
  - each user story can get associated to a value and a cost
- goal: put the greatest possible values of stories into production
- players: developers, customers, and business representatives
- moves:
  - write a story (customer) and assigns a value to it (business)
  - estimate the cost of a story (developer)
  - make commitment: decide release date and addressed stories (joint business/developer)
    - story-driven commitment: business add stories incrementally; developers compute release date (incrementally)
    - date-driven commitment: business pick a date; developers announce manageable overall cost; business pick stories up to it
  - value and risk first: development order stories so that
    - a fully working, but sketchy, system can be delivered ASAP
    - more valuable stories are moved earlier in the schedule
    - riskier stories are moved earlier in the schedule
  - split story so that part of it can fit in a release (business)
  - other moves...(see `http://c2.com/cgi/wiki?PlanningGame`)

Release planning — discussion

- user stories vs requirements specification
  - user stories are way simpler and less detailed
  - intuition: the details (in general) are subject to change
  - but: written documentation might work well for large projects
- yet another alternative: prototyping user interface
- estimate required resources is hard, might work well only with very experienced developers
- small releases are less risky in general

Practice: Small releases

- Timeboxed
- As small as possible, but still delivering business value
  - e.g., no release to “implement the database”
- Get customer feedback early and often
- Do the planning game after each iteration
  - Do they want something different?
  - Have their priorities changed?

Small releases — discussion

- Small releases are really valuable
  - Manage the risk of delivering something wrong
  - Helps the customer to define better requirements
- Release every few weeks
- Large projects are not so flexible
  - Try to release something, even if you know it will be changed

Practice: Testing

- Test-Driven Development (TDD)
  - Write tests before code
  - Tests are automated
  - Rely on some unit testing framework
  - Must run at 100% before proceeding
- Acceptance Tests
  - Written with the customer
  - Acts as “contract”
  - Measure of progress

Test-Driven Development

- Developers write unit tests before coding
- Motivates coding
  - Improves design: cohesion and coupling
  - Provides regression tests
  - Provides specification by example

TDD — an example

Implement a stack.
Which tests would you design before coding?

TDD — an example

Implement a stack.
Which tests would you design before coding?

- When I create a stack it should be empty
- When I push an element on an empty stack the size should be 1
- When I push 3 elements on an empty stack the size should be 3
- When I pop an element from a stack with one element, the stack should be empty
- When I pop an element from a stack with 3 elements, the size should be 2
- When I pop an element from an empty stack, it should result in underflow condition
- When I push 5 elements on a stack of capacity 4, it should result in overflow condition

TDD — discussion

- TDD is good for most projects, not for all
  - The real world is different: you always need the functionality “for tomorrow”!
- one might argue that testing simple logic is overhead and hence write unit testing for complex logic only

Practice: Pair programming

Two software engineers work on one task at one computer

The driver has control of the keyboard and mouse and writes the implementation

The observer watches the driver's implementation

- identifies defects and participates in on-demand brainstorming
- performs "on the fly" code review
- has more time than the driver to think at the big picture

- The roles of driver and observer are periodically rotated

Pair programming — expected benefits

- Pairs produce higher quality code
- Pairs complete their tasks faster
  - to be worth: at least twice as fast
- Pairs enjoy their work more
- Pairs feel more confident in their work

Pair programming — discussion

- Pair programming is great for complex and critical logic
  - When developers need good concentration
  - Where quality is really important
  - Especially during design
  - Reduces time wasting, e.g., on line chatting, social network
- Trivial tasks can be done alone
- Code review among peers is often a viable alternative

Practice: Refactoring

- Improve the design of existing code without changing its functionality
  - Relies on unit testing to ensure the code is not broken
- Bad smells in code:
  - Long method / class
  - Duplicate code
  - Method does several different things (bad cohesion)
  - Too many dependencies (bad coupling)
  - Complex / hard-to-read code

Refactoring — discussion

- Delivering working software faster is important!
  - You can write the code to run somehow
    - With simple design
    - With less effort
  - Later you can refactor the code if necessary
- Refactoring is not a reason to intentionally write bad code!
  - Good coding style is always important

Practice: Simple, incremental design

- No Big Design Up Front (BDUF)
  - Reduces the overhead
  - Ship working functionality faster and get feedback early
- "Do The Simplest Thing That Could Possibly Work"
  - Later use refactoring to change it
- Not too much formal documentation

Simple design — discussion

Simple design does not mean “no design”

- It is about establishing priorities
- It’s a set of tradeoffs you make
- If something is important for this release and for the whole system, it should be designed well
- Don’t lose time to design something you will not use soon!

Practice: Collective code ownership

- Code do belong to the project, not to an individual engineer!
- Any engineer can modify any code
- Better quality of the code
- Engineers are not required to work around deficiencies in code they do not own
  - No need to wait for someone else to fix something
  - Faster progress

Collective code ownership — discussion

Collective code ownership is absolutely indispensable

- You need to fight the people who don't agree with this!
- Fire people writing unreadable and unmaintainable code
- Don't allow somebody to own some module and be irreplaceable

Practice: Continuous integration

1. Pair writes up unit test cases and code for a task (part of a user story)
2. Pair unit tests code to 100%
3. Pair integrates
4. Pair runs ALL acceptance test cases to 100%
5. Pair moves on to next task with clean slate and clear mind

- Should happen once or twice a day

Requirements:

- "($\leq$) 10 minutes build"

Continuous integration — discussion

Integrating often is really valuable

- Sometimes you cannot finish a task for one day and integrate it
- For small projects with small teams integration is not an issue
- For large and complex projects it's crucial
  - Think of automated build environment

Practice: Customer involvement

- Customer available on site
  - Clarify user stories
  - Make critical business decisions
- Developers don't make assumptions
- Developers don't have to wait for decisions
- Face to face communication minimizes the chances of misunderstanding

Customer involvement — discussion

On-site customer does not work! (or does it?)

- Customers are busy
  - Meetings every day might work better
- Customers are not competent!
  - Customers always say “Yes, this is what I want” and later say the opposite
  - You need to think instead of them
  - Use prototyping

Practice: Energized work

- Work only as many hours as productive

  fresh and eager every morning, and
  tired and satisfied every night

- Burning the midnight oil kills performance
  - Programmer productivity is seen to be 4-5 hours a day
  - Upper limit: 40 hours a week (i.e., crunch is a bad idea)
- Tired developers make more mistakes
  - Slows you down more in the long run
- If you mess with people's personal lives (by taking it over), in the long run the project will pay the consequences

Energized work — discussion

- 40 hours a week or 40 hours without a sleep?
  - Come back to the real world!
  - Overtime is not recommendable but often can not be avoided
- Better planning can help
- Highly skilled senior engineers always suffer of overtime and high pressure
  - That's how the business works!

Practice: Coding standards

- Use coding conventions
  - Rules for naming, formatting, etc.
  - Write readable and maintainable code
- Method commenting
  - Self-documenting code
  - Don't comment bad code, rewrite it!
- Refactor to improve the design
- Use code audit tools

Coding standards — discussion

- Coding standards are important
  - Enforce good practices to whole the team — tools, code reviews, etc.
- Standards should be simple
  - Complex standards are not followed
  - Standards should be more strict for larger teams
  - Developers don't like utter rules like "comment any class member"

Practice: Stand up meeting

Start the day with 15-minute meeting

- Everyone stands up (so the meeting stays short) in circle
- Going around the room everyone says specifically:
  - What they did the day before
  - What they plan to do today
  - Any obstacles they are experiencing
- Can be the way pairs are formed

Scaling XP

- XP seems to allow smaller teams to accomplish an awful lot
- XP seems to hit single-team challenges around 12-16 developers
- XP can scale by building recursive teams
- Recommended to build small team first, incrementally grow, and use first team to seed recursive teams
- XP has been used on teams of 40–50

Documentation

- XP is a minimalist process
- Many teams successful with index cards and large Post-It
- XP is not anti-documentation, but encourages doing the least amount that is really needed
- Document when needed for distributed sharing, historical needs, summarizing, etc.
- Documentation “surrogates” are always present
  - tests
  - prototypes
  - involved customers

Outline

How XP (tries to) solve some sw.eng. problems

Problem	**Solution**
slipped schedule	short development cycles
cancelled project	intensive customer presence
cost of changes	extensive ongoing testing, systems always running
defect rates	unit tests, acceptance tests
misunderstand the business	customer involvement
business changes	changes are welcome
staff turnover	intensive teamwork, collective code ownership

Applicability of XP

- domains with changing requirements
- high-risk project (e.g., tight schedule)
- small project team
- extended development team
  - developers, managers, and customers together
  - co-located
- automated testability

Mix and match

- The practices in different agile methods can be extracted and combined
- Establish your own process
  - Build it step-by-step
  - Adapt good practices one by one
- Example:
  1. Pair programming and its variation
  2. Daily 15-minutes meeting
  3. Test-driven development

References

Kent Beck
Extreme Programming Explained: Embrace Change.
Addison-Wesley, 2nd edition, 2004.

Martin Fowler
Refactoring: Improving the Design of Existing Code
Addison-Wesley, 1st edition, 1999.