Process Examples: TSP & XP

15-313:
Foundations of Software Engineering

Jonathan Aldrich

Announcements: Project

- You will make an enhancement to some open source project
- Goal: insight into open source processes and practices
- This week
  - Form teams of 2
  - by Saturday: pick a project, look for a task, email both to donna@cs
  - Bring up your proposal with project members
  - by Tuesday
    - build the project
    - draw an architectural diagram relevant to your task
    - describe the task
- No more homeworks (after this week)
- By April 29 / May 1
  - Make your enhancement
  - Submit it to the project
    - include tests that fit into project's testing strategy
    - package fix as a patch
    - appropriate documentation
    - "sell" your enhancement to the project
  - Write a report
    - task, design, interaction with community, lessons learned, code
  - Present in class
- Scope
  - 2 people, 2 weeks = pretty small
  - bug fixes are OK

Outline

- **Team Software Process**
- Extreme Programming
- Comparison

Team Software Process (TSP)

- Developed by Watts Humphrey of the CMU SEI
- Process is completely defined
  - Even up to individual reporting forms
- Scales well to medium/large projects
- Moderate overhead, but still adoptable
- Substantial use in industry

Process Challenges & TSP Response

- Unclear goals that differ among team
  - TSP: Up-front process to set goals with all team members
- Unclear division of responsibility
  - TSP: Set of predefined roles are assigned up front
- Unclear or poorly thought through plan
  - TSP: Up-front planning: estimate work, divide into cycles, ordering within cycle, who does what
- Bad communication
  - TSP: Weekly meetings with defined content
- Poor quality
  - TSP: Defined quality management practices in process

TSP Development Strategy

- Iterative development
  - Minimum functionality at end of cycle 1
- Produce conceptual design
  - Only for planning purposes
- Allocate functions to development cycles
- Size and time estimates for products
  - Detailed for current cycle, high-level for others
- Assess risks
  - Rate by impact, likelihood, and assign to team members
- Develop configuration management plan
  - Copies of each version
  - Record of who/what/when/why for each change
- Document all of the above on standard forms

TSP Development Cycle Plan

- List of tasks
  - Size estimate: <10 hours per engineer
    - e.g. for 4 engineers, <40 hours
  - Includes coding but also non-coding tasks like design, tests, meetings, etc.
  - Assign effort by week and by engineer
- Unplanned tasks
  - Major: produce a new plan
  - Minor: allocate 5-10% of time to unplanned tasks
    - Important to account for earned value

TSP Tracking

- Tasks
  - Date task completed
  - Time spent on each task
    - Spend more time in design than coding
    - Spend 50% of design time in reviews
  - Earned value by task, week, total to date
- Defects
  - Phase injected
    - Do we need to improve a particular phase?
  - Phase removed
    - Do we need to catch errors earlier?
    - Yields
      - 75% removed before first compile
      - 85% removed before first unit tests
      - 97% removed before integration test
      - 99% removed before system test
  - Broken down by module
    - Enables finding problem code

TSP Requirements

- Review system "need statement" for necessary clarifications
- Divide requirements among team by area
- Document requirements
  - a use case for each function
  - source of the requirement
- Inspect requirements document within team
  - 30 minutes per page
- Review requirements document with customer
- Check in requirements
  - Further changes require change control documentation

TSP Design

- Define components
  - Allocate use cases to each
- Set up design standards
  - naming conventions
  - how to document interfaces, error messages
  - how to represent design
- Develop detailed designs for each component
  - Produced and reviewed by team member
- Inspect design
  - Every use case covered
  - Design is complete and correct
- Check in design document

TSP Implementation

- Standards-based
  - naming, interface, coding standards
  - measurement standards for sizes, defects
  - defect prevention
- Development
  - Detailed design, personal design review
  - Develop unit test
  - Detailed design inspection
  - Code
  - Engineer reviews code personally, then compiles
- Team inspection
- Unit test
- Review quality

TSP Postmortem

- Analyze project data
  - Identify problem areas
  - Suggested resolutions
- Peer reviews / constructive criticism
- Summary report

Outline

- Team Software Process
- **Extreme Programming**
- Comparison

Extreme Programming (XP)

- Developed by Kent Beck and Ward Cunningham
  - Chrysler benefits organization project
- An iterative/spiral process
  - Divides development into short iterations delivering functionality
- Lightweight practices
- Increasingly popular in industry
  - But primarily for small teams/projects
  - Scale-up is widely questioned
- Seen as "fun," but requires great discipline

XP Requirements

- User stories on 3x5 cards
  - Scenario / use case for system
  - Requirements
    - Testable with automation
    - Progress toward customer's goal
    - Doable in <1 two-week iteration
    - Estimatable

XP "Planning Game"

- Input: stories
  - Customer assigns priority
  - Development assigns cost in "ideal time"
- Set release date
  - Customer picks date
    - developers compute total cost of stories available
  - Customer picks functionality
    - developers compute release date
- Set order
  - Respect dependencies
  - Working system first
  - Value first
  - Risk first
- Change control
  - Development may change order due to dependencies
  - Customer adds stories
    - either push back release, or reject other stories

XP Tracking

- Load factor = total time in last iteration / ideal time of stories completed
  - Divide available time by load factor to determine "ideal time" available for each iteration

XP Design

- System metaphor
  - Lightweight architecture
- CRC card design
- Do the simplest thing that could possibly work

XP Implementation

- Pair programming
  - Two developers coding at one workstation
  - Lightweight reviewing process
    - Spreads expertise around – switch frequently
  - Empirical data on success
    - 15% slower than 2 independent programmers
    - 50% fewer bugs!
- Refactoring
  - Once and only once
    - immediately refactor code to eliminate duplication
  - Fix design problems immediately (“code smells”)

XP Testing

- Test first
  - Write tests before code
  - Implement only enough code to pass the test
    - Don’t write code if you don’t need it right now—you may never need it!
- Automation
  - All unit tests should be automated
- Quality standards
  - Add a unit test whenever a bug is discovered
  - Only check in code when 100% of test cases pass

XP Process Improvement

- Meet at the end of each iteration
- Discuss issues, tweak process

Outline

- Team Software Process
- Extreme Programming
- **Comparison**

Comparison

- TSP
  - Lots of documents
  - Requirements document
  - Formal design
  - Reviews
  - Write tests first
  - Earned value
- XP
  - Just the code & personal knowledge
  - User stories, frequent customer interaction
  - Informal CRC design, constant refactoring
  - Pair programming
  - Write tests first
  - Load factor

Agile Manifesto

Individuals and interactions over processes and tools

Working software over comprehensive documentation

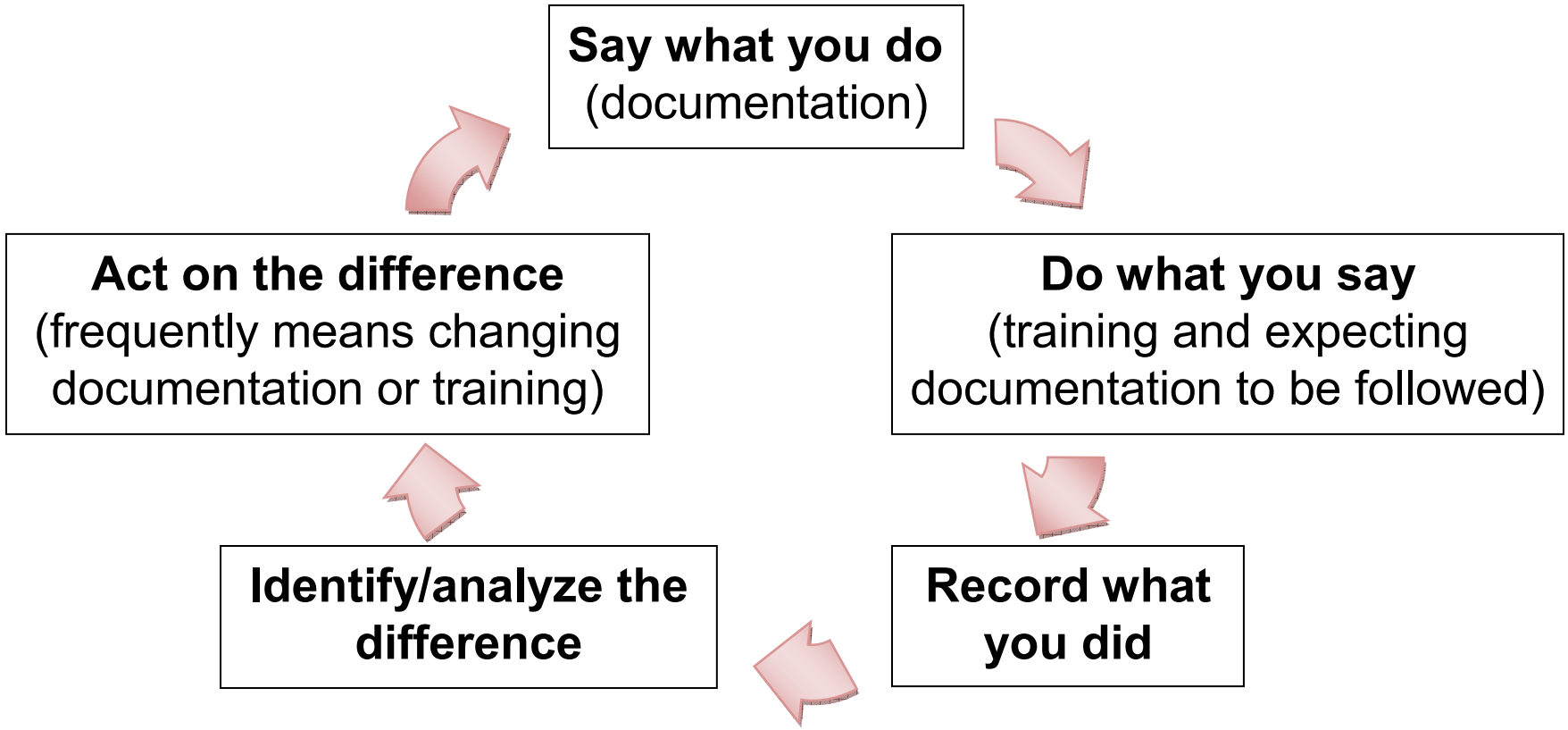
Customer collaboration over contract negotiation

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.

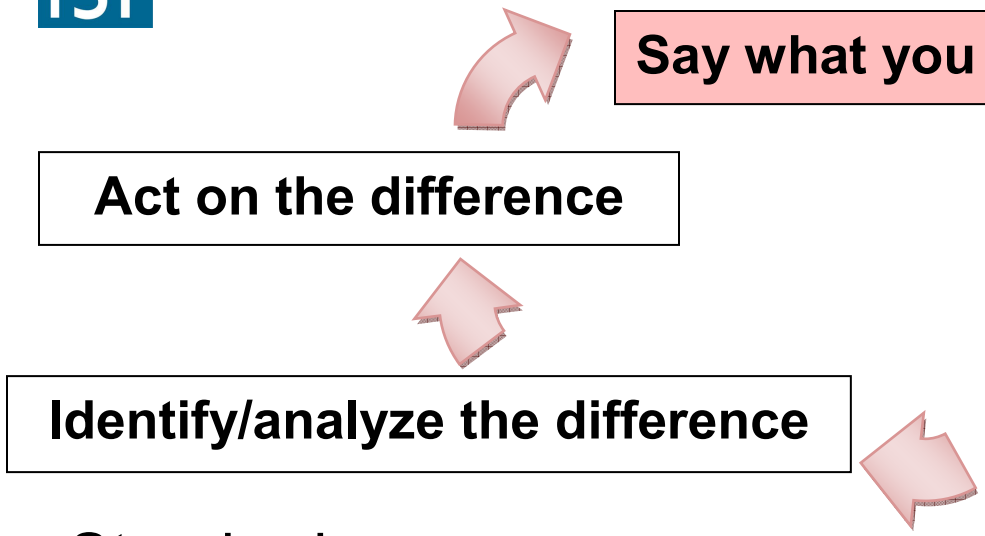
Continuous improvement loop

[source: Larry Maccherone]



Say what you do

[source: Larry Maccherone]

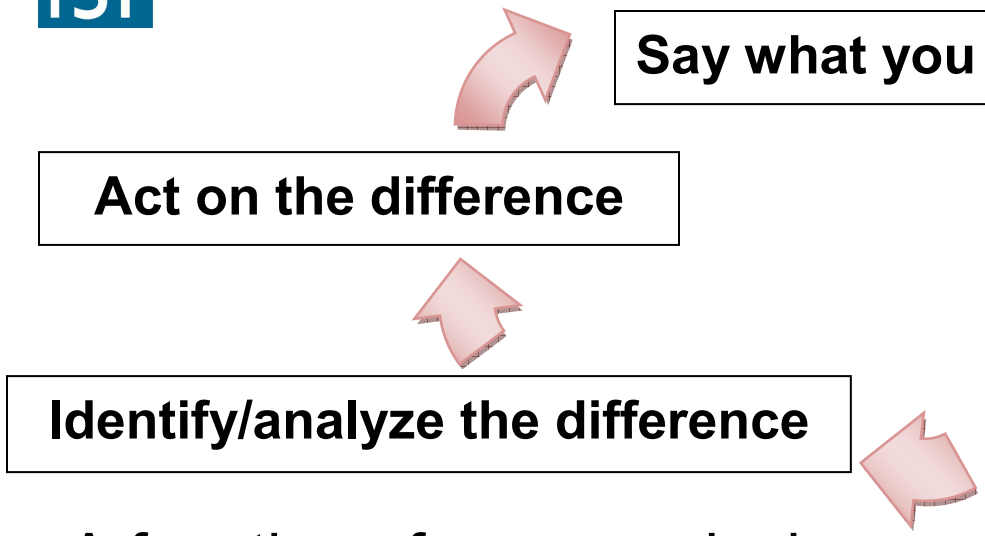


- Standards
- Procedures/work instructions
- Training manuals
- Checklists
- Measurement information model
- Key metrics and target values

Specifying desired outcomes in standards, checklists and key measures is better than trying to spell out every step in scripts, procedures or work instructions.

Do what you say

[source: Larry Maccherone]

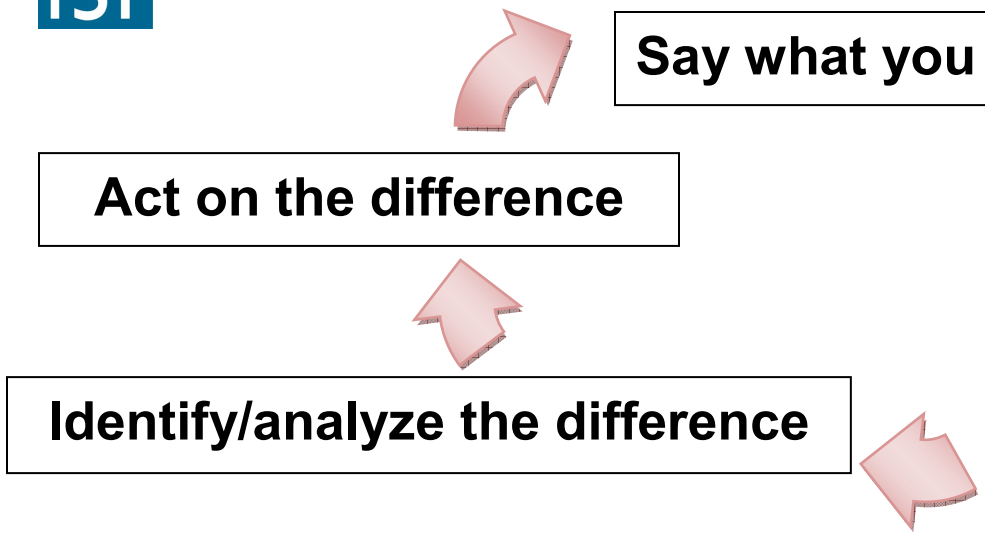


- A function of process design
- Training not just on process steps but on underlying reasons
- Internal and external audits
- Driven by conviction and understanding

You can't impose discipline from the top down. The process and training program must be designed to enable it to emerge from the bottom up.

Record what you did

[source: Larry Maccherone]

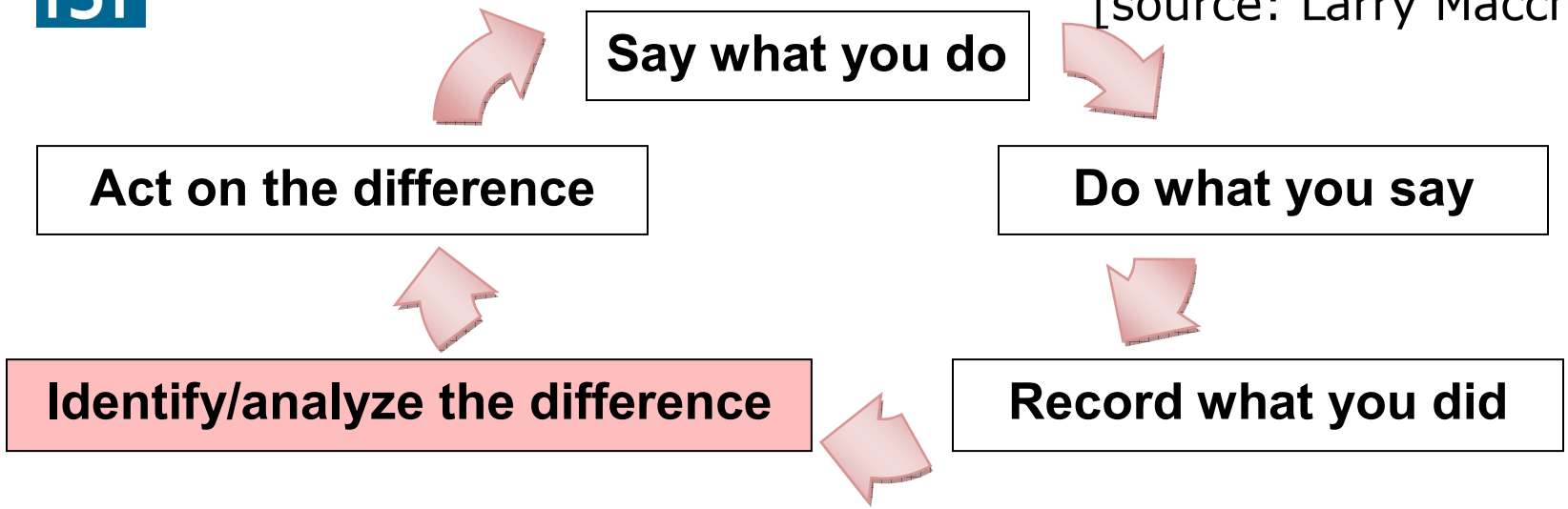


- Measurements captured in process
- Samples gathered post-process

Passive data capture is preferred over requiring manual input. If manual input is required, user must believe in value of data and should get immediate pay-back if possible.

Identify/analyze the difference

[source: Larry Maccherone]

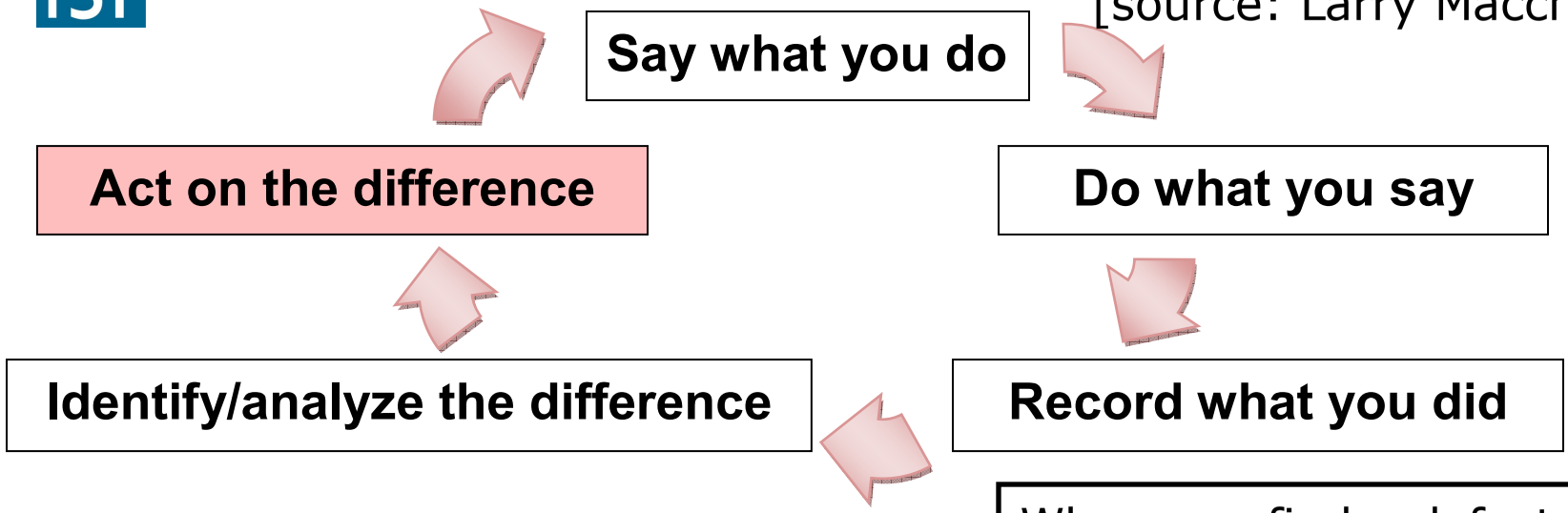


- Difference between "Say what…" and "Do what…"
- Identified in-process (defects, non-conformance reports) and…
- Retrospectively by looking at output of "Record what..."

In software, this is where statistical process control is often misapplied.

Act on the difference

[source: Larry Maccherone]



- How to prevent future deviations
- Change documentation, metrics, etc.
- Training – best when it's passive "training" like IDE feedback

When you find a defect:
1.Fix it
2.Find/fix others like it
3.Catch similar again in future
4.Prevent injection in future
Valid for process defects