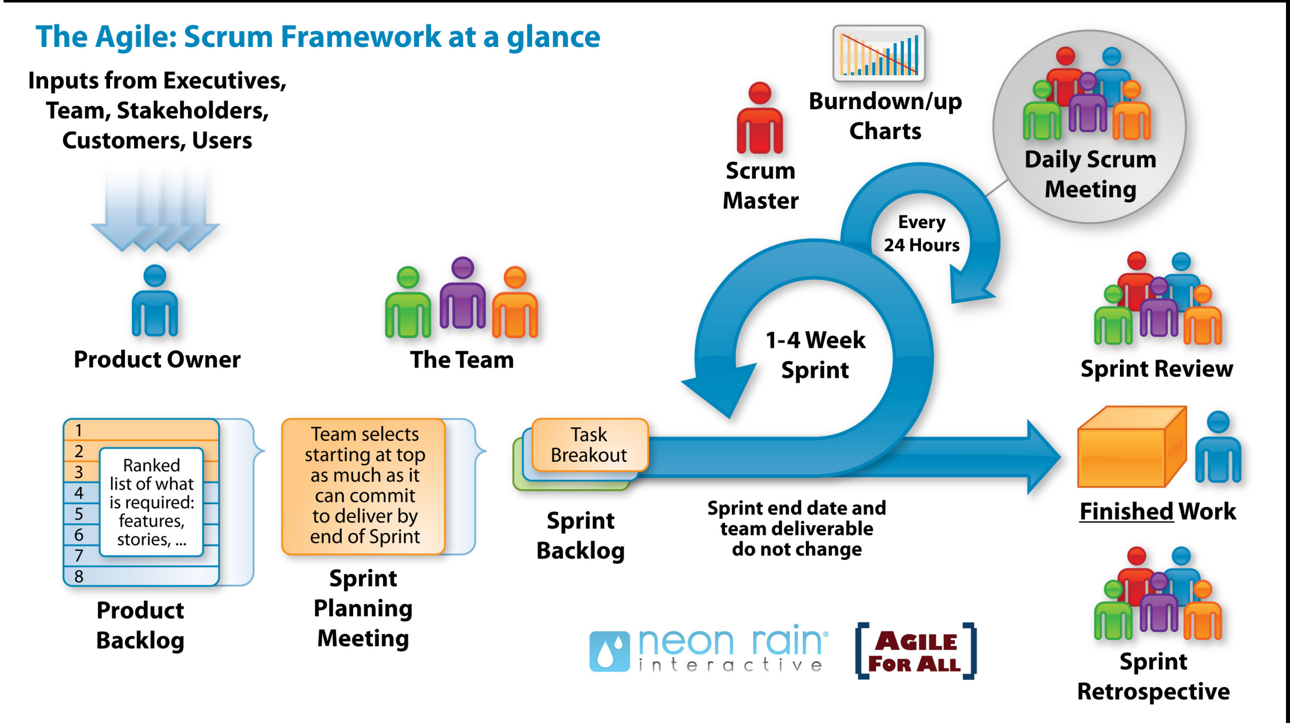
agile software development

The Agile Manifesto

- *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.*

Scrum

- Product owner
- Scrum master
- Daily scrum (<15 minutes)
  - What have you done since yesterday?
  - What are you planning to do today?
  - Any impediments/stumbling blocks?
- Sprints (7-30 days)
  - sprint planning meeting
  - sprint review meeting (the demo)
  - sprint retrospective

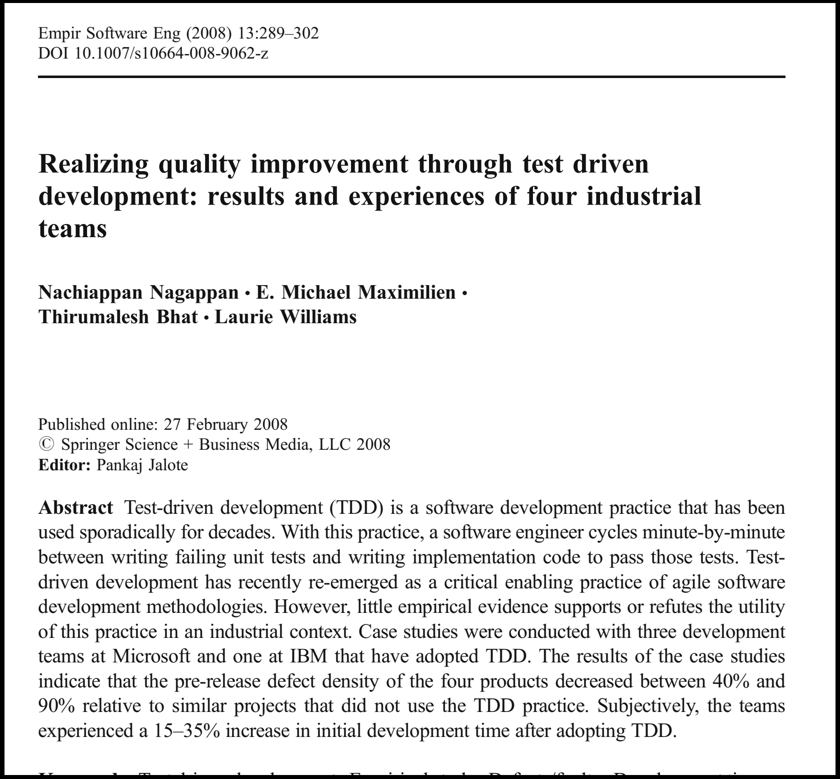

The Agile: Scrum Framework at a glance
Inputs from Executives, Team, Stakeholders, Customers, Users
Product Owner
The Team
Scrum Master
Burndown/up Charts
Daily Scrum Meeting
Every 24 Hours
1-4 Week Sprint
Sprint Review
Finished Work
Sprint Retrospective
1 2 3 4 5 6 7 8
Ranked list of what is required: features, stories, ...
Product Backlog
Team selects starting at top as much as it can commit to deliver by end of Sprint
Sprint Planning Meeting
Task Breakout
Sprint Backlog
Sprint end date and team deliverable do not change
neon rain interactive
Agile For All

Scrum in CS4560/5560

- Two scrum meetings per week
  - Capture three items for everyone in the meeting log
  - Other elements of the meeting log:
    - Who, when, where, what
- One sprint (milestone) once every week or two weeks

Test-Driven Development (TDD)

- Related to test-first programming in extreme programming
- Tool support: JUnit (http://www.junit.org/) (Eclipse built-in)



Empir Software Eng (2008) 13:289–302
DOI 10.1007/s10664-008-9062-z

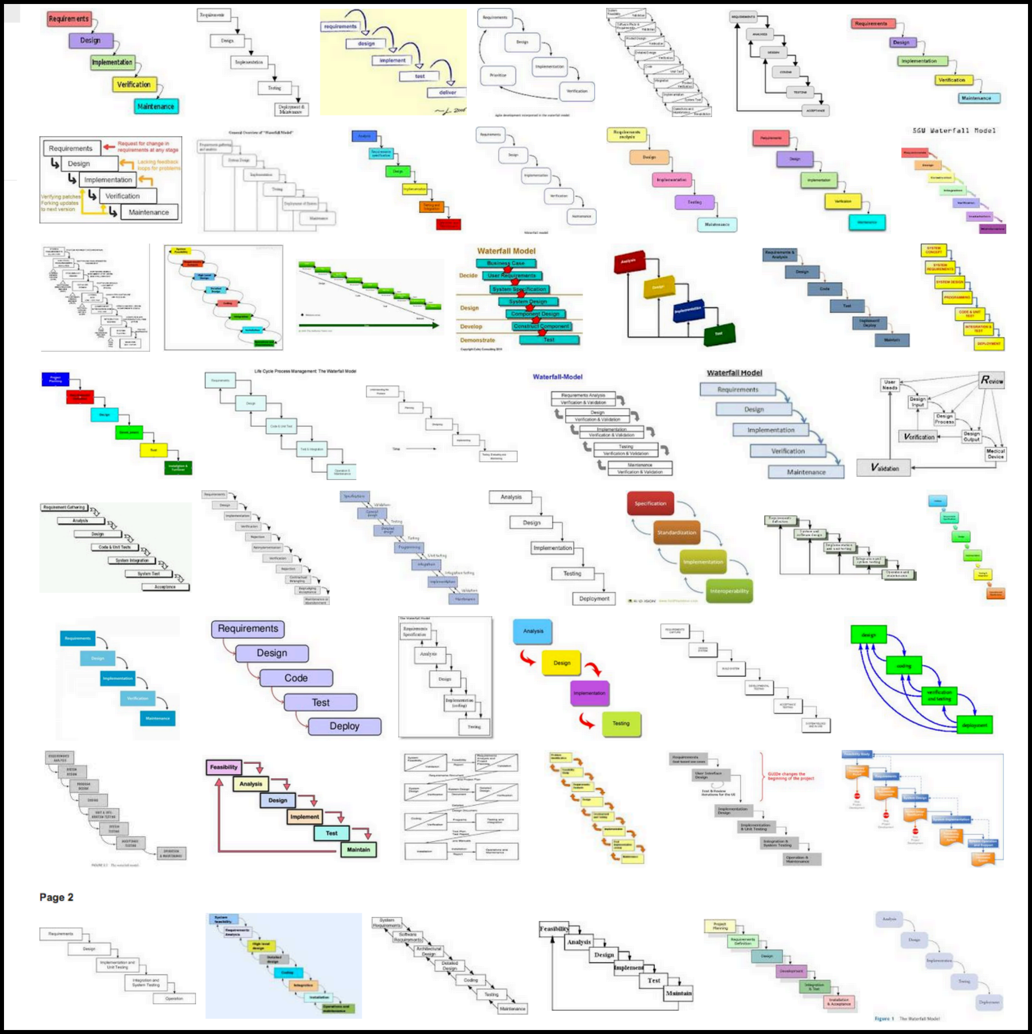
Realizing quality improvement through test driven development: results and experiences of four industrial teams

Nachiappan Nagappan · E. Michael Maximilien · Thirumalesh Bhat · Laurie Williams

Published online: 27 February 2008
© Springer Science + Business Media, LLC 2008
Editor: Pankaj Jalote

Abstract Test-driven development (TDD) is a software development practice that has been used sporadically for decades. With this practice, a software engineer cycles minute-by-minute between writing failing unit tests and writing implementation code to pass those tests. Test-driven development has recently re-emerged as a critical enabling practice of agile software development methodologies. However, little empirical evidence supports or refutes the utility of this practice in an industrial context. Case studies were conducted with three development teams at Microsoft and one at IBM that have adopted TDD. The results of the case studies indicate that the pre-release defect density of the four products decreased between 40% and 90% relative to similar projects that did not use the TDD practice. Subjectively, the teams experienced a 15–35% increase in initial development time after adopting TDD.

The Waterfall Model



The Unified Software Development Process

- The main process to cover this quarter
- Object-oriented
- Iterative
- Use case-centered
- Supported by UML (The Unified Modeling Language) http://www.omg.org/spec/

SOFTWARE PROCESSES ARE SOFTWARE TOO

Leon Osterweil

University of Colorado Boulder, Colorado USA

1. The Nature of Process.

The major theme of this meeting is the exploration of the importance of .ul process as a vehicle for improving both the quality of software products and the the way in which we develop and evolve them. In beginning this exploration it seems important to spend at least a short time examining the nature of process and convincing ourselves that this is indeed a promising vehicle.

We shall take as our elementary notion of a process that it is a systematic approach to the creation of a product or the accomplishment of some task. We observe that this characterization describes the notion of process commonly used in operating systems-- namely that a process is a computational task executing on a single computing device. Our characterization is much broader, however, describing any mechanism used to carry out work or achieve a goal in an orderly way. Our processes need not even be executable on a computer.

It is important for us to recognize that the notion of process is a pervasive one in the realm of human activities and that humans seem particularly adept at creating and carrying out processes. Knuth [Knuth 69] has observed that following recipes for food preparation is an example of carrying out what we now characterize as a process. Similarly it is not difficult to see that following assembly instructions in building toys or modular furniture is carrying out a process. Following office procedures or pursuing the steps of a manufacturing activity are more widely understood to be the pursuit of orderly process.

The latter examples serve to illustrate an important point-- namely that there is a key difference between a process and a process description. While a process is a vehicle for doing a job, a process description is a specification of how the job is to be done. Thus cookbook recipes are process descriptions while the carrying out of the recipes are processes. Office procedure manuals are process descriptions, while getting a specific office task done is a process. Similarly instructions for how to drive from one location to another are process descriptions, while doing the actual navigation and piloting is a process. From the point of view of a computer scientist the difference can be seen to be the difference between a type or class and an instance of that type or class. The process description defines a class or set of objects related to each other by virtue of the fact that they are all activities which follow the dictated behavior. We shall have reason to return to this point later in this presentation.

For now we should return to our consideration of the intuitive notion of process and study the important ramifications of the observations that 1) this notion is widespread and 2) exploitation of it is done very effectively by humans. Processes are used to effect generalized, indirect problem solving. The essence of the process exploitation paradigm seems to be that humans solve problems by creating process descriptions and then instantiating processes to solve individual problems. Rather than repetitively and directly solving individual instances of problems, humans prefer to create generalized solution specifications and make them available for instantiation (often by others) to solve individual problems directly.

One significant danger in this approach is that the process itself is a dynamic entity and the process description is a static entity. Further, the static process description is often constructed so as to specify a very wide and diverse collection of dynamic processes. This leaves open the distinct possibility that the process description may allow for process instances which do not perform "correctly." Dijkstra makes this observation in his famous letter on the GOTO statement, [Dijkstra 69] observing that computer programs are static entities and are thus easier for human minds to comprehend, while program executions are dynamic and far harder to comprehend and reason about effectively. Dijkstra's point was important then and no less significant now. Processes are hard to comprehend and reason about, while process descriptions, as static objects, are far easier to comprehend. Finally it is important to also endorse Dijkstra's conclusion that our reasoning about process descriptions is increasingly useful in understanding processes as the descriptions are increasingly transparent descriptions of all processes which might be instantiated.

In view of all of these dangers and difficulties it is surprising that humans embark upon the indirect process description/instantiation/execution approach to problem solving so frequently. It is even more startling to observe that this approach is successful and effective so often. This suggests that humans have an innate facility for indirect problem solving through process specification. It is precisely this innate ability which should be able to propel us to become far more systematic and effective in the development and evolution of computer software. What currently stands most directly in our way is our failure--to date--to understand our most central and difficult problems in terms of the process description/instantiation/execution paradigm.

© 1987 ACM 0270-5257/87/0300/0002$00.75