

Position Paper for the OOPSLA 2003 Workshop

“Commonalities of Agile Methodologies”

Jens Coldewey
Coldewey Consulting
Curd-Jürgens-Str. 4
D-81739 München
Germany
Tel: +49-700-26533939
Fax: +49-89-74995703
email: jens_coldewey@acm.org
<http://www.coldewey.com>

Human Issues in Agile Processes

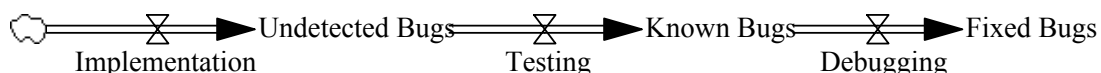
Looking at the different agile methods from a systems thinking perspective, there is one important observation you can make for all of them. They all concentrate on establishing feedback loops rather than trying to predict the project right from the beginning. Some examples of feedback loops are:

- Regular meetings or workshops to establish the scope for the next chunk of work together with the client (e.g. *Sprint Planning Meeting* in Scrum, *Planning Game* in XP, *Product-Tuning Workshops* in Crystal Clear, *Project Reviews* in DSDM)
- Regular events to adjust the methodology (e.g. *Learning Sessions* in ASD, *Methodology-Tuning Workshops* in all Crystal Methods)
- Technical feedback loops to determine the progress of the work (e.g. *Test Cases* in XP, *Automated Tests* in Crystal Clear)

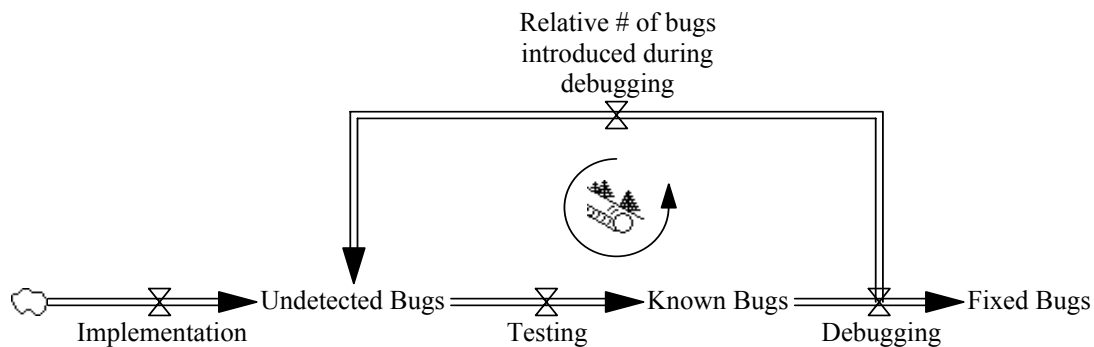
All of these feedback loops have in common that they don't measure the progress of the project against a given target, such as a plan. Rather these loops are un-biased measurements of the current state of the project. This reflects the approach of agile development to „control the development of software by making many small adjustments, not by making a few large adjustments“ like Kent Beck puts it.

It is one of the general ideas of management driven by systems thinking that it is usually wiser to control a complex system, such as a project, not by trying to predict it's outcome and react on divergence from a given target but by understanding the underlying dynamics and trying to push the „small lever to move the world“, adjustments that have a multiplying (or to be precise „non-linear“) effect on the outcome. I want to explore this idea a little bit deeper discussing the *Automated Tests* loop.

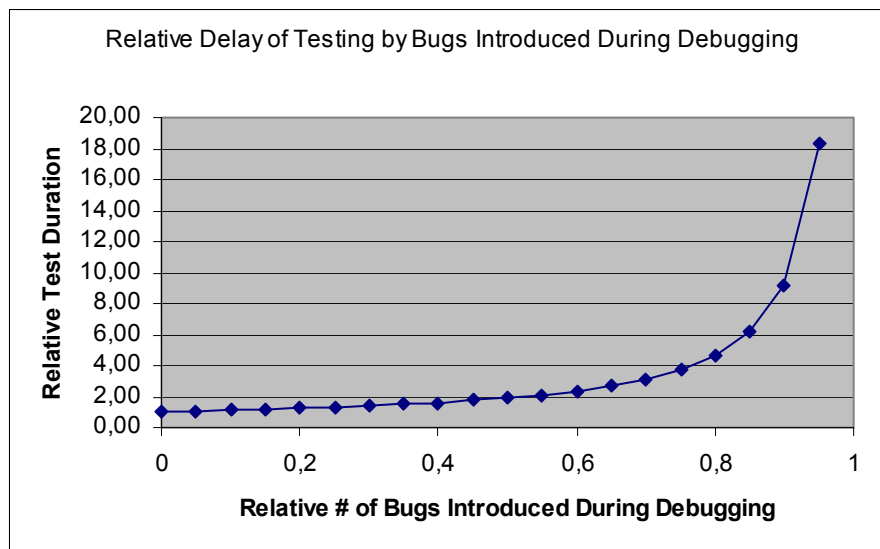
The following diagram shows a „classic“ view on fixing bugs: As you implement your software you inevitably make mistakes which sneak as undetected bugs into your product. To detect these bugs you use testing, which tries to uncover as many bugs as possible. As soon as a bug is known, someone in the teams has to fix it during debugging. In this model, there are only linear dependencies: The time the whole process needs seems to depend linearly on a set of factors, such as number of bugs you pipe in (implementation quality), testing productivity, and debugging productivity. Classic processes try influence these factors by measures to avoid mistakes such as up-front analysis, up-front design, and reviews, and measures that try to raise productivity, usually using the inadequate tool of overtime.



Often enough this approach fails. The reason is that this model ignores a small but important detail: It is not only implementation that produces bugs, but any programming activity. Particularly debugging is error prone, if the programmers are not used to change existing code without screwing it up. The next figure shows a more complete model of this process:



This feedback is a destabilizing loop as depicted by the little avalanche icon. These loops are non-linear, as the graph below shows. This graph is the result of a set of simulations done on a simplified system dynamics model of a project. If you introduce a new bug with every second bug fix, the test duration doubles. If your rate is even worse, the test duration starts to escalate, the project suffers from severe overrun and eventually crashes. Automated Testing concentrates on the non-linear feedback loop in the model. It reduces the relative number of bugs introduced during debugging. The strategy used ensures, that this „evil“ factor decreases over project time rather than ignoring it — the surest strategy to experience a blow-up. Instead of using complicated and expensive efforts to control the linear factors, it uses a cheap and pragmatic technique to control the non-linear feedback loops.



This is a general strategy common to all agile methodologies. Understanding these strategies might help to understand why agile development works despite of it's sometimes radical neglect of traditional techniques of software engineering.