

# Software Development is a Special Case of Maintenance

Chris Verhoef

University of Amsterdam, Programming Research Group  
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

x@wins.uva.nl

Usually organizations make major investments in automating their business processes in order to stay competitive, or to deliver better service to the public. Such investments often reflect the long-term strategy of these organizations not just the development of the initial software. Many initial decisions will have pervasive consequences in the decades to come. The use of a two-digit year field is such a decision: it has all the characteristics of having far reaching (negative) consequences. There are many more such decisions: the choice of the hardware, operating system, languages, and what have you. Such decisions ultimately affect the modifiability of the ensuing software system.

There is another important dynamic masking the problems concerning the modifiability of software. People are natural learners, so the competence of a maintenance team of a particular system tends to grow over time. This increase in learning possibly masks the deterioration of the code itself. If the quality attribute maintainability is itself not maintained, a system deteriorates due to the addition of faults and performance inefficiencies. Furthermore, design integrity breaks down, documentation is not kept current, and coding style becomes unfathomable. One measure of maintainability is the so-called ripple effect, the number of separate code areas that have to be changed to effectuate a single modification to the code. One hardware manufacturer studied this ripple effect to modifications to its operating system. They found that each modification led to about 300 other modifications. In the words of Gerald Weinberg: this operating system was equivalent to a nuclear reactor, one that was on the verge of turning into a nuclear bomb [3, p. 237, 243].

The phenomenon of software deterioration has been empirically measured by Belady and Lehman [1] and was formulated in their famous Laws of Software Evolution. Some software managers do not know that aging software becomes brittle (this can be due to the competence/masking dynamic mentioned earlier). Based on earlier modification experiences, their demands for rather sophisticated enhancements increase whereas the potential to do so is decreasing when time passes. Pushing the maintenance crew harder will ultimately result in a productivity collapse and high turn over rates.

No matter the care that has been taken towards the initial development process and the decisions with respect to hardware, software, and languages used, some of these choices will turn out to be less optimal. For instance, using the above-mentioned operating system, for the hardware manufacturer

stopped support for their products. If the software is not as successful as was anticipated when development started, the software will retire and be forgotten. If on the other hand the software is a success we often see a “software mitosis”, that is, multiple versions of the software come into existence, each with their own special features. This often leads to an explosion of customer disruptions, which in turn cause a dramatic productivity collapse of the development and maintenance teams. One possible solution for multiple versions, is to establish a base line software system from which an entire product-line can be derived, so that the multiple versions do not digress and start being systems on their own, with all the maintenance burden that comes with them. But this is mostly an after-the-fact observation, and large investments are necessary to migrate a multiple version system to a core asset architecture that serves as a product-line from which the multiple versions can be inferred.

Technology to help modifying such large amounts of software in an automated fashion becomes more and more a necessity in the IT industry. It is our experience that many of the software modifications are so company and/or system specific that no-one can expect to buy an off-the-shelf tool for the specific task. Therefore, a lot of research is necessary to set up architectures that can handle the construction of tools that aid in the difficult tasks that maintenance and software renovation require. This should be done in a cost-effective way.

After 25 years of experience in the telecommunications area, Ivar Jacobson learns us in his textbook on use cases that after the first release, the real development starts. For, a system normally develops through changes incorporated in new versions. Seen from this perspective, new development is only a special case, the first version [2].

**Note** This one page abstract serves as a primer of a keynote address to the 3rd Annual IASTED International Conference on Software Engineering and Applications, October 6–8, 1999 in Scottsdale, Arizona, USA.

## References

- [1] B.L. Belady and M.M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976.
- [2] I. Jacobson, Christerson M, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering – A Use Case Driven Approach*. Addison-Wesley, 1992.
- [3] G.M. Weinberg. *Quality Software Management: Volume 1 Systems Thinking*. Dorset House, 1992.