An Essential Distinction of Agile Software Development Processes Based on Systems Thinking in Software Engineering Management

Peter Wendorff
ASSET GmbH
Am Flasdieck 5
46147 Oberhausen, Germany
+49 208 628 7220
P.Wendorff@t-online.de

ABSTRACT
In the late 1990s a number of so-called "agile" software development methods have been proposed to overcome problems experienced with more traditional methods. These new agile methods have led to a controversal discussion within the software engineering community. This has illustrated the need for an integrating theoretical framework to clarify the distinguishing aspects of agile software development methods. In this paper we use general systems theory to characterise software development methods. We demonstrate that general systems theory allows a clear and meaningful characterisation of essential aspects of agile methods.

Keywords: Software management, systems thinking, agile software development

1 INTRODUCTION

General systems thinking was originally proposed in the 1950s as an analytical paradigm to stress the common foundations of different scientific disciplines like biology, psychology, the social sciences, etc. [11] Since those beginnings systems thinking has become an established perspective on the management process in complex organisations [6], [12], [13], [14].

One goal of systems thinking in management is conceptual understanding of the structure and behaviour of complex organisations. The benefits of systems thinking can be illustrated using the infamous Brooks' Law, "Adding manpower to a late software project makes it later." [4, p. 25] Brooks admission that stating this law he was "oversimplifying outrageously" points to its limited applicability. There is some evidence that the simplicity of Brooks' Law may have elevated it into a kind of managerial "mantra" that may even be misleading [10]. Contrary to this oversymplifying form of Brooks' Law, a systems thinking perspective suggests that adding the right people to a late project early enough can even save the project [1].

Adding the right people to a late project early enough can save a project. If managers fallaciously apply Brooks' Law and do not introduce appropriately qualified new staff into the project team early enough, they probably do commit an error. This is a common error of judgement among managers [10]. This error is often disastrous because later in the project the likelihood increases that Brooks' Law applies, making it much more difficult to save the late project. The important issue to notice in this situation is the long time distance between the error and its harmful consequences. In large software projects this time span may be months. In these complex organisational systems time delays are a major factor that prevents managers from understanding a project's dynamics, possibly leading to wrong decisions.

Agile software development methods address many common problems in software projects where cause and effect may be separated in time considerably. In this paper we assume a systems perspective and look at the assumptions, problems, and solutions that have informed agile software development methods.

2 A SYSTEMS PERSPECTIVE ON SOFTWARE DEVELOPMENT PROCESSES

A common systems perspective on software development is shown in Figure 1 [14]. This system consists of three subsystems, namely Controller System, Software Development System, and Customer System. The Software Development System has Requirements and Resources as major inputs, and Software as major output.

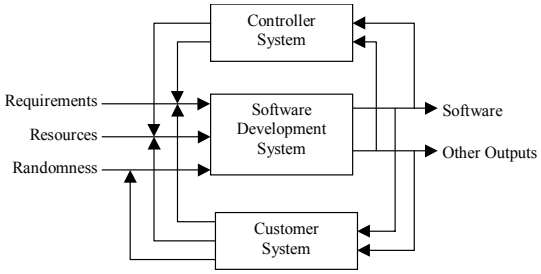

Figure 3: A Systems Perspective on Software Development [14] (Weinberg, 1992, Figure 11-4)

Systems

A system can be regarded as "a way of looking at the world." [15, p. 52] This definition stresses the point that systems are often abstract models that are purposefully constructed by researchers in order to acquire knowledge. In Figure 1 a system is depicted that is supposed to produce software.

Subsystems

Figure 1 shows an important concept of systems thinking, namely the subdivision of a larger system into smaller subsystems. The larger system comprises all that is

shown in Figure 1, which is subdivided into three smaller subsystems, namely Controller System, Software Development System, and Customer System. The use of the term subsystem is a matter of perspective, because any subsystem is also a proper system, but the term stresses that the system in question is part of a larger system under investigation.

Environment
No interesting system exists in isolation, it is rather embedded in a larger system called "environment". The environment of a particular system basically means all that is outside of that system [15].

Flows
A system receives inputs from its environment, and it supplies outputs to its environment. In social systems these inputs and outputs are mostly flows of information [6]. For example, in Figure 1 the Customer System has Software (e.g. information about the software's actual functionality) as one input, and it produces Requirements (e.g. information about the software's desired functionality) as one output.

States
Subsystems process inputs in order to produce outputs. Often a system can be in different states. The outputs of such a system are not only dependent on the inputs, but they also depend on the system's state. The state of a system is by definition not directly accessible from outside of that system, and that makes it difficult for an external observer to understand its behaviour [14]. In Figure 1, for example, important state variables of the Customer System may be the availability and qualification of its members, and obviously these variables will vary over time.

Delays
The state of a system acts as a memory and can decouple inputs and outputs in time. This can generate dynamics that result in time delays between input changes and corresponding output changes of a system [6]. Delays can cause two effects that are often undesirable. First, the system needs more time to respond to changes. Second, delayed flows of information result in outdated information, and outdated information may lead to inadequate decisions.

Both of these undesirable effects of delays can be illustrated referring to Brooks' Law and Figure 1. The first effect can be observed when new staff is added to the Software Development System. The output of this subsystem will not increase immediately, instead the output will probably rise over time as the new members become more familiar with the project. The second effect may be witnessed if, for example, managers assess the Software Development Process on the basis of delayed progress reports. In that case managers might wrongly add manpower to a project that had been late but has just catched up, simply because they rely on outdated decision information.

3 AGILE SOFTWARE DEVELOPMENT

So-called "agile" software development methods share a core of values and principles published as the "Manifesto for Agile Software Development" on the World Wide Web [2]. Extreme Programming (XP) [3], [8], the Crystal Methodologies [5], and Adaptive Software Development (ASD) [7] are some popular agile methods.

Division of Labour
Division of labour into smaller, simple, routine, and well-defined tasks by managers is the classical management approch to improving the productivity of personnel. The tasks are matched to formally defined roles, which are performed by specialised personnel. For each task the necessary inputs as well as the required outputs are defined, and the task is completed successfully when the actual outputs match the required outputs.

Traditional management may suggest to segregate the definition of a software development process and its subsequent execution into two separate tasks that are carried out by different specialists. This approach is, for example, reflected in "The Unified Software Development Process" [9], where the definition of the process may be assigned to specialised "process engineers".

From a systems perspective traditional division of labour may have two competing effects. First, increased specialisation usually results in increased efficiency of the specialised subsystems. Second, increased specialisation leads to an increase in the number of subsystems to coordinate, and that will usually delay the response of the whole system to changes in its environment.

Agile software development favours "individuals and interactions over processes and tools" [2]. This clearly reflects the priorities of agile software development. At first, individuals should define appropriate processes to support their work, and only then these processes should guide the behaviour of the individuals. In this sense, agile software development rejects the separation of process definition and process execution, instead it suggests an integrative approach to these two activities. This means that the people who use the process, continuously develop and refine the process definition during the execution of the process itself.

Advocates of agile software development assume a volatile environment, where the processes must adapt to different needs frequently and quickly. Then the classical management approach to segregate process definition and process execution may intolerably delay this adaptation. Given that situation, a systems perspective supports the integrative approach to process definition and process execution that is suggested by agile software development.

Information Supply
Figure 1 shows subsystems that are bonded by flows that mostly represent information. The functioning of the subsystems critically depends on their inputs, and therefore these subsystems will usually make provisions to ensure this information supply. A subsystem typically has little control over its environment, and as a consequence, it is uncertain that the required information can be obtained from the environment when needed.

Traditional management often uses so-called "buffering

strategies" to ensure stability of critical inputs to a system from the environment [12]. One possible procedure of that kind is stockpiling of resources that are needed as input of systems. This approach is followed by most traditional software development methods, that produce and maintain detailed documentation in parallel to the source code of the executable computer programs. These documents serve as stocks of information that are used to decouple interdependent subsystems in time and space.

From a systems perspective traditional buffering strategies may have two competing effects. First, buffers decouple chains of interdependent subsystems and thereby ensure the smooth functioning of the whole system. Second, buffering procedures lead to stocks, stocks represent state variables, and state variables introduce delays into systems.

Agile software development favours "working software over comprehensive documentation" [2]. This illustrates two issues in agile software development. First, the number of information stocks is reduced. Second, the size of the remaining information stocks is reduced. The problems that can arise from huge information stocks can be illustrated in the case of written documentation in a software project. Correct documentation is often very helpful, but outdated documentation can be extremely harmful. The larger the amount of documentation becomes, the more effort is needed to find the required information, and the more effort is needed to keep the information up to date.

One alternative to buffering is closer integration [6]. Agile software development also calls for "customer collaboration over contract negotiation" [2]. Referring to Figure 1, this can be viewed as closer integration of the Customer System and the Software Development System. Customer collaboration improves the flow of information from the customer to the software developers, and therefore it reduces the need for buffering. Complex contracts are in a way huge stocks of critical information, and accordingly, one aim of agile software development is to reduce the size and complexity of these stocks as well.

In general, agile software development avoids large stocks of information, instead it relies on closer integration of subsystems.

Advocates of agile software development assume a volatile environment, where the inputs to subsystems are volatile. In such a situation the classical management approach of buffering may introduce intolerable delays. Given that situation, a systems perspective supports the restrictive approach to documentation that is suggested by agile software development.

4 CONCLUSION

Agile software development methods have been proposed for an organisational environment that is characterised by change and uncertainty. One aim of these methods is to enable a fast response of the software development process to changes in the given situation.

The application of systems thinking to agile software development methods focuses attention on the role of delays in software engineering processes. Delays can have two undesirable consequences. First, they can lead to late decisions, that will usually be suboptimal. Second, they can lead to wrong decisions, that may well be disastrous.

In this paper we have identified two sources of delays in software development processes, that are generally addressed by agile methods.

The first source of delays is the separation of process definition and process execution, that is a frequent practice in software engineering management. Contrary to this, agile software development methods propose an integrative approach toward process definition and process execution.

The second source of delays are buffering strategies, that are widely used in software engineering management. Contrary to this, agile methods aim at reducing the number, size, and complexity of buffers, and they call for closer integration of subsystems instead.

We think that these two issues are essential and distinctive ideas informing agile software development methods.

REFERENCES

1. Abdel-Hamid, T., Madnick, S.E. *Software Project Dynamics: An Integrated Approach*, Prentice-Hall 1991.
2. Agile Alliance Web Site: *Manifesto for Agile Software Development*. On-line at: http://agilemanifesto.org/
3. Beck, K. *Extreme Programming Explained: Embrace Change*, Addison-Wesley 2000.
4. Brooks, F. P. *The Mythical Man-Month*, Addison Wesley Longman 1995.
5. Cockburn, A. *Agile Software Development*, Addison-Wesley 2001.
6. Gharajedaghi, J. *Systems Thinking: Managing Chaos and Complexity*, Butterworth-Heinemann 1999.
7. Highsmith, J. A. *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*, Dorset House Publishing 2000.
8. Institute of Electrical and Electronics Engineers: *Dynabook on Extreme Programming*. On-line at: http://computer.org/seweb/dynabook/Index.htm
9. Jacobson, I.; Booch, G.; Rumbaugh, J. *The Unified Software Development Process*, Addison Wesley Longman 1999.
10. McConnell, S. C. Brooks' Law Repealed, *IEEE Software*, Nov./Dec. 1999, pp. 6-8.
11. Schermerhorn, J. R. *Management*. Wiley, 2001.
12. Scott, W.R. *Organisations: Rational, Natural, and Open Systems*, Prentice-Hall 1998.
13. Senge, P. N. *The Fifth Discipline: The Art and Practice of the Learning Organisation*. Doubleday Books 1994.
14. Weinberg, G.M. *Quality Software Management (Volume 1): Systems Thinking*, Dorset House Publishing 1992.
15. Weinberg, G.M. *An Introduction to General Systems Thinking*, Dorset House Publishing 2001.