Characterization of the Waterfall Process Model Adopted by the Japanese Software Industry

Osaka Gas Information System Research Institute Co., Ltd. Ryuichi HOSOYA*  
Hitotsubashi University Taro KAMIOKA**  
Osaka Gas Information System Research Institute Co., Ltd. Lan ZHANG*

Abstract: This paper identifies some key characteristics of the implementation of the waterfall process model of software development that is in adoption by the Japanese software industry. It emerged in the hierarchical structure of the industry and evolved to overcome some of the shortcomings of the waterfall process model. Despite the common view of the waterfall model as being rigid, the Japanese version of it comes with flexibility and pragmatism. The element that adds these to it is called “suri-aware,” which is a method of rapidly resolves mismatches between any two stages in the process.

Keywords: Software process model, outsourcing, waterfall

INTRODUCTION

Since late 1990’s, the waterfall process model (the waterfall model, henceforth) of software development process has been accused of failing many software development projects. It is commonly understood that the waterfall model lacks adaptability to uncertainties of requirements, effort distribution is far from even and the cost of change (CoC) rises exponentially over time (Royce, 1998; Boehm, 1981). Although in an attempt to fix the defects of the waterfall model, there has been proposed many extensions and replacements to it, the waterfall model still remains to be the most popular process model (ITPro, 2003; Cusumano et al., 2003).

Although the industries of the United States and Japan use the same term “waterfall model” to describe the type of process they adopted, they didn’t originate from the same root. The software industry of the United States adopted the waterfall model because it was adopted as part of the acquisition standard by Department of Defense (DoD) and other governmental organizations of the United States. The Japanese software industry adopted it in a different background.

From 1950’s to 1970’s, Japan’s former Ministry of International Trade and Industry (MITI) played a central role in forming the computer and software industry structure. Under MITI’s strong influence, the Japanese computer and software industry formed hierarchies, each of which is dominated by a major domestic elec-
tric manufacturing company such as Fujitsu, Hitachi, Toshiba, NEC, and so on. Most of the major system integration projects were lead by those dominant companies and work was split and given to medium- and small-scale companies, whose typical relationship with the dominants was fixed and long term.

The purpose of this paper is to argue that the waterfall model was adopted by the Japanese software industry because it fits the industry’s hierarchical structure and it evolved to overcome the defects of itself described above.

For this purpose, the paper will 1) overview the waterfall model, 2) characterize it by comparing it with some alternative process models, namely Rational Unified Process (RUP) and Extreme Programming (XP), 3) identify the typical software project setup in Japan and 4) describe the adaptation of the waterfall model in the Japanese software industry and the way it alleviates some of the shortcomings of the waterfall model.

OVERVIEW OF THE WATERFALL MODEL

The historically most popular software process model is known as the waterfall model (Royce, 1998). In a software development that is based on it, the process is divided into a sequence of stages such as system/software requirements, analysis, program design, coding, testing and operations (including transition) (Figure 1) (Royce, 1970).

![Diagram of the waterfall model](image)

**Figure 1 - The waterfall model**

Due to the model’s sequential nature, coding cannot take place before requirements and design are completely elaborated. However, under the situation where requirements are uncertain up-front, as typically observed in development of user-interactive software, the waterfall model may lead to unpredictable fallback from the coding or testing stage to an upstream stage such as requirements, which would in turn causes unpredictable rework that involves several upstream and downstream stages. The primary reason for this to happen is that the uncertainties in requirements wouldn't reveal until coding and/or testing is done and the program can be executed to exhibit its functionality, look-and-feel and performance. This is time when the project members realize that the design and the actual program have failed to meet the requirements, if at all. When they do, they must identify which parts of the results of which stages failed and require a fix. When the result of a stage has been fixed, the correspondent parts of the results of all succeeding stages have to be fixed, too. It is this characteristic of the waterfall model which tends to cause unpredictable and excessive addition of development effort caused by rework in the downstream stages.

CHARACTERIZATION OF PROCESS MODELS

Several fixes to the waterfall model has been proposed by a number of software engineers and researchers. The spiral model (Boehm, 1988) allows the process to gradually mature requirements by going through several iterations of the waterfall model stages. RUP (Kruchten, 1999) is a sophistication of the spiral model. It expands the spiral model by defining the four-phase iterative process with a set of concrete milestones as a guideline to determine when to stop iteration in one phase and move on to the next phase. With RUP, CoC over time is supposed to rise linearly rather than exponentially as with the waterfall model (Kroll, 2001).

Agile processes such as XP are also iterative. XP claims that the CoC over time rises only slowly (sub-linearly) by having the software under development
always tested and right and keeping artifacts that requires maintenance other than source code and test scripts to absolute minimum (Beck, 2000).

As one can see from these discussions, CoC is an important characteristic of a process model. Figure 2 compares the waterfall model, RUP and XP in terms of how CoC changes over time.

![Figure 2 - Cost of change over time](image)

Effort distribution is another characteristic of a process model. Figure 3 is an adoption from (Boehm et al., 2000); it compares effort distribution characteristics of comparable phases of the waterfall model and RUP. As described above, uneven distribution of effort is a major defect of the waterfall model; however, as can be observed in Figure 3, effort distribution of RUP is also comparably uneven. Therefore, uneven distribution of effort is not unique to the waterfall model. A true shortcoming of the waterfall model is the sensitivity to the project size; variation in effort distribution from design stage and thereafter is rather large while RUP’s effort distribution is less affected by the project size. This makes prediction and control of effort distribution in a waterfall-based project more difficult since size estimation always incurs error and size itself can even vary as requirements change during the development process.

![Figure 3 - Comparison of effort distribution patterns: waterfall and RUP](image)

The primary source of differences between the waterfall and iterative models in these characteristics is the way they divide work (Figures 4 and 5). The former first identifies groups (or subsystems) of interdependent functionalities, each of which fulfills one or more requirements, of the system and then horizontally divides the work into stages (requirements, analysis, design, etc.). Each group of functionalities goes through one big sequence of these stages. Note that a stage can be performed by different team collaborating with other teams working on the same group. Coding and testing won’t be performed until the later stages, and therefore there will be delays in identification of risks, design breakage and requirements inconsistency. The latter, on the other hand, process goes phase-by-phase. Each phase goes through one or more iterations; each iteration stays relatively small and focused on a limited set of use cases (or, episodes or stories, depending on what particular iterative process model you like), and as one moves to the next iteration, he/she can incrementally deal with new use cases. Each iteration goes through a flow of tasks such as analysis, design, coding and testing; the iron rule here is that you end up with tested and executable code. This process setup has two advantages over that of the waterfall model:
1. stakeholders are exposed to a working prototype at the end of each iteration so they work out requirements and design issues rapidly, and 2. implementation issues that relate to the entire software can be dealt with earlier.
THE WATERFALL MODEL AND SURI-AWASE

Cusumano, et al. showed the possibility that software development projects in Japan, compared with those in other regions including India, the United States and Europe, exhibit high productivity and low post-production bug density while they adopting the waterfall model more than these projects in the other regions (Cusumano et al., 2003). A different survey indicates 82.6% of Japanese software development projects adopt the waterfall model (ITPro, 2003). So, what is the recipe of success of these projects? This paper argues that they achieve high performance by controlling uncertainties during development by using a method called “suri-awase” (meaning in Japanese to have groups of people interact closely and coordinate their results). In this paper, we prefer to use this Japanese word since there seems to be no word in English that matches it well in the meaning. The concept of suri-awase has already been formalized for the design process in Japanese automotive industry (Fujimoto, 2001), but it has not been discussed much for the software development process. In fact, suri-awase plays a central role in a typical Japanese software development project (Kamioka et al., 2006). Suri-awase allows a project to rapidly elaborate premature requirements to control uncertainty while using the waterfall model.

SETUP OF SOFTWARE DEVELOPMENT PROJECT IN JAPAN

In Japan, a software development project typically involves members from the information systems department of the user company to validate requirements and deliverables, and members from a software company, such as the primary contractor, a subcontractor, a sub-subcontractor, and so on, who commit to the deliverables (Figure 6).

![Figure 6 - Trade structure of Japanese software industry](image)
es.

Under the project setup described in this section, the waterfall model is the desirable choice because stream-wise division of work is consistent with the stage-wise process setup of the waterfall model. For example, subcontractors C1 and C2 can do the work for Teams T and U in Figure 4, respectively, while the primary contractor assigns its internal resources for Teams X and Y.

REQUIREMENTS PREMATURITY AND SURI-AWASE

According to the IEEE 830-1998 standard (IEEE830-1998 1998), “a properly written SRS (software requirements specifications) limits the range of valid designs, but does not specify any particular design.” The same goes for any two development stages of a development process: information provided by a stage as specifications limits the range of valid output of its succeeding stages, but does not specify any particular output of it. In other words, specifications should drive the design activities of a stage, but should not be prohibitive of it.

In the earlier sections, we used to term “premature requirements” without defining it. In this paper, it means that requirements or their specifications do not satisfy one or more of the following eight criteria: correct; unambiguous; complete; consistent; ranked for importance and/or stability; verifiable; modifiable; traceable. These are the goodness criteria of software requirements specifications as defined by the IEEE 830-1998 standard.

In our experience, most software requirements specifications are premature to some extent throughout the development process. The survey that the author conducted in Japan in 2006 shows documents with incompleteness and/or ambiguity are commonly used in domestic software development projects. In the survey, 45.0% of 311 respondents, most of which are from a subcontractor company, answered those incomplete and ambiguous specifications documents are seen in more than 60% of the projects they had participated in (Table 1). This implies that a development process not only designs the software, but at the same time develops requirements even after the requirements stage has completed. It is the fundamental assumption of suri-aware that requirements and design affect each other and therefore must be co-developed.

Table 1 - Adoption rate of ambiguous and incomplete specification document in software development projects

<table>
<thead>
<tr>
<th>Question: How often do you see incomplete or ambiguous specifications documents in projects?</th>
<th>0-29% of the time</th>
<th>30-59% of the time</th>
<th>60-100% of the time</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection items</td>
<td>63 (20.3%)</td>
<td>108 (34.7%)</td>
<td>140 (45.0%)</td>
<td>311</td>
</tr>
</tbody>
</table>

Under the subcontracting setup based on the waterfall model as described in the above sections, there always exists asymmetry of information lies between the user company and the primary contractor, and between the primary contractor and subcontractors. The asymmetry of information is caused by two mechanisms induced in the subcontracting setup; they are called knowledge gaps (KGs) and information gaps (IGs) (Hosoya et al., 2006).

Figure 7, adopted from (Hosoya et al., 2006), depicts a typical setup of a subcontracted software development project with respect to what group holds what knowledge (the left-hand side) and information (the right-hand side), respectively. In this setup, one can observe there are gaps between two sets of knowledge (knowledge gap or KG) as follows:

- **KG1** - The user company knows what software should be built, but doesn’t know what kind of software is feasible.
- **KG2** - The primary contractor knows what software can be built, but doesn’t fully know how to actually design it; they, at least in part, depend on subcontract-
tors to come out with software design.

In other words, one party knows what they need, but doesn’t know about its feasibility. Knowledge gaps bring an advantage to project setup: allowing knowledge gaps in the project setup allows specialization by each party in a particular knowledge area and division of responsibility and tasks among parties.

A knowledge gap creates an information gap, or IG. An information gap is a discrepancy between what is explicitly specified and how the specifying party wants it to be made. Figures 7 and 8 also depicts two information gaps:

- **IG1** - The user company identifies some of their requirements, but the primary contractor doesn’t know all of the requirements. Hidden requirements are yet to be elicited.
- **IG2** - The primary contractor provides software requirements specifications to the subcontractors, but they do not include all the necessary specification details; also given specifications are not fully verified.

These gaps must be filled sooner or later; otherwise the development process would fail to deliver the right product.

Asymmetry of information introduced by KGs and IGs causes agency costs. The survey cited above also shows that 51.1% of the same respondents answered that they include 10 to 29% room for rework in the estimated effort presented to their clients and 29.6% of them answered that they do 30% or more (Table 2). Effort estimation that includes a priori efforts for rework may serve as a factor of increase in total cost of development. Precaution should be taken to always keep symmetry of information minimal (but not too low because otherwise increased cost of rigid elaboration of requirements and design would overwhelm the cost increasing effect of asymmetry of information).

### Table 2 - Rework estimation by contractors

<table>
<thead>
<tr>
<th>Question: What you estimate efforts, on average how much of the total work effort would you predict to be spent for rework?</th>
<th>0-9%</th>
<th>10-29%</th>
<th>30-59%</th>
<th>60% or above</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection items</td>
<td>I don’t know</td>
<td>30</td>
<td>159</td>
<td>83</td>
<td>9</td>
</tr>
<tr>
<td>Numbers of respondents selected</td>
<td>30 (9.6%)</td>
<td>159 (51.1%)</td>
<td>83 (26.7%)</td>
<td>9 (2.9%)</td>
<td>311</td>
</tr>
</tbody>
</table>

The primary means of working out the problem of asymmetry of information between two parties in the subcontracting setup that has been discussed is suri-awase. Suri-awase is driven by prematurity in requirements. Typically requirements specifications contain settled (matured) requirements that wouldn’t need adjustment and premature ones that need refinement and verification. With suri-awase, refinement and verification won’t happen in the requirements stage, but will rather take place in the succeeding design stage (Figure 9).

Premature requirements need to go through coordination with the design details or otherwise they are unverifiable. As design stage progresses, requirements are interwoven between the design details and the following events will occur for the premature parts of the
requirements:

- **Discrepancies of requirements with respect to design and/or technical constraints reveal and need to be resolved.** For example, the requirements say "Any screen that shows search results must show at least 10 results in such a way that user can view them without having to scroll," but as the design stage goes the designer concluded that with some screens this cannot be satisfied unless the size constraints for those screens are changed;

- **The design demands some details of requirements left incomplete to be completed.** For example, for Screen A, the optional types of data that may be printed for employees in the search result table aren’t specified because they depend on how many types can be processed within the given search time constraints; this cannot be determined until database’s production design is done and search performance simulation is possible.

These events trigger suri-aware between the requirements and the design stages, and therefore by the primary contractor and a subcontractor. The result is refined and verified requirements.

As a result of suri-aware between the requirements and the design stages, the amount of information or specifications related to requirements might change as they are elicited and refined. This effect of suri-aware is measurable using the Function Point Analysis method which is a standard maintained by International Function Point Users Group (IFPUG) for measuring the amount of functionality contained in requirements as function points. In most cases of suri-aware, function points of requirements should increase; therefore the project can anticipate future increase in workload by tracing variation in function points.

If the software development process were strictly sequential, asymmetry of information has no way to be resolved, hence the need for interaction between the requirements and the design stages to coordinate their results. Suri-aware does this by setting up a session where members involved in related stages rapidly coordinate requirements and specifications to generate a match between them. During the session, contents of requirements and specifications may be modified radically. The primary goal here is to seek the optimal quality of product with as less compromise as possible. The session typically lasts from several hours to several weeks.

Information gaps and suri-aware can occur between any two stages of the process, not just between the requirements and the design stages as illustrated above in detail. Take any three consecutive stages S1, S2 and S3 (S1 being upstream followed by S2 and S3), then suri-aware between S2 and S3 would cause changes in the result of S2, and this change in turn may necessitate further suri-aware between S1 and S2. The development process will, however, complete eventually since each time suri-aware is conducted, information gaps between two stages get reduced.

In fact, the process with suri-aware is compatible with the waterfall model as proposed by Walker Royce in (Royce, 1970). When someone states "With the waterfall model, you can never go through the stages backwards," he/she refers to the “the stagewise model" (Boehm, 1988; Benington, 1956). Royce’s model is an enhancement to the stagewise model with the addition of a backward path between any two consecutive stages (Figure 1). Suri-aware takes advantage of the backward path. Also notice that suri-aware, although not proven, should alleviate exponential rise of CoC over time that is common in waterfall-based processes by rapidly coordinating prematurity of requirements and design. The result is an implementation of the
waterfall model that aims for high productivity and optimal quality of product while allowing alleviation of some of its shortcomings including uneven effort distribution and exponential rise in CoC.

**CONCLUSION**

It is the author's view that a process model is selected and adapted so that it fits the inter-organization- al trade structure and custom of software development.

The waterfall model has often been considered an inflexible, excessively rigorous approach. When it became an acquisition standard of DoD, some ideas that added flexibility to it, such as a fallback path to a preceding stage and a type of prototyping flow (Royce, 1998), has been neglected. The DoD version of the waterfall model with emphasis on rigorosity is an overkill for most software projects in private sectors. Iterative and agile processes such as RUP and XP can be seen as a counter-response to such a rigorous approach.

The Japanese software industry, on the other hand, has a different background. It had MITI instead of DoD. MITI successfully copied the hierarchical industrial structure of the manufacturing industry to it. MITI's initial focus was on how to grow computer hardware manufacturers and there is no trace of their attempt to standardize acquisition process of software. A waterfall-type of process used by Japanese computer and software companies emerged mostly spontaneously so that it fits the industry structure. Therefore the Japanese version of the waterfall model was never as rigorous as that of DoD. In fact, some aspects of flexibility and pragmatism that XP emphasizes can also be found in it: involvement of the user in requirements elaboration, keeping documentation minimum, colocation of the entire project teams, emphasis on informal communication, etc.

Suri-awase, which is interaction between two groups of project members responsible for different stages of the process, plays a central role in the implementation of the waterfall model by the Japanese software industry. It rapidly resolves mismatch between what is specified and what is really wanted. This mismatch can be understood as asymmetry of information, which causes agency costs, through a conceptual framework of knowledge and information gaps. By controlling information gaps, the hierarchical subcontracting setup of development projects overcomes late resolution of risks and design breakage and the uneven effort distribution which are inherent in the waterfall model.

Some might argue that asymmetry of information, requirements prematurity and some aspects of suri-awase should also be observed in software development projects conducted in countries other than Japan. In fact, the author is not to argue that each of these are unique to Japanese software industry. Through the survey described above, however, the author had an impression that a number of respondents that are experienced in Japanese software development projects see suri-awase as a useful means to drive projects toward success while adopting some version of the waterfall process model. To these respondents, XP or RUP is not necessarily a working replacement to the waterfall process model. This view of the relationship between suri-awase and the waterfall process model might be particular to Japan.

We have not studied well how suri-awase affects the exponential rise of CoC and sensitivity to project size incurred in the waterfall model. Although, in theory, suri-awase may alleviate these shortcomings, but we still lack the evidence. This issue remains as future work.

**REFERENCES**


