

# Knowledge maturing as a process model for describing software reuse

Hans-Jörg Happel, Andreas Schmidt  
FZI Research Center for Information Technologies, Karlsruhe  
{happel, andreas.schmidt}@fzi.de

***Abstract.** Software reuse has become a major topic in software engineering since reusing artifacts has an important effect on the cost and quality of software products. Accordingly, understanding and managing the mechanisms of software reuse is important for every organization that develops software. In this paper we describe a knowledge maturing process in software engineering and use it to analyze two cases of reusable software artifacts. We argue that the "reusability" of a software artifact is not just an immanent property of the artifact itself, but depends on the "maturity" of the knowledge the artifact embodies. We show that the notion of knowledge maturing can serve as a useful lens for understanding reuse processes and suggest further investigations towards a more holistic concept of reusability.*

## 1 Introduction

Software engineering is about systematic approaches to the creation of software applications [IEEE04a]. Since most applications are not totally new in general, but share a decent overlap in their functionality, software reuse deals with the "reuse of existing software or software knowledge to construct new software" [FK05].

The general process of software reuse is to identify generic functionality beyond a number of systems, either by pursuing a systematic analysis of the underlying subject matter (domain analysis) or by analyzing existing software for overlapping functionality. In both cases, the final step is to form a "reusable asset", which may be a piece of software or packaged experience [BCR94]. Thus, the underlying assumption is that software reuse is an engineering activity that ends with the completion of reusable artifacts, which may then be reused by various parties. Although there is a large body of knowledge regarding software reuse, the state of practice is not considered satisfactory [Szy02, MYAM99].

Thus, we propose a different view on software reuse in this paper. In contrast to the position of "reuse engineering", we describe software reuse

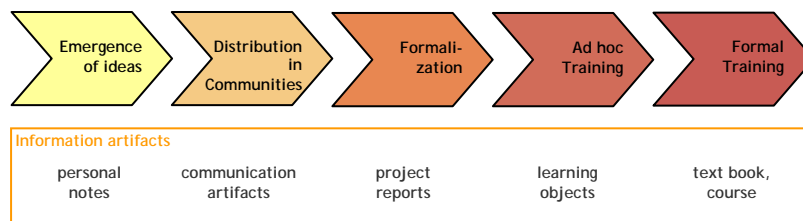
as a learning process. We argue that the probability of reuse (“reusability”) is not an immanent property of reusable artifacts, but depends on the maturity of the knowledge embodied. Reusable artifacts cannot be engineered in a generic way, but depending on the knowledge maturity of the subject matter, specific support can be given to raise the reuse of software artifacts.

To back this argumentation, we first describe a model for knowledge maturing in section two, which we afterwards apply on two cases of successful software reuse: Design Patterns and logging. Finally, we summarize and reflect the findings with respect to future work.

## 2 The knowledge maturing process

### 2.1 Idea of Knowledge Maturing

When comparing the approaches of knowledge management and e-learning to learning in organizations, Schmidt noticed that the conceptual incompatibilities (and the resulting organizational and technical conflicts) can be traced back to the different nature of knowledge they are dealing with [Sch05]. Accordingly, he conceptualized the so called “knowledge maturing process” which describes the nature of knowledge throughout different phases of evolution.



**Fig. 1:** The Knowledge Maturing Process as described in [Sch05]

The knowledge maturing process is divided into five phases (c.f. Figure 1). The first one is characterized by the *emergence of a new idea*. Several people identify a topic from different viewpoints that deserves further investigation. They start discussing informally, shifting into the phase of *distribution in communities*. In the process of discussion, a common terminology begins to arise, which can be considered a first important step of knowledge maturing. Subsequently, the community begins to establish regular communication channels, creating and exchanging first artifacts like project reports or documents (*formalization*). This step of knowledge

maturing opens up the field towards broader discussion and criticism by disseminating persistent thoughts to the community. This is also a prerequisite for others to pick up the topic and refine it didactically for *ad hoc training* purposes. When the topic is then well established and explicitly connected to related topics, it may end up within full-fledged courses in curricula (*formal training*), thus representing mature or “ripe” knowledge.

## **2.2 Applying Knowledge Maturing to Software Engineering**

This knowledge maturing process is not limited to the area of e-learning, but is intuitively applicable also to other fields that create artifacts by refining knowledge such as science in general or software engineering.

As a first example, we take the “typical lifecycle of components” as described in [Bos00]. It involves several steps from identifying functionality as a conceptual entity and modelling it as a subsystem to the incorporation of this functionality into the development infrastructure and maps quite smoothly into the knowledge maturing process, as illustrated in Table 1.

A similar argument is made by Henderson and Clark, who investigate the development process of physical products. They argue that there is little agreement about subsystems when new technologies arise. After a phase of experimentation, a dominant solution design emerges, which “incorporates a range of basic choices that are not revisited every time” [HC90]. This set of choices is called “architectural knowledge”, which is described as “stable” or “institutionalized”, once a dominant design is in place.

While these examples map quite well into the general phases of the knowledge maturing process from the e-learning context, there are some differences which can be traced back to an interdependency between the knowledge maturing over time, and the solution space of the product architecture, which narrows accordingly. “Mature” knowledge implies standardized interfaces and even dominant designs in the solution domain. Henderson and Clark show that in product development these interfaces have implications on the internal information processing of an organization, as well as on its market power. Several authors claim that the same is true for software products [BCK03, Cop99]. This argument will be supported by the logging case, described in the following section.

## **3 Reusing software knowledge: Two Case Studies**

While Bosch describes knowledge maturing in software engineering in an abstract way, we will now illustrate the concept by giving two concrete examples. First, we will describe the popular concept of “Design Patterns”

by means of knowledge maturing. Afterwards, the aspect of “logging” inside object-oriented software systems will be described in a similar manner. Both examples are no case studies in the traditional sense. As evidence for our argumentation, we primarily rely upon publicly available data, either from internet archives or publications.

### 3.1 The case of Design Patterns

#### 3.1.1. Short introduction

Design Patterns are no concrete software artifacts, but describe patterns of how to structure the static or dynamic interaction of components in object-oriented systems. The essence of a design pattern is its specific name, a problem setting in which a pattern can be useful and a concrete solution of how to apply the pattern in that context [GHJV95]. So design patterns are a kind of “best practice” knowledge, yielding characteristic structures (i.e. patterns) of interactions in concrete systems when applied.

#### 3.1.2. The evolution of the Design Patterns concept

**The roots:** The idea of design patterns stems from civil architecture where it was introduced by Christopher Alexander in the 1960s [Ale64]. He argues that technological progress rendered individual experience and craftsmanship an unsuitable source of design knowledge in an accelerating and complex world. Accordingly, he notes a lack of guidance for architects and engineers to come up with solutions to problems in a systematic way, and provides a list of patterns that capture experience and make it available in a standard form [Ale77].

**Identification:** Remaining a mere contribution to the discourse of architectural theory for some time, the concept of patterns was discovered by some members of the “object-orientation” sub-community of software engineering. Beck and Cunningham [BC87] applied it to the creation of user interfaces in the Smalltalk programming language. Some time later, Erich Gamma started his dissertation about how to develop a general theory of patterns in software engineering.

**Generalization:** Soon after, there were first attempts to bring together actors in the field to join forces on the new concept of Design Patterns. The group of Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides – later to be known as the “Gang of Four” – first met in 1990 to discuss creating a catalogue of design patterns.<sup>1</sup> In 1991, Bruce Anderson gave the

---

<sup>1</sup> <http://c2.com/cgi-bin/wiki?HistoryOfPatterns>

first workshop on patterns at the OOPSLA conference [And92]. In late 1993, Kent Beck and Grady Booch assembled a group of people interested in patterns at a mountain retreat which was the foundation of what is now known as the Hillside Group.<sup>2</sup>

**Formalization:** In the same year, Gamma et al. presented their progress at the ECOOP conference [GHJV93]. In 1994, the researchers in the field met at the first Pattern Languages of Programs conference (PLoP 1994<sup>3</sup>), an event that has been organized every year since then.

**Standardization:** In 1995, the book on Design Patterns was finally published by Gamma et al. [GHJV95]. Having a “textbook” at hand, soon a number of courses dedicated to the topic were created around the world.<sup>4</sup> The book was translated into several languages and is considered a seminal contribution to the discipline of software engineering.

**Institutionalization:** The general idea of patterns soon expanded to ideas like architectural patterns [BMR+96], analysis patterns [Fow97] and specific application areas like HCI [Bor01]. Education on patterns is institutionalized in software engineering curricula [IEEE04b] and well-engineered software systems are expected to make heavy usage of patterns.

### 3.1.3 Summary

It is not hard to fit the history of Design Patterns into the knowledge maturing process, described before (see Table 1). Besides the historical “facts” mentioned, Gamma et al. support this point explicitly in their book. With respect to what we would call the early stages in the knowledge maturing process they write: “Design patterns ... aren't the designs people tend to generate initially. They reflect untold redesign and recoding as developers have struggled for greater reuse and flexibility in their software.” [GHJV95]

Then the “Gang of Four” went on to refine and formalize the ideas and results of discussions to finally come up with their book: “So although these designs aren't new, we capture them in a new and accessible way: as a catalogue of design patterns having a consistent format.” [GHJV95]

Thus, they pushed the idea of Design Patterns, by providing a comprehensive work that is well-suited for reference and teaching. The knowledge, which had once been “discovered” and discussed by a small number of specialists, has become mature, and is nowadays available for a large number of software developers: “A designer who is familiar with such

---

<sup>2</sup> <http://hillside.net/history.html>

<sup>3</sup> <http://hillside.net/plop/pastconferences.html>

<sup>4</sup> see e.g. <http://www.ipd.uka.de/~tichy/entwurfsmuster.html>

patterns can apply them immediately to design problems without having to rediscover them.” [GHJV95]

## 3.2. Logging in Java

### 3.2.1 Short introduction

When developing software applications, debugging is an important activity to remove defects and raise the quality of the software product. While debuggers have some shortcomings (e.g. when dealing with distributed applications), embedding log statements that emit runtime information to the system console became a low-tech alternative. Log statements are recordable for auditing purposes and can be analyzed by system administrators for software maintenance [Gül04, Ham02].

### 3.2.2 The emergence of logging components

**Identification:** Thus, logging has become a best practice in software engineering and is widely used. In the Java environment, novice developers often use plain “System.out.println()” statements to generate logging information. However, when developing larger applications this soon becomes clumsy and inflexible. Accordingly, developers identified logging as a concern and started to encapsulate logging mechanisms in distinct classes, so that e.g. the output format can be easily changed.

In the late 1990s, some people began to work on more sophisticated logging components. One component that evolved from work in an EU project is the open source library “log4j”, which was published in early 1999 [Gül04].

**Generalization:** During that time several people discussed on that topic in order to find a standard solution, because there were “currently no general purpose logging or tracing APIs in the Java platform.” [Ham02] In December 1999, a “Java Specification Request” (JSR47) was approved, that was intended to push forward that topic [Ham02]. It “tried to draw from numerous sources and ... support a wide set of requirements.”<sup>5</sup> Some time later, at the end of the year 2000, the log4j library became part of the Apache Jakarta project, thus having a much higher visibility than before.<sup>6</sup>

**Formalization:** Meanwhile, the work on JSR47 made progress. While people from the log4j project were involved in its specification, they were not content with the direction the specification took. In June 2001, Ceki Gülcü, the lead developer of log4j, asked for support to “lobby Sun” for

---

<sup>5</sup><http://www.ingrid.org/jakarta/log4j/jakarta-log4j-1.1.3/docs/pub-support/GrahamHamilton.html>

<sup>6</sup>[http://www.theserverside.com/news/thread.tss?thread\\_id=3288](http://www.theserverside.com/news/thread.tss?thread_id=3288)

ensuring a higher level of compatibility to log4j [Gül01]. However, changes were refused by Sun which argued that the Java release in which JSR47 should be incorporated was already scheduled.

**Standardization:** In December 2001, the final approval ballot for JSR47 resulted in a majority of votes for that proposal, with the representative of the Apache Foundation voting against it. In February 2002, Java 1.4 was released as the first JDK coming with a logging functionality included (package java.util.logging). In May the same year, JSR47 was formally released as a specification.

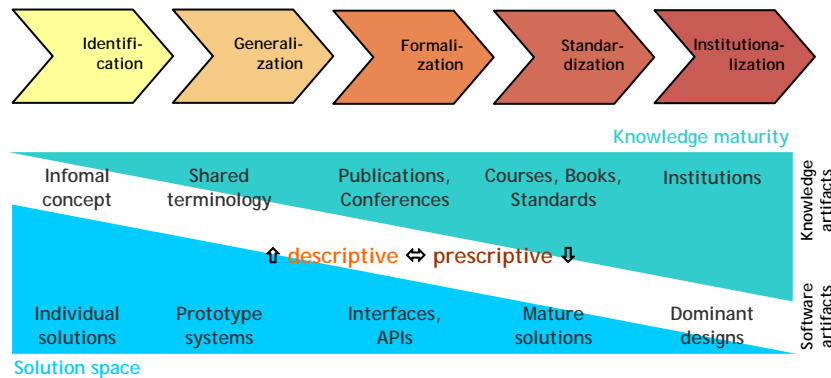
**Institutionalization:** Nowadays, by the inclusion of java.util.logging into the standard JDK development environment, Sun has made it a de-facto standard. While some major projects are still relying on log4j because of its advanced functionality, java.util.logging has also become a dominant logging component in the field.

### 3.2.3 Summary

From the initial idea, logging has become a mature component of Java systems, with stable implementations and textbooks [Gup03] available. Much like GUI components, logging libraries are widely accepted and used in a large number of software development projects.

## 4 Knowledge maturing in Software Engineering

When looking at the two cases of Design Patterns and logging described before, the knowledge maturing process in general seems to be a viable lens to analyze software reuse.



**Fig. 2:** The Knowledge Maturing Process in Software Engineering

In Figure 2, we slightly adjusted the names of the different phases, making them appropriate for our context of discussion. Second, we added the solution space dimension like introduced at the end of section two. One can interpret the process of system development as a decision process, narrowing the variety of the solution space to small number of dominant alternatives [HM03].

We argue that both dimensions – the solution space and the knowledge maturity – are mutually dependent. Looking at the cases in section 3 reveals that during the process of maturing, knowledge does not just get more “formalized” and “ripe”, but also undergoes a fundamental change in its impact. In the early stages of the maturing process, knowledge is derived descriptively from existing solutions: “Most of these designs have never been documented before. They are either part of the folklore of the object-oriented community or are elements of some successful object-oriented systems” [GHJV95].

However, once they are “ripe enough” and a formalized representation arises, the same knowledge turns to be prescriptive. While systems had been implementing patterns *by intuition* of the developers before, most systems that were created after that contain such patterns, because developers *have learned so* from their codification in textbooks and training material.

Phase	Identification	Generalization	Formalization	Standardization	Institutionalization
Software artifacts	Individual solutions	Prototype systems	Interfaces, APIs	Mature solutions	Dominant designs
[Bos00]	“functionality is identified as a conceptual entity and starts to become modelled as a subsystem”	“researchers ... start to develop prototype systems that generalize the functionality captured in a number of subsystems”	“commercial companies ... turn the prototype system into a product”	“the product ... is used by the majority of software engineers as part of software development.”	“the product is incorporated in the infrastructure ... [it] disappears as an independent part of software applications”
Logging	Developers creating own “Logging” or “Debugging” classes in their code	First detailed work on logging components; first release of log4j	Start of JSR47; Log4j becomes a Jakarta project	Release of JDK 1.4 with logging; approval and release of JSR47	JDK-logging is established; log4j is still in use
Design Patterns	Developers making explicit design decisions for reusability; first people discussing those issues	First publications on that topic; workshops at conferences; formation of Hillside Group	Start of the PLoP conference series; Publication of “Design Patterns”	University course and professional training; more books and expansion of the pattern idea to other subject areas	Design Patterns are core part of Software Engineering curricula and well-engineered systems

**Tab. 1:** Summary of the knowledge maturing steps in the example cases

Thus, knowledge artifacts and solution space are mutually dependent with changing directions in the course of knowledge maturing – increasing knowledge maturity coincides with a decreased solution space. While the software artifacts are constantly modified and improved in the maturing process, their usage by third parties also increases. We would argue that they can be called a “reusable” artifact, as soon as the usage exceeds the development activity.

## 5 Conclusion

In this paper, we applied the notion of knowledge maturing to the field of software engineering by analyzing the history of two kinds of reusable artifacts – Design Patterns and logging libraries. Both cases show, that the nature of knowledge constantly changed during their evolution.

We argue, that the conceptualization of software reuse as an engineering process with reusable artifacts as clearly defined output is too narrow. Especially in distributed settings, centralized, “factory”-style approaches are difficult to apply. Instead we described how software reuse can be understood as a learning process where knowledge artifacts are produced in the course of discussing prototype implementations. Later, those artifacts mature and in turn influence implementations by narrowing the solution space for the software systems produced.

Accordingly there are different phases of reuse, with a different “reusability” of the software artifacts. While usual conceptions of reuse finish when putting reusable artifacts into a library where potential users can find it, we argue that context-dependent support based on the level of knowledge maturity and the user’s knowledge might be more suitable to transfer knowledge about reuse opportunities.

It remains up to future work how to support this and how to conceptualize and measure knowledge maturity. Here, formal and informal learning methods derived from the original knowledge maturing process, incorporating both developers and users of reusable artifacts, are a promising starting point. Also, since we described two inter-organizational settings in section 3, the implications for intra-organizational reuse have to be investigated.

## References

- [Ale64] Alexander, C.: Notes on the Synthesis of Form. Harvard University Press, 1964.
- [Ale77] Alexander, C.: A Pattern Language. Oxford University Press, 1977.

- [And92] Anderson, B.: Towards an architecture handbook. In Addendum To the Proceedings on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '92) ACM Press, New York, NY, 167-168.
- [BC87] Beck, K.; Cunningham, W.: Using Pattern Languages for Object-Oriented Programs. In: OOPSLA-87 Workshop on the Specification and Design for Object-Oriented Programming.
- [BCK03] Bass, L.; Clements, P.; Kazman, R.: Software Architecture in Practice. Addison Wesley, 2003.
- [BCR94] Basili, V.R., Caldiera, G.; and Rombach, H.D.: The Experience Factory. Encyclopaedia of Software Eng., John Wiley&Sons, New York, 1994.
- [BMR+96] Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; and Stal, M.: Pattern-oriented Software Architecture, Volume 1: A System of Patterns. Wiley, 1996.
- [Bor01] Borchers, J.: A Pattern Approach to Interaction Design. Wiley, 2001.
- [Bos00] Bosch, J.: Design and use of software architectures: adopting and evolving a product-line approach. ACM Press/Addison-Wesley Publishing Co., 2000.
- [Cop99] Coplien, J. O.: Reevaluating The Architectural Metaphor: Toward Piecemeal Growth. In: IEEE Software 16 (1999), September/October, no. 5, pp. 40-44.
- [FK05] W.B. Frakes and K. Kang, "Software Reuse Research: Status and Future", in: IEEE Trans. on Softw. Eng., vol 31, no. 7, 2005, pp. 529-536.
- [Fow97] Fowler, M.: Analysis Patterns: Reusable Object Models. Addison Wesley, 1997.
- [GHJV93] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Abstraction and Reuse of Object-Oriented Design. ECOOP 1993: 406-431.
- [GHJV95] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
- [Gül01] Gülcü, C.: JSR47 vs. log4j. 06 2001. <http://www.ingrid.org/jajakarta/log4j/jakarta-log4j-1.1.3/docs/critique.html> (retrieved on 2006-10-06)
- [Gül04] Gülcü, C.: The Complete log4j Manual. <https://www.qos.ch/shop/products/log4j/log4j-Manual.jsp>. 2004 (retrieved on 2006-10-06)
- [Gup03] Gupta, S.: Logging in Java with the JDK 1.4 Logging API and Apache log4j. Apress, 2003.
- [Ham02] Hamilton, G.: JSR 47 Logging API Specification. <http://jcp.org/en/jsr/detail?id=47.2002> (retrieved on 2006-10-06)
- [HC90] Henderson, R. M.; Clark, K.B.: Architectural innovation: the reconfiguration of existing product technologies and the failure of established firms. In: Administrative Science Quarterly 35 (1990), pp. 9-30.
- [HM03] Herbsleb, J. D. ; Mockus, A.: Formulation and preliminary test of an empirical theory of coordination in software engineering. In: Proceedings of the 9th European software engineering conference (ECSE/FSE). ACM Press, 2003, pp. 138-137.
- [IEEE04a] IEEE Computer Society: Guide to the Software Engineering Body of Knowledge. 02/2004. <http://www.swebok.org> (retrieved on 2006-10-06)
- [IEEE04b] The Joint Task Force on Computing Curricula (IEEE/ACM): Software Engineering 2004 - Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering. <http://sites.computer.org/ccse/SE2004Volume.pdf> (retrieved on 2006-10-06)
- [MYAM99] Mili, A., Yacoub, S., Addy, E., Mili, H.: Toward an Engineering Discipline of Software Reuse. IEEE Software, vol. 16, no. 5, pp. 22-31, Sept/Oct, 1999.

- [Sch05] Schmidt, A.: Knowledge Maturing and the Continuity of Context as a Unifying Concept for Knowledge Management and E-Learning. In: Proceedings of I-KNOW 2005, Special Track on Integrating Working and Learning, Graz, June 2005.
- [Szy02] Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley, 2002.