# Checkable Code Decisions to Support Software Evolution

Martin Küster, Klaus Krogmann
FZI Forschungszentrum Informatik
Haid-und-Neu-Str. 10-14, 76131 Karlsruhe, Germany
{kuester,krogmann}@fzi.de

## 1 Introduction

For the evolution of software, understanding of the context, i.e. history and rationale of the existing artifacts, is crucial to avoid "ignorant surgery" [3], i.e. modifications to the software without understanding its design intent. Existing works on recording architecture decisions have mostly focused on architectural models. We extend this to code models, and introduce a catalog of code decisions that can be found in object-oriented systems. With the presented approach, we make it possible to record design decisions that are concerned with the decomposition of the system into interfaces, classes, and references between them, or how exceptions are handled. Furthermore, we indicate how decisions on the usage of Java frameworks (e.g. for dependency injection) can be recorded. All decision types presented are supplied with OCL-constraints to check the validity of the decision based on the linked code model.

We hope to solve a problem of all long-lived systems: that late modifications are not in line with the initial design of the system and that decisions are (unconciously) overruled. The problem is that developers will not check all decisions taken in earlier stages, and whether the current implementation still complies with them. Automation of the validation of a large set of decisions, as presented in this work, is a key factor for more concious evolution of software systems.

## 2 Decision Catalog

We developed an extensive catalog of recurring design decisions in Java-based systems. Some of the decision types are listed in Table 1. It lists the decision types and the associated constraints (in natural language) that are checked by the OCL interpreter.

Due to space restrictions, we cannot go into detail of each decision type. Elements of all object-oriented languages, such as class declarations including generalizations and interface implementations, are covered as well as member declarations, such as field and method declarations. To show that the approach is not restricted to elementary decisions in object-oriented systems, we give more complex decision types, such as wrapper exception or code clone. Especially code clones can be acceptable if the decision model records the intention of the developer that cloned the code.

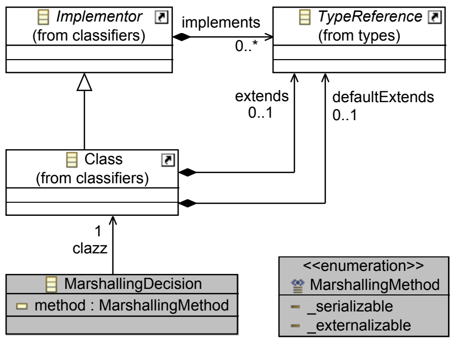

Figure 1: MarshallingDecision and related artifacts. Code Decision Metamodel (CDM) elements shaded in grey, Java code model elements shaded in white.

Important framework-specific decision types are left out from discussion: those for dependency injection (vs. constructor usage) and those for special classes (e.g. Bean classes). They require more complex linkage (not only to Java code, but also to configuration files in XML). The mechanism to state the decision invariant, however, is exactly the same.

Fig. 1 gives a model diagram of the decision type MarshallingDecision. The decision states what mechanism is used to marshal a class (using standard serialization or hand-written externalization).

## 3 Automatic Checks of Decisions

We propose a tight integration with models of Java code constructed in textual modeling IDEs. For that, we operate on the code not on a textual level, but on a model level based on *EMFText*[1]. This enables linkage between decision models and code models.

Typical difficulties of models linking into code, esp. dangling references caused by saving and regenerating the code model from the textual representation, are solved by "anchoring" the decision in the code using Java annotations. The reference is established by comparing the `id` of the anchor in the code with the `id` of the decision. This kind of linkage is stable even in the presence of complex code modifications, such as combinations of moving, renaming, or deleting and

[1]http://www.emftext.org/index.php/EMFText

| **Code Decision Element** | **Description of Constraint** |
|---|---|
| **Object Creation** | • objects of designated type are created only via the defined way. |
| **Inheritance / Abstraction** | • Class extends indicated class or one of its sub classes / is abstract |
| **Cardinalities and Order** | • Field is of the respective type (Set, SortedSet, List, Collection) |
| **Composition** | • Container class has a reference to part class<br>• Part class is instantiated and the reference is set within the constructor of container class or as part of the static initializer<br>• *(bi-directional case)* part class holds a reference to container class, too |
| **Field Initialization** | • all fields are initialized (*only!*) as defined |
| **Marshalling** *(Interface examp.)* | • Marshalled class must implement the specified interface |
| **Wrapper Exception** | • class $E$ must extend Exception<br>• methods containing code causing library exception must throw user-defined exception and must not throw library exception |
| **Code Clones** | • code was copied from indicated method according to clone type<br>• clones may differ no more than defined: exact clones must stay exact, syntactically identically may not contain modified fragments |
| **Utility Class** | • class must be `final`<br>• (empty) `private` constructor<br>• provides only static methods |
| **Singleton** *(Pattern example)* | • contains `private static final` field with self-reference<br>• contains a `public static synchr.` method getting the reference |
| **...** | • ... |

Table 1: Extract of the catalog of discussed code decisions.

re-inserting fragments.

The decision types are equipped with OCL constraints. These constraints use the linked code elements to check whether the defined design decision still holds in the current implementation. For example, given the MarshallingDecision from Fig. 1, the OCL will check if the class that is referenced by *clazz* (derived reference) implements `java.io.Externalizable` (if this method is chosen).

## 4 Related Work and Conclusion

The initial ideas of recording decisions during the design of object-oriented systems is from Potts and Bruns [4]. The process of object-oriented analysis is captured in a decision-based methodology by Barnes and Hartrum [1] capturing the argumentation of encapsulation or decomposition. For architectural models of software, the need to collect the set of decisions that led to the architectural design was first pointed out by Jansen and Bosch [2].

In this paper we presented a novel approach to model-based documentation of recurring object-oriented design decisions. We oulined an extract of our catalog of decision types in object-oriented systems. All decisions are equipped with OCL constraints. If applied to existing code, these types make it possible to check whether the defined decision still holds in the current implementation or if it is violated.

Currently, we are re-engineering a commercial financial software. This real-world case study helps to complete the catalog and evaluate the benefits of the model-based approach, which is checking, finding rationales and intent, and links to drivers of decisions, during the evolution phase.

## References

[1] P. D. Barnes and T. C. Hartrum. A Decision-Based Methodology For Object-Oriented Design. In *Proc. IEEE 1989 National Aerospace and Electronics Conference*, pages 534–541. IEEE Computer Society Press, 1989.

[2] A. Jansen and J. Bosch. Software Architecture as a Set of Architectural Design Decisions. In *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, pages 109–120. Ieee, 2005.

[3] D. L. Parnas. Software Aging. In *Proc. 16th International Conference on Software Engineering (ICSE '94)*, pages 279–287, 1994.

[4] C. Potts and G. Bruns. Recording the Reasons for Design Decisions. In *Proc. 10th International Conference on Software Engineering (ICSE 1988)*, pages 418–427. IEEE Computer Society Press, 1988.