A Theory for Software Design Extraction

Ben A. Sijtsma\textsuperscript{1} and Joseph W.L.J. Mager
Shell Research B.V.
Koninklijke/Shell-Laboratory, Amsterdam
P.O. Box 3003, 1003 AA Amsterdam, The Netherlands

Abstract

In this paper we consider the problem of extracting a design for an existing software system. This problem is clearly relevant to understanding and maintaining software systems. The basis of our approach is a formalization of top-down design with information hiding. This formalization allows an accurate and concise formulation of constructing an a posteriori design for a software system using only the source text. The theory predicts that in general there are many possible designs and that these designs can be partially ordered and form a lattice. The lattice has a smallest element and this element is the best top-down design in the sense that it has the most information hiding. We show how this element can be constructed.

A tool has been built on the basis of the theory developed here. It has been applied to many software systems, written in various programming languages. These applications have shown that the theory is successful and can be used for understanding, documenting, maintaining, and restructuring software systems, without requiring from the user a detailed knowledge of the underlying theory.

Keywords: Software Engineering, Reverse Engineering, Maintenance, Theory of Design Extraction.

1 Introduction

For software maintenance, as well as any other activity involving the understanding of a software system, it is important to know the underlying design. Unfortunately, the design of a system is often poorly documented. Moreover, the design evolves, or may even deteriorate, as the system is maintained, so, the original design documentation becomes obsolete. However, the maintainer needs to know the design so that he can safely adapt the system without introducing unwanted interactions or side effects.

In this paper we develop a theory for extracting the design from an existing software system using only the source text. The design that is extracted is a top-down design with information hiding [7]. It turns out that generally there are many possible designs and that these designs can be partially ordered and that they form a lattice [5]. The lattice has a smallest element and this element is the best top-down design in the sense that it has the most information hiding.

A great deal of research has been carried out in reconstructing or recovering the design. A common approach is to use the cross-reference information. In [4, 6, 8, 9] this information is used to define various (pseudo-)metrics, like “cohesion”, “coupling”,

\textsuperscript{1}to whom correspondence should be addressed
"alteration distance", and "similarity". These metrics are then used to partition the units into clusters. The initial hierarchy of clusters thus obtained is in some cases improved upon by merging certain clusters. In [3] domain information too is employed to recover the design. In our approach we also use the cross-reference information but the difference is that we do not define any metric but develop a formalization of top-down design instead. As a result of this formalization we are able to give a concise and formal description of design extraction.

The structure of this note is as follows. In Section 2 we explain our notation for and operations on relations. Section 3 deals with a formalization of top-down design with information hiding. Section 4 contains the formal statement of what we mean by software design extraction. In Section 5 some properties of structure extraction are given and in Section 6 we briefly consider the set of all possible top-down designs for a system. The construction of the best top-down design is given in Section 7. Finally, the conclusions are presented in Section 8.

2 Relations

In the following we will use relations quite a lot. For completeness's sake some conventional notations are introduced below.

A relation on a set D is a set of pairs in D × D, i.e. a relation is a subset of D × D. If R is a relation on D and a, b ∈ D, then a R b denotes (a, b) ∈ R. If it is clear from the context we will often omit "on D" in "a relation on D".

Note that relations are sets and hence usual set operations, such as ∪ and ∩, can be applied to them. The composition is another operation on relations. The composition of two relations, say R1 and R2, is denoted by R1 o R2, and is defined by R1 o R2 = {a, c : (∃b :: a R1 b ∧ b R2 c) : (a, c)}. We assume that o has higher binding power than ∪ or ∩, so A ∪ B o C means A ∪ (B o C).

Let R be a relation on D. Then Rn, where n is a natural number, is defined inductively by: R0 = {a : a ∈ D : (a, a)} and Rn+1 = R o Rn.

A relation R is transitive if and only if R2 ⊆ R. Note that R2 ⊆ R implies that Rn ⊆ R, for all n > 0. The transitive closure of a relation R, denoted by R+, is defined by R+ = (∪n : n > 0 : Rn). The transitive reflexive closure of a relation R, denoted by R∗, is defined by R∗ = R0 ∪ R+. Alternatively, R+ can be defined by R+ = R∗ o R = R o R∗.

The inverse of a relation R, denoted by R−, is defined by a R− b iff b R a. Some straightforward properties of the inverse are: (R1 o R2)− = (R2− o R1−), (R+)− = (R−)∗, and (R∗)− = (R−)*. In view of this we will write R− to denote either (R−)∗ or (R+)−; the same goes for R∗.

Let R be a relation on D and d ∈ D. The set d R is defined by d R = {a ∈ D : d R a : a}. Similarly, the set R d is defined by R d = {a ∈ D : a R d : a}, i.e. d R = R− d.

A relation R is cyclic if and only if (∃d :: d R+ d) and R is acyclic if it is not cyclic. Furthermore, a relation R on D is weakly connected if and only if (∃d :: (∀a : a ∈ D : d R+ a)). Such a d is called the root of R. A weakly connected relation R on D is a tree on D if the root has no parents and every element but the root has precisely one parent, i.e. if R root = ∅ and (∀a ∈ D : a ≠ root : (∃!b :: b R a)).