Chapter XIV
From Business Rules to Application Code:
Code Generation Patterns for Rule Defined Associations

Jens Dietrich
Massey University, New Zealand

ABSTRACT

Rules that define relationships between objects are an important part of the specifications of software systems. However, support for the explicit representation of those rules in modelling languages is still immature and there is little support to assist software engineers in implementing them. The result of this practice is hand-crafted and error-prone applications. In this chapter, we analyse some common patterns used to implement rules and discuss the shortcomings associated with those patterns. We then discuss several options to explicitly represent rules, and how to automate the generation of application code from rules.

INTRODUCTION

Object-oriented programming (OOP) has become the dominant programming paradigm over the last twenty years. OOP facilitates the modularisation and reuse of software, and supports the "divide and conquer" approach to master large and complex software engineering projects. Several modelling techniques have been proposed in order to support object-oriented software engineering, the most successful ones being the Unified Modelling Language (UML) (UML, 2004) and the Eclipse Modeling Framework (EMF) (Budinsky, Brodsky, Merks, Ellersick & Grose, 2003). As far as modelling is concerned, models such as UML diagrams provide a useful level of abstraction from the programming language used. While it was very common in the early years of object-

oriented programming and design to manually translate these models into programming language artefacts, code generation has become more and more common. The Model Driven Architecture (MDA) (http://www.omg.org/mda/) initiative of the Object Management Group (OMG) aims at embedding this practise into a conceptual framework that is based on the idea of model transformations.

In general, the gap between modelling and programming has narrowed significantly in recent years as the abstraction level of programming languages has gone up. Programming language features related to low level resource management like explicit memory management (destructors) have disappeared from modern languages and have been replaced by services provided by compilers, virtual machines or application servers. New features have been added to languages like Java and C# to make them more modelling like. For instance, annotations can be used to stereotype artefacts, assertions and test cases can be used to express constraints and therefore method semantics, and generic types facilitate the representation of one-to-many relationships.

However, there is one aspect where a deep conceptual gap between modelling languages and programming languages remains: relationships. While relationships are first-class citizens in modelling languages like UML, EMF and ER, they are not explicitly represented in modern programming languages. The state of the art is that relationships are manually coded. This is an error-prone process, and the hand-crafted code resulting from it is hard to maintain. In particular, it is very difficult to reverse engineer models from this code. Østerbye (2007) has compared this practice with "translating while loops into goto statements". The situation can be improved by generating code, and many case tools contain simple, template-based code generators for this purpose. There has been research into adding explicit relationships to programming language starting with Rumbaugh's Data Structure Manager (DSM) (Rumbaugh, 1987). A more recent approach to add relationships to the Java language is RelJ (Bierman & Wren, 2005). An alternative solution that has been explored is to add relationships libraries to Java (Østerbye, 2007) and AspectJ (Pearce & Noble, 2006). Code generation patterns for relationships have been investigated by Noble (1997) and Genova, del Castillo and Llorens (2003). The focus of this research is the representation of explicit relationships between objects. However, in many cases relationships are not explicitly defined but instead specified by rules. The support modelling languages have to express this kind of rules is weak. UML for instance has the concept of derived association for this purpose. The isDerived attribute in Association is defined as a boolean that "specifies whether the association is derived from other model elements such as other associations or constraints" (UML, 2004). However, this definition is very vague and hardly suitable to automate the implementation of derived associations.

In this paper we investigate several commonly encountered implementation patterns for rules defining relationships between objects. We then point out the weaknesses of this approaches and discuss several alternative strategies. Finally, we present a simple scripting language for rules and a rule compiler that addresses the problems discussed earlier.

IMPLEMENTATION PATTERNS

In object-oriented design, classes are used to represent vocabularies. More recently the term ontology has become fashionable to refer to these vocabularies, emphasizing the computation-independent aspect of the concepts described. Many software engineers prefer the term domain model. Business rules are then used to define relationships between instances of these classes. These rules are recorded during requirement analysis, then become part of the design model and are finally