Reverse Engineering Domain Models from Source Code

Daniel Ratiu
Technische Universität München
ratiu@in.tum.de

Abstract

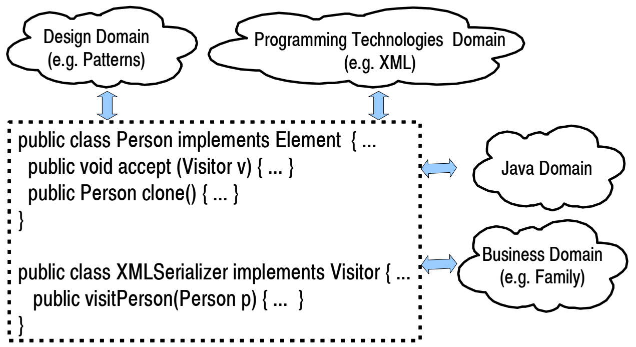
Programs model the real world: they act and respond to domain experts inputs as such they would know about a certain situation from the business domain they implement. Thereby, besides the technical information about how the computation is realized, programs contain a lot of domain knowledge. Therefore, programs and especially domain specific APIs can be seen as rich knowledge bases that accurately represent aspects of the domain. In this paper we advocate the need and feasibility for reverse engineering models that capture the business domain knowledge implemented in programs. We discuss the challenges involved in the reverse engineering of several kinds of domain models that are at different levels of complexity such as simple terminologies, taxonomies, light-weighted domain ontologies, domain specific invariants, domain specific laws, and behavioral models. We present our experience with manually reverse knowledge engineering the domain models and with automatic extraction of domain ontologies fragments by analyzing the commonalities of different domain specific APIs.

1 Introduction

Programs are used by domain experts as tools for resolving domain specific problems. They respond to the actions of their users as such they would know about a certain situation of the real world. Empirical studies on program comprehension show that rather than being an amorphous mass of computation out of which humans oriented behavior emerges, there is a strong correspondence between parts of programs and the domain knowledge that they implement (this stays at the basis of concepts assignment [2] and concepts location [4]). We conjecture that the source code of programs written so far (the entire source code written in any language) is one of the largest source of domain information in a machine processible form. Programs contain (many times only implicitly) knowledge about concepts from the business domain, about their relations, about the laws and constraints governing these relations, or about the behavior of domain entities.

We propose a mental experiment about an activity that is the reverse of program understanding (let's call it 'domain understanding by reading the code'): a person that has high experience with programming and has access to the source code of programs about a business domain X, wants to understand the domain X by carefully reading the code. Besides the low effectiveness of such an endeavor in the large due to the difference in the abstraction level at which humans think and at which the programs code is written, we can imagine that somebody can learn new things about the business domain of an application just by reading its code.

Besides the knowledge about the business domain, programs code contain a significant amount of knowledge of technical nature such as about programming technologies, algorithms, or design. In practice units of knowledge are scattered between many program modules and a single program module contains more knowledge units (phenomena known as delocalization [3] and interleaving [8]) – e. g. Figure 1 illustrates how are multiple kinds of knowledge used in different program parts: different classes reference knowledge about the business domain (e. g. family), the programming technologies (e. g. XML), architecture (e. g. Visitor pattern), and Java core library. Thereby the structure of the program is only weakly related to the partition of



```
public class Person implements Element { ...
  public void accept (Visitor v) { ... }
  public Person clone() { ... }
}

public class XMLSerializer implements Visitor { ...
  public visitPerson(Person p) { ... }
}
```

Figure 1. Weaving of different kinds (dimensions) of domain knowledge in programs

the domain knowledge that it implements. Due to delocalization and interleaving, in order to extract models about the business domain, we need on the one hand to distinguish between program fragments that implement the business domain and those that reference programming knowledge, and on the other hand to re-assembly pieces of models that refer to the business domain and that are implemented scattered in the code.

A special category of program fragments that reflect domain knowledge in a more explicit form are the domain specific APIs. We regard APIs in a wider sense, namely as the published interfaces of a sub-system and do not restrict ourselves to standard APIs that come with programming languages. In Figure 2 we illustrate the reflexion of knowledge in APIs and how it is subsequently used by programmers to implement other programs. Highly qualitative APIs reflect the domain knowledge in a faithful manner in order to make the APIs more usable.

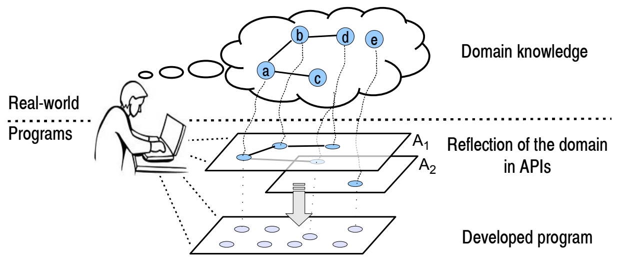


Figure 2. Knowledge reflexion in APIs

Domain models in MDE. When we refer in the following of this paper to domain models, we understand the models of the business domain. In the MDA terminology [1], our understanding of domain models is closest to computational independent models (CIM). These models represent the highest level of abstraction at which a software system can be seen since they abstract from the computational and platform specific issues. CIM capture only the essential complexity of the business domain without any implementation complexity. CIM are adequate means for reuse the domain knowledge in order for example to enable the migration to domain specific languages. Furthermore, they can be understood and validated by domain experts that have no knowledge about programming and thereby are important communication means.

In this paper we advocate the need and investigate the challenges (Section 2) and feasibility (Section 3) to reverse engineer computational independent models (CIM) from the source code. In Section 4 we sketch our previous work in this direction.

2 Challenges Reverse Engineering Domain Models

A reverse engineered domain model is *complete* if it contains the whole business domain knowledge that a domain expert would agree to be implemented in the analyzed system. A reverse engineered model is *sound* if the knowledge that it contains is accurate for the domain (it faithfully reflects the domain). The purpose of reverse engineering CIM from code is to recover domain models that have a high completeness and soundness degrees. In order to address this goal we need to face several challenges:

Bridge the conceptual gap between the source code and domain knowledge. Programs are written in general purpose languages. The language constructs are highly general and they do not "know" anything about the business domain. Most of the times the domain information is contained only informally in the names of program entities (identifiers) – for example, by obfuscating these names we would obtain programs that could not be understood by anybody. Thereby, the first challenge is to make disciplined use of the informal information contained in identifiers.

Eliminate the algorithmical encoding. Due to the big abstraction gap between the domain knowledge and the programming languages used to implement it, by programming happens a steep encoding of domain concepts in the code. Furthermore, there is a wide variety of possibilities to implement (encode) the same domain phenomena. In order to recover the domain knowledge, different implementations need to be interpreted in an unified manner.

Eliminate the noise. Besides the information about the modeled domain, the code contains a large amount of noise in form of implementation details. In order to obtain accurate domain models, we need to filter out these details and keep only the information that is relevant for the business domain. Due to the high degree of interleaving between the implementation details and domain knowledge, eliminating the noise in an automatic manner is (almost) impossible.

3 Envisioned Milestones

To quest the above challenges we need to have an incremental approach. In the widest sense, the recovery of accurate business domain models that are both complete and sound would require human intelligence. However, we advocate that the process can be (partly) automated and done in an incremental manner. Below we present several milestones in recovering increasingly complete and complex domain models. In Figure 3 is a simple program fragment that will be used in the following as running example.

```
public class Person extends Root {
   String firstName, familyName;
   String postalAddress;
   int age;
   Person father;
}

public class Mother exends Person  { ...}
```

```
public class Family {
   List<String> members, adultMembers;
   String postalAddress, serializedAddress;
}
```

Figure 3. Examples of code fragments

Reverse engineering domain terminology. Program identifiers contain a big part of the terminology of the business domain. This terminology can be put in direct correspondence with the domain concepts. For example, the terminology that can be extracted from the code fragment from above is 'person', 'name', 'age', and 'mother'. Each of these words reference well defined concepts from the family domain.

However, besides the words that refer to domain concepts, programs contain a lot of noise (e. g. misspelled identifiers, prepositions) and implementation details (e. g. 'int', 'String'), or just general words(e. g. 'first'). Another dimension of difficulties is the fact that many times the names are compound words (e. g. 'familyName') or that the concepts that are not directly lexicalized as individual words occur in the code in form of identifiers that contain more words (e. g. 'adultMembers', 'postalAddress'). Many compound identifiers contain besides words denoting domain concepts also words that are related to the implementation (e. g. 'serializedAddress'). The noise requires that the automatically extracted terminology to be reviewed by a domain expert. Once the terminology is extracted, we can enrich it with relations between concepts.

Reverse engineering taxonomies. The most fundamental relation between concepts is the taxonomical (is-a) relation. In the case of object-oriented programs it can be easier to extract since this relation is implemented usually through the sub-type relation between a class and its subtypes or as the has-type relation between variables and their types.

For example, from the above code fragment we can extract the relation "Mother is-a Person", or "Father is-a Person". However, not all inheritance relations are relevant for the domain – e. g. "Person is-a Root", or "Age is-a int" do not make sense from the point of view of the knowledge about the family domain. Therefore due to this noise the extracted taxonomy needs to be manually validated.

Reverse engineering complex domain relations. Besides the 'is-a' relation between domain concepts, there is a wide variety of other relations. These relations are more difficult to extract since they are implemented indirectly (and more ambiguously) in programs. Even basic relations such as those between concepts and their properties ('has-property') or between concepts and their parts ('has-part') are hard to distinguish in automatic manner (both are usually implemented as attributes of classes – e. g. 'members' of a 'Family', or the 'age' of a 'person').

Other more complex relations from the business domain (e. g. cause-effect) are only deeply encoded in programs and are not visible in the program syntax.

Reverse engineering domain specific laws and invariants. Once the concepts and relations are gathered, we need to enrich them with semantic information such as constraints, invariants, or domain specific laws. By doing this we reach a full description of statical situations that can occur in the business domain.

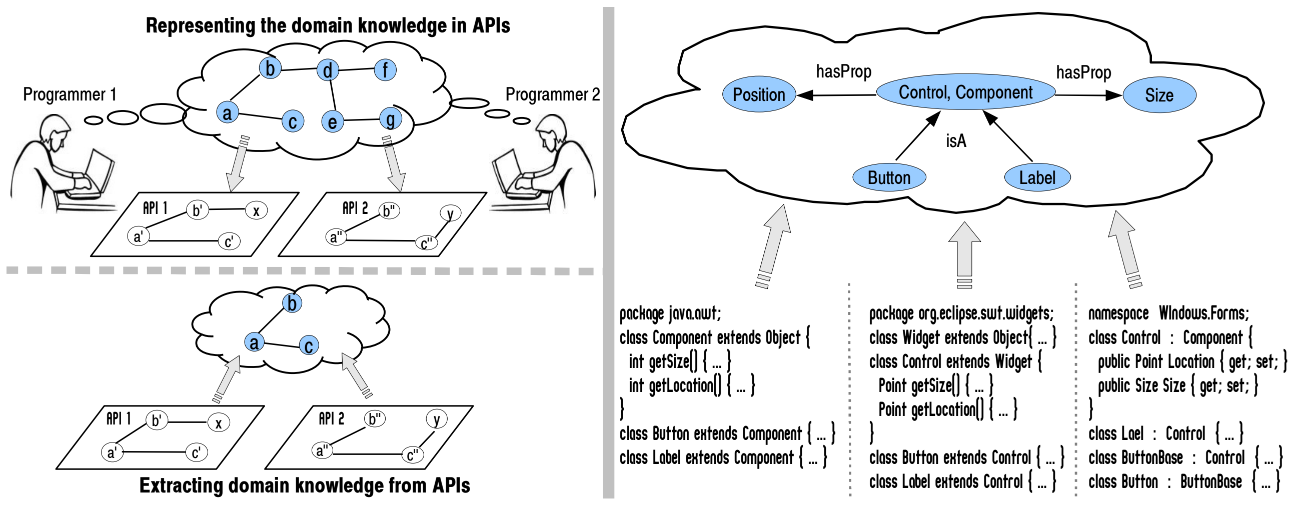
Example of constraints are the fact that the age of a person is always bigger than zero, or that each family has at least one member. Examples of invariants are the fact that adult members of a family are a subset of the set of members, or that the postal address of each member of a family is the same as the address of the family itself. Examples of laws in the family domain is that the current age of a person is given by the difference between the current calendar date and the birth date, or that persons become adults when they are eighteen years old.

Reverse engineering behavioral models. In order to describe the dynamics of the entities from the business domain we need to recover the behavioral models. The most simple behavioral models are simple business rules. More complex behavioral models are algorithms or transformation rules that change the state of the entities under consideration.

Examples of business rules at the level of the family domain is that when they marry the persons change their marital status, their identity cards, etc.

4 Our Experience with Recovering Light-Weighted Domain Ontologies

In the previous sections we presented the most important problems that make the recovery of business domain models from the code difficult (i. e. noise, conceptual gap, algorithmical encodings) and a set of increasingly complex models that can be seen as milestones in the extraction of complete domain models. In the following we sketch our experience with two approaches to extract domain models from the code. Our extracted models are light-weighted domain ontologies (i. e. they contain only domain concepts



```
package java.awt;
class Component extends Object {
  int getSize() { ... }
  int getLocation() { ... }
}
class Button extends Component { ... }
class Label extends Component { ... }
```

```
package org.eclipse.swt.widgets;
class Widget extends Object{ ... }
class Control extends Widget {
  Point getSize() { ... }
  Point getLocation() { ... }
}
class Button extends Control { ... }
class Label extends Control { ... }
```

```
namespace  WIndows.Forms;
class Control  :  Component {
  public Point Location { get; set; }
  public Size Size { get; set; }
}
class Lael  :  Control  { ... }
class ButtonBase  :  Control  { ... }
class Button  :  ButtonBase  { ... }
```

Figure 4. Extracting light-weighted ontology about GUIs from APIs

and relations and do not capture the constraints, invariants, laws or the behavior of the domain entities).

Analyze the similarities of programs that address the same domain Many times there are different programs (or parts thereof) that implement the same domain. In these cases, the commonalities that they share are related to the domain information (as exemplified in Figure 4). By analyzing these commonalities, we can eliminate a significant quantity of noise. In [7] we present an approach for (semi-)automatically extracting fragments of ontologies by analyzing the commonalities of the APIs that address the same domain. Comparing more APIs makes it feasible to extract domain ontologies that offer a good domain coverage.

One of the knowledge domains mostly covered by APIs is programming technologies domain (we consider that GUI is the business domain of the Java AWT library). We started to build a repository[1] that contains light-weighted ontologies about programming technologies[6]. Our ontology fragments share common sense, basic knowledge that is well known to any programmer and that is at a rather superficial level of formalization. For example, in the case of technologies related to graphical widgets (GUIs), such knowledge is that buttons are graphical components, have labels, layout information, can be displayed, can contain other graphical components, can update their views, or that there are other kinds of input-components such as check-boxes and radio buttons.

Manual reverse knowledge engineering By using adequate tools for extracting and prioritizing the vocabulary used in the source code, the manual reverse knowledge engineering can be feasible [5]. We advocate for a manual (but tool supported) and incremental approach that recovers domain models (or parts thereof) at the levels of details presented in the previous section.

[1] http://www4.in.tum.de/~ratiu/knowledge_repository.html

References

[1] Mda guide - version 1.0.1. Technical report, Object Management Group.

[2] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *ICSE '93*, pages 482–498. IEEE CS Press, 1993.

[3] S. Letovsky and E. Soloway. Delocalized plans and program comprehension. *IEEE Software*, 3(3):41–49, 1986.

[4] V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *IWPC'02*. IEEE CS Press, 2002.

[5] D. Ratiu and F. Deissenboeck. From reality to programs and (not quite) back again. In *Proceedings of the 15th International Conference on Program Comprehension (ICPC '07)*, 2007.

[6] D. Ratiu, M. Feilkas, F. Deissenboeck, J. Juerjens, and R. Marinescu. Towards a repository of common programming technologies knowledge. In *Proceedings of the International Workshop on Semantic Technologies in System Maintenance (STSM'08)*, 2008.

[7] D. Ratiu, J. Juerjens, and M. Feilkas. Extracting domain ontologies from domain specific APIs. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR'08)*. IEEE CS, 2008.

[8] S. Rugaber, K. Stirewalt, and L. M. Wills. The interleaving problem in program understanding. In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE'95)*, 1995.