Exploring the Limits of Domain Model Recovery

Paul Klint, Davy Landman, Jurgen Vinju
Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
{Paul.Klint, Davy.Landman, Jurgen.Vinju}@cwi.nl

***Abstract*—We are interested in re-engineering families of legacy applications towards using Domain-Specific Languages (DSLs). DSL design is hard because next to language engineering skills it requires a deep and complete understanding of the domain. Is it worth to invest in harvesting domain knowledge from the source code of legacy applications?**

Reverse engineering domain knowledge from source code is sometimes considered very hard or even impossible. Is it also difficult for "modern legacy systems"? More specifically, we would like to know if there are opportunities to harvest domain knowledge from high-level object-oriented code.

To explore this question, we compare manually recovered domain models of two open-source applications to a reference model extracted from domain literature.

The recovered models are accurate: they cover a significant part of the reference model and they do not contain much junk. We conclude that manually recovering domain knowledge from "modern legacy" code is viable and therefore is a valuable component of a domain re-engineering process.

I. Introduction[1]

There is ample anecdotal evidence [1] that the use of Domain-Specific Languages (DSLs) can significantly increase the productivity of software development, especially the maintenance part. DSLs model expected variations in both time (versions) and space (product families) such that some types of maintenance can be done on a higher level of abstraction and with higher levels of reuse. However, the initial investment in designing a DSL can be prohibitively high because a complete understanding of a domain is required. Moreover, when unexpected changes need to be made that were not catered for in the design of the DSL the maintenance costs can be relatively high. Both issues indicate how both the quality of domain knowledge and the efficiency of acquiring it are pivotal for the success of a DSL based software maintenance strategy.

In this paper we investigate the source code of existing applications as valuable sources of domain knowledge. DSLs are practically never developed in green field situations. We know from experience that rather the opposite is the case: several comparable applications by the same or different authors are often developed before we start considering a DSL. So, when re-engineering a family of systems towards a DSL, there is opportunity to reuse knowledge directly from people, from the documentation, from the user interface (UI) and from the source code. For the current paper we assume the people are no longer available, the documentation is possibly wrong or incomplete and the UI may hide important aspects, so we scope the question to recovering domain knowledge from source code. Is valuable domain knowledge present that can be included in the domain engineering process?

From the field of reverse engineering we know that recovering this kind of design information can be hard [2]. Especially for legacy applications written in low level languages, where code is not self-documenting, it may be easier to recover the information by other means. On the other hand, if a legacy application was written in a younger object-oriented language, should we not expect to be able to retrieve valuable information about a domain? This sounds good, but we would like to observe precisely how well domain model recovery from source code could work in reality. Note that both the quality of the recovered information and the position of the observed applications in the domain are important factors.

A. Positioning domain model recovery

One of the main goals of reverse engineering is *design recovery* [2] which aims to recover design abstractions from any available information source. A part of the recovered design is the domain model.

Design recovery is a very broad area, therefore, most research has focused on sub-areas. The *concept assignment problem* [3] tries to both discover human-oriented concepts and connect them to the location in the source code. Often this is further split into *concept recovery* [4]–[6][2], and *concept location* [7]. Concept location, and to a lesser extent concept recovery, has been a very active field of research in the reverse engineering community.

However, the notion of a concept is still very broad and *features* are an example of narrowed-down concepts and one can identify the sub-areas of *feature location* [8] and *feature recovery*. *Domain model recovery* as we will use in this paper is a closely related sub-area. We are interested in a pure domain model, without the additional artifacts introduced by software design and implementation. The location of these artifacts is not interesting either. For the purpose of this paper, a domain model (or model for short) consists of entities and relations between these entities.

Abebe et al.'s [9], [10] *domain concept extraction* is similar to our sub-area. As is Ratiu et al.'s [11] *domain ontology recovery*. In Section IX we will further discuss these relations.

B. Research questions

To learn about the possibilities of domain model recovery we pose this question: how much of a domain model can be

[1]The introduction and the conclusions of this paper were revised after considering the helpful feedback of the program committee of ICPC 2013.

[2]Also known as *concept mining*, *topic identification*, or *concept discovery*.

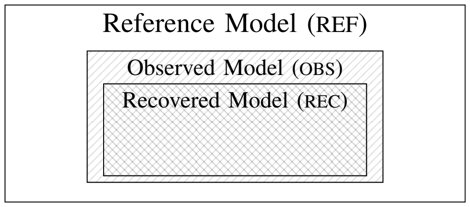

Fig. 1. Idealized picture of domain model recovery.

recovered under *ideal* circumstances? By ideal we mean that the applications under investigation should have well-structured and self-documenting object-oriented source code.

This leads to the following research questions:

Q1. Which parts of the domain are implemented by the application?
Q2. Can we manually recover those implemented parts from the object-oriented source code of an application?

Note that we exclude automated recovery here because any inaccuracies introduced by tool support could be misleading.

Figure 1 illustrates the various domains that are involved: The *Reference Model* (REF) represents all the knowledge about a specific domain and acts as oracle and upper limit for the domain knowledge that can be recovered from any application in that domain. The *Recovered Model* (REC) is the domain knowledge obtained by inspecting the source code of the application. The *Observed Model* (OBS) represents the part of the reference domain that an application covers, i.e. all the knowledge about a specific application in the domain that a user may obtain by observing its external behavior and its documentation but not its internal structure.

In Section II we describe our research method, explaining how we will analyze, amongst others, the relations between OBS/REF, REC/REF and OBS/REC in order to answer Q1 and Q2. The results of each step are described in detail in Sections III to VIII. Related work is discussed in Section IX and Section X (Conclusions) completes the paper.

II. Research method

In order to investigate the limits of domain model recovery we study *manually* extracted domain models. The following questions guide this investigation:

A) Which domain is suitable for this study?
B) What is the upper limit of domain knowledge, or what is our reference model (REF)
C) How to select two representative applications?
D) How do we recover domain knowledge that can be observed by the user of the application (Q1 & OBS)?
E) How do we recover domain knowledge from the source code (Q2 & REC)?
F) How do we compare models that use different vocabularies (terms) for the same concepts? (Q1, Q2)?
G) How do we compare the various domain models to measure the success of domain model recovery? (Q1,Q2)?

In Figure 1 we already illustrated the relations between the Observed (OBS) and Recovered Model (REC). Figure 2 describes

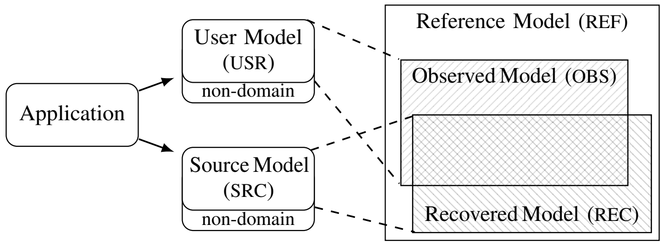

Fig. 2. Actual picture of domain model recovery for one application.

how OBS and REC relate to an application during actual domain model recovery. Of particular interest are the size and relative positions of the Observed Model and the Recovered Model. Ideally, both domain models should completely overlap, but there could be entities in OBS not present in REC. The largest difference between Figures 1 and 2 are the entities in REC that are not in OBS, these entities could be a failure in our construction of the Observed Model (OBS) or they could be concepts not exposed in any interface.

We will now answer the above questions in turn. Although we are exploring manual domain model recovery, we want to make this manual process as traceable as possible since this enables independent review of our results. Where possible we automate the analysis (calculation of metrics, precision and recall), and further processing (visualization, table generation) of manually extracted information. Both data and automation scripts are available online.[3]

A. Selecting a target domain

We have selected the domain of project planning for this study since it is a well-known, well-described, domain of manageable size for which many open source software applications exist. We use the Project Management Body of Knowledge (PMBOK) [12] published by Project Management Institute (PMI) for standard terminology in the project management domain. Note that as such the PMBOK covers a lot more than just project planning.

B. Obtaining the Reference Model (REF)

Validating the results of a reverse engineering process is difficult and requires an oracle, i.e., an *actionable* domain model suitable for comparison and measurement. We have transformed the descriptive knowledge in PMBOK into such a reference model using the following, traceable, process:

1) Read the PMBOK book.
2) Extract project planning facts.
3) Assign a number to each fact and store its source page.
4) Construct a domain model, where each entity, attribute, and relation are linked to one or more of the facts.
5) Assess the resulting model and repeat the previous steps when necessary.

The resulting domain model will act as our Reference Model. and Section III gives the details.

[3]See http://www.cwi.nl/~landman/icsm2013/.

C. Application selection

In order to avoid bias towards a single application, we need at least two project planning applications to extract domain models from. Section IV describes the selection criteria and the selected applications.

D. Observing the application

A user can observe an application in several ways, ranging from its UI, command-line interface, configuration files, documentation, scripting facilities and other functionality or information exposed to the user of the application. In this study we use the UI and documentation as proxies for what the user can observe. We have followed these steps to obtain the User Model (USR) of the application:

1) Read the documentation.
2) Determine use cases.
3) Run the application.
4) Traverse the UI depth-first for all the use cases.
5) Collect information about the model exposed in the UI.
6) Construct a domain model, where each entity and relation are linked to a UI element of the application.
7) Assess the resulting model and repeat the previous steps when necessary.

We report about the outcome in Section V.

E. Inspecting the source code

We have designed the following traceable process to extract a domain model from each application's source code, the Source Model (SRC):

1) Read the source code.
2) Collect source locations (file name and line number) related to the application's model.
3) Construct a model, where each entity, attribute, and relation is linked to a source location in the application's source code.
4) Assess the model and repeat the previous steps when necessary.

The results appear in Section VI.

F. Mapping models

After performing the above steps we have obtained five domain models for the same domain, derived from different sources:

- The Reference Model (REF) derived from PMBOK.
- For each of the two applications:
  - User Model (USR).
  - Source Model (SRC).

While all these model are in the project planning domain, they all use different vocabularies. Therefore, we have to manually map the models to the same vocabulary. Mapping the USR and SRC models onto the REF model, gives the Observed (OBS) and Recovered Model (REC).

The final mapping we have to make, is between the SRC and USR models. We want to understand how much of the User Model (USR) is present in the Source Model (SRC). Therefore, we also map the SRC onto the USR model, giving the Intra-Application Model (INT). The results of all these mappings are given in Section VII.

G. Comparing models

To be able to answer Q1 and Q2, we will compare selected pairs of the 11 produced models. Following other research in the field of concept assignment, we use the most common information retrieval (IR) approach, *recall* and *precision*, for measuring quality of the recovered data. Recall measures how much of the expected model is present in the found model, and precision measures how much of the found model is part of the expected model.

To answer Q1, the recall between REF and USR explains how much of the domain is covered by the application. Note that the result is subjective with respect to the size of REF: a bigger domain may require looking at more different applications that play a role in it. By answering Q2 first, analyzing the recall between USR and SRC, we will find out whether source code could provide the same recall as REF and USR. The relation between REF and SRC will confirm this conclusion. Our hypothesis is that since the selected applications are small, we can only recover a small part of the domain knowledge, i.e. a low recall.

The precision of the above mappings is an indication of the quality of the result in terms of how much extra (unnecessary) details we accidentally would recover. This is important for answering Q2. If the recovered information would be overshadowed by junk information, the recovery would have failed to produce the domain knowledge as well. We hypothesize that due to the high-level object-oriented designs of the applications we will get a high precision.

Some more validating comparisons, their detailed motivation and the results of all model comparisons are described in Section VIII.

III. Project Planning Reference Model

Since there is no known domain model or ontology for project planning that we are aware of, we need to construct one ourselves. The aforementioned PMBOK [12] is our point of departure. PMBOK avoids project management style specific terminology, making it well-suited for our information needs.

A. Gathering facts

We have analyzed the whole PMBOK book. This analysis has been focused on the concept of a *project* and everything related to *project planning* therefore we exclude other concepts and processes in the project management domain.

After analyzing 467 pages we have extracted 151 distinct facts related to project planning. A *fact* is either an explicitly defined concept, an implicitly defined concept based on a summarized paragraph, or a relations between concepts. These facts were located on 67 different pages. This illustrates that project planning is a subdomain and that project management as a whole covers many topics that fall outside the scope of the current paper. Each fact was assigned a unique number

TABLE I

NUMBER OF ENTITIES AND RELATIONS IN THE CREATED MODELS, AND THE AMOUNT OF LOCATIONS IN THE PMBOK BOOK, SOURCE CODE, OR UI SCREENS USED TO CONSTRUCT THE MODEL.

Source	Model	# entities	# relations			unique observations
			associations	specializations	total	
PMBOK	REF	74	75	32	107	83
Endeavour	USR	23	30	8	38	19
	SRC	26	51	8	59	80
OpenPM	USR	22	24	3	27	13
	SRC	28	44	6	50	68

and the source page number where it was found in PMBOK. Two example facts are: "A milestone is a significant point or event in the project."(id:108, page: 136) and "A milestone may be mandatory or optional." (id:109, page: 136).

B. Creating the Reference Model REF

In order to turn these extracted facts into a model for project planning, we have translated the facts to entities, attributes of entities, and relations between entities. The two example facts (108 and 109), are translated into a relation between the classes Project and Milestone, and the mandatory attribute for the Milestone class. The meta-model of our domain model is a class diagram. We use a textual representation in the meta-programming language Rascal [13] which is also used to perform calculations on these models (precision, recall).

Table I characterizes the size of the project planning reference domain model REF by number of entities, relations and attributes; it contains of 74 entities and 107 relations. There is also a set of 49 attributes, but this is incomplete is due to the lack of details in PMBOK. Therefore, we did not use the attributes of the reference model to calculate similarity.

The model is too large to include in this paper, however for demonstration purposes, a small subset of the model is shown in Figure 3.

Not all the facts extracted from PMBOK are used in the Reference Model. Some facts carry only explanations. For example "costs are the monetary resources needed to complete the project". Some facts explain dynamic relations that are not relevant for an entity/relationship model. These two categories explain 55 of the 68 unused facts. The remaining 13 facts were not clear enough to be used or categorized. In total 83 of the 151 observed facts are represented in the Reference Model.

C. Discussion

We have created a Reference Model that can be used as oracle for domain model recovery and other related reverse engineering tasks in the project planning domain. The model was created by hand by the second author, and care was taken to make the whole process traceable. We believe this model can be used for other purposes in this domain as well, such as application comparison and checking feature completeness.

Threats to validity: We use PMBOK as main source of information for project planning. There are different approaches to project planning and a potential threat is that some are not

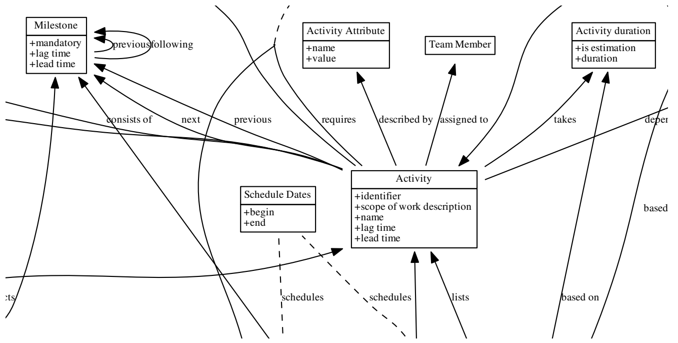

Fig. 3. Fragment of reference model REF visualized as UML class diagram.

covered in this book. Since PMBOK is an industry standard (ANSI and IEEE), we consider this to be a low-risk threat and have not mitigated it.

Another threat is that model recovery by another person could lead to a different model. The traceable extraction of the reference model makes it possible to understand the decisions on which the differences are based. Due to the availability of our analysis scripts, the impact of differences can be easily computed.

IV. APPLICATION SELECTION

We are interested in finding "ideal" project planning systems to manually read and extract domain models from. The following requirements have guided our search:

- Source code is available: to enable analysis at all.
- No more than 30 KSLOC: to keep manual analysis feasible.
- Uses an explicit data model, for example Model View Controller (MVC), or an Object-relational mapping (ORM): to ensure that domain elements can be identified in the source code.

We have made a shortlist of 10 open source project planning systems[4]. The list contains applications implemented in different languages (Java, Ruby, and C++) and sizes ranging from 18 KSLOC to 473 KSLOC.

From this Endeavour and OpenPM satisfy the aforementioned requirements. Endeavour is a Java application that uses a custom MVC design with ThinWire as front-end framework, and Hibernate as ORM. OpenPM uses Java servlets in combination with custom JavaScript. It also uses Hibernate as ORM. Table II and III describe the structure and size of the two applications[5]. Note that OpenPM's `view` package contained MVC controller logic, and the `servlets` the MVC views.

Both systems aim at supporting the process of planning by storing the process state but they hardly support process enforcement, except recording dependence between activities.

V. OBTAINING THE USER MODEL

We have used the UI and documentation of the applications to construct the User Model (USR). Use cases were extracted

[4] ChilliProject, Endeavour, GanttProject, LibrePlan, OpenPM, OpenProj, PLANdora, project.net, taskjuggler, Xplanner+.

[5]Number of files and SLOC are calculated using the cloc tool [14].

TABLE II
ENDEAVOUR: STRUCTURE AND SIZE.

Package	# files	SLOC	description
model	29	4474	MVC model.
view	108	10480	MVC view (UI).
controller	49	3404	MVC controller.
Total	186	18358	

TABLE III
OPENPM: STRUCTURE AND SIZE.

Package	# files	SLOC	description
model	29	5591	MVC model.
view	21	1546	MVC controller.
servlets	33	3482	MVC view (UI).
test	75	7137	UI & integration tests.
Total	158	17756	

from the documentation when possible.[6] Following these use cases, a depth-first exploration of the UI is performed. For every entity and relation we have recorded in which UI screen we first observed it. Table I describes the User Models for both Endeavour and OpenPM.

For example the Task entity in Endeavour's USR Model was based on the sub-window "Task Details" of the "Home" window.

A. Discussion

We have tried to understand the domain knowledge represented by the applications by manually inspecting it from the user's perspective. Both applications used Ajax to provide an interactive experience.

Endeavour uses the Single Page Application style, with a windowing system similar to MS Windows®. The UI is easy to understand, and different concepts are consistently linked across the application. OpenPM uses a more modern interface. However, we experienced more confusion on how to use it. It assumes a specific project management style (SCRUM), and requires more manual work by the user.

We have observed that creating a User Model is simple. For systems of our size, a single person can construct a User Model in one day. This is considerably less than creating a Source Model and suggests that the UI is an effective source for recovering domain models.

Threats to validity: We use the User Model as a proxy for the real domain knowledge exposed by the application. The limit of this knowledge is hard to define, but we believe our approach is an accurate approximation.

We can not be sure about our coverage of the User Model. It could be possible there are other interfaces to the application we are unaware of. Moreover, there could be conditions, triggers, or business rules only observable in very specific scenarios. Some of these issues will be observed in the various model comparisons. We are not aware of other approaches to further increase confidence in our coverage.

[6]Unfortunately, OpenPM does not provide documentation.

VI. OBTAINING MODELS FROM SOURCE CODE

A. Domain model recovery

We have chosen the Eclipse Integrated Development Environment (IDE) to read the source code of the selected applications. Our goal was to maximize the amount of information we could recover. Therefore, we have first read the source code and then used Rascal to analyze relations in the source code. Rascal uses Eclipse's JDT to analyze Java code, and provides a visualization library that can be used to quickly verify hypothesis formed during the first read-through.

For the actual creation of the model, we have designed and followed these rules:

- Read only the source code, not the database scheme/data.
- Do not run the application.
- Use the terms of the applications, do not translate them to terms used in the Reference Model.
- Include the whole model as seen by the application, do not filter out obvious implementation entities.
- Do read comments and string literals.

We have used the same meta-model as used for describing the Reference Model. We replaced the fact's identifiers with source locations (filename and character range), which are a native construct in Rascal. To support the process of collecting facts from the source code we added a menu-item to the context-menu of the Java editor to write the cursor's source location to the clipboard.

The domain model for each application was created in a similar fashion as we did when creating the reference model. All the elements in the domain model are based on one or more specific observations in the source code.

For example the relation between Task and Dependency in Endeavour's SRC model is based on the `List<Dependency> dependencies` field found on line 35.

B. Results

Table I shows the sizes of the extracted models for both applications expressed in number of entities, relations and attributes and the number of unique source code locations where they were found.

1) Endeavour: In Endeavour 26 files contributed to the domain model. 22 of those files were in the model package, the other 4 were from the controller package. The controller classes were single occurrences, 155 of the source locations were from the model package.

2) OpenPM: In OpenPM 22 files contributed to the domain model. These files were all located in the model package.

C. Discussion

We have performed domain model recovery on two open source software applications for project planning.

Both applications use the same ORM system, but a different version of the API. Endeavour also contains a separate view model, which is used in the MVC user interface. However, it has been implemented as a pass-through layer for the real model.

TABLE IV
CATEGORIES FOR SUCCESSFULLY MAPPED ENTITIES

Mapping name	Description
Equal Name	Entity has the same name as an entity in the other model. Note that this is the only category which can also be a failure when the same name is used for semantically different entities
Synonym	Entity is a direct synonym for an entity in the other model, and is it not a homonym.
Extension	Entity captures a wider concept than the same entity in the other model.
Specialization	Entity is a specific or concrete instance of the same entity in the other model.
Implementation specialization	Comparable to specialization but the specialization is related to an implementation choice.

TABLE V
CATEGORIES FOR UNSUCCESSFULLY MAPPED ENTITIES

Mapping name	Description
Missing	The domain entity is missing in the other model, i.e. a false positive. This is the default mapping failure when an entity cannot be mapped via any of the other categories.
Implementation	The entity is an implementation detail and is not a real domain model entity.
Domain detail	The entity is a detail of the sub domain.
Too detailed	An entity is a domain entity but is too detailed in comparison with the other model.

Threats to validity: A first threat (to internal validity) is that manual analysis is always subject to bias from the performer. We have mitigated this by maximizing the traceability of our analysis: we have followed a fixed analysis process and have performed multiple analysis passes over the source code.

A second threat (to external validity) is the limited size of the analyzed applications, both contain less than 20 KSLOC Java. Larger applications would make our conclusions more interesting and general, but they would also make the manual analysis less feasible.

VII. MAPPING MODELS

We now have five domain models of project planning: one reference model (REF) to be used as oracle, and four domain models (SRC, USR) obtained from the two selected project planning applications. These models use different vocabulary, we have to map them onto the same vocabulary to be able to compare them.

A. Lightweight domain model mapping

We manually map the entities between different comparable models. The question is how to decide whether to entities are the same. Strict string equality is too limited and should be relaxed to some extent.

Table IV and V show the mapping categories we have identified for the (un)successful mapping of model entities.

TABLE VI
ENDEAVOUR: ENTITIES IN THE MAPPED MODELS, PER MAPPING CATEGORY

Category	USR	REF	SRC	REF	SRC	USR
Equal Name	7	7	7	7	21	21
Synonym	2	3	2	3	3	2
Extension	0	0	0	0	0	0
Specialization	5	3	5	3	0	0
Implementation specialization	1	1	1	1	0	0
Total	15	14	15	14	24	23
Equal Name†	1	-	1	-	0	-
Missing	1	-	2	-	0	-
Implementation	1	-	2	-	2	-
Domain Detail	5	-	6	-	0	-
Too Detailed	0	-	0	-	0	-
Total	8	-	11	-	2	-

† A false positive, in Endeavour the term Document means something different then the term Documentation in the Reference Model.

TABLE VII
OPENPM: ENTITIES IN MAPPED MODELS, PER MAPPING CATEGORY

Category	USR	REF	SRC	REF	SRC	USR
Equal Name	1	1	1	1	18	18
Synonym	3	3	4	4	4	4
Extension	1	1	1	1	0	0
Specialization	0	0	0	0	0	0
Implementation specialization	1	1	1	1	0	0
Total	6	6	7	7	22	22
Missing	2	-	2	-	1	-
Implementation	12	-	17	-	5	-
Domain Detail	0	-	0	-	0	-
Too Detailed	2	-	2	-	0	-
Total	16	-	21	-	6	-

B. Mapping results

We have manually mapped all the entities in the User Model (USR) and the Source Model (SRC) to the Reference Model (REF), and SRC to USR. For each mapping we have explicitly documented the reason for choosing this mapping. For example, in Endeavour's SRC model the entity *Iteration* is mapped to *Milestone* in the Reference Model using specialization, with documented reason: "*Iterations split the project into chunks of work, Milestones do the same but are not necessarily iterative.*"

Table VI and VII contain the number of mapping categories used for both applications, per mapping. For some mapping categories, it is possible for one entity to map to multiple, or multiple entities to one. For example the *Task* and *WorkProduct* entities in Endeavour's SRC model are mapped on the *Activity* entity in the Reference Model. Therefore, we report the numbers of the entities in both the models, the source and the target.

The large number of identically named entities between Endeavour and the reference model is due to the presence of a similar structure of five entities, describing all the possible activity dependencies.

An example of a failed mapping is the *ObjectVersion* entity in

TABLE VIII
ENTITIES FOUND IN THE VARIOUS DOMAIN MODELS.

Source	Model	Entities[†]
PMBOK	REF	Action, **Activity**, **Activity Attribute**, **Activity Dependency**, **Activity duration**, Activity list, Activity resource, Activity sequence, Activity template, Approver, Budget, Change Control Board, **Change request**, Closing, Communications plan, Composite resource calendar, Composite resource calendar availability, Constrain, Corrective action, **Defect**, Defect repair, **Deliverable**, **Documentation**, Environment, Equipment, External, **FinishFinish**, **FinishStart**, Human Resource Plan, Information, Internal, Life cycle, Main, Material, **Milestone**, Objective, Organisation, Organizing, People, Person, Phase, Planned work, Portfolio, Preparing, Preventive action, Process, Product, **Project**, Project management, Project plan, **Project schedule**, Project schedule network diagram, Quality, **Requirement**, Resource, Resource calendar, Resource calendar availability, Result, Risk, Risk management plan, Schedule, Schedule Dates, Schedule baseline, Schedule data, Scope, Service, Stakeholder, **StartFinish**, **StartStart**, Supplies, **Team Member**, Work Breakdown Structure, Work Breakdown Structure Component, Work Package
Endeavour	USR	**Actor**, **Attachment**, **Change Request**, Comment, **Defect**, **Document**, Event, **FinishFinish**, **FinishStart**, Glossary, **Iteration**, **Project**, **ProjectMember/Stakeholder**, Security Group, **StartFinish**, **StartStart**, **Task**, **Task Dependency**, Test Case, Test Folder, Test Plan, **Use Case**, **X**
Endeavour	SRC	**Actor**, **Attachment**, **ChangeRequest**, Comment, **Defect**, **Dependency**, **Document**, Event, **FinishFinish**, **FinishStart**, GlossaryTerm, **Iteration**, Privilege, **Project**, **ProjectMember**, SecurityGroup, **StartFinish**, **StartStart**, **Task**, TestCase, TestFolder, TestPlan, TestRun, **UseCase**, Version, **WorkProduct**
OpenPM	USR	Access Right, **Attachment**, Button, Comment, Create, Delete, **Effort**, Email Notification, FieldHistory, HistoryEvent, **Iteration**, Label, Link, ObjectHistory, **Product**, Splitter, State, Tab, **Task**, Type, Update, **User**
OpenPM	SRC	Access, Add, **Attachment**, Comment, Create, Delete, **Effort**, EmailSubscription, EmailSubscriptionType, Event, FieldType, FieldVersion, Label, Link, **Milestone**, ObjectType, ObjectVersion, **Product**, Remove, Splitter, **Sprint**, Tab, **Task**, TaskButton, TaskState, TaskType, Update, **User**

† Bold entity in Reference Model is used in application models. Bold entity in application model could be mapped to entity in Reference Model.

the Source Model of OpenPM. This entity is an implementation detail. It is a variant of the Temporal Object pattern[7] where every change of an entity is stored to explicitly model the history of all the objects in the application.

Table VIII contains all the entities per domain model, and highlights the mapped entities.

C. Discussion

We have used a lightweight approach for mapping domain models. Our mapping categories may be relevant for other projects and can be further extended and evaluated.

At most half of the domain models recovered from the applications could be mapped to the reference model. The other half of the extracted models regarded details of the domain or the implementation.

Threats to validity: A threat to external validity is that we have used an informal approach to map the domain models of the two applications to the reference model. The mapping categories presented above, turned out to be sufficient for these two applications, however we have no guarantees for other application of these categories. The categories have evolved during the process and each time a category was added or modified all previous classifications have been reconsidered.

[7]See http://martinfowler.com/eaaDev/TemporalObject.html.

VIII. Comparing the models

We now have five manually constructed and six derived domain models for project planning:

- One reference model (REF) to be used as oracle.
- Four domain models (SRC, USR) obtained from each of the two selected project planning applications.
- Six derived domain models (OBS, REC, INT) resulting from the mapping of the previous four (SRC, USR).

How can we compare these models in a meaningful way?

A. Recall and Precision

The most common measures to compare the results of an IR technique are *recall* and *precision*. Recall and precision are calculated between two datasets: the expected dataset and the retrieved one. Recall is a measure for how much of the expected dataset is retrieved. Precision is a measure for how much of the retrieved dataset was actually expected. Often it is not possible to get the 100% in both, and we have to discuss which measure is more important in the case of our model comparisons.

We have more than two datasets, and depending on the combination of datasets, recall or precision is more important. Table IX explains in detail how recall and precision will be used and explains for the relevant model combinations which measure is useful and what will be measured.

Given two models M_1 and M_2, we use the following notation. The comparison of two models is denoted by $M_1 \diamond M_2$ and results in recall and precision for the two models. If needed, M_1 is first mapped to M_2 as described in Tables VI and VII.

B. Results

Tables X and XI shows the results for, respectively, Endeavour and OpenPM. Which measures are calculated is based on the analysis in Table IX.

C. Relation Similarity

Since recall and precision for sets of entities provides no insight into similarity of the relations between entities, we need an additional measure. Our domain models contain entities and their relations. Entities represent the concepts of the domain, and relations their structure. If we consider the relations as a set of edges, we can directly calculate recall and precision in a similar fashion as described above.

We also considered some more fine grained metrics for structural similarity. Our domain model is equivalent to a subset of UML class diagrams and several approaches exist for calculating the minimal difference between such diagrams [15], [16]. Such "edit distance" methods give precise indications of how big the difference is. Similarly we might use general graph distance metrics [17]. We tried this latter method and found that the results, however more sophisticated, were harder to interpret. For example, USR and REF were 11% similar for Endeavor. This seems to be in line with the recall numbers, 6% for relations and 19% for entities, but the interesting precision results (64% and 15%) are lost in this metric. So we decided not to report these results and stay with the standard accuracy analysis.

TABLE IX
RECALL AND PRECISION EXPLAINED PER MODEL COMBINATION.

Retrieved	Expected	Recall	Precision
USR	REF	Which part of the domain is covered by an application. This is subjective to the size of REF.	How many of the concepts in USR are actually domain concepts, e.g., how much implementation details are in the *application*?
SRC	REF	How much of REF can be recovered from SRC. If high then this should confirm high recall for both USR ◇ REF and SRC ◇ USR.	How much of SRC are actually domain concepts, e.g., how much implementation junk is accidentally recovered from *source*?
SRC	USR	How much of USR can be recovered by analyzing the source code (SRC). This gives no measure of the amount of actual domain concepts found.	How many details are in SRC, but not in USR? If USR were a perfect representation of the application knowledge, this category would only contain dead-code and unexposed domain knowledge.

TABLE X
ENDEAVOUR: RECALL AND PRECISION.

Comparison	Recall		Precision	
	entities	relations	entities	relations
USR ◇ REF	19%	6%	64%	15%
SRC ◇ REF	19%	6%	56%	13%
SRC ◇ USR	100%	92%	92%	74%

TABLE XI
OPENPM: RECALL AND PRECISION.

Comparison	Recall		Precision	
	entities	relations	entities	relations
USR ◇ REF	7%	3%	23%	16%
SRC ◇ REF	9%	6%	25%	18%
SRC ◇ USR	100%	80%	79%	44%

D. Discussion

1) Low precision and recall for relations: On the whole the results for the precision and recall of the relation part of the models are lower than the quality of the entity mappings. We investigated this by taking a number of samples. The reason is that the Reference model is more detailed, introducing intermediate entities with associated relations. For every intermediate entity, two or more relations are introduced which can not be found in the recovered models.

These results indicate that the recall and precision metrics for sets of relations underestimate the structural similarity of the models.

2) Precision of OBS: USR ◇ REF: We found the precision of OBS to be 64% (Endeavour) and 23% (OpenPM), indicating that both applications contain a significant amount of entities that are unrelated to project planning as delimited by the Reference Model. For Endeavour, out of the 8 unmappable entities (see Table VI in section VII), only 2 were actual implementation details. The other 6 are sub-domain details not globally shared within the domain. If we recalculate to correct for this, Endeavour's Observed Model even has a precision of 91%. For OpenPM there are only 2 out of the 16 for which this correction can be applied, leaving the precision at 36%. For the best scenario, in this case represented by Endeavour, 90% of the User Model (USR) is part of the Reference Model (REF).

3) Recall of OBS: USR ◇ REF: The recall for the Observed Model (OBS) is for Endeavour 19% and for OpenPM 7%.

TABLE XII
COMBINED: RECALL AND PRECISION.

Comparison	Recall		Precision	
	entities	relations	entities	relations
USR ◇ REF	22%	7%	40%	14%
SRC ◇ REF	23%	9%	36%	13%

Which means both applications cover less then 20% of the project planning domain.

4) Precision of REC: SRC ◇ REF: The precision of the Recovered Model (REC) is for Endeavour 56% (corrected 88%), and for OpenPM 25% (corrected 39%). This shows that for the best scenario, represented again by Endeavour, the Source Model only contains 12% implementation details.

5) Recall of REC: SRC ◇ REF: The recall for the Observed Model (REC) is for Endeavour 19% and for OpenPM 9%. The higher recall for OpenPM, compared to OBS, for both entities and relations is an example where the Source Model contained more information then the User Model, which we will discuss in the next paragraph.

6) Precision and recall for INT: SRC ◇ USR : How much of the User Model can be recovered by analyzing only the Source Model? For both Endeavour and OpenPM, recall is 100%. This means that every entity in the USR model was found in the source code. Endeavour's precision was 92% and OpenPM's 79%. OpenPM contains an example where information in the Source Model is not observable in the User Model: comments in the source code explain the *Milestones* and their relation to *Iterations*.

The 100% recall and high precision mean that these applications were indeed amenable for reverse engineering (as we hypothesized when selecting these applications). We could extract most of the information from the source code.

For this comparison, even the relations score quite high. This indicates that User Model and Source Model are structurally similar. Manual inspection of the models confirms this.

7) Recall for Endeavour and OpenPM combined: Endeavour's and OpenPM's recall of USR ◇ REF and SRC ◇ REF measure the coverage of the domain a re-engineer can achieve. How much will the recall improve if we combine the recovered models of the two systems?

We only have two small systems, however, Table XII contains the recall and precision for Endeavour and OpenPM combined. A small increase in recall, from 19% to 23%, indicates that there is a possibility for increasing the recall by observing more systems. However, as expected, at the cost of precision.

8) Interpretation: Since our models are relatively small, our results cannot be statistically significant but are only indicative. Therefore we should not report exact percentages, but characterizing our recall and precision as *high* seems valid. Further research based on more applications is needed to confirm our results.

IX. Related work

There are many connections between ontologies and domain models. The model mappings that we need are more specific than the ones provided by general ontology mapping [18].

Abebe and Tonella [9] introduced a natural language parsing (NLP) method for extracting an ontology from source code. They came to the same conclusion as we do: this extracted ontology contains a lot of implementation details. Therefore, they introduced an IR filtering method [10] but it was not as effective as the authors expected. Manual filtering of the IR keyword database was shown to improve effectiveness. Their work is in the same line as ours, but we have a larger reference domain model, and we focus on finding the limits of domain model recovery, not on an automatic approach. It would be interesting to apply their IR filtering to our extracted models.

Ratiu et al. [11] proposed an approach for domain ontology extraction. Using a set of translation rules they extract domain knowledge from the API of a set of related software libraries. Again, our focus is on finding the limits of model recovery, not on automating the extraction.

Hsi et al. [19] introduced ontology excavation. Their methodology consists of a manual depth-first modeling of all UI interactions, and then manually creating an ontology, filtering out non-domain concepts. They use five graph metrics to identify interesting concepts and clusters in this domain ontology. We are interested in finding the domain model inside the user-interface model, Hsi et al. perform this filtering manually, and then look at the remaining model. Automatic feature extraction of user interfaces is described in [20].

Carey and Gannod [6] introduced a method for concept identification. Classes are considered the lowest level of information of an object-oriented system and Machine Learning is used in combination with a set of class metrics. This determines interesting classes, which should relate to domain concepts. Our work is similar, but we focus on *all* the information in the source code, and are interested in the maximum that can be recovered from the source. It could be interesting to use our reference model to measure how accurately their approach removes implementation concerns.

UML class diagram recovery [21], [22] is also related to our work but has a different focus. Research focuses on the precision of the recovered class diagrams, for example the difference between a composition and aggregation relation. We are interested in less precise UML class diagrams.

Work on recovering the concepts, or topics, of a software system [4], [5] has a similar goal as ours. IR techniques are used to analyze all the terms in the source code of a software system, and find relations or clusters. Kuhn et al. [5] report on the importance of identifier naming and the difficulty of evaluating their results. Our work focuses less on structure and grouping of concepts and we evaluate our results using a constructed reference model.

Reverse engineering the relation between concepts or features [8], [23], assumes that there is a set of known features or concepts and tries to recover the relations between them. These approaches are related to our work since the second half of our problem is similar: after we have recovered domain entities, we need to understand their relations.

DeBaud et al. [24] report on a domain model recovery case study on a COBOL program. By manual inspection of the source code, a developer reconstructed the data constructs of the program. They also report that implementation details make extraction difficult, and remark that systems often implement multiple domains, and that the implementation language plays an important role in the discovery of meaning in source code.

We do not further discuss other related work on knowledge recovery that aims at extracting facts about architecture or implementation. One general observation in all the cited work is that it is hard to separate domain knowledge from implementation knowledge.

X. Conclusions

We have explored the limits of domain model recovery via a case study in the project planning domain. Here are our results and conclusions.

A. Reference model

Starting with PMBOK as authoritative domain reference we have manually constructed an actionable domain model for project planning. This model is openly available and may be used for other reverse engineering research projects.

B. Lightweight model mapping

Before we can understand the differences between models, we have to make them comparable by mapping them to a common model. We have created a manual mapping method that determines for each entity if and how it maps onto the target model. The mapping categories evolved while creating the mappings. We have used this approach to describe six useful mappings, four to the Reference Model and two to the User Model.

C. What are the limits of domain model recovery?

We have formulated two research questions to get insight in the limits of domain model recovery. Here are the answers we have found (also see Table IX and remember our earlier comments on the interpretation of the percentages given below).

Q1: Which parts of the domain are implemented by the application? Using the user view (USR) as a representation of the part of the domain that is implemented by an application, we have created two domain models for each of the two selected applications. These domain models represent the domain as exposed by the application. Using our Reference Model (REF) we were able to determine which part of USR was related to project planning. For our two cases 80% and 90% of the User Model (USR) can be mapped to the Observed Model (OBS). This means only 20% and 10% of the UI is about topics not related to the domain. From the user perspective we could determine that the applications implement 19% and 7% of the domain.

The tight relation between the USR and the SRC model (100% recall) shows us that this information is indeed explicit and recoverable from the source code. Interestingly, some domain concepts were found in the source code that were hidden by the UI and the documentation, since for OpenPM the recall between USR and REF was 7% where it was 9% between SRC and REF.

So, the answer for Q1 is: the recovered models from source code are trustworthy, and only a small part of the domain is implemented by these tools (they implement only 10-20% of the domain).

Q2: Can we recover those implemented parts from the source of the application? Yes, see the answer to Q1. The high recall between USR and SRC shows that the source code of these two applications explicitly models parts of the domain. The high precisions (92% and 79%) also show that it was feasible to filter implementation junk manually from these applications from the domain model.

D. Perspective

For this research we manually recovered domain models from source code to understand how much valuable domain knowledge is present in source code. We have identified several follow-up questions:

- How does the quality of extracted models grow with the size and number of applications studied? (Table XII)
- How can differences and commonalities between applications in the same domain be mined to understand the domain better? Here we may explore the results of the variability engineering community.
- How does the quality of extracted models differ between different domains, different architecture/designs, different domain engineers?
- How can the extraction of a User Model help domain model recovery in general. Although we have not formally measured the effort for model extraction, we have noticed that extracting a User Model requires much less effort than extracting a Source Model.
- How do our manually extracted models compare with automatically inferred models?
- What tool support is possible for (semi-)automatic model extraction?
- How can domain models guide the design of a DSL?

Our results of manually extracting domain models are encouraging. They suggest that when re-engineering a family of object-oriented applications to a DSL their source code is a valuable and trustworthy source of domain knowledge, even if they only implement a small part of the domain.

References

[1] M. Mernik, J. Heering, and A. Sloane, “When and how to develop domain-specific languages.” *ACM Comput. Surv.*, no. 37, pp. 316–344, 2005.

[2] T. Biggerstaff, “Design recovery for maintenance and reuse,” *Computer*, vol. 22, no. 7, pp. 36–49, Jul. 1989.

[3] T. J. Biggerstaff, B. G. Mitbander, and D. Webster, “The concept assignment problem in program understanding,” in *Proc. 15th international conference on Software Engineering*, ser. ICSE ’93. IEEE Computer Society Press, May 1993, pp. 482–498.

[4] E. Linstead, P. Rigor, S. K. Bajracharya, C. V. Lopes, and P. Baldi, “Mining concepts from code with probabilistic topic models,” in *22nd IEEE/ACM International Conference on Automated Software Engineering*, 2007, pp. 461–464.

[5] A. Kuhn, S. Ducasse, and T. Gîrba, “Semantic clustering: Identifying topics in source code,” *Information & Software Technology*, vol. 49, no. 3, pp. 230–243, 2007.

[6] M. M. Carey and G. C. Gannod, “Recovering Concepts from Source Code with Automated Concept Identification,” in *15th International Conference on Program Comprehension*, 2007, pp. 27–36.

[7] V. Rajlich and N. Wilde, “The Role of Concepts in Program Comprehension,” in *10th International Workshop on Program Comprehension*, 2002, pp. 271–280.

[8] T. Eisenbarth, R. Koschke, and D. Simon, “Locating features in source code,” *IEEE Trans. Software Eng.*, vol. 29, no. 3, pp. 210–224, 2003.

[9] S. L. Abebe and P. Tonella, “Natural language parsing of program element names for concept extraction,” in *18th IEEE International Conference on Program Comprehension*, 2010, pp. 156–159.

[10] ——, “Towards the Extraction of Domain Concepts from the Identifiers,” in *18th Working Conference on Reverse Engineering*, 2011, pp. 77–86.

[11] D. Ratiu, M. Feilkas, and J. Jürgens, “Extracting domain ontologies from domain specific APIs,” in *12th European Conference on Software Maintenance and Reengineering*, vol. 1, 2008, pp. 203–212.

[12] P. M. Institute, Ed., *A Guide to the Project Management Body of Knowledge*, 4th ed. Project Management Institute, 2008.

[13] P. Klint, T. van der Storm, and J. J. Vinju, “RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation,” in *Proc. 9th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, September 2009, pp. 168–177.

[14] “Count Lines of Code Tool,” http://cloc.sourceforge.net.

[15] U. Kelter, J. Wehren, and J. Niere, “A Generic Difference Algorithm for UML Models,” in *Software Engineering 2005*, ser. LNI, P. Liggesmeyer, K. Pohl, and M. Goedicke, Eds., vol. 64, 2005, pp. 105–116.

[16] D. Ohst, M. Welle, and U. Kelter, “Differences between versions of UML diagrams,” *SIGSOFT Softw. Eng. Notes*, vol. 28, no. 5, pp. 227–236, Sep. 2003. [Online]. Available: http://doi.acm.org/10.1145/949952.940102

[17] H. Bunke and K. Shearer, “A graph distance metric based on the maximal common subgraph,” *Pattern Recognition Letters*, vol. 19, no. 3-4, pp. 255–259, 1998.

[18] N. Choi, I.-Y. Song, and H. Han, “A survey on ontology mapping,” *SIGMOD Rec.*, vol. 35, no. 3, pp. 34–41, Sep. 2006. [Online]. Available: http://doi.acm.org/10.1145/1168092.1168097

[19] I. Hsi, C. Potts, and M. M. Moore, “Ontological Excavation: Unearthing the core concepts of the application,” in *10th Working Conference on Reverse Engineering*, 2003, pp. 345–352.

[20] M. Bacíková and J. Porubän, “Analyzing stereotypes of creating graphical user interfaces,” *Central Europ. J. Computer Science*, vol. 2, no. 3, pp. 300–315, 2012.

[21] K. Wang and W. Shen, “Improving the Accuracy of UML Class Model Recovery,” in *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, 2007, pp. 387–390.

[22] A. Sutton and J. I. Maletic, “Mappings for Accurately Reverse Engineering UML Class Models from C++,” in *12th Working Conference on Reverse Engineering*, 2005, pp. 175–184.

[23] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, “Reverse engineering feature models,” in *Proc. 33rd International Conference on Software Engineering*, 2011, pp. 461–470.

[24] J.-M. DeBaud, B. Moopen, and S. Rugaber, “Domain Analysis and Reverse Engineering,” in *Proc. the International Conference on Software Maintenance*, 1994, pp. 326–335.