System Generation for Time and Activity Management Product Lines

Jenya Levin

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirements
For the Master's degree in Computer Science

Ottawa-Carleton Institute for Computer Science
University of Ottawa
Ottawa, Ontario, K1N 6N5
Canada

TABLE OF CONTENTS

LIST OF FIGURES AND TABLES

GLOSSARY

Base case. The smallest unit of functionality that allows the system to perform its primary function. In the case of time and activity management systems, the base case allows the person to enter a comment describing the activities they performed for a certain amount of time on a certain date.

Cluster. A group of classes that together provide particular functionality (implement a particular group of features) for the application.

Common base. A group of clusters that is common to most applications within a domain. In the case of time and activity management systems, the common base includes clusters related to time entry, user management, task management, company structure, and excludes payroll, accrual, invoice generation, etc.

CRUD. Create, Read, Update, Delete – basic functions in data storage and user interfaces.

ERD. Entity-relational diagram used to model relational data structures.

Feature. As used in this work, denotes a particular functional property of an application. Features are documented via use cases. Classes addressing closely-related features are grouped into functionality clusters. During the system generation stage, each invoke() statement in the VML invocation file corresponds to a feature.

Invocation file. File containing VML invoke() statements, each of which corresponds to a feature. By parsing the invocation file, the VML4Umple compiler generates code for each feature in the order listed.

Product line. Several products that have many common elements and a few elements that differ from one to others.

TAM systems. Time and Activity Management systems – systems that deal with time entry against activities, such as personal time management utilities and business employee time tracking.

UI. User Interface – means by which the users interact with a system. Used here to primarily denote graphical user interfaces.

UML. Unified Modeling Language, general-purpose modeling language accompanied by graphical notation.

UMLet. Tool geared towards graphical UML modeling.

Umple. Textual modeling language based on UML.

VML. Variability Modelling Language, language for modeling commonalities and variabilities in software product lines.

VML4Umple. Product line modeling language where VML is used for hierarchical model of features, whereas Umple is used to model the actual feature implementation.

ABSTRACT

This thesis investigates a product line derivation methodology to create a variability model of a whole domain. From this variability model, we can then use one-step code generation to create distinct products that meet differing sets of requirements.

We derive a time and activity management (TAM) product line from four existing systems using our methodology. We describe the product line using the VML4Umple language we adopt by combining the strengths of VML and Umple notations. From this we show how it is possible to generate any number of Umple models of TAM systems, each with a different combination of features. The results can be compiled to either Java or PHP, allowing for rapid development of TAM systems.

CHAPTER 1

INTRODUCTION

1.1. Motivation and Objectives

Every business needs to track the time worked by employees. Many companies that perform work for multiple clients need a more precise time tracking scheme that captures the amount of time spent working on a particular project, so that the corresponding client can then be billed accordingly. The time tracking needs vary based on the nature of a business: what works well at a medical clinic or a lawyer's office might not work at a software development company or a university. Additionally, individuals need to manage time on personal projects, be it paid work, study time, or errands.

The need of tracking time leads to development of time tracking applications. A business can either buy an off-the-shelf application to track its employees' time, pay a subscription fee to access a remotely-hosted application, or develop an in-house system to address their needs. Many time tracking systems have sprung to life – both commercial and open source, – similar in goals but differing in implementation. As a result, unless a company can afford developing an application in-house, they have to settle for one of the existing systems, which might not fit their business model well. This leads to adjustments where a company either has to modify its business processes to fit the time tracking tool,

or has to pay for customization of the tool to suit its needs. This in turn leads to the existence of many versions of a tool, each slightly different from the others, making maintenance a nightmare.

Based on the commonalities among the time tracking tools, as well as some differences among them that address varying business needs, the time tracking tool domain appears to be a good candidate for a product line. Once a product line is created, an application suiting a particular set of business needs can be rapidly developed from the common base. The product line can then be expanded with new features if needed without having to re-develop the functionality used in the existing systems.

This work addresses the following problems:

P1. The need to re-write systems from scratch within a domain where most systems are similar.

P2. Difficulties maintaining multiple versions of the same system that have slight differences.

P3. Inability of the current generation of product line tools to work in a way that is model-driven.

Our objectives to combat those problems are:

O1. Generate a product line for a domain where most systems are similar. This will remove the need to re-write systems from scratch (addressing P1) and simplify maintenance (addressing P2).

O2. Find a combination of product line tools and notations that allows for model-driven development (addressing P3).

O3. Attempt to derive a general methodology from the steps used to achieve O1 (allowing to solve P1 and P2 for multiple domains).

1.2. Audience

The TAM product line research is intended to make the rapid application development possible in the time and activity management domain. Thus, the target audience of this research includes the developers working on TAM systems.

The product line approach, applicable in the TAM domain, can be applied in other domains, in which case the methodology used in this research would be of help. Someone might like to model a more specific case of TAM applications, such as appointment-based time tracking used in clinics, or a course-based time tracking used in universities. Domains other than time tracking might benefit from a product line approach as well. In each case our methodology and modeling language choices would serve as an example. The case studies are

extensively documented (Levin, 2009) with each modeling stage captured in UML models as well as explained in text.

We also showcase the model-driven analysis using the Umple language, which has compact syntax and powerful capabilities. It allows for the model and code synchronization, as well as generation of object-oriented application code. Developers or modelers interested in using Umple in their projects might find our case studies useful.

1.3. Organization

In Chapter 2, we begin with an overview of purpose of software modeling, and several modeling notations. Our focus is primarily on UML and Umple notations, as they are used in our case studies. We also review the research on product families and product lines, focusing on ways to express application variabilities.

Chapter 3 contains case studies of the four time and activity management applications: Klok, Leia, Anuko Time Tracker, and TimeTrex. Klok is a free, stand-alone, closed source application. Leia is a proprietary web-based application, although we do have access to its database design and source code. Anuko Time Tracker and TimeTrex are both open source web-based applications. The case studies are extensively documented: all the models that could not fit into this work can be viewed online (Levin, 2009).

In Chapter 4, we describe the methodology used to derive the time and activity management product line, followed by the detailed step-by-step analysis of each application's features and ways they are represented within the product line. At the end of Chapter 4, we present the complete TAM product line together with invocation examples for the time tracking applications. More examples can be seen online (Levin, 2009).

Chapter 5 contains the summary and discussion of our findings, as well as possible directions for future work.

There are four Appendices, each containing support materials for a particular case study, including an application interface screenshot and use cases addressed by the application. Additional support materials for the case studies can be viewed online (Levin, 2009).

CHAPTER 2

MODELING AND PRODUCT LINES

In our research, we use architectural modeling to provide a high-level overview of time and activity management systems. Modeling allows us to operate on a high level of abstraction, so that we may compare varying systems while disregarding implementation differences. Section 2.1 provides a brief discussion of concerns that are addressed by our models and the details we have chosen to omit.

Time and activity management applications are similar in their data models, as they all keep track of the same information. Thus we have chosen to use UML class diagrams in our case studies to model and compare different applications. A brief description of UML follows in Section 2.2.

UML, being primarily a graphical notation, is not well-suited for automated model processing, analysis, and code generation. To address this, we have turned to Umple – a textual language based on UML, allowing us to express class diagrams as textual models. Umple is described in Section 2.3.

As we are looking at applications that exhibit certain similarities, while still having a few differences among them, we have chosen to treat them as a product line of

time and activity management software. An overview of product lines and product families is presented in Section 2.4.

There are multiple ways to express commonalities and variabilities among products that comprise a product line. Section 2.5 contains a discussion of existing methods, their advantages and shortcomings, focusing on VML – a notation we chose to model the time and activity management product line. Section 2.6 gives an overview of the technologies we selected for our research.

In this work, we analyze and model only the data structures of the applications. This is done for simplicity. Our methodology, however, is not constrained to the data layer. Umple (and thus VML4Umple) is capable of handling arbitrary Java code, so it can theoretically be used for business logic. A more formal representation of logic in state machines and constraints would be an improvement, and is already under development. Automated UI generation from Umple models is also in progress. Thus, our product line model allows for generation of data structures and business logic.

2.1. Modeling Purpose and Notations

According to Taylor et al, "An architectural model is an artifact that captures some or all of the design decisions that comprise a system's architecture. Architectural modeling is the reification and documentation of those design decisions… An architectural modeling notation is a language or means of

capturing design decisions."(Taylor et al., 2009) Additionally, some modeling notations (for instance, UML) are used to capture requirements, functional design, and data design. We have compiled a technical report on modeling notations, which details a variety of Architecture Description Languages (ADLs). Here we summarize our findings. For more detailed view, please see the report itself at (Levin, 2009).

In the report, we have analyzed and compared the following modeling notations: natural language; informal graphical styles; Unified Modeling Language (Booch et al., 1999; Object Management Group, 2008); early Architecture Description Languages – Darwin (Imperial College, 1997), Rapide (Luckham et al., 1995), and Wright (Allen et al., 1997); domain-specific and style-specific Architecture Description Languages – Koala (Ommering et al., 2000), Weaves (Gorlick et al., 1991), and AADL (Feiler et al., 2006); extensible Architecture Description Languages – Acme (Garlan et al., 1997), ADML (Spenser, 2000), and xADL (Dashofy et al., 2005); User Requirements Notation (Weiss et al., 2005; University of Toronto, 2000); and Dialog Flow Notation (Book et al., 2003; Book et al., 2004).

One of the modeling notations covered in our report is especially interesting, as it concerns architecture of a product family. Koala (Ommering et al., 2000) language developed by Philips is used to describe the architecture of consumer electronic devices. Koala models are closely tied to implementation and thus can

be automatically verified for correctness and completeness. Ideally, this is the point at which we would like to arrive with modeling, implementing, and verifying the systems belonging to the time and activity management product line.

Basic architectural concepts captured in the models are components, connectors, interfaces, configurations, and rationale for architectural decisions. Architectures may include static as well as dynamic aspects (those that change over time), and modeling notations exist to capture both. Dynamic aspects can be modeled using static models or dynamic models (those that visualize the behaviour of a running system and are updated on-the-fly). Modeling in our case studies is concerned with static aspects, as we are dealing with use cases and data representations of the systems, neither of which changes at run-time.

Systems may include functional aspects (system's functionality) and non-functional aspects (constraints on what a system does). We focus on modeling functional aspects, as we try to generalize behaviour of several systems.

A model can be associated with several visualizations, where each visualization is a different way of representing the information organized by the model. Some notations, like UML, have a canonical visualization. There are three types of visualizations: textual, graphical, and hybrid. Textual visualizations are easily accessible and editable; they can store the entirety of the model in one file; they can be parsed, processed, and automated if their syntax is associated with a

particular meta-language. However, textual representations are linear and do not work well for graph-like structure depiction. Graphical visualizations are best at depicting spatial information, presenting additional information such as colours, symbols, and other decorations, scrolling, zooming, showing and hiding elements, and being directly manipulated with a mouse. However, they depend on costly tools and cannot be directly used in automated processing. Hybrid visualizations combine graphical and textual elements (such as UML with annotations in OCL (Object Management Group, 2009)).

In our research, we use graphical visualizations to represent the time and activity management systems during the modeling stage. This allows us to determine the similarities and differences in models easily, based on visual models. At later stages, to facilitate automated code generation, we convert the models from the UML graphical visualization to the textual representation in Umple.

Modeling includes a trade-off between flexibility of being able to describe a variety of systems and being able to utilize the semantically powerful features of more strict notations for automating model manipulation. We focus on class diagrams textually represented in Umple, since that allows us to automate code generation.

The majority of models developed during our work are presented in UML, using the UMLet (Auer et al., 2003) graphical modeling tool. To manipulate the models

textually, we use Umple (Forward et al., 2009), a text-based modeling language that compiles to Java or PHP. We have written a converter in Java to transform UMLet XML-based files to the Umple notation. To model the product line variabilities and the invocation for the TAM applications, we used the Variability Modelling Language (VML) (Loughran et al., 2008). The final product line model is thus expressed in our adaptation of VML combined with Umple which we call VML4Umple. To generate a particular system from the domain, we create a file that uses VML invoke() statements to list the required features for the system. The combination of the product line model and the invocation file can then be processed by the VML4Umple compiler that can generate the application code either in Umple, of in an object-oriented language. Currently Java and PHP code generation is supported.

2.2. Unified Modeling Language

Unified Modeling Language (UML) brings together concepts from several earlier notations: Booch diagrams (Booch, 1986), OMT (Rumbaugh et al., 1991), OOSE (Jacobson et al., 1992), and Statecharts (Harel, 1987). It is an extensive notation with multiple viewpoints, allowing for both static (class and use case diagrams) and dynamic (activity and state diagrams) aspect modeling.

UML started as a design modeling language and as of UML 2.0 support has been added for architectural modeling (Taylor et al., 2009). UML is supported by a

variety of open source and proprietary tools. In our research, we use an open-source UML visualization tool called UMLet (Auer et al., 2003). It is available as a stand-alone application as well as an Eclipse plug-in. The models are stored in a notation that uses XML and can be exported to JPG, PDF, EPS, and SVG. In the course of our research, we have also written an UMLet to Umple converter for UML class diagrams, in Java.

UML has several advantages over other notations: there is a multitude of constructs such as classes, associations, states and activities; multiple viewpoints are supported; it allows for static and dynamic aspect modeling; and it is widely adopted. Through a variety of viewpoints, UML can capture the information at different levels of abstraction, thus aiding design and architecture stages of development.

Details of UML semantics can be found in the UML specification (Object Management Group, 2008), or the many books written about it (for instance, (Lethbridge et al., 2005)). In our time and activity management application research, we make use of UML class diagrams and use case diagrams, so below is a short review of the elements involved.

In class diagrams, classes and relationships between them are key elements. A **Class** describes a set of objects that share same operations, relationships, attributes, and behaviour. A **Class** implements one or multiple interfaces. An

Association is a relationship that specifies a connection between objects, such as aggregation, for instance (where a link is between a whole and its parts). **Generalization** is a relationship where child objects (specialized elements) can be substituted for the parent object (a generalized element).

In use case diagrams, actors, use cases available to them, and relationships between the actors are involved. An **Actor** is a role played by a user or a system. A **Use case** is an element that represents a set of actions available to a particular actor. **Actor generalizations** are used to denote overlapping roles by extending use cases available to another actor (Object Management Group, 2008).

2.3. The Umple Language

Umple (Forward et al., 2009) is a model-oriented language family based on object-oriented language concepts. It has support for domain concepts such as classes, attributes, associations with different multiplicities, and several software patterns. State machine support is currently in the works as well. Umple tools are available in IBM's Rational Software Modeler, and as an Eclipse plug-in. There is also the Umple Online application (Forward, 2009a) that allows one to try out Umple without the need to install any software.

We chose Umple based on several advantages it has over object-oriented languages for implementation of our models. Umple produces significantly fewer lines of code with higher readability, as compared to Java or PHP (Forward et al.,

2009). It can be generated directly from the data model of an application, as Umple classes map to database tables, attributes to data fields, and associations to key - foreign key relationships.

Code to manage associations and code to access and modify attributes with single as well as multiple values is generated by the Umple compiler. Thus Umple takes care of the boilerplate code, and the amount of hand-written code is minimized. This in turn minimizes time spent in development and faults encountered during implementation. Umple provides a concrete syntax for key elements of UML class diagrams, thus being a natural choice to model an application's data objects.

Using Umple allows us to take the model-driven approach to the research. We are able to adjust the models and quickly generate the corresponding data objects by converting the UML graphical model created in the UMLet tool to Umple, and compiling Umple code into either Java or PHP. Any small change in a model can be easily propagated to the code base.

2.4. Product Families and Product Lines

In the global markets coarsely segmented by different cultural factors and standards, some segments are too small to warrant independent product development. For such segments, it makes sense to pursue a product family approach, where assets can be reused across products created for different segments. (Kuusela et al., 2000)

The need to design, implement, and maintain applications that have a set of similar functions, but differ from each other based on platform, version, or target audience, has given rise to research in product families, product lines, and process families.

The notion of a program family was first mentioned by Parnas (Parnas, 1976). He explored and contrasted the approaches to development of program family applications.

Several works addressing the subject were published in 1996. For example, Sutton et al. described product and process families, their properties, and relationships between them (Sutton et al., 1996). Cugola et al. looked into requirement definition for process languages (Cugola et al., 1996) based on the work of Parnas (Parnas, 1976). Di Nitto et al. addressed the differences between product families and product lines and identified areas of further research needed (di Nitto et al., 1996).

The majority of the product line research described below deals with analyzing the results of switching to product line development (Cugola et al., 1996; di Nitto et al., 1996). The literature describes how the requirement definition, documentation, organizational processes and development practices are affected (Lutz, 1999; Schmid et al., 2000). Other researchers look into requirements definition for product lines from the start (Kuusela et al., 2000; Ram et al., 1997).

There is also research that deals with modeling commonalities and variabilities (Loughran et al., 2008; Ardis et al., 1999; Dhungana et al., 2009; Acher et al., 2009; Metzger et al., 2007; Mietzner et al., 2009; Sánchez et al., 2009) and product generation (Batory et al., 2002).

Our objective of creation a product line from several similar systems that come from different sources and belong to the same domain is not addressed. We have also not found any description of a generic product line derivation methodology.

We have built on the research of Sánchez et al. in adopting VML for our VML4Umple notation for variability modeling (Sánchez et al., 2009).

2.4.1. Definitions of Product Family/Line

Sutton et al. described a product family as a collection of products similar in some ways and systematically different in others, the emergence of which is usually caused by one of the following: successive revisions of a single application; versions of a single application for different platforms; or versions with different sets of features (e.g. an "educational" versus a "professional" version of the same software) (Sutton et al., 1996).

Di Nitto et al. addressed the lack of a common lexicon when talking about product lines and families, and attempted to derive a definition incorporating existing descriptions (di Nitto et al., 1996). They defined product family as a

"collection of software products based on the same set of assets" (requirements, designs, software components), "but still having significant variations". In contrast, product line was defined as a "collection of different software products, each based on different assets", offering complementary features, designed to jointly operate through integration and interoperation. An example of a product family would be programs with the same features running on different platforms, whereas an example of a product line would be an office suite that includes separate applications: a text editor, a spreadsheet application, etc. - while sharing common assets, such as the help system.

In subsequent parts of this work, we shall be using the term "product line" to describe a group of applications having some commonalities as well as some variabilities.

2.4.2. Issues in Product Family/Line Development

Di Nitto et al. touched upon several issues involved in development of product line software (di Nitto et al., 1996). These include design for reuse, reuse of existing components, configuration tracking, and maintenance of common assets.

Cugola et al. addressed the structure of the design process for a product family, and concluded that process does not necessarily need to be modified to accommodate product families, but it might be more important to follow in case of product families (Cugola et al., 1996).

2.4.3. Case Studies

Ram et al. looked into issues that arise while developing software for the Nokia Synfonet line of products (Ram et al., 1997). Requirements for it include being configurable, distributed, embedded, real-time, dependable, as well as including several optional levels of software and hardware fault tolerance and function protection. The authors suggested that "architecting should start with specifying the partition in different component domains along with a scheme for integration and coordination of the parts". It should also include an explanation of how the partition and its integration address the specified architectural concerns, such as timeliness, capacity, availability, effective division of work, standards compliance, existing parts utilization, or controlled propagation of change.

Lutz has focused on an interferometer (telescope) subsystem design (which was a part of a product family), and the evaluation of this as a reusable component (Lutz, 1999). Several stages of the case study are discussed: product family definition, hazard analysis, and design evaluation.

They suggest that near-commonalities (features common among almost all product family members) can be represented as variabilities. Product family requirements need to anticipate both future feature additions and possible reductions. Dependencies among options need to be represented. Ardis et al.

suggest writing such constraints as commonalities, where the commonality is the required relationship between the parameters of variation (Ardis et al., 1999).

Most of the case studies mentioned above have focused on design of product lines and evaluation of feasibility of product line development. They have also focused on creating product lines either from the very beginning of product development, or from multiple versions of the same system. None of them have described a methodology to derive a product line from several different applications in the same domain, which is a part of our objectives.

Lutz also has touched on the possibility of organizing the products into a hierarchy where products at the same node share the same value for many parameters of variability (Lutz, 1999). They note that it did not add any insight in their application, but might be beneficial for larger product families.

At the end of Lutz' study (Lutz, 1999), a review was conducted by an engineer experienced in interferometers. It resulted in deletion of 9 out of 29 commonalities, increase of variabilities from 23 to 35, and modifications to 4 variabilities. Among the lessons learned, they mentioned thorough documentation of variabilities as a safest course of action, even at the cost of minimizing possible commonalities. Safe reuse largely depends on the underlying assumption of commonalities being true.

Kuusela et al. introduced another hierarchy approach (Kuusela et al., 2000). They proposed a definition hierarchy where design objectives are defined by other design objectives or design decisions (see **Figure 1** for an example of a requirements definition hierarchy). They are arranged into a logical AND tree subject to the following rules:

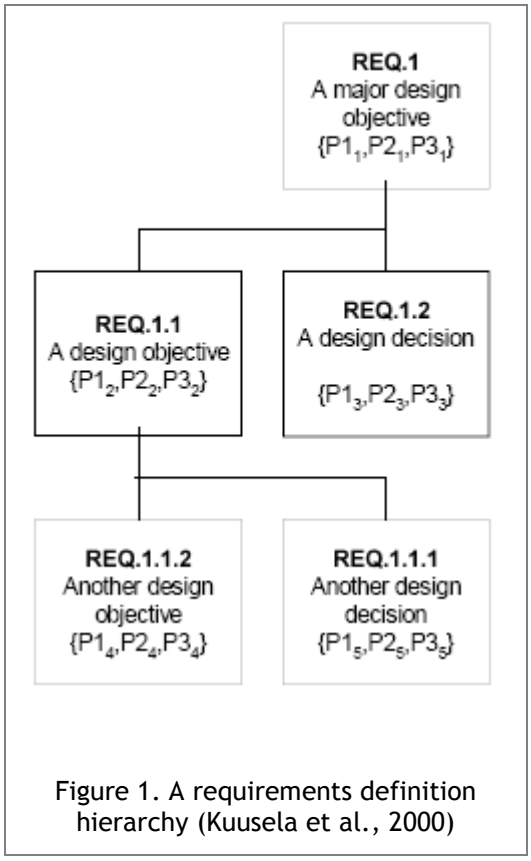

Figure 1. A requirements definition hierarchy (Kuusela et al., 2000)

1) Child requirements define the meaning of parent requirements.
2) Top nodes represent architectural drivers and quality attributes.
3) Edges represent refinement of design objectives and/or design decisions.
4) Each key architectural driver defines a sub-tree under the root node.
5) Parent nodes have higher priority than their child nodes.
6) Lowest priority is "irrelevant" - it represents design decisions or objectives that do not belong to the description of the specific product.

Product families in this approach share most of the quality attributes, even

though their definitions can vary significantly among product family members. This approach simplifies testing: if all sub-nodes pass their tests, the super node can be assumed to pass the test as well. Requirements are more easily visualized in a hierarchy than if they were to be explained through textual description. Structuring helps resolving missing requirements and inconsistencies: by reversing the tree, it can be determined whether the collection of the sub requirements fully and unambiguously defines the super requirement. If that is not the case, some requirement is missing and needs to be added. This process must be repeated for every member of the product family.

The requirements-based hierarchical approach was of interest to us, since isolating similar interdependent clusters of functionality in different systems is similar in nature.

Another case study by Schmid et al. deals with introducing a software modeling concept in a supermarket chain subsidiary company, founded to develop a family of new merchandising information systems (Schmid et al., 2000). They attempted to address the following documentation problems that frequently occur in practice:

- Standard approaches to documentation do not fit real requirements;
- Documentation requirements may change over time and make old documentation and processes obsolete;

- The entrance barrier to changing the documentation approach is high due to re-documentation effort;
- There is a lack of time and expertise to devise a new approach.

Schmid et al. present QIP (Quality Improvement Paradigm) - an iterative, goal-driven framework for the continuous improvement software development, closed-loop process for planning, executing, evaluating improvements to software development environments, and incorporating experience gained from improvement efforts into future development (Schmid et al., 2000). Application of QIP resulted in a brief start-up phase with improved documentation, and also with ease of documentation reaching a reasonably good level. Training, trial usage, active cooperation of the people involved, comprehensive examples, as well as detailed guidance and support in the early stages were needed to successfully complete the project.

A company switching from individual product development to development based on a product line model will likely face the documentation issues described in (Schmid et al., 2000). This might be a factor affecting adoption of our methodology.

2.4.4. Product Generation

Batory et al. look at two classifications of software components: 1) object-oriented: method, class, package; and 2) feature refinement (or layer): module encapsulating a feature (Batory et al., 2002).

Refinements cross-cut classes. Layers (features) are building blocks of software; facets are building blocks of layers. Facets are not classes - they arise when feature refinements encompass more than an individual program or package. Refinements can be broken down into *gluons* - elements arranged in regular ways to form both facets and "atomic" elements.

Batory et al. introduce Origami - a model of gluons, revealing a mathematical structure of software (Batory et al., 2002). It is based on GenVoca (Batory et al., 1992) - a methodology and technology for generation of product lines from feature refinements. Software is extended based on component additions and removals. Origami can be scaled to generate product families as well as standalone products.

GenVoca function (set of classes and class refinements) is applied to a GenVoca constant (set of classes) - some classes are extended and some classes are added. Class extensions encapsulate new data members, methods, and method overrides of the parent class. Linear inheritance chains (or refinement chains) are constructed by application of several functions to the constant, and in the

resulting application, only the bottom-most classes of the chains are instantiated, as they implement all the roles assigned to them.

Layers are orthogonal to facets, so the relationship between them can be expressed as a matrix, where each entry (gluon) lists the name of a module that is implemented as a facet within a layer. If each row is a layer, each column is a facet, and vice-versa.

GenVoca models are 1D - they are composed by the sets of constants and functions. Gluon models are 2D, and can be *n*-dimensional. The Origami model scales to *n* dimensions. Each new row in a matrix requires a gluon for every existing column (or the identity function if no implementation is needed). The same is true if a new column is added. An application is created by folding an Origami matrix.

2.5. Expressing Variabilities

Dhungana et al. present an interesting perspective on variability (Dhungana et al., 2009). They suggest similarity between software variability and genetic variability, comparing individual species to natural product lines and individuals to products. For these natural product lines, both commonalities and variabilities are coded in genes, which can be "turned on" and "turned off", corresponding to each feature to be either present or absent in an individual. Reproduction is compared to product instantiation, allowing for assignment of features to a particular

individual by binding the corresponding variation points. The primary difference between the customization processes in nature versus that in software is the presence of random aspects in nature, whereas software customization is deterministic (Dhungana et al., 2009). Complex dependencies among features in software (similarly to those among genes in biology) are common, making arbitrary combinations of features (or genes) not viable.

To further explore the similarities between the product lines in software versus those in nature, Dhungana et al. map the several other genetic concepts to those of software development. Genotype ("the set of genes present in the DNA of an organism" (Dhungana et al., 2009)) is compared to how variability in software is implemented. Phenotype (the appearance of particular traits in an individual) is compared to the characteristics of a software application visible to the user. Alleles ("alternative forms of the same gene" (Dhungana et al., 2009)) are similar to alternative forms of the same feature.

The work of Dhungana et al. is recent and largely exploratory, inviting the reader to consider the similarities between software product lines and genetics (Dhungana et al., 2009). Considering the success of genetic algorithms in other areas of computer science (such as neural networks, for instance), this approach seems promising.

Archer et al. explore variability expression in video surveillance domain (Acher et al., 2009). They use feature models, one to describe tasks (domain variability) and the other to describe software variability. The authors suggest applying modeling techniques not only to software implementation, but to the specification of the software features as well. To model the variabilities, they use the Feature-Oriented Domain Analysis (FODA) method presented in (Kang et al., 1990) which uses feature diagrams. Features in the diagram are organized by hierarchy, with edges breaking the features down into sub-features. Each feature is represented by a node on the tree and can be either mandatory or optional. FODA also supports "requires" and "excludes" constraints, specifying dependencies among features. The FORM method (an extension of FODA) allows for additional kinds of constraints (Kang et al., 1998).

As Metzger points out, the FODA method focuses on "separating the documentation of *software variability* from the base models" (Metzger et al., 2007) (original emphasis).

Mietzner et al. look at the variability modeling in Software as a Service (SaaS) applications (Mietzner et al., 2009). These applications can be hosted on the same infrastructure for multiple tenants while being customizable to suit each tenant's needs. The authors distinguish two types of variability: external – that "communicated to the customer of the product line" (Mietzner et al., 2009); and internal – that "only visible to the developers of the product line" (Mietzner et al.,

2009). The language used by the authors is the OVM – Orthogonal Variability Model language, presented in (Metzger et al., 2007). It includes variation points and variants. Each variation point documents a variable item. Each variation point can have several variants, which document "possible instances of a variable item" (Mietzner et al., 2009). Both variation points and variants can be either mandatory (they have to be bound) or optional. OVM also supports "requires" and "excludes" constraints.

The authors are undecided on a formalism to use with their model diagrams, suggesting several systems that are worth exploring, such as Object Constraint Language (OCL (Object Management Group, 2009)). They are also looking into transformation engines so that the rules can be executed (Mietzner et al., 2009). Thus even though OVM operates with useful concepts, such as mandatory and optional variation points and variants, there does not appear to be a corresponding textual notation we can use in our models.

Sánchez et al. note that there are several modeling languages involved in product line modeling: (a) a language to specify the variability of the product line; (b) a language for modeling the assets (such as UML), and (c) a language directing the composition of reusable assets (Sánchez et al., 2009). The goal, they state, is to come up with a notation that can handle all the aspects of the software product line modeling. Sánchez et al. use a Smart Home System to demonstrate the

capabilities of the Variability Modelling Langauge (VML) described in (Loughran et al., 2008).

A VML model can include multiple Concerns. Each Concern deals with a particular set of features (a particular cluster of functionality). In the case of time and activity management systems, for instance, there are single-user and multi-user systems (reflected by the User cluster). Thus one of the Concerns would deal with the user-related data and the interaction between a time entry and a user.

Within each concern there could be multiple VariationPoints (either *Optional* - to be optionally included, *Parameter* (the parameter determines the implementation of a feature), or *Alternative* - one of the alternatives must be included). In our model we do not use the *Parameter* VariationPoints, so we shall omit them in further discussions. *Alternative* VariationPoints have multiple Variants within them. A VariationPoint corresponds to a feature, while a Variant represents a possible implementation of that feature. In the time and activity management systems, for instance, the time can be entered as duration (e.g., 1 hour) – which would correspond to Variant 1, or as a combination of start time and end time (e.g., 10:00 to 11:00) – corresponding to Variant 2.

To derive a model for a particular system, a series of invoke statements is used. Each statement can either invoke a single Variant within a particular *Alternative*

VariationPoint of a particular Concern (see (1) below); or invoke a single *Optional* VariationPoint of a particular Concern (see (2) below).

1) invoke (ConcernA, VariationPointB, VariantC) - include the code from VariantC of the *Alternative* VariationPointB in ConcernA.

2) invoke (ConcernA, VariationPointD) - include the code from the *Optional* VariationPointD in ConcernA.

Figure 2 shows examples of an optional and an alternative variation point.

VML seems the most promising for the purpose of expressing variabilities in the time and activity management product line. It is hierarchical, allowing for modeling features belonging to different feature sets. Concerns map to clusters of functionality (a set of related features), VariationPoints to features, and Variants to feature implementations. VML also allows for modeling dependencies among features. We have combined VML with Umple which is used to model the reusable software assets. Umple allows us to define code in fragments, while the compiler takes care of combining the fragments into class definitions. The Umple compiler generates Java or PHP code directly from the Umple model.

The combination of VML and Umple allows us to describe the product line and invoke specific features for a particular product, as well as generate the corresponding object-oriented code directly from the product line model. We call

this combined language VML4Umple, following the naming convention presented in (Sánchez et al., 2009).

.

```
Concern CTimeEntry{
  // time entry can have a rejection comment
  VariationPoint VPTimeEntryRejectedComment{
    Kind: Optional;
    class TimeEntry{
      String rejectedComment;
    }
  }
  // either duration or both start and end time are required
  VariationPoint VPEntryDuration{
    Kind: Alternative;
    Variant VDuration{
      class TimeEntry{
        Double duration;
      }
    }
    Variant VStartEndTime{
      class TimeEntry{
        Time startTime;
        Time endTime;
      }
    }
  }
}
```

Figure 2. Examples of an optional and an alternative variation point.

VML is still evolving and some of the syntax has changed since the papers on which we base our work have been published. However, since the variability elements that need modeling stay the same, our syntax can be adjusted once the VML notation becomes stable. Our methodology and the results of our research, however, should not be affected.

2.6. Selected Technologies

Modeling is key in software development, be it architectural modeling, software specifications, or software asset modeling. Keeping models consistent with the implementation is an important aspect of software documentation and maintenance.

UML is an effective notation to model various aspects of a software system. We are primarily interested in modeling use cases and data structures of the time and activity management systems. UML is well-suited for both.

Furthermore, expressing our UML models in the Umple textual notation gives us the ability to quickly edit the models and automate some stages of implementation, such as generation of the application code.

Product lines allow for asset reuse. Developing products via product lines allows for speedy development, incremental testing, and streamlined deployment. Thus our intention of creating a product line for the time and activity management applications should allow for efficient creation of customized time tracking systems.

To model the TAM product line, we have adopted the VML4Umple notation – a combination of VML control structures with the software assets modeled in

Umple. This allows us to express the product line and product invocations in textual notation, providing means for automated product generation.

CHAPTER 3

CASE STUDIES

3.1. Methodology

To understand time and activity management applications and to see how they could be considered cases of an abstract product family, we have conducted the case studies described below. We have selected four applications: Klok - a small one-person time tracking system; Leia and Anuko Time Tracker - two medium-size multi-user applications; and TimeTrex - a large multi-user time and activity management system. See **Table 1** for the feature summary of the four applications.

For each of the four applications that we have chosen for analysis, we followed a number of steps, to make sure they are analyzed in a consistent manner. We had access to the source code for three out of four applications: Leia, TimeTrex, and Anuko Time Tracker. Klok – the simplest time tracking application we analyzed – did not have the source code available.

Features and characteristics	**Klok** (McKeown, 2009)	**Leia** (Lixar I.T. Inc., 2009)	**Anuko TimeTracker** (Anuko International Ltd., 2009)	**TimeTrex** (TimeTrex Payroll Services, 2009)
License	Free, closed-source	Commercial, closed-source	Open source	Open source
Users	Single-user	Multi-user	Multi-user	Multi-user
Database tables	2	55	16	99
Time Entry	✓	✓	✓	✓
Project	✓	✓	✓	✓
Task	--	✓	✓	✓
User	--	✓	✓	✓
Company	--	✓	✓	✓
Helper	--	✓	✓	✓
Pay Period	--	✓	--	✓
Accrual	--	--	--	✓
Cron	--	--	--	✓
Department	--	--	--	✓
Help	--	--	--	✓
Hierarchy	--	--	--	✓
Holiday	--	--	--	✓
Message	--	--	--	✓
Pay Stub	--	--	--	✓
Policy	--	--	--	✓
Tax	--	--	--	✓

Table 1. Case study applications - summary and feature comparison

First, we created the class diagram for an application, based on its user interface (UI). Secondly, if we had access to the database schema, we generated the Entity Relationship Diagram (ERD) from it, and used the ERD to create another, more accurate, class diagram for the application. Both diagrams were modeled using the UMLet modeling tool (Auer et al., 2003). We analyzed the differences between the class diagrams modeled from the UI and those modeled from the ERD, to determine the reasons for the discrepancies (if there were any).

Thirdly, by looking at all four applications, we determined similar clusters of functionality present in more than one application, and attempted to group together classes related to the same cluster. Each cluster is a group of classes that work together to provide a particular feature of a system. For instance, classes related to project management, project review, project milestones, etc. are all grouped into the Project cluster.

We modified the class diagrams by colour-coding separate clusters. To assist in this step, use cases were created for each application, helping us isolate the common usage patterns within the domain of time and activity management.

To simplify use cases, instead of creating separate use case for creation, updating, listing, and deleting entities, we used a combined use case for CRUD (Create, Read, Update, Delete) for a particular entity if the user type (actor) was allowed to

fully manage it. If the user type was only allowed to create an entity, the use case was included only for the "Create" action.

To avoid confusion due to naming differences, for generalization purposes we use the definitions of project, activity, and task as they appear in Leia: project is a large unit of work done for a client (the organization or individual itself can be a client as well); activity is a unit of work to support the main project work (such as project management, human resource management, company meetings, etc.); and task is the smallest unit of work that can belong to a project or an activity. All three of these are commonly referred to as work items.

Fourthly, we looked at the clusters with functionality not directly related to time tracking, such as overtime and accrual policies, payroll, currency management, helper system information, invoicing, and so on. Classes related to these clusters were then removed from the class diagrams, leaving only the classes that represent the core functionality for time tracking.

At this stage, we compared the four resulting colour-coded class diagrams, to determine the common elements as well as the variabilities between systems, together with the constraints and dependencies among the system elements.

For each system, we translated the complete UML class diagram and the reduced diagram with only the core classes into the Umple language (Forward et al., 2009), using a helper script we wrote. We then compiled the Umple code generated

from the reduced diagram of the studied application, to produce skeleton time tracking systems in Java and PHP.

Below are more detailed descriptions and highlights of the four case studies described above: Klok (3.2), Leia (3.3), Anuko Time Tracker (3.4) and TimeTrex (3.5). Some support materials for the case studies (application screenshots, class diagrams, and use case diagrams) are included in Appendices A through D. The complete set of support materials can be viewed online (Levin, 2009).

3.2. Klok

Klok (McKeown, 2009) is a single-user time-tracking application, aimed at logging time for personal and professional projects. Klok keeps track of projects and sub-projects, allowing explicit time allocation to them. It also provides a timer that a user can manually start and stop to keep track of time spent working on a particular project. Klok allows for view and export of weekly timesheets. It also has functionality for viewing weekly, monthly, and project-based summary reports.

We have modeled the structure of Klok by analyzing the user interface only, as the source for the application is not available. As can be seen in **Figure 3**, Klok consists of the Project class storing project name, related contact information, work time estimate, archived status, and colour code. Projects can have zero or more sub-projects.

The time entries are logged against a project. Because the application is simple, there are only two clusters that can be isolated, each consisting of one class: Project class belongs to the **Project** cluster, and TimeEntry class belongs to the **Time Entry** cluster. Klok is the simplest system that we have analyzed. Time entry is the central functionality of time tracking applications, keeping records of dates and times worked. Klok expands the base case of time tracking by adding the Project class, allowing the time entry against a particular project, thus making logging and reporting more precise. Since even in personal time tracking multiple projects are usually involved, both classes of the application would be essential even for a single-user system (more so for a multi-user one). Thus for Klok the cluster diagram was not further reduced.

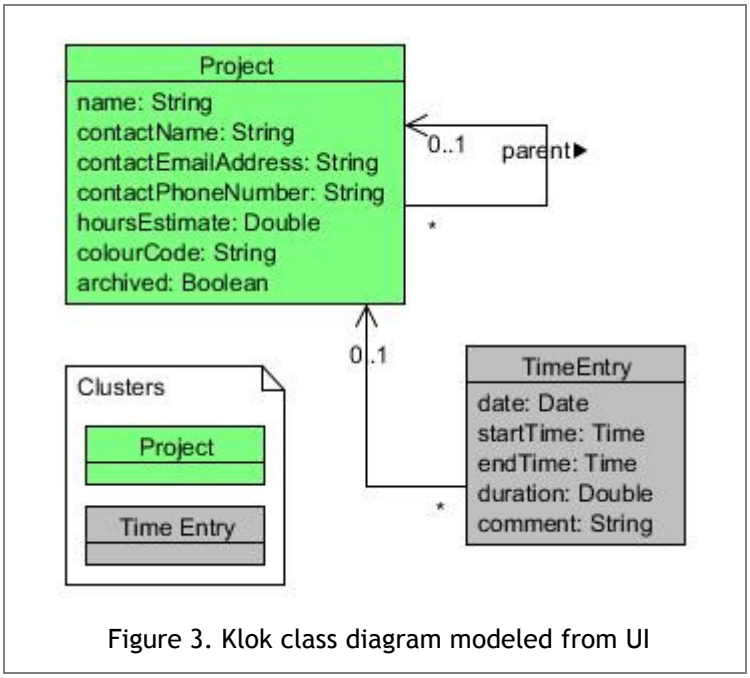

Figure 3. Klok class diagram modeled from UI

The Umple code resulting from the Klok class diagram is shown on **Figure 4**. To view the Java and PHP code generated by the Umple compiler, please see Appendix A.

```
class Project{
 String name;
 String contactName;
 String cotactEmailAddress;
 String contactPhoneNumber;
 Double hoursEstimate;
 String colourCode;
 Boolean archived;
}

class TimeEntry{
 Date date;
 Time startTime;
 Time endTime;
 Double duration;
 String comment;
}

association {
 0..1 Project parent  <- * Project;
}
association {
 1 Project <- * TimeEntry;
}
```

Figure 4. Klok architecture expressed in Umple

3.3. Leia

Leia is a multi-user application developed by Lixar I.T. (Lixar I.T. Inc., 2009), allowing for several user types, each having different access permissions. There are two main work entities in Leia: project (an item of work to be done for a particular client) and activity (support work item, such as human resource

management, infrastructure management, time off, etc.). Smaller units of work – tasks – can be created within projects and activities. Time can be logged against tasks, or directly against a project or an activity.

Figure 5. Leia single task view screen

The use case diagram is included in Appendix B. Employees can create and manage tasks, log time against tasks, projects, and activities, export timesheets, and adjust their preferences. Payroll users can create and manage pay periods and approve timesheets. Division managers can approve timesheets and run time

entry reports. Project managers can create and manage projects, assign project roles, and approve time entry. Administrators can do all of the above, as well as create service items, project codes, client and user accounts, grant access rights, and run extensive reports.

First, we created a class diagram for Leia from the UI. To demonstrate this approach, we include **Figure 5** which shows a screenshot of the Leia single task view screen. This screen includes a task's name, description, priority, due date, estimated hours, and status attributes. These correspond to the attributes of the Task class. The associations of the Task class can be deduced from this screenshot as well: a task is associated with an activity, with one or more users to whom it is assigned, as well as with a user that created the task. This corresponds to the associations between Task and Activity classes, and two associations between Task and User classes. The other classes and associations were modeled through similar analysis of the user interface screens.

To see how accurate the class diagram created from the UI was, we then generated the ERD (since we did have access to the database schema for Leia) and created a more precise class diagram based on the ERD. There were several differences between the two models. The UI-based model is shown in **Figure 6**, the table discusses the differences between the UI- and the ERD- based models can be seen on **Figure 7**. The ERD-based model is shown on **Figure 8**.

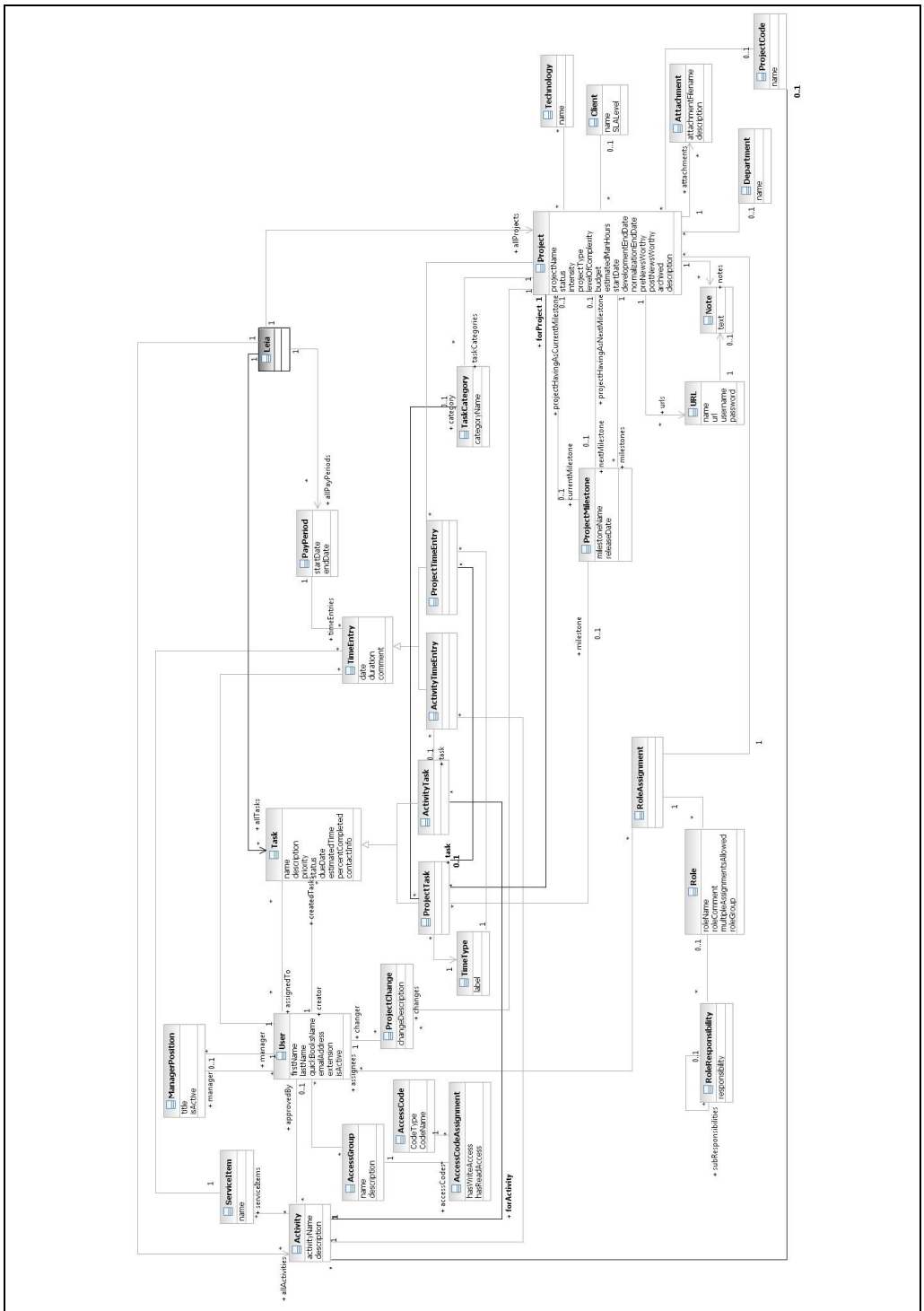

Figure 6. Leia class diagram modeled from user interface

Module	Leia from ERD	Leia from UI	Notes
Client	Client	Client	
Project	Project	Project	
Project	ProjectURL	URL	
Project	*enum in code*	Department	
Project	ProjectMilestone	ProjectMilestone	
Project	Technology	Technology	
Project	ProjectTechnology	*association between Project and Technology*	
Project	TechnologyGroup	--------	*dropdown option groups*
Project	QBBillingCode	--------	*not obvious from UI*
Project	QBProjectCode	ProjectCode	
Project	ProjectType	*attribute of Project*	
Project	ProjectIntensity	*attribute of Project*	
Project	ProjectStatus	*attribute of Project*	
Project	ProjectHistory	ProjectChange	
Project	ProjectReviewScale	--------	*locked conditional access*
Project	ProjectReview	--------	*locked conditional access*
Project	ProjectReviewSignature	--------	*locked conditional access*
Project	TimelineExpectationsScale	--------	*locked conditional access*
Project	ProjectReviewResource	--------	*locked conditional access*
Project	ProjectReviewResourceRole	--------	*locked conditional access*
Project	ProjectUserRole	*association between User and ProjectTask*	
Project	ProjectUserRoleHistory	--------	*not obvious from UI*
Activity	Activity	Activity	
Activity	QBActivityCode	ServiceItem	
Activity	ActivityQBActivityCode	*association between Activity and ServiceItem*	
User	Role	Role	
User	RoleGroup	*attribute of Role*	
User	RoleGroupItem	--------	*association between Role and RoleGroup*
User	AccessGroup	AccessGroup	
User	UserAccessGroup	AccessGroup	
User	AccessCode	AccessCode	
User	AccessCodeType	*attribute of AccessCode*	
User	AccessFlagsGroup	AccessCodeAssignment	
User	AccessFlagsUser	AccessCodeAssignment	
User	User	User	
User	UserFilterSession	--------	*not obvious from UI*
User	UserManager	*association between User and ManagerPosition*	
User	Manager	ManagerPosition	
Payroll	PayrollTimesheet	--------	*locked conditional access*
Payroll	PayrollTimesheetStatus	--------	*locked conditional access*
Payroll	PayPeriod	PayPeriod	
Task	TaskUser	*association between User and ProjectTask | ActivityTask*	
Task	Task	Project Task, Activity Task	
Task	TaskHistory	--------	*missing*
Task	TaskStatus	*attribute of Task*	
Task	TaskAssignmentHistory	--------	*missing*
Task	WorkItemNote	Note	
Task	WorkItemAttachment	Attachment	
Task	WorkItemTaskCategory	TaskCategory	
Task	WorkItem	Task	
Task	WorkItemHistory	--------	*missing*
TimeEntry	WorkItemUserTime	TimeEntry, ActivityTimeEntry, ProjectTimeEntry	
TimeEntry	TimeType	TimeType	
TimeEntry	WorkItemUserTimeStatus	*attribute of Task*	
System	Error	--------	*not obvious from UI*

Figure 7. Comparison between UI- and ERD-based class diagrams

Some of the classes have not been modeled due to locked conditional access to the functionality they represent (such as ProjectReview and associated classes, as they are only accessible to the Project Manager for a particular project). Some

other classes were not obvious from the UI (such as QBBillingCode which is a supporting class to associate projects with those defined in QuickBooks accounting software; or UserFilterSession that keeps track of user-defined filters). Some classes have been modeled as associations (RoleGroupItem class is represented in the diagram as an association between Role and RoleGroup). Others have been modeled as attributes of the classes they described (such as ProjectType being modeled as the attribute of the Project class).

Based on the use cases, we have isolated the following clusters within Leia structure: **Pay Period** – payroll functionality, such as pay period management and timesheet approval; **Project** – functionality dealing with projects, their history, status, milestones, clients, user roles within a project, and project reviews; **Time Entry** – everything that has to do with logging time against a work item; **Company** – internal company project and activity codes, as well as their mapping to QuickBooks codes; **User** – user management, user session filters, role groups within the company (such as Project Manager, Developer, Graphic Designer), roles, and division management; **Task** – small units of work against which time is being logged, their status, history, attachments, and categories; and **Helper** functionality – helper classes that deal with user permissions, error handling, and other cross-cutting concerns. The colour-coded clusters are shown on **Figure 8**.

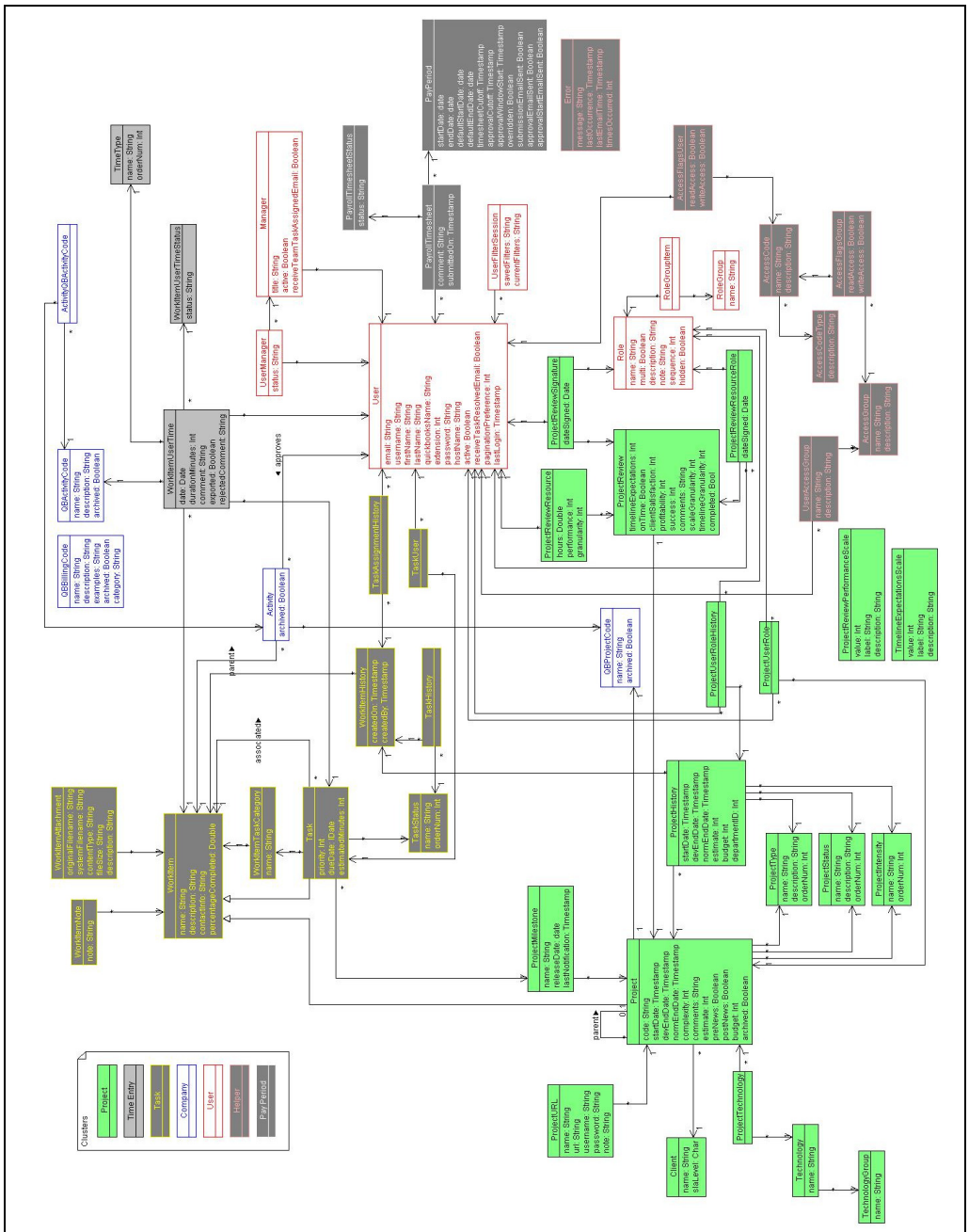

Figure 8. Leia class diagram modeled from the ERD with highlighted clusters

Compared with Klok, Leia is a more complex application. By looking at the two together, we considered how we can reduce Leia to Klok without losing functionality absolutely required for a time and activity management application. To accomplish this, we started with the full class diagram that had clusters highlighted in different colours, and one-by-one removed the classes deemed unnecessary. The reduced diagram is shown on **Figure 9**. Below is the explanation of the reasoning why the particular classes are to be removed.

First, we kept the **Time Entry** cluster as it is the essence of time-tracking applications. Another cluster that has to be present is the **Project** one. However, several classes within that cluster can be removed from the diagram without affecting time and activity management functionality of Leia.

We chose to exclude the ProjectReview class and its associated classes, as this functionality differs among companies and often takes place offline. ProjectTechnology and its associated classes can also be removed, as it either can be omitted (in case a company is not a technology company, or if the company uses the same technology for all of its projects) or can be modeled as an association of the Project class. ProjectURL can be modeled as an attribute of the Project class. ProjectHistory and ProjectRoleHistory classes keep a log of changes to the project and so are helper classes that can be left out of the model as housekeeping functionality. This also applies to the classes in the **Helper** cluster

and the UserSessionFilter class, which have been removed from the diagram as well.

Klok does not include user management, as it is a single-user application. However, most time and activity management application include multiple users. Usually there are at least two user types: Employee and Manager (which also might be called Supervisor or Admin). Therefore, for Leia, we chose to leave in the **User** cluster, including the Manager, UserRole, and UserRoleGroup classes. The latter defines positions within the company and is also often present in time and activity management systems.

Internal project and activity codes included in the **Company** cluster were also removed as these are specific to this application. We did keep the Activity class since it is central to activity management.

The **Task** cluster has been trimmed to remove WorkItemAttachment and WorkItemNote that can be modeled as WorkItem attributes. TaskCategory has been removed as it is a helper class that provides additional grouping. Other companies might have other ways of grouping tasks, such as parent-child relationships, for instance. TaskStatus can be modeled as a Task attribute. The TaskHistory, WorkItemHistory, and WorkItemAssignment classes have been removed based on the same reasoning as the ProjectHistory class.

We have removed the entire **Pay Period** cluster as it is specific to payroll practices of a particular company. Some time and activity management applications will not have payroll at all (personal or academic activity management, for instance), others will have contractors invoicing the company at irregular intervals. Therefore, the **Pay Period** cluster is not essential for the time and activity management applications.

The resulting diagram contains five clusters: **Time Entry**, **Project**, **Task**, **Company**, and **User**. Compared to the simplest application, such as Klok, we have added user and company management. Project and task management can be considered to be a part of the same module, where Project is a grouping encompassing multiple other, smaller, projects – tasks. Klok has a similar functionality by allowing for creation of child projects.

The reduced class diagram shown in **Figure 9** contains fewer classes than the full one, and so the Umple code generated from the reduced diagram is more compact. From the Umple code we have generated Java and PHP classes for Leia in a manner similar to Klok. The Umple, Java, and PHP code for Leia can be viewed online (Levin, 2009).

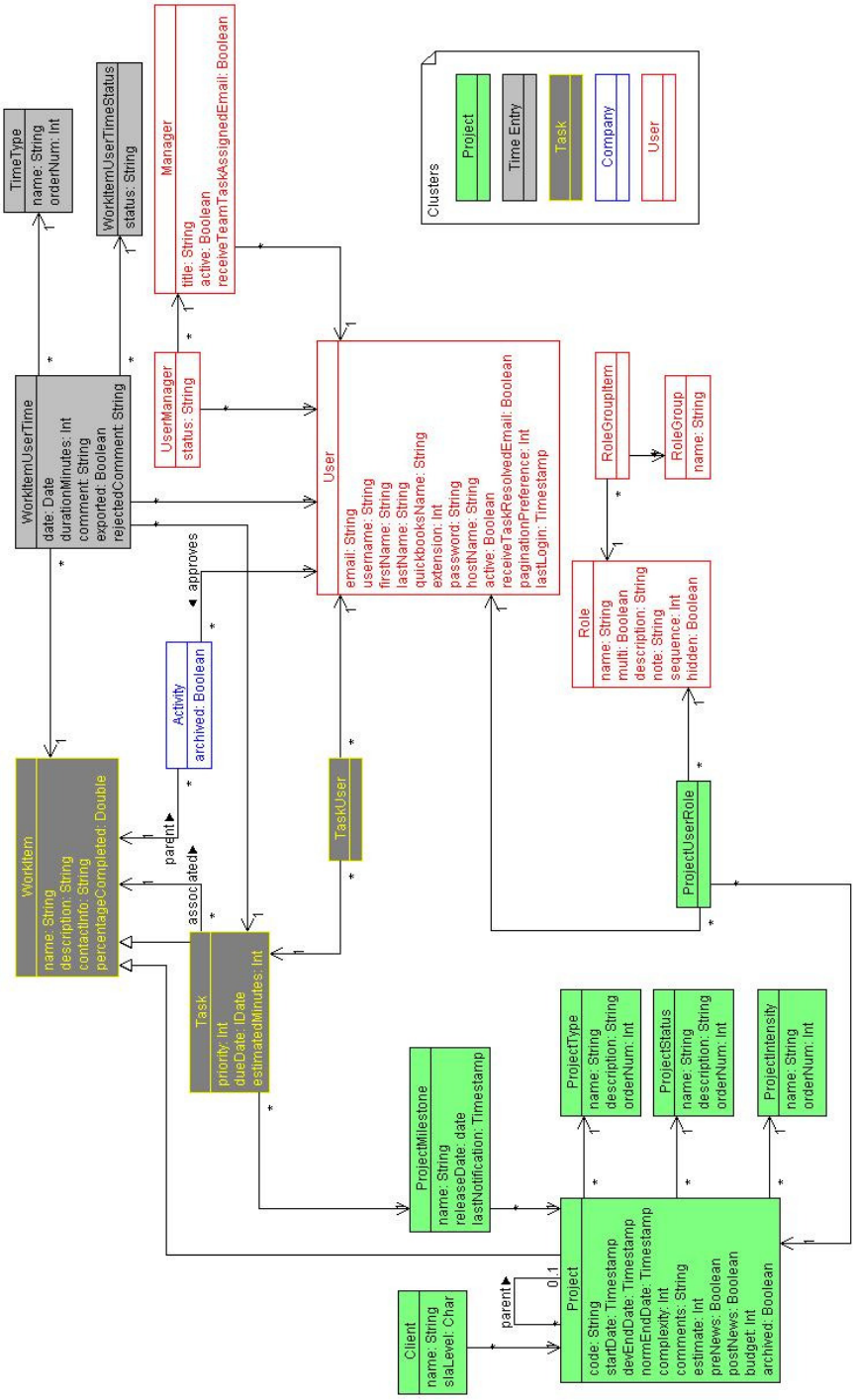

Figure 9. Leia reduced class diagram modeled from the ERD with highlighted clusters, showing only the clusters related to time and activity management

3.4. Anuko Time Tracker

Anuko Time Tracker (Anuko International Ltd., 2009) is a multi-user application with two user types differing in access permissions: Employee and Manager. Employee users can edit their profile, enter time, view and export their timesheets. Manager users can do all an Employee user can do, as well as create projects, tasks, users, clients, and export the timesheet data. The use case diagram is included in Appendix C. This application is slightly more complex than Klok, but simpler than Leia, due to fewer user types and use cases.

In Anuko Time Tracker, there are projects to designate a larger unit of work and activities to represent smaller units of work. In the use case diagram, we kept to the previously stated definition of work items, so activities are specified as tasks. However, in the class diagrams, to keep consistent with the application's ERD, we kept the table names as class names. Therefore, the Activity class in the class diagram maps to the "CRUD Task" use case in the use case diagram, because it deals with creation and management of the smallest units of work within the system.

We have modeled the full Anuko Time Tracker class diagram based on its ERD. The ERD is included in Appendix C, and the class diagram is shown in **Figure 10**. As in the previous case studies, we have identified clusters within the class diagram, to group the similar functionality. Once again, we colour-coded similar

clusters to those of Leia and Klok: **Project** – corresponding to projects and users involved in them; **Users** – containing user information and preferences; **Company** – containing company and client entities; **Task** – classes dealing with "activities"; **Time Entry** – ActivityLog class; and **Helper** – TmpRef class used in user creation, report filters, and system configuration data.

In the Anuko Time Tracker ERD, the client records are not connected to particular projects, as they are in Leia. Therefore, instead of putting the Client class into the **Project** cluster, as we did for Leia, it was put into the **Company** cluster, together with other data defined for the company using the product.

After colour-coding the separate clusters, we once again went through the class diagram removing classes that we deemed not necessary for time and activity management applications. This included all the classes in the **Helper** cluster, as well as the InvoiceHeader class needed only for specifying information related to invoices, and thus not required.

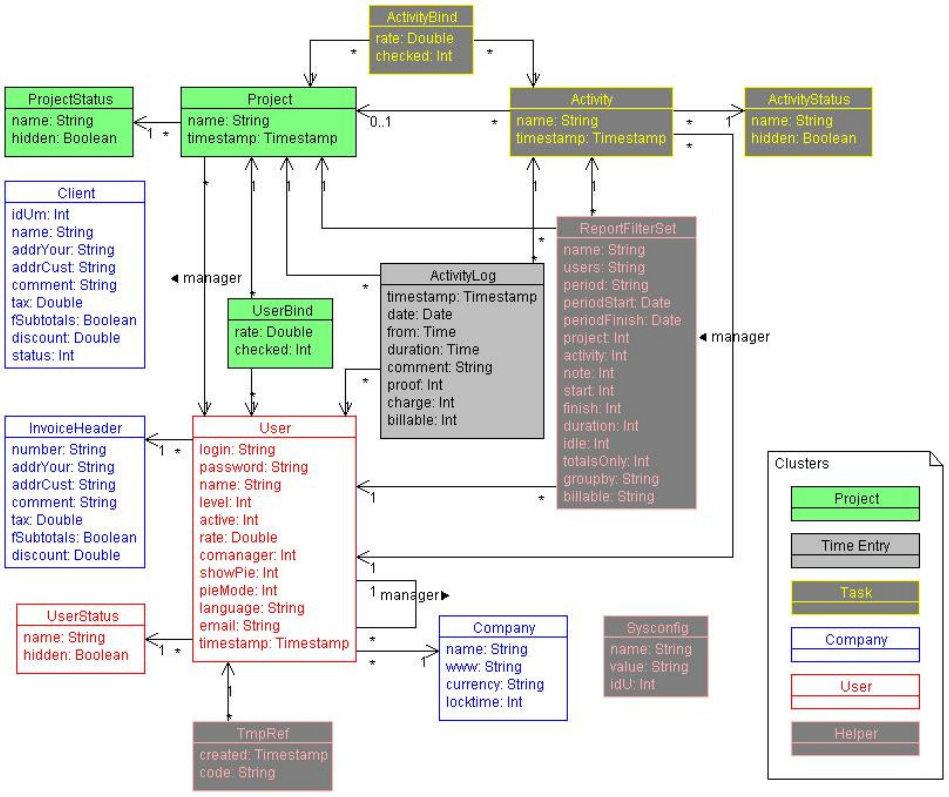

Figure 10. Anuko Time Tracker UML class diagram modeled from the ERD with highlighted clusters

All the other clusters were kept, arriving once more to the five core clusters: **Project**, **User**, **Company**, **Task**, and **Time Entry**. The Anuko Time Tracker reduced class diagrams with clusters highlighted is shown in **Figure 11**.

We have generated Umple code, and from it in Java and PHP for both full and reduced versions of the class diagram. The code can be viewed at (Levin, 2009).

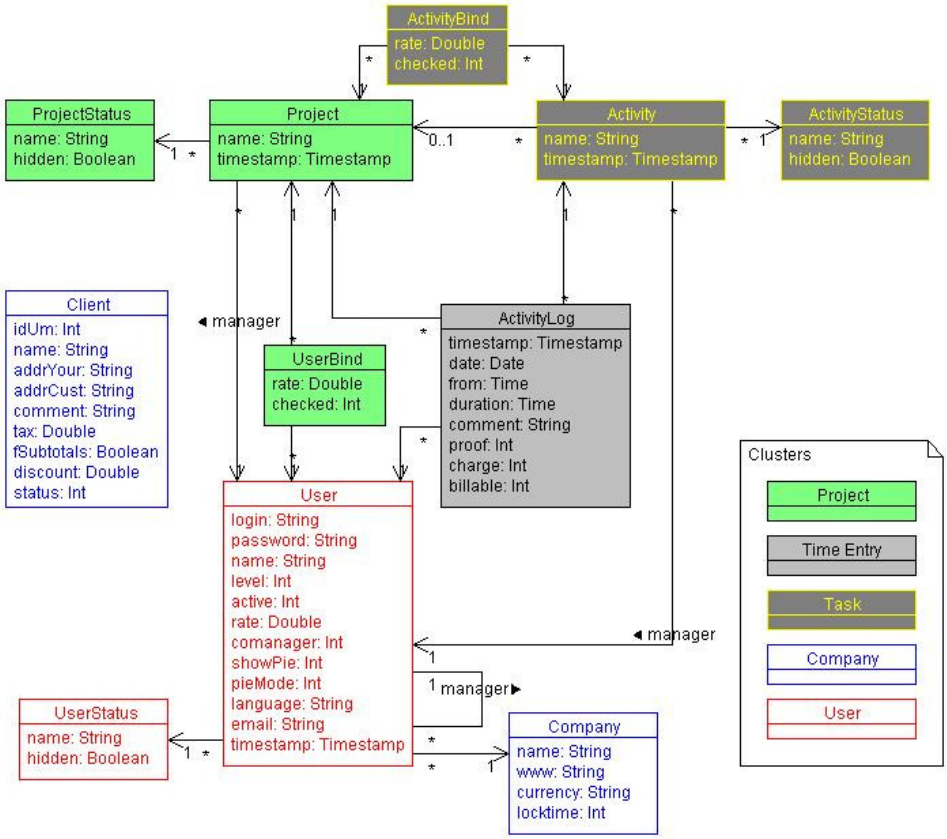

Figure 11. Anuko Time Tracker reduced UML class diagram modeled from the ERD with highlighted clusters, showing only the clusters related to time and activity management

3.5. TimeTrex

So far, we have looked at the very small application with minimum functionality required for time tracking (Klok) and two medium-size applications (Leia and Anuko Time Tracker) which also include user management, company information, and at least two types of work items: projects and tasks. The fourth test case (TimeTrex) deals with a large application, supported by 99 database

tables. Our goal was to see if we can isolate the same functionality clusters within this large application and create from them an implementation with minimal functionality that would still support time and activity management tasks.

TimeTrex (TimeTrex Payroll Services, 2009) is a multi-user system which includes two user types: Employee and Administrator. The use case diagram is included in Appendix D. Employee users can enter time, view and export their timesheets, edit their profile and preferences, create and manage messages, and create requests directed to managers. Administrator users can do everything Employee users can, plus authorize, decline, and pass requests; create and manage tasks, projects, users, clients, task and job groups, recurring and current schedules, user access rights, company information, policy and payroll information; create, manage, and export lookup data; and run administrative reports.

We have created the TimeTrex class diagram based on the user interface. It contains 21 classes, the attributes and associations for which we could deduce from the UI. This diagram is shown in **Figure 12**.

TimeTrex is an Open Source application, so we were able to create a database and generate the ERD from it. Unfortunately, since the TimeTrex MySQL database uses an engine that does not support foreign keys, we had to determine the associations between classes ourselves, based on database field names that

referred to other tables. In the case of Anuko Time Tracker we were faced with the same issue; however, it did not hold us back significantly, due to the small number of tables involved. For the TimeTrex database, the task was much more complicated, because of the sheer size of the database and the number of associations among entities. Several tables did not appear to exist in the database, yet their IDs were referenced from other tables, and so we created classes for those. In the class diagram, they are identified using the fuchsia colour (for those classes that appear to deal with hierarchy of objects in the application) and gray (for the rest of the objects referenced from other tables).

Once the class diagram was completed, we attempted to identify classes belonging to common functionality clusters. We have categorized the classes into 17 clusters: **Accrual** – data on employee time accrual; **Company** – company and branch information, company deductions, user count within the company; **Helper** – authentication and station information, error handling, system log, bread crumbs, user date totals, etc.; **Cron** – data used in running cron jobs; **Department** – department information; **Help** – data used in the help system; **Hierarchy** – object hierarchy data; **Holiday** – public holidays; **Message** – messages passed between users; **Pay Stub** – pay stub entries, as well as current and recurring amendments; **Pay Period** – payroll data; **Policy** – exception, accrual, meal, overtime, premium, and absence policies; **Project** – job data against which time is logged; **Task** – recurring and current schedules; **Tax** –

income tax forms and rates for Canada and USA; **Time Entry** – punch in/out data; and **User** – identification, preferences, wages, status, and other data on users within the system.

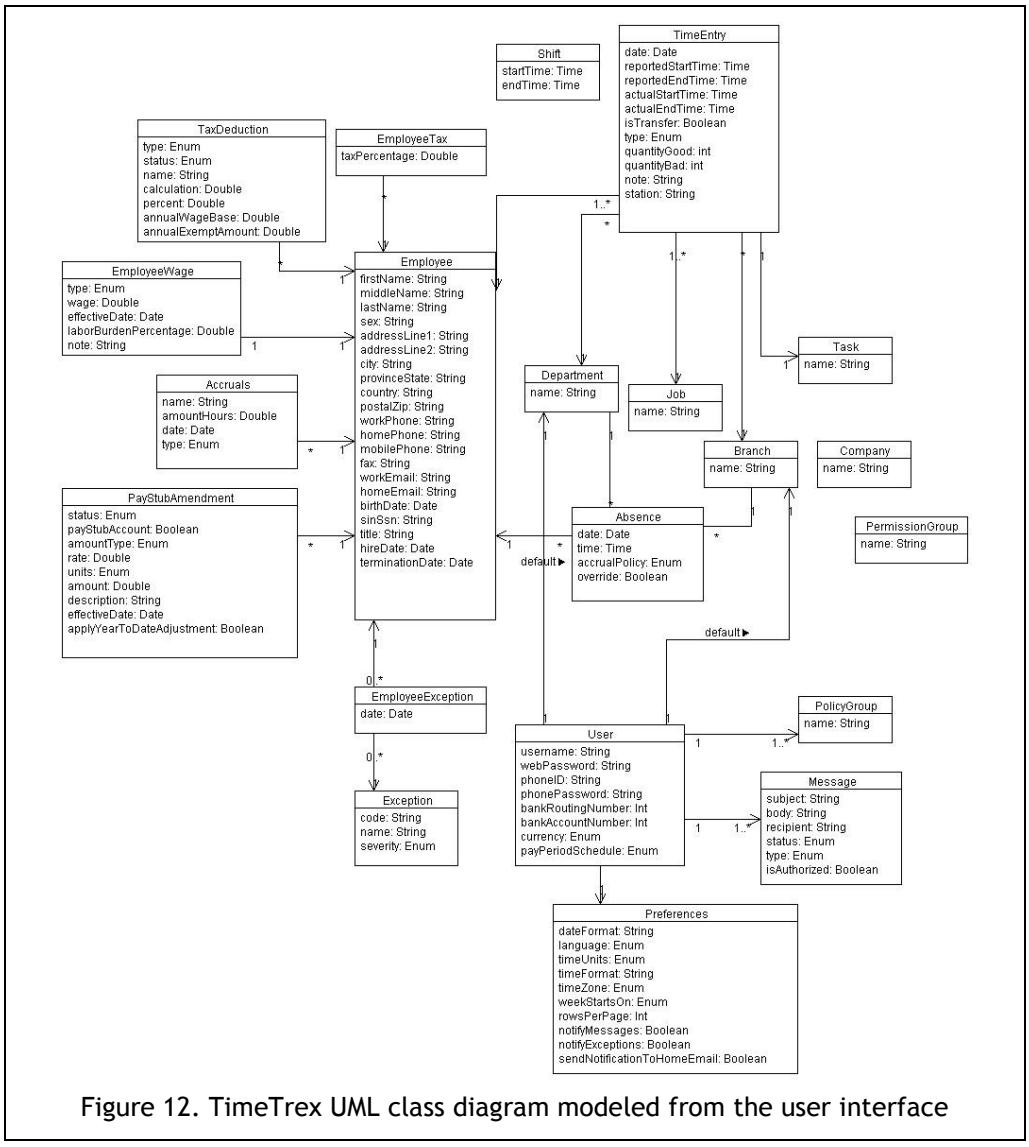

Figure 12. TimeTrex UML class diagram modeled from the user interface

The diagram with clusters highlighted is shown in **Figure 13**. Due to the complexity of the application and the number of classes and associations, the diagram is rather large, and so is not readable. We include it here for purposes of illustration of the size and complexity of the TimeTrex system. A large version of the diagram can be viewed online (Levin, 2009). To assist further discussion, we are also including an abstracted diagram showing all 17 clusters and relations between them, in **Figure 14**.

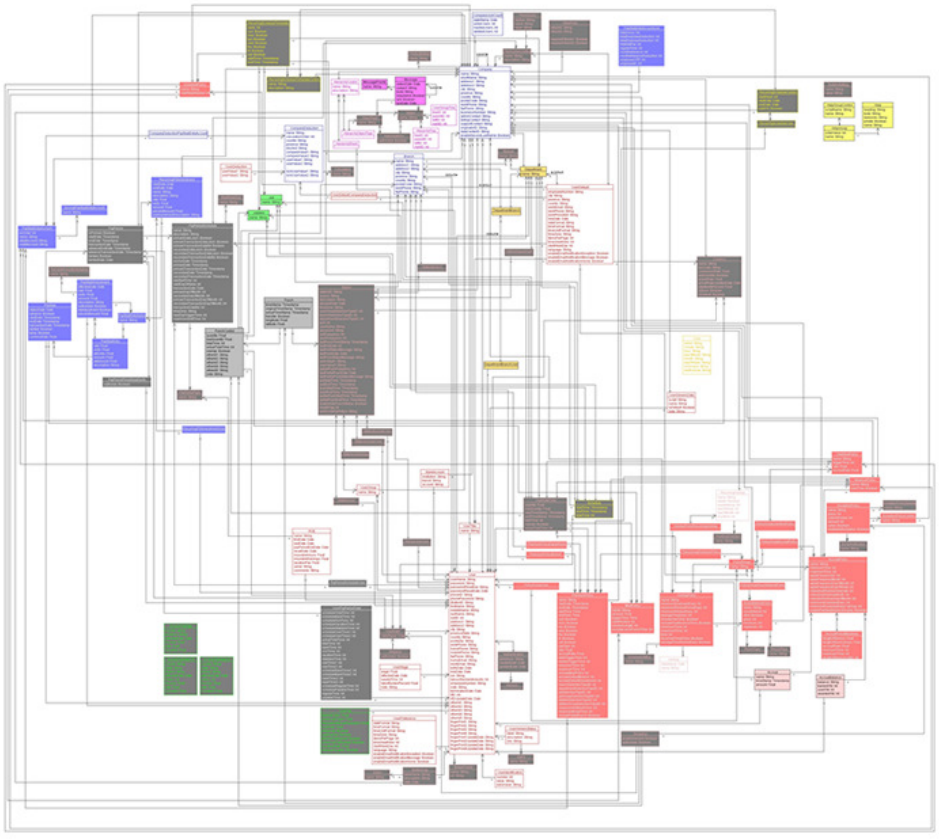

Figure 13. TimeTrex UML class diagram modeled from the ERD with highlighted clusters

For the large systems such as TimeTrex we could have created several smaller UML package diagrams with defined interfaces. This would be useful if we were to examine each cluster (or package) on its own and have a more clear view of the interfaces among clusters. However, the huge numbers of associations (such as those among TimeTrex classes) would potentially result in the package diagrams being more complex than one large model, making it harder to view all the relationships within the model and to see the big picture.

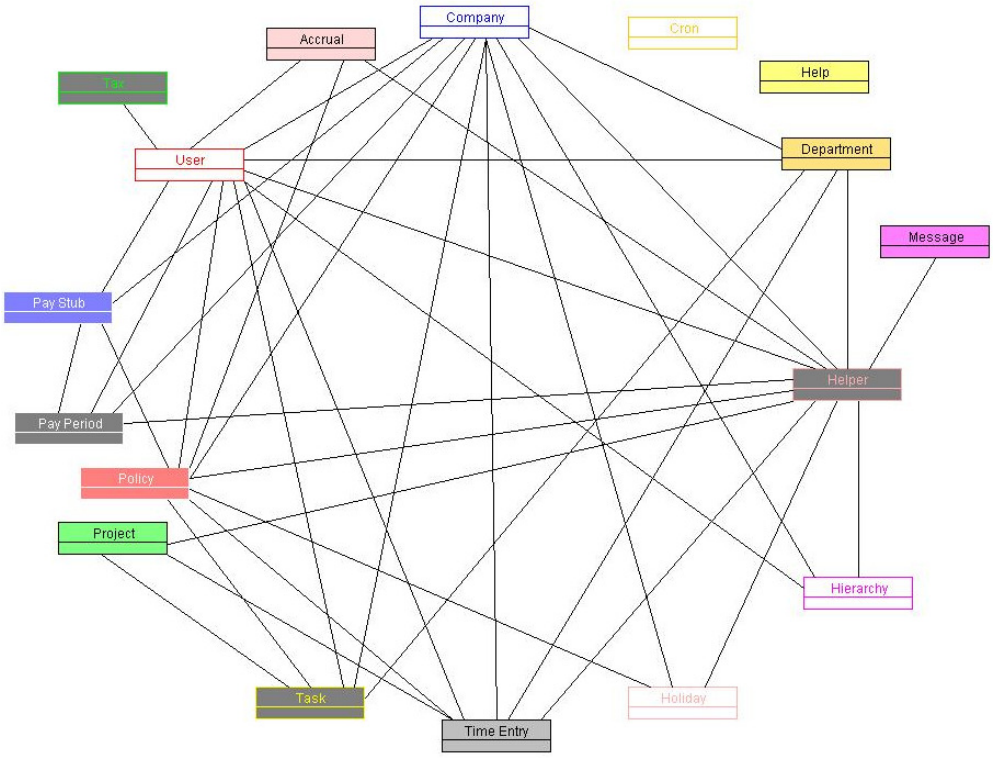

Figure 14. TimeTrex cluster diagram modeled from the ERD

We tried to be as granular as possible in our groupings, to allow for more flexible analysis later. The UMLet tool we used to create the class diagrams and highlight

the clusters has a limited number of colours available, so we tried to keep to those allowing for best readability.

Once the clusters were identified, our next step was to try reducing the diagram to a smaller subset of classes that would allow us to generate a simple time and activity management system from the same objects used in TimeTrex.

To reduce the diagram to a set of essential clusters, we started with removing the entire clusters that did not provide necessary functionality. **Accrual**, **Pay Period**, **Pay Stub**, and **Tax** clusters are all related to payroll; **Policy**, and **Holiday** clusters deal with lookup and policy data. **Cron**, **Hierarchy**, and **Help** clusters are used for UI generation and housekeeping scripts. **Message** cluster relate to messages sent between users. All these clusters could be removed without affecting the core functionality of a time and activity management system.

The **Department** cluster classes were moved into the Company cluster, as they relate to organizational units within the company. Most of the **Helper** cluster classes were removed. The only two that remained were UserDate and UserDateTotal as they are involved in time entry. Both of these classes were therefore moved into the **Time Entry** cluster.

There are 5 clusters remaining. The **Project** cluster contains Job and JobItem classes, and **Task** cluster contains the Schedule class. All three classes are involved in managing work items against which time can be logged.

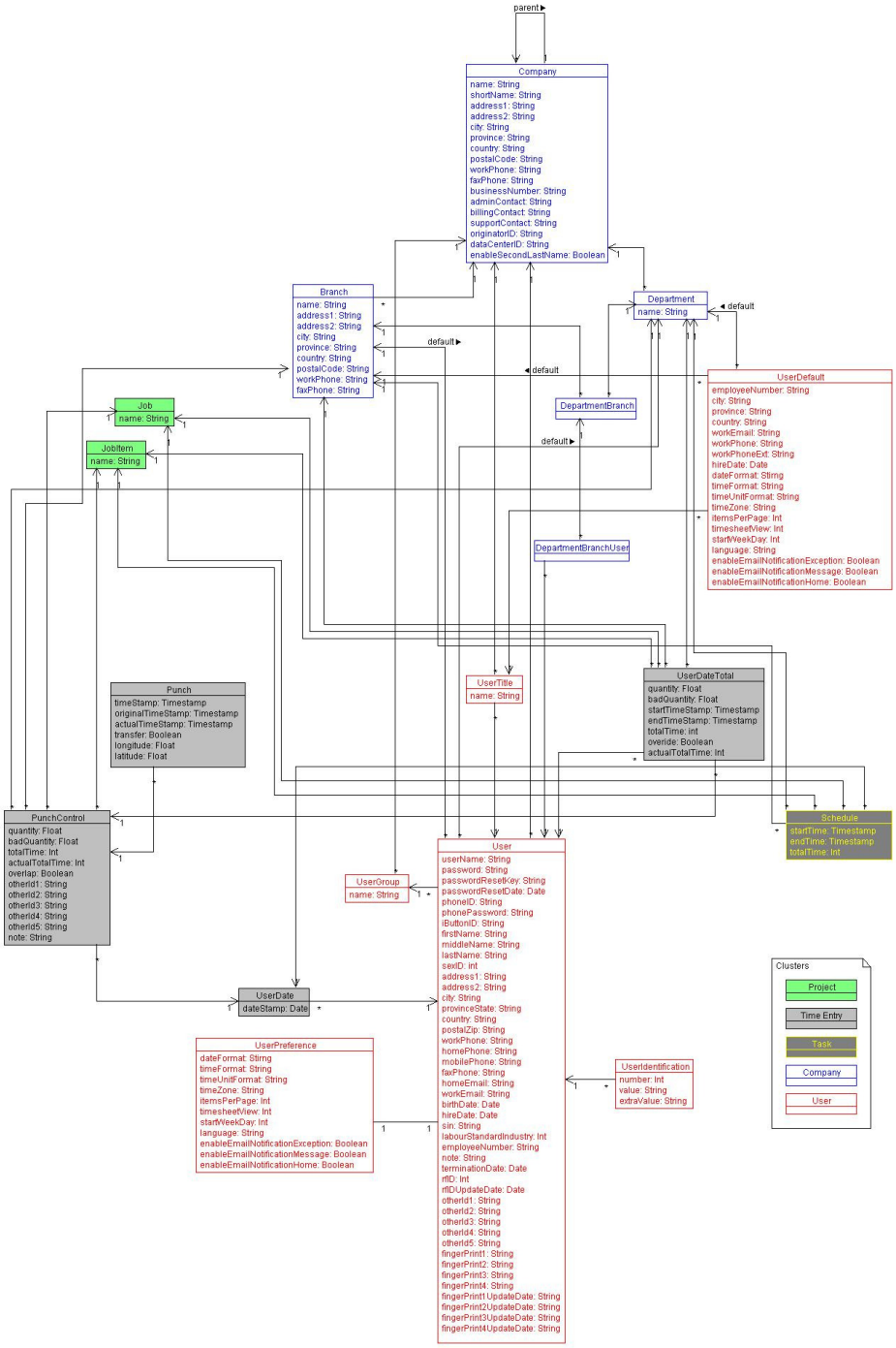

Figure 15. TimeTrex reduced UML class diagram modeled from the ERD with highlighted clusters, showing only the clusters related to TAM

Time logging happens against the job and the job item, whereas the schedule separates work on the same job into separate time periods. This is why we have separated the classes in such a manner, although other interpretations are possible. The reduced cluster class diagram is shown in **Figure 15**.

We have kept Company and Branch classes in the **Company** cluster, and have included with them the classes from the **Department** cluster. We have dropped all the other company-related classes as they are either involved in housekeeping or in payroll-related functionality.

In the **User** cluster, we have kept the classes that deal with user contact data, identification, preferences, default settings and grouping, and the user role within the company. The rest of the classes have been dropped, as they deal with financial information and additional user data not essential for time and activity management applications.

We retained the entire **Time Entry** cluster consisting of Punch and PunchControl classes, and included with it the UserDate and UserDateTotal from the **Helper** cluster, as discussed above.

After the reduction, we ended up with a diagram containing 18 classes grouped into 5 clusters: **Project**, **User**, **Company**, **Time Entry**, and **Task**.

Based on the reduced class diagram for TimeTrex, we have created the corresponding Umple code, and from it have generated Java and PHP classes for a reduced time and activity management application. The code is available at (Levin, 2009).

CHAPTER 4

PRODUCT LINE DERIVATION

4.1. Product Line Derivation Notation and Methodology

We have looked at four time and activity management applications – one small one, two medium ones, and one large one. For each application, we were able to identify clusters with similar functionality, reduce the application set of classes to those required by a time and activity management system, and generate textual UML models (using Umple) and source code (in Java and PHP) for the classes and associations of each system.

It is clear that applications in the time and activity management domain have certain similarities (the clusters we identified as common) and certain variabilities (the different classes and associations within the clusters). Therefore, it should be possible to express time and activity management application architecture as a product line architecture. The main obstacle in this process is the difference in design of these four systems: the same functionality is achieved using different classes, attributes, and associations, making generalization difficult.

One possibility would be to encode variabilities in such a way that we could generate the four systems precisely as they are in their reduced form. This would

be an interesting exercise but the resulting architecture would ultimately be useless, as no other time and activity management system could be generated from such a specification.

The second way would be to try reducing the four systems even further, removing all but absolutely necessary classes and attributes. This still however, would not solve the problem of generalizing the different ways of implementing the same feature.

The third way would be to follow a bottom-up approach by starting with the simplest case of time entry and building the other features upon it. This way we would not be following precisely the naming convention of each of four systems or the exact same attributes represented in each. The resulting generalization, however, should allow us to generate a variety of time and activity management systems based on a selection of required features. Among those, we should be able to generate the four systems closely resembling our case study applications in feature sets.

We have decided to follow the third approach, as it potentially results in the most useful definition of the product line. The resulting model could then be further extended to generate full-fledged time and activity management applications, including accrual calculations, policies, and other modules we have removed during the reduction process.

4.1.1. Product Line Derivation Notation

We shall use the following notation elements in the product line model building below:

1. Application.Class (for instance, Klok.TimeEntry) to refer to a model of Class in Application (class name and attributes);

2. Application.Class1 – Application.Class2 (for instance, Klok.TimeEntry – Klok.Project) to refer to an association of Class1 and Class2 in Application;

3. TAM.Class (for instance, TAM.TimeEntry) to refer to a model of Class in the Time and Activity Management (TAM) product line;

4. TAM.Class1 – TAM.Class2 (for instance, TAM.TimeEntry – TAM.Project) to refer to an association of Class1 and Class2 in the TAM product line;

5. `attribute` (for instance, `duration`) to refer to a class attribute;

6. `o:attribute` – optional attribute (for instance, `o:comment`);

7. a*nm*:`attribute` where *n>0* denotes the alternative attribute index, and *m>0* denotes the option index within the *n*th alternative attribute (for instance, `a11:duration`, `a12:startTime`, `a12:endTime` to

denote that either `duration` or a combination of `startTime` and `endTime` are required).

4.1.2. Product line derivation methodology

1. Select several systems from the domain and model each system in UML

When selecting the systems from which to derive your product line, several things are important to keep in mind. Select systems that differ in size and complexity. Otherwise you might miss essential features that arise only in large systems or arrive to an overly complicated base case if you do not examine smaller systems. It is good to have access to the database schema as it simplifies the modeling. However, if you only go with open source software because you can get access to the database schema, you might overlook features provided by closed-source software, which might be essential to the domain. If you do not have access to a copy of a closed-source application that you would like to analyze, look into a possibility of using a trial version to analyze the user interface and model the application features that way.

People applying our approach, particularly as it involves reverse engineering, should be aware of possible licensing issues and contract violations (prohibitions against reverse engineering). In our research we reverse engineered the systems based on the available code, databases and UI, and did not copy any code from the existing systems so we are not breaking copyright.

If you are modeling the system from the database, each table can become a class and each field an attribute. If you have a database model that supports foreign keys, the associations can be derived from those. If not, look for the references to other table IDs as indications of relationships among tables. A database table reflecting a many-to-many relationship among tables becomes a class in UML model, associated to the classes that correspond to those tables.

If you do not have access to the system's database, you can still try to model its data structures based on its user interface. The resulting model might not reflect the application's data structures exactly, but it will capture the features supported by the system.

2. Identify clusters of functionality

Look for groups of classes that together address a particular set of features (or a module). For instance, in the TAM domain, all the classes with the project-related data would be grouped into a Project cluster, whereas all those dealing with pay period calculations would be grouped into a Pay Period cluster.

Some classes logically belong to more than one cluster. For instance, classes related to user roles on projects relate to both User and Project clusters. Those can be put into either cluster. In step (**7**) when you are building the model, you will include these classes with that of the two clusters that you model last. For instance, if you choose to model the User cluster first, you will include project

role classes and associations with the Project cluster. These classes will require variation points and variants that model classes to which they are associated from both Project and User clusters.

You can create a variability model without grouping classes into clusters. However, it will make identifying related classes more difficult as you try to determine which features are essential to your domain and which are peripheral.

The clusters can be quite big if there are many related classes. If that is the case, see if the cluster you have identified is really addressing only one feature set that cannot be broken down further. For example, in the TAM domain, classes related to project reviews are as much project-related as the project milestone class. However, time entries can be made against milestones, so project milestone class is relevant to the time entry systems. Project reviews have more to do with project management and thus are from a neighbouring domain. Therefore they can either be separated into their own Project Review cluster and be removed in step (**3**) or stay in the Project cluster and be removed in step (**4**).

3. Remove clusters that are not directly related to the domain, together with the associations that connect them to the remaining classes.

Look for clusters that deal with additional features not essential to your domain. These could be helper classes that are system- or platform- dependent and will not be needed by all systems, or those that belong in the display layer. They can

also be features from the domains closely related to yours. For example, the Accrual cluster in TimeTrex is related to accounting and is only relevant to time entry if a company that uses the tool for time entry needs to tie the time entries into their accrual policies. This cluster is thus not essential for the TAM domain and can be removed.

Keep in mind that these clusters can later be added to the system if desired after the essential features have been modeled. Removing them here allows you to concentrate the modeling effort on those clusters that are essential to the system, which can be hard with multiple associations and classes cluttering the model.

4. From the remaining clusters remove classes that are not directly related to the domain, together with the associations that connect them to the remaining classes.

The reasoning and the process are similar to step (**3**). Removing the unnecessary classes here allows us to further simplify the model. If desired, they can later be added to the product line.

5. Identify a base case for the domain.

Think of the smallest set of features that allows one to accomplish the main task of a system within the domain. For example, for a TAM system, we need to be able to track time spent each day on various activities. So, the minimal amount of

information we need to track is the date, the duration of time spent, and the activity it is spent on (which can be achieved with a comment). Thus our base case becomes a class with a date, duration, and comment.

The base case does not have to be present in each application. For example, some systems will track all time entries against a project. Later, when we add a project-related concern to the product family, comment will become part of one of the variants within an alternative variation point, the other one being the time logged against a project.

It might be possible that a domain does not have a single base case, as the task can be accomplished by more than one set of features. In that case, identify each set of features, so that you have multiple "base cases".

6. Model the base case in VML4Umple

If there is functionality that is required for all systems in the domain (a commonality), include an invoke statement for the corresponding features in the product family. For example, in the TAM domain, each time entry in the system must include a date. Thus, invocation of this field becomes part of the product line model.

In the case of more than one base case, we need to create an alternative variation point where each such base case is a variant.

7. Build the product line from the base case up

For each of the applications, go through each cluster in the order of simpler clusters to more complicated clusters with more associations.

1. If a cluster does not exist in product line, create a concern for it. Otherwise, find a corresponding concern.

2. For each class in the cluster: if a class exists with the same or different name that has similar functionality, find a corresponding class, renaming it for clarity if desired.

3. For each attribute in the class, if it is not directly related to the domain (UI-related, device-specific, etc.) skip the attribute, do not include it. Most likely it needs to be a part of a different layer or of additional clusters.

4. For each attribute in the class, if the attribute exists - find a corresponding attribute, renaming it for clarity if desired.

5. For each attribute in the class, if an attribute does not exist with a same or different name that has similar functionality, create a variation point for the class with the attribute. If the attribute is optional, make the variation point *optional*, otherwise make the variation point *alternative* and create two variants, one for each alternative.

6. For each attribute in the class, if the variation point depends on another variant or variation point previously defined, use the "requires" keyword to indicate the dependency

7. For each class, if a class is associated with others, add all the associations to the classes that are already modeled. If a class is associated with another class which has not yet been defined, make a note of the association and when you model the corresponding class, include the association.

You should now have a full product line based on the reduced systems you derived in (**4**). If you now wish to include any of the classes or clusters you discarded in steps (**2**) and (**3**), you can do so by following the process described in (**7**). The difference is that you are now not building on the base case from (**6**), but on all of the features you modeled in (**7**).

A similar process can perhaps be used to connect a VML4Umple product line model for one domain (such as TAM) to a VML4Umple model of a neighbouring domain (such as accounting). Theoretically, the two models can be linked together by associations among related classes, similar to the way packages are linked though interfaces.

To generate a full system from the product line you derived, create a VML invoke file with an invoke statement for every feature you wish to include in the end

system. If you have an existing invoke statement that you have used before to generate a system, you can adjust it to reflect the set of features required for the new system, instead of writing an invoke file from scratch.

4.2. Product Line Derivation

The simplest possible case in time and activity management is entering time in a spreadsheet-like fashion, with minimum information. This would include time entry by a single user filling out the date, time duration, and comment for each time entry. No tasks, projects, company or user information would be present. Thus we shall take this scenario as our base case. All the required information can be captured in a single class that we shall name TimeEntry, with three required fields: `date`, `duration`, and `comment`.

4.2.1. Integrating Klok Functionality into the Product Line

Our simplest case study, Klok, is a step above the base case: it logs time against a project. Thus in addition to our TimeEntry class, we now also have the Project class connected to the TimeEntry by an optional association such that for each TimeEntry instance there is a 0 or 1 project. In the case of the Klok.TimeEntry, `comment` is optional, and `startTime` and `endTime` are used instead of `duration`. So, we write it as `a11:duration, a12:startTime, a12:endTime`.

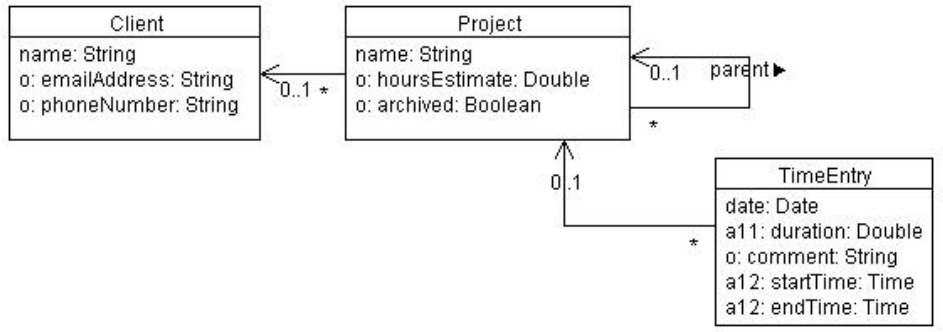

Figure 16. Partial product line with base case and Klok functionality – class diagram

From Klok.Project, we add the `name`, `o:hoursEstimate`, and `o:archived`. We leave out the attribute `colourCode`, as it is UI-related and not essential for the model. For the remaining contact fields, we shall add another optional class Client that will keep track of client-related data. Even though Klok does not have a Client class, Anuko and Leia do, therefore we choose to isolate the client-related data in a separate class. Each project can have 0 or 1 client associated with it. We also keep the parent association between projects, as it is present in multiple time and activity management systems.

The resulting small product line model is shown in **Figure 16** and the corresponding VML4Umple code – in **Figure 17**.

At this point, we have a small product line with a few optional and alternative attributes, and optional classes. We can generate a time management system for the base case as well as one with the functionality of Klok.

```
Concern CRequired{
  // base case - date is required for time entry
  VariationPoint VPTimeEntryDate{
    Kind: Optional;
    class TimeEntry{
      Date date;
    }
  }
}
invoke(CRequired, VPTimeEntryDate);
// either duration or both start and end time are required
Concern CTimeEntry{
  VariationPoint VPEntryDuration{
    Kind: Alternative;
    Variant VDuration{
      class TimeEntry{
        Double duration;
      }
    }
    Variant VStartEndTime{
      class TimeEntry{
        Time startTime;
        Time endTime;
      }
    }
  }
}
Concern CTimeEntryAgainstProject{
  // time can be entered against project or via a comment for each time entry
  VariationPoint VPEntryAgainstProject{
    Kind: Alternative;
    Variant VComment{
      class TimeEntry{
        String comment;
      }
    }
    Variant VProject{
      class Project{
        String name;
      }
      association {
        0..1 Project <- * TimeEntry;
      }
      association {
        0..1 Project parent  <- * Project;
      }
    }
  }
  // project can optionally include estimate expressed in hours
  VariationPoint VPEstimate requires VPEntryAgainstProject(VProject){
    Kind: Optional;
    class Project{
      Double hoursEstimate;
    }
  }
  // project can optionally be set as archived
  VariationPoint VPArchive requires VPEntryAgainstProject(VProject){
    Kind: Optional;
    class Project{
      Boolean archived;
    }
  }
  // client information for a particular project can optionally be stored
  VariationPoint VPClient requires VPEntryAgainstProject(VProject){
```

```
    Kind: Optional;
    class Client{
      String name;
    }
    association {
      0..1 Client <- * Project;
    }
  }
  // client can optionally have an email address
  VariationPoint VPClientEmail requires VPClient{
    Kind: Optional;
    class Client{
      String emailAddress;
    }
  }
  // client can optionally have a phone number
  VariationPoint VPClientPhoneNumber requires VPClient{
    Kind: Optional;
    class Client{
      String phoneNumber;
    }
  }
}
```

Figure 17. Partial product line with base case and Klok functionality – VML4Umple

4.2.2. Integrating Leia Functionality into the Product Line

Now that we have a product line that incorporates the base case and the smallest time entry system, we can extend it to include the functionality of a medium-size application. Both Leia and Anuko are fit candidates. We picked Leia as we would like to apply the terminology used in its model for Activities and Tasks. We will then adopt the same terminology when incorporating Anuko functionality into the product line.

We start from the product line derived in 4.2.1 and proceed by examining each class, attribute, and association from the Leia model for inclusion into the product line model.

Leia.WorkItemUserTime serves the same purpose as the TAM.TimeEntry. Date and comment fields are already present in TAM.TimeEntry. The `durationMinutes` attribute in Leia corresponds to the `duration` attribute in the product line. We need to add two optional fields to TAM.TimeEntry: `exported` and `rejectedComment`. These are used in timesheet submission and approval process. Leia.TimeType and Leia.WorkItemUserTimeStatus represent lookup tables. Both of these together with their associations to the TimeEntry class will be added to the product line model as optional classes. We will omit the `orderNum` attribute of Leia.TimeType as it is used for ordering items during display, and is not directly relevant to the data model. We shall rename the WorkItemUserTimeStatus class to TimeEntryStatus for consistency. As in Leia the time entry type and status are required fields, both associations are 1 – *. To make the product line more flexible, however, we shall convert the cardinality of both to 0..1 – *. The required fields can then be enforced in the business logic layer if needed.

Another cluster which is already represented in the product line is the Project cluster. Thus we will look at the Leia model of the Project cluster next. TAM.Client already has a `name` attribute. We will add an optional `slaLevel` attribute to it, which is used to keep track of a client's level under the Service Level Agreement (SLA). All the attributes of Leia.Project will be added to TAM.Project as optional attributes with the exception of `name`, `estimate`, and

`archived` (as their equivalents already exist); `preNews` and `postNews` (as those are preferences for whether to notify selected people of the project beginning and end respectively, and are better managed through user email preferences); and `devEndDate` and `normEndDate` (that stand for end of development and end of normalization phases respectively). The last two attributes are dependent on the internal process of a company that manages the projects, and thus will only be relevant in a small subset of cases. Instead, we shall add an `endDate` attribute that would signify the completion due date of a project, to provide a process-independent counterpart to the `startDate` attribute. We will also change the type of the `hoursEstimate` attribute of TAM.Project to Double to accommodate the way it is used in Leia, and will rename it to `timeEstimate` for flexibility.

Leia.Project is associated with three lookup classes: ProjectType, ProjectStatus, and ProjectIntensity. In each of these we shall ignore the display-related `orderNum` attribute. The cardinality of the associations is changed to 0..1 – * for the most flexible product line definition.

Projects in Leia can be associated with several milestones. This is a common functionality in project management applications, so we shall keep the ProjectMilestone class with attributes `name` and `releaseDate`. The attribute `lastNotification` is relevant to email preferences and is better managed elsewhere in the application, so we will omit it.

The remaining class in Leia that we associated with the Project cluster is ProjectUserRole. Since it requires the User cluster with both User and Role classes present to be useful, we shall include it in variants related to the User cluster, which are discussed later.

So far we have covered the TimeEntry and Project clusters present in both Klok and Leia. The next cluster that appears in Leia is the Company cluster, containing entities relevant to the internal company structure. In the case of Leia, the only class in this cluster is Activity. As we noted before, we shall keep to the terminology used in Leia for defining what constitutes a Task, Project, and Activity. A Task is the smallest unit of work against which users can log time. A Project is a larger unit of work done for a particular client. In cases where a single user tracks their time spent working on personal projects, the user themselves can be thought of as the client. A typical software project can have multiple milestones. An Activity is a larger unit of support type of work. This includes office and infrastructure work, project and team management, human relations management, vacation and time off, company functions, and so on. Both Project and Activity can be divided into multiple Tasks. For example, there can be a "Deployment" task for a software project, or a "Recruitment" task for the "Human Relations Management" activity.

Leia.Activity, Leia.Task, and Leia.Project all extend Leia.WorkItem. This is a design choice made by the developers. We will not preserve it, to keep the classes

separate, so as not to create additional overhead if only one of them is used. Thus the TAM.Activity class will have the following attributes: `name`, `description`, and optional attribute `archived`. We shall omit the `contactInfo` and `percentageCompleted` attributes, as they are deprecated in the newer versions of Leia due to lack of use. Just like time can be logged against a project, it can also be logged against an activity, so we will create the 0..1 Activity ← * TimeEntry association.

Leia Task cluster contains Leia.WorkItem, Leia.Task, and Leia.TaskUser. The latter is used to keep track of assignments of tasks to users and requires the User cluster. Thus we shall include it in variants related to the User cluster, which are discussed later.

Following the discussion regarding the Activity class, we shall omit the WorkItem class from the Task cluster for similar reasons. The TAM.Task class will include Leia.WorkItem attributes `name` and `description`, as well as optional attributes `priority`, `dueDate`, and `estimatedTime` (as a more generic version of the `estimatedMinutes` attribute).

TAM.Task can be associated with TAM.Activity, TAM.ProjectMilestone, or directly with TAM.Project. Also, time can be entered against a task. Thus we have 4 classes potentially associated with TAM.Task.

The last remaining cluster to add to the product line is the User cluster. It is optional, as systems can be either multi-user or single-user. We shall keep the following attributes from Leia.User: `firstName`, `lastName`, `username`, `password`, `isActive`. The attributes `email`, `extension` (renamed to `phoneExtension`), `hostName`, and `lastLogin` will be added as optional. We shall omit the attributes `receiveTaskResolvedEmail` and `paginationPreference`, as they are better related to user preferences, and not directly to the user. We will also omit `quickbooksName`, as it is closely tied to a particular payroll implementation.

TAM.User is connected to all the other clusters. The TimeEntry cluster is connected by an association between TAM.User and TAM.TimeEntry (a time entry is logged by a particular user). The Task cluster is connected through TAM.TaskUser (a task can be assigned to a particular user). The Activity cluster is connected by an association between TAM.User and TAM.Activity (in Leia, each activity has assigned to it a user that approves the time entries against that activity – we will call that person that activity's manager). The Project cluster is connected through the ProjectUserRole class to both TAM.User and TAM.Role (a user can have one or more roles in a project). For the latter, we have to include the Role, RoleGroupItem, and RoleGroup classes. Even in a system where each project has only one role, a user can be assigned to the project by being assigned to that role. For TAM.Role we will omit `note`, `sequence`, and `hidden` attributes (as

they are related to the UI), and keep the attributes `name`, `description`, and `multi` (whether more than one person can be assigned to a role), which we will rename to `multipleUsersAllowed`.

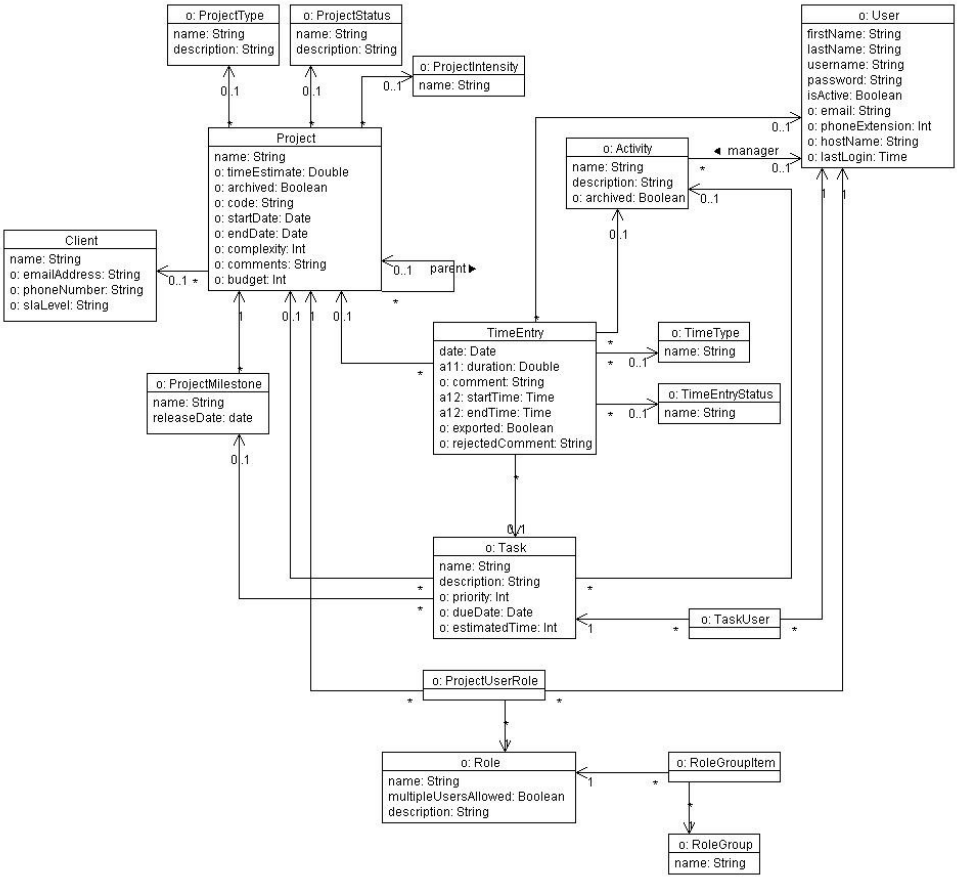

Figure 18. Partial product line with base case, Klok, and Leia functionality – class diagram

The remaining two classes in the Leia model are UserManager and Manager. Manager class designates a user as a manager of a particular team, whereas UserManager maps users to their managers. This arrangement is counter-

intuitive. It would make more sense to have a Department class that keeps track of specific department information including the department's manager user, and a class mapping users to a particular department. TimeTrex has a concept of a Department that appears to be applicable. Thus we shall keep this functionality out of the product line until we examine TimeTrex to see if there is a way to define departments and managers in a more scalable and flexible way.

This concludes the integration of Leia classes into the TAM product line. So far we should be able to produce systems that closely resemble the base case, Klok, and Leia. The resulting model is shown on **Figure 18**.

4.2.3. Integrating Anuko Time Tracker Functionality into the Product Line

To integrate the Anuko Time Tracker classes into the product line, we start with the ActivityLog class. It serves the same purpose as the TAM.TimeEntry class. We shall add the following attributes as optional: `timestamp`, `proof`, `charge`, and `billable`. As the last three appear to be flags, we will change their type from Int to Boolean. Anuko.ActivityLog is connected to the Project, Activity, and User clusters. Since these connections already exist in our product line model, there is no need to add them.

The Anuko.Project class has the `name` and `timestamp` attributes. We shall add the latter to TAM.Project as an optional attribute, renamed to `lastModified` for clarity. Anuko.ProjectStatus contains the `hidden` attribute that appears to be used for display purposes. Therefore we shall omit it. The `name` attribute is already represented in TAM.ProjectStatus. Anuko.UserBind keeps track of assignment of a user to a project. In the product line model, this role is played by TAM.ProjectUserRole. We shall add the optional attribute `rate` used in Anuko.UserBind to TAM.ProjectUserRole.

Task cluster in the Anuko Time Tracker is represented by three classes: Activity, ActivityBind, and ActivityStatus. Anuko.Activity is similar to TAM.Task (we shall add the optional attribute `timestamp` to TAM.Task, renamed to `lastModified` for clarify). Anuko.ActivityBind functionality is presented in TAM.TaskUser (we shall add the optional attribute `rate` to TAM.TaskUser). We shall add TAM.TaskStatus and include the Anuko.ActivityStatus `name` attribute, omitting the `hidden` attribute as we did for TAM.ProjectStatus before. In Anuko Time Tracker, an Activity can have a manager, so we shall include a corresponding optional association between TAM.Task and TAM.User.

The Company cluster in Anuko Time Tracker includes two classes: Client and Company. Client appears to be a look-up class not connected to other classes by associations. In our product line, client is connected to a project and thus belongs to the Project cluster. We shall keep that as a reasonable assumption: after all, if a

client is being invoiced, some work has been performed for them, which means a corresponding project must exist. TAM.Client already keeps track of the client's name. The rest of the attributes from Anuko.Client will be added to TAM.Client as optional fields, with the exception of `fSubtotals` (as it appears to be display-related) and `addrYour` (as it appears to stand for the company's address and is better represented in the Company class). We will rename `addrCust` to `address` for clarity, and will use it to represent the client's address.

In Anuko Time Tracker users belong to a company. Leia was designed to be used by a single company, so our model up to now associated users with an implicit company. However, even in case of a single company it might be useful to keep company-centric information in the system, such as company name for branding, address for invoice generation, or company's web site URL for reference. Thus we shall include the Company class into our product line connecting it to the TAM.User class by an optional association. We will keep the `name` attribute, `www` (renamed to `url`), `address` (formerly `addrYour` from Anuko.Client), `currency`, and `locktime`. All the attributes except for the `name` will be optional.

The Anuko TimeTraker User cluster contains User and UserStatus classes. We will add the UserStatus class to the product line omitting the `hidden` attribute as we did before for the ProjectStatus and TaskStatus. From Anuko.User we will keep `level`, `rate`, `language`, and `timestamp` (renamed to

`lastModified`). The attributes `showPie` and `pieMode` appear to be display-related, so they are omitted. The attribute `language` will be used to mean the primary language used to interact with a user in the workplace, as opposed to the user interface language preference that will be addressed when we look at user preference representation in TimeTrex.

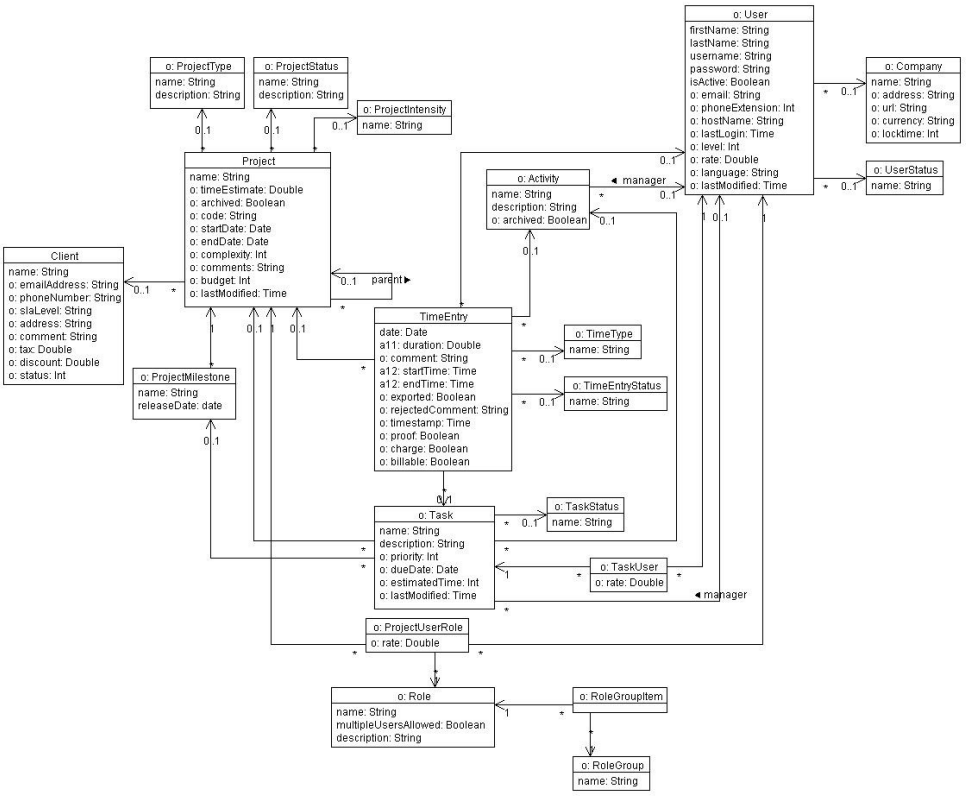

Figure 19. Partial product line with base case, Klok, Leia, and Anuko Time Tracker functionality – class diagram

The attribute `comanager` is related to the management hierarchy within the company. We shall deter its inclusion until we examine the TimeTrex company

structure model, as it is the most complex one. For the same reason we shall omit the manager reflexive association on the User class.

At this point, we have specified a product line model, from which systems closely resembling the base case, Klok, Leia, and Anuko Time Tracker can be generated. **Figure 19** shows the corresponding model of the product line.

4.2.4. Integrating TimeTrex Functionality into the Product Line

Due to the large number of classes in the original TimeTrex system and the large number of associations among them, in the reduced TimeTrex system almost every class has a large number of associations to other classes. In order to more effectively generalize the TAM product line model, we shall drop many of these associations and re-connect the classes in a more logical manner, allowing for a more scalable design.

The Project cluster in TimeTrex incorporates two classes: Job and JobItem. Each time entry is logged against both the Job and the JobItem, and there is no apparent relationship between these two classes, so it seems logical to combine them into one class, against which the time is logged. The only field of this class is `name`, and so the logical choice for its equivalent in the TAM product line model would be TAM.Project. Since we already have associations between

TAM.TimeEntry and TAM.Project, there is no need to create an additional class or extra associations in the product line model.

The Task cluster in TimeTrex is represented by one class – TimeTrex.Schedule, with attributes `startTime`, `endTime`, and `totalTime`. TimeTrex connects this class with TimeTrex.Job class in a way similar to how TAM.Task is connected to TAM.Project. We shall add `startTime` and `endTime` as optional attributes to the TAM.Task class to indicate the planned start and end time of a task. The `totalTime` attribute is equivalent to the `estimatedTime` attribute in TAM.Task, so we shall omit it. TimeTrex.Task is connected to the User module via the TimeTrex.UserDate class, which is equivalent to the TAM.Task – TAM.TaskUser association. TimeTrex.Task is also connected to TimeTrex.Branch and TimeTrex.Department classes from the Company cluster. We shall return to these associations later when we are examining the Company cluster of the TimeTrex system.

The time entry in TimeTrex is performed with "punch in" / "punch out" actions. An employee uses a particular device to indicate the time they start and end a particular job, thus creating time entry records of an equivalent duration on a given date against a particular job. In addition to entering the time by hand, a wide variety of hardware "punch" devices is supported: fingerprint readers, phones, cell phones, proximity card readers, etc. The TimeTrex.PunchControl class appears to deal with the hardware sampling of the "punches" entered

(stored by TimeTrex.Punch) and determining quantity of "good" (legitimate) entries in the total number of entries. To abstract from the implementation details, we shall only use the TimeTrex.Punch class and assume that a provided punch is legitimate. The controller then can be programmed in a way suiting a particular application, without enforcing a specific data model. TimeTrex.Punch attributes `originalTimestamp`, `actualTimeStamp`, `transfer` (as optional), `longitude`, and `latitude` will be included in TAM.Punch. We shall also add the Boolean attribute `direction`, to distinguish between "in" and "out" punches. TAM.TimeEntry will be connected with the TAM.Punch class by an optional association, where each time entry can be associated with multiple punches. The punch timestamp fields would tell the story of when the work on the task started and ended.

Two other classes within the TimeTrex TimeEntry cluster are UserDate and UserDateTotal, both of which are connected to TimeTrex.PunchControl. TAM.TimeEntry already includes the time entry date, start, end, and total time. Attributes `quantity`, `badQuantity`, and `actualTotalTime` in the TimeTrex.UserDateTotal class seem to depend on the punch controller implementation, so we shall omit them. The `override` flag shall be added to the Punch class as an optional attribute, to mark whether the punched-in time entry has been overridden.

The Company cluster is quite extensive in the TimeTrex system. A company can contain multiple departments and branches. We shall add the following classes to the TAM product line: Department (including the `name` attribute), Branch (`name`, `address`, `city`, `province`, `country`, `postalCode`, `phoneNumber`, and `faxNumber`), DepartmentBranch class to represent the many-to-many association between the two, and DepartmentBranchUser to represent the ability of an employee to belong to several departments in different branches, allowing for organizational flexibility. Additional fields from TimeTrex.Company shall be added to TAM.Company as optional attributes: `shortName`, `city`, `province`, `country`, `postalCode`, `phoneNumber`, `faxNumber`, `businessNumber`, `adminContact`, `billingContact`, `supportContact`. We shall omit `originatorID`, `dataCenterID`, and `enableSecondLastName`, as the first two appear to be related to business logic, and the last one – to internationalization.

A parent reflexive association shall be added to TAM.Company. Associations between TAM.Company and TAM.Department as well as TAM.Company and TAM.Branch shall also be added. TAM.User will optionally be associated with a company, or with a branch within a company, or with a department within a company, or with a combination of a branch and department. A department can potentially have sub-divisions, so we shall add a parent reflexive association to TAM.Department. Each department can be associated with one user in the

manager role. This will allow us to model the organizational structure of companies with multiple branches, each containing a hierarchy of departments, each of which has a manager (with one person being able to perform manager roles for multiple departments). For example, if a company has two branches in two different cities, and each branch has a Development and a Business department, where a Development department contains several development teams, each team might have a separate team leader (manager) and each department can have a manager as well. A wide variety of organizational structures can thus be accommodated.

In the TAM product line, a user can be assigned to any task, project, or activity, and can log time against any project, task, or activity. Any restrictions to this arrangement based on the organizational structure can be enforced by business logic. This approach allows us to avoid a complicated network of optional and alternative dependencies between classes at the data model level.

The last remaining cluster in TimeTrex is the User cluster. It consists of several classes: User, UserTitle (within a company), UserGroup (within a company), UserIdentification, UserPreference, and UserDefault. We shall omit UserDefault, as it can potentially be associated with Department, Branch, Company, DepartmentBranchUser, or UserTitle, depending on the company structure. It primarily has to do with the initial display settings and newly-created users, contributing to usability, so we shall omit it in the data model. We shall keep the

UserGroup class as it allows for additional grouping of users within a company (albeit a user can only belong to one user group), and UserTitle class. We already have a role-based system for user roles within the projects, but that does not include user roles within the company. As TAM.User already exists, we need only to add more optional attributes to it from TimeTrex.User: `phoneID`, `phonePassword`, `middleName`, `sex` (instead of `sexID` as that would require a look-up table), `address`, `city`, `province`, `country`, `postalCode`, `workPhoneNumber`, `homePhoneNumber`, `mobilePhoneNumber`, `faxNumber`, `homeEmail`, `workEmail` (we'll rename `email` for clarity), `birthDate`, `hireDate`, `socialInsuranceNumber` (instead of `sin`), `employeeNumber`, `note`, and `terminationDate`.

The following attributes shall be omitted. The attributes `passwordResetKey` and `passwordResetDate` are dependent on the authentication system; `iButtonID` only applies when iButton device is used to punch in time; `labourStandardIndustry` is related to labour policies; `rfID` and `rfIDUpdateDate` only apply when RFID is used to punch in time; `otherId` attributes are placeholder fields; and fingerprint-related attributes are only relevant when fingerprint readers are used.

We shall omit the UserIdentification table as it appears to be implementation-dependent. We already have a way to keep track of the last time a user logged into the system through the TAM.User lastLogin attribute.

We will keep the UserPreference table while omitting the following fields: `enableEmailNotificationException`, `enableEmailNotificationMessage`, `enableEmailNotificationHome`, and `timesheetView`, as they are application-dependent. The other preferences have to do with date and time format, time zone, user interface language, and number of items displayed per page, which are relevant in a large variety of applications.

The resulting product line model is displayed on **Figure 20**. The corresponding VML4Umple code is not included here as it is rather long. The complete product line VML4Umple model contains annotations for each feature, listing from which of the case study systems it originated. It is available at (Levin, 2009).

4.3. Time and Activity Management Product Line Model

To be able to work with the product line expressed in VML4Umple, a parser had to be created to process the VML4Umple code for the product line together with the invoke statements and output the Umple code for the resulting system. A. Forward has extended the current Umple compiler implementation to allow for this functionality. He created a command-line tool that given the product line VML4Umple file, the invocation file for a desired system, and the name for the resulting Umple file, generates the Umple code for the application with specified features.

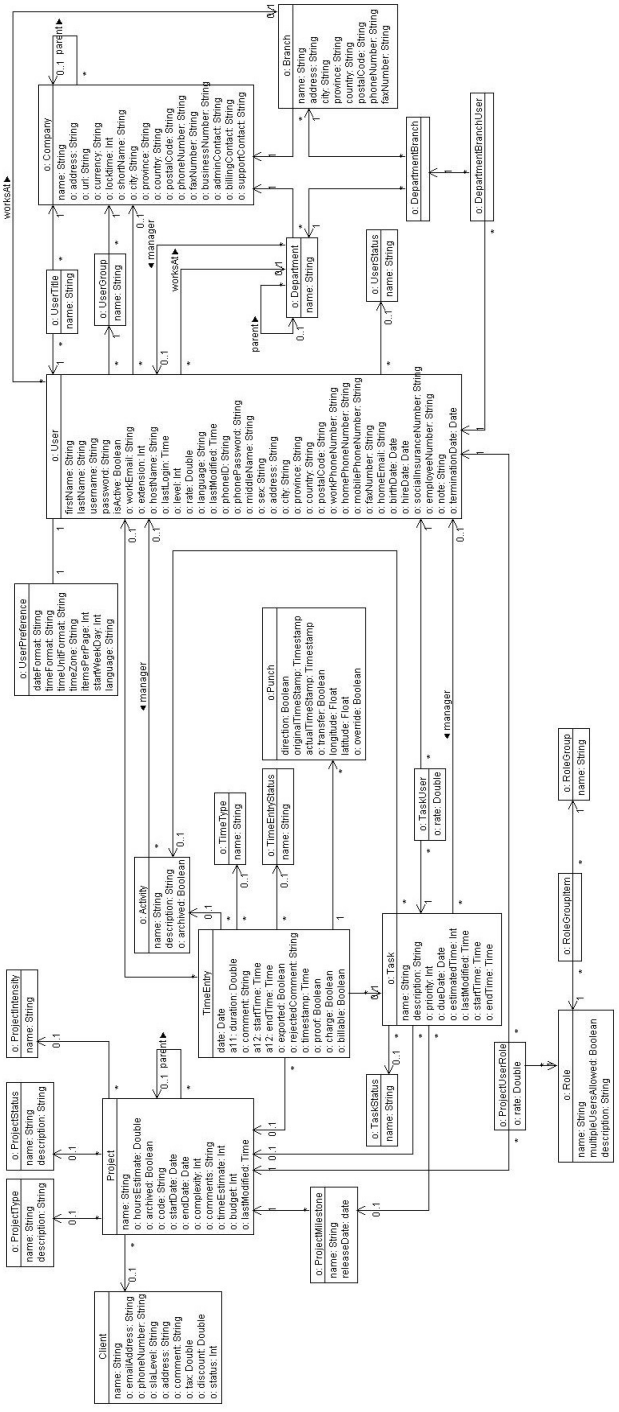

Figure 20. Complete product line with base case, Klok, Leia, Anuko Time Tracker, and TimeTrex functionality – class diagram

The Umple code thus created can then be piped into the Umple compiler to generate the Java or PHP implementation of the particular system. An online implementation also exists for demonstration purposes – by pasting the product line code followed by the invocation code into the text field at (Forward, 2009b), the corresponding Umple, Java, or PHP code can be generated. Further in this work an example of a system's invocation is provided.

Due to VML not explicitly addressing the expression of commonalities, we had to make an adjustment to the product line family VML4Umple code, to be able to specify the single mandatory feature for all of the time and activity management systems. All the TAM systems must have a date attribute in the TimeEntry class. VML does not allow for "mandatory" variation points, and so we specified this variation point as "optional", immediately followed it by its invocation within the product line specification itself, instead of including it in all the invocation files. See **Figure 21** for the corresponding VML4Umple code.

```
Concern CRequired{
  VariationPoint VPTimeEntryDate{
    Kind: Optional;
    class TimeEntry{
      Date date;
    }
  }
}
invoke(CRequired, VPTimeEntryDate);
```

Figure 21. Mandatory feature specification in the VML4Umple model.

Once the product line is expressed in VML4Umple, each system can be generated by the following steps:

1) Create an invocation file by writing a series of invoke statements to build the system's features. See **Figure 22** for the invocation of a base case system (time can be logged against a particular date, with a particular duration and a comment to note what the time was spent on). See **Figure 23** for the invocation of a system that has features similar to those of the Klok system. The other case study systems require more extensive invocation and so the code is too long to be included here. It can be viewed in the case study materials listed online (Levin, 2009).

```
// Base case invocation in VML
// Time Entry
// log time duration
invoke(CTimeEntry, VPEntryDuration, VDuration);
// log time entry comments
invoke(CTimeEntryAgainstProject, VPEntryAgainstProject, VComment);
```

```
// Umple code for the base case system

class TimeEntry{
  Date date;
}
class TimeEntry{
  Double duration;
}
class TimeEntry{
  String comment;
}
```

Figure 22. Invocation of the base case system and the resulting Umple code.

2) Using the VML parser (Forward, 2009b), compile the product line VML

model together with the invocation file created in (1). The parser outputs a series of Umple statements.

3) Run the Umple code output by the parser through the Umple compiler, which generates the code for the desired system (at this time, in either Java or PHP).

```
// Invocation of a system similar to Klok

// Time Entry
// log time start and end times
invoke(CTimeEntry, VPEntryDuration, VStartEndTime);

// Project
// enter time against projects (client-related work items)
invoke(CTimeEntryAgainstProject, VPEntryAgainstProject, VProject);
// store optional comments for time entries
invoke(CTimeEntryAgainstProject, VPTimeEntryComment);
// allow projects to have parent projects
invoke(CTimeEntryAgainstProject, VPProjectParent);
// store project time estimates
invoke(CTimeEntryAgainstProject, VPProjectEstimate);
// allow archiving projects
invoke(CTimeEntryAgainstProject, VPProjectArchive);
// associate projects with clients
invoke(CTimeEntryAgainstProject, VPClient);
// store client email
invoke(CTimeEntryAgainstProject, VPClientEmail);
// store client phone number
invoke(CTimeEntryAgainstProject, VPClientPhoneNumber);
```

```
// Umple code for the system based on Klok

class TimeEntry{ Date date; }
class TimeEntry{ Time startTime; Time endTime; }
class Project{ String name; }
association { 0..1 Project <- * TimeEntry; }
class TimeEntry{ String comment; }
association { 0..1 Project parent  <- * Project; }
class Project{ Double timeEstimate; }
class Project{ Boolean archived; }
class Client{ String name; }
association { 0..1 Client <- * Project; }
class Client{ String emailAddress; }
class Client{ String phoneNumber; }
```

Figure 23. Invocation of the system based on Klok and the resulting Umple code.

You will notice that the Umple code that is output by the VML parser declares the same classes repeatedly, with different attributes. In a programming language like Java this would not be a proper syntax. However, the capabilities for merging properties in the Umple language allow this syntax. The separate class declarations are accumulated to create a complete specification of a class. So the following syntax examples in (1) and (2) are equivalent for the Umple compiler:

```
class TimeEntry { Date date; } class TimeEntry { Double duration; } (1)
class TimeEntry { Date date; Double duration; } (2)
```

After gradually integrating the features from our four case studies into the model, we have arrived at a time and activity management product line that allows us to generate applications closely resembling all four of our case studies, the base case application, and a large variety of "in-between" applications using subsets of the features we have modeled. This includes single- and multi-user applications, systems suited to a variety of organizational structures, project user role definition, and time entry against projects, project milestones, tasks, and activities.

CHAPTER 5

CONTRIBUTIONS, DISCUSSION AND FUTURE WORK

5.1. Contributions

The contributions of this work are as follows:

C1. Case study in the time and activity management domain.

We have modeled four systems within the time and activity management domain. We have analyzed the commonalities and variabilities among the systems, and identified similar clusters of functionality.

C2. Time and activity management product line.

We have generated a usable product line in the TAM domain. We have thoroughly documented the derivation process, including intermediate and final models, generated code, and suggestions on possible future extensions and automation.

C3. Creating a variability model of an entire domain.

We have modeled the commonalities and variabilities in the time and activity management domain in such a way that other applications in this domain can be

added to our variability model, as long as they are built on the `date – duration – comment` base case.

C4. Variability in UML modeling and VML4Umple notation.

We have introduced the notation for UML optional and alternative variability modeling for classes and class attributes. We have also come up with VML4Umple notation, leveraging the hierarchical feature-based VML notation and compact UML-based Umple language, allowing us to define product line features with Umple fragments and generate object-oriented code from the product line model.

C5. One-step generation of code from a variability model.

To generate a code for a system based on a product line model, the only step required is to run an invocation file listing the features desired in the resulting system.

C6. Generic product line derivation methodology.

We have described in detail and presented with the TAM example the methodology to derive a product line for domain with several similar systems.

5.2. Discussion

As previously stated, the majority of businesses need to keep track of the time spent by employees on work tasks. Instead of re-inventing the time tracking applications with similar features, with our TAM product line it is possible to rapidly create an application fitting the needs of a particular business. This would also be useful to individuals keeping track of their own time.

As stated earlier, in this work for simplicity we are modeling only the data structures of the application, but our methodology overall is not intrinsically constrained. This is due to the ability of VML*4*Umple to handle arbitrary Java code, allowing for business logic specification. Further improvements to the Umple language will allow more formal business logic modeling using constraints and state machines.

Our target audience (software developers and maintainers) can benefit from our research in several ways. Our time and activity management line product line derivation could be used by developers as an example of the steps needed to derive a product line. The generic product line derivation methodology we have described can be used in other domains. Once a product line is derived, product development can be done via product line model modification and code generation. New products can be created by invoking a required combination of product line features from the product line. Maintenance can also be done

through modifying the model and generating the code to update all affected versions. This model-driven development reduces the amount of boilerplate code that has to be written by hand, reduces the number of defects (as a large portion of code is generated), and helps ensure the synchronization between the software documentation and implementation.

Our approach can be used to either create a product line from scratch, or to derive one from the existing products. In the former case, a base case has to be identified by analysis of the requirements and taking into account possible variations on the functionality. In the latter case, by analyzing existing products, a common base case can be identified among the existing systems. Starting with the base case the remaining variability model can then be built.

5.2.1. Product Line Derivation Methodology Analysis

Our methodology can be used in domains other than time and activity management, allowing for the creation of other product lines. We are considering here form-based applications, such as point of sale systems, shopping carts, product catalogues, conference and university registration systems, and so on. Mature domains with multiple systems implementing similar functionality can benefit from automated application creation based on required feature sets.

Initial derivation of the product line for any of these types of systems can be done in a manner similar to that used in this work. First, several existing systems

would have to be analyzed – the more complex the domain the more case studies would be required to be able to effectively generalize the domain. Secondly, these systems would need to be iteratively brought to the common base. Thirdly, by analyzing each system separately, a product line can be built that encompasses features from all the case study applications. Once the product line is complete, any of the case study systems, as well as systems with any other possible combination of features which have been modeled, can be generated from the product line with the help of an invocation file.

Other approaches to creating a product line might be possible. One might decide to pick a particular application design and attempt to generalize it to a product line. However, this approach might not suit other (perhaps more often used) designs that address a different set of features. For instance, one might decide that time will always be logged against a project, and build a product line where that is the base case. This will prevent generation of the systems that log time against smaller units of work, or by simply providing comments for each time entry. Thus we believe that analysis of several applications from the domain is essential. Picking applications different in complexity, as well as a combination of open source and proprietary applications, is also important, as each covers a feature set targeted at a slightly different audience. This helps ensure the generality of the resulting product line.

Through research of existing literature, we have not found any mention of an existing time and activity management product line. It is possible that another quite different TAM product line would be created if a different set of applications had been analysed. However, to be possible to generate the systems we used in our case studies, this other TAM product line would have to allow for the same base case. Since the only mandatory features of our product line are: having a date for a time entry, having either duration or start and end time for a time entry, and having a comment or an associated project for it, any other set of features can be built on top of our product line. Thus any other TAM product line can be combined with ours to create a more extensive set of features from which time tracking applications can be generated.

This last conclusion is also supported by our observations during the derivation of the product line. After adding features originating from Klok, the corresponding features from Leia did not have to be added (such as logging time against a project). Features from Anuko TimeTracker did not add much to the intermediate version of a product line based on Klok and Leia. TimeTrex analysis contributed mostly to the company structure (departments and branches) and to additional attributes for Company and User classes, but the time tracking functionality did not significantly change. If we were to analyze several additional applications, it appears that each next one would have only a small subset of features to offer for the inclusion in the product line.

5.2.2. Methodology Improvements and Automation

The approach we used in creating the TAM product line is somewhat time-consuming. The greatest amount of time is taken to model the complex applications, such as TimeTrex. It is helpful if the application is developed with a model-driven approach and already has UML class diagrams documenting the underlying data model.

Once the UML class diagrams are created, they have to be analyzed and reduced to a common base. This requires human involvement, since judgment calls need to be made as to the similarities and differences among the systems. This is hard to automate as different systems might name and arrange the structures addressing the same functionality differently.

Once the systems are reduced to a common base, product line construction requires the identification of a base case and required features. The remaining features are manually added to the product line as optional. Perhaps this stage can be automated, if during the previous analysis stage the similar structures in different systems are mapped to each other. For instance, in the TAM product line, Anuko TimeTracker ActivityLog class signified the same functionality as Leia WorkItemUserTime class. Naming them both TimeEntry after realizing that they perform the same function, simplified the generalization stage.

After the product line is created, generation of a particular system requires creation of an invocation file for that system. We have done this manually. However, during the derivation of the product line the features can be annotated with a list of systems from which they came (which we have done with comments) in a way that can be automatically parsed. The parser can then be extended to take those annotations into account if a system to be invoked is one of the original case studies.

5.2.3. Evaluation of the Chosen Technologies

The technologies used during this work proved to be helpful. UML is a good choice for models as they are closely related to the ERD representations of applications' data structures. The UMLet tool we used allowed for rapid creation of UML diagrams, their export to JPG and PDF formats for documentation, as well as their storage in a textual notation that could be processed in an automated way.

The Umple language was helpful in specifying classes, attributes, and associations in a compact notation with subsequent possibility to rapidly generate the corresponding object-oriented code. The property of Umple that allows specification of classes in fragments was extremely useful in the specification of the product line. Otherwise the definition of the product line would have to involve "remove feature" statements as well as "add feature" statements to be

able to define classes with more or fewer attributes. The Umple Online tool (Forward, 2009a) was very useful in trying out quick code examples.

VML proved a good choice as well, containing the number of concepts sufficient to construct a functional product line, yet requiring a parser to deal with only a few keywords. The hierarchical nature of VML in combination with a way to specify alternative and optional features as well as requirements dependencies makes it easy to learn and use. It would be helpful, however, if we had access to more examples of VML use: the examples provided in VML papers and online are partial and not always unambiguous. The VML Online tool written by Andrew Forward was a big help at the product generation stages.

5.2.4. Design of the TAM Product Line

The systems generated from the TAM product line which are based on the original applications used in the case studies, do not support all the features of the original systems in exactly the same way. This is due to two factors. First of all, between the original systems and the "reduced" systems, many elements were removed. Those were the elements not directly relevant to time tracking (such as policies, currency, payroll functionality, and so on), elements related to user interface (such as order in which to display the elements on the screen), and support structures (such as currency and tax tables, cron tasks, etc.). These features can be added into the product line as optional. However, some design

decisions will have to be made on whether each feature belongs to the data structure layer, the business logic layer, or the display layer of the TAM applications.

The second factor is due to the fact that the data structures in the original systems were designed differently from each other. For instance, in Klok, the time is logged directly against the Project class. In Leia, however, even though through the user interface the time appears to be logged directly against the Project class (or the Activity or Task classes), the Project class extends the WorkItem class. Thus, at the data structure level, the time is logged against the WorkItem class. This is based on different design decisions in the architecture of Klok versus Leia. However, to make the product line general, we had to settle on one way of representing the Project – TimeEntry association. Thus the model we selected would not exactly match that of all the original applications, but the initial functionality is preserved.

Due to the two factors described above, it is hard to estimate how "close" the generated systems for the original case studies would be to the original applications. The features in the generated systems should completely correspond to those of the original applications in the reduced stage (once the functionality not directly related to time entry has been removed). The similarities in the design of the generated systems with those of the original ones will vary.

The changes in design might result in the generated applications being "superior" or "inferior" to those of the original applications. This will also vary based on the design of the original system and the design decisions made during the construction of the product line. This quality of the application design is dependent on the skill of the individuals involved in design decisions and the complexity of the application's feature set.

5.3. Summary

We have addressed the problems stated in the Introduction:

P1. We can now generate any TAM system with a combination of features that have been modeled directly from the product line model. The applications similar to those we analyzed no longer have to be written from scratch.

P2. To introduce a modification to the code base of multiple systems, all we have to do is to modify the product line. The subsequent code generation will propagate the modification to all systems based on the product line, thus alleviating maintenance issues.

P3. The code for the systems is generated directly from the product line model using the VML4Umple compiler, which allows for model-driven development.

We have achieved the following objectives:

O1. We have generated a product line for time and activity management applications, allowing for generation of a variety of TAM systems and for maintaining multiple systems from one product line model.

O2. The combination of VML and Umple allowed us to describe systems in a flexible way. VML has the ability to define multiple concerns (each to address a particular feature within the system) through the hierarchical structure of variation points and variants. This, combined with Umple's ability to define code in compact fragments and generate object-oriented code, makes VML4Umple a good notation to facilitate model-driven development.

O3. We have derived a general methodology to allow the creation of product lines in other domains.

5.4. Future Work

There are several avenues for future work based on our findings.

As mentioned in the section 5.1.2, the generation of the systems based on the original applications that were used in the case study can be automated. To achieve this, the product line features would need to be annotated with the name of the application(s) in which they originated. The parser would need to be modified to take annotations into account when generating a time tracking

application. For instance, a statement "invoke(TimeTrex)" could invoke all the features from the product line that originated with the TimeTrex application.

An application can be written to simplify the feature selection for time tracking applications that are to be generated from the TAM product line. The product line features (VariationPoints and Variants) can be represented through a user interface as a dependency tree, allowing a developer to pick which features in the time tracking application would need to be supported. A selection of features that require presence of other features would automatically trigger the selection of those requirements. Once all the features for a system have been selected, the invocation file, the Umple code and the object-oriented code for the corresponding application could be generated. The invocation file and the Umple code would serve as documentation of the generated application, as well as to allow manual modification to further tweak the resulting system.

The application described above could accept an invocation file to pre-select features in the dependency tree presented to a developer. That way, if a developer does not wish to manually adjust the invocation file, it could be done through the user interface.

Another part of the process that can be partially automated would be the addition of features from a new application into an existing product line. Using a UML model of the new application, the corresponding feature set can be presented as a

dependency tree, alongside the dependency tree of the product line. The features from the new application can then be either mapped to the existing features (resulting in no need for their implementation), or can be designated as alternative or optional (resulting in their inclusion in the tree). Not all features would be easy to migrate in this way (for instance, an alternative feature might require modifications to an existing alternative feature). However, allowing some of the features to be integrated in this manner, would allow for the focus on the remaining features, shortening the integration time. The more feature sets from different applications are added to the product line in such a manner, presumably the more the future integrations can be shortened. This is because the product line would be able to accommodate a larger number of permutations of feature sets.

Product lines can be created for a variety of domains other than time and activity management. Applications such as registration systems for events, blogs and forums, budget applications, shopping cart and point-of-sale systems, task management and scheduling, calendars, and many others can benefit from being generated from a product line. Moreover, there are enough applications on the market to make into case studies for product line development in all of the aforementioned domains.

This work focuses on generating the data structure layer of applications. However, in combination with the existing work on automation of user interface

creation for Umple systems, it would be possible to also generate the corresponding CRUD user interface code for the systems generated from UML models. This would allow a developer to pick a set of features they need in an application, and both the data structure code and the user interface implementation would then be generated by the VML4Umple parser. This would take care of a large amount of boilerplate code in both the data and the UI layer, leaving the developer to tweak the code as required and add the business logic. This approach would significantly speed up custom application development.

APPENDIX A

KLOK CASE STUDY SUPPORT MATERIALS

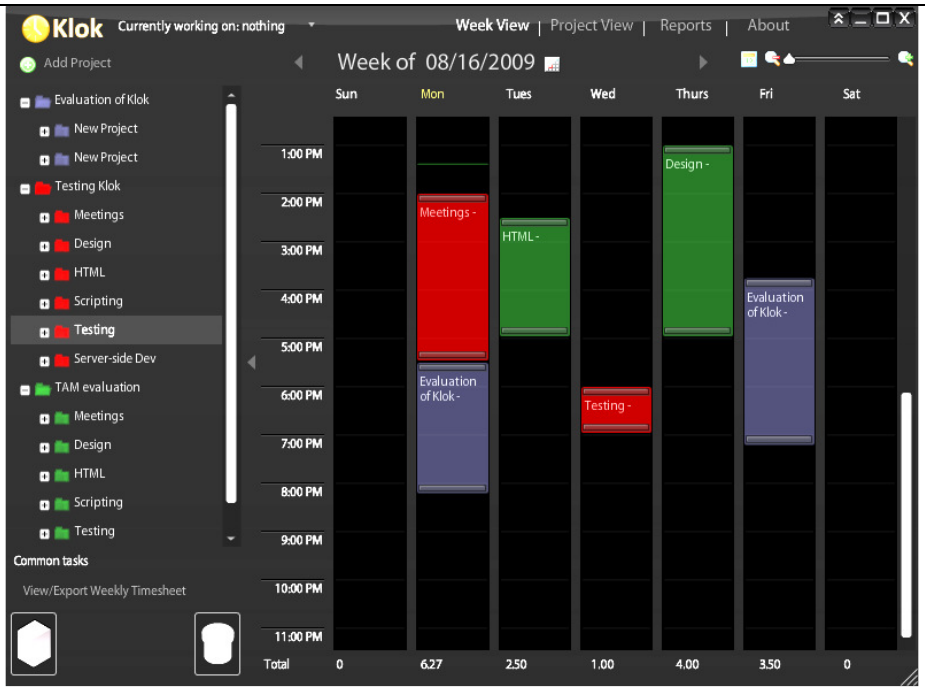

Figure 24. Klok timesheet screen

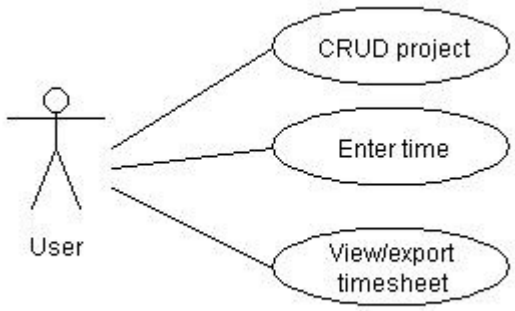

Figure 25. Klok use cases

Java Project class

```
/*This code was generated using the Umple 1.6.0.1717 modeling
language!*/


public class Project
{

  //------------------------
  // MEMBER VARIABLES
  //------------------------

  //Project Attributes
  private String name;
  private String contactName;
  private String contactEmailAddress;
  private String contactPhoneNumber;
  private double hoursEstimate;
  private String colourCode;
  private boolean archived;

  //Project Associations
  private Project parent;

  //------------------------
  // CONSTRUCTOR
  //------------------------

  public Project(String aName, String aContactName, String
aContactEmailAddress, String aContactPhoneNumber, double
aHoursEstimate, String aColourCode, boolean aArchived)
  {
    name = aName;
    contactName = aContactName;
    contactEmailAddress = aContactEmailAddress;
    contactPhoneNumber = aContactPhoneNumber;
    hoursEstimate = aHoursEstimate;
    colourCode = aColourCode;
    archived = aArchived;
  }
```

```
  //------------------------
  // INTERFACE
  //------------------------

  public boolean setName(String aName)
  {
    name = aName;
    return true;
  }

  public boolean setContactName(String aContactName)
  {
    contactName = aContactName;
    return true;
  }

  public boolean setContactEmailAddress(String
aContactEmailAddress)
  {
    contactEmailAddress = aContactEmailAddress;
    return true;
  }

  public boolean setContactPhoneNumber(String
aContactPhoneNumber)
  {
    contactPhoneNumber = aContactPhoneNumber;
    return true;
  }

  public boolean setHoursEstimate(double aHoursEstimate)
  {
    hoursEstimate = aHoursEstimate;
    return true;
  }

  public boolean setColourCode(String aColourCode)
  {
    colourCode = aColourCode;
    return true;
  }

  public boolean setArchived(boolean aArchived)
  {
    archived = aArchived;
    return true;
  }

  public String getName()
  {
    return name;
```

```
  }

  public String getContactName()
  {
    return contactName;
  }

  public String getCotactEmailAddress()
  {
    return cotactEmailAddress;
  }

  public String getContactPhoneNumber()
  {
    return contactPhoneNumber;
  }

  public double getHoursEstimate()
  {
    return hoursEstimate;
  }

  public String getColourCode()
  {
    return colourCode;
  }

  public boolean getArchived()
  {
    return archived;
  }

  public Project getParent()
  {
    return parent;
  }

  public void setParent(Project newParent)
  {
    parent = newParent;
  }

  public void delete()
  {
    parent = null;
  }

}
```

Java TimeEntry class

```
/*This code was generated using the Umple 1.6.0.1717 modeling
language!*/


import java.sql.Date;
import java.sql.Time;

public class TimeEntry
{

  //------------------------
  // MEMBER VARIABLES
  //------------------------

  //TimeEntry Attributes
  private Date date;
  private Time startTime;
  private Time endTime;
  private double duration;
  private String comment;

  //TimeEntry Associations
  private Project project;

  //------------------------
  // CONSTRUCTOR
  //------------------------

  public TimeEntry(Date aDate, Time aStartTime, Time
aEndTime, double aDuration, String aComment, Project
aProject)
  {
    date = aDate;
    startTime = aStartTime;
    endTime = aEndTime;
    duration = aDuration;
    comment = aComment;
    setProject(aProject);
  }

  //------------------------
  // INTERFACE
  //------------------------

  public boolean setDate(Date aDate)
  {
    date = aDate;
    return true;
```

```
}

public boolean setStartTime(Time aStartTime)
{
  startTime = aStartTime;
  return true;
}

public boolean setEndTime(Time aEndTime)
{
  endTime = aEndTime;
  return true;
}

public boolean setDuration(double aDuration)
{
  duration = aDuration;
  return true;
}

public boolean setComment(String aComment)
{
  comment = aComment;
  return true;
}

public Date getDate()
{
  return date;
}

public Time getStartTime()
{
  return startTime;
}

public Time getEndTime()
{
  return endTime;
}

public double getDuration()
{
  return duration;
}

public String getComment()
{
  return comment;
}
```

```
  public Project getProject()
  {
    return project;
  }

  public void setProject(Project newProject)
  {
    if (newProject == null)
    {
      throw new RuntimeException("Cannot set project to
null");
    }
    project = newProject;
  }

  public void delete()
  {
    project = null;
  }

}
```

PHP Project class

```
/*This code was generated using the Umple 1.7.4.1970 modeling
language!*/

class Project
{

  //------------------------
  // MEMBER VARIABLES
  //------------------------

  //Project Attributes
  private $name;
  private $contactName;
  private $cotactEmailAddress;
  private $contactPhoneNumber;
  private $hoursEstimate;
  private $colourCode;
  private $archived;

  //Project Associations
  private $parent;

  //------------------------
  // CONSTRUCTOR
```

```
  //------------------------

  public function __construct($aName, $aContactName,
$aCotactEmailAddress, $aContactPhoneNumber, $aHoursEstimate,
$aColourCode, $aArchived)
  {
    $this->name = $aName;
    $this->contactName = $aContactName;
    $this->cotactEmailAddress = $aCotactEmailAddress;
    $this->contactPhoneNumber = $aContactPhoneNumber;
    $this->hoursEstimate = $aHoursEstimate;
    $this->colourCode = $aColourCode;
    $this->archived = $aArchived;
  }

  //------------------------
  // INTERFACE
  //------------------------

  public function setName($aName)
  {
    $this->name = $aName;
    return true;
  }

  public function setContactName($aContactName)
  {
    $this->contactName = $aContactName;
    return true;
  }

  public function setCotactEmailAddress($aCotactEmailAddress)
  {
    $this->cotactEmailAddress = $aCotactEmailAddress;
    return true;
  }

  public function setContactPhoneNumber($aContactPhoneNumber)
  {
    $this->contactPhoneNumber = $aContactPhoneNumber;
    return true;
  }

  public function setHoursEstimate($aHoursEstimate)
  {
    $this->hoursEstimate = $aHoursEstimate;
    return true;
  }

  public function setColourCode($aColourCode)
  {
```

```
  $this->colourCode = $aColourCode;
  return true;
}

public function setArchived($aArchived)
{
  $this->archived = $aArchived;
  return true;
}

public function getName()
{
  return $this->name;
}

public function getContactName()
{
  return $this->contactName;
}

public function getCotactEmailAddress()
{
  return $this->cotactEmailAddress;
}

public function getContactPhoneNumber()
{
  return $this->contactPhoneNumber;
}

public function getHoursEstimate()
{
  return $this->hoursEstimate;
}

public function getColourCode()
{
  return $this->colourCode;
}

public function getArchived()
{
  return $this->archived;
}

public function getParent()
{
  return $this->parent;
}

public function setParent($newParent)
```

```
  {
    $this->parent = $newParent;
  }

  public function delete()
  {
    $this->parent = null;
  }

}
```

PHP TimeEntry class

```
/*This code was generated using the Umple 1.7.4.1970 modeling
language!*/

class TimeEntry
{

  //------------------------
  // MEMBER VARIABLES
  //------------------------

  //TimeEntry Attributes
  private $date;
  private $startTime;
  private $endTime;
  private $duration;
  private $comment;

  //TimeEntry Associations
  private $project;

  //------------------------
  // CONSTRUCTOR
  //------------------------

  public function __construct($aDate, $aStartTime, $aEndTime,
$aDuration, $aComment, $aProject)
  {
    $this->date = $aDate;
    $this->startTime = $aStartTime;
    $this->endTime = $aEndTime;
    $this->duration = $aDuration;
    $this->comment = $aComment;
    $this->setProject($aProject);
  }
```

```
//------------------------
// INTERFACE
//------------------------

public function setDate($aDate)
{
  $this->date = $aDate;
  return true;
}

public function setStartTime($aStartTime)
{
  $this->startTime = $aStartTime;
  return true;
}

public function setEndTime($aEndTime)
{
  $this->endTime = $aEndTime;
  return true;
}

public function setDuration($aDuration)
{
  $this->duration = $aDuration;
  return true;
}

public function setComment($aComment)
{
  $this->comment = $aComment;
  return true;
}

public function getDate()
{
  return $this->date;
}

public function getStartTime()
{
  return $this->startTime;
}

public function getEndTime()
{
  return $this->endTime;
}

public function getDuration()
{
```

```
    return $this->duration;
  }

  public function getComment()
  {
    return $this->comment;
  }

  public function getProject()
  {
    return $this->project;
  }

  public function setProject($newProject)
  {
    if ($newProject == null)
    {
      throw new Exception("Cannot set project to null");
    }
    $this->project = $newProject;
  }

  public function delete()
  {
    $this->project = null;
  }

}
```

APPENDIX B

LEIA CASE STUDY SUPPORT MATERIALS

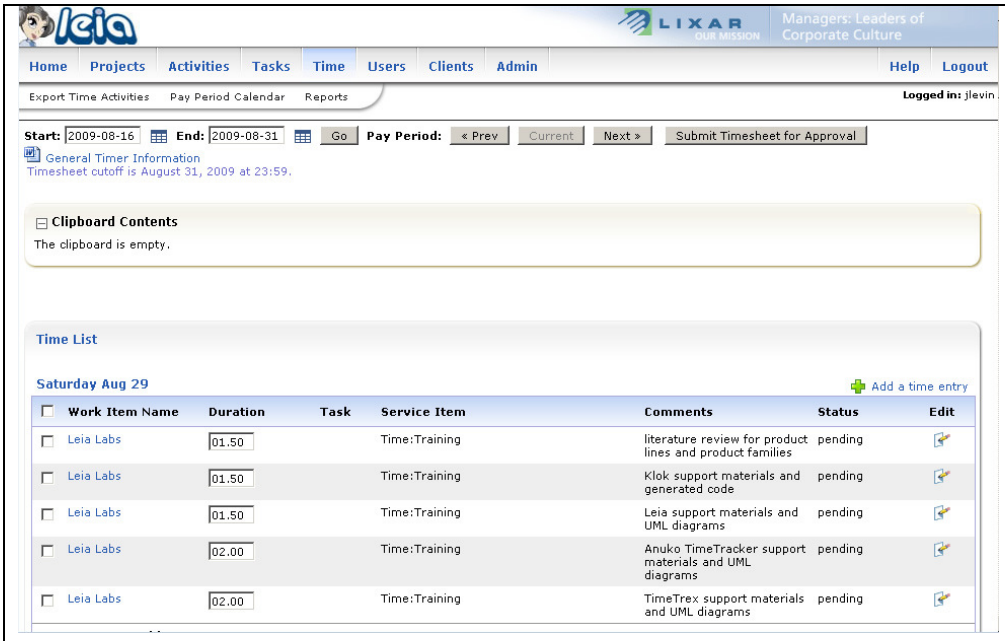

Figure 26. Leia timesheet screen

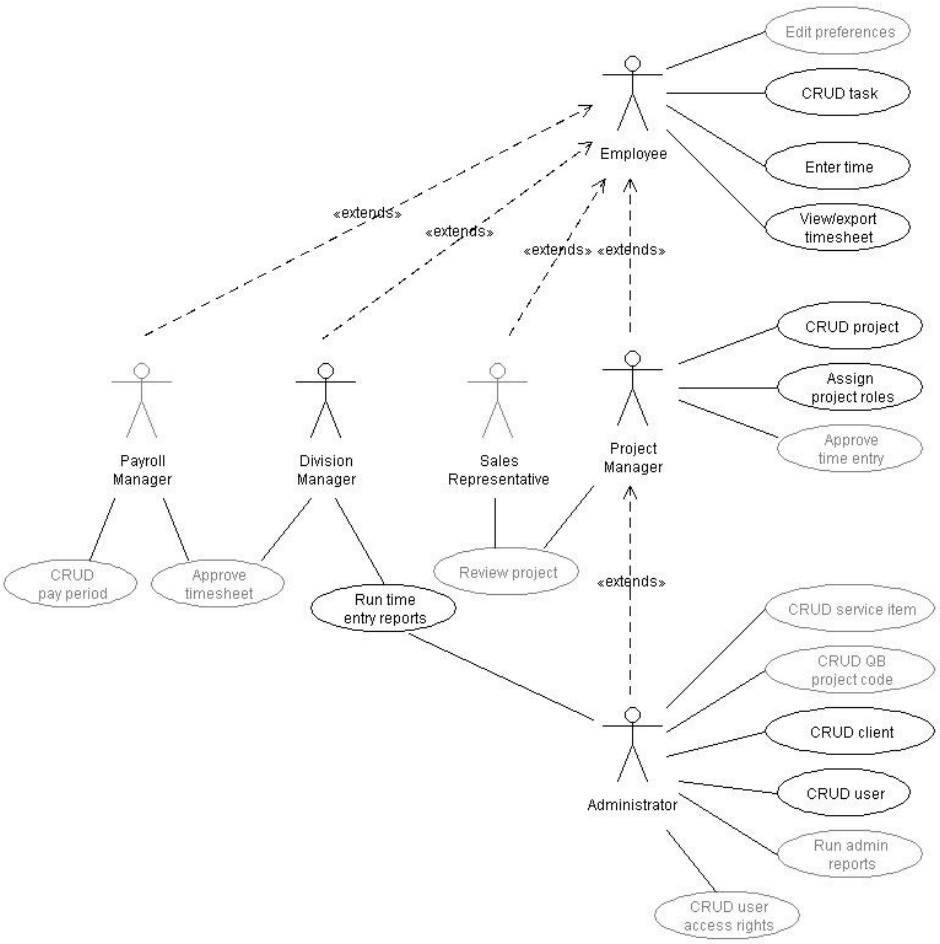

Figure 27. Leia use cases (those grayed out are not relevant to time and activity management)

APPENDIX C

ANUKO TIME TRACKER CASE STUDY SUPPORT MATERIALS

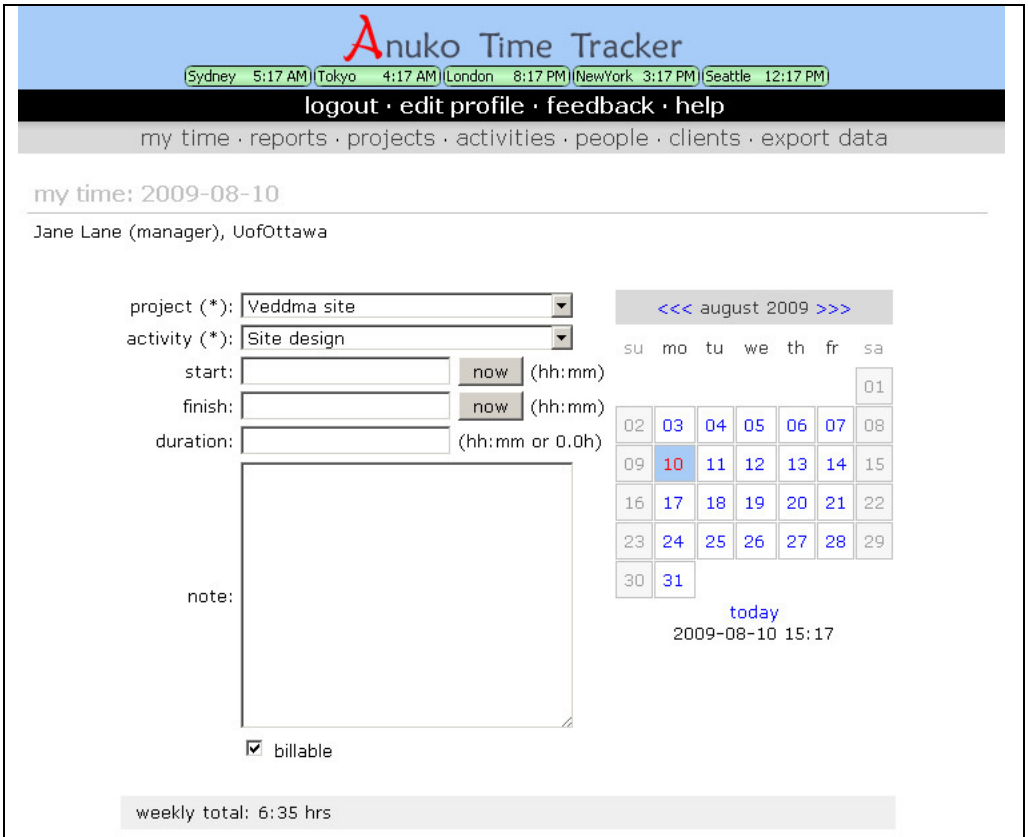

Figure 28. Anuko Time Tracker time entry screen

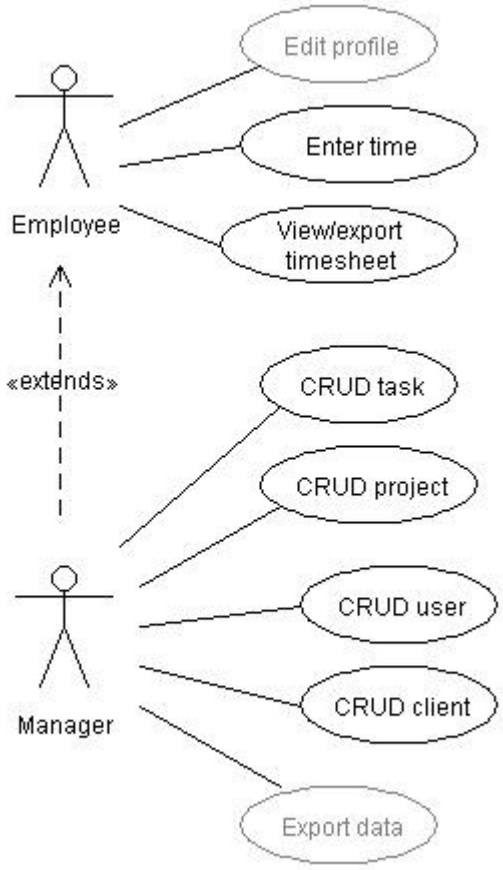

Figure 29. Anuko Time Tracker use cases (those grayed out are not relevant to time and activity management)

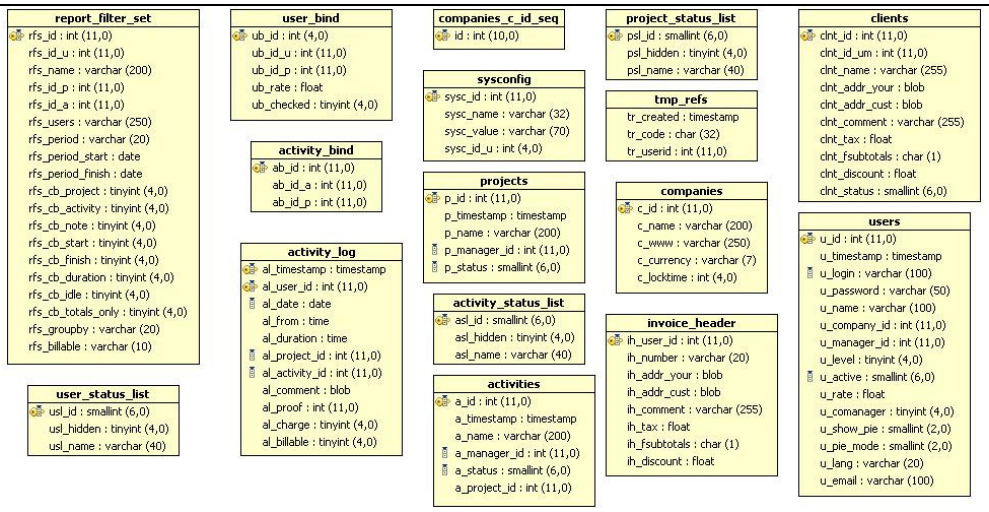

Figure 30. Anuko Time Tracker ERD (without foreign keys)

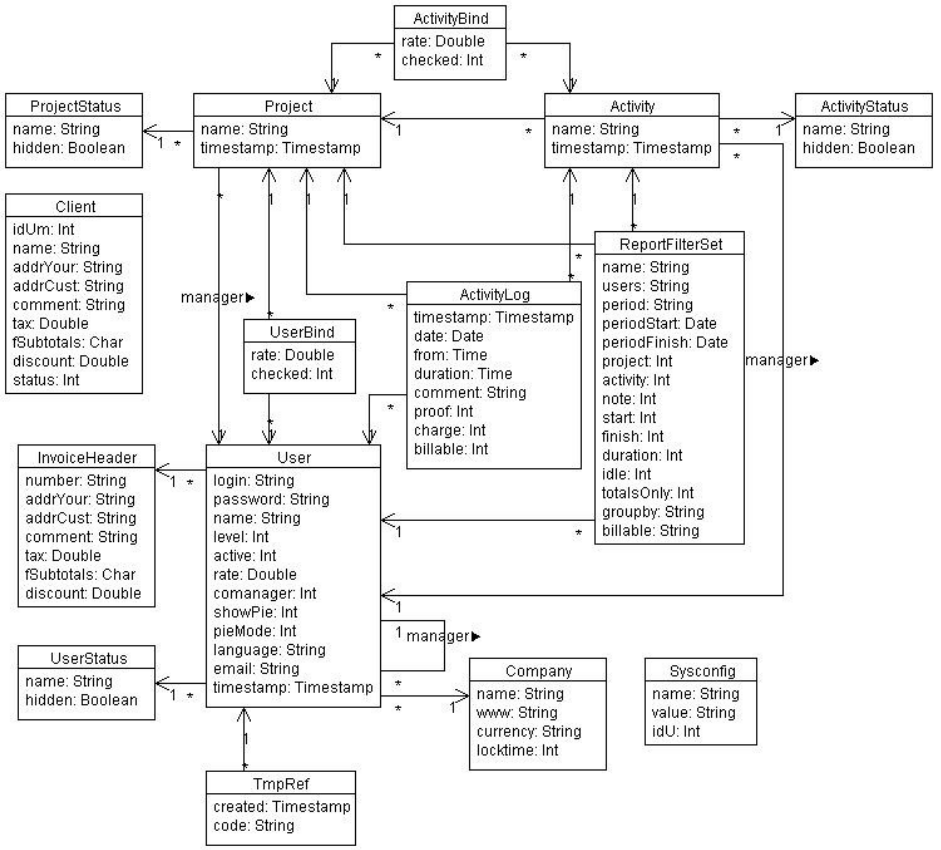

Figure 31. Anuko Time Tracker UML class diagram modeled from the ERD

APPENDIX D

TIMETREX USE CASE SUPPORT MATERIALS

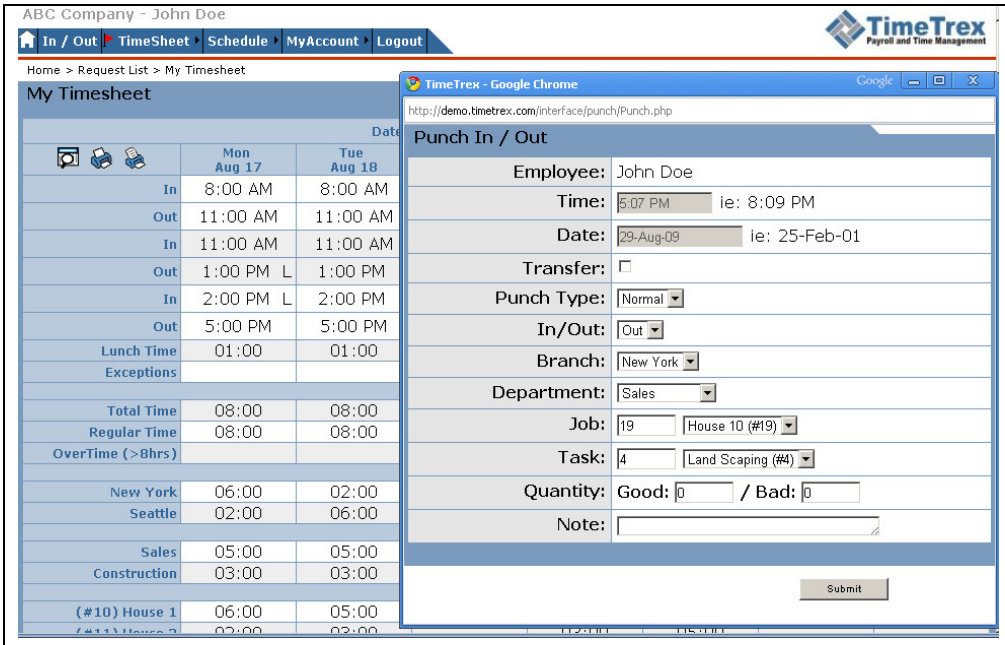

Figure 32. TimeTrex time entry screen

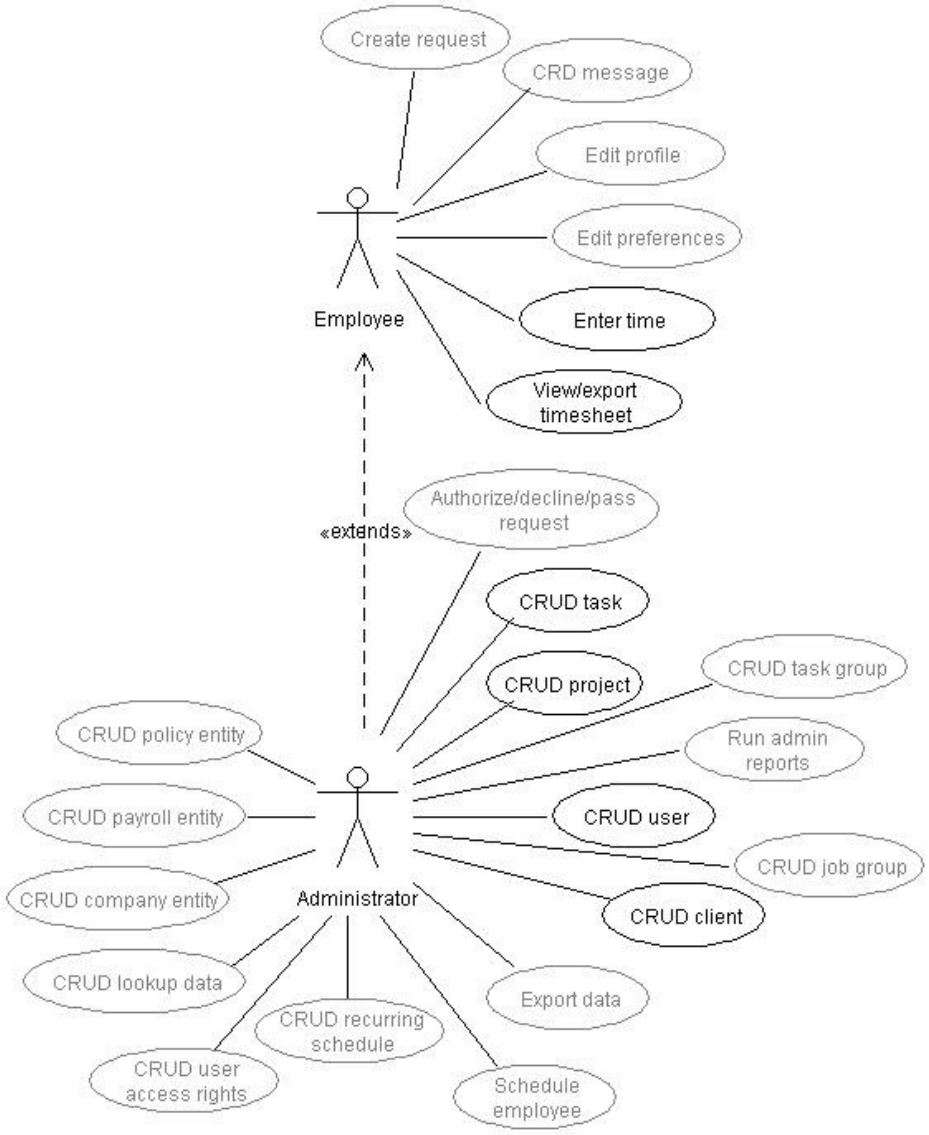

Figure 33. TimeTrex use cases (those grayed out are not relevant to time and activity management)

BIBLIOGRAPHY

Acher, M., Lahire, P., Moisan, S. and Rigault, J.-P. 2009. Tackling high variability in video surveillance systems through a model transformation approach. *Proceedings of the 2009 ICSE Workshop on Modeling in Software Engineering,* pp.44-49.

Allen, R. and Garlan, D. 1997. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology, Vol. 6, Issue 3*, pp.213-249.

Anuko International Ltd. 2009. Anuko Time Tracker. http://www.anuko.com/content/time_tracker/, accessed in 2009.

Ardis, M.A. and Cuka, D.A. 1999. Defining families - commonality analysis. *ICSE '99: Proceedings of the 21st international conference on Software engineering*, Boston, Massachusetts, United States, pp.671-672.

Auer, M., Tschurtschenthaler, T. and Biffl, S. 2003. A flyweight UML modelling tool for software development in heterogeneous environments. *Proceedings of the 29th Conference on EUROMICRO*, Belek-Antalya, Turkey, pp.267-272.

Batory, D. and O'Malley, S. 1992. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology, Vol. 1, Issue 4*, pp.355-398.

Batory, D., Lopez-Herrejon, R.E. and Martin, J. 2002. Generating Product-Lines of Product-Families. *ASE '02: Proceedings of the 17th IEEE international conference on Automated software engineering*, Edinburgh, Scotland, UK, p.81.

Booch, G. 1986. Object-oriented development. *IEEE Transactions on Software Engineering, Vol. 12, Issue 2*, pp.211-221.

Booch, G., Rumbaugh, J. and Jacobson, I. 1999. The Unified Modeling Language user guide. Addison Wesley Longman Publishing Co., Inc.

Book, M. and Gruhn, V. 2003. A dialog flow notation for web-based applications. *Proceedings of the Seventh IASTED International Conference*

on Software Engineering and Applications, Marina del Ray, CA, United States, pp.100-105.

Book, M. and Gruhn, V. 2004. Modeling Web-based dialog flows for automatic dialog control. *Proceedings of the 19th IEEE international conference on Automated software engineering*, Linz, Austria, pp.100-109.

Cugola, G. and Ghezzi, C. 1996. Program families: some requirements issues for the process languages. *Proceedings of the 10th International Software Process Workshop*, Dijon, France, p.48.

Dashofy, E.M., Hoek, A.V.D. and Taylor, R.N. 2005. A comprehensive approach for the development of modular software architecture description languages. *ACM Transactions on Software Engineering and Methodology (TOSEM), Vol. 14, Issue 2*, pp.199-245.

Imperial College. 1997. The Darwin Language - Version 3d., http://www-dse.doc.ic.ac.uk/Software/Darwin/darwin-lang.pdf, accessed in 2009.

Dhungana, D. and Groher, I. 2009. Genetics as a role model for software variability management. *ICSE-Companion 2009. 31st International Conference on Software Engineering 2009 - Companion Volume*, Vancouver, BC, Canada, pp.239-242.

di Nitto, E. and Fuggetta, A. 1996. Product lines: what are the issues? *Proceedings of the 10th International Software Process Workshop, 1996. Process Support of Software Product Lines*, Dijon, France, pp.51-53.

Feiler, P.H., Lewis, B.A. and Vestal, S. 2006. The SAE Architecture Analysis & Design Language (AADL) a standard for engineering performance critical systems. *Proceedings of the 2006 IEEE Conference on Computer Aided Control Systems Design,* Munich, Germany, pp.1206-1211.

Forward, A. 2009a. Umple language online, http://cruise.site.uottawa.ca/umpleonline/, accessed in 2009.

Forward, A. 2009b. VML Online. http://cruise.site.uottawa.ca/umpleonline/vml.html, accessed in 2009.

Forward, A., Lethbridge, T.C. and Brestovansky, D. 2009. Improving Program Comprehension by Enhancing Program Constructs: An Analysis of

the Umple Language. *ICPC '09: IEEE 17th International Conference on Program Comprehension, 2009*, Vancouver, BC, Canada, pp.311-312.

Garlan, D., Monroe, R. and Wile, D. 1997. Acme: an architecture description interchange language. *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, Toronto, ON, Canada, p.7.

Gorlick, M.M. and Razouk, R.R. 1991. *Proceedings of the 13th international conference on Software engineering*, Austin, TX, United States, pp.23-34.

Harel, D. 1987. Statecharts: A visual formalism for complex systems. *Science of Computer Programming, Vol. 8, Issue 3*, pp.231-274.

Jacobson, I., Christerson, M., Jonsson, P. and Övergaarg, G. 1992. Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley.

Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E. and Peterson, A.S. 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study. *Carnegie Mellon University. Technical report CMU/SEI-90-TR-21, ESD-90-TR-222.*

Kang, K.C., Kim, S., Lee, J., Kim, K., Shin, E. and Huh, M. 1998. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering, Vol 5, Issue 1*, pp.143-168.

Kuusela, J. and Savolainen, J. 2000. Requirements engineering for product families. *Proceedings of the 22nd international conference on Software engineering*, Limerick, Ireland, pp.61-69.

Lethbridge, T.C. and Laganière, R. 2005. Object-oriented software engineering: practical software development using UML and Java. McGraw-Hill, Inc.

Levin, J. 2009. System generation for time and activity management product lines - support materials, http://www.site.uottawa.ca/~tcl/gradtheses/jlevin/, accessed in 2009.

Lixar I.T. Inc. 2007. Leia. http://www.lixar.com/, accessed in 2009.

Loughran, N., Sánchez, P., Garcia, A. and Fuentes, L. 2008. Language Support for Managing Variability in Architectural Models. *Software Composition, Vol. 4954*, pp.36-51.

Luckham, D.C. and Vera, J. 1995. An event-based architecture definition language. *IEEE Transactions on Software Engineering, Vol. 21, Issue 9*, pp.717-734.

Lutz, R.R. 1999. Toward safe reuse of product family specifications. *Proceedings of the 1999 symposium on Software reusability*, Los Angeles, California, United States, pp.17-26.

McKeown, R. 2009. Klok. http://klok.mcgraphix.com/, accessed in 2009.

Metzger, A., Heymans, P., Pohl, K., Schobbens, P. and Saval, G. 2007. Disambiguating the Documentation of Variability in Software Product Lines: A Separation of Concerns, Formalization and Automated Analysis. *RE '07. 15th IEEE International Requirements Engineering Conference, 2007*, Delhi, India, pp.243-253.

Mietzner, R., Metzger, A., Leymann, F. and Pohl, K. 2009. Variability modeling to support customization and deployment of multi-tenant-aware Software as a Service applications. *Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems*, Vancouver, BC, Canada, pp.18-25.

Object Management Group.2008. Object Management Group, Unified Modeling Language (UML), version 2.1.2. http://www.omg.org/technology/documents/formal/uml.htm, accessed in 2008.

Object Management Group. 2009. Object Constraint Language. http://www.omg.org/spec/OCL/, accessed in 2009.

Ommering, R.V., Linden, F.V.D., Kramer, J. and Magee, J. 2000. The Koala Component Model for Consumer Electronics Software. *Computer, Vol. 33, Issue 3*, pp.78-85.

Parnas, D.L. 1976. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering, Vol. 2, Issue 1*, pp.1-9.

Ram, A., Kellock, H. and Hjort, P. 1997. Architecting families of software-intensive products. *Proceedings of the 19th International Conference on Software Engineering, 1997*, Boston, Massachusetts, United States, p.580.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. 1991. Object-oriented modeling and design. Prentice-Hall, Inc.

Sánchez, P., Loughran, N., Fuentes, L. and Garcia, A. 2009. Engineering Languages for Specifying Product-Derivation Processes in Software Product Lines. In *Software Language Engineering: First International Conference, SLE 2008. Revised Selected Papers.* Toulouse, France, pp.188-207.

Schmid, K., Becker-Kornstaedt, U., Knauber, P. and Bernauer, F. 2000. Introducing a software modeling concept in a medium-sized company. *Proceedings of the 22nd international conference on Software engineering*, Limerick, Ireland, pp.558-567.

Spenser, J. 2000. Architecture Description Markup Language - Creating an Open Market for IT Architecture Tools. http://www.opengroup.org/tech/architecture/adml/background.htm, accessed in 2009.

Sutton, S.M. and Osterweil, L.J. 1996. Product families and process families, *Proceedings of the 10th International Software Process Workshop*, Dijon, France, p.109.

Taylor, R.N., Medvidovic, N. and Dashofy, E.M. 2009. Software Architecture: Foundations, Theory, and Practice. Wiley Publishing.

TimeTrex Payroll Services. 2009. TimeTrex. http://www.timetrex.com/, accessed in 2009.

University of Toronto. 2000. GRL - Goal-oriented requirement language. http://www.cs.toronto.edu/km/GRL/, accessed in 2000.

Weiss, M. and Amyot, D. 2005. Designing and Evolving Business Models with URN. *Montreal Conference on eTechnologies (MCeTech)*, Montréal, QC, Canada, pp.149-162.