Test-driving EvoSpaces: software visualisation in the real world

Travail de diplôme réalisé en vue de l'obtention du diplôme HES

par :

Raphaël Marmier

Conseiller au travail de diplôme :

Philippe Dugerdil, Professeur de génie logiciel

Genève, le 14 décembre 2007

Haute École de Gestion de Genève (HEG-GE)

Informatique de Gestion

Déclaration

Ce travail de diplôme est réalisé dans le cadre de l'examen final de la Haute école de gestion de Genève, en vue de l'obtention du titre (…). L'étudiant accepte, le cas échéant, la clause de confidentialité. L'utilisation des conclusions et recommandations formulées dans le travail de diplôme, sans préjuger de leur valeur, n'engage ni la responsabilité de l'auteur, ni celle du conseiller au travail de diplôme, du juré et de la HEG.

« J'atteste avoir réalisé seul le présent travail, sans avoir utilisé des sources autres que celles qui sont citées dans la bibliographie. »

Fait à Genève, le 14 décembre 2007

Raphaël Marmier

Remerciements

Je remercie le professeur Dugerdil de m'avoir permis de travailler sur ce passionnant projet qu'est EvoSpaces. Je le remercie aussi pour sa très grande disponibilité et ses précieux conseils.

Ce travail n'aurait pas été possible sans le soutien de ma femme Anouk, et de mon fils Jonas, que je remercie du fond du coeur.

Je tiens à remercier Eric Rose qui a relu de grandes parties de ce mémoire et en a corrigé l'Anglais, le rendant beaucoup plus lisible.

Je remercie enfin tous ceux qui m'ont apporté leur appui, à un moment ou un autre. Ils sont trop nombreux pour être tous cités ici.

Abstract

EvoSpaces is a prototype software visualisation tool that represents the source code of an application as a city in 3D. The size and shape of the buildings are mapped to software metrics. A special “night mode” allows visualising execution traces as rays of light between buildings.

Our mission was to try EvoSpaces on an industrial-sized real world application.

The first part of this work consisted in adapting EvoSpaces to a new database schema, generate metrics on the code of our target application and load execution traces. To do so, we implemented the Abstract Factory pattern in the database backend and wrote specialised tools.

In the second part, we defined what is expected from EvoSpaces on a software maintainer’s point of view, and the design rules it must follow in order to be useful. Following a scenario of discovery, we asked ourselves typical maintainer questions. We then checked if we could answer them with the tool.

Through this experimentation, we uncovered a flaw in how metrics were mapped to buildings, for which we provide a solution. We also made a number of small improvements and bug fixes.

In this work, we believe we have brought EvoSpaces to the next level. The revised display combined with new software metrics has demonstrated its usefulness. More generally, the concept of driving a 3D world with software metric is validated.

In conclusion, we believe that EvoSpaces includes now all the necessary elements to become a product for real world use.

Finally, we assert that imagining and developing innovative software metrics is the key to EvoSpaces future.

Table of Contents

Index of Tables

Index of Figures

Introduction

In the seventies and eighties, large organisations turned to computers and software to support their activities. These software systems grew along one axis: the proportion of activities under software management.

For the past 15 years, networking and interconnection became another axis of growth.

As a result, software systems grow in size and in complexity to a point where they become unmaintainable. With time, behaviours are added and documentation gradually gets out of synch. After a few years, maintainers do not dare modify things for fear of introducing subtle bugs. Rather, they only add code, often introducing duplicate features. Complexity increases even more.

Paradoxically, these factors contribute to increase the already long lifespan of software systems.

The development of software engineering and object-oriented programming brought help in managing complexity. However, these techniques have allowed software systems to grow larger. The complexity and maintenance problems have remained.

Today, there is a strong need for efficient *reverse engineering* techniques for recovering software architecture and business rules from difficult to maintain software applications, either to regain control, clean up and re-documents the application or to start from scratch entirely and reproduce the original software's behaviour.

Reverse engineering can be described as

> *"the process of analyzing a subject system to: identify the system's components and their interrelationships and; create representations of the system in another form or at a higher level of abstraction." [Chikofsky&Cross 1990]*

Reverse engineering large information systems requires specific tools and methodologies. Philippe Dugerdil, professor of Software Engineering at the Geneva University of Applied Science, conceived such methodology [Dugerdil, 2005]. Under his supervision, his laboratory is developing the necessary tools.

Philipe Dugerdil's method combines standard business analysis based on the *Rational Unified Process* with architecture extraction through code instrumentation and analysis of execution traces. A good example of its application to a real-world case is Sébastien Jossi's work on *Progrès* [Jossi, 2006].

In parallel to this effort, Philippe Dugerdil's team participates to a multiyear research project supported by the Hasler Stiftung together with the University of Zurich (Prof. Harald Gall) and University of Lugano (Prof. Michele Lanza)[1] One of the outcome of this project is a prototype 3D visualisation tool called EvoSpaces.

This tool represents the source code of an application as a city, with houses and skyscrapers representing files. It allows visualisation of the program's execution based on execution traces. Emphasis has been put on intuitive representation and fast navigation.

EvoSpaces aims to let software engineers quickly discover and understand a large software system and help them day after day in their maintenance tasks.

Our mission in this diploma work is to allow EvoSpaces to operate on a sizable, real-world software application, report on the necessary work, propose and possibly implement improvements and finally investigate possible use scenarios. If time permits, we would like to investigate a better paradigm for the city layout.

The real world application we plan to load into EvoSpaces is called PasEvi. It is provided by the State of Geneva's IT Department.

This diploma work is organised as follows:

- Presentation of EvoSpaces and PasEvi.
- Presentation of the work that was necessary to load and to run PasEvi with EvoSpaces. This includes adapting EvoSpaces to a new database schema and generating metrics on PasEvi's source code and load execution traces.
- Presentation of our testing process. We analyse what a users expects from EvoSpaces and how it should perform. We then explore a scenario of application discovery and try to solve real questions using EvoSpaces. In the process, we present the improvements we made to EvoSpaces.
- Presentation of proposed functional and architectural enhancements to EvoSpaces.

[1] EvoSpaces: Multi-dimensional Navigation Spaces for Software Evolution: www.inf.unisi.ch/projects/evospaces/

1. Guided tour of EvoSpaces

1.1 Introduction

EvoSpaces is a software visualisation tool developed by Sazzadul Alam under the direction of Philippe Dugerdil.

It aims to allow engineers to easily grasp the architecture of a complex, industrial-size software system[2] and to navigate its structure with ease.

This involves finding a way to "represent structural and metrics information of the system in the same visual space" (Alam, 2007, p. 1). Also, this is done using an intuitive display paradigm.

The originality of EvoSpaces resides in its use of a familiar visual metaphor: a 3D landscape. This landscape can be navigated "à la Doom" at all times, regardless of display mode and parameters.

Figure 1

Mozilla 1.7 in EvoSpaces' main view.

Labels can also be toggled on to identify elements by their source code name.

[2] Which we will refer to as the *target system* for the rest of this diploma work.

The version of the program we start with displays the source code of Mozilla 1.7, written in C and C++. Buildings represent files. Blocks, represented on the ground by light green squares, represents directories. Blocks are nested accordingly and a colour gradient is used to distinguish among them.

Building dimensions and appearance are governed by measurements taken on the corresponding entity according to user selected metrics.

Generally speaking, the drawing of objects relative to each other is related to content.

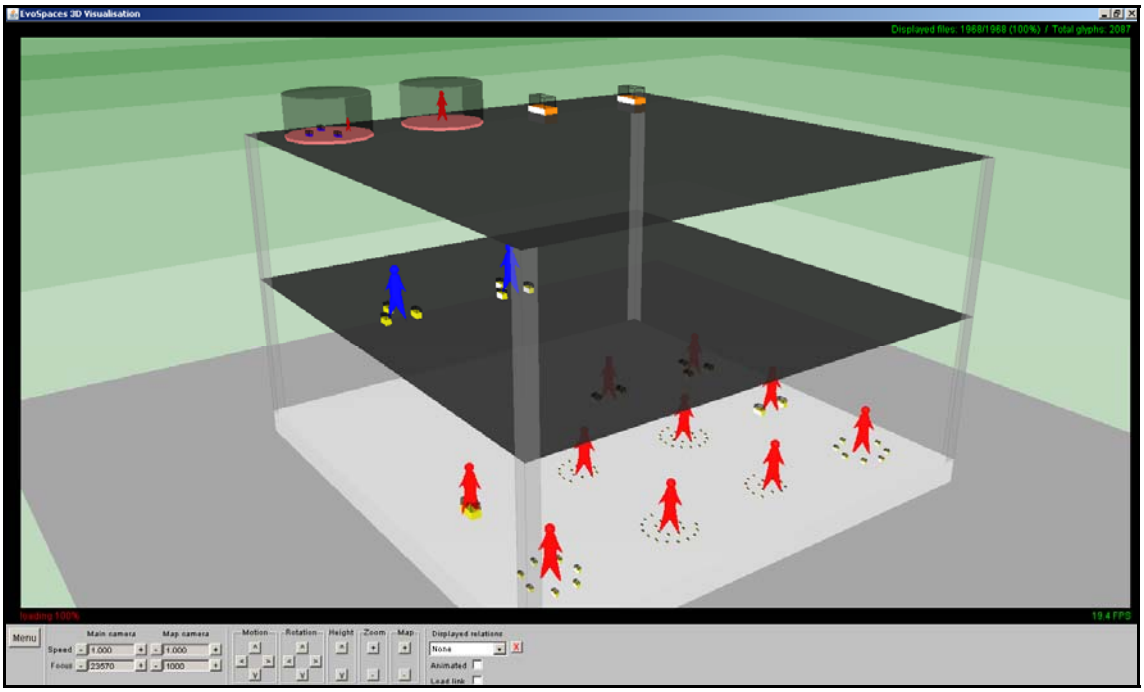
The program has two modes. The daylight mode is used to navigate the code and inspect it in details. The night mode is used to display execution traces.

1.2 Daylight mode

This mode is dedicated to understanding the software system under scrutiny. Buildings have meaningful surface and clicking on them reveal the content of the represented entity.

Figure 2

Open building revealing code elements

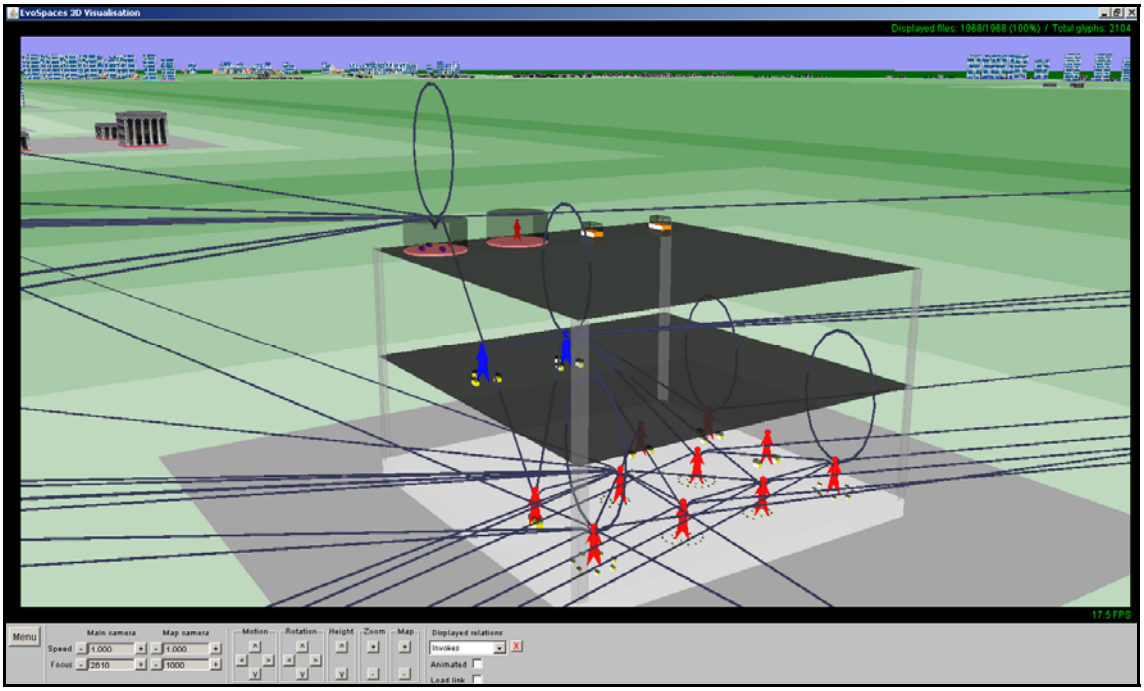


The stickmen represent methods (blue) or functions (red). The small boxes represent variables. When they are displayed close to a function, they are local to it. On the upper floor we find attribute variables and possibly internal classes (puck shapes).

Various kinds of associations, such as invocation or inheritance, can be displayed. They are represented as pipes connecting entities. They can be animated to show in which direction of the association goes. In EvoSpaces terminology, associations are also called “relations”.

Figure 3

Invocation associations of a file



This figure shows invocation associations of a file. We can see various invocations to and from the file’s functions and methods. A feature that is made clearly visible here is internal invocations. Another feature is the display of recursive invocations by drawing a looped pipe on the method or function.

1.3 Night mode

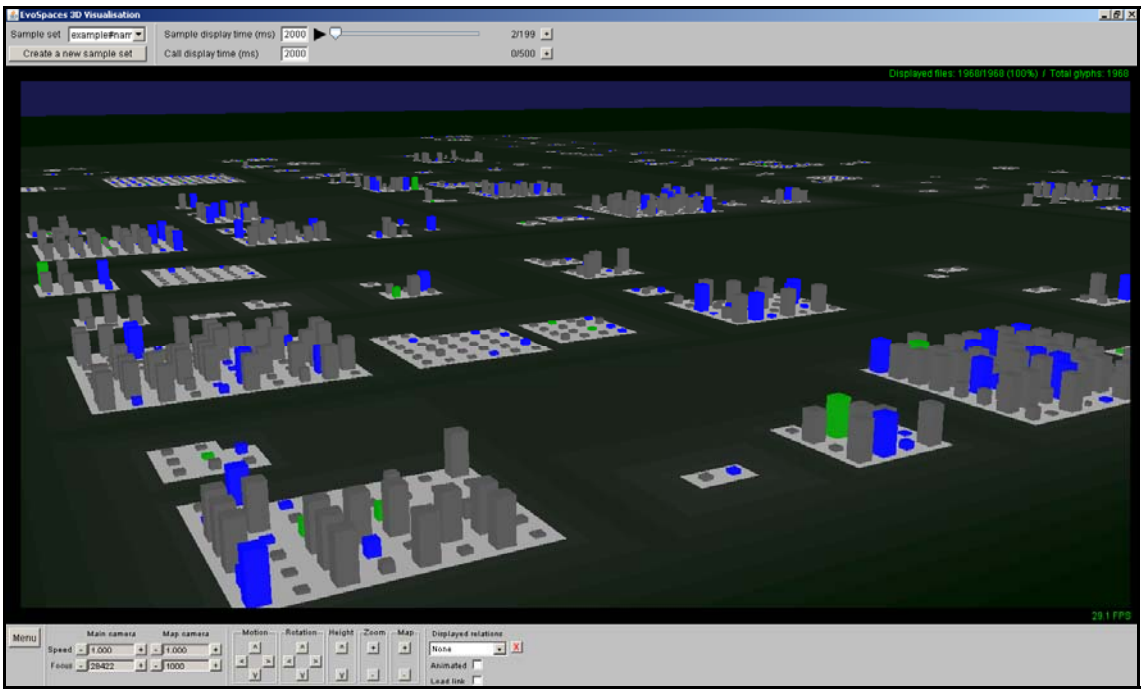
The night mode is dedicated to dynamic analysis. Execution traces can be loaded and segmented at will. Display follows a two stage approach: macro and micro.

The macro approach displays each segment in turn, highlighting involved buildings with a colour corresponding to the number of times it is involved in the segment.

The micro approach displays the execution call by call, for the current segment. A line is drawn between involved buildings for each single call.

Figure 4

Trace segmentation: files involved in a segment



2. EvoSpaces internals

This chapter is an overview of the architecture and inner working of EvoSpaces.

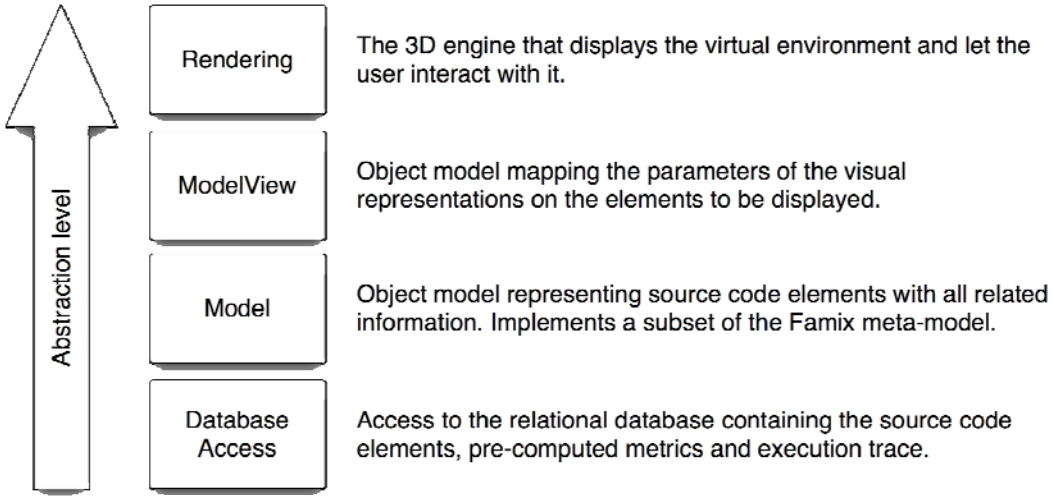
Our knowledge of Evospaces comes from three sources: papers on EvoSpaces; direct answers from Mr. Dugerdil and Mr. Alam; and our own experiments and investigations.

2.1 Architecture

EvoSpaces' architecture is carefully layered to allow flexibility and future growth. The following figure presents the different layers and their function [Alam, 2007, p. 2].

Figure 5

EvoSpaces architecture



Source: Alam 2007, p. 2

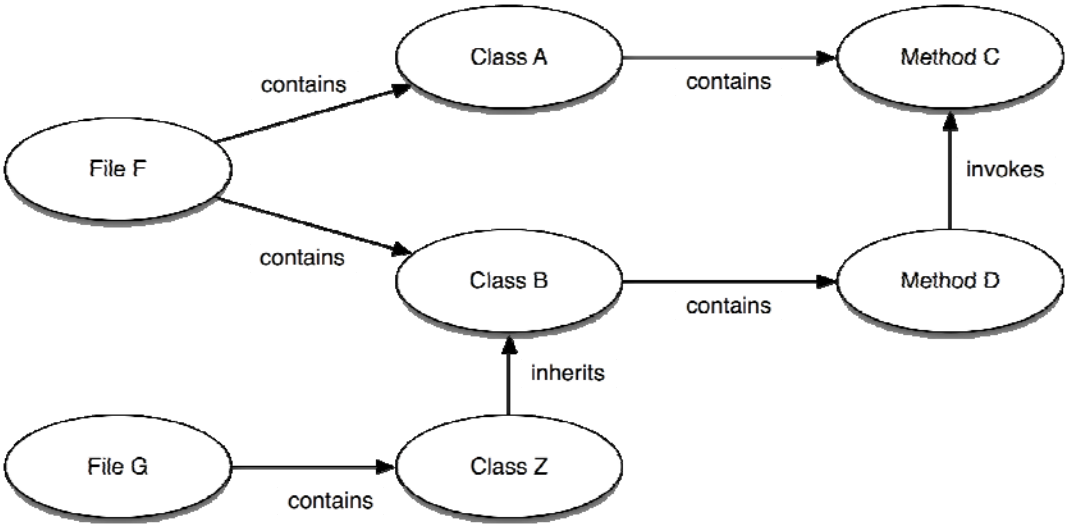
This architecture is detailed in the following sub-chapters.

2.2 The Model

At the core of EvoSpaces we find a generic object model, called the Model. The Model stores an internal representation of source code of the *target system*. A generic graph of entities and associations, like the one on following figure, is used.

Figure 6

Generic code graph



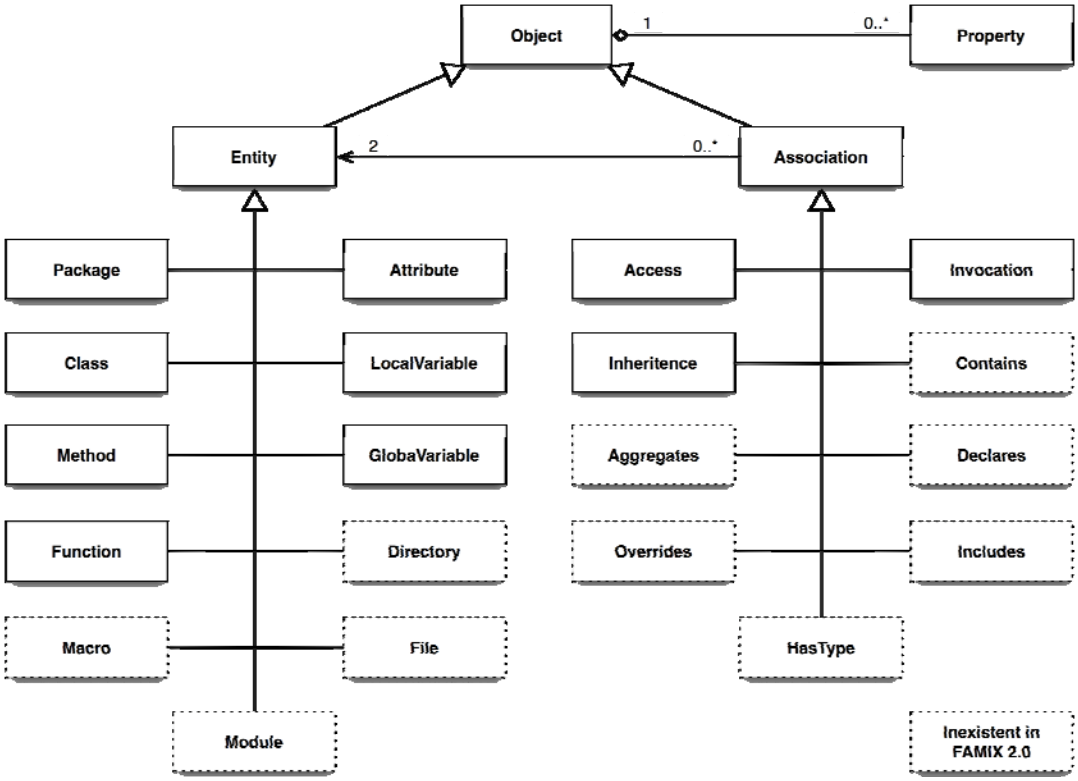
Source: Alam 2007, p. 2

This representation has two advantages: it can represent the code of most programming languages, object-oriented or not, and it makes most manipulations generic, helping keeping other layers simple.

To allow such representation the Model supports a derivative of the Famix 2.0 meta-model [Demeyer, 1999], which is presented in the following figure.

Figure 7

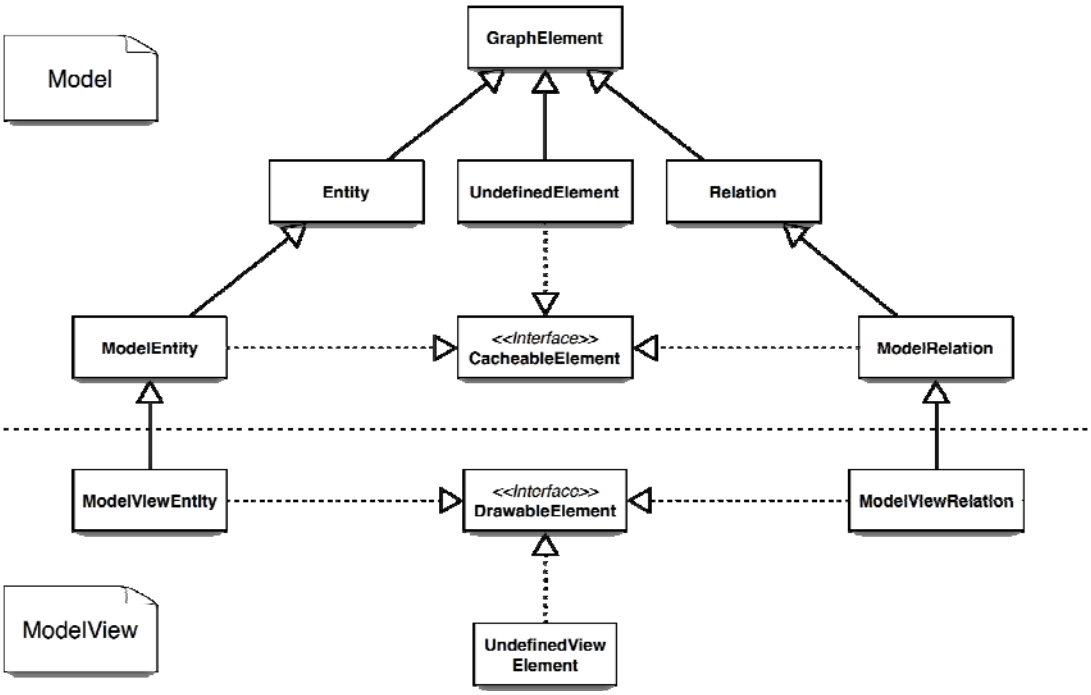
EvoSpaces Famix-like Model



This model is implemented through the following class hierarchy:

Figure 8

Model and ModelView layers object model



The upper part of the diagram represents the object model of the Model layer. ModelEntity and ModelRelation are abstract classes that implement common behaviour. There is a specialised class for each entity and relation (association). Each also has a counterpart in ModelView, derived from either ModelViewEntity or ModelViewRelation.

2.3 The ModelView

The ModelView layer is responsible for the generation and conservation of rendering instructions and parameters used to display entities and associations.

As we said, each entity and association of the Model layer has a counterpart as a ModelView layer entity or association. These classes are named after those in the Model, with the prefix ‘MV’ added. Their role is to store all visualisation and rendering information for their Model counterpart.

A number of helper classes operate on the ModelView objects. Using metrics from the database, glyphs are assigned to objects (i.e. software elements), parameters such as building height and width are computed, position of each object in the 3D space is calculated, rendering instructions are sent to the renderer, and so forth.

Currently, only one city layout engine has been developed: file city. This means only files can be mapped to buildings. Nevertheless, EvoSpaces has been designed to be extensible and new city layout engines, based on classes for example, can be developed relatively easily.

2.4 Rendering

This layer contains the user interface and the 3D scene generation engine, which is based on OpenGL.

2.5 Database backend

A relational database is used to store all the source code elements, the execution traces and the program's parameters.

There is a specialised layer called DBAccess that abstracts database access. A set of Builders[3] loads entity and association objects. A class is responsible for abstracting all other requests as methods.

2.6 Database schema

The data comprises several distinct parts.

- Elements of the source code, stored as entities and associations following more or less closely the Famix 2.0 standard.
- Measurements on source code entities according to a set of metrics.
- Execution traces listing method calls in sequential order.
- Application settings and user preferences.

The elements of the source code are extracted by a source code analysis tool called Evolizer[4]. Evolizer is being developed as part of another on-going project at the University of Zurich by Professor Harald Gall's team. Unfortunately, we at the Haute Ecole de Gestion de Genève did not have access to Evolizer, requiring us to have the source code analysed in Zurich.

[3] See Builder pattern [GoF, 1994]

[4] A platform to analyze source code and software project data : http://seal.ifi.uzh.ch/194/

Evolizer stores its results in its own database schema that has recently been changed. There are two challenges here:

- The database schema is largely undocumented.
- We have no information on how Evolizer analyses and deconstructs source code, and the rules it follows when generating the data.

2.7 Source code

The *target system*'s source code is kept on disk as files and directories. It is used to display, upon user's request, the actual code in an external editor.

2.8 Technology

Evospaces is written entirely in Java as an Eclipse plug-in. It relies on a cross-platform OpenGL library, JOGL, for all display rendering. All other UI elements are built using standard AWT components.

The database engine is currently MySQL 5, but all SQL is conformant to SQL92 so migrating to another DBMS should not pose any problem.

In principle, due to the chosen technology, EvoSpaces can run on Windows, Linux, Solaris and MacOSX without modifications. In practice, some minor issues remain and the platform of choice is Windows for now.

2.9 Development environment

Evospaces is developed as an Eclipse java project. MySQL 5 is needed, as well as the MySQL jdbc driver and the JOGL native libraries. A powerful graphic card is recommended, along with 1Gb of RAM.

3. Presentation of PasEvi

PasEvi is the acronym for *PAtrimoine et SItes.Evaluer, Valoriser, Inventorier*. It is a program in use at the State of Geneva. It is a knowledge base application destined to manage information produced by two entities of the Direction du patrimoine et des sites (DPS), the Service des monuments et des sites (SMS) and the Inventaire des monuments d'art et d'histoire du canton de Genève (IMAHGe).

PasEvi is a large application written entirely in Java. It is recent, features a robust architecture and is generally speaking the product of good software engineering practices.

4. Loading PasEvi in EvoSpaces

4.1 Analysis of PasEvi's source code

As we said earlier, we don't have a code analyser of our own and we rely on Harald Gall's Evolizer tool to load PasEvi's source code into a relational database.

As a consequence, we had to prepare a CVS repository with PasEvi's source code on a laptop computer that Mr. Alam and Mr. Dugerdil took to Zurich. There, they ran Evolizer on it and brought the result back in the form of a MySQL database dump.

4.2 Understanding the data

There are two aspects about understanding data stored in a relational database:

- The database schema, which models more or less closely the structure of the data.
- The data itself, in which can be hidden additional structural information.

All we had to begin with was a database dump in SQL. We knew Evolizer had evolved a lot in two years and we expected the database schema to be very different from the one of previously used by EvoSpace. Worse, we had almost no documentation[5] on the new schema.

We also had no documentation on how and under which rules the data was generated, so we would have to discover that while working on EvoSpaces.

We decided to load the dump into MySQL and use Azzuri Clay data modelling application to connect to the database and to reverse engineer the schema. Doing the same on the old schema would allow us to compare them and find were the data we needed had migrated, and how it was stored.

4.2.1 Recovering the new schema

At first, we thought that this would be enough to provide us with a clear picture of the data. Unfortunately, Evolizer's schema only implements tables, but lacks integrity constraints on foreign keys. We had no choice but to carefully scan through all tables and identify the foreign key columns ourselves.

[5] We were provided with a short explanation on the role most important tables.

At this point, we could have picked any graphical data modelling tool and redraw the whole model. But this meant painstakingly recreating all the schema objects from scratch.

Instead we used the CayenneModeler tool, which is part of the Apache Cayenne Object Relational Mapper project[6], to reverse engineer the database schema. This provided us with only tables. We then proceeded to apply integrity constraints.

Cayenne Modeler has a very simple and powerful interface to help set foreign key column and input integrity constraints on them. We proceeded to add them to the model. While most of them were obvious, many were not and required careful examination of the actual data. This process was quite time consuming.

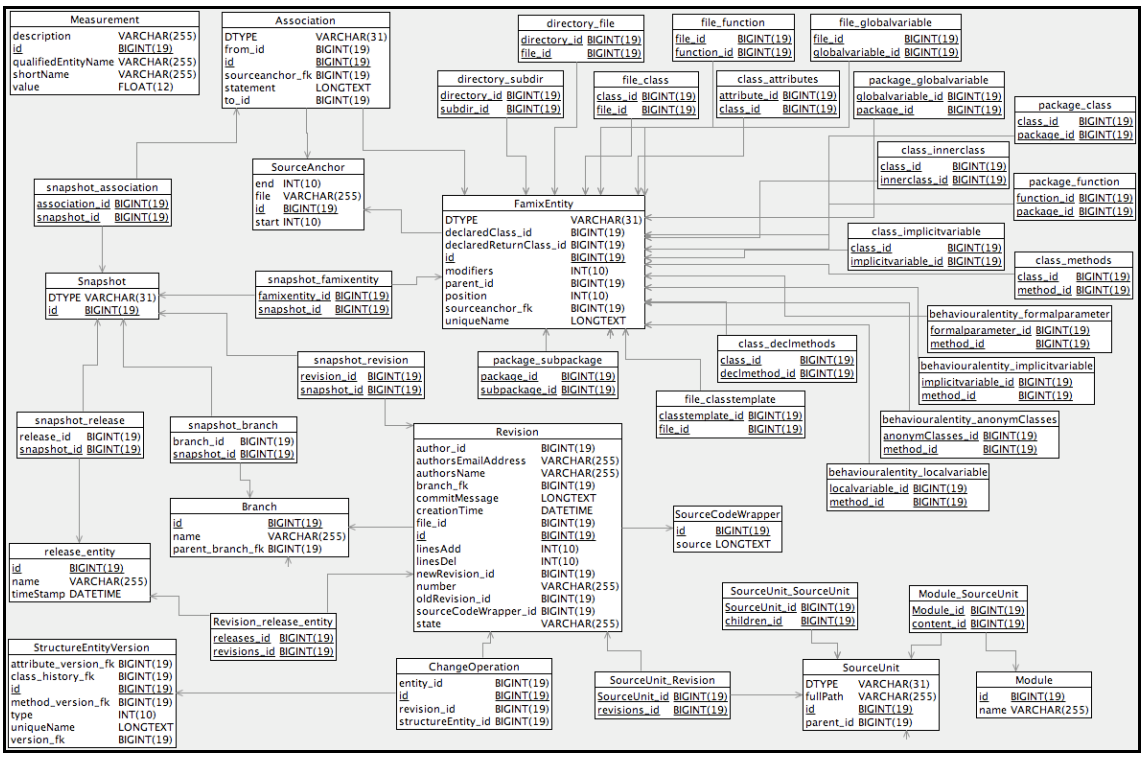
Once this was done, we used Cayenne Modeler to generate the new complete SQL *DDL* of the schema, after which we loaded it into Azzuri Clay[7]. The end result is presented in the next figure.

[6] http://cayenne.apache.org

[7] http://www.azzuri.jp/en/software/clay/index.jsp

Figure 9

Evolizer 2007 database schema (partial)



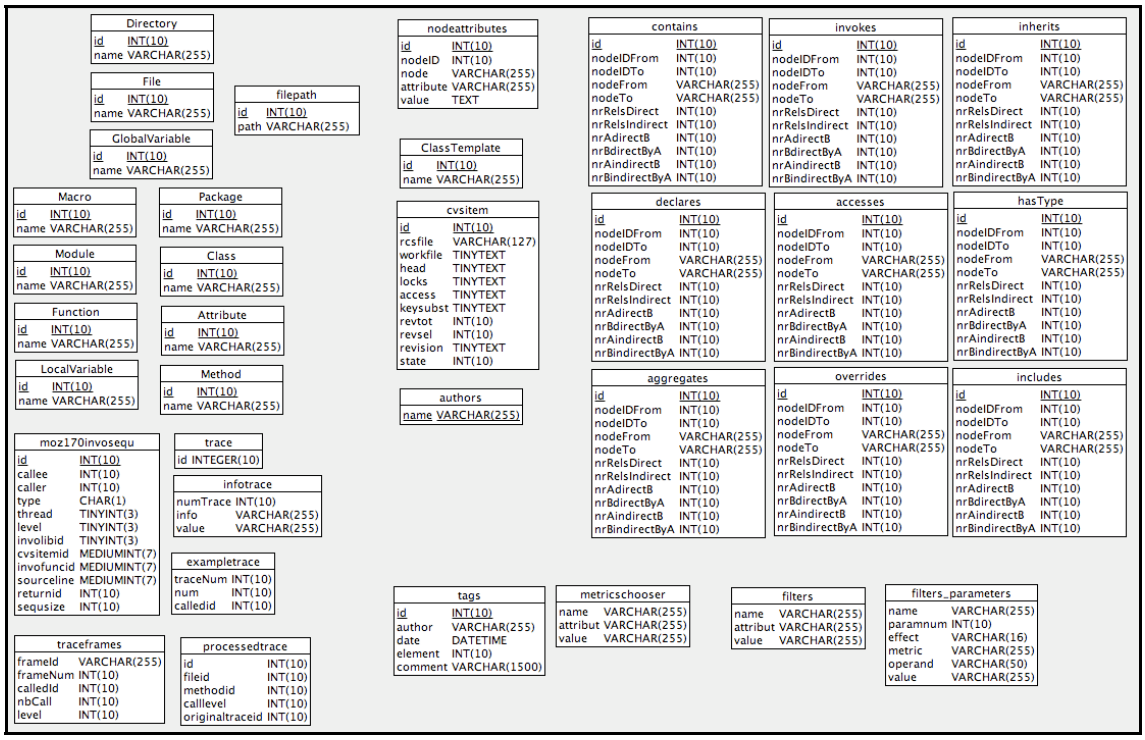
4.2.2 Recovering the old “Mozilla 1.7” schema

We also needed to understand the schema of the database currently used by EvoSpaces, which contains Mozilla 1.7’s source code. But this time we didn’t need to reconstruct the foreign key constraints, because we could easily look up this information directly in the source code of the database backend. Also, as we will see, the older schema’s foreign key associations are much easier to guess by just looking at the table columns.

We reverse-engineered a running copy of the old database with Azzuri Clay to obtain the following diagram.

Figure 10

Evolizer 2005 database schema (partial)



By convention, we will be calling the schema we received with the PasEvi dump *Evolizer 2007* and the "old" schema on which EvoSpaces was built *Evolizer 2005*[8].

4.2.3 EvoSpaces specific tables

EvoSpaces specific tables are shown on Evolizer 2005 schema diagram above. These tables are *infotrace*, *exampletrace*, *processedtrace*, *traceframes*, *tags*, *metricschooser*, *filters*, *filters_parameters*, *authors*.

4.2.4 Comparing the new to the old

It was immediately apparent that the new schema had not much in common with the old one. Notably:

- All entities are stored in a single table *famixentity* in the new one, while they were scattered over eleven tables in the old one.
- There is now only one table *associations* for storing all associations, while previously they were spread over nine tables.

[8] Even though we are not sure that the old schema was at all created with Evolizer.

- No more *nodeattributes* table that used to store everything related to an entity, but instead new columns in *famixentity* and *associations* and new tables such as *sourceanchor*.
- There is now a table *measurements* dedicated to metrics, which were previously stored in *nodeattributes.* However, this table has been found empty because the metric generation tool was not available on the newest Evolizer version that loaded the database, so no metric data was computed.
- A flurry of new join tables has appeared, such as *file_class*, *class_innerclass*, *class_attributes*, etc...

4.3 Generating new measurements with software metrics

As we just said, Evolizer metrics generation engine was not available. We had to solve this problem by providing at least basic metrics, such as the number of line of code (LOC) and the number of lines and the number of methods and functions per file, so the 3D scene can be constructed properly.

Measurements are linked to entities in the database. Most measurements are derived directly from data found in the database itself.

4.3.1 EvoTools

To group all of the data processing utilities developed in the context of this diploma work and possibly afterward, we decided to launch a separate project under Eclipse which we called “EvoTools”.

Each of the tools comprising the EvoTools project is a command line utility that is run independently, has its own set of arguments and generate its output either on the standard output or writes directly to the database.

4.3.2 Metrics

> *“But what is a metric ? It is the mapping of a particular characteristic of a measured entity to a numerical value.” (Lanza & Marinescu, 2006, p. 11)*

4.3.2.1 Static metrics

Static metrics, metrics derived from the source code, are very useful source of information for the software maintainer. Static metrics represent synthetic information on the characteristics and structure of a program. They help building a mental picture

of the source code of large and complex application. This is why the display of EvoSpaces is based on static metrics.

Static metrics also have the advantage of being stable when no change occurs to the source code.

However, static metrics give no information about the actual execution of a program. For example, it gives no information on how many times a method is actually called when the program is in use.

4.3.2.2 Dynamic metrics

Dynamic metrics are metrics measuring elements of an execution trace. A dynamic metric gives information on what really happen at runtime. In particular, it gives precious information about the real use of program components. For example, it can help determine classes that are never used and should be removed.

4.3.2.3 Static versus dynamic metrics

A software maintainer tends to be more concerned by a static view of the source code he has to maintain. He will want to know which parts of the program are going to be affected by changes and additions to the source code. As such, static metrics are more important to him.

Dynamic metrics, describing runtime patterns of the program in relation to actual use, are more useful to developers that seek to extend and evolve the program. Based on them, the developer will be able to make decisions affecting performance and user experience in general.

4.3.3 Generating measurements

Most of these metrics are derived from the content of the source code database. As such, they rely heavily on Evolizer's analysis of the source code. We decided to take a conservative approach and make as little use as possible of potentially de-normalised data, like purely associative tables such as class_methods, *class_attributes*, as we have no clue about the rules that presided to their constitution.

For each class metric, we had to develop an equivalent file metric, in order to be able to use it within the current file city layout. This is done under the assumption that the common object-oriented practice of defining only one class per file has been honoured. Actually, this seems to be generally the case in PasEvi.

Another important aspect of metric generation the fact that each target entity has to have a value computed for it (usually 0), even when it isn't involved in the measured fact itself. This makes the SQL queries more complicated.

4.3.4 Storing measurements

We were not satisfied with the *measurements* table defined in the Evolizer 2007 schema. The new schema uses a *qualifiedEntityName* in place of entity id as foreign key to the code entity to which the measurement applies. Not only does it break compatibility with the previous schema, it also represents a departure from good practice.

Because of this, we choose to create our own table to store the generated measurements. We called this table *mesures* (french word for measurements) and it is composed of the following columns: *id*, *entity_id*, *metric*, *value*.

4.3.5 Measurements

We choose to generate the ten metrics that are detailed below. The first six are fairly standard metrics for measuring size. The last four are metrics that measure static dependencies.

4.3.5.1 Total lines of code (TLOC) and number of lines (nbLines).

We had to parse the source code ourselves to generate these measurements.

The TLOC is obtained by counting the lines after stripping the comments. No effort has been made to compact multi-line instructions.

NbLines is obtained by merely counting the number of lines in the file.

4.3.5.2 Number of methods by class and number of methods by file

We obtained this information by querying the *famixentity* table and using the parent_id column.

For classes, we could have used the *class_methods* table with the same end result.

For files, we had to build a complex request that reuses the number of methods by class metric. In fact, we had to do so to make sure the methods of internal classes are also included.

In hindsight, we realised we could probably have used a SQL join on the *sourceanchor* table, because we discovered that all entities have a counterpart in it. The result would

have been slightly different though, as it would have ignored the implicit <init>() methods that are included by Evolizer.

4.3.5.3 Number of attributes by class and number of attributes by file

Here the case is similar to the previous one and we applied the same methodology.

4.3.5.4 Number of classes calling class and number of files calling file

Using method invocation information from the *association* table, we built the list of all classes that have methods calling at least one of the methods of a given class.

This metric gives us an idea of the number of classes that could potentially be affected by the modification of a given class.

The file version is based on the same principles.

4.3.5.5 Number of classes called by class and number of files called by file

Using method invocation information from the *association* table, we built the list of all classes that a given class potentially relies on. As above, we compute the list of method calls to methods in other classes.

This metric can be viewed as a measure of frailty. It underlines classes that are more likely to be impacted by the modification of the program as a whole.

The file version is based on the same principles.

4.4 Adapting the database backend

The deep differences in the database schema left us with the following alternatives:

- Adapting the database backend to the new schema.
- Converting the data itself from the new to the old schema.

After deliberation, we choose the first one, although it requires us to perform major adjustments on the database backend. The second alternative, which looks promising because we stick with a single schema, was just too uncertain. In particular, it could well prove excessively complex or even impossible.

However, choosing the first scenario forces us to preserve absolute compatibility of DBAccess to the rest of EvoSpaces.

We could also have chosen to create an entirely new and clean schema, based on the Famix 2.0 standard, and adapt EvoSpaces to it. This would have been ideal from a

software engineering standpoint, but it would have required from us to convert both the old and the new database to this new schema.

4.4.1 DBAccess

DBAccess is designed to completely abstract database access. Two classes serve as Builders [GoF, 1994] for entities and associations: *EntityBuilder* and *RelationBuilder*. A third one, *DataRetriever*, abstracts out all other queries behind ad-hoc methods.

In principle, porting these classes to a new database schema should be as easy as rewriting the SQL instructions of their queries. Of course, this assumes that all data present in the old database is also present in the new one in more or less the same form.

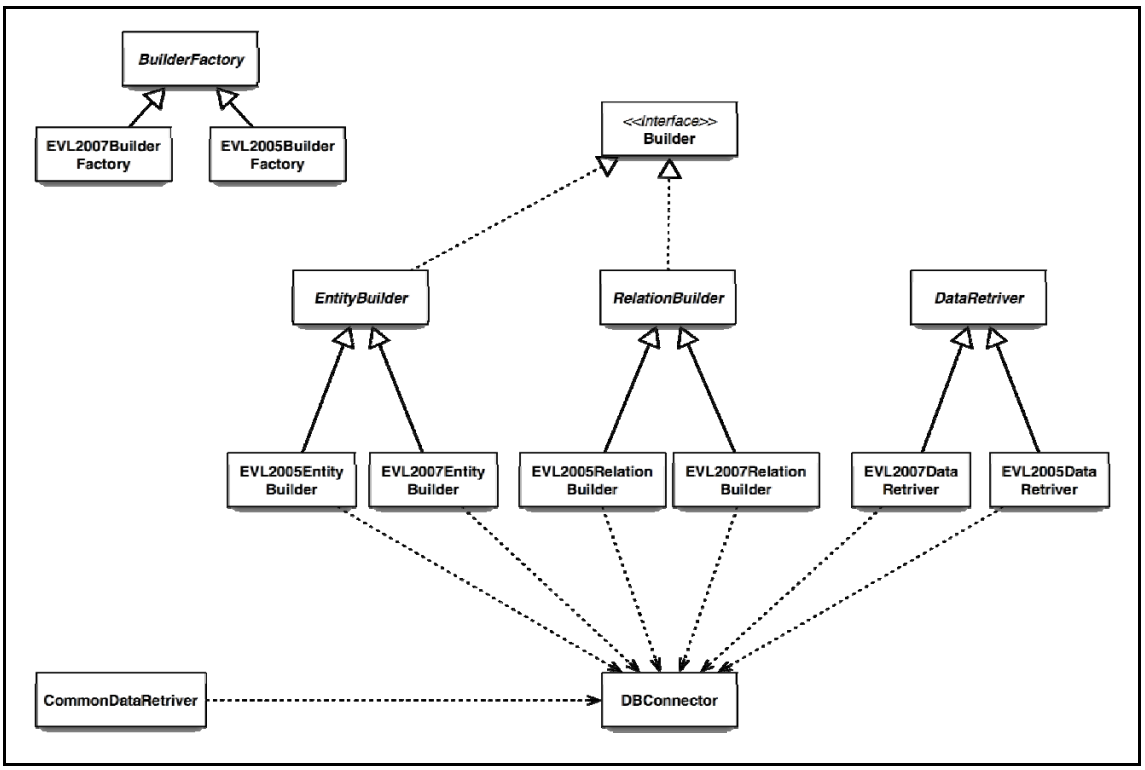
Unfortunately, it wasn't the case, and the porting effort was everything but painless.

4.4.2 Redefining DBAccess' architecture

Modifying DBAccess directly would render us incompatible with our old database containing the Mozilla 1.7 sources. To avoid this, we decided to reorganise DBAccess around the Abstract Factory design pattern [GoF, 1994]. The new architecture is shown on the following diagram:

Figure 11

DBAccess : new architecture



BuilderFactory, EntityBuilder, RelationBuilder and DataRetriever operate as Singletons [GoF, 1994]. Each of the new BuilderFactory class is responsible for readying the complete set of classes that make up the DBAccess package.

At start-up, BuilderFactory is initialised with an instance of either EVL2005BuilderFactory, to access the older database, or EVL2007BuilderFactory to use the new one. For now, the choice of BuilderFactory is dependant on a new command line parameter that must be set at startup, either “2005” or “2007”. By default, the old schema is chosen.

The first time EntityBuilder, RelationBuilder and DataRetriever are asked for their unique instance, they call the BuildFactory’s singleton instance and obtain from it an instance of the appropriate derived class. This instance is then use as a singleton to serve all future requests.

This architecture, while a little complicated on the surface, allows to preserve previous semantics. As a benefit, very few changes had to be made outside DBAccess. For example, outside classes continue to call for EntityBuilder.getUniqueInstance() just as before.

There is a special case however: a good part of DataRetriever concerns tables that are entirely managed by EvoSpaces. Blindly applying the pattern in this case would lead to prejudicial code duplication. For this reason, we created a separate DataRetriever class, CommonDataRetriever, to collect all operations on these tables in one place. Fortunately, this required only a few changes throughout the rest of the code.

Before we go into the details of the work of adapting the different components of DBAccess, we have to point out that, even though we will talk about it as a sequence, most of this work was done in parallel.

4.4.3 Adapting the entity loader

The EntityBuilder loads an entity identified by its ID from the database and returns an instance of the corresponding Model entity.

All EvoSpaces entities derive from the ModelEntity class. There is one subclass per entity type. The particularity of this implementation is that absolutely all information relating to an entity is stored in the form of “properties”, that is, key – value pairs, held in a HashTable object.

All entity types found in the new database existed already in the old one, except FormalParameter. Mapping these duplicate types to their respective, already existing entity classes was easy.

FormalParameter represents a single parameter of a method or function. Because it is never displayed and it never contains any reference to another entity, we could safely ignore it.

We had a few hurdles to overcome in order to port the EntityBuilder to the new database:

- Some entity properties[9] have vanished.
- Entities properties are stored in a different way.

4.4.3.1 Survey of properties

Entity properties used to be stored as key – value pairs in the *nodeattributes* table of the old schema, which made loading them straightforward.

[9] See the Famix diagram above for the explanation on how properties relate to entities.

In the new schema, they are represented by columns in various tables, which brings the data model closer to the Famix standard, but made our task more difficult.

The first task was to find out which properties had survived in the new schema, and when they had, where they were now stored. The following table shows what happened to the properties in the new schema:

Table 1

Entity properties in Evolizer 2007

Evolizer 2005	**Evolizer 2007**	**Used by EvoSpaces**	**Famix 2.0**
accessControlQualifier	n.a.	No	Yes
file	*file* column in table *sourceanchor*	**Yes**	Yes
hasClassScope	n.a.	No	Yes
isConstructor	n.a.	No	Yes
isDestructor	n.a.	No	No
isVirtual	n.a.	No	No
lineNo	Famix compliant sourceanchor with start and stop position	**Yes**	No
typeCategory	n.a.	No	No
uniqueName	*uniquename* column in table *famixentity*	**Yes**	Yes
name	n.a.	**Yes**	Yes

We were only interested in the properties that are actually in use in EvoSpaces. So we scanned EvoSpaces' source code for all access to entity properties. We could do that quickly by leveraging the unix foundation of MacOSX, our work platform:

```
find . -exec grep -no '\(getProperty[(].*[)]\)' {} \; -print
```

Issued at the root directory of the sources of EvoSpaces, this unix[10] command lists all calls to the getProperty() method of an entity[11]. By inspecting the output, we determined the four properties that are used. These were the properties we had to recover.

[10] We did all our work under MacOSX, which has a unix foundation. This let us harness the power of the unix shell and utilities when we had searches like this one to do.

[11] We later discovered that the Eclipse environment provides a similar function!

We see that there are basically two cases: some properties have changed location and/or format, while some others have simply vanished.

4.4.3.2 Designing a new property loader

The property loader used to be very simple, as it just blindly loaded the contents of the table *nodeattributes* corresponding to the entity and stored all records as properties. At the same time, it loaded all measurement data about the same entity in the process. This behaviour has to be reproduced as well.

Now the property loader had to make explicit queries to the database to access precise columns. At the same time, it still had to load arbitrarily named measurements for the entity.

We chose to add a new method dedicated to loading all the “standard” properties such as *name*, *uniqueName* and *file*. After calling it, the loader scans the *measures* table to grab all measurements for the entity.

4.4.3.3 Recovering uniquename and file

This was the straightforward part. *uniqueName* and *file* are columns in the *famixentity* respectively *sourceanchor* tables. It was just a matter of fetching them.

4.4.3.4 Recovering lineNo property

The current version of Evolizer doesn’t generate line number reference anymore. It is now Famix conformant and now stores, for each entity, the start and the end indices of definition alongside of the source file’s name in the *sourceanchor* table.

In principle, it is possible to determine the line number from the start index by counting the number of line-feeds appearing before it.

Here however, we had to give up, as we found out that the start indices reported by Evolizer were actually not reliable. Indeed, it seems that when the entity is a class, the start index is computed from a source file that has been stripped of its comments. When the entity is a method, the index corresponds to the beginning of the method name in the declaration, not the beginning of the declaration.

Intentional or not, these seem to be departures from the Famix 2.0 standard, which states about the source anchor of an element:

> *« Identifies the location in the source where the information is extracted. […] <start_index> and <end_index> are indices starting at 1 and holding the*

beginning/ending character position in the source file. » [Demeyer, Tichelaar, Steyaert, 1999, p. 8]

This is rather indicative of a work in progress and the behaviour is likely to change again in the near future. As a consequence, there is no point for us to invest time into recovering line numbers until Evolizer has stabilized on this issue.

4.4.3.5 Recovering the name property

In the old Evolizer 2005 database, there used to be a *name* property that was shorthand for the *uniqueName*, which as the name implies has to be unique and thus can be very long. Unfortunately, the current version of Evolizer doesn't generate this information anymore.

We decided to add a column name in the *famixentity* table and to generate that property once and for all before EvoSpaces is run for the first time.

To do this, we added a tool to EvoTools called "dbPrep". dbPrep accesses the database and uses a regular expression to generate the short name from the *uniquename.* Then it stores it under the column *name*.

For now, dbPrep understands the following entities' names: *FamixDirectory*, *FamixFile*, *Class*, *Attribute*, *LocalVariable*, *GlobalVariable*, *Method*.

For all other types, the name is just copied from *uniquename*.

4.4.4 Adapting the association loader

The association loader is implemented in the class RelationBuilder, which has several capabilities. It can load an association identified by its ID from the database and returns an instance of the corresponding Model association. It can also load all associations into an entity or load only 'contains' associations into it.

In EvoSpaces, all classes implementing associations derives from the class ModelRelation.

The association model of EvoSpaces entirely reflects the one found in the old database schema. There were nine possible associations, stored in separate tables: *contains*, *invoke*, *inherits*, *declares*, *accesses*, *hasType*, *aggregates*, *overrides* and *includes*.

The new database has a single table *association*. In this table, the column DTYPE informs us on the association type. Six types can be found: *Access*, *Invocation*, *CastTo*, *Inheritance*, *SubTyping* and *CheckInstanceOf*.

Only three association types correspond exactly: Access, Invocation and Inheritance.

Table 2

Associations in Evolizer 2007

Evolizer 2005	Evolizer 2007	Used by EvoSpaces	Famix 2.0
contains	*Parent_id* column in *famixentity* + multiple "join" tables	**Yes**	No
invoke	Invocation	**Yes**	Yes
inherits	Inheritance	**Yes**	Yes
declares	n.a.	Yes	No
accesses	Access	**Yes**	Yes
hasTypes	n.a.	Yes	No
aggregates	n.a.	Yes	No
overrides	n.a.	Yes	No
includes	n.a.	Yes	No
n.a.	CastTo	No	No
n.a.	SubTyping	No	No
n.a.	CheckInstanceOf	No	No

4.4.4.1 Restoring Invocation, Inheritance and Access associations

For the three most important association types, *Invocation*, *Inheritance* and *Access*, porting was straightforward as it was just a matter of modifying the simple SQL queries inside loader methods.

We choose to ignore the three new association types *CastTo*, *SubTyping* and *CheckInstanceOf* because we lacked information on them and on how they were generated and because we doubted their real usefulness.

4.4.4.2 Do we need the other associations?

The *declares*, *aggregates*, *overrides* and *includes* association were used in EvoSpaces merely as information to display when requested. We decided we could safely spare ourselves the effort of regenerating them, as their added value is not clear to us.

However, the *contains* association was a different matter entirely. It is used at the ModelView level to determine the layout and the content of the buildings. Because of this, we had to either replace it with similar information or rewrite substantial portions of the ModelView layer.

4.4.4.3 Understanding contains associations

The *contains* association is special. It represents a generic view of the link between a package and its sub-packages, between a class and its methods, between a local variable and its method, and so on. In essence, it is a human concept. Quite unsurprisingly, it is not defined in the Famix standard as there is no single way of defining what it includes and what it doesn't. The added value of *contains* is that it offers a hierarchical graph representation of the sources that is not necessarily a strict projection of the application's code graph.

When we look at other similar associations we find that *declares* seems to be the collection of all declarations. As such, it is either equivalent to or a subset of *contains*. *aggregates* and *overrides* are obviously derived data.

The fine print is that all of these association types actually represent data that derives from what should have been entity attributes from the beginning. In the old database, it was used in place of proper entity attributes, such as *belongsToClass* for a method or attribute entity.

With the old schema, EvoSpaces had to use the *contains* association as it provided necessary information that was not found elsewhere.

With the new schema, this is not necessary. However, the *contains* association has a strong advantage: it makes traversal of the code graph from top to bottom very easy.

4.4.4.4 Regenerating the contains association

The only way to proceed was to continue loading *contains* associations in entities, because DBAccess must stay fully compatible with the rest of EvoSpaces. Either we generated it on the fly with the RelationBuilder or we regenerated it with our dbPrep tool and stored its result in the *association* table before running EvoSpaces.

We chose the latter, as we had to run dbPrep anyway, and there is no point wasting CPU cycles at every start-up of EvoSpaces. Moreover, we soon discovered that we had to generate a unique key for each *contains* association record and make sure it was unique throughout all associations. As a direct access of the data is always possible through DataRetriever, we had no choice but to generate the *contains* associations before EvoSpaces runs.

4.4.4.5 Determining what contains associations should include

At first sight, the *contains* association seems to have a direct equivalent in the new database schema: the column *parent_id* in table *famixentity*. We went on eagerly to

generate it based on this information so we could have a first look at PasEvi in EvoSpaces' display.

The packages and buildings looked ok, but nothing was displayed inside them when we opened them. We realised that we had to go through a more careful examination of the *contains* association as it was in the old database in order to understand what was lacking in our new version.

Each *contains* association entry can be viewed as a pair of entities, one being the parent, the other being the child. Cardinalities between entities are determined by three factors:

- The programming language used.
- The structure of the sources, for example if more than one non-internal class are defined in a source code file.
- The choices made by the generator to include or not include some information, for example, information about files containing internal classes.

The *contains* association of the old database:

Table 3

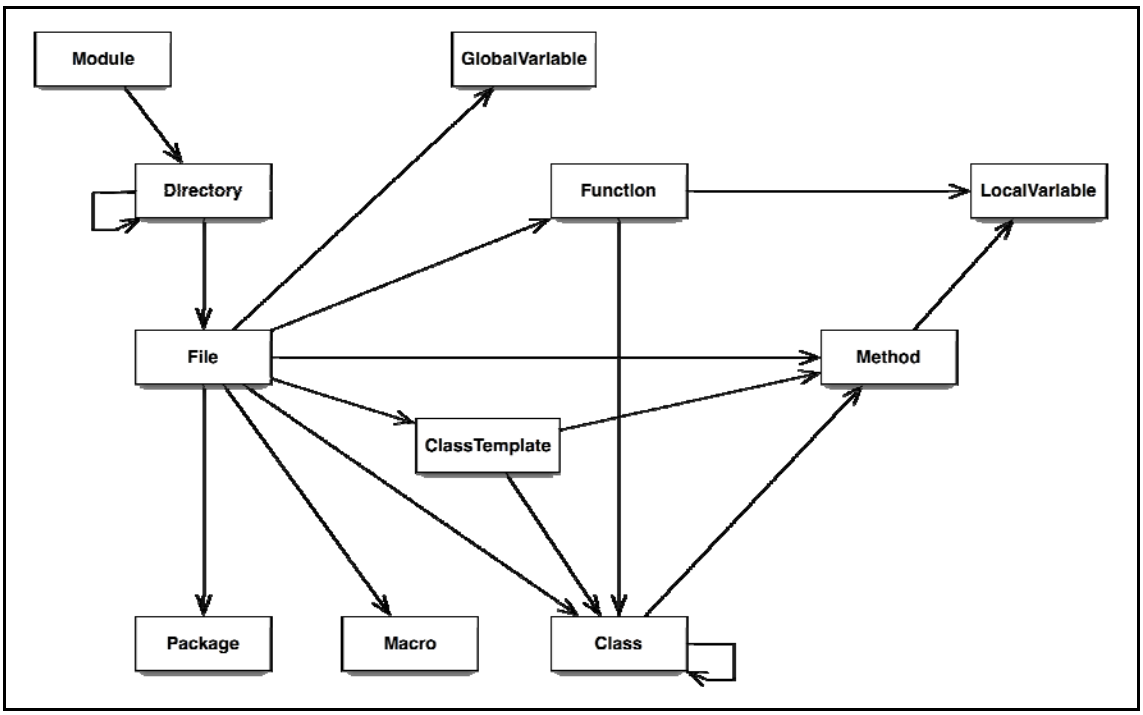
Evolizer 2005 *contains* pairs

From	To	Number of occurrence
method	localvariable	59015
class	method	14688
file	method	13159
file	macro	11042
function	localvariable	8118
file	class	2446
file	globalvariable	2022
directory	file	1968
file	function	1717
directory	directory	364
classtemplate	method	176
class	class	167
file	classtemplate	22
module	directory	18
classtemplate	class	3
function	class	1
file	package	1

This table translates to the following diagram:

Figure 12

Evolizer 2005 *contains* association



The contains association of the old database is fairly complex due to the large number of entity types.

We see that the *contains* associations define a directed graph spanning all the code entities. This is important because EvoSpaces' ModelView layer starts from the topmost directory and lays out directories and subdirectories as nested colour squares on the grounds (figuring packages) and files as building. Then the ModelView layer reaches further down into the graph to find and position attributes, methods, local variables, built-in classes, and so on, until it reaches the end of each leaf of the graph.

We said we tried to generate new contains association out of the information stored in the *parent*_id column of table *famixentity*. Here is what it looks like:

Table 4

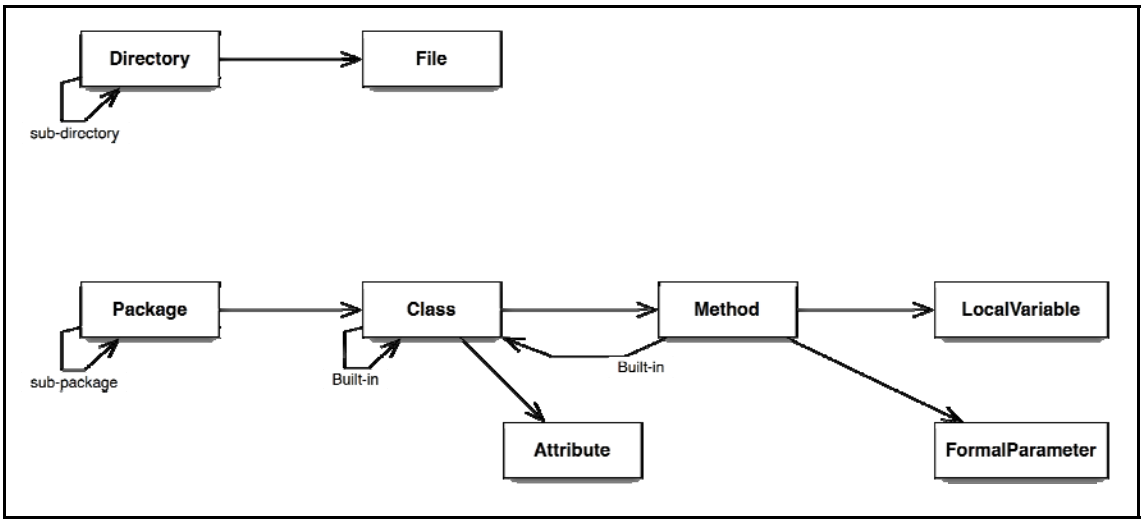
Evolizer 2007 *contains* pairs generated from *famixentity.parent_id*

From	To	Number of occurrence
Class	Method	6593
Method	LocalVariable	6566
Method	FormalParameter	4954
Class	Attribute	4505
FamixDirectory	FamixFile	605
Package	Class	605
FamixDirectory	FamixDirectory	176
Package	Package	175
Method	Class	77
Class	Class	42

Which translates to the following diagram:

Figure 13

Evolizer 2007 *contains* pairs generated from *famixentity.parent_id*



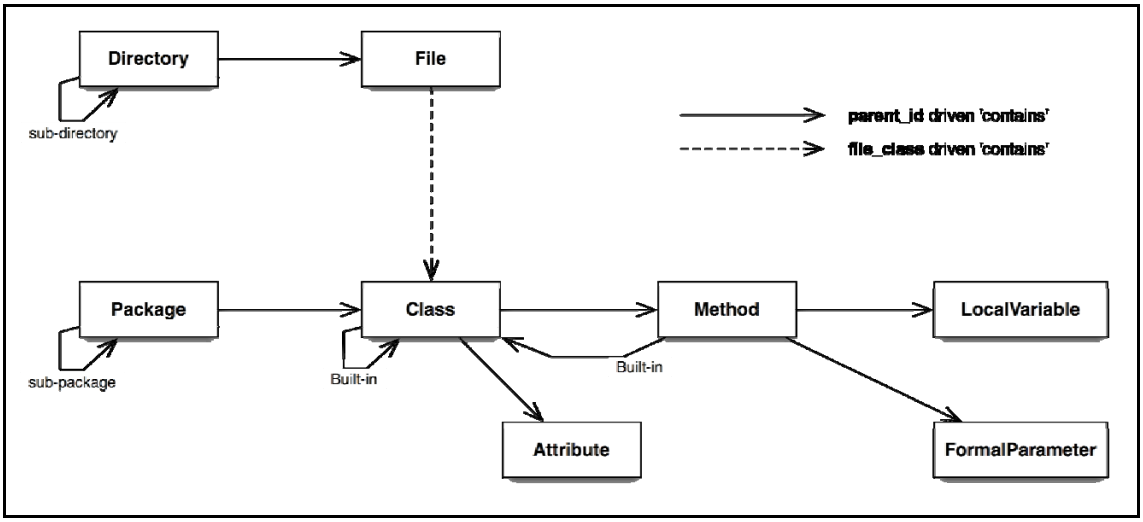
It is immediately obvious that the *contains* associations we got represented a fragmented graph. Starting from the topmost directory, it is not possible to reach all entities.

If we compare with the old diagram, we see that we need at least a pair linking file entities to class entities. This is what we decided to add to our generated contains association.

We found the necessary information in the *file_class* join table. In particular, we checked that it didn't include links from files to built-in classes. After doing this, we had the following situation:

Figure 14

Evolizer 2007 *contains* pairs from famixentity.parent_id and *file_class*



The display the interior of building worked. However, it didn't produce the expected result: all classes were displayed as built-in classes with a puck shape, because we had forgotten that buildings represent files and not classes.

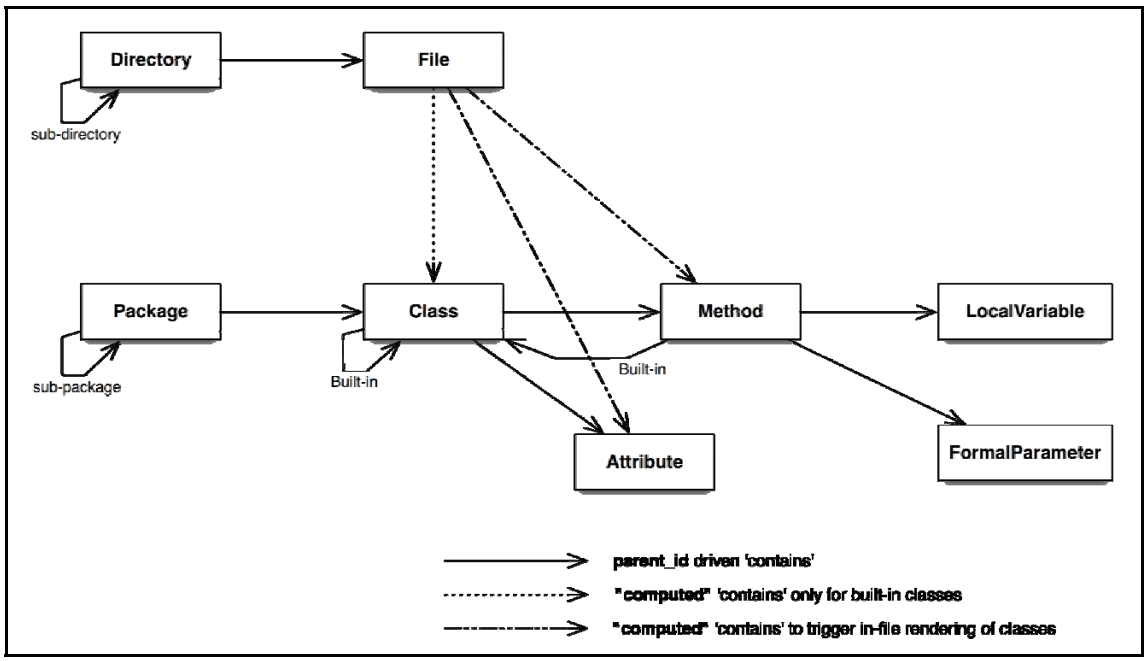
Ideally, we should be able to switch between a file centric and a class centric display. However, only the former exists at this time, and we didn't have the time to implement the latter.

As a workaround, we realised we had some leverage on the display through the *contains* association itself. By carefully adjusting its content, we actually succeeded in displaying file and classes as merged entities. That is, the content of the class defined in a file is displayed as belonging to the file itself, and the class is not displayed.

The following diagram show the *contains* association after this "hack":

Figure 15

« Hacked » Evolizer 2007 *contains* pairs



By adding a links connecting file entities to method and attribute entities, and suppressing the link between file and class entities, the desired result was obtained.

The only drawback with this approach is that built-in classes that are defined outside of any method were ignored. By querying the database we learned that there are 42 such cases. As a consequence, we had to link file entities to built-in class entities. This done, built-in classes were now displayed correctly.

EvoSpaces was now properly displaying PasEvi's source code.

4.4.5 Adapting the query layer

The query layer is implemented by the class DataRetriever. This class provides all kinds of helper methods that shield the database from direct access.

The first task in adapting this layer was to distribute the methods between specific DataRetriever classes and the CommonDataRetriever class. We found many methods did not require adapting.

Then we went on translating old SQL queries into new ones yielding the same resultset. After that, we were done adapting the DBAccess layer.

4.5 Adapting the ModelView layer to the new data

As we know, the ModelView layer contains all the parameters necessary to render entities in the 3D view. Besides this, the ModelView layer contains all the classes that do the actual job of generating these parameters.

After porting the builders, there were still some issues with the ModelView layer which we had to fix before things worked.

However, as this is very technical and strictly related to EvoSpaces' internals, readers not involved in the development of EvoSpaces can skip this chapter entirely without missing anything.

Before exposing these bugs, it is necessary to introduce the interior of this layer.

4.5.1 ModelView layer: the big picture

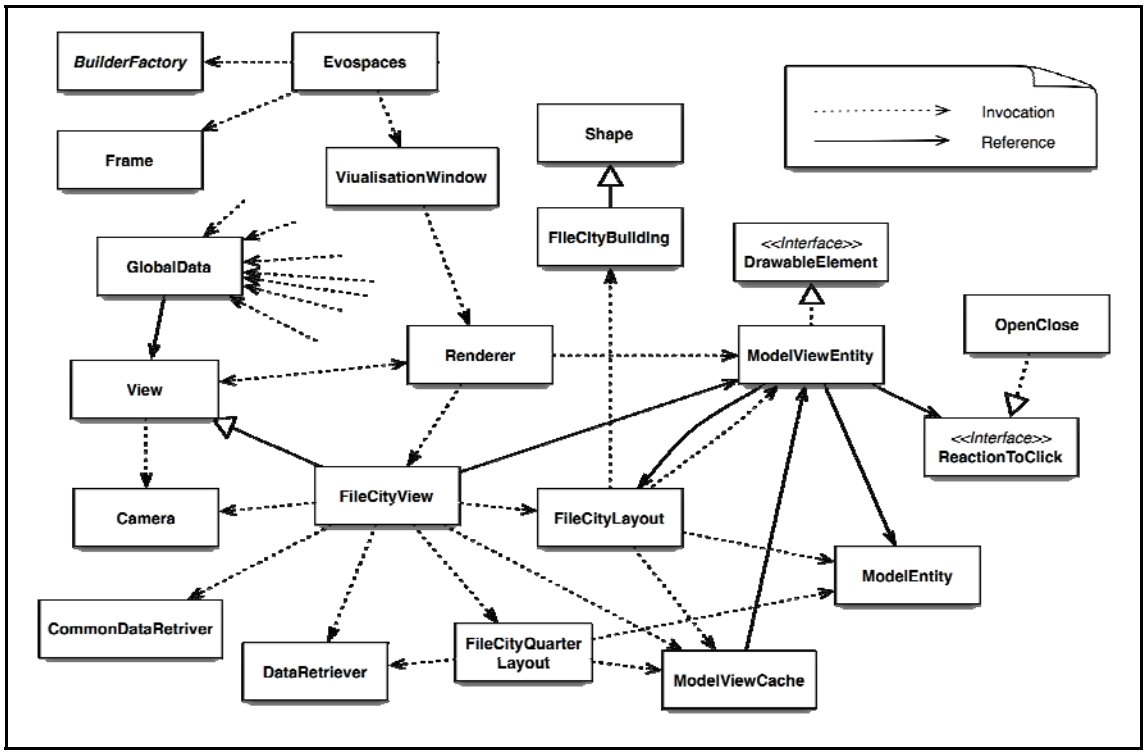
The ModelView layer is quite complex so we do not provide a detailed explanation of its functionality here. We cover the minimum necessary to understand the information relating to it.

A good way to understand this layer is to view it in the context of EvoSpaces' start-up.

The following figure shows the object model of all classes taking part in the start-up sequence of EvoSpaces. That is, all classes involved from the moment the program is launched until the view is displayed and the program is ready to process user commands.

Figure 16

EvoSpaces' object model involved in initialisation



Notice the extensive use of the Singleton design pattern [GoF, 1994]. Most of the ModelView layer classes are singletons, as are those of the DBAccess layer.

Singletons are clean and practical, as singleton instances can be called from anywhere in the program without maintaining a direct reference to them. But they can make it hard to understand the actual structure of the program. Here, we based the diagram not only on references, but also on invocations during the program's start-up.

Now let's see what how these classes work together.

4.5.2 EvoSpaces start-up sequence

In many ways, understanding EvoSpaces' start-up sequence is like understanding the whole program. Here is a quick summary of the start-up sequence[12].

Execution starts in the EvoSpaces class (shown on top of the above diagram). It reads the command line parameters and chooses the database configuration accordingly to

[12] For more information on the start-up sequence, we provide sequence diagrams in the annexe.

user's choice and instantiates the corresponding BuilderFactory. It then opens the connection to the database engine and finally setup the VisualisationWindow.

The VisualisationWindow is the application's main window. Once it is setup, it initialises the Renderer.

The Renderer sets-up the OpenGL context for the 3D display. It then calls the default View instance, in our case FileCityView.

FileCityView first initialises the Camera and the ModelViewCache. Then it gets the IDs of all file entities from DataRetriever and then requests the ModelViewCache to load all these ModelViewEntitys. Then it asks FileCityLayout to prepare them for layout.

FileCityLayout instanciates a subclass of FileCityBuilding for each ModelViewEntity. Each FileCityBuilding instance generates all necessary parameters for rendering the building in the 3D view, like wall position, dimension and texture. When this is done, execution is returned to FileCityView.

FileCityView then calls FileCityQuarterLayout. FileCityQuarterLayout is responsible to position buildings respective to each other on the map. It position directory entities, which are represented as concentric squares on the ground to reflect their hierarchy, and place each building inside its own directory. Lastly, it sets-up the camera at one corner of the city. It then returns execution to FileCityView which returns it to the Renderer.

The Renderer then adds itself as listener for events triggered inside the OpenGL view and does a few more set-up operations before giving execution back to the OpenGL context. EvoSpaces is now ready to respond to user input.

4.5.3 Handling default metric selection at start-up

We found that the two metrics used by default in determining height and look of buildings was hard-coded in classes FileCityLayout and FileCityViewMetricChooser. We created two variables in the application-wide settings' class, GlobalData, to store these metrics.

Proper values are determined at start-up in the EvoSpaces class, along with the database and BuilderFactory.

4.5.4 Providing filesystem root to FileCityQuarterLayout

FileCityQuarterLayout needs to traverse the directory hierarchy from the root node to each leaf. The problem is: it has to know the ID of that root node somehow.

In fact, we found it was just hardcoded in FileCityQuarterLayout.

We choose to create a new getRoot() method in DataRetriever to provide the root node to FileCityQuarterLayout. This way, the correct root node can be provided regardless of the database in use.

4.5.5 Handling special files in FileCityLayout

As EvoSpaces was developed with a code database containing the source of a C/C++ application, it had to deal with two different kinds of source files: .c or .cpp normal code files and .h header files.

Header files are specification files that describe the functions contained in the corresponding C/C++ library. They are used by the linker when including a pre-compiled library into a C/C++ program. There are many of these in Mozilla.

EvoSpaces applies a special treatment to header files. They are displayed with a “town hall” texture.

File type is determined in FileCityLayout before attributing building types to files. However, the conditional statement treated header files as the general case and .cpp files as the special case, preventing the proper display of PasEvi‘s java source files. The fix consisted in making things correspond to reality, namely, treating header files as a special case and all others as the general case.

4.5.6 Finding and fixing a cache poisoning bug

Once we had completed our modifications and corrected a few bugs we had introduced ourselves, EvoSpaces was working properly. We could navigate the city, open and close buildings and load associations.

But invariably, after some time, the application would crash on a ClassCastException involving a cast attempt on a ModelViewRelation (an association at the ModelView level) to a ModelViewEntity:

```
Exception in thread "AWT-EventQueue-0" java.lang.ClassCastException:
  modelView.relations.MVAccess cannot be cast to modelView.ModelViewEntity
        at modelView.manageableViewElements.layouts
                  .FloredBuildingLayout.layout(FloredBuildingLayout.java:17)
        at modelView.FileCityView.loadSubEntities(FileCityView.java:785)
```

Because of this, we had to analyse and understand the whole layout and rendering process. It took quite a while to track the bug back to its source, as it is actually very far from the classes where the exception takes places.

4.5.6.1 Tracing the problem back to its root

The exception is raised when a building has been opened and the FloredBuildingLayout.layout() method has been called. The method is responsible to display all the file's sub-entities on floors inside the building. At some point, it asks the ModelViewEntity representing the file to load all its sub entities associations and provide them in the form of a linked list:

```
LinkedList elements = element.getContainsChildEntities();
```

Later, it iterates over the list of entities to work on each one of the separately:

```
ModelViewEntity tmp = (ModelViewEntity) elements.get(i);
```

This is where the cast fails and exception is raised. In place of a ModelViewEntity, we receive a ModelViewRelation. Why?

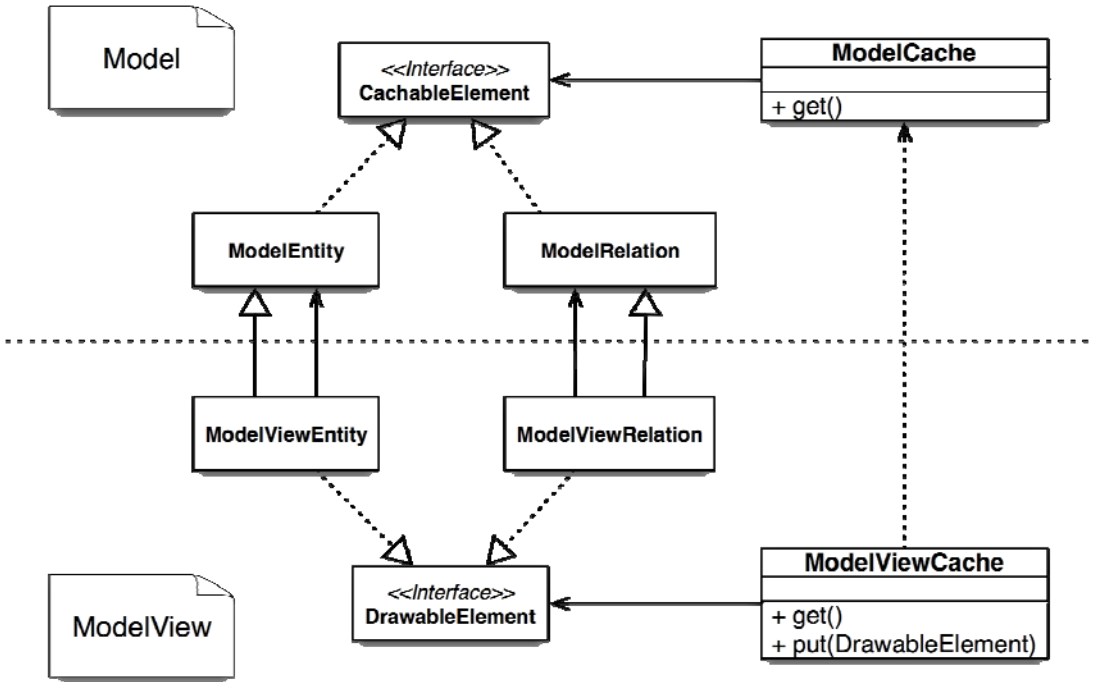
The getContainsChildEntities() method of the class ModelViewEntity loops over all *contains* associations of the entity. For each, it calls ModelViewCache.get() to obtain the corresponding ModelViewEntity. This done, it returns them as a linked list. We see that the entities all come straight from the cache. So if we get a ModelViewRelation, it must have been in the cache already. To understand how this happens, we had to look into the EvoSpaces cache mechanism.

4.5.6.2 EvoSpaces caching infrastructure

The caching infrastructure is best described by the following diagram:

Figure 17

EvoSpaces' caching infrastructure



The motivation behind caching is to have a fast intermediary mediating the access to a slow data source. The intermediary keeps the data it loaded for future access according to a policy.

From the diagram above, we see that the caching model of EvoSpaces is fairly complex. Each layer has its own cache.

The ModelCache accesses the database through the BDAccess and caches elements conforming to the CachableElement interface. As we see on the diagram, this means cached elements can be either ModelEntity or ModelRelations.

The ModelCache get() method takes an integer ID as a parameter and returns the corresponding element. Under the hood, it checks if it is already in cache, and if not, it uses the ID to fetch it from the database. This has an important implication: entities and relations cannot share ID keys in the database.

Surprisingly, the actual code of the get() method does not allow the loading of ModelRelation instances. Only ModelEntity can be loaded, so in fact, the ModelCache is a ModelEntity cache only. As a side effect, this lifts the above restriction on the database.

The ModelViewCache keeps elements conforming to the DrawableElement interface. Its get() method works similarly to the one of the ModelCache, except that rather than fetching the object from the database, it calls the ModelCache get() method to obtain the ModelEntity or ModelRelation corresponding to the ID and derives the appropriate ModelView element from it.

It has to be noted that due to the above discovery about the ModelCache only loading entities, any attempt to get a ModelViewRelation instance from the ModelViewCache will lead to an error.

However, we were really baffled to find a public put() method in the ModelViewCache class. Its code is as follows:

```
public synchronized void put(DrawableElement drawableElement) {
    cache.put(new Integer(drawableElement.getUniqueKey()),
                                                drawableElement);
}
```

This method allows manual insertion of elements into the cache! A cache should never expos such a method. It should never be necessary as a cache should always have sole responsibility for loading the elements it stores.

As it turns out, this was our culprit: a method in FileCityView was calling put() to add a ModelViewRelation to the ModelViewCache, provoking an ID collision. Later on, when that ID is requested, the relation is returned where an entity is expected, provoking the ClassCastException we saw above. Disabling that call to put() solved the problem.

This cache poisoning bug reveals a very important fact about the new database: ID keys are not unique across entity and associations, so the cache's design needs to be revised to be brought to its full potential. Currently, ModelRelations are simply not cached.

4.6 Loading execution traces

An execution trace is a list of sequential method calls that were recorded during the normal use of the program, for example, during the execution of a particular use case.

As we saw in the guided tour, EvoSpaces has a special “night mode” to display execution traces.

Traces are divided into segments. All files involved in a segment are displayed in colour according to their frequency of use in the segment. Additionally, it is also possible to watch the trace “execute” call by call inside a given segment. A ray of light is then drawn from building to building.

4.6.1 Execution traces support in EvoSpaces

EvoSpaces’ support for execution traces is quite complete. Loading traces did not involve modifications of the program itself.

Traces are stored in the table *exampletrace*. Several traces can be stored at once, each being identified by a number. Another table, *infotrace*, stores meta-information about each trace.

Each call contains the following information: a sequence number, a filename, a method signature and a depth of call value.

The sequence number is the ordinal place of the call in the trace, counting from the beginning.

The filename and method signature are identifying the method being called.

The depth is the level of nesting of a given call. It is used to determine whether a call is returning to its caller method (depth decreases by one), a part of a sequence of call (depth stays the same) or a method call inside its caller method (depth increases by one).

The table *exempletrace* stores data about the file and method signature as reference to the corresponding entities in the database. This means that to load a new trace, we have to match each filename and the method signature to the corresponding entity ID in the database.

4.6.2 Execution trace format

The format of the trace depends on the instrumentor used to trace execution. The instrumentor used on PasEvi was developed by Sébastien Jossi under the supervision of Philippe Dugerdil. For each call, it gives the file name, the method signature used and the return type and various other information items. A line of trace looks like this[13]:

```
EviGcRefCdProjet.java getReferenceOd(String)) As GenOdStandard
ProjetDetail.CodeTypeProjet, null, null, …
```

The extra closing parenthesis is an artefact of the instrumentor and can be safely ignored. Each parameter and return value type is listed exactly as it was found in the source code. For each call line, there is an end-of-call line indicating when the method is about to return. An example of such a line:

```
END EviUcProjetDetail.java validerChangement(EviOdProjet))
```

However, there is no guarantee that each call will be matched by a corresponding end-of-call. There can be extra end-of-call lines to calls that were never listed if the tracing was begun after the application started. There can also be end-of-call lines missing because the tracing was stopped before the application terminated.

Raw trace lines do not include information about the depth of call. This information is necessary for meaningful display of traces in EvoSpaces. The raw traces must be processed in order to determine depth and add it to each line.

Philippe Dugerdil has developed a tool to do this. It takes a raw trace as input and outputs the same trace with depth information. A few lines look like this[14]:

```
581 EviOdLocalisation.java
setSroRef(pas.evi.localisation.od.EviOdSourceOrigine)
579 EviOdFicheOrigine.java setLocalisations(GenOdListLien)
578 EviOdFicheDescriptive.java setOrigines(GenOdListLien)
578 EviOmFicheDescriptive.java chargerProjet(GenOdStandard)
579 EviOdFicheDescriptive.java getPrjRef()
```

Our task was to load these processed traces into EvoSpaces. As we saw, we had to match the filename and method signature to entity IDs in the database.

4.6.3 TraceLoader, a tool for loading traces

As with metric generation, we created a tool for loading traces in the database that we added to our EvoTools package.

---

[13] A sample of raw execution trace is available in the annexe.

[14] A longer sample of processed execution trace is available in the annexe.

4.6.3.1 Architecture

TraceLoader is a command line tool that uses standard unix output lines to write its results. It reads a trace line after line specified as a parameter, does its work, and writes each result line on the standard output (*stdout*). In parallel, it writes status and debugging information on the standard error (*stderr*).

Each line of output is a SQL insert command for loading a single line of trace into the table *exampletrace*. The last line is the insertion command to add the meta information about the trace in the table *infotrace*.

The standard usage is to redirect the standard output to a file, then to load the file into the database. The tool doesn't write directly to the database for performance reasons, as this can considerably slow the processing of large traces which can easily contain millions of calls.

However, the tool doesn't write directly to a file so to make it possible to send the output directly to a database.

This makes TraceLoader a little tricky to use for person not used to the command line, and we provide various example of command lines use in the annexe.

TraceLoader has two modes of operation: raw and standard. The raw mode takes a raw trace as input file and its calls will lack depth information. This is of limited use and was implemented only for the case where Philippe Dugerdil's trace processor is not available. The standard mode is the normal mode of operation and it reads a processed trace complete with depth information.

4.6.3.2 Matching file and methods

The challenge behind the loading of execution traces is matching the methods signature to the corresponding entity in the database. This task is complicated by two facts:

1. The information we get is incomplete and not totally unambiguous.
2. There are severe performance constraints.

The first one forced us to establish strategies to reduce as much as possible the likeliness to make confusions. The second one forced us to code in a very linear style and to depart a little from good practices in structuring our code. It also forced us to take care of not doing unnecessary work.

Our tool processes traces line-by-line. It asks the database for all the methods having the same method name (the part of the signature before the parameters, like the 'get' of the method get()) and pertaining to the same file.

If there is only one, the match is considered successful and the corresponding file and method entities Ids are used.

If there is more than one, a specific strategy needs to be used. These cases are presented below.

Before talking about the special cases, we must present the database preparation required to obtain acceptable performance.

4.6.3.3 Preparing the database for trace loading

To be able to match the name part of method signature easily, we decided to add a column to the *famixentity* table and add a simple name-shortening task in our dbPrep tool. This way, method names are prepared once and are stored into that column before anything else is done.

Moreover, the SQL query that gives us the list of methods matching a given name, complete with all the data necessary to treat the special cases, is fairly complex and ran too slow for a program that treats hundreds of thousands of items.

For this reason, we decided to create a de-normalised table[15] loaded with the result of this SQL query for all methods. The user must just run a special SQL query creating this table once before running the TraceLoader tool. This query can be found at the beginning of the TraceLoader.java file.

Then, for each call of the trace the tool only has to run a very simple and fast SQL query on a single table.

4.6.3.4 Dealing with problematic cases

Our call data only includes filenames and method signatures. This isn't enough to prevent all risk of ambiguity. In java in particular, there are cases where we could find more than one method with the same name the same file:

[15] This table is de-normalized in the same way as found in data-warehouse storage. For each entity of interest, all the information describing it is replicated in the same record. This make accessing all information on this entity very fast.

1. There is more that one class defined in the same file and they share the exact same method signature (name and parameters).
2. There is more that one class defined in the same file and they share only the method name.
3. There could be more than one file with the same name in a different directory. It is plausible that they contain method with the same name or signature.
4. Any combination of the two above.

The first case can happen only when there are more than class defined in the file. In this scenario, there is no way to determine which method has been actually called.

The second case is the most common. It happen when method overloading is used inside a class. Method overloading refers to having several methods defined with the same name but with different parameters. In this case, parameters have to be compared one by one to determine which method was actually called.

The third case happens twice in PasEvi but the impact is not important.

4.6.3.5 Matching method parameters

The TraceLoader tool matches method parameters. When it encounters several possibilities for a given method, it iterates over them, loads the parameter information from the database and compares each one until it finds a complete match.

Parameter entities are stored in table *famixentity* and include a *declaredClassId* column pointing to another entity in the same table. Evolizer stores every type it has encountered as an entity in the *famixentity* table. Built-in types such as int or long are stored as a “Class” entity.

For each class, it determines the fully qualified name by adding the complete package information, such as pas.evi.co.acteur.EviCoActeur. This is also the case for classes that originate in system libraries such as java.lang.String.

This is an important point, and a strong one in favour of Evolizer, because without this precision, it would be impossible to match types with any security: in the code, many classes are often qualified with abridged name which are reproduced as is by the instrumentor. The instrumentor, in its current implementation, does not parse “import” declarations that indicate the package to which the class belongs, requiring matching possibly abridged class names against complete classnames in the database. There is

still the possibility of confusion arising when different packages have classes with the same name. We identify these conflicts by querying the database. [REMPLIR …]

However, there are still a few problems:

- Arrays of built-in types are surprisingly recorded as <Array> in *famixentity*. This leads to a loss of information and possible trouble in case of method overloading such as doTask(int[]){} doTask(long[]){}, which is a plausible scenario. Fortunately, this does not arise in PasEvi. We can simply match any array to <Array>.
- When Evolizer cannot determine the fully qualified name of a class, it stores it as <undef>.ClassName. This can happen when external packages are missing when the source code is analysed. In this case, a class can exist twice in the famixentity table: one with package <undef>, and one with the fully qualified name[16]. This happened once in our database, without consequences.

Small bugs in the instrumentor caused the few other problems we encountered. They were easily located and it was possible to find a solution for all of them without losing any information.

[16] When the class has been referred to in the code both with and without its fully qualified name.

5. Using EvoSpaces with PasEvi

Now that we have successfully loaded PasEvi in EvoSpaces, it is time to try to use it. In this chapter, we will begin to explore what how EvoSpaces can help developers and maintainers.

We will first have a look at how a software maintainer is operating when confronted to an entirely new program and the typical question he asks.

We will then try to answer these questions using EvoSpaces. Then, we will use this experience to hint at possible use scenarios and suggest improvements and future developments.

5.1 The maintainer confronted to a new application

Discovering a new industrial-grade software system is very challenging for any software maintainer. He must understand it well enough to make modifications to it, yet he has usually little preliminary knowledge on it and only a limited time at disposal.

The maintainer will enter a learning process that will let him build a gradually richer mental representation of the program.

It is our vision that this process is highly iterative, characterised by a succession of questions and discoveries. The human is the real driver behind the process, and the questions he asks depends on what he wants to know. To find answers to his questions, the human navigates the program he is discovering. This navigation is crucial to the whole process.

There are many paths to mastering a program, and this mastery is always relative to a goal. Actually, maintaining a program in the long run implies many run of this process. The first few are intended to building a general knowledge of the program, the next ones are intended to learn enough to successfully intervene in the code.

Upon discovering a program for the first time, a maintainer will seek to build himself a basic, schematic view of the program's architecture. This schematic will serve as a framework on which an ever-richer representation will be constructed.

It is important to note that the background and prior knowledge of the maintainer has a strong influence on the process. It determines with what mental representation he will start, what questions he will ask and how he will interpret his discoveries.

We isolated the following key elements:

- A-priori knowledge
- Goals
- Questions
- Schematic representation that serve as a mental framework
- Navigation

To help the learning process, an application such as EvoSpaces should provide the schematic representation and the navigation, along with access to the information necessary to answer the questions.

5.2 EvoSpaces and the learning process

The aim of EvoSpaces is to boost the learning process of an application maintainer. To do so, it provides the three elements exposed above.

5.2.1 Schematic representation

The city view proposes a schematic representation that intentionally makes use the very developed human capability of knowing his position and navigating in an urbanised environment. This way, the maintainer is relieved from conscientiously figuring out “were he is”, as he must do when navigating source files in a hierarchical file system.

In order for this to work well, the display must follow a rule: “The display must never work against human intuition”.

Another rule that we can derive from the necessity of the representation being schematic is that the display should not be too cluttered. Important information should not be hidden by less important details. We can define the rule as: “Information should be displayed conservatively”.

5.2.2 Navigation

Intuitive and easy navigation is very important, because the learning process implies moving back and forth between detailed view of specific elements and more global views highlighting program-wide features.

The speed of the navigation is also important. We believe that there is a threshold in speed under which a user gives up. If it is too slow and tiring, the user will use other means or simply do without the information he was seeking.

So this can be thought as another rule: “The information needed should always be very fast to reach”.

5.2.3 Elements of answer

Two forms of elements of answer can be found in EvoSpaces:

- Structural information carried by the layout of the buildings, their content such as methods and the pipes connecting them to represent associations such as invocation.
- Software measurements on elements being displayed according to metrics.

The first one is part of the schematic representation and is static information.

The second ones are the foundation of real answers to the maintainer’s questions. Metrics can be all sorts of thing, such as discrete values on some characteristic or synthetic information about elements.

The strong point of EvoSpaces is to use any metric to set the aspect and dimensions of buildings, and to allow looking at the resulting city from far as well from close. The user can adjust his perspective on the city, letting him look at the same information from many viewpoints.

However, to be useful “metrics must be relevant to the questions asked”

5.3 Testing Evospaces

To test EvoSpaces in real-world conditions, we start with our a-priori knowledge. It includes our background in system engineering and very little knowledge of PasEvi. We defined the following maintainer goal: *to obtain general knowledge on the architecture of PasEvi*.

We are going to ask questions relating to this goal and see how EvoSpaces performs at answering them.

In the process, we also check whether EvoSpaces conforms to the rules we defined above. These rules were:

1. “The display must never work against human intuition”
2. “Information should be displayed conservatively”
3. “The information needed should always be very fast to reach”
4. “Metrics must be relevant to the questions asked”

5.3.1 First run

One of the first questions that could come when discovering a new program is where code is concentrated. What are the places in which a lot of “activity” takes place. To answer this question, we loaded two very common metrics in EvoSpaces’ metric mapper:

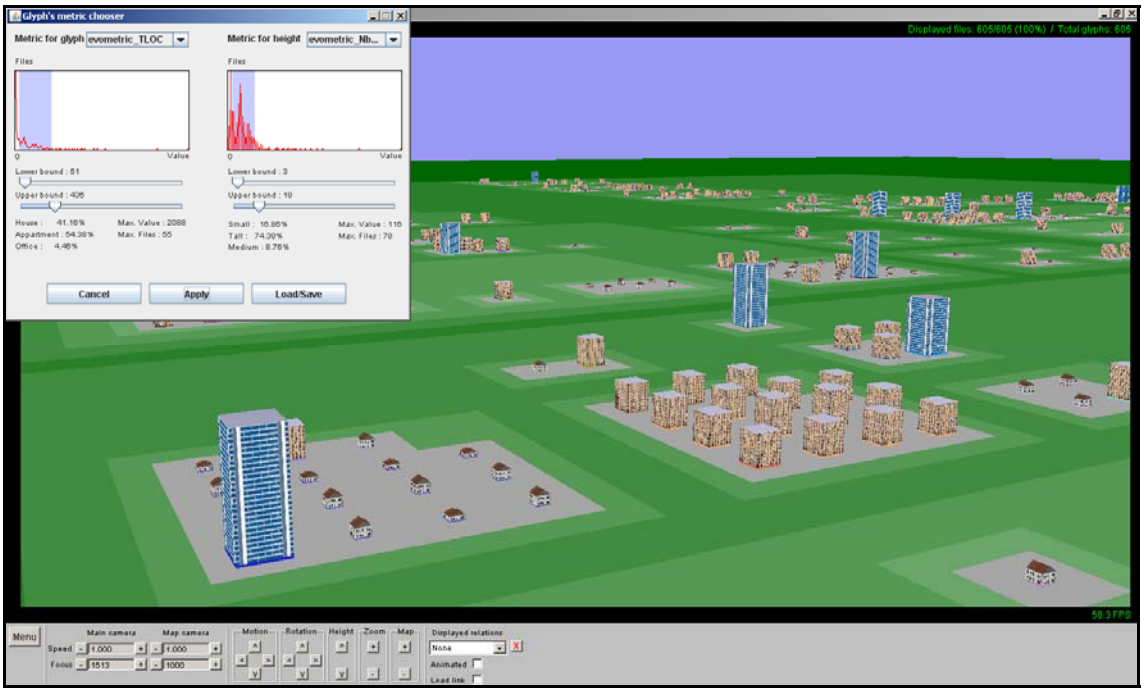
- We assigned the number of lines of code (TLOC) to metric1 (Glyph), which determines wether the building will look like a house, a “block” or an office tower.
- We assigned the number of methods (nbMethods) to metric2 (height), which determine the height of the buildings.

These are metrics which answers simple questions yet allows to quickly obtain an overall idea of a program.

The result can be seen in the following screenshot.

Figure 18

“First run”



Files with many lines of code are immediately visible as office tower and very small one as houses. We get an immediate feeling of how packages are organised. Here for example, the package one the left has one big file with many small ones, probably serving as helpers for specialised processing.

If we would like more info on one of these files, we can right-click on it to open the contextual menu and select “show information” to open an information panel.

5.3.2 Adding immediate information feed-back

However, using a contextual menu quickly becomes tedious, because we tend to be interested in many files. This contradicts our rule number 3, which states: “the information needed should always be very fast to reach”. We would like basic information to display when we click on the building.

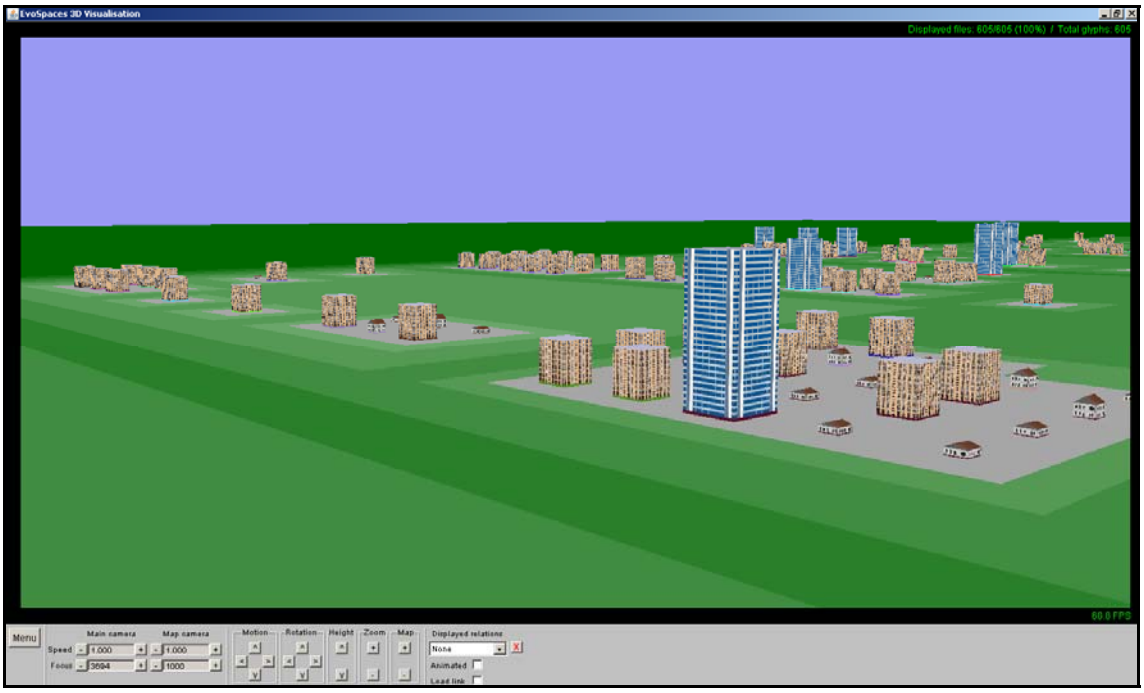
We implemented this feature as a single line of information displayed in the black line immediately under the display. The information includes the full path of the file, the file name and both metrics with their value for the file (examples will come below).

5.3.3 Reinforcing intuition: replacing texture

As we were exploring the PasEvi city, we found that the “block” buildings were especially hard to tell apart from a distance. Because their façade contains many windows, they sort of merge together. This is more obvious on the next screenshot.

Figure 19

Texture for block buildings has too many windows



We replaced the texture with another one we found on the Internet[17] and cropped to keep the number of windows low. The result can be seen further down.

5.3.4 Where is metric2?

Until now, we have concentrated on one metric, the number of lines of code, and where satisfied with the result. However, we have a problem appreciating the metric2, which is set to the number of methods.

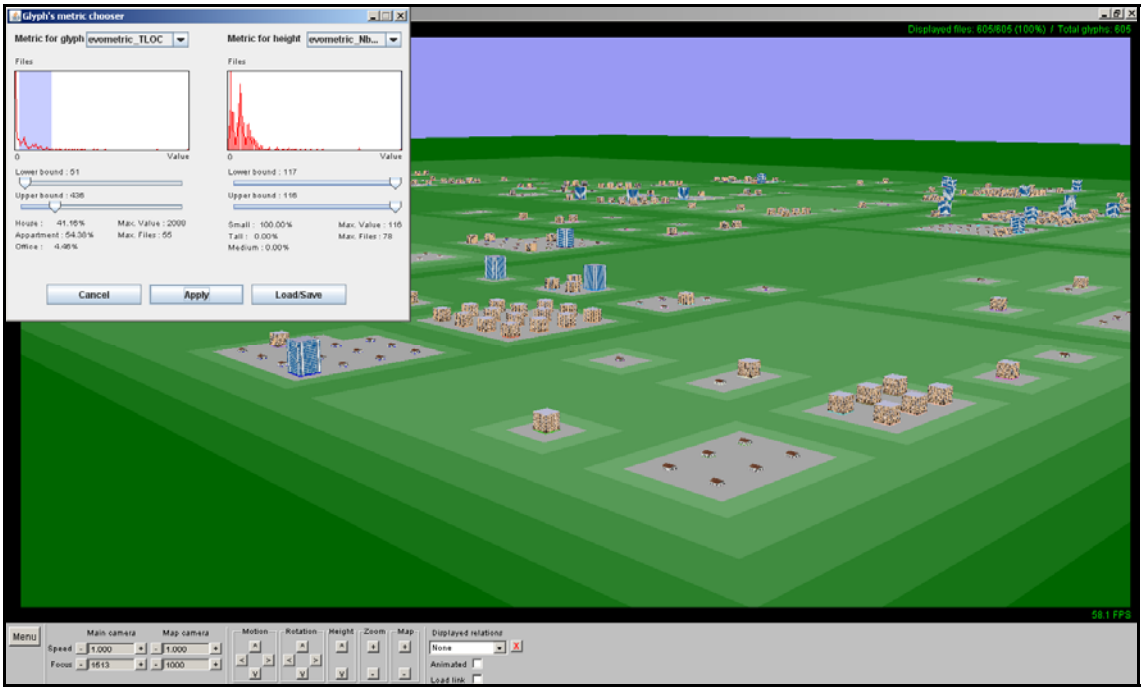
At first, we thought that tall buildings indicated many methods. But after checking, it is not the case. Why? Shouldn’t the height metric have a direct connection to building height? Where has it gone?

[17] http://www.france-textures.fr.st

To find out, we decided to play with the metric2. We first set its thresholds to the maximum value, so that there would be only small buildings on display. This is how it looks like:

Figure 20

Height metric set to all buildings small



It is obvious now that the relative height of building is not determined by the metric2. On the contrary, the building type is what really determines the perceived height of a building. The metric2 only determines whether a bigger or smaller version of a given building type is displayed.

In our opinion, this is a clear violation of rule number one: "The display must never work against human intuition". In particular, this is a violation of the "representation condition", which states:

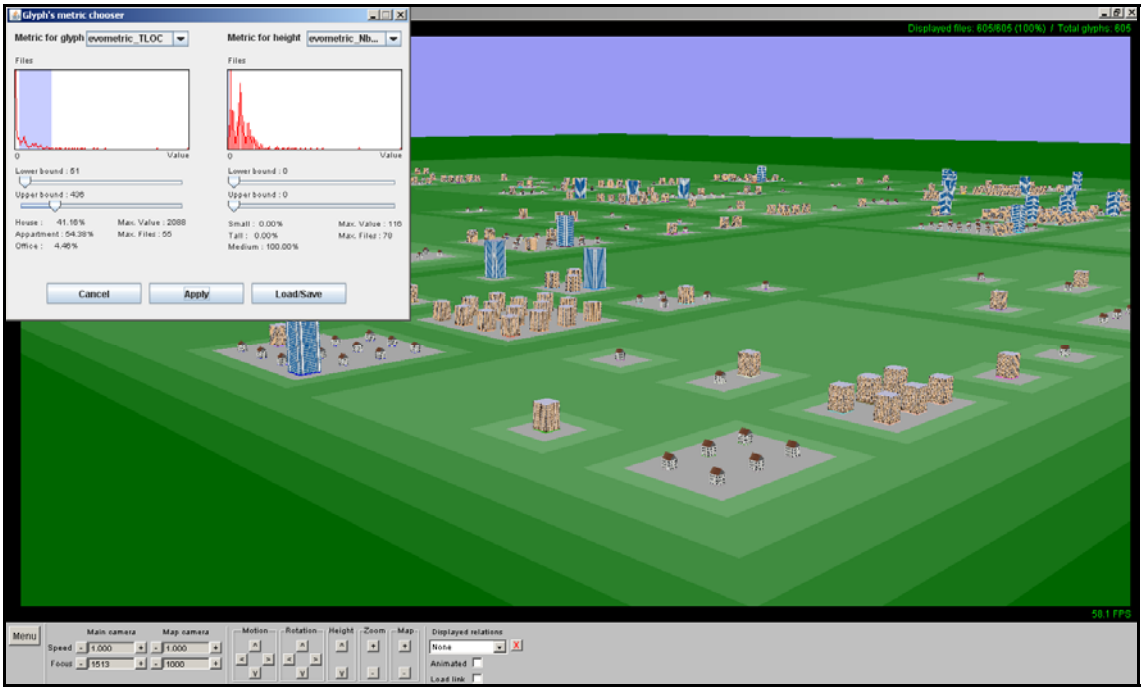
> *A measurement mapping M must map entities into numbers and empirical relations into numerical relations in such a way that the empirical relations preserve and are preserved by the numerical relations" (Fenton & Pfleeger, 1996)*

Michele Lanza summarizes this problematic: *"In other words, if a number a is greater than* a *number* b*, the graphical representation of* a *and* b *must preserve this fact."* (Lanza & Marinescu, 2006).

To make it perfectly clear, here is what happens when we set both thresholds of metric2 to 0, which sets all building to “maximum height”:

Figure 21

Height metric set to all buildings tall



All building simply grew bigger in their category. In no way they look like they are all tall, as they should.

5.3.5 Proposition for intuitive metric display

We found that the metric2 didn’t really “show up” in the display because it ran in contradiction with human intuition.

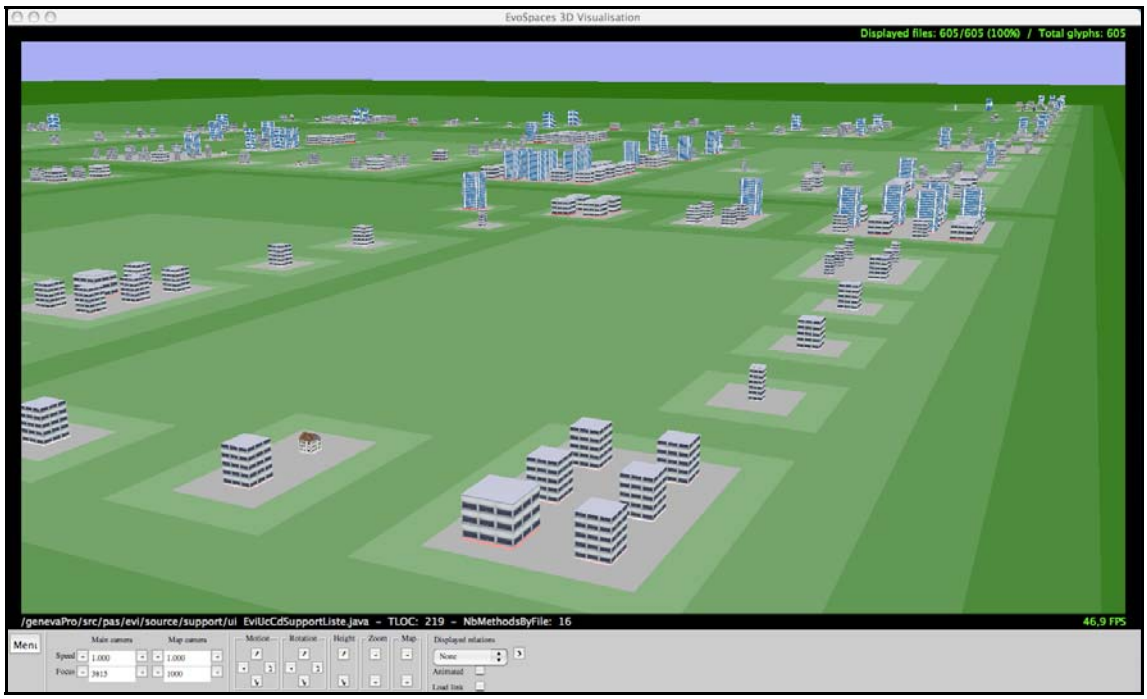
We propose a very simple fix: to apply the metric1 to the width of the building and to set the building type according to the height.

Tall buildings will be office towers, medium ones will be blocks and small ones will be houses. “Fat” buildings indicates a high metric1, thin ones indicates a low metric1.

This makes both metrics obvious, intuitive and visible from far.

Figure 22

Making metric more obvious with new texture



A third metric hint has been added: the coloured base of buildings is filled according to the number of lines of code metric (TLOC). It uses a gradient with the maximum set at 1000 lines of code.

We applied the texture in such way that it exaggerates the perception of width. Wide buildings look wider and thin one look thinner, even from faraway.

5.3.6 Defining new metrics to answer new questions

The standard metrics we have used until now are useful, but they bring limited information.

A typical maintainer's question is: "Which part of the code will likely be impacted by modification of the program?"

To answer this kind of question, we designed and generated two metrics (see above, the chapter on metric generation):

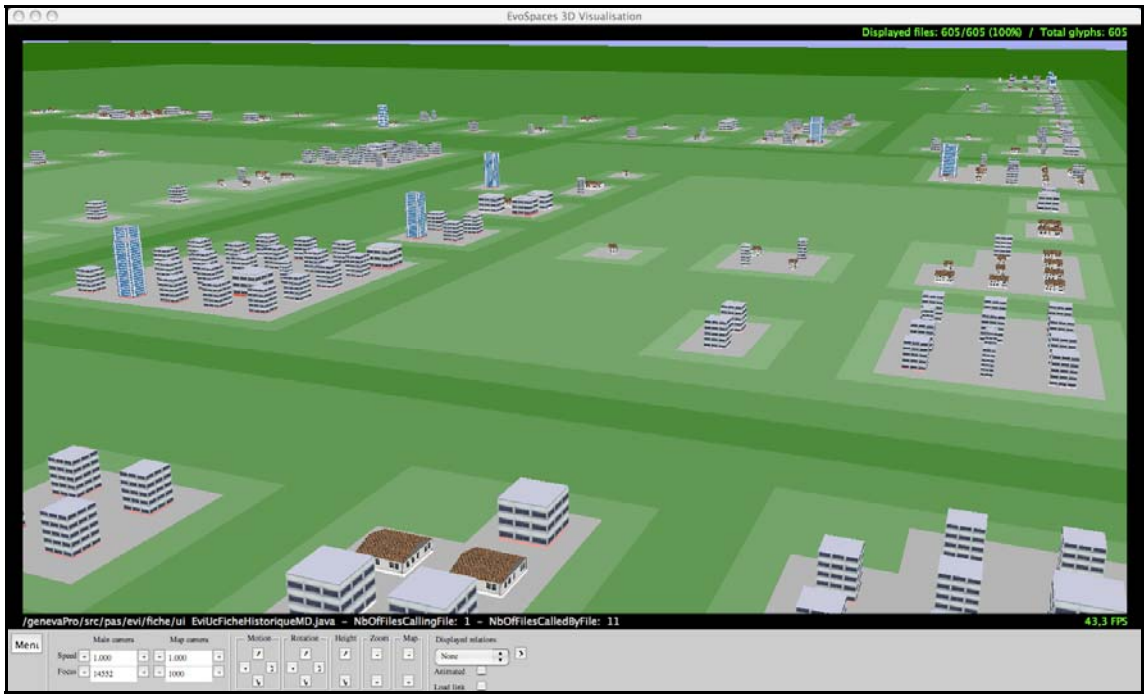
- The number of files "calling" a file. Represents how many files depend on one or more methods of this file. This gives the number of files potentially affected by a modification in this file.

- The number of files “called” by a file. Represents how many files the present file calls methods in. This gives a measure of the likeliness a file is to be affected by a change somewhere else in the program.

These metrics also give an indication on the role of a file in the program. They can be combined to detect files that represent “crossroads”. Then next screenshot gives us an example:

Figure 23

Finding « crossroad » files with call metrics

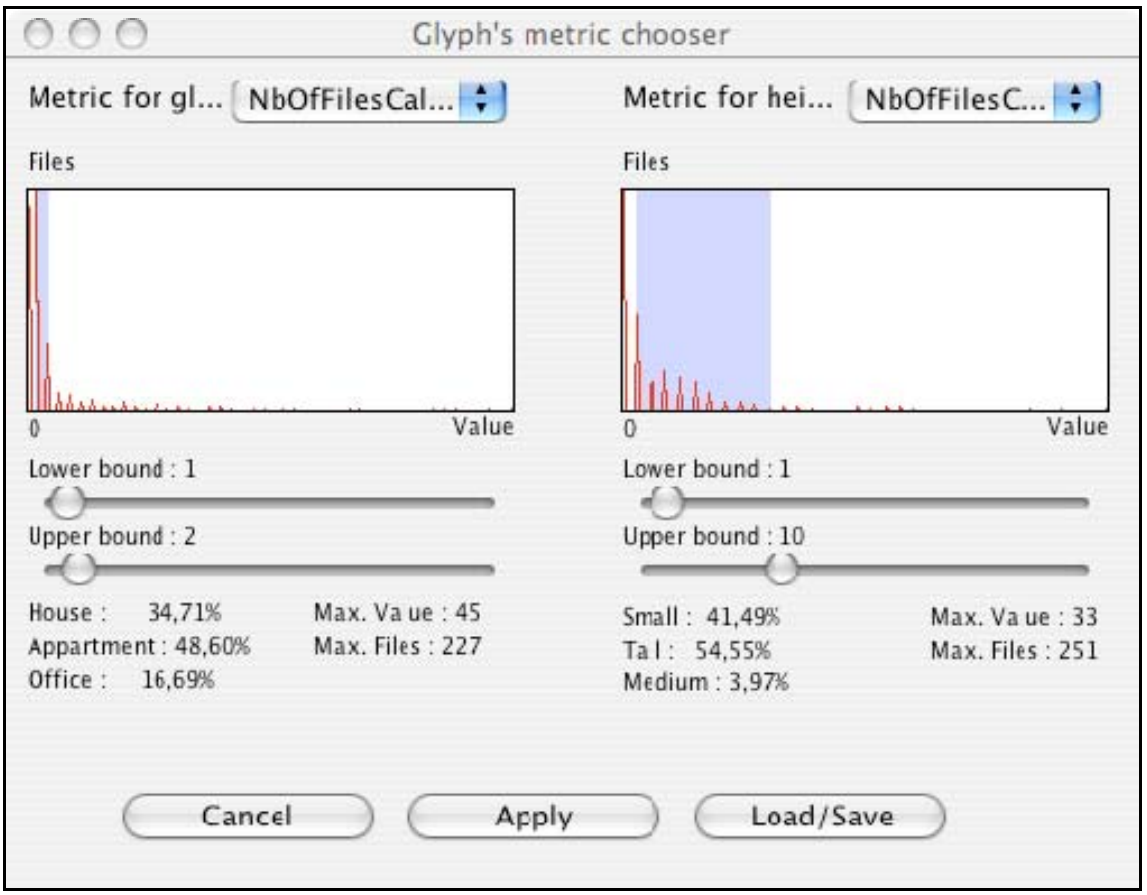


The relative role of each file is made obvious by the careful use of the metrics. Large tower indicate files calling a lot of other, houses indicate files that are only receive calls.

The choice of particular thresholds is very important. Here are the settings used in the previous screenshot:

Figure 24

Call metrics thresholds setting

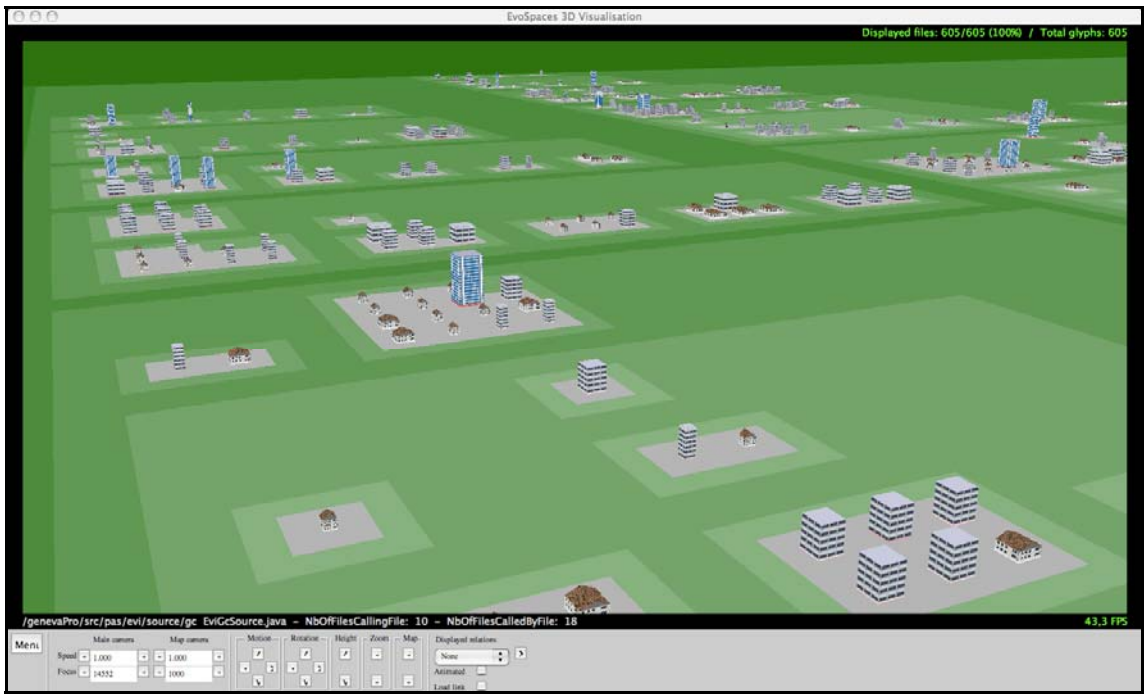


By setting the threshold of metric1 at 1 and 2, we know that the thin buildings are never called, the medium ones are called by exactly one file, and the wide ones by to or more files.

By setting the threshold of metric2 at 1 and 10, we know that houses are never calling, block buildings are calling a few files (less than 10), and office towers are calling 10 or more. Here is another perspective on the same scene.

Figure 25

Finding « crossroad » files, other perspective



We would like to stress this point about metric settings: optimal settings are dependant on the metric itself. We strongly suggest that proper default values be stored alongside metrics in the database.

Moreover, some metric may have several set of “optimal” threshold values, each uncovering different kind of information.

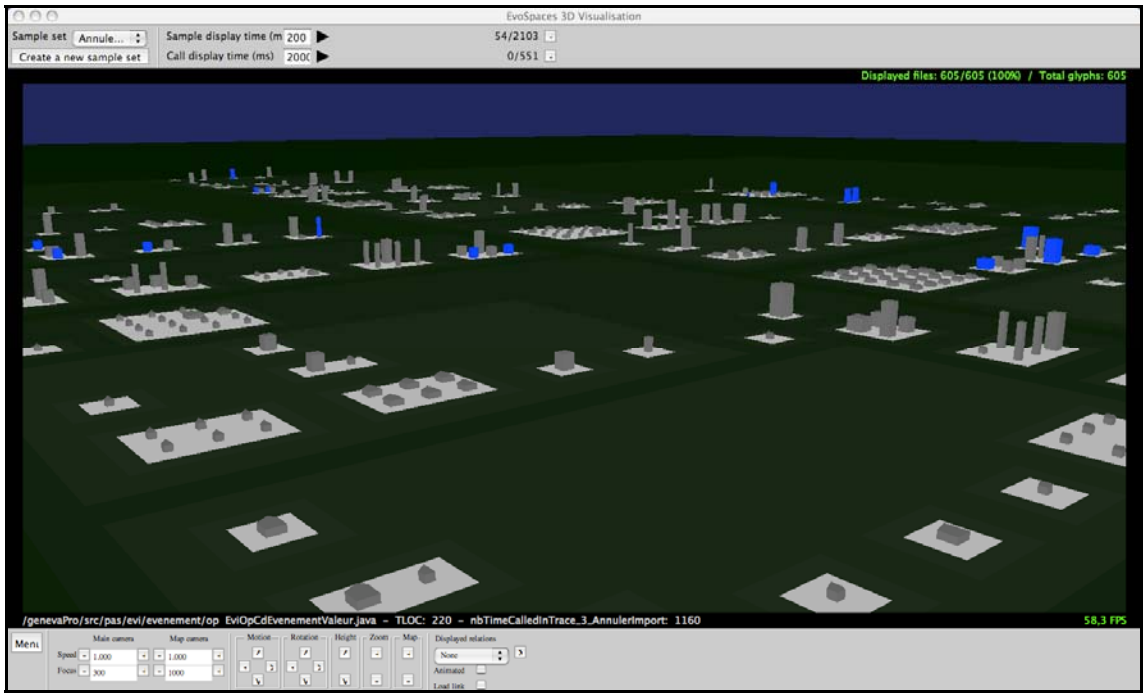
5.3.7 What about the run-time: adding trace metrics

Until now, all our metrics where generated from static code information. However, we can also derive synthetic information from execution traces in the form of trace metrics.

To demonstrate this capability, we generated a metric measuring the number of time a file is called in a single trace. Here is the resulting display, in the night view:

Figure 26

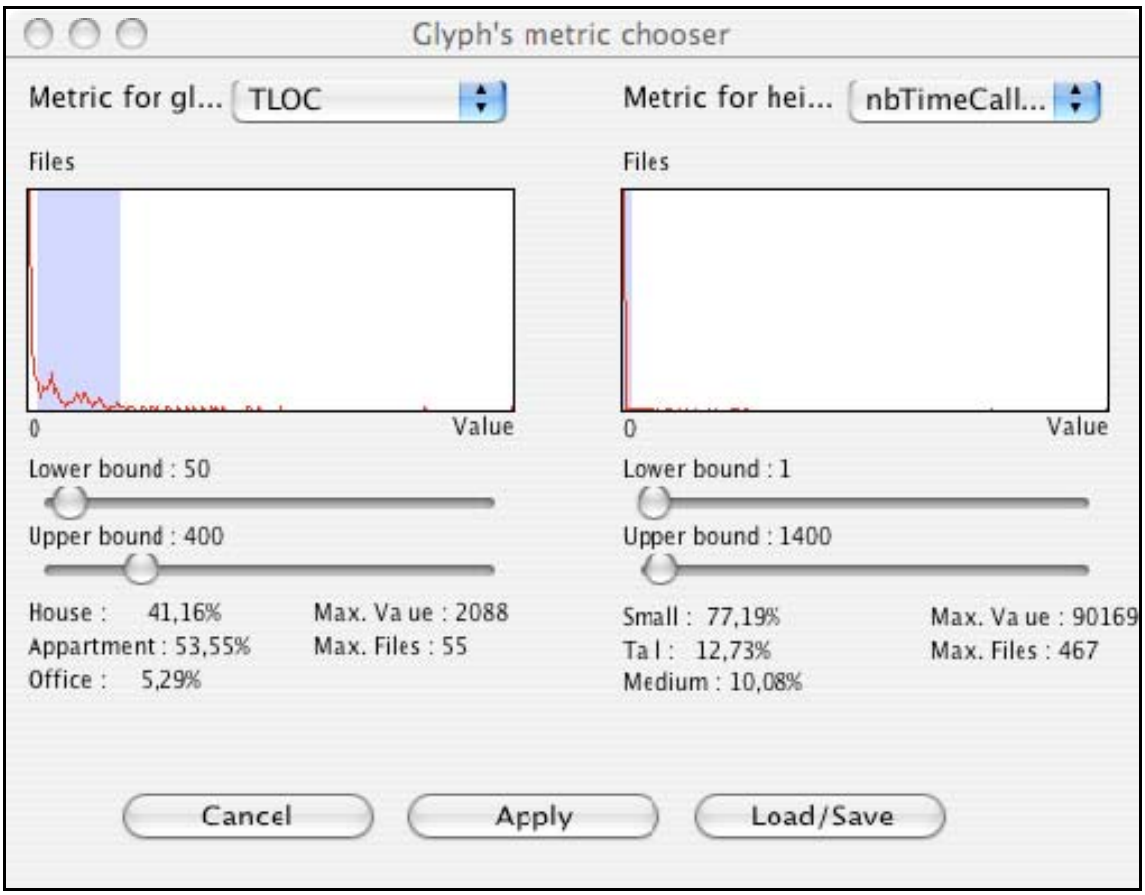
Night view with number of calls trace metric



The metric thresholds for this view are set as follow:

Figure 27

Thresholds for number of calls trace metric



The choice of 1 as the lower metric2 threshold make all files not involved in the trace to be displayed as houses.

5.3.8 Conclusions on the tests

Due to the limited time we have at our disposal, we could not present our tests in more details. But we believe that it is now conformant to the rules we defined. Moreover, we consider that it has successfully passed its first real-world test.

6. Proposition for future EvoSpaces development

6.1 Functionalities and user interface

6.1.1 User interface and display

The building's content should be displayed on top of the buildings. The current display of transparent building induces a loss of information on the file that are precisely those under scrutiny. We believe drawing the content on the rooftop would solve this problem.

The current metrics should be displayed on the top black edge of the display along with their current threshold values.

Currently, associations are drawn without respect on their direction. A colour code should mark the begin and the end of association lines on the display.

6.1.2 City layout

The current content-driven layout could be improved by finding a more compact way of laying out packages.

6.2 Application architecture

6.2.1 Proposition for new dedicated DB schema

Time was lacking and we could not produce a new database schema, as we intended to initially.

We recommend implementing a schema as close as possible of the Famix 2.0 standard. It contains all the fields necessary to describe the entities and associations.

A set of special tables must be added to store information not covered by Famix, in particular files and directories, which are better kept separately.

The rule of data modelling should be applied as well as those of common sense.

It can be desirable to implement tables with data derived from other tables, in order to speed performance and present the source code in a different light. They would be filled with data such as the *contains* association, which is so useful to EvoSpaces in building its display.

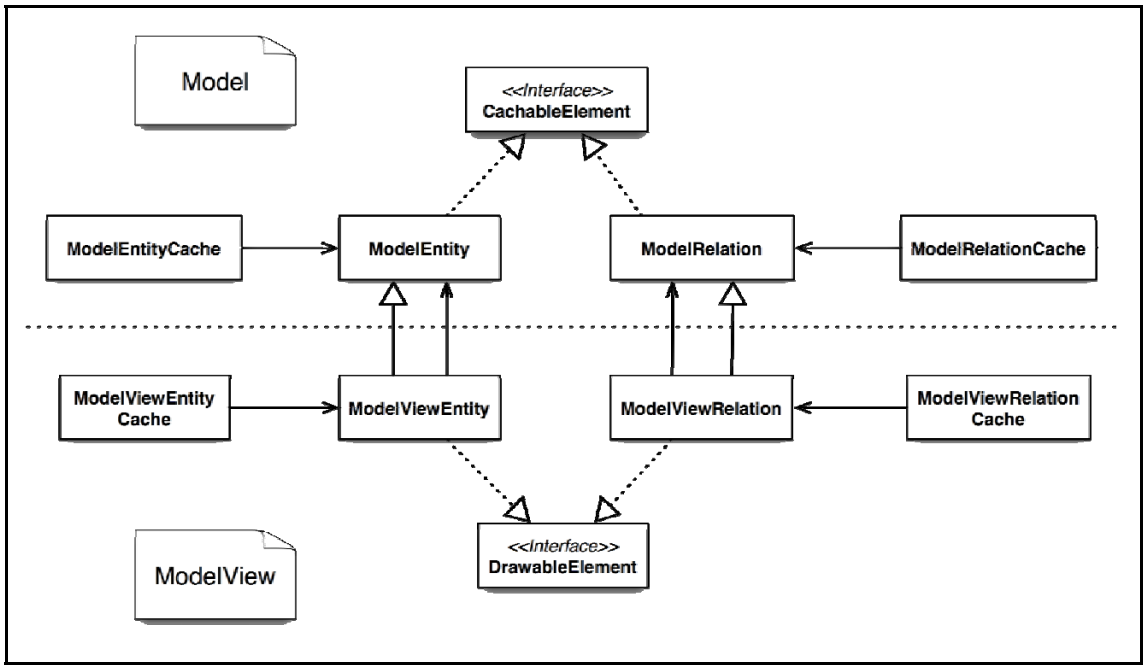
These tables would be filled by a utility in EvoSpaces before the first run.

6.2.2 A new cache infrastructure

The cache infrastructure should be re-engineered according to the following diagram.

Figure 28

Proposed cache architecture



The separation of the entities and association prevents any future problem with IDs, and allows to cache associations as well as entities.

There is no drawback to implementing this architecture, because there was effectively no gain in having a unified cache. The clients always knew in advance the kind of elements they were about to load, so they will have no problem in determining the appropriate cache to use.

6.2.3 Other architectural enhancements

As a means to clarify the source code, Famix-standard attributes of entities and associations should have their own attribute variables in ModelEntity and ModelRelation. Ideally, they should match those of the new database schema.

A centralised start-up configuration mechanism should be designed. It should allow saving the configuration data, such as database settings, in configuration file.

Conclusion

The first part of this work consisted in adapting EvoSpaces to a new database schema, generate metrics on the code of our target application and load execution traces. This part of the work was the most technical one and required from us strong analytical skills. It also greatly improved our experience in relational databases.

In the second part, we defined what is expected from EvoSpaces on a software maintainer's point of view, and the design rules it must follow in order to be useful. We showed how these rules helped us uncover a flaw in how metrics were mapped to buildings.

In this work, we believe we have brought EvoSpaces to the next level. The revised display combined with new software metrics has demonstrated its usefulness. More generally the concept of driving a 3D world with software metric is validated.

The role of software metrics must also be underlined. They are the key information provider. This is why we think that imagining and developing innovative software metrics is the key to EvoSpaces future.

Also, improvements to the layout algorithm would be welcome, in the form of a more compact and orderly city.

We believe that EvoSpaces now includes all the necessary elements to become a product for real world use.

Glossary

Target system : The software system under analysis.

DDL : Data definition Language, the SQL instructions creating the tables and constraints in a relational database.

Evolizer 2005 : The database schema that was used to store the analysed source code of Mozilla 1.7 by Evolizer or the tool that had this role back in 2005.

Evolizer 2007 : The database schema used by Evolizer as of September 2007 and that comes with the analysed PasEvi source code.

Bibliography

E.J. Chikofsky, J.H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software Engineering Journal*, pp. 13-17, Jan. 1990.

Dugerdil, P., Architecture-based software reengineering. Geneva: Haute Ecole de Gestion (University of Applied Science), February 2005.

Jossi, S, *Reverse-Engineering du système d'information*, Geneva: Haute Ecole de Gestion (University of Applied Science), December 2006.

Alam, S., Dugerdil P., *EvoSpaces: 3D Visualization of Software Architecture*, HEG – Univ. Of Applied Sciences, Geneva, Switzerland, July 2007

Dugerdil, P., Alam, S., *Who is Working Tonight? Understanding Systems through Execution Trace Visualization in the 3D Space*, Department of Information Systems, Geneva: Haute Ecole de Gestion (University of Applied Science), July 2007.

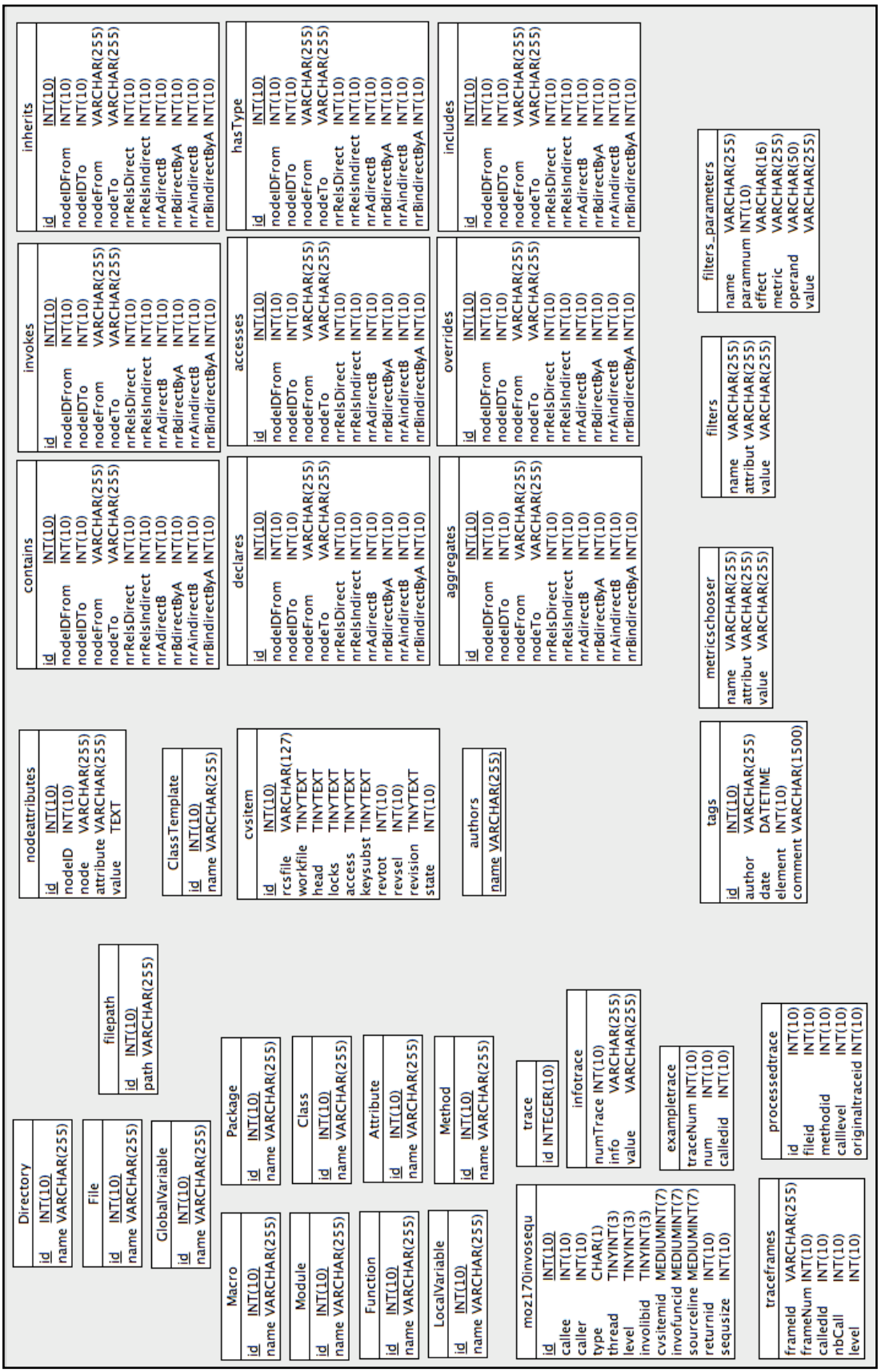
Demeyer, S, Tichelaar, S, Steyaert, P, *The FAMIX 2.0 specification, V2.0*, http://www.iam.unibe.ch/~famoos/FAMIX/Famix20/famix20.pdf, sept. 1999.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. M. (GoF), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley Professional, 1994.

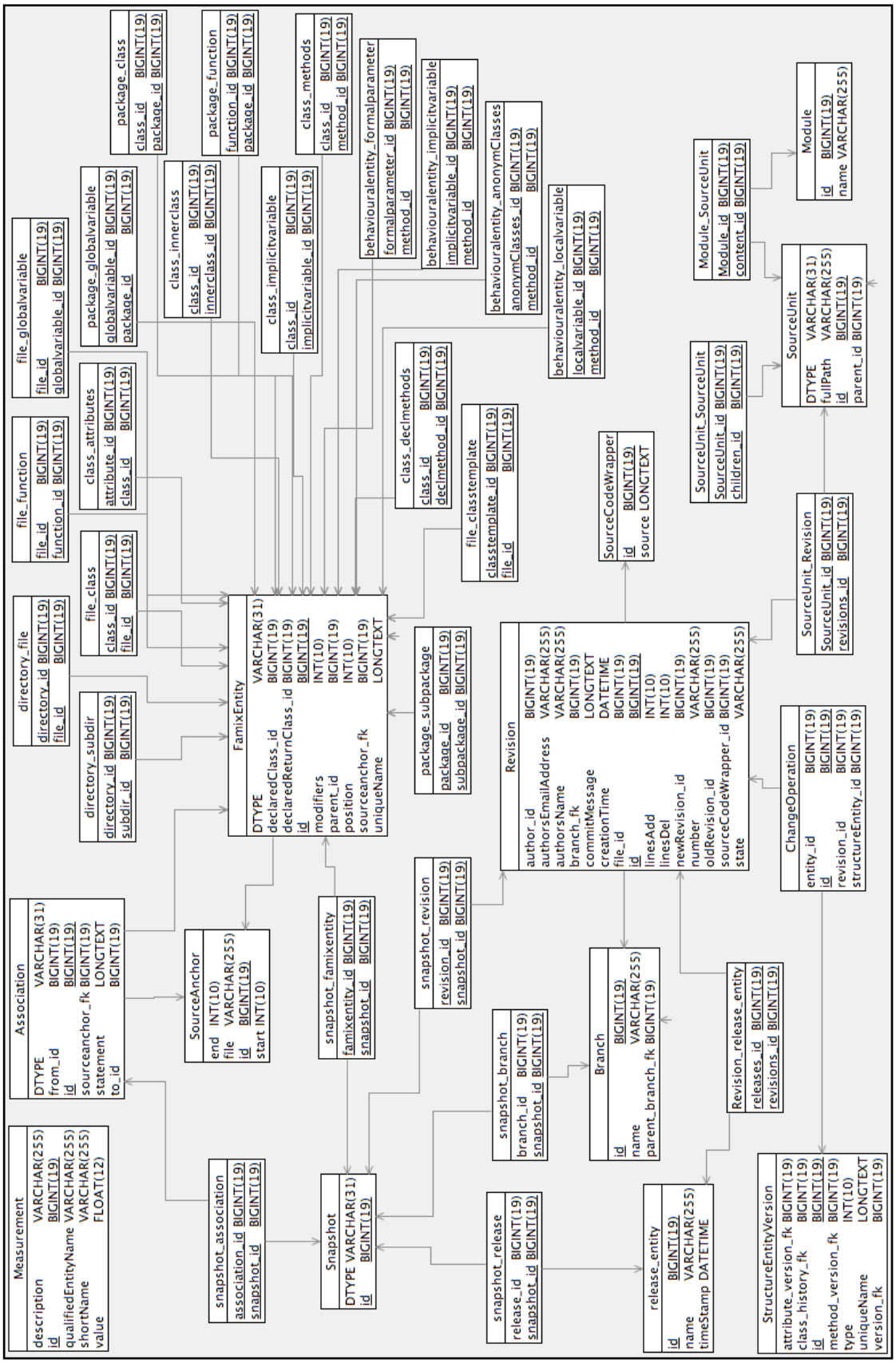
Fenton, N., Pfleeger, S. L., *Software Metrics: a Rigorous and Practical Approach*, International Thomson Computer Press, London UK, second edition, 1996.

Lonza, M., Marinescu, R., *Object-Oriented Metrics in Practice*, Springer, 2006.

Annexe 1
Evolizer 2005 database schema (partial)



Annexe 2
Evolizer 2007 database schema (partial)



Annexe 3
Raw execution trace sample

```
END EviUcCdProjetListe.java isCellEditable(int, int)) As boolean
EviUcCdProjetListe.java getRowCount()) As int null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null
END EviUcCdProjetListe.java getRowCount()) As int
EviUcCdProjetListe.java mouseClicked(MouseEvent)) null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null
EviUcCdProjetListe.java _maListe_mouseClicked(MouseEvent)) null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null
EviUiCdProjetListe.java composantAction(String, int)) ucCdProjetListe, null,
null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null, null
EviUiCdProjetListe.java getGc()) As GenGcAble null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null
END EviUiCdProjetListe.java getGc()) As GenGcAble
END EviUiCdProjetListe.java composantAction(String, int))
END EviUcCdProjetListe.java _maListe_mouseClicked(MouseEvent))
END EviUcCdProjetListe.java mouseClicked(MouseEvent))
EviImCdProjetImpl.java lire(GenOdStandard)) As GenOdStandard null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null
EviImCdProjetImpl.java getOm()) As EviOmCdProjet null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null
END EviImCdProjetImpl.java getOm()) As EviOmCdProjet
EviOmCdProjet.java lire(GenOdStandard, Connection)) As GenOdStandard null,
null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null, null, null
END EviOmCdProjet.java lire(GenOdStandard, Connection)) As GenOdStandard
EviOmCdProjet.java getOp()) As EviOpCdProjet null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null
END EviOmCdProjet.java getOp()) As EviOpCdProjet
EviOpCdProjet.java readData(ResultSet)) As GenOdStandard null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null
EviOdCdProjet.java setCode(java.lang.String)) null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null
END EviOdCdProjet.java setCode(java.lang.String))
EviOdCdProjet.java setLibelle(java.lang.String)) null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null
END EviOdCdProjet.java setLibelle(java.lang.String))
END EviOpCdProjet.java readData(ResultSet)) As GenOdStandard
END EviImCdProjetImpl.java lire(GenOdStandard)) As GenOdStandard
EviGcCdProjet.java objetChanged()) null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null
END EviGcCdProjet.java objetChanged())
EviGcRefCdProjet.java setReferenceOd(String, GenOdStandard))
ProjetDetail.CodeTypeProjet, null, null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null
EviGcProjet.java setCdProjet(String, EviOdCdProjet))
ProjetDetail.CodeTypeProjet, null, null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null
EviGcProjet.java getObjet()) As GenOdStandard null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null
```

Annexe 4
Processed execution trace

```
583 EviOdDateFourchette.java getBorneInf()
583 EviOdDate.java setValeur(int)
584 EviOdDate.java updateValues(boolean)
585 EviOdDate.java getValeur()
585 EviOdDate.java getAnnee()
585 EviOdDate.java getMois()
585 EviOdDate.java getJour()
583 EviOdCollectivite.java getDateDissolution()
583 EviOdDateFourchette.java getBorneSup()
583 EviOdDate.java setValeur(int)
584 EviOdDate.java updateValues(boolean)
585 EviOdDate.java getValeur()
585 EviOdDate.java getAnnee()
585 EviOdDate.java getMois()
585 EviOdDate.java getJour()
583 EviOpCollectivite.java validFromDB(int)
583 EviOdCollectivite.java setInstance(java.lang.Boolean)
583 EviOdCollectivite.java getDateCreation()
583 EviOdDateFourchette.java getTexte()
583 EviOdDateFourchette.java toString()
583 EviOdCollectivite.java setzDatationDebut(java.lang.String)
583 EviOdCollectivite.java getDateDissolution()
583 EviOdDateFourchette.java getTexte()
583 EviOdDateFourchette.java toString()
583 EviOdCollectivite.java setzDatationFin(java.lang.String)
582 EviOdActeur.java setColRef(EviOdCollectivite)
581 EviOdActeur.java setModifiable(boolean)
579 EviOdFicheDescriptive.java setActeurEvaluation(EviOdActeur)
579 EviOdFicheDescriptive.java getActeurEvaluation()
579 EviOdActeur.java getzNomComplet()
580 EviOdActeur.java getType()
579 EviOdActeur.java getzNom()
580 EviOdActeur.java getType()
579 EviOdActeur.java getColRef()
579 EviOdActeur.java getColRef()
579 EviOdCollectivite.java getNom()
579 EviOdFicheDescriptive.java setzActeurEvaluation(java.lang.String)
578 EviOmFicheDescriptive.java lireListeOrigines(EviOdFicheDescriptive,
Connection)
579 EviOmImportableBase.java getImOrigineFiche()
579 EviImFicheOrigineImpl.java rechercher(GenOdStandard, int, int)
580 EviImFicheOrigineImpl.java getOm()
580 EviOmFicheOrigine.java rechercher()
580 EviOmFicheOrigine.java getOp()
580 EviOpFicheOrigine.java getClauseWhere(GenOdStandard)
580 EviOpFicheOrigine.java readListData(ResultSet)
581 EviOdFicheOrigine.java getDatationOrigine()
581 EviOdDateFourchette.java setNuance(String)
581 EviOdFicheOrigine.java setFidRef(int)
581 EviOdFicheOrigine.java getDatationOrigine()
581 EviOdDateFourchette.java getBorneInf()
581 EviOdDate.java setValeur(int)
582 EviOdDate.java updateValues(boolean)
583 EviOdDate.java getValeur()
583 EviOdDate.java getAnnee()
583 EviOdDate.java getMois()
583 EviOdDate.java getJour()
583 EviOdDate.java getValeur()
583 EviOdDate.java getValeur()
583 EviOdDate.java getValeur()
```

Annexe 5
Running TraceLoader

Traceloader converts a trace file in SQL INSERT statements.

To use TraceLoader, change directory to the EvoTools program directory. The synopsis of the command is:

```
java traceloader/TraceLoader <std|raw> <tracename> <path to tracefile>
```

This will print all output on the console.

This is how to convert a processed trace and store the result to a file on windows:

```
java traceloader/TraceLoader std mytrace \path\to\tracefile > result.sql
```

The > sign redirects the output of TraceLoader to the file result.sql.

Annexe 6
Tools we used

Eclipse	Development environment	www.eclipse.org
MySQL	DBMS	www.mysql.org
MySQL Administrator	MySQL administration tool	www.mysql.org
MySQL Query Browser	MySQL query application	www.mysql.org
Cayenne Modeler	GUI database modeler that supports reverse engineering	Apache Fondation
Azzurri Clay	Database modeling and reverse engineering	www.azzurri.jp /en/software/clay/index.jsp
OmniGraffle Pro 4	Powerful tool for drawing diagram and export them in many formats	www.omnigroup.com
Jedit 4.3	Java text editor with regular expressions supoort	www.jedit.org