Christian-Albrechts-University of Kiel
Department of Computer Science
Software Engineering Group

Runtime Visualization of Static and Dynamic Architectural Views of a Software System to identify Performance Problems

Bachelorthesis

2010-03-25

Written by: Christian Wulf
born in Kiel on 1986-12-20

Supervised by: Prof. Dr. Wilhelm Hasselbring
Dipl.-Inf. Jan Waller

Abstract

Today, most enterprise software system's complexity exceed by far the human's capacity to quickly and correctly analyze it. For this reason, We describe and evaluate several, especially three-dimensional visualization approaches that enable static and/or dynamic visual analysis. Furthermore, we present and review our own 3D-approach consisting of a city metaphor model and a sample implementation called Dynamic Visualizer (DyVis) that focuses on detecting and analyzing performance problems. As the first approach, it completely integrates dynamic program trace information into the static context of a software system so that one can easily follow and interact with a given trace.

Contents

1 Introduction **1**

2 Approaches **3**
2.1 UML . . . 3
2.2 The city metaphor . . . 6
2.2.1 Representation of software elements . . . 6
2.2.2 Interaction & Navigation . . . 8
2.3 Different representations of an execution trace . . . 9
2.3.1 Dynamic visualization based on the city metaphor . . . 10
2.3.2 Graph-based visualization . . . 12
2.4 Eclipse Modeling Framework . . . 15

3 Evaluation of the approaches **18**
3.1 Bases of evaluation . . . 18
3.1.1 Evaluation criteria . . . 19
3.1.2 Shneideman's taxonomy . . . 20
3.2 UML . . . 22
3.3 CodeCity . . . 24
3.4 Evospaces . . . 26
3.5 TraceCrawler . . . 28
3.6 Conclusion/overview . . . 31
3.7 Eclipse Modeling Framework . . . 33

4 An extended city metaphor **35**
4.1 Overview . . . 35
4.2 Model package: visualization . . . 36
4.3 Model package: dataTypes . . . 37

5 DyVis - A prototype implementation **38**
5.1 Overview . . . 38
5.2 Functions . . . 38
5.3 Evaluation . . . 41
5.4 Future work . . . 43

6 Related work **45**

7 Conclusions **47**

References **48**

A Documentation **i**

B Attachments **xviii**

1 Introduction

Today, most enterprise software systems exceed by far a degree of complexity that software engineers are able to grasp by just looking at the program code. For this reason, specifications and tools have been invented which facilitate development and analysis of software systems to be more clearly represented and controllable.

The release of the first version of the Unified Modeling Language (UML) by the Object Management Group (OMG) had probably the most impact on software engineering in the nineties. The UML is a standardized modeling language that allows to create visual models of object-oriented software systems. It provides several diagrams to visualize both the static structure of a system (in terms of software entities and their relationships) and the dynamic aspects, namely class instantiations and message interactions in context of time. Hence, software architects can design on a higher level of abstraction. Additionally, there are tools that can translate these high-level models into equivalent source code in a specific programming language. Inversely, it is possible to generate a static overview of the structure of a software system out of its source code by reverse-engineering tools such as ArgoUML[1] or BOUML[2] to get a better idea of the interna of a system.

In addition to those software engineering tools mentioned above, there has been work on tools that analyse the runtime behavior of a system. A so-called monitoring tool, such as Kieker, is able to collect information on methods, e.g., associated class, method name, method parameters and execution time of a method invocation, by means of software probes inserted into the target system. Hence, it is possible to detect hotspots and bottlenecks in a running system or to follow the execution trace of a single action in detail.

Unfortunately, these execution traces produce enormous quantities of information [3, 1] making them difficult to interpret and to display. For example, a sequence diagram of the UML, whose drawing surface is two-dimensional, already lacks in an appropriate overview if it simply presents a few instantiations of classes or a minor number of method invocations.

Recent approaches [6, 7, 8, 9, 3, 11, 1] expand the 2D-visualization of the UML diagrams already considered by another dimension to describe the statics and dynamics of software systems. The third dimension provides more space and fully utilizes the human's capabilities in orientating and navigating. For example, as the human subconscious accomplishes several tasks in everyday life, it can also perform some of the cognitive processing for navigation and visual interpretation in such a

[1] `http://argouml.tigris.org/`
[2] `http://bouml.free.fr/index.html`

virtual reality. Thus, using a 3D-visualization tool can be more intuitive and can lead to a more intuitive and expressive representation of a software system.

In this manner, Wettel and Lanza [11] describe their approach in terms of their tool CodeCity. It depicts the statics of an object-oriented software system as a three-dimensional city that can be traversed and interacted with. Rectangular districts and buildings symbolize packages and classes, respectively. A building's height represents the number of methods, while its width and length represent the number of attributes.

Dugerdil and Alam [1] extend the software city metaphor as they distinguish between day and night. In daylight mode, their tool called EvoSpaces is able to depict the static view of a software system as well as CodeCity does. But in night mode, EvoSpaces can additionally display an execution trace by highlighting the participated classes and representing their relationships in terms of solid pipes.

Greevy et al. [3] describe another approach. Their visualization tool called TraceCrawler (an extension to CodeCrawler [2]) renders both the static structure as a class hierarchy and the dynamic behavior as towers of communicating instances. Classes are placed on a plane above the ground and each instantiation of a class spawns a box above the ground level of its corresponding class representation, like a floor in a building. An edge between two objects symbolizes a message, i.e., a method invocation between the instances of two different classes.

We propose a meta-model to describe the three-dimensional representation of the dynamics in context of the statics of a software system based on the city metaphor by combining recent approaches with own ideas. Additionally, we introduce and evaluate the tool DyVis that was developed within the scope of this thesis. It serves as a proof-of-concept for the own approach and provides functionality to analyze execution traces and identify performance problems, in particular bottlenecks.

Structure of this thesis. In the next Section, we describe in detail the approaches mentioned above and the Eclipse Modeling Framework (EMF) to specify our meta-model. After that, in Section 3, we review the approaches and the use of the EMF in respect of our purpose. Section 4 introduces the meta-model with its entities, each of them with a full description and the justification of its existence in this model. The subsequent Section presents the prototype implementation DyVis based on the meta-model and an evaluation of it. In Section 6, we provide an overview of related work and compare our visualization technique with other visualization-based approaches. The last Section concludes the thesis and gives an outlook on possible future work.

2 Approaches

Recently, there has been some novel work on visualizing the architecture and dynamics of object-oriented software systems. The 3D approach in combination with an appropriate metaphor of representation provides another, different, more intuitive view of a system. In the following, we briefly describe the standardized 2D-visualizing UML for the purpose of criticism and comparison to 3D approaches in Section 3. Subsequently, we present in detail the approaches and ideas that provide the basis for our own approach in visualizing execution traces within a 3D city metaphor.

2.1 UML

The Unified Modeling Language (UML) is a general-purpose modeling language in the context of software engineering. The Object Management Group (OMG), creator and manager of the UML, has developed this standardized language specification to depict software components in a uniform way. UML 2.2[3] provides 14 different types of diagrams that software engineers can use to construct, specify, document and visualize object-oriented software systems. Figure 1 shows a hierarchical overview of all these available diagrams.

The UML distinguish between structure and behavior diagrams. The former are intended to depict the statics, the latter are used to describe the dynamics of a software system. Both categories contain diagrams for low and higher levels of abstraction. In addition to diagrams representing the system itself, an activity diagram can visualize (human) actors and their associations to particular system components and system functions. However, we will focus on low-level diagrams due to our purpose of visualizing basic, reconstructed software elements and their interactions among each other during runtime. In order to display higher-level views, please refer to Section 5.4. Below, we describe an assortment of structure diagrams and subsequently, we describe low-level behavior respectively interaction diagrams in more detail.

An object-oriented software system—in the simplest case—consists of at least a number of basic *class* elements. For this reason, UML provides the *class diagram* that represents classes and their relationships among each other. Generally, a rectangular box symbolizes a class containing the class name itself as well as all attributes and methods both with their names, types and return types, respectively,

[3]While writing this thesis, version 2.2 was the current version of the UML specification.

and visibilities[4]. A line linking two classes, for instance a dashed line or a solid line with an arrow at one end, represents a relationship between them, such as generalization, composition or usage. Each end of a relationship has properties that specify the role of the association end, its multiplicity, navigability and constraints. For more details, please refer to the official UML specification [5].

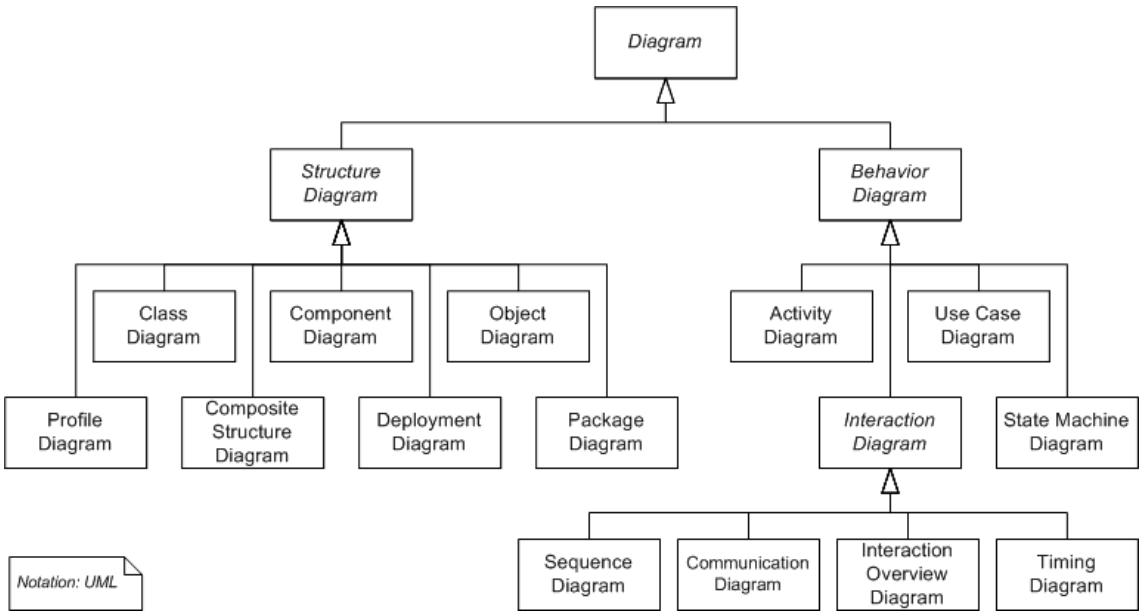


Figure 1: Hierarchy of diagrams in UML 2.2 [5, p. 686]

An *object diagram* represents a complete or partial view of a system at a specific time. Hence, contrary to a class diagram, it depicts objects with their concrete attribute values (cf. class diagrams above). A link, symbolized by a solid line, between two objects represents an instance of an association. Usually, object diagrams are used to provide examples or act as test cases for corresponding class diagrams.

For reasons of clear arrangement and structuring, the object-oriented programming paradigm proposes a hierarchical order in terms of packages where each of them contain classes with similar meanings. For this purpose, the UML provides the *package diagram* that represents those packages as well as their dependencies and usage associations among each other. The correspondent logical view, however, is described by the UML's *component diagram*. It depicts components, i.e., collections of packages and classes, in combination with their interfaces and dependencies among themselves. Thus, this type of diagram describes the architecture of a system on a more higher level of abstraction than a package diagram does.

[4]The visibility or scope, e.g., `private` or `public`, is an essential feature of the object-oriented programming to support information hiding.

Besides structure diagrams, there are several interaction respectively behavior diagrams, such as the *sequence diagram* and the *communication diagram*. The former type of diagram shows the process or program trace of a particular actor's action, method call or message dispatch through out its participating processes or objects. In this context, the three types of invocation are called *messages* generically. A sequence diagram integrate messages, objects and time in a two-dimensional coordinate system whose y-axis points down to the mathematical negative direction. The x-axis represents the participants whereas the y-axis represents the point in time. Messages are drawn as horizontal dashed lines, each between two participants. A message is at least labeled with its message name and can optionally be tagged with parameters and a return type. In more detail, a box on the x-axis represents a particular participant as it contains the participant's name. Participants are arranged in parallel and, in each case, offer a parallel vertical dashed line below their box representations, called life-line. The life-line of an object indicates the time or duration it exists in memory. Hence, the viewer can follow the trace by pursuing the message interactions from top to bottom. Further extensions and information can be found in the official specification [5].

Communication diagrams are similar to sequence diagrams. Both depict the dynamic collaboration between elements. A communication diagram, however, does not consider the time, but focuses on the context. It shows objects in a freely arrangeable form and represents basic associations among them. To maintain the order of messages, each message label contains a unique number. Thus, the order of numbering correspond to the chronological order. In conclusion, a communication diagram comprises information of a class, object and sequence diagram to represent both the static and dynamic.

In addition to these two interaction diagrams, the UML specifies *interaction overview diagrams* as well as *timing diagrams*. The former visualizes a sequence of activities similar to the UML's activity diagram, but activities are replaced by frames. Each frame consists of either a sequence diagram or, recursively, another interaction overview diagram. In this manner, it can reduce the complexity of an intricate scenario that would otherwise require multiple if-then-else paths to be illustrated as a single sequence diagram.

Timing diagrams are a special form of sequence diagrams and focus on timing constraints. Software modelers use them to precisely document a schedule of interaction or state changes. A detailed description can be found in the official UML specification [5].

2.2 The city metaphor

First research reports on three-dimensional software visualization [7, 9] have brought up a new approach to analyze and identify software systems and their particular components. Unfortunately, due to lack of locality, i.e., objects can be freely moved in space and the viewer is allowed too much freedom of movement, these kinds of representation lead to disorientation and hence, fail to produce an intuitive alternative to two-dimensional visualizations, such as UML.

To revive the thoughts of a 3D representation of a software system, Wettel and Lanza [11] enhance the visualization technique. They provide locality by using a well-established metaphor to embed the represented elements, e.g., classes and packages, into a familiar context, preventing disorientation in space. They propose a 3D visualization which describes an object-oriented software system as a city with districts and buildings representing integral parts of the system.

Additionally, the authors choose a different level of granularity than Knight and Munro [6] or Panas et al. [8] did in previous work on visualizing software systems as 3D cities. Wettel and Lanza claim that the level of granularity is crucial to properly support the city metaphor and for this reason, the latter approaches fail to achieve an adequate degree of popularity. Hence, they especially focus on representing classes and packages because these software constructs are the cornerstones of the object-oriented paradigm and thus the most important orientation points for developers. However, their approach shows the static structure only, that is, they do not consider dynamic aspects.

2.2.1 Representation of software elements

Wettel and Lanza [11] focus on object-oriented programs, that is, they depict the constructs that need to be understood in this context, such as packages, classes, methods, attributes and all their relationships. The authors represent classes as buildings located in city districts, displayed as stacked platforms at different altitudes, which in turn represent nested packages.

They justify their representation by stating that a city, with its downtown area and its suburbs, is a familiar notion with a clear concept of orientation. A large city is still an intrinsically complex construct which can only be explored incrementally, in the same way as the understanding of a complex system increases step by step. The authors argue that using an all too simple visual metaphor leads to incorrect oversimplifications and thus conclude that we have to cope with the fact that software is complex. Furthermore, they do not display class internals, e.g., methods and attributes, because it is unnecessary for a large-scale understanding

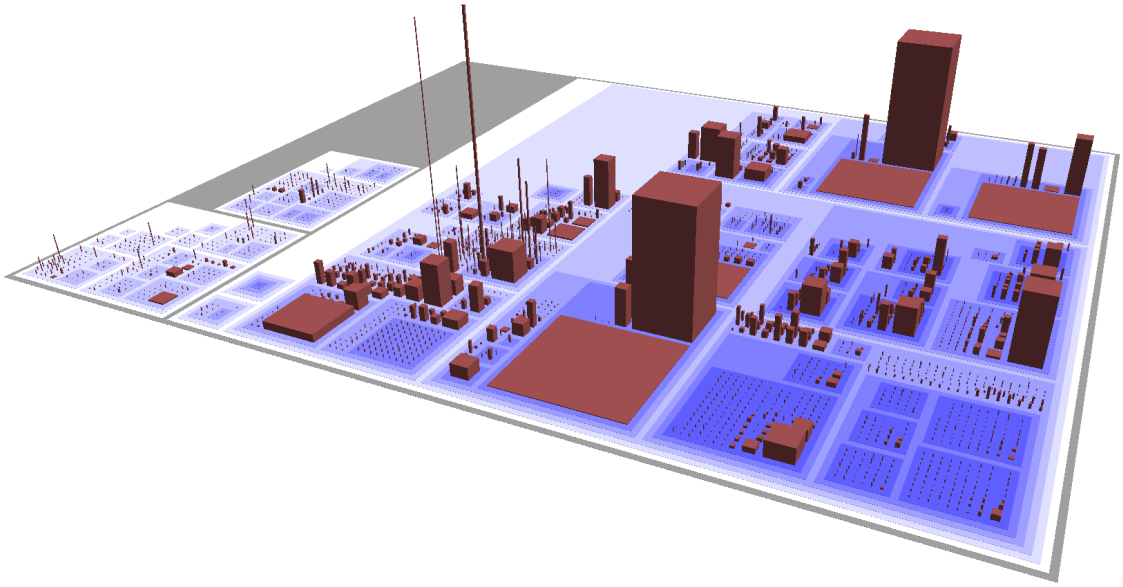


Figure 2: Example of a city rendered by CodeCity [11]

and also contradictory to the way one explores a city: A person does not begin to explore a real city by looking into particular houses.

However, they integrate the number of methods and attributes within the metric for the representation of the buildings (see Figure 2). The height of the buildings represent their number of methods (NOM), while the width and length represents the number of attributes (NOA). For instance, potential data classes (with very few or no methods and a lot of attributes) are easy to locate within such a complex scene as they look like wide plains with no significant heights, similar to parking lots. Wettel and Lanza [11] justify their two metrics by stating that, in reality, tall buildings are often associated with business, and therefore they map the NOM metric to the height to denote the quantity of functionality of a class.

Furthermore, the authors examine different mappings between software metrics and visual properties of the city elements. The linear mapping, for instance, results in a very large value range because the number of methods in classes can vary quiet widely. As a result of this mapping, the overview of the system can be compromised if there are classes with 10 and 1,000 methods, for example. The majority of the buildings, i.e., those with an average size or less, would hardly be visible since they are considerably smaller than extremely tall buildings (see Figure 2). A normalization of the heights solves this problem but a more serious problem persists: The viewer simultaneously perceives an extensive range of building sizes resulting in a sensory overload. As a consequence of this, Wettel and Lanza [11] introduce the boxplot-based and threshold-based mapping to reduce this kind of confusion and

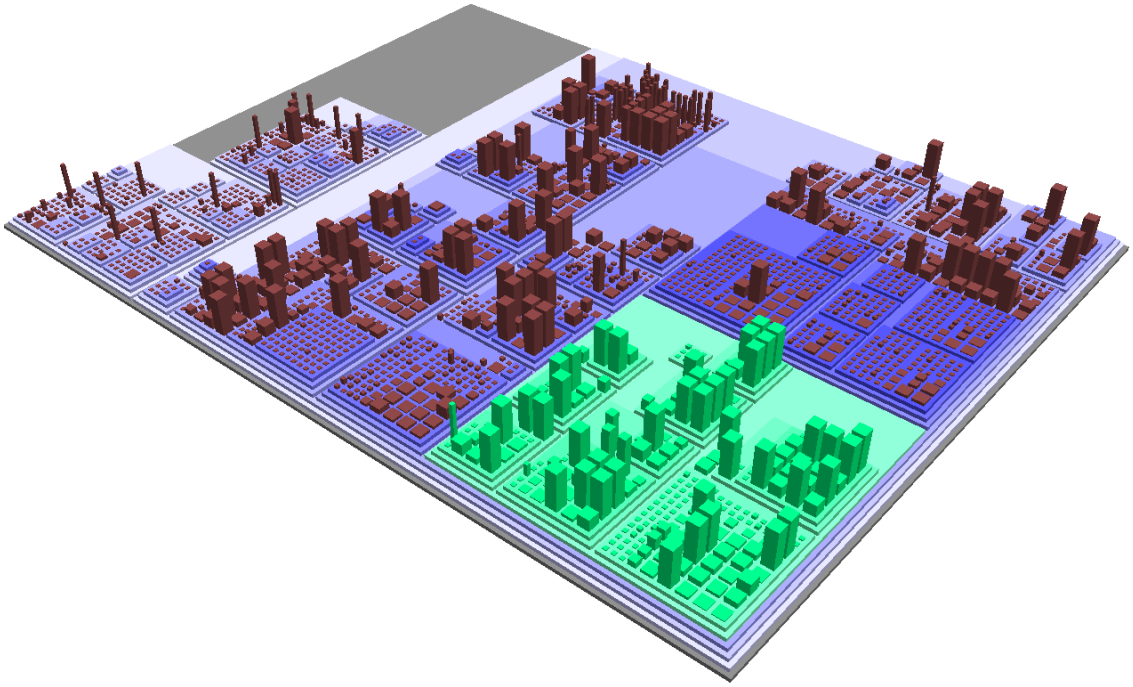


Figure 3: Selection of a district [11]

disorientation of the viewer. Instead of representing the NOM exactly, they categorize the heights of the buildings in 5 stages (very small, small, average, tall, very tall) depending on the NOM, whose boundaries are defined according to the chosen mapping strategy. The *boxplot-based mapping* balances the city because it produces boundaries relative to the values of the considered system and thus cannot be used to compare among cities. The *threshold-based mapping* overcomes this disadvantage by using well-defined thresholds which, however, can be hard to determine. Figure 3 and Figure 4 show examples of a city with a boxplot-based mapping and a clearly visible package topology.

2.2.2 Interaction & Navigation

Wettel and Lanza [11] distinguish between the vertical and the horizontal navigation mode. The former let the viewer be able to zoom in and out to inspect particular districts or buildings in more detail. It is also possible to navigate forward and backward, to hover left and right, and to orbit around the city. The latter navigation mode places the viewer into the city, in the midst of the buildings, enabling driving around the city. However, the authors intentionally prohibit to pass through buildings or go below the ground to support the familiarity of the viewer.

In CodeCity, one can select and interact with any software element or groups of elements. Wettel and Lanza support selection, spawning, tagging, and filtering.

For example, Figure 3 shows the selection of a particular district in ArgoUML. Furthermore, spawning different views of only particular parts of the system allows to continue the exploration in much more detail and without irrelevant information in terms of the current selection. To mark any building as important or less relevant, it is possible to tag a set of buildings, that is, to colour a selection or to use transparency. Figure 4 illustrates a combination of these tagging techniques to emphasize some elements (yellow and red-black buildings). The rest of the classes (violet buildings) are set to 60 per cent of transparency. For a better usability, Wettel and Lanza also provide a query engine to automatically search for elements matching a query, e.g., a particular name, type or category (all root classes).

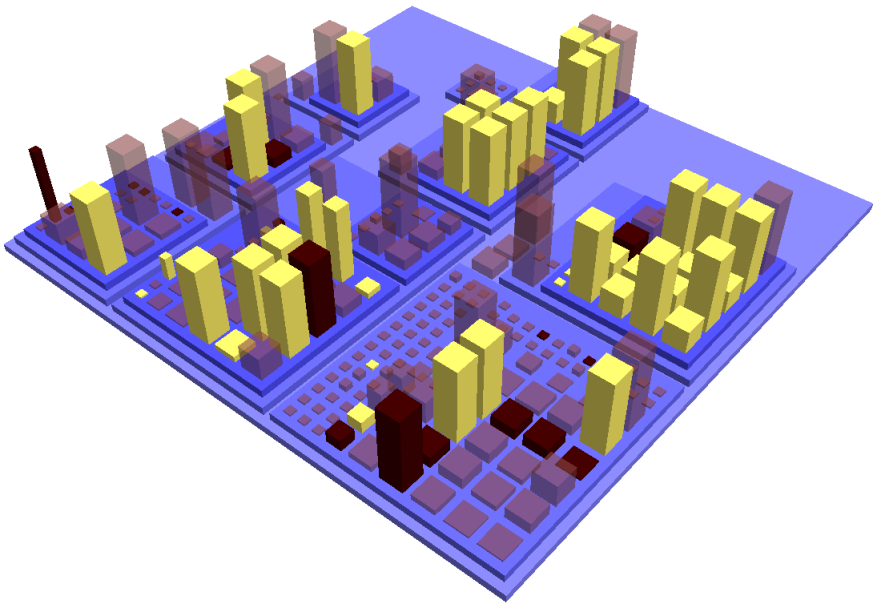


Figure 4: Color and transparency tagging [11]

The authors decide against rotating or moving elements of the city. They limit the capability of interaction, again, to not lead the viewer to disorientation. However, the viewer can very well perform multiple tasks, such as inspecting the model entity or accessing the source code of a selected class by right-clicking on it.

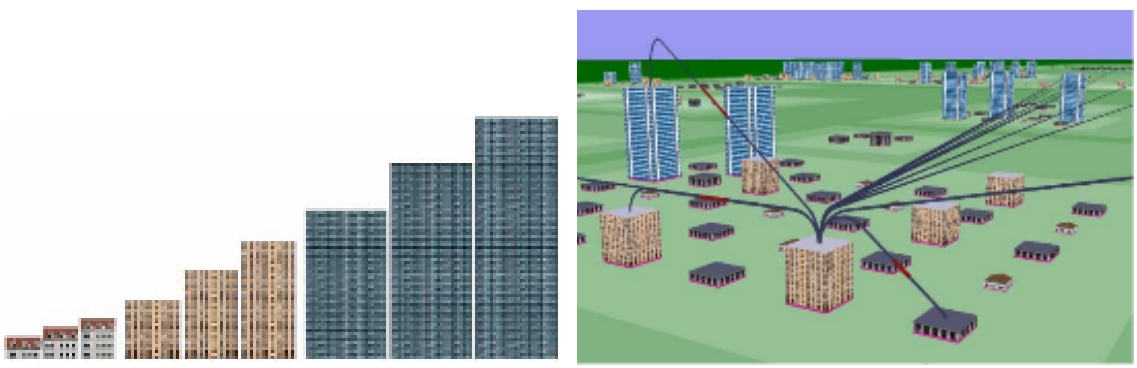
2.3 Different representations of an execution trace

Similarly, there has been some innovative work in visualizing dynamic behavior within a 3D environment. Below, we describe the two most promising approaches whose visualizations are based on the city metaphor and on a graph metaphor, respectively.

2.3.1 Dynamic visualization based on the city metaphor

Dugerdil and Alam [1] use the city metaphor to visualize the static structure in combination with an execution trace implemented in their tool *Evospaces*. They introduce a distinction between day and night where, during day time, Evospaces displays the structural and architectural information of the considered system similar to CodeCity [11]. At night, however, their tool represents program execution by highlighting active classes and showing the execution trace of monitored application flows. In the following, we describe their concept in more detail.

In day mode, they represent classes as buildings and depict relationships as solid pipes between the buildings (see Figure 5(b)). A moving red segment, displayed on each pipe, represent the direction of the flow of information. To prevent the linear metric problems mentioned in 2.2.1, the authors sort the metric[5] values in 9 categories, each represented as a particular building height in the 3D space. Furthermore, they map each block of three categories with a different texture to additionally distinguish among them (see Figure 5(a)). Rectangular zones on the ground represent the containment, that is, directories or package hierarchies, in different brightness values indicating the hierarchy level.

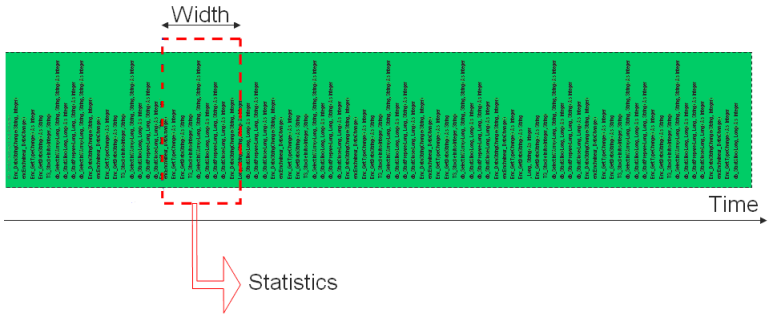


(a) Glyphs used in Evospaces (b) Relationships between classes

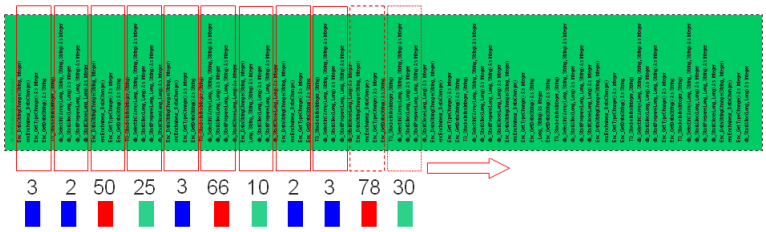
Figure 5: Day view of Evospaces [1]

At night, buildings represent classes as well as in day mode. However, solid pipes do not represent relationships between classes anymore, but represent an execution trace in two special views, namely the macroscopic and the microscopic view. The texture mapping is replaced by three different colors indicating the occurrences, i.e., the number of method executions (high, medium, low) within the trace. The sky, the ground and uninteresting classes within the 3D space are dark to emphasize the participating classes and the trace itself. The authors define uninteresting classes

[5]Dugerdil and Alam do not mention which metrics they use.



(a) The segmentation of the trace



(b) Class occurrences mapped to colors

Figure 6: The segmentation technique [1]

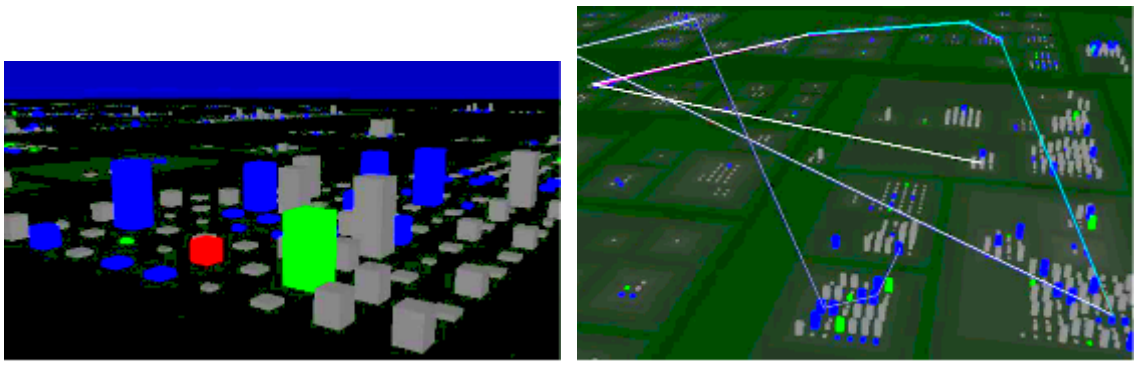
as not-present and temporally omnipresent[6] classes within an execution trace.

Dugerdil and Alam [1] use a segmentation technique to reduce the massive amount of information (here: method invocations) generated by a monitoring tool. They split the trace up into contiguous segments of a given duration (*Width* in Figure 6(a)) and compute statistics in each of the segment. According to the chosen view, either the sequence of segments or the sequence of single method calls in a selected segment represents the trace. Hence, they can depict thousands of software events in a movie that does not last hours, but has an appropriate total playing time.

In the macroscopic view, Evospaces plays a movie where each frame represent the state of the trace at that time, i.e., Evospaces displays the number of method invocations of each class by coloring the corresponding building as shown in Figure 6(b). Thus, active regions and the most active classes can easily be located at any moment in time (see Figure 7(a)).

The microscopic view enables to step through single method calls and in addition to that, to use an effect of visual persistence. The latter lets the user see all method invocations during a predefined number of frames. Figure 7(b) shows the microscopic view with a set of method invocations between classes where the luminosity of each call indicates the order of execution. The lower the luminosity

[6]Dugerdil and Alam define a class as *temporally omnipresent* if its presence is evenly distributed throughout the trace.



(a) Macroscopic view (b) Microscopic view with persistence

Figure 7: Night view of Evospaces [1]

level, the older in time.

2.3.2 Graph-based visualization

Greevy et al. [3] present a novel approach to the visualization of large execution information in the context of static structural information. They combine both, the dynamic behavior and the static structure of a system within a 3D environment displayed by their tool called *TraceCrawler*. It prerecords a static model of the system being examined, which can be easily generated by a variety of reverse engineering tools, e.g. Bauhaus[7], Rigi[8], and Moose[9]. Subsequently, *TraceCrawler* can process dynamic runtime data of that system and is able to show it post mortem, either step by step, be it manually or automatically, or all in one.

The authors use a graph-based visualization technique where nodes represent classes and objects, and edges indicate inheritance relationships and message interchanges. Figure 8 illustrates these visual components with a schematic view of the 3D visualization. Nodes at the very bottom form a plain representing the static structure of the software system, i.e., the class hierarchy. By implication, black edges connecting these class nodes intuitively illustrate inheritance relationships between two classes. The user can select three metrics from a range of metrics supported by TraceCrawler and map these to the length, width and color of the nodes. Nodes on top of a class node represent objects or rather instances of the underlying class, which have been created while monitoring the system. A red line between two instances of different classes illustrates a message interchange, i.e., a method invocation.

Greevy et al. distinguish two views using the representation last-mentioned,

[7] `http://www.iste.uni-stuttgart.de/ps/bauhaus/demo/index.html`
[8] `http://www.rigi.csc.uvic.ca/index.html`
[9] `http://moosetechnology.org/`

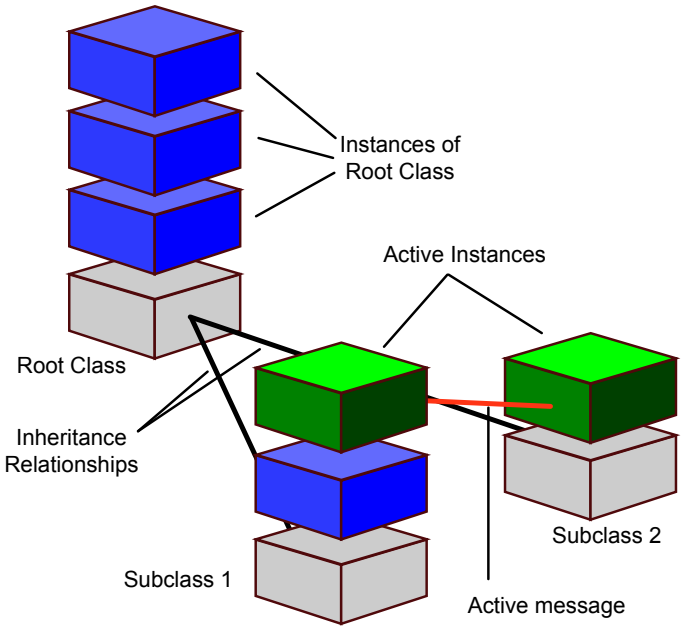


Figure 8: A schematic view of the 3D visualization [3]

namely the Dynamic Feature-trace View and the Instance Collaboration View. The former allows to step through the traces manually or as a movie whereas the latter gives a static, statistical overview of object-instantiations and message interchanges. However, in both views the time is not considered. The authors purposely do so because they focus on feature[10]-centric reverse engineering and not on supporting software development activities such as debugging and performance optimizations.

In more detail, the Dynamic Feature-trace View enables to move forward and backward within a trace of a feature at each point in time where the currently active objects and message are highlighted in green. Moreover, the user can examine the current state of the trace, for instance looking at the underlying source code, and is able to use a search mechanism to, e.g., search for a name of a method resulting in navigating to the next occurrence within the trace. Supportingly, a tree view is available displaying each event of the trace in the order it has occurred within the considered program.

The Instance Collaboration View depicts all events and object-instantiations after execution instead of showing a particular state of the program. It perfectly illustrates areas of activities (here called *feature hotspots*) in terms of towers of objects or merged lines of messages. In Figure 9 for instance, Greevy et al. present the login feature of the web-based object-oriented wiki system *SmallWiki* with more

[10]Greevy et al. define a feature as an user-triggered action, e.g., the sequence of all events resulting from a button-click.

than 4,000 events revealing the Response, Login and PageView classes as heavily communicating with other classes during the execution.

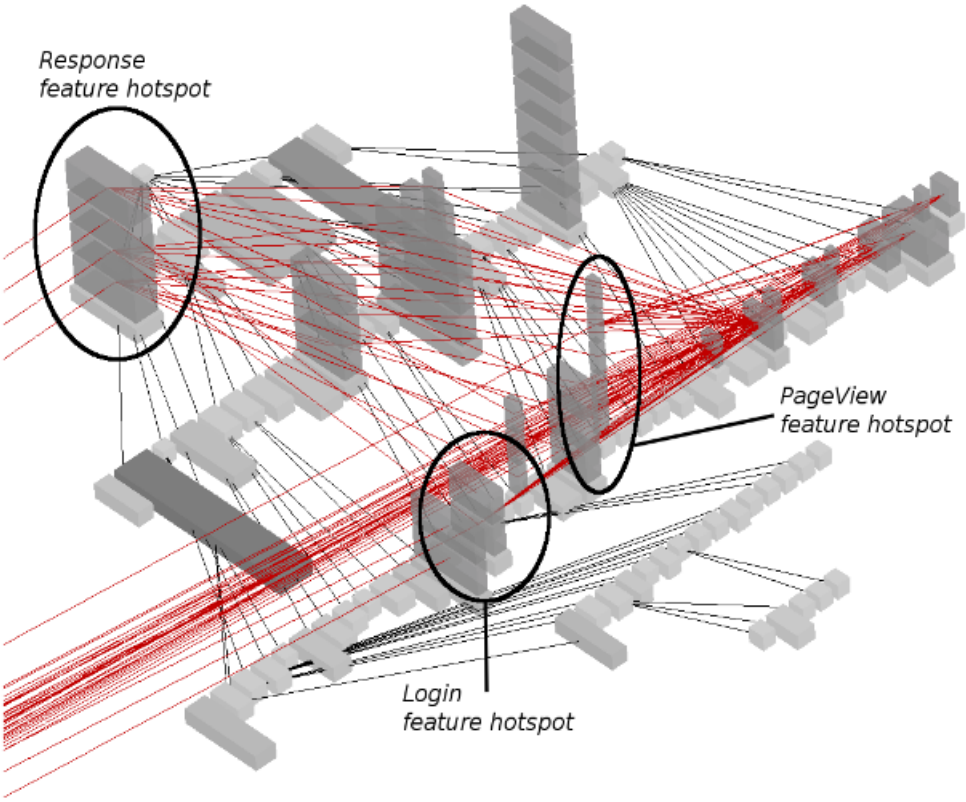


Figure 9: The Instance Collaboration View [3]

2.4 Eclipse Modeling Framework

Frequently, applications need a particular input or produce a special output format. In order to define such a format, one needs to specify a protocol or a meta-model for the data model. Moreover, data in the given format has to be read or written. For this purpose, in order to read data, an application must have a parser to extract tokens with respect to the given format. Subsequently, it must be able to group them into appropriate objects when using an object-oriented programming language.

Without having an special editor for this format, one does not only know what model elements to define, but also what grammar elements to use in order to specify the model elements. Furthermore, it is often necessary to update the meta-model specification leading to a recreation of the generated classes. Thus, individual code adaptations get lost.

The Eclipse Modeling Framework (EMF) [4] provides a generic parser, maintains individual code adaptations, is able to create an editor according to the format specification and offers a lot more useful features. EMF is a Java framework and code generation facility for building tools and other applications based on a structured model. It helps to quickly turn models into efficient, correct and easily customizable Java code. Most importantly, EMF provides the foundation for interoperability with other EMF-based tools and applications. Additionally, it can also transform a few other model formats into the EMF's model format *Ecore*.

EMF consists of the four components EMF.EMOF, EMF.Ecore, EMF.Edit and EMF.Codegen. The first allows to define simple meta-models by the use of object-orient concepts and provides the basis for EMF's Ecore model. Essential MOF (EMOF) is a subset of the OMG's MOF 2.0 specification[11] and gives restricted but easy access to the basic MOF elements. For more complex model definitions, please refer to the MOF specification.

EMF.Ecore represents the core framework including the Ecore model. The latter is the foundation of all individual models which can be created with EMF. EMF.Ecore provides basic generation and runtime support to create Java implementation classes for a model. Moreover, it offers several other benefits like model change notification, persistence support including default XML Metadata Interchange (XMI) and schema-based Extensible Markup Language (XML) serialization, a framework for model validation and a very efficient reflective API for manipulating EMF objects generically.

EMF.Edit extends and builds on the core framework, adding support for generating adapter classes that enable viewing and command-based editing of a model, and even a basic working model editor. The corresponding editor classes can be automatically and manually changed and adapt to the user's will to easily generate an individual editor. EMF.Codegen is responsible for creating such editor and model classes.

EMF uses XMI as its canonical form of a model definition, so there are several ways to define an ECore-based model. The first and direct way is to use a simple XML or text editor to create and describe the desired ECore-based model. Alternatively, the user can export the XMI document from a modeling tool, such as Rational Rose[12], and convert it to the EMF specification by using EMF's converter. It is also possible to define a model by annotating Java interfaces with model properties. The last approach is most applicable in creating an application that must read or write

[11] `http://www.omg.org/mof/`

[12] `http://www-142.ibm.com/software/dre/ecatalog/detail.wss?locale=en_US&synkey=O051890Y47758Q69`

a particular XML file format.

Figure 10 gives an overview of the complete ECore model. It supports standard elements, such as packages, classes, attributes and operations, as well as newer artifacts like enumerations and annotations.

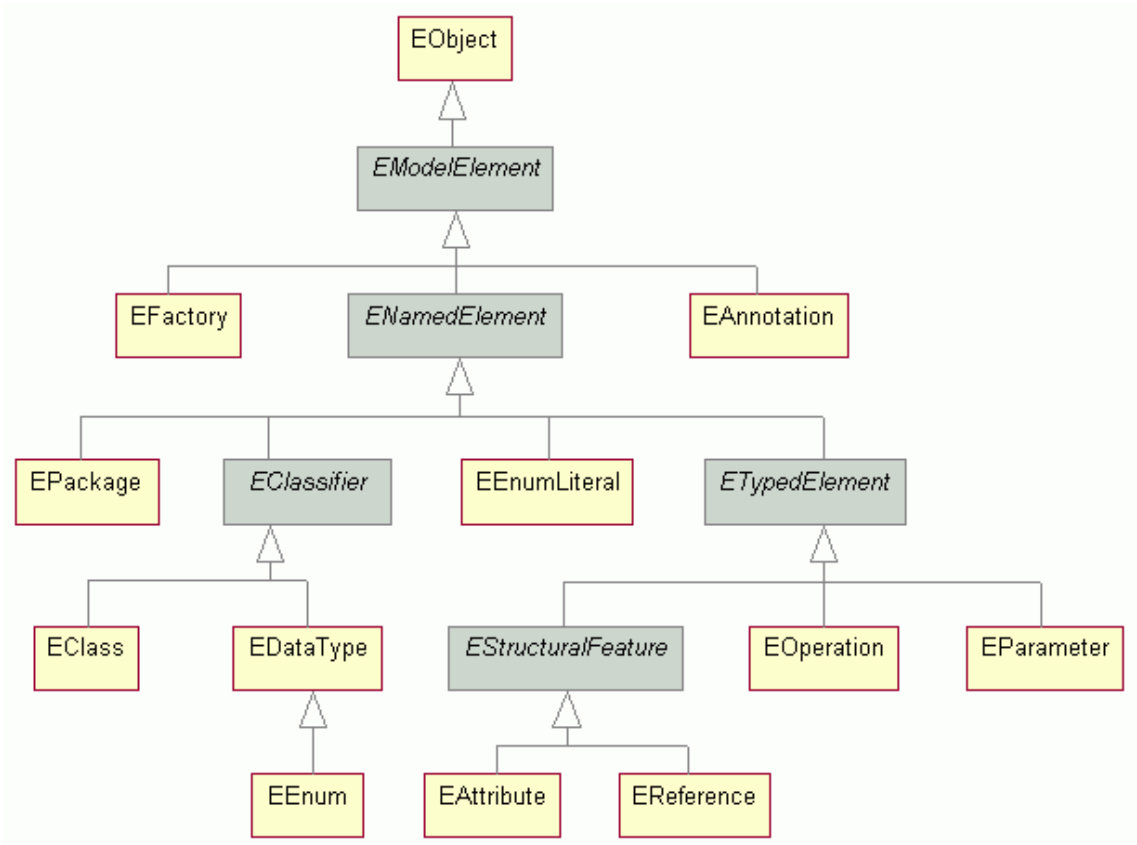


Figure 10: Hierarchy of the ECore model (shaded boxes are abstract classes) [4]

Once an EMF model is specified, the EMF generator can create a corresponding set of Java implementation classes. The user can edit these generated classes to add methods and instance variables and still regenerate from the model as needed. Figure 11 shows an example model with the generated `setAuthor` method of the `Book` class. EMF automatically creates routines and condition branches to preserve the two-way reference declared in the model. Due to the two-way reference, the old writer's reference `author` to the regarded book is removed in line 7 and, subsequently, the book reference is added to the new writer's book list. Finally, the book instance itself sets its new writer `newAuthor` at line 10.

(a) Model with a two-way reference

```
1   public void setAuthor(Writer newAuthor)
2   {
3   if (newAuthor != author)
4   {
5   NotificationChain msgs = null;
6   if (author != null)
7     msgs = ((InternalEObject)author).eInverseRemove(this, ..., msgs);
8   if (newAuthor != null)
9     msgs = ((InternalEObject)newAuthor).eInverseAdd(this, ..., msgs);
10  msgs = basicSetAuthor(newAuthor, msgs);
11  if (msgs != null) msgs.dispatch();
12  }
13  [...]
14  }
```

(b) The generated `setAuthor` method

Figure 11: Code generation example of EMF [4]

3 Evaluation of the approaches

In this section, we compare the approaches among each other and evaluate them. For this purpose, based on the approaches and the taxonomy of Shneideman [10], essential criteria in terms of 3D visualization were worked out. Table 1 shows these criteria in the first column. Below, we describe the bases of evaluation in more detail, i.e., the taxonomy and the evaluation table. Subsequently, we evaluate each of the four approaches on these bases.

3.1 Bases of evaluation

Criteria	Approach
Intuition	
Metaphor	*description*
Layout	2-D, 2½-D, 3-D
Navigation (incl. Zoom)	god-mode, bird-mode, central-mode
Filter	yes/no
Usability [10]	0-7
Mapping	
Scalability	high, medium, low
Metrics	LOC, NOM, NOA, NoMΔt, NoObj
Visual representations	object design, length, width, height, texture, color
Statics	
Packages	*criterion→visual representation*
Class attributes	*criterion→visual representation*
Class methods	*criterion→visual representation*
Inheritance	*criterion→visual representation*
Class associations	*criterion→visual representation*
Design patterns	*criterion→visual representation*
Dynamics	
Class instances	*criterion→visual representation*
Method invocations	*criterion→visual representation*
Duration of execution	*criterion→visual representation*
Online-Visualization	yes/no
Handling of large data	*description*

Table 1: Evaluation criteria

We classify each criteria according to the four categories *Intuition*, *Mapping*, *Statics* and *Dynamics*. The first category contains criteria that influence the orien-

tation of the user within the 3D space. The second category consists of the available input (metric) and output (visual representation) of the used mapping as well as a common scalability rating. The last two categories comprise the most important static and dynamic criteria and their corresponding mappings chosen in the given approach. After precisely describing each criterion of each category, Shneideman's taxonomy is explained.

3.1.1 Evaluation criteria

Intuition *Metaphor* describes the type of the used metaphor, for instance, a graph-based metaphor or a city-based metaphor. *Layout* indicates the number of utilized dimensions in space. In particular, $2^1/_2$-D means that all three dimensions are in use, but there exists a ground area where nothing is displayed below. Generally, the layout depends on the chosen metaphor.

Navigation defines the view as well as the movement and zoom ability the user owns within the given space of the approach. We choose the suffix *mode* in order to describe all these three different characteristics in one word. Here, god-mode indicates the ability to arbitrarily move in space, particularly through objects, and change the viewing direction and position in the scene to the ones the user likes. The bird-mode limits this navigation capability insofar as it provides a fixed viewing direction from the top downwards to the scene. Furthermore, it is impossible to navigate through objects. The central-mode has much stricter constraints and is mainly used in combination with the city-metaphor. In this mode, the viewer does not look down to the scene, but is directly located within the center of the scene. Moreover, the user can indeed move towards the cardinal points, he/she is, however, not able to increase the altitude. Although, none of the four approaches, we evaluate, uses the central-mode, there are a few tools [6, 8] that do use it. We just mention it for the sake of completeness.

Filter simply indicates whether or not the approach supports a filter feature in order to extract and therefore focus on selectable scene objects. *Usability* describes the amount of Shneideman's criteria [10] the given approach fulfills. For more information about the criteria, please refer to Section 3.1.2.

Mapping *Scalability* roughly describes the display behaviour of the approach's tool concerning small, medium and huge amount of data. For example, a high value indicates a continuing intuitive, structured illustration of even a huge quantity of data. *Metrics* represents the metrics and *Visual representations* represents the corresponding visual entities the approach uses. The meaning of NoMΔt is similar

to the one of NOM. Instead of considering the whole duration of a trace, the number of methods is measured within a particular time interval, usually to produce more fine-grained statistics. NoObj stands for the number of objects, i.e., the number of instances of a class. Object design is rather a placeholder for a description of the design of the visual representation than a visual entity such as width.

Statics For each attribute, a mapping function describes which visual representation illustrates which criterion. For example, be the criterion *package* in category *Statics*. Then, a corresponding mapping function can look like package→box. This means that a box illustrates a package in this concrete visualization tool. Although none of the approaches, discussed here, (can) represent *design patterns*, we include this criterion because it is an important aspect of the architecture of a software system. Displaying design patterns could lead to a better comprehension due to higher abstraction.

Dynamics Each of the first three attributes has a mapping function equally defined as the ones in the category *Statics*. However, none of the approaches, discussed here, (can) represent the *duration of execution* of one or more methods within a given trace. Again, we include this criterion due to its importance for dynamic analysis and to refer to our prototype implementation in Section 5 that supports displaying this information. *Online-Visualization* specifies whether or not the approach supports continuous visualization of a running system. *Handling of large data* describes the used method(s) to compress or filter the huge amount of monitoring data in order to produce reliable statistics or to obtain specific information.

3.1.2 Shneideman's taxonomy

Since graphical user interfaces can be quiet complex, Shneideman [10] proposes a type by task taxonomy of information visualization in order to design intuitive, easily understandable, advanced graphical user interfaces. He extends the Visual Information-Seeking Mantra *overview first, zoom and filter, then details on demand* by adding the three criteria *Relate, History and Extract*. By doing so, he does not just describe a useful starting point for designing graphical user interfaces, but rather a complete methodology and, more important for our purpose, a good basis for evaluation. More precisely, we use his taxonomy to evaluate a part of the intuition of each approach's tool, namely the usability as a whole.

> *"Information exploration should be a joyous experience."*
> Shneideman [10, p. 336]

Overview Gain an overview of the entire collection. An overview helps to orientate oneself in the given domain and to find a particular area of interest. Hence, overview strategies include zoomed out views, each with a movable viewing area. By means of such a field-of-view box, the user can control the display contents.

Zoom Zoom in on items of interest. Users typically have an interest in some special portion of a collection and, therefore, need to control the zoom focus and zoom factor. Additionally, smooth zooming helps them to preserve their sense of position and context. A very intuitive way to zoom in is by pointing to a location and issuing a zooming command, e.g., cf. Google Maps[13].

Filter Filter out uninteresting items. Dynamic queries applied to items in the collection, prepared or user-definable, allow to eliminate unwanted items so that users can quickly focus on their interests. Shneideman [10] additionally proposes an advanced, graphical filtering mechanism. However, we do not want to go into detail here because we focus on other aspects. Nevertheless, his purpose could be implemented as an extension to our prototype.

Details-on-demand Select an item or group and get details when needed. Once a collection has been focused, trimmed to a manageable amount of items and zoomed in, it should be easy to browse the details about these individual items. The usual approach is to click on an item and get a pop-up window with specific information of this item.

Relate View relationships among items. After selecting an item, it should be possible to let automatically highlight items with similar attributes, relationships etc. These user interface actions can improve understanding of complex relationships among items.

History Keep a history of actions to support undo, redo and progressive refinement. It is rare that a single user action produces the desired outcome. Information exploration is inherently a process with many steps, so keeping a history of actions allows to retrace these steps. Thus, steps can also be combined or refined.

Extract Allow extraction of sub-collections and of the query parameters. Saving the current view or state to a file in one or more formats enables to reload this view

[13] http://maps.google.de/

automatically. Moreover, extraction facilitates to work cooperatively, to integrate a special viewing area as an image in a presentation and to print.

3.2 UML

Since there are many tools that implement the UML differently, we do not evaluate a specific tool. Instead, we review the potentiality of the UML.

Intuition

UML uses several metaphors, but all are mainly graph-based. Despite of the lifelines, a sequence diagram consists of boxes, representing instantiated classes, and lines, representing message interactions. These boxes and lines are simple node and edge representations. Also, a package diagram contains boxes, representing packages, and lines, representing associations among one another. In a use-case diagram, ellipses represent use cases that are connected by lines which in turn represent associations among those use cases. Even here, the ellipses are just another type of node representation. Only the stick-figure seems to be interpreted as a humanoid figure. To display each diagram, UML uses 2-D layouts. Hence, the user can navigate best in bird-mode. The line of sight can be set to point directly down to the diagram because there is no third dimension, i.e., no height to be considered. By zooming in and out, it is possible to get an overview of the current diagram or focus a particular subregion of it. As an extension, for example, the user could zoom through a component in a component diagram so that the corresponding package diagram of this component is displayed then. Following this idea, this could be recursively done by zooming in and out different levels of abstraction. UML also provides possibilities to color, tag and filter specific diagram elements. The latter, however, would have to be implemented by generating a new diagram with the selected elements out of the whole diagram. In this manner, a tool based on the UML could satisfy all criteria of the taxonomy of Shneideman [10] and would therefore be optimal in respect of usability.

Mapping

UML's great amount of different diagrams allows to display several different architectural and dynamic views of abstraction. Although—or because—some diagrams cannot directly be reconstructed out of the source code, they are able to depict higher levels of abstraction. In this manner, a combination of diverse types of reconstructed and manually created and edited diagrams can lead to a high scalability.

However, a single diagram does not scale well. For example, a sequence diagram already becomes too complex and thus confusing, when displaying 15 participants or 20 method invocations. As several authors of the approaches describe (cf. Section 2), their tools record and successfully illustrate traces with hundreds of participants and thousands of method invocations.

UML does not directly make use of metrics in any of its diagrams. A class diagram, for instance, only contains the metrics NOA and NOM indirectly by listing the concrete attributes and methods in each class box. The size of a class box, i.e., the length and the width, and the length of an association line (or arrow) as well as the distance between classes among each other have no meaning according to the specification. Admittedly, UML is designed to depict packages, classes, and methods in detail. Hence, the size of all these scene objects result from the length and amount of textual descriptions.

UML uses simple geometric forms, such as 2D-boxes, lines, and diamonds, as well as several kinds of arrows to represent a software system. To highlight such a scene object or to indicate a special regions, it is possible to use colors and to tag by means of comment fields. Furthermore, box entries, which are parts of a scene object, contain textual descriptions of, e.g., attributes, methods and associations. Apart from these simple, quiet abstract scene objects, UML provides several good views and interaction possibilities to represent a software system in more detail as well as to modify or commonly work with such a representation.

Statics

In a package diagram, UML represents the low-level package structure by means of rectangular containers. Additionally, it is possible to depict not only dependencies but also other relationships among packages. Most importantly, UML supports displaying class attributes, class methods and all kinds of association types in one single diagram, namely the class diagram. Since package and class diagrams can be combined or used cooperatively, UML provides a very detailed view on the statics of a software system.

Dynamics

In order to represent dynamics aspects of a system, sequence diagram are common in use. A sequence diagram displays in detail message interactions and trace participants, e.g., method invocations and class instances, respectively. Again, UML provides a detailed view on the dynamics of a software system, but lacks in an appropriate overview. For this reason, the OMG proposes the interaction overview

diagram that can include references to all available interaction diagrams, especially sequence diagrams, in one single diagram.

However, since an interaction overview diagram is intended to illustrate a complex process of more than one use case or event, it does not help in depicting a single action. If the latter additionally consists of hundreds of method invocations, even a sequence diagram produces an unclear, confusing and hard readable result.

UML does not directly support displaying durations of execution. In return, tagged comments could be used as workaround. Due to the problematic of representing even a minor amount of dynamic data, we do not propose UML as basis for online-visualization at this moment. But when someday a newer version of UML provides an appropriate diagram for such dynamical requirements, it would be easy to implement both techniques for handling large amount of data and support for online-visualization. Such features are tool-specific and therefore independent of any (UML) specification.

3.3 CodeCity

Intuition

CodeCity [11] is based on the city metaphor and thus, it uses a $2^1/_2$-D layout. The compromise between 2D and 3D produces the impression of both information overview and intuitive orientation in space together in one. At first, the user can roughly look at the city representation of a chosen application by just focusing on the districts, i.e. without looking at the buildings or other details. After getting an overview, the user can zoom in into a particular subregion and is able to examine more fine-grained representations. Due to the ground layout, there is always an orientation point in space and the entity *height* is well-defined. Hence, the user always knows where the top and the bottom are. The navigation in bird-mode also supports the user's orientation in this three-dimensional space by introducing constraints that limit the user's capabilities. CodeCity does not permit to change the viewing direction or to move through buildings. The former avoids to look above and thus losing sight of the scene. The latter adds the realism to the city the user expects and knows from the real world. Additionally, the filter feature improves the usability as the undesirable regions are filtered or faded out. Therefore, only the objects of interest are visible or highlighted.

According to Shneideman [10], CodeCity satisfies 4 of 7 criteria of his taxonomy, namely Overview, Zoom, Filter, Details-on-Demand.[14] In this manner, it has

[14]Wettel and Lanza [11] do not mention whether or not CodeCity comprises a redo/undo feature

an average graphical user interface in terms of usability.

Mapping

In context of scalability, Wettel and Lanza [11] mention that their tool CodeCity scales up to industrial-size software systems, such as ArgoUML (over 2,500 classes, 130+ kLOC), Azureus[15] (over 4,500 classes) and VisualWorks[16] (over 8,000 classes). It does intentionally not support the LOC-metric or a linear metric-mapping. Instead, CodeCity uses the metrics NOM and NOA as well as a metric-based categorization system to represent a system's architecture. Hence, the authors have chosen the most important metrics in terms of software architecture.

Furthermore, it does not only utilize the three dimensions but also makes use of colors that help to quickly categorize subregions or focus on a particular area of interest. Admittedly, the number of attributes are mapped to both the length and the width of a building so that CodeCity effectively uses mere two dimensions. However, it is difficult to distinguish between length and width in a three-dimensional space. For example, if the user rotates the view by an angle of 90 degrees, the meanings of length and width are swapped. For this reason, Wettel and Lanza [11] have found a good solution with this mapping.

Statics

CodeCity represents the low-level package structure by stacked platforms at different altitudes. It shows the hierarchy in a well-integrated, not overwhelming way. Truly, the package structure is important indeed, especially in order to get an overview at the first phase of exploration, but a system's behavior is based on their classes and relationships among themselves. Thus, in our opinion, the authors have correctly settled on a subtle but expressive representation. Also the attribute and method mappings are well-chosen. Since tall buildings are often associated with business, Wettel and Lanza [11] note, they map the NOM metric on the height to denote the amount of functionality of a class. However, CodeCity does not depict inheritance or other types of relationships between classes, although there is enough potentiality, i.e. space and human perceptivity, for more representative scene objects (c.f. Evospaces implementation of relationships).

---

and a save/load mechanism to export the current view and load a saved view.

[15] `http://azureus.sourceforge.net/`

[16] `http://www.cincomsmalltalk.com/userblogs/cincom/blogView?content=visualworks_info`

Dynamics

As CodeCity is, so far, intended to display the statics of a software system only, it does not support any dynamic features mentioned in the criteria table.

Conclusion CodeCity is an user-supporting tool to visualize the low-level structure as well as the number of methods and attributes of even a huge number of classes. This 3D-approach is a good choice for locating data-classes and function-classes without having an extra detection algorithm.

3.4 Evospaces

Intuition

Evospaces [1] is based upon the idea of Wettel and Lanza [11], i.e., it uses the city metaphor. Hence, it uses a $2^1/_2$-D layout to arrange buildings and districts on top of a visible ground. It also makes use of the bird-mode in oder to navigate and zoom in space. However, Dugerdil and Alam [1] do not mention if their tool supports transparency or is able to spawn a new view with only the scene objects of interest. Therefore, Evospaces does not provide any type of filter.

In context of usability, it completely satisfies 3 of 7 criteria of Shneideman [10], namely *Overview*, *Zoom* and *History*. Apart from the criteria *Extract* and *Filter* that are discussed above, Evospaces does not match the criteria *Details-on-demand* and *Relate* either. It gives no additional information when selecting a particular scene object.

Mapping

Dugerdil and Alam [1] apply their tool to Mozilla in order to test scalability and usefulness. However, they do not mention what product of Mozilla they use. In return, they claim that their tool is able to display a substantial part of a very large system in combination with an execution trace. They use unknown and fixed metrics for the height and texture mapping. The length and width are equal and also fixed for all buildings. The user can only define the threshold values and colors for the occurrence categories low, mid and high. In other respects, Evospaces does not let the user select any metrics or visual representations. It illustrates a software system as a city with districts, buildings and solid pipes similar to what CodeCity does.

Statics

Evospaces represents packages and classes as districts and buildings, respectively. Indeed, Dugerdil and Alam [1] do not mention what metrics they use for the heights and textures, but it is likely to be NOM and NOA. Ultimately, they describe the most important properties of a class. Hence, comparing Evospaces in day view with CodeCity, Evospaces uses textures instead of mapping a metric to the surface area of buildings. In return, CodeCity's buildings only have one global color—if not manually highlighted—and Evospaces's buildings all have the same area size. Textures are more realistic than one global color and therefore, in our opinion, more intuitive for a human. However, different, simple colors rather have the ability to highlight and focus on specific subregions and buildings than textures are able to do. Dugerdil and Alam [1] choose the texture-mapping for the day view and the color-mapping for highlighting the participants of a trace in the night view. Thus, in the sense of a representation of the statics of a software system, the authors propose a better representation than CodeCity's author do.

Evospaces generically displays relationships between classes as solid pipes between their rooftops. From our point of view, Dugerdil and Alam [1] have well integrated relationships in the statical, three-dimensional view. Admittedly, they could have specified them more precisely to distinguish between, e.g., inheritance/-generalization, aggregation, and association relationships.

Dynamics

Unlike TraceCrawler (see below), Evospaces does not provide a representation for class instances. However, Evospaces could not even use TraceCrawler's representation. It maps the NOM to the height of each class representation whereas TraceCrawler displays each class as a flat 2D-rectangle and places instances as stacked floors on top of the corresponding class. Admittedly, Evospaces could at least show the number of instances of each class at each point in time when using the microscopic view, i.e., stepping through each event within one segment.

Illustrating a software system and corresponding traces by introducing day and night is a good extension to the city metaphor. As representing all important statical and dynamical information in one single view would lead to sensory overload, Dugerdil and Alam [1] choose an absolutely appropriate compromise. Luminance varying solid pipes depict method invocations in chronological order and non-black colored buildings symbolize participants of the given program trace.

Both the macroscopic and the microscopic view use this night representation to display a segment and a sequence of events of one specific segment, respectively.

For this, the authors use a segmentation technique that split a trace up to contiguous segments and then computes statistics in each of the segments. Again, they propose a good approach to give an overview of the given trace first and then to inspect a particular segment in detail. Also, their filter that fades out non-participants and temporally omnipresent classes to focus on the current trace is a good support.

Conclusion Evospaces successfully integrates textures and dynamic data into the city metaphor. Especially the segmentation technique in combination with the macroscopic and microscopic view seems to be promising. In addition to analyzing the static architecture, this 3D-approach is therefore a good choice to locate participants and to depict the order of method calls of a given execution trace. It is, however, not useful for identifying bottlenecks or analyzing a trace in more detail due to the lack of information about time aspects.

3.5 TraceCrawler

Intuition

TraceCrawler is based on the graph metaphor because it uses simple boxes that are linked by lines. It arranges these basic elements in a three-dimensional space with a ground resulting in a $2^1/_2$-D layout. However, TraceCrawler does not draw this ground explicitly. It only displays all classes at the same altitude. Hence, by looking at the graph from beneath the ground, the user can see through it. Additionally, the user is even able to move through it. So, although a ground definition usually prevents from losing orientation in terms of top and bottom, this ground definition lacks in important properties in order to improve orientation in space.

Concerning the navigation, the tool allows to navigate in god-mode so that the movement and viewing direction is arbitrary. However, this mode can lead to disorientation, especially in a 3D space. For instance, the ability to move through boxes can confuse while moving and zooming. While visualizing, TraceCrawler does not support a filter mechanism to focus on a specific feature or to filter out a set of classes and their instantiations. The only way to visualize a special subset of features is to solely record the traces of these features by instrumenting them in the source code. Furthermore, apart from defining the three metrics for the class nodes, it does not allow the user to edit the scene objects, e.g., to tag or to color them. In this manner, it does not provide any filter.

Thus, in context of the taxonomy of Shneideman [10], TraceCrawler provides an overview insofar that the user can zoom out until the whole scene is visible.

Admittedly, this overview does not include a categorization view, e.g., a package view, to display a more abstract level. Shneideman, in turn, does not define the overview in terms of level of abstraction. Thus, TraceCrawler completely satisfies the first criterion. As mentioned before, TraceCrawler does not support filter. In the Instance Collaboration view, the pop-up window of a selected class gives details of this class' properties and direct access to the corresponding source code. Due to the nature of the representation, TraceCrawler depicts statical and concrete dynamical relationships. Yet it is not possible to automatically highlight a group of scene objects with similar properties as defined in Shneideman's criterion *Relate*. On the righthand side of the Dynamic Feature-trace view, there is a history of the feature's trace that enables to select and display an arbitrary position in the trace. Greevy et al. [3], however, do not mention whether or not their tool supports an undo/redo functionality. TraceCrawler does not support saving and loading a particular view, probably because the user can hardly edit the view. In this manner, it completely satisfies three and partially two of the seven criteria of the taxonomy of Shneideman [10].

Mapping

Greevy et al. [3] have successfully applied their tool to feature traces of SmallWiki and Moose[17]. In one case study, examining the former, they choose five features that involve 8.000 interactions among the instances of 280 different classes. In the other case study, examining the latter, the authors choose feature traces that consist of over 70.000 events among objects of 780 various classes. Especially, the latter case study demonstrates a high scalability.

TraceCrawler illustrates each instance of a class as stacked 3D-boxes by using the integrated number of objects (NoObj) metric. It allows to pick metrics in order to map them to the length, width and color of these boxes. This a significant advantage over CodeCity with its fixed, not user-selectable metrics. The user can choose out of several metrics, such as NOM or NOA, to individually design and arrange the software representation.

In order to distinguish message edges from inheritance edges and to highlight active processes, TraceCrawler uses non-changeable colors. It also makes use of colors to distinguish classes from instances and to categorize the frequency of runtime events. In this manner, TraceCrawler is able to highlight particular objects of interest and only lacks in textures.

[17] `http://www.moosetechnology.org/`

Statics

TraceCrawler simply represents the statics and dynamics of a system by using 3D-boxes and lines. It lacks in a grouping representation such as a package hierarchy and does not depict associations between classes, apart from inheritance associations. However, Greevy et al. [3] do not intend to use their tool for a complete or complex static representation, but to visualize and analyze the dynamic behavior of a software system. Thus, in our opinion, TraceCrawler may lack in a more detailed static representation. Admittedly, the authors could have implemented at least the low-level package structure. In return, as mentioned before, the user can freely select metrics for the class attributes and methods.

Dynamics

Greevy et al. [3] intend to depict the dynamics with their tool. Therefore, it is able to represent class instances and method invocations. TraceCrawler symbolizes the former as colored, stacked 3D-boxes and the latter as simple lines between two 3D-boxes. By doing so, it is possible to inspect in detail each step in a trace. However, it can be quiet "wired" and thus lead to confusion. A complementary solution could be a more abstract view that combines messages from one source to instances of the same class, as a bus combines multiple data wires on a hardware chip.

TraceCrawler does not show the response time and duration of execution of a method invocation and a trace, respectively. Additionally, it does not support online-visualization. Both are important to analyze a system's behavior in more detail. As TraceCrawler is in an experimental phase, Greevy et al. [3] have not implement a dynamic filter yet. Thus, instead instrumenting the entire system and filtering out the unwanted information, they statically instrument only such subsystems they want to analyze. However, they mention that they plan to collect traces via selective instrumentation and post-filtering.

Conclusion Different from others, TraceCrawler also represents class instances besides method invocations. Thus, it is possible to analyze a software system's behavior in more detail than Evospaces. Additionally, it illustrates inheritance relationships, too. However, it uses the less intuitive graph metaphor. Hence, TraceCrawler is a good choice to highlight hotspots.

3.6 Conclusion/overview

In this section, we reviewed four approaches in context of static and dynamic architectural views of a software systems. We chose exactly these four approaches because each contains special concepts and ideas that others do not. Hence, each one constitutes a representative of another, different equivalence class.

In this manner, UML stands for all two-dimensional approaches. CodeCity represents each tool that depicts the statics of a software system as a city in an three-dimensional environment. On the other hand, TraceCrawler describes a graph-based 3D-representation with the focus on the dynamical architecture. Evospaces is an representative for tools that try to combine both the statical and dynamical aspects of a software system in context of the city metaphor.

We are aware that we describe and compare tools, having a purely statical view, with tools that almost exclusively focus on a dynamical view on a software system. However, all approaches show individual concepts and ideas that DyVis implements or should implement in future. Table 2 gives an overview of the approaches and their criteria described in Section 3.1.

In conclusion, using a three-dimensional environment, all authors propose a $2^1/_2$-D layout for optimal orientation in space. Full 3-D layouts, as Vizz3D [9] uses, are becoming very rare in the domain of program visualization. For moving without causing confusion, the bird-mode seems to be the best choice. The user can still navigate in all directions but is not able to change the viewing direction or to move through scene objects. In context of usability, all tools support *Overview* and *Zooming*. Apart from that, the tools, however, differ in several, different criteria of Shneideman [10]'s taxonomy. Admittedly, intuition is probably not important for a scientific research project so that the authors do not always emphasize on usability.

In respect of category *mapping*, all approaches show a high scalability or potentiality for high scalability in terms of today's requirements. However, they are also designed for such complex tasks. This comparison shows that there are appropriate compression techniques and representation methods to handle and display such a huge amount of statical and dynamical data.

Criteria	UML	CodeCity	TraceCrawler	Evospaces
Intuition				
Metaphor	graph-based	city	graph	city
Layout	2-D	2½-D	2½-D	2½-D
Navigation (incl. Zoom)	bird-mode	bird-mode	god-mode	bird-mode
Filter	yes	yes	no	yes
Usability [10]	7	4	3	3
Mapping				
Scalability	high	high	high	high
Metrics	-	NOM, NOA	NoObj, user-selectable	2x unknown, NoMΔt
Visual representations	container, box, box entry, line, arrows, color	district, building, length, width, height, color	3D-box, line, length, width, color	district, building, solid pipe, texture, height, color
Statics				
Packages	package→container	package→district	-	package→district
Class attributes	attribute→box entry	attribute→length&width	-	unknown
Class methods	method→box entry	method→height	-	unknown
Inheritance	inheritance→line	-	inheritance→line	unknown
Class associations	association→line	-	-	association→solid pipe
Design patterns	-	-	-	-
Dynamics				
Class instances	instance→box	-	instance→3D-box	-
Method invocations	invocation→line	-	invocation→line	invocation→solid pipe
Duration of execution	tagged comment	-	-	-
Online-Visualization	-	-	-	-
Handling of large data	ind. of UML	-	filter	segmentation & filter

Table 2: Evaluation overview

3.7 Eclipse Modeling Framework

EMF provides a feature-rich development environment for defining model specifications and generating corresponding Java classes automatically. When specifying a new ECore-based model by means of the integrated default graphical user interface (GUI), an ecore-file stores all necessary model information. With this file, one can quickly instruct EMF to produce a genmodel-file that in turn offers fast access to automatically convert the model into equivalent Java classes and to automatically generate a default model instance editor.

The integrated default GUI easily allows to specify a model by using a tree structure. Since EMF's basic *ECore* model is strictly object-oriented, the tree nodes depict packages, classes, attributes etc. Thus, they are similar to the corresponding Java classes that can be generated by means of the genmodel-file. This direct relation to the object-oriented paradigm facilitates using the GUI.

When defining a model, EMF lets the user specify the multiplicities of an association between two classes. By setting the property *EOpposite* of an EReference, EMF's generator produces corresponding Java setter and getter methods that preserve the multiplicities (Figure 11). However, this feature is not unique to EMF. The Java Architecture for XML Binding (JAXB)[18] is also able to generate correct multiplicities.

Moreover, besides the default EMF data types, one can use any other Java class as type of a variable. By declaring a new *EData type*, i.e. an alias name, with a full reference name to the desired Java class (including the package hierarchy), EMF accepts external data types. Again, this feature is also available by JAXB.

Furthermore, EMF can keep individual changes in Java source files after regenerating them out of the model definition. By removing the default Javadoc comment *@generated*, the body of the given method will not be overwritten anymore. For this reason, when using a global repository such as the version control system SVN[19], not only the ecore-file but also the generated and modified source files should be contained in that data storage.

The generatable editor lets the user easily and quickly define a model instance with respect to the model specification. Additionally, since the editor source code is available, it is fully customizable. Hence, this model instance editor is another helpful EMF tool.

In conclusion, EMF greatly supports and facilitates the definition and distribution of model specifications. Although JAXB has many of EMF's features as well,

[18] https://jaxb.dev.java.net/
[19] http://subversion.apache.org/

EMF currently provides several additional unique features. However, only a few of them were required for developing DyVis. In return, EMF provides all features that JAXB provides and additionally, its ECore model is an industrial standardization.

4 An extended city metaphor

In order to visualize our own approach in terms of the city metaphor, we need a model specification of all the elements that should be displayed. For this purpose, EMF was used. Below, we describe in detail this metaphor model that our prototype implementation DyVis uses. Figure 12 shows the metaphor model as class diagram.

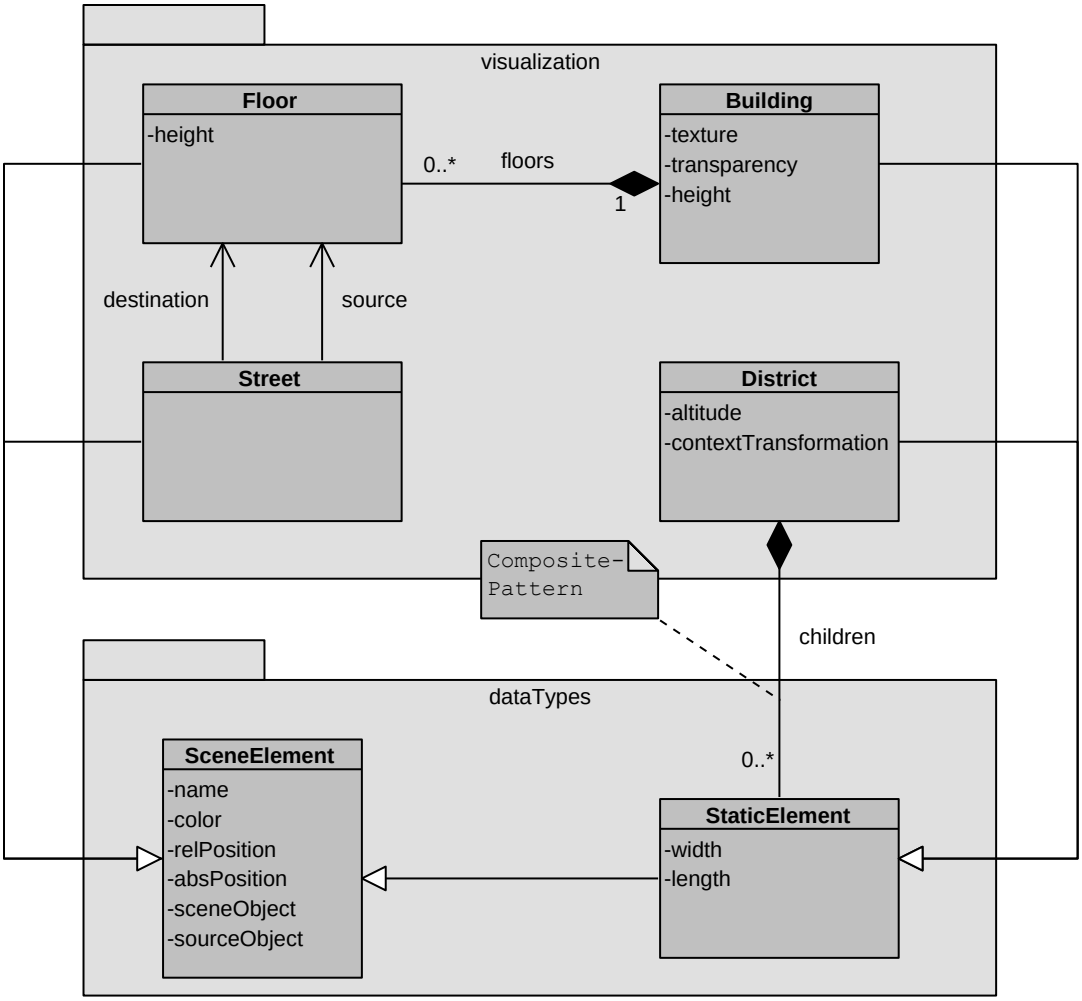


Figure 12: Class diagram of the metaphor model

4.1 Overview

The metaphor bases on the city metaphor Wettel and Lanza [11] propose. However, due to the purpose of recognizing performance problems, we include the approach of Greevy et al. [3] as well. Hence, the resulting metaphor model consists of a combination of *CodeCity* and *TraceCrawler*.

We decide against the approach of Dugerdil and Alam [1] because their tool *EvoSpaces* displays method invocations from the top of one to another building. Thus, it is incompatible with *TraceCrawler*'s metaphor model where an arbitrary number of stacked class instance boxes are possible. A method invocation could therefore begin or end far beyond the viewport.

However, our metaphor model allows to describe each of these three approaches separately. It includes all necessary entities to represent CodeCity, EvoSpaces, or TraceCrawler. To display one of them, only the metrics would have to be adapted.

My metaphor model provides the four entities district, building, floor and street. The former two depict the static low-level structure whereas the latter two illustrate the dynamic low-level constitution. The first and second entity represent a package and a class, respectively. The third and fourth entity represent a class instance and a method invocation, respectively.

In the following, each subsection describes one model package with special ideas and assumptions.

4.2 Model package: visualization

District Inherits from *StaticElement* in Section 4.3. A district is drawn as a cuboid and symbolizes a package. It can contain an arbitrary number of `children` of `StaticElement`, i.e. buildings and districts. These children are displayed on top of the district. Furthermore, a district has an `altitude` that defines the height of the cuboid. The `contextTransformation` attribute is used for internal computations in order to set the relative position of each child correctly.

Building Inherits from *StaticElement* in Section 4.3. A building is drawn as a flat cuboid and symbolizes a class. Besides the inherited properties of the `StaticElement`, it holds all the floors that currently belongs to this building. Additionally, it consists of a `texture`, a `transparency` value, and a `height`. The first points to an image file within a file system. The second describes the degree of transparency. The third defines the height of the cuboid. However, the height for a building is only minimal so that the user does not mistake a building for a floor.

Floor Inherits from *SceneElement* in Section 4.3. A floor is drawn as a cuboid above a building and symbolizes a class instance of the class represented by the underlying building. For convenience, a floor has a reference to its building. It also owns a `height` attribute that specifies the height of the cuboid.

Street Inherits from *SceneElement* in Section 4.3. A street is drawn as a line or pipe between two floors and symbolizes a method invocation. Thus, it contains the caller and the callee floor representing the caller instance and the callee instance, respectively.

In order to represent a static method invocation, i.e., the callee is not a class instance, each building has one floor from the beginning. This floor does not depict an instance of the corresponding class, but the class itself. Hence, each street symbolizing a static method call ends in the lowest floor of the corresponding class.

4.3 Model package: dataTypes

SceneElement A scene element defines a general visual entity. Each entity of the model package *visualization* inherits from this abstract entity. It provides the `name` and the `color` of the inheriting entity. The name describes the particular entity. The color is mainly used for metrics and visually grouping similar scene elements. Furthermore, it contains the relative and the absolute position. The relative position is helpful for arranging the floors of a given building, for example. The absolute position is appropriate to set the position of the streets. Moreover, the `SceneElement` serves as interface between the low-level abstraction and implementation layer. Hence, it additionally consists of an arbitrary `source object` and `scene object`. The first enables the 1:1 mapping to the low-level equivalent. The second allows the 1:1 mapping to the implementation correspondent.

StaticElement Inherits from SceneElement 4.3. A static element defines a general static entity such as buildings and districts. It provides the `width` and `length` of the inheriting entity. However, we follow the approach of Wettel and Lanza [11], i.e. buildings and districts have exclusively squared representations. Thus, width and length are always equal. They can display the NOA or the NOM, for instance.

5 DyVis - A prototype implementation

Dynamic Visualizer (DyVis) is our prototype implementation to demonstrate our metaphor model and ideas in order to visually analyze and recognize performance problems. DyVis is written in Java and uses the frameworks Kieker, EMF, and Java3d[20]. It provides several, useful functions to depict and inspect the statics and dynamics of a given software system. The remainder of this section gives an overview of DyVis, describes its functions, gives an evaluation of it and shows possible future work.

5.1 Overview

Figure 13 illustrates the whole application DyVis. At the top, we can see the menu bar and beneath, the trace control buttons to step forward or, respectively, backward within a trace. The largest area of the application is the display window that shows the three-dimensional scene. On the right side, we can choose between the global and local mode as well as directly select any event of the given trace. The global mode allows to iterate over all events sorted by time in ascending order. The local mode allows to iterate over all events of a given thread sorted by their execution order index[21] in ascending order. The individual threads are represented by the topmost nodes in the tree view. The table at the bottom shows information about the currently selected scene element. On the bottom right, the status bar displays both the index of the current event and the number of available events.

5.2 Functions

DyVis provides the following functions that are also illustrated in Figure 14. Here, the different system contexts describe a group of similar functions.

File menu

DyVis needs the static context first in order to display the dynamic program flow. Thus, the user can load the statics and after that, the dynamics from the menu at the top.

At the beginning, a minimal static model specification was defined in order to depict and test the static architecture of a software system. Hence, there is an implementation of the static loader for this simple own format. Additionally, DyVis

[20] `https://java3d.dev.java.net/`

[21] The execution order index defines the order in which the events were invoked.

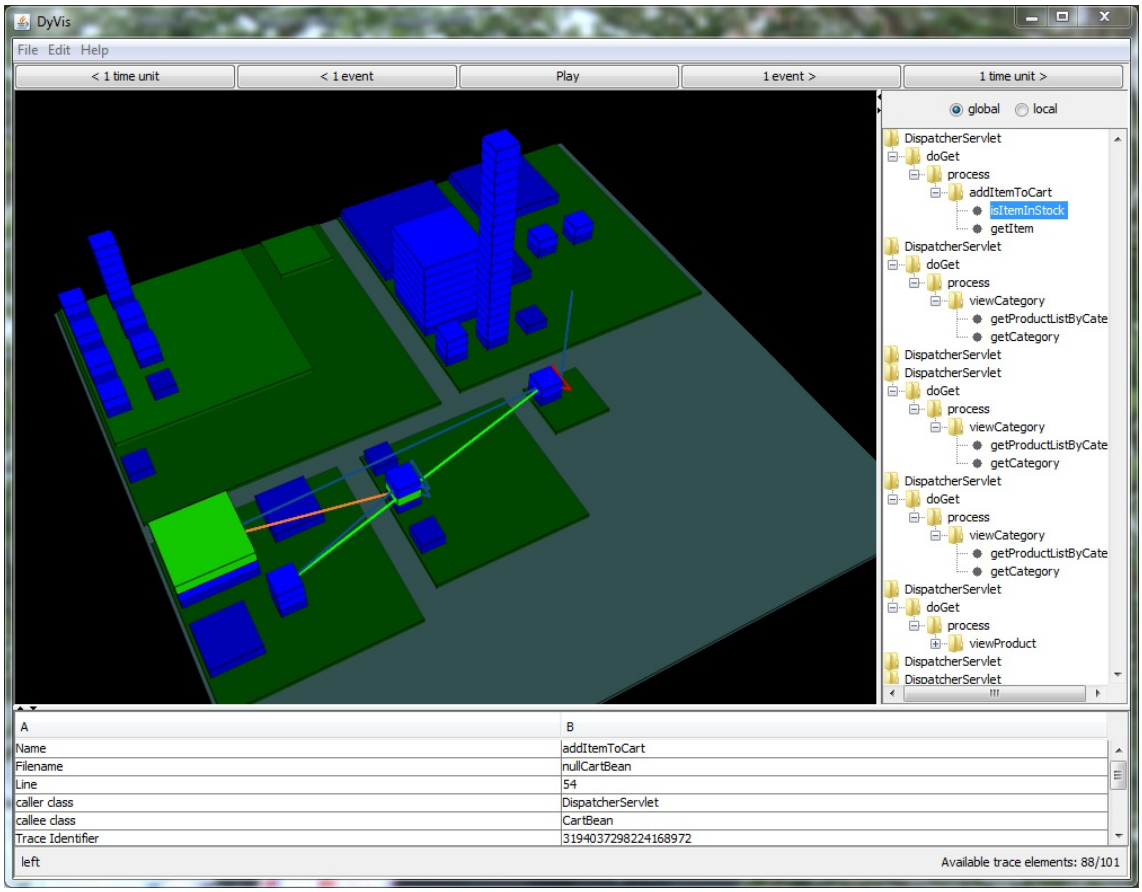


Figure 13: The application DyVis

contains an implementation that enables reading KDM[22] conform files. Thus, with an appropriate parser, e.g. KADis[23], DyVis can represent the static architecture of any program independent of the programming language.

DyVis uses Kieker's interface for reading dynamic log data. Thus, it is also possible to integrate it into Kieker. Moreover, one can make and save a snapshot to an image file to easily insert the given architecture into a presentation or paper. Figure 15 shows a snapshot of DyVis that illustrates the statics and dynamics of the sample application iBATIS JPetStore[24].

Interaction

The user is able to zoom in and out the city by scrolling the mouse wheel accordingly. By pressing one of the keys W,A,S,D one can move forward, to the left, backwards and to the right, respectively. It is also possible to rotate the view by dragging

[22]The OMG's Knowledge Discovery Meta-model (KDM) is a meta-model for representing information related to existing software assets and their operational environments.

[23]`http://sourceforge.net/projects/kadis/`

[24]`http://ibatis.apache.org/java.cgi`

the left mouse button. With a single left click in combination with CTRL, an arbitrary scene element can be selected. Then, at the bottom of the application, an information bar displays specific data about the selected object. A right-click on a scene element opens a pop-up menu that offers special functions for the highlighted element, e.g. open the according source file.

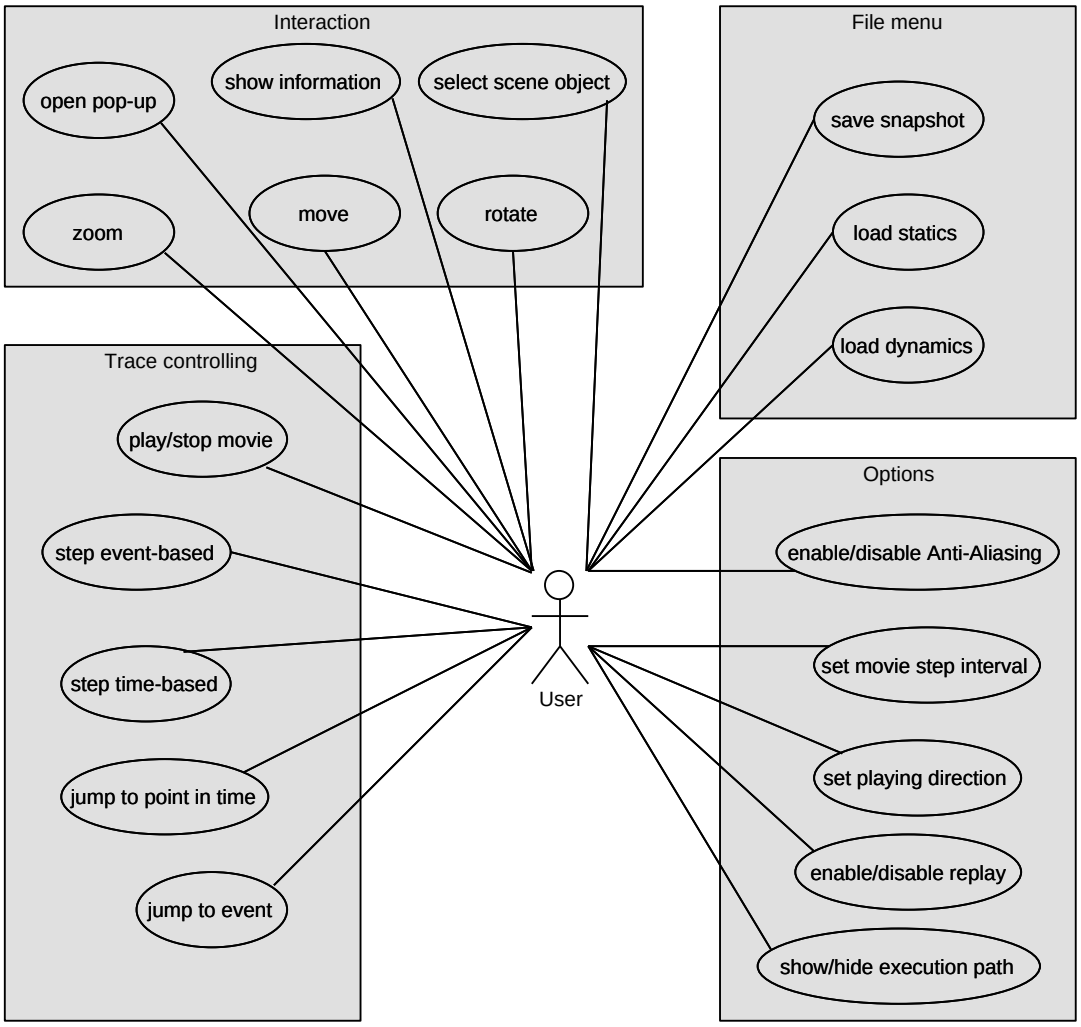


Figure 14: The functions of DyVis

Trace controlling

DyVis provides several ways to view a trace. It offers both time-based and event-based stepping. The former allows to step through consecutive points in time. The time interval is user-definable. The latter allows to iterate over the events according to their execution order index. Instead of using the given buttons to step through the trace's events, the left and right arrow keys can also be used. For the sake of

usability, DyVis provides the functionality to directly jump to a specific point in time or event. Furthermore, it offers the possibility to start and stop the movie modus. Within this modus, DyVis automatically steps through the trace according to an user-definable step time interval. The playing direction can be a selectable combination of forward/backward and time-based/event-based.

Options

DyVis offers some options to individually configure the view and handling of the current trace. As described above, one can adjust the movie step time interval and the playing direction. Moreover, the user can en- or disable anti-aliasing and replaying the movie. Additionally, it is possible to show or, respectively, hide past events, i.e. en- or disable the corresponding execution path.

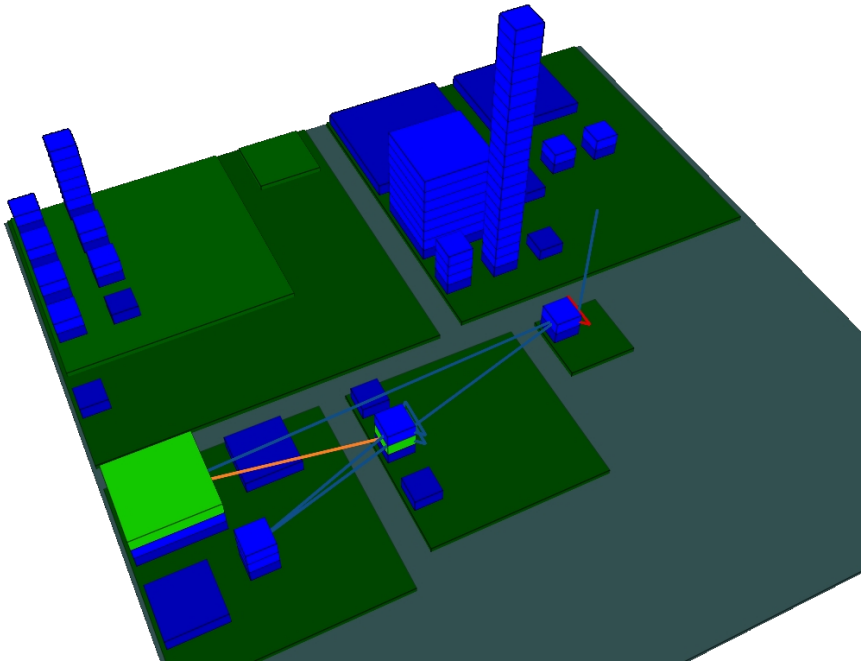


Figure 15: A snapshot of DyVis

5.3 Evaluation

iBATIS JPetStore is a sample web application designed to illustrate some Java technologies. Apart from the used web JSP-files, it consists of 28 Java classes (exclusively interfaces and abstract classes) with an overall of 2800 LOC. By applying KADis and Kieker on JPetStore, DyVis could be evaluated in terms of our evaluation criteria introduced in Section 3.1. Table 3 shows the evaluation results. They are discussed in detail below.

Criteria	Approach
Intuition	
Metaphor	city
Layout	2½-D
Navigation (incl. Zoom)	bird-mode
Filter	no ('yes' in future)
Usability [10]	4 (7 in future)
Mapping	
Scalability	medium ('high' in future)
Metrics	LOC, NOM, NOA, NoObj (user-selectable in future)
Visual representations	object design, length, width, height, color ('texture' in future)
Statics	
Packages	package→district
Class attributes	attribute→length & width
Class methods	-
Inheritance	-
Class associations	-
Design patterns	in future
Dynamics	
Class instances	instance→floor
Method invocations	invocation→street
Duration of execution	duration→color of street
Online-Visualization	no ('yes' in future)
Handling of large data	- (filter & segmentation in future)

Table 3: Evaluation table of DyVis

Intuition As described in the previous section, DyVis uses the city metaphor in combination with a 2½-D layout. The user navigates in bird-mode similar to CodeCity. Currently, DyVis does not support view filtering up to now, i.e. hide particular scene elements or make those transparent that are not of the user's interest.

According to Shneideman [10], DyVis satisfies 4 of 7 criteria of his taxonomy, namely *Overview*, *Zoom*, *Details-on-demand*, and *Relate*. However, a 2-D top view for a complete overview would have to be implemented. Admittedly, DyVis does not meet the criteria *Filter*, *History* and *Extract*. However, it can be easily enhanced to satisfy them.

Mapping DyVis's scalability was tested with JPetStore and Vuze[25]. The former shows an appropriate loading time and a well arranged layout for the static architecture. Admittedly, the latter consisting of about 477000 LOC leads to an inadequate loading time (11 sec. with a today's default office PC) and layout. However, this shortcoming resulting from the huge amount of packages and classes can be elim-

[25] `http://azureus.sourceforge.net/download.php`

inated by a better layout algorithm. By using a different XML framework than JAXB[26] that only uses one CPU core, DyVis' loading time can also be enhanced extremely.

In respect of dynamic scalability, DyVis was tested with an execution trace of JPetStore that contains 106 events. Jumping to one of the last events with showing the according execution path takes no user perceivable amount of time.

Presently, DyVis uses several integrated metrics that are not changeable by the user. Admittedly, a fully user-selectable metric definition and assignment would be more flexible (s. Future Work in Section 5.4). As described above, DyVis displays districts, buildings, floors and streets, each with a particular length, width, height and color. However, it does not support textures so far.

Statics DyVis represents packages as districts and maps class attributes to the length and width of the corresponding building. Since it is intended to focus on dynamic aspects, it does not visualize class methods, inheritance, class associations, and design patterns. The latter, however, should be illustrated in a future version of DyVis.

Dynamics DyVis represents class instances as floors and method invocations as streets between floors. Thus, one is able to comprehend the program flow in detail. In order to quickly set the user's focus on time consuming method calls, corresponding streets are colored according to a color category system depending on the method duration. DyVis' architecture supports online-visualization, but there is no implementation so far.

For a trace log consisting of by far more than 1000 events, a filter and/or segmentation technique could be useful, especially in context of online-visualization. This is also planned, but not yet implemented.

Additional notes For a better orientation and overview in space, DyVis does intentionally not display interfaces, abstract classes and empty packages because they are not important for dynamic analysis.

5.4 Future work

DyVis already offers basic features to analyze and identify performance problems. However, it does not yet support online-visualization, for example. By reading traces over a network connection or by integrating DyVis into Kieker directly, such

[26] `https://jaxb.dev.java.net/`

a visualization could be realized. It should be noted though that only a visualization of statistics is possible because the amount of trace data per time exceeds by far the human's visual receptivity.

For optimal usability as mentioned in Section 2, DyVis should possess a collision detection in order to not zoom through the ground or buildings. Moreover, also mentioned in Section 2, it should realize a smooth, continuous zoom and movement technique for a better orientation. DyVis should therefore provide a completely user-definable mouse behavior including the movement speed and mouse key behavior.

Furthermore, DyVis' architecture allows to dynamically change the metric for each visible attribute, such as length, color and height. These features should be configurable by a file or best via a graphical user interface.

Additionally, it is easy to implement spawning a part of the currently displayed city. This would allow the user to focus on a particular section of the application as referred to as *Extract* by the taxonomy of Shneideman [10].

Extending and individualizing the design of buildings in a restricted, not excessive way can arrange the city more clearly and therefore improves the orientation by structuring the scene. DyVis could visualize a specific building or group of buildings that forms a design pattern as a factory with a smoking chimney, enclosed by a wall or fence, or linked by a bridge, for example.

In order to display more complex systems in an appropriate and useful way, the layout algorithm must be adjusted at first. Second, if there is a definition of a higher level of abstraction, e.g., given by annotated classes or specified in a file, the city metaphor could be extended to a more common landscape metaphor. Such a landscape may then consists of multiple cities where each city represents one component.

6 Related work

In order to understand the static architecture of a software system by visualization, Knight and Munro [6] motivate the usage of metaphors by quoting other visualization researches. Furthermore, they compare different three-dimensional approaches by means of quoting, for instance graph-based and real-world visualizations. They introduce their proof-of-concept *Software World* that implements a real-world city metaphor. Each district represents a class from the source code and buildings symbolize methods. A building consists of floors and doors where the number of doors represents the number of parameters the corresponding method owns. Each floor stands for ten lines of code of the method. For a more human environment (admittedly without any meaning), Knight and Munro decide to enclose the whole city with a fence, to use a block structure for layout and to draw streets between blocks of buildings. Thus, they try to aid the navigation and orientation of the user within their visualization system.

Knight and Munro were one of the first researchers who studied and implemented a three-dimensional approach. Hence, the metrics that they use in their *Software World* are not appropriate to today's software languages and systems. For example, they do not consider the package structure and class attributes. Furthermore, their method representation contains too much details.

Panas et al. [8] show even more realism in their three-dimensional city visualization and additionally depict the dynamics, too. They map components (mainly Java classes) to buildings and packages to cities. To increase realism, they added trees, streets, and street lamps. Cars moving through the city indicate a program trace. Dense traffic as well as the speed and type of vehicle depict the grade, performance, and priority of communication between components. Inter package communication are displayed by the top-down satellite view where streets symbolize two-directional and water unidirectional calls. Clouds cover cities that are not of current interest to the user and thus hidden. Moreover, Panas et al. [8] introduce burning and flashing buildings as well as differently colored buildings that illustrate hot execution spots, frequent component modifications and criteria such as functional and non-functional cross-cuttings.

For representing the static architecture, Marcus et al. [7] also follow the three-dimensional approach with their framework sv3D. They mathematically define an sv3D application[27] as a quadruple $P = V, D, S, M$. V indicates the visual metaphor to be used, D represents the trace data files, S the corresponding source code files,

[27] An sv3D application is a program that uses the sv3D framework.[7]

and M defines the mapping between data and visualization as a set of relations. Their implementation uses containers and poly cylinders whose height, depth, color, and position are mapped to respectively one data entity. Their visualization is similar to multiple, consecutively placed bar charts.

Hence, all approaches above use or even extend the idea of realistic metaphors in order to improve the understanding of a given software system. However, only Panas et al. [8] integrate dynamic data as well and thus enable first trace analysis. Their prototype mainly focuses on the static analysis though because the user cannot inspect a concrete method invocation with information such as its duration and name. One can only see low or heavy communication without any concrete values.

7 Conclusions

In this thesis, we introduced several approaches for visualizing the statics and dynamics of software systems. Besides the two-dimensional UML, we presented the three-dimensional *CodeCity*, *Evospaces* and *TraceCrawler*.

Additionally, we compared the approaches among each other. For this purpose, we introduced and used several evaluation criteria. The comparison revealed that all tools provide advantages but also disadvantages, especially in context of program trace analysis.

Furthermore, DyVis was introduced. It implements both diverse ideas of the approaches and own ideas to especially identify performance problems in software systems. It provides time-based and event-based control over a given trace and can display helpful information about static and dynamic elements. Currently, DyVis requires KDM-based models for the static input and is therefore independent of the programming language. For dynamic input, it uses Kieker trace log files. However, particular attention was paid to DyVis' architecture so that it is easy to extend or change the input and output specifications.

The evaluation with JPetStore shows that DyVis combines the statics and dynamics of the sample application in a clearly arranged way. It provides the most important functions to analyze execution traces, such as time-based and event-based trace controlling, usage of metrics, and details-on-demand. Additionally, it offers supporting functionality that facilitates using DyVis and working with (huge) trace data, e.g. a movie-modus, further pop-up related functions, and the snapshot feature. Hence, DyVis is not just another 3D-approach to visualize software systems, but demonstrates an extended and improved visualization based on other approaches' benefits without having their disadvantages.

References

[1] Philippe Dugerdil and Sazzadul Alam. Execution Trace Visualization in a 3D Space. In *Proceedings of the Fifth International Conference on Information Technology: New Generation (ITNG)*, pages 38–43, April 2008.

[2] Orla Greevy, Michele Lanza, and Christoph Wysseier. Visualizing Feature Interaction in 3-D. In *Proceedings of the 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, pages 1–6, 2005.

[3] Orla Greevy, Michele Lanza, and Christoph Wysseier. Visualizing live software systems in 3D. In *Symposium on Software visualization*, pages 47–56. ACM, 2006.

[4] Eclipse Modeling Framework Group. *Eclipse Modeling Framework*, 2009. URL `http://help.eclipse.org/ganymede/index.jsp?topic=/org.eclipse.emf.doc/references/overview/EMF.html`.

[5] Object Management Group. *UML Superstructure 2.2*, Februar 2009. URL `http://www.omg.org/spec/UML/2.2/`.

[6] Claire Knight and Malcolm Munro. Virtual but Visible Software. In *Proceedings of the International Conference on Information Visualisation*, pages 198–205, 2000. doi: http://doi.ieeecomputersociety.org/10.1109/IV.2000.859756.

[7] Andrian Marcus, Louis Feng, and Jonathan I. Maletic. 3D representations for software visualization. In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 27–ff, 2003.

[8] Thomas Panas, Rebecca Berrigan, and John Grundy. A 3D Metaphor for Software Production Visualization. In *Proceedings of the Seventh International Conference on Information Visualization*, page 314. IEEE Computer Society, 2003.

[9] Thomas Panas, Rüdiger Lincke, and Welf Löwe. Online-configuration of software visualizations with Vizz3D. In *Proceedings of the 2005 ACM symposium on Software visualization*, pages 173–182. ACM, 2005.

[10] Ben Shneideman. The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. In *IEEE Symposium on Visual Languages*, pages 336–343, September 1996.

[11] Richard Wettel and Michele Lanza. Visualizing Software Systems as Cities. In *Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, pages 92–99, June 2007.

Appendices

A Documentation

Documentation

Dynamic Visualizer (DyVis)

Version

2010-03-25

Author

Christian Wulf

Contents

1 Manual **1**
1.1 First steps 1
1.2 Trace controlling 2
1.3 The viewport 3
1.4 Options 3

2 Minimum Requirements **5**
2.1 Software 5
2.2 Hardware 5

3 Architecture **6**
3.1 Components 6
3.2 Packages 7
3.3 Models 8
3.4 The transformation process 10

4 Development Environment **12**
4.1 Software 12
4.2 Hardware 12

References **13**

1 Manual

DyVis provides several functions to analyze the static and, especially, the dynamic architecture of a Java software system. This section serves as manual. The remainder of it describes the usage of DyVis. To get an overview, Figure 1 shows the whole application with seven markers. Each of them illustrates a particular part of DyVis that is described in more detail below.

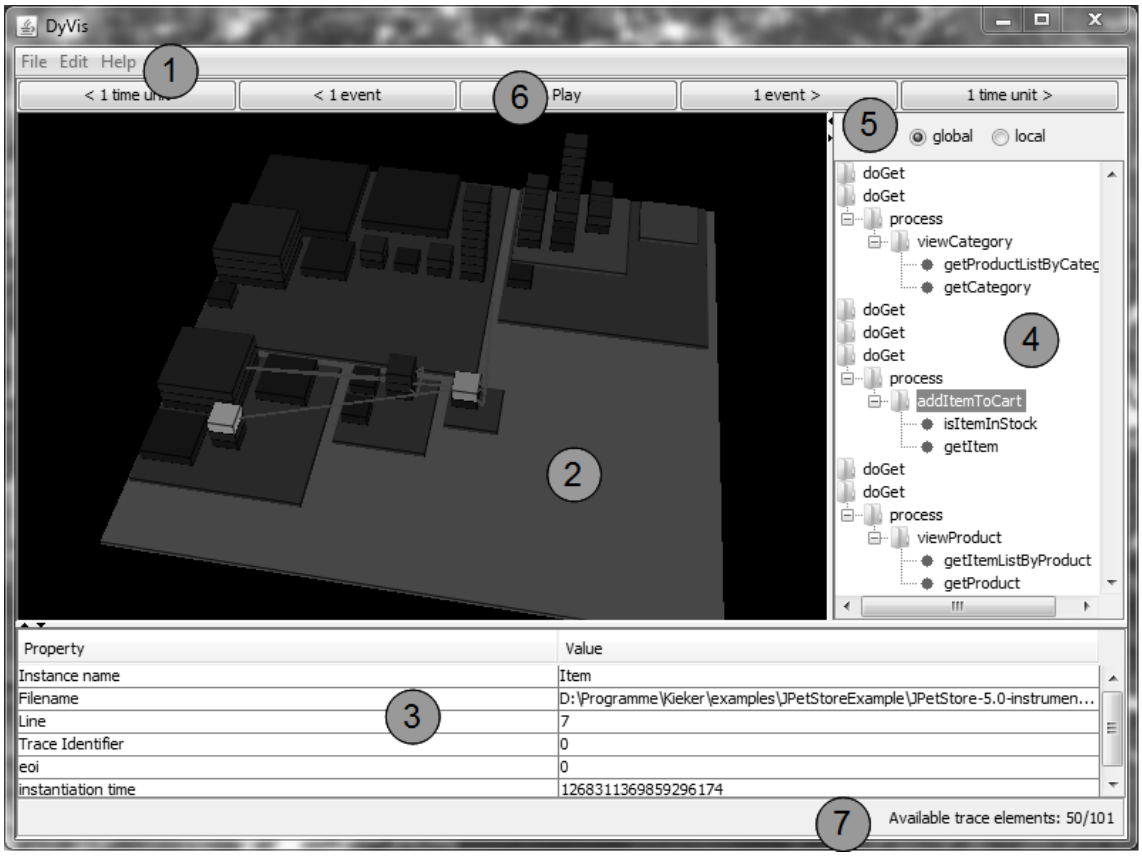


Figure 1: The graphical user interface of DyVis

1.1 First steps

In order to visually analyze a software system's behavior, the underlying static architecture and a corresponding trace must be loaded. Figure 2 shows the file menu that gives access to loading both the necessary statics and dynamics. It is available from the top of the application (marker 1 in Figure 1). *Load Dynamics* appears gray and is, therefore, unusable until the static architecture has been loaded.

After choosing a KDM [1] conform file over the file dialog, the containing data is processed and finally visualized in the center of the application (marker 2 in Figure 1). It is now possible to navigate through the scene and to interact with

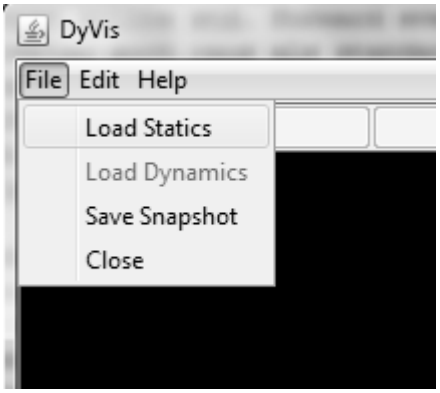


Figure 2: The file menu of DyVis

the given scene elements. By using the keys W,A,S,D one can move forward, to the left, backward, and to the right, respectively. By turning the mouse wheel up and down, one can zoom in and out the scene, respectively. To rotate the whole scene around an imaginary axis in front of the viewport, drag the left mouse button and move the cursor to the left or right, respectively. By pressing the left mouse button in combination with the key CTRL, the scene element underneath the cursor is selected. The table at the bottom of the application (marker 3 in Figure 1) shows information about the currently selected element, e.g. name and filename. A right-click also selects an element and opens a pop-up menu that enables additional context sensitive functions, e.g. open the corresponding source file.

1.2 Trace controlling

After choosing and loading a Kieker trace log file over another file dialog, each top-most node of the tree view on the right side of the application (marker 4 in Figure 1) represents a method call of a separate thread. The children of one particular node symbolize submethod calls within the parent method. By selecting any of the nodes, the corresponding event is displayed in the viewport. According to the trace mode (global or local; marker 5 in Figure 1), DyVis additionally shows either all previous events or only all previous events of the corresponding thread.

Furthermore, the trace control buttons (marker 6 in Figure 1) are now available. They are used to step forward/backward the trace either time-based (buttons: $<$ 1 time unit $>$) or event-based (buttons: $<$ 1 event $>$). The used time unit can be set in the options dialog (Figure 3(a)). Again, according to the trace mode, the events are sorted either by time or by their execution order indices[1], both in ascending order. The status bar at the very bottom (marker 7 in Figure 1) dis-

[1]The execution order index defines the order in which the events were invoked.

plays the current event and the number of available events. Moreover, there is the play/pause button that starts/stops the movie modus. This modus enables automatically stepping through the trace. Continuously, DyVis displays the next event after a user-definable step time interval.

1.3 The viewport

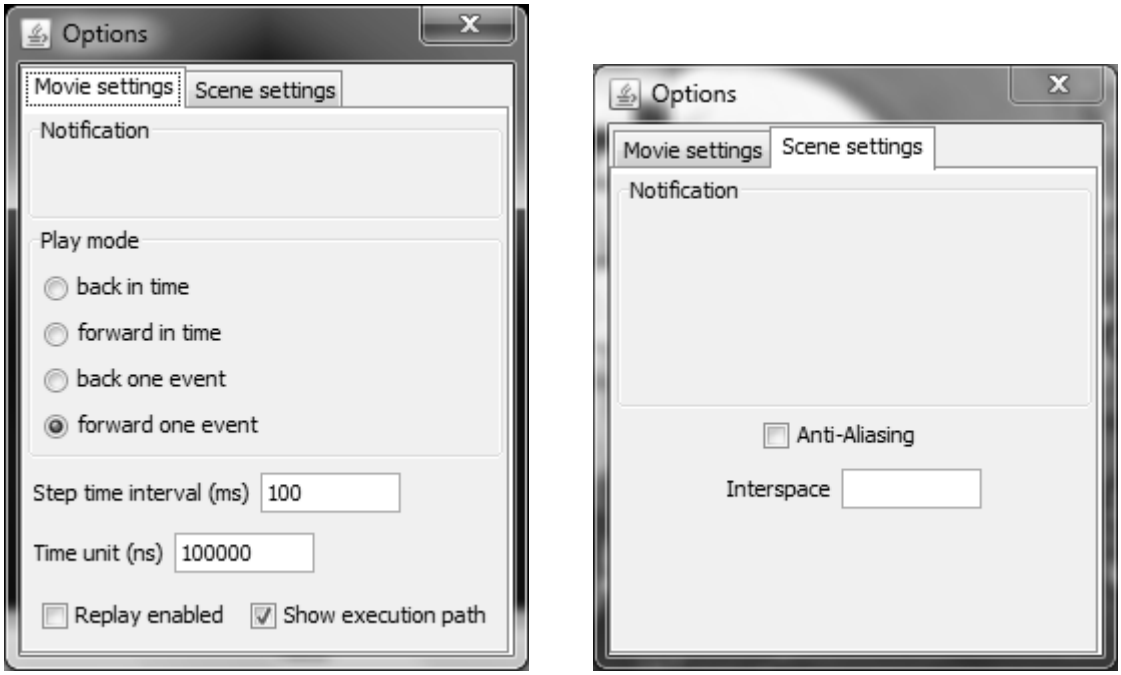
If both the static and dynamic data are loaded, several scene elements are displayed and represent a city. The stacked platforms are districts of that city and symbolize packages with their hierarchy. A flat, squared area is the ground of a building that, in turn, symbolizes a class. Boxes on top of a building are its floors and symbolize instances of the corresponding class. However, the undermost floor is not an concrete instance, but is used to represent static method calls of the corresponding class. A line or pipe between two floors is a street and illustrates a method call from one class instance to another class instance. Its color indicates the percentage of time that the method used in relation to the total time its corresponding thread used. A darker blue indicates a low percentage of time. A brighter red indicates a high percentage of time.

1.4 Options

DyVis provides several options to individually configure the behavior and appearance of it. These options are available from the help menu (marker 1 in Figure 1). Figure 3 shows the options dialog with its *movie settings* (Figure 3(a)) and its *scene settings* (Figure 3(b)). In the following, the different tabs are described.

The *movie settings* tab allows to choose one of the four play modes. This play mode describes the playing direction and the step type (either time-based or event-based). Furthermore, the dialog provides setting the `step time interval` and the `time unit` by entering a time value into the corresponding text field and then pressing ENTER. Moreover, one can enable/disable the `replay modus` and activate/deactivate the `execution path`. The former allows to step back from the last event to the very first event. The latter allows to show/hide the previous events of the currently selected event.

The *scene settings* tab lets the user activate or deactivate anti-aliasing at runtime. Furthermore, it is possible to configure the interspace between the visualized buildings. After a restart of DyVis, the new layout configuration is adopted.



(a) The movie settings to individually configure the movie modus

(b) The scene settings to individually configure the scene appearance

Figure 3: Available options

2 Minimum Requirements

DyVis requires the following software and hardware on the deployed system.

2.1 Software

- Any operating system that supports Java
- Java Runtime Environment (at least version 1.6)
- Java3D [3] (at least version 1.5.0)
- Any KDM-conform [1] parser for generating the static architecture model
- Kieker with DyVisExecutionRecord/Probe and DyVisObjectRecord/Probe for generating the trace data file

2.2 Hardware

- CPU: at least Pentium 4 with 2 GHz
- RAM: depends on the complexity of the static architecture (recommended 2 GB and more)
- Graphic-card: at least DirectX 9/OpenGL 3 compatible
- Free HDD space for the KDM-file and Kieker log file (min. 100 MB)

3 Architecture

Below, DyVis' architecture is described in detail. Each of the following section presents DyVis' components, packages, and models, respectively. We intentionally avoid displaying classes because first an corresponding class diagram can be reconstructed from the source code and second an appropriate description is available by the detailed JavaDoc. The latter can be easily converted into a better readable html-documentation.

3.1 Components

Figure 4 sketches the components of DyVis using the Unified Modeling Language (UML) [2]. The components *View*, *Controllers*, and *Models* implement the MVC pattern. More precisely, they implement the model-view-scene controller pattern that Panas et al. [4] propose to separate the metaphor elements from the visible Java3D [3] scene elements.

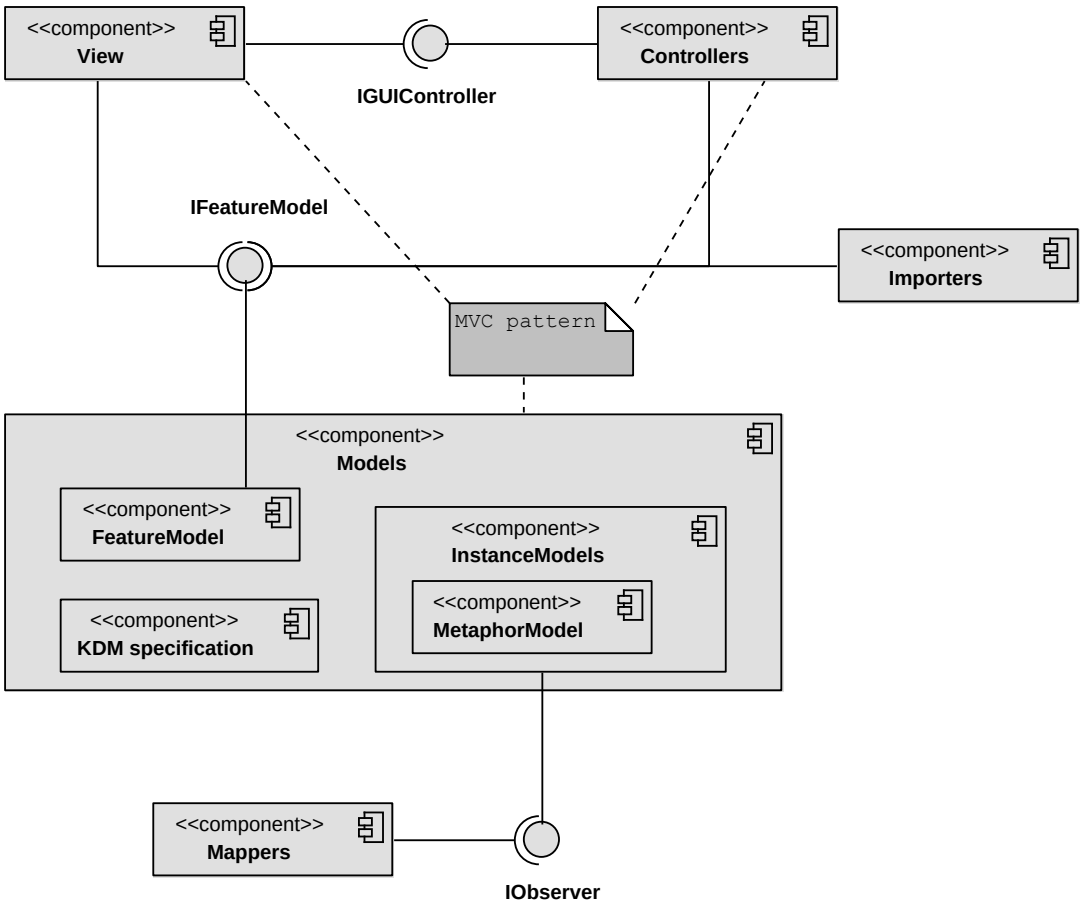


Figure 4: UML components of DyVis

3.2 Packages

DyVis consists of the following packages that are described in more detail below.

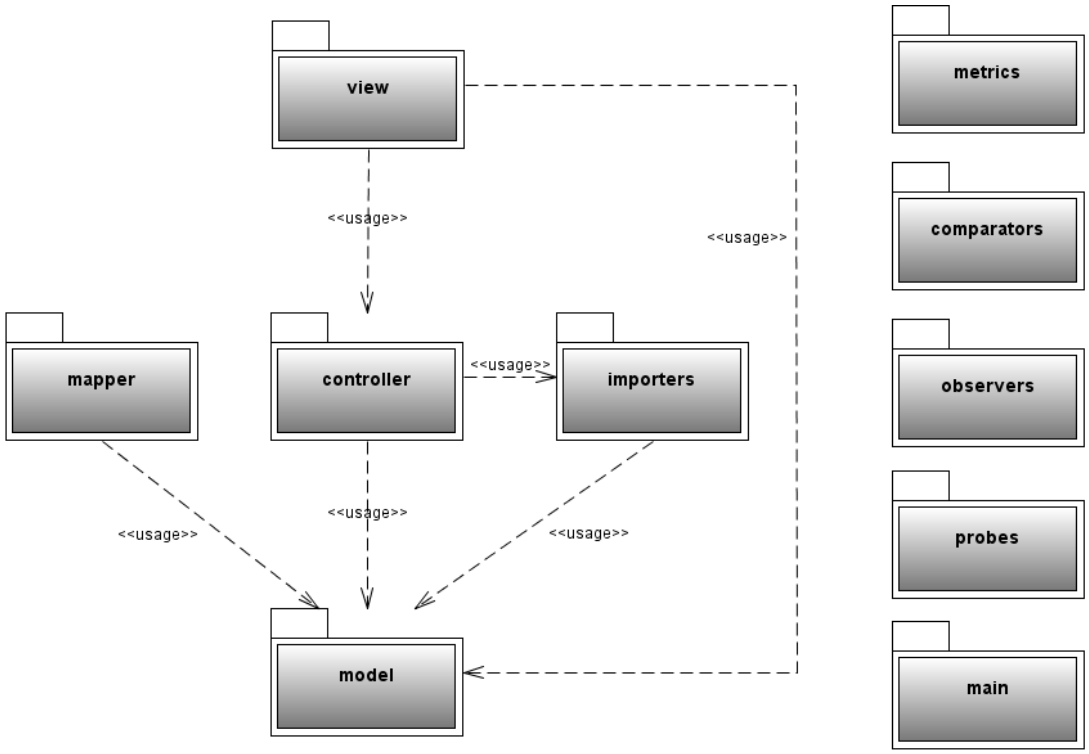


Figure 5: UML packages of DyVis

Model This package contains the model interfaces and implementations of the `FeatureModel`, `SourceModel`, `MetaphorModel`, and `SceneModel`. A more detailed description of each model can be found in Section 3.3.

Controller This package contains all interfaces and implementations of the controllers that the view uses. There is one controller for each different task.

View This package contains all visible components, such as the menu bar, the viewport, and the status bar. It also comprises the scene elements, e.g. the box and the pipe.

Importer This package contains the interfaces and implementations of the static and dynamic importers. They are responsible for reading in the static architecture and the trace data. They also convert foreign external formats into DyVis' internal feature model structure.

Mapper This package contains the mapper and additional utility classes. There is one mapper for each model-to-model transformation.

Main This package contains the executable main class and common utility classes that are used by several other packages.[2]

Comparators This package contains comparators for specific classes that are used by several other packages.[2]

Metrics This package contains interfaces and implementations of specific metrics, such as `ColorMetric` and `LengthMetric`. They are used by the `MetricApplier` in package Mapper.[2]

Observers This package contains interfaces for particular observers that, in particular, model and controller classes implement.[2]

Probes This package contains Kieker record classes and aspects. DyVis also uses the record classes to be able to read the trace data, i.e. to display dynamic architectural views.

3.3 Models

DyVis performs several model-to-model transformations in order to finally display the individual scene elements. Figure 6 roughly illustrates this process. Figure 7 and Figure 8 give a more detailed view. In the following, each model and each mapper is described in detail. Furthermore, the different transformation steps are explained.

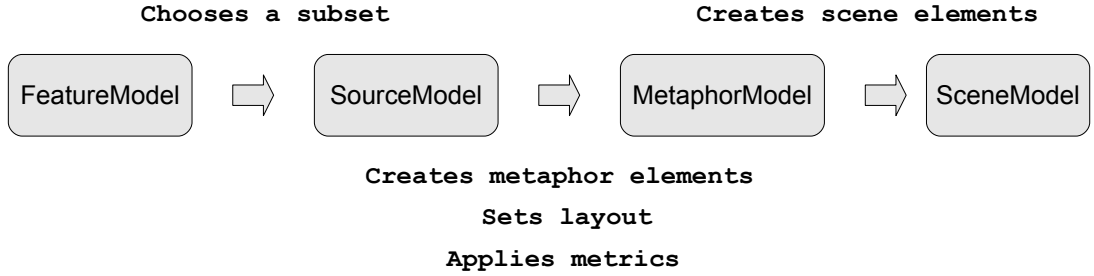


Figure 6: Model transformations in DyVis

[2]These dependencies are deliberately not drawn in Figure 5 in order to focus on the associations of the more important packages.

FeatureModel The feature model comprises all available source code information that DyVis is able to use in order to display static and dynamic architectural views. Apart from import classes (package Importer and Figure 7), all classes use only data that the feature model directly or indirectly provides[3]. Thus, the entities available in this model constitutes the features that DyVis is able to offer. For this reason, this model is called *feature model*.

SourceModel The source model consists of a subset of the feature model. It contains all source code elements whose representatives are presently being displayed by DyVis. By selecting a particular trace event over the graphical user interface (GUI), for example, the trace controller (`ITraceController` in package Controller) receives this user input. Then, it adds and removes the corresponding dynamic elements (class instances and method invocations) to and from the source model, respectively. When the source model is changed, it notifies its observers, especially the source-metaphor mapper.

Source-metaphor mapper The source-metaphor mapper is informed by the source model. When receiving a particular source element by the source model, it is responsible for creating/deleting the corresponding metaphor element in the metaphor model. Moreover, it invokes both the metric applier and the layout manager. The former applies appropriate, user-selectable metrics on the newly created metaphor element. The latter sets, amongst others, the position of that metaphor element.

MetaphorModel The metaphor element holds all elements of the given metaphor whose representative scene elements are presently being displayed by DyVis. When the metaphor model is changed, it notifies its observers, especially the metaphor-scene mapper.

Metaphor-scene mapper The metaphor-scene mapper is informed by the metaphor model. When receiving a particular metaphor element by the metaphor model, it is responsible for creating/deleting the corresponding visible scene element in the scene model. Moreover, it performs several technology-dependent actions, such as optimizations.

[3]Mostly, the feature model is not accessed directly, but indirectly by the source model (SourceModel).

SceneModel The scene model consists of the elements that are directly visible to the user. When the scene model is changed, it notifies its observers, e.g. loading dialogs.

3.4 The transformation process

In order to get a deeper understanding of the transformation process, as an example, this section describes the loading process of the statics of a given software system.

Figure 7 shows the participants of the import process of both the statics and the dynamics. If the user selects a KDM [1] conform file, the `KDMImporter` loads this file. After that, it reads each entry (e.g. a class definition) and creates a corresponding entity of the `FeatureModel`. After processing all available entities, the `FeatureModel` now contains the whole data in its own format.

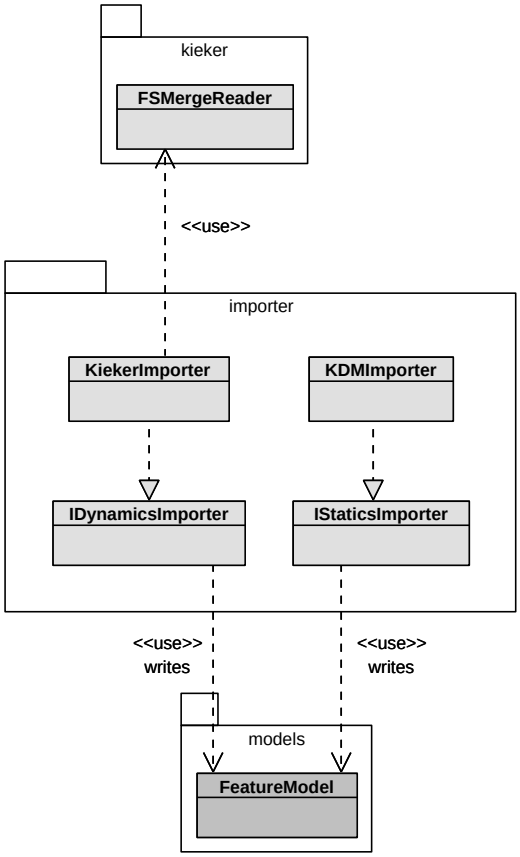


Figure 7: UML class diagram of participants of the import process in DyVis

Figure 8 shows the following model-to-model transformation. When the `KDMImporter` has closed the file, i.e. finished its work, `FeatureModel` informs the `SceneController`. This controller simply forwards the root package of the given

software system to the `SourceModel`. Additionally, it creates a class instance for each class containing the source code information of the later visible, undermost floor.

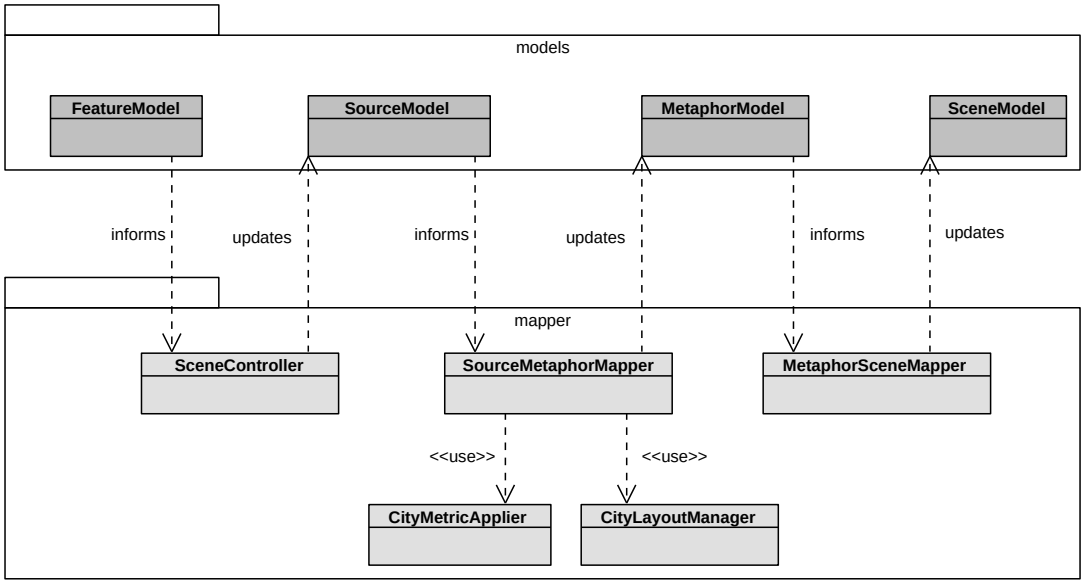


Figure 8: UML class diagram of the model-to-model transformation in DyVis

When setting the root package, the `SourceModel` informs the `SourceMetaphorMapper` about this action. The `SourceMetaphorMapper`, then, reads the information about the packages, class, attributes, and methods from the `SourceModel` and starts several, consecutive tasks. It first creates the corresponding metaphor entities in the `MetaphorModel`. Afterwards, by means of the `CityMetricApplier`, it applies metrics on all newly created metaphor elements and writes the results in the corresponding attributes of them. In the end, the `SourceMetaphorMapper` sets the positions of each metaphor element via the `CityLayoutManager`. Now, the `SourceMetaphorMapper` informs the `MetaphorModel` that it has finished writing/updating the model.

When receiving the "UpdateFinished"-message, the `MetaphorModel` informs the `MetaphorSceneMapper`. The `MetaphorSceneMapper` reads in the metaphor elements and creates corresponding visible Java3D [3] scene elements, e.g. boxes and cylinders. It then stores them in the `SceneModel`[4]. The `MetaphorSceneMapper` simply uses the positions of the metaphor elements to correctly arrange the scene elements within the viewport.

[4]In Java3D, an abstract scene graph structure is used to store the scene elements in an efficient way.

4 Development Environment

4.1 Software

- Platform
  - Java 1.6.0 Update 18 64-bit
  - Java3D 1.5.1 [3]
  - Tortoise SVN 1.6.7 64-bit
- Tools
  - Eclipse Galileo 3.5.2
    - Plugin: EMF 2.5.0
    - Plugin: SVNKit 1.3.0 (for SVN 1.6.2)
  - Adobe Acrobat Reader 9.3.1
  - TexMakerX 1.9.3
  - Visual Paradigm 7.2 Standard Edition
  - KADis 1.0
- Operating System
  - Windows 7 64-bit

4.2 Hardware

- 1 personal computer with Internet access

References

[1] Object Management Group. *Knowledge Discovery Metamodel*, January 2008. URL `http://www.omg.org/technology/documents/modernization_spec_catalog.htm`. (2010.03.25).

[2] Object Management Group. *UML Superstructure 2.2*, February 2009. URL `http://www.omg.org/spec/UML/2.2/`. (2010.03.25).

[3] Sun Microsystems. *Java3D*, December 1998. URL `https://java3d.dev.java.net/`. (2010.03.25).

[4] Thomas Panas, Rüdiger Lincke, and Welf Löwe. Online-Configuration of Software Visualizations with Vizz3D. In *Proceedings of the 2005 ACM symposium on Software visualization*, pages 173 – 182, 2005. doi: http://doi.acm.org/10.1145/1056018.1056043.

B Attachments

One CD containing

- the thesis as pdf-document labeled *Bachelorthesis.Christian.Wulf.pdf*,
- the source code of the own program, and
- the documentation of it as pdf-document.