Info Vis

Article

Understanding software evolution with software cities

Information Visualization
12(2) 200–216

DOI: 10.1177/1473871612438785
ivi.sagepub.com
SAGE

Frank Steinbrückner and Claus Lewerentz

Abstract

Software cities are visualizations of software systems in the form of virtual cities. They are used as platforms to integrate a large variety of product- and process-related analysis data. Their usability, however, for real-world software development often suffers from their inability to appropriately deal with software changes. Even small structural changes can disrupt the overall structure of the city, which in turn corrupts the mental maps of its users. In this article we describe a systematic approach to utilize the city metaphor for the visualization of evolving software systems as growing software cities. The main contribution is a new layout approach which explicitly takes the development history of software systems into account. The approach has two important effects: first, it creates a stable gestalt of software cities even when the underlying software systems evolve; thus, by preserving its users' mental maps these cities are especially suitable for use during ongoing system development. Second, it makes history directly visible in the city layouts, which allows for supporting novel analysis scenarios. We illustrate such scenarios by presenting several thematic cities' maps, each capturing specific development history aspects.

Keywords

Software cities, software landscapes, software evolution, software visualization, metrics, software comprehension

Introduction*

Teams of many different people with strongly varying interests, roles, responsibilities, and backgrounds are involved in the development of large software systems. How do they communicate about the software they are defining, constructing, maintaining, and administrating? Owing to the complex and intangible character of software it is very difficult to create a commonly shared picture or mental map of the software system. Software visualization, and in particular the visualization of software structures as given by the existing program code and enhanced with additional development information (see refs. 2, 3), aims at creating materializations of software systems to provide a common basis for the localization of product properties (such as size or quality indicators) and process events (such as errors found or changes made).

To serve as such a common basis for a broad range of different application scenarios, software visualizations should specifically meet the following requirements (besides more general visualization requirements):

- The visualization should provide a specific and meaningful gestalt for each software system to support orientation and spatial memory.
- The visualization should allow for representing the software system at different levels of

Brandenburg University of Technology Cottbus, Cottbus, Germany

Corresponding author:
Claus Lewerentz, Platz der Deutschen Einheit 1, 03046 Cottbus, Germany.
Email: cl@tu-cottbus.de

detail (e.g. architectural components, code components) and at different development stages in a uniform and consistent way.
- The visualization needs to be robust against small changes in the software system. It has to support the observation of evolving systems over many different versions, i.e. it has to reflect structural and property changes without disrupting the overall picture.
- The visualization should allow for easily integrating many different data in a consistent and systematic way.
- The visualization process and the visualizations themselves should scale up to systems of several million lines of code.

In literature a large variety of different approaches and visual metaphors for the visualization of large software systems are proposed. As discussed in previous work (e.g. refs 2,3,4–6), the metaphors of maps, cities, and landscapes very well meet most of the requirements outlined above. However, only a few approaches explicitly address the question of how to produce robust visualizations for evolving software systems. In contrast to these solutions, in our approach robustness will be achieved by explicitly representing the development history in the software cities layouts.

The "Related work" section reviews the most relevant related work on the use of map, city, and landscape metaphors in software visualization. "The EvoStreets approach" section explains and illustrates the concepts of our visualization approach, which is the main contribution of this paper. In the "Comprehension scenarios" section we discuss and illustrate new application scenarios supported by our approach. The "Discussion" discusses advantages, shortcomings, and open issues of our approach. "Tool support" gives some details on our visualization prototype and, finally, "Experiences" summarizes this article and points out ongoing work.

Related work

Knight and Munro[5,7] describe software worlds as consisting of countries, cities, districts and fine-grained elements such as houses and gardens. These hierarchically structured elements are used to visualize hierarchically structured Java programs with methods, classes, files, and directories being mapped onto buildings, districts, cities, and countries, respectively. In ref. 8 a variant of this metaphor is presented to visualize more abstract software components in component cities.

Panas et al.[9] explore the city metaphor at a very detailed level. They indicate the wide range of visual elements (cars, clouds, fire, flashes, etc.) the city metaphor provides, and they discuss their applicability to visualize static, dynamic, and production cost-related information. Elements such as trees, streets and street lamps are added to support intuitive interpretation and increase realism.

Later, Panas et al.[2] use the city metaphor to provide single-view visualizations designed to address the information needs of different software development and maintenance stakeholders. Their cities represent software systems at different levels from coarse-grained directories to fine-grained class member functions. The cities layouts are computed using two methods: coarse-grained elements are positioned by a force-directed algorithm whereas fine-grained elements (buildings) are laid out compactly in a grid-based manner. The resulting cities are augmented with data obtained from analyses (runtime, metric, static/structure, and repository analyses).

Alam and Dugerdil[10–12] describe an approach in which software artifacts are represented by office buildings or city halls. These elements may be organized in cities whose layouts are computed by several different layout strategies (concentric and chessboard layouts,[11] containment layouts[12]). Animated solid pipes can interactively be displayed to show element references and their directions.[11] In ref. 10 this approach is enhanced by a highlighting technique called night view to point out elements involved in a given execution trace.

In the literature more approaches (e.g. refs 4,13,31) for visualizing software systems as maps, cities, or landscapes, or in 2.5-dimensional (2.5D) space are proposed. However, to the best of our knowledge, only a few approaches explicitly consider the evolution of software systems. Kuhn et al.[3] describe thematic software maps that position software artifacts on a two-dimensional (2D) map according to their vocabulary. Their layout algorithm produces relatively stable maps of evolving systems. These maps are used as consistent platform to locate additional analysis data.

Wettel and Lanza[6] propose an approach in which the software decomposition and artifacts are mapped onto city districts and buildings, respectively. Building size is derived from basic size measures using a box-plot-based mapping technique.[14] The resulting cities serve as a platform that can be enriched by more sophisticated analysis data to support the identification of design problems[15] and to gain insight into the structural evolution of software systems.[16] They present a set of techniques to visualize evolutionary data on both a fine- and a coarse-grained abstraction level. Their time traveling technique allows one to step backwards and forwards in time, and thus to visualize the software system at its different development stages. For this

purpose, each software artifact is uniquely assigned a fixed area in the city. As a result, this approach yields stable layouts sequences, which, however, can only be achieved because all versions of the software system are known in advance. As soon as new versions become available, new visualization sequences must be computed, which in turn can differ significantly from the original sequence and the mental models constructed by its users so far. A similar approach, though based on treemaps, is described by Langelier et al.[17]

The approach described in the following sections aims at providing stable software cities without knowing the development history in advance. In contrast to previous work in the field of dynamic graph drawing (e.g. refs 18–20) this will be achieved by explicitly capturing the development history in the city shape itself.

The EvoStreets approach

Using a geographical metaphor (as cities) for the visualization of software systems guided us to adopt techniques developed by cartographers over many years. A basic process in map making is the use of a three-staged representation chain. The original geographic data of a landscape or city are collected in a so-called *primary model*. These data are used to create a *secondary model* that comprises all aspects of the primary model that should be represented as a common view (2D as well as 2.5D models) for the geographic reality. The secondary model is used as basis from which *tertiary models* or *thematic maps* are derived. Thematic maps represent and emphasize particular aspects of the real-world system (landscape, city). Typical techniques for building thematic maps from secondary models are selections, projections, coloring, or superposition of symbols and diagrams.[21] Thus, a secondary model typically plays the central role to provide a stable reference map for a large variety of particular visualization scenarios for the very same geographic reality. We adopt this process because it helps to fully exploit the capabilities of cartographic city and landscape modeling, as will be shown in the following sections.

Logical primary model

The primary model for software cities captures structural and analysis data. Structural data refer to the static software structure, i.e. the decomposition into subsystems and modules, as well as dependencies among modules. For Java systems, structural data typically include the package hierarchy, classes, and inheritance, aggregated method usage, and attribute and type access relations between classes. Even though in this article all of our examples are Java systems, we will consistently use the terms *subsystem* and *module* to indicate decomposition elements (such as Java packages) and decomposition leafs (such as Java classes), respectively, as these terms are not associated to a specific programming language or abstraction level. Thus, subsystem and module can also denote directories and header and implementation files for C++ systems, or even such coarse-grained units as hierarchically structured architectural components. We use the more general terms *system element* or *element* to denote both subsystems and modules.

The structural information is enriched by additional analysis data, which can be distinguished into product-related analysis data, gained from a variety of different software analysis tools (e.g. Findbugs, PMD, Checkstyle), as well as process-related analysis data such as developer activities, development effort, or test coverage. Whereas structural data about the system decomposition are essential for our visualization approach, analysis data can optionally be integrated to support particular usage scenarios.

The primary model is populated with structural and analysis data for several versions of the software systems. These versions can be selected arbitrarily, for example on a calendar time basis (daily, weekly, etc.), by revision number from a version control system such as subversion (SVN) or concurrent versioning system (CVS), or according to defined project milestones. Thus, the primary model contains product- and process-related data for a series of different development stages.

The primary model internally is represented as a dynamic, attributed graph and serves as a central data model to exchange data between project analysis tools and the visualization tools. All secondary and tertiary models are derived completely from data in the primary model. Thus, throughout this paper, any discussions on evolutionary aspects of software systems refer to the systems evolution as captured in the primary model: when we talk about *new* and *deleted* elements, this refers to the difference to the previous version contained in the primary model.

Geometric secondary model

The secondary model is a 2.5D geometric model, representing primary model data, that captures the main structural and evolutionary properties of the software system. It has to fulfill the aforementioned requirements, particularly to provide a specific, stable gestalt for each software system. This gestalt is mainly influenced by the particular layout of the system elements.

There are at least three different aspects of the primary model that are usually used to generate the layout:

Figure 1. A hierarchical street layout representing the decomposition of the visualization tool CrocoCosmos (Revision 0, 389 Java classes).

- the decomposition hierarchy (e.g. package and class nesting);
- dependencies (e.g. call and access relations, type–inheritance relation); and
- element properties (e.g. type, size).

Whereas most other (graph) layout approaches are based on decomposition hierarchy plus element properties (e.g. Code City[6]) or decomposition hierarchy plus structural dependency relations (e.g. Software Landscapes[4,20]), the layout approach described in the next section (called EvoStreets) combines decomposition hierarchy, element properties, and development time. Making creation time of system elements a first-class property of the layout is the main novel aspect of this approach. A consequences of including development time in layout strategies is stronger requirements on the stability of layouts over different stages in a system's version history.

In the following sections we describe how such secondary models are derived from primary models. We use the source code of our visualization tool *CrocoCosmos*[22] as an example system throughout this document.

The EvoStreets layout. In the secondary model the decomposition hierarchy is represented by a hierarchical street system, as shown in **Figure 1**. Each street represents a certain subsystem. Contained subsystems form orthogonal branching streets. The system level is represented by a main road from which the top-level subsystems branch off. The width of the subsystem streets is inversely proportional to the subsystem's depth in the hierarchy.

Modules are represented as square building plots that are attached to the street representing their containing subsystem. For Java systems, a street with attached building plots shows a package with its contained classes.

The size of the building plots is used to represent any important property of the corresponding modules. In this article, it is proportional to the coupling between the represented module and the rest of the system (number of incoming and outgoing static dependencies from/to all other classes of the system), i.e. the number of method calls, attribute and type accesses, and inheritance relations. Besides the content size of modules, their coupling is a central basic property for many application scenarios because it indicates the potential impact that changes to this module may have to the rest of the system.

System evolution and layout stability. So far, the secondary model provides layouts representing modules, their dependency property, and the hierarchical system decomposition. During a software system's life cycle the program structure undergoes many changes. New elements are added, elements are removed or relocated to other subsystems, and element properties change.

Such system modifications are reflected in the geometry of the secondary model. As the secondary model's purpose is to serve as a stable geometric base representation for the system during its entire lifetime, the overall structure of the geometric model must not be disrupted by such changes. We addressed this robustness or geometric stability requirement by taking an incremental layout approach with the following design decisions:

- New elements are represented by new building plots and streets. They are attached to the (outer) end of the street representing their containing subsystem. In general, for this purpose the street must be extended. Sets of new elements that are jointly added to the same street are evenly distributed to both sides of the street.
- Building plots and streets remain on the same side of the street to which they are initially attached. Their initial order along that street is preserved in all subsequent layouts.
- If an element is removed from the software system its representation is not removed from the secondary model but marked as disused.
- The information whether an element was relocated to another subsystem or removed from the system is not stored in our primary models. Thus, elements that were moved to another subsystem are treated as removed elements in the context of their source subsystem and as new elements in the context of their target subsystem.

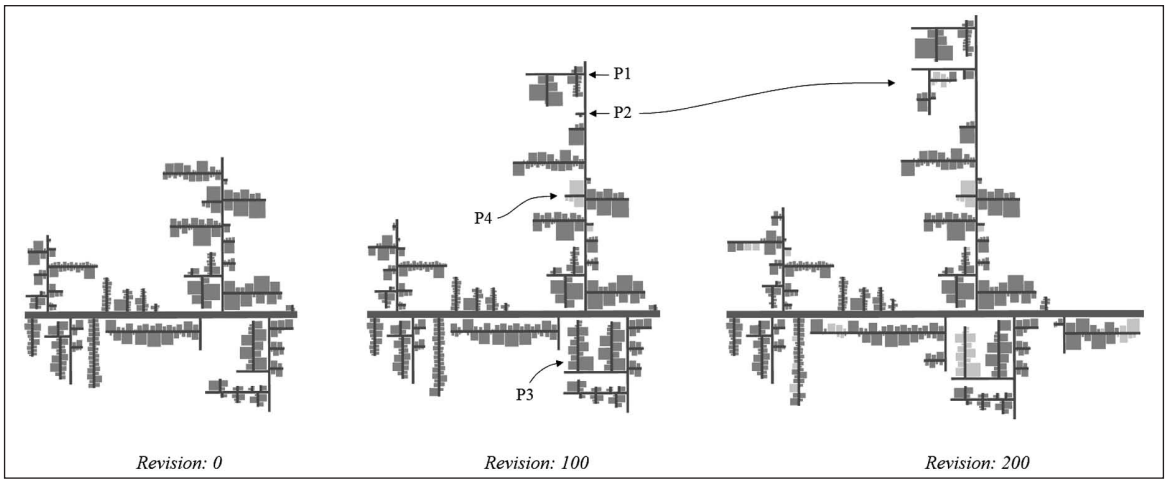

Figure 2. Three development stages of CrocoCosmos.

- If the size of a building plot changes (e.g. due to changes in the corresponding element property), the following neighbor plots down the same side of the street are shifted along the street. These shifts can induce further shifts if a street has to be extended.

Figure 2 illustrates the effects of these design decisions for our example system visualized at three different revision points. The system grows from initially 389 classes to 439 classes in revision 100 and to 466 classes in revision 200. Streets represent Java packages and building plots represent Java classes.

- New elements: Between revision 0 and 100 two new packages, P1 and P2, are added. Their representations are attached to the end of the street, which represents their parent package. This street has to be extended. However, the rest of the layout remains unaffected by this modification.
- New elements: Between revision 0 and 100 another package, P3, is added. Because there is enough empty space available for its representation, this modification has no impact on the rest of the layout.
- Growing elements: Between revision 100 and 200 package P2 grows (a new sub-package is added). As its representation now requires more space, the representation for package P1 must be shifted upwards. Similarly, if in revision 100 one more sibling to P3 would be added to the containing package, and assuming that there would not be enough space for its representation, some streets would have to be shifted. In both cases, the layout has to be adapted locally, but the rest of the layout remains unaffected.
- Removed elements: Between revision 0 and 100 package P4 is removed from the system. This modification has no effect on the layout as representations of removed elements are not removed from the secondary model. In Figure 2 these elements are drawn in light gray color. If the representation for P4 would be removed from the secondary model, packages P1 and P2 could be moved downwards. This would yield more compact layouts at the cost of lower stability.
- As the example illustrates, the layouts are not fixed in the sense that each representation remains at its absolute position. Instead, changes in the primary model may cause local layout adaptations that result in shifting building plots and streets. However, as the order of neighboring representations is preserved, the overall structure of the layout is not disrupted from one version to another.

A small empirical study[23] showed that the EvoStreets approach yields rather stable layouts for typical development histories. We compared the layouts produced by three different approaches (EvoStreets, a packing approach similar to ref. 6, and a force-directed graph drawing approach[20]) with respect to layout stability. In the study we used five similarity measures from the literature,[24,25] each addressing a specific similarity aspect (such as changes in positions, orderings, or neighborhoods) and applied them to five small to medium-sized software systems (up to 3023 classes and in a total of 55 versions). The results show that the EvoStreets approach yielded the best stability results by providing the highest similarity values for 82% of all layout transitions. The issue of layout stability, however, still needs more comprehensive investigations.

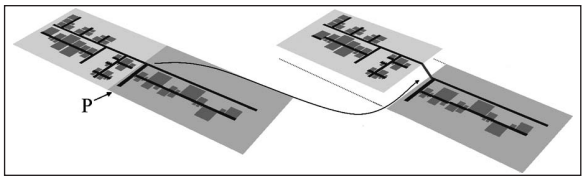

Figure 3. Using elevation levels to visualize creation time.

Representing development history. The above-described layout approach for adding new elements produces a geometrical ordering of system element representations along streets according to their creation time. However, as entire sets of elements may be added between two observed versions, the location order of their representations cannot be strictly interpreted as the chronological order of creation times. Elements in continuous segments of streets may have been added in the same version, i.e. have the same creation time.

The street on the left-hand side of Figure 3 represents a Java package that contains six sub-packages. They were added as two sets in two successive versions. In the first version, the five upper-left sub-packages (three above and two below the street) were added jointly. In the second version sub-package P (containing only one sub-package) was added. One standard way to represent this information would be to use coloring in a thematic map on the tertiary model level. To represent the development history defined by creation time, i.e. the age of elements already as a geometric secondary model property, we extend our flat city space to a leveled landscape. Creation time of an element is mapped to an elevation level. Older elements are positioned on higher levels than younger ones. As a consequence, streets and their attached plots will stretch over different elevation levels. In the example below, sub-package P is younger than the other sub-packages. Therefore, its representation is less elevated. As a consequence, the street representing the containing package declines, as indicated on the right-hand side of Figure 3.

The fact that our incremental layout approach creates a chronological order of representation elements allows for combining all elevation levels into a landscape with a rather continuous terrain on which the city elements, streets, and building plots are positioned (see Figure 4(b)).

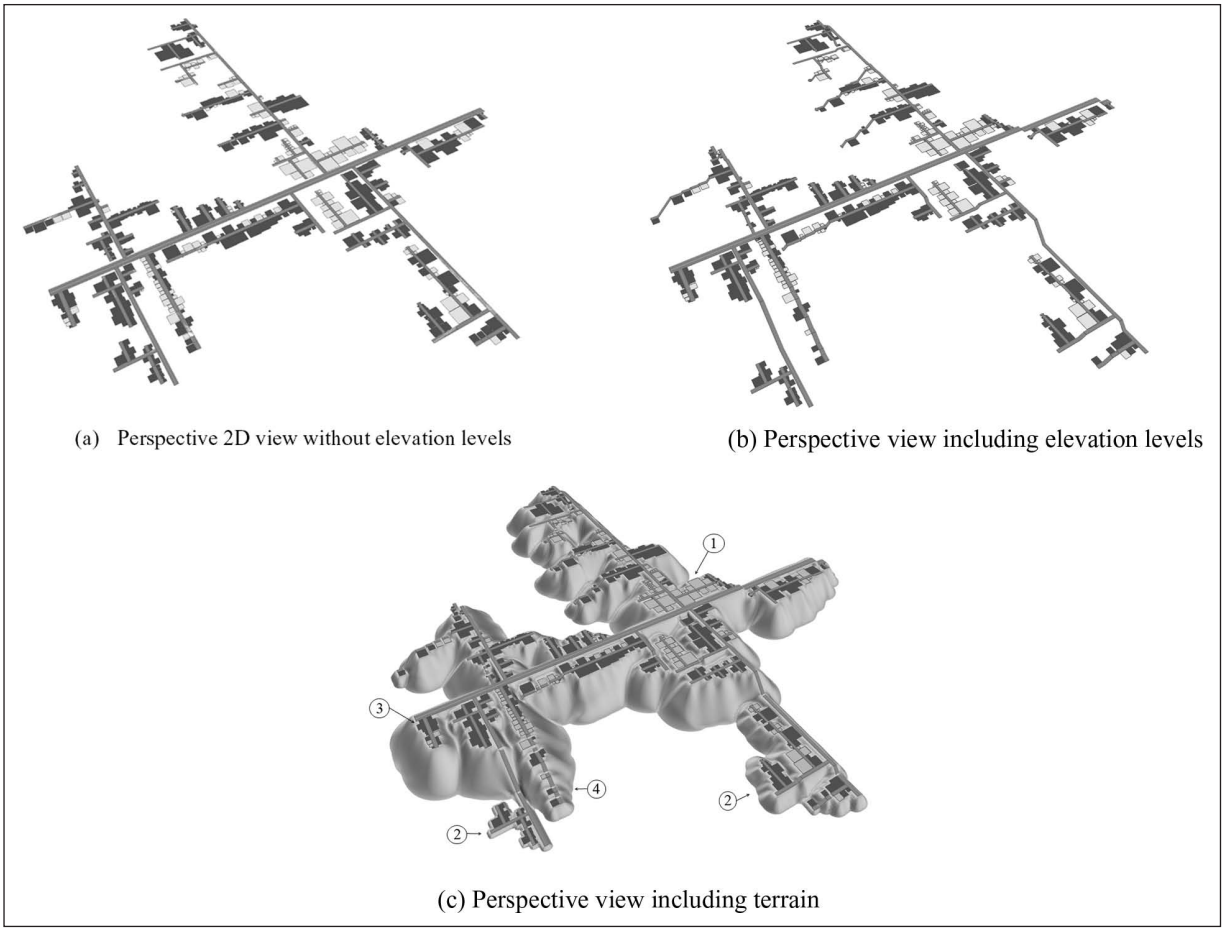

Figure 4. Using terrain to visualize evolution (CrocoCosmos at revision 800, 463 Java classes): (a) perspective 2D view without elevation levels, (b) perspective view including elevation levels, and (c) perspective view including terrain.

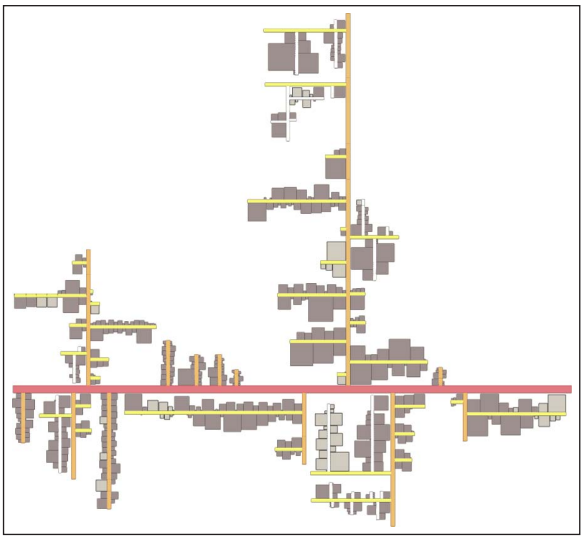

Figure 5. 2D city plan projection.

The terrains are constructed following an approach presented in ref. 26 for the generation of implicit surfaces. Implicit surfaces are described by a set of arbitrary generator objects that influence a global density field. An implicit surface is defined as the set of those points where the density field equals a certain predefined threshold. In our context, building plots and streets are interpreted as generator objects that influence the global density field depending on their position and height. In contrast to ref. 26, the resulting global density field is not compared against a threshold to compute an implicit surface. Instead, density is interpreted as height, which yields smooth terrain surfaces.

As shown in Figure 4(c) secondary models can show several visual patterns that reflect particular situations in development history:

(1) *Disused sites* are regions of empty plots representing parts of the system that have been removed from the system or which migrated to another subsystem.
(2) *Suburbs* represent local system extensions, i.e. sets of modules that were added to a subsystem during a relatively short period of time.
(3) *High plateaus* represent subsystems that have been part of the software for a relatively long time but which were not extended after their initial creation.
(4) In contrast to high plateaus, *mountain slopes*, i.e. streets that continuously descend on hill sides, represent continuously extended subsystems.

The overall approach is based on two decisions regarding the growing directions. First, new structures are added toward the far end of their containing element's street representation; as a result, the city tends to grow at its periphery. Alternatively, new structures could be placed at the beginning of each containing element's street representation; as a result, the city would tend to grow from its center. This, however, would cause instabilities to existing structures as they would be shifted toward the outside.

Second, we decided to place older structures on higher elevation levels; as a result, the city grows from top to bottom. Alternatively, we could have decided to place newer structures on higher elevation levels. Given the decision that the city grows at its periphery, growing from bottom to top would cause the city center to be placed in a valley surrounded by mountains for each suburb. The streets would be embedded in deep valleys, which would create much more occlusion of streets and buildings.

We decided for the combination of growth at the periphery and growth from top to bottom because it creates stable city layouts with a coherent shape and less occlusion. Thus, the resulting secondary model provides stable, coherent city layouts representing modules and their properties, the hierarchical system decomposition, and the structural system evolution.

Thematic tertiary models

As in cartography, our secondary models are used as stable bases to derive tertiary models or so-called thematic maps. They are designed to support specific application scenarios by visualizing particular aspects of a given software system and its development history. Typical transformations from secondary to tertiary models are selections, projections, generalization, coloring, and superposition of symbols and diagrams. In the following sections we will outline typical transformations and thematic map elements used in our approach.

Our geometric base model is a 2.5D terrain model which can be visualized and explored in an interactive three-dimensional (3D) environment. This allows for using further 3D elements to represent properties. On the other hand one has to care for appropriate projections of such 2.5D terrain models into 2D representations.

All maps described below can be viewed interactively for any point in development time, which additionally supports the comprehension of when components were created or deleted and how the component content changes over time.[16]

Map projections. We use two types of projections for building thematic maps, namely city plans (flat topographical maps) and panoramas (perspective maps).

City plans are produced by a simple orthogonal projection of the terrain onto the base level. They have a standard orientation and avoid any problems caused by occlusion or perspective distortion. Highlighting and coloring of streets and areas and the use of text labels are standard ways of representing properties of system elements. City plans (as in Figure 5) are used for all purposes that require an easy system overview, structure comprehension, and navigation. They serve as the base map to position additional categorical or ordered information such as current development hot spots or the degree of test coverage.

Panoramas are projections from an arbitrary viewing perspective (angle and viewpoint) and create some degree of occlusion and perspective distortion (see Figure 4). Typically, a bird's-eye perspective with a viewing angle of 45° is used. The orientation may vary depending on which parts of the system are to be emphasized.

Panoramas allow for creating both system overview and detailed views and are particularly well suited to show 2.5D terrains and cityscapes. This allows us to use "buildings" as additional 3D representation elements. We use panorama views whenever the age of software elements, i.e. the terrain level in the secondary model plays an important role for the intended application scenario.

In city plans as well as on panoramas standard cartographic generalization techniques for representing such maps on different scale levels could be applied.

Contour line. In cartography color and contour maps are used to represent smooth,continuous phenomena. The use of contour lines is the right choice if the number of data values is high and coloring is to be used for representing other properties.[21] We use contour lines for visualizing the elevation of the terrain in the same way as it is done in topographical maps. Every contour line connects points of a certain elevation. The spacing of elevation values and the form of contours can be chosen such that a good readability is achieved (e.g. draw every fifth contour with a stronger line).

Contour lines can be embedded into city plans as well as in panoramas. In city plans they allow for representing elevation levels in the flat projection. In panoramas contour lines considerably enhance the readability of the terrains particularly to overcome perspective distortion in terrain elevation. They allow for an exact comparison of altitudes in distant regions of the scene. Relatively dense contour lines can even completely replace the solid terrain surface (Figure 6).

In our geometric model elevation encodes creation time in terms of versions stored the primary model.

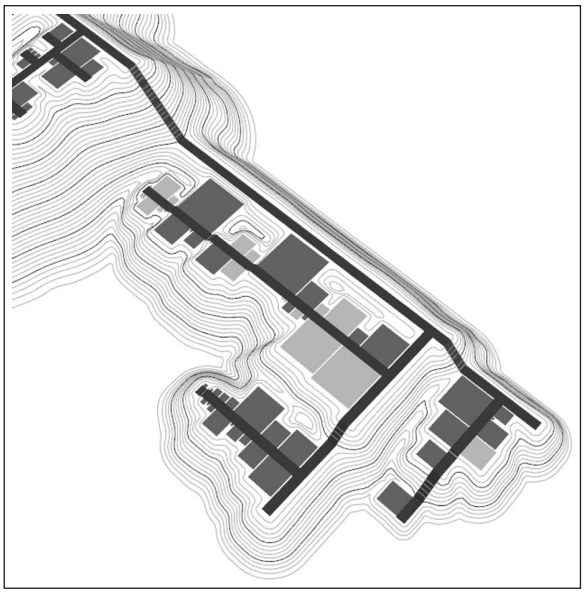

Figure 6. Contour lines on a 2.5D terrain.

Therefore, we can represent each of these versions by a contour line. If, for example, the primary model is populated on a daily or weekly basis then each contour line represents a corresponding day or week, respectively.

Module properties. In city plans modules are represented simply by their corresponding build plot (which already represents one property). Additional module properties can be represented only by coloring the building plot or by positioning simple 2D icons or labels at a module location.

In panoramas, however, we can make use of the third dimension and allow for arbitrary geometric bodies such as cylinders or cuboids to represent module properties as "buildings". A set of module properties can be mapped to geometric and visual building properties (see Figure 7(a)) such as height (e.g. for module size), base area (e.g. for module coupling), and color (e.g. for subsystem membership).

A more generalized and expressive form of buildings is a so-called property tower (Figure 7b). They consist of a configurable number of *segments* in the form of cylinders or cuboids which are stacked on each other. Height, radius, color and transparency of each segment (in the case of using cuboids' depth and width instead of radius) can be used to encode specific properties of the module.

Property towers have to be tailored to each application scenario. One possibility to do this is by specifying the number of segments to be stacked above each other in advance and by defining a separate mapping from module properties (such as size, complexity) onto each segment's visual properties. In this case each segment represents a particular set of module properties.

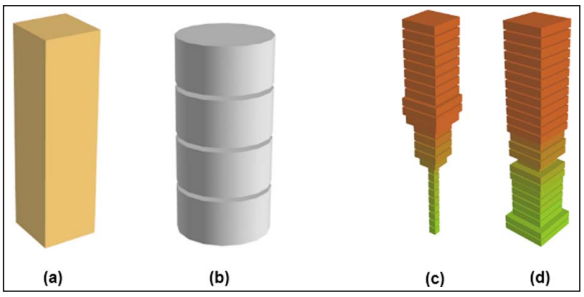

Figure 7. Property towers and evolution towers.

Another possibility is to represent each module version with a separate segment and to stack these segments for each module as the module evolves. Then each segment represents the same set of module properties for a particular version of the software system. These specialized property towers are called *Evolution Towers* because they reveal the evolution of module properties. Two examples of such evolution towers are shown Figure 7(c) and (d). Both represent 25 versions (from bottom to top) of an example module but with different visual mappings. In the left tower we can see a correlation between segment color and segment size, i.e. the respective properties (Lines-Of-Code [LOC] represented by color, module coupling represented by size) evolve similarly. In contrast, in the right tower both properties are not correlated (LOC represented by color and complexity represented by size) but evolve independently for this module.

By statically representing module properties for a sequence of module versions, evolution towers easily allow for detecting evolutionary patterns such as pulsar, supernova, or white dwarf (as described in ref. 27). Thus, they are especially useful during fine-grained analysis of a particular module's evolution.

We used this basic concept for visualizing software modules in various ways for building a number of different towers tailored toward representing specific sets of information.

Module dependencies. In software city visualizations logical (i.e. syntactic or semantic) dependencies between modules are usually displayed as arcs between buildings. As this often leads to occlusion when too many relations are displayed at once, techniques such as edge bundling[28] can be used. Besides this standard dependency representation in the EvoStreets approach, we use the hierarchical street system for routing and bundling module interconnections along the street system. An example can be found in **Figure 14**.

Comprehension scenarios

Software city visualizations in general provide an overall picture of a software systems structure. They support building a common mental map by providing a unique location for each system element. As discussed in the literature (see "Related work" section), this can be used for the illustration and communication of properties of system elements.

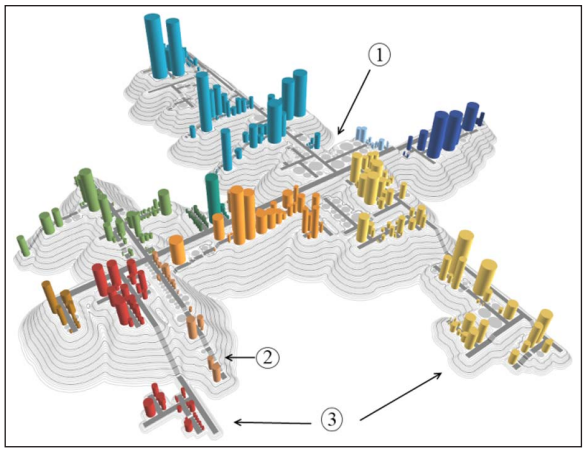

Figure 8. Size-based evolution map for CrocoCosmos.

The EvoStreets approach particularly focuses on aspects of evolution by capturing the information about the creation time of system elements in the layout and by creating stable layout sequences during the evolution of systems. This allows for supporting several application scenarios which are based on information about the history of development activities and their effects on the system structure. Typical questions developers and project managers in this context ask are "How did this component evolve?", "Who is working on what?",[29] or "Who owns a piece of code?" and "What is the history of a piece of code?".[30]

In the sequel we illustrate how the EvoStreets thematic maps can help to answer such questions.

Metrics and system evolution

A first example is the *evolution map* (Figure 8), which uses contour lines to show the versions of each subsystem and module. Simple property towers represent modules. Their base area is proportional to the module's dependencies, the height is proportional to the module's size (number of contained elements), and the color indicates the containing top-level subsystem.

On the evolution map (Figure 8) one can easily recognize the visual patterns mentioned before: (1) regions of empty plots represent parts of the system that were removed or migrated to other subsystems; (2) a steadily growing subsystem is represented by a continuously declining street. The buildings along that street are relatively small, and some of them even have been removed again; (3) peripheral suburbs represent local system extensions that were added to the system during a certain period of time. The red suburb in

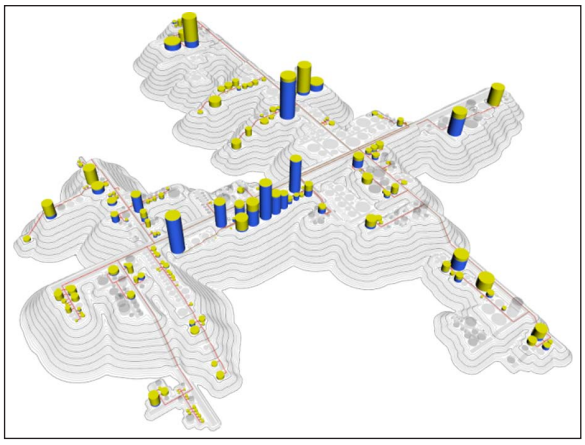

Figure 9. Dependency-based evolution map for CrocoCosmos.

Figure 8 mainly consists of small buildings representing small modules with low coupling. In contrast, the yellow suburb consists of some relatively large buildings which represent large and strongly coupled modules. Thus, the yellow suburb represents a larger functional extension of the system than the red suburb does.

Another example of an evolution map is shown in Figure 9. Here each module is represented by a tower with two segments. The height of each segment represents the modules incoming (blue) and outgoing (yellow) structural dependencies, i.e. number of modules using the module or being used by the module. As the base area is proportional to the dependencies, this property is strongly emphasized in this type of map. It can easily be seen that there is one subsystem in the center which contains several old modules with very high incoming coupling. These old modules implement the system's central data structure.

As these examples show, evolution maps on the one hand give a high-level overview of a systems decomposition and structural evolution, and thus they can be very helpful, especially for project newcomers. Besides that, evolution maps allow one to study particular aspects such as size or dependencies in the context of the systems evolution: it becomes apparent whether quality problems such as complex, large, or strongly coupled modules affect mainly new parts of the system or whether these are old problems, which is especially important for directing restructuring efforts.

Understanding modification history

Another application scenario uses a panorama map together with simple property towers depicting the number of modifications as height and being colored according to their last modification date. The resulting *modification history map* (Figure 10) shows three modification history properties for each component, i.e.

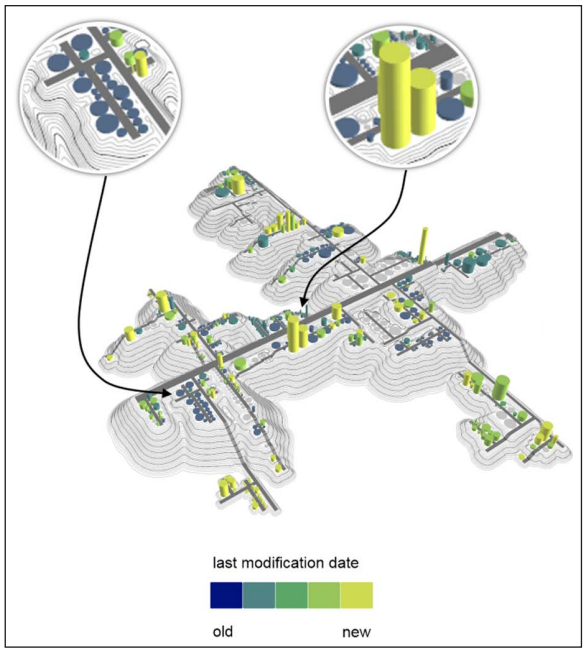

Figure 10. Modification history map for CrocoCosmos.

creation date, last modification date, and number of modifications.

On the modification history map one can easily distinguish different situations: (1) flat blue buildings on high elevation levels, often located on separated plateaus, represent modules which have been part of the system for a long time, which have barely been modified after their initial creation, and which have not been modified for a long time. There are two possible reasons for such a situation – either these modules implement very stable functionality or their functionality is not used any longer, and therefore they are not maintained at all. In this case these modules and subsystems could be candidates for being removed from the system. (2) Tall yellow buildings at high elevation levels represent old modules that have been part of the system for a very long time; they have been changed a lot since their initial creation but were modified very recently. Buildings like these might indicate potential design problems.

Modification history maps can be used in different application scenarios. For example, they support the identification of code regions which are often changed, and thus might be candidates for reviews and reengineering activities. They can also support the communication between developers, managers, and customers, e.g. when illustrating potential risks of customer-side feature requests. They help to discuss which parts of the system would have to be modified and whether these parts are well understood (as they were often and very recently modified) or whether they are old and rarely modified parts for which expertise might be missing.

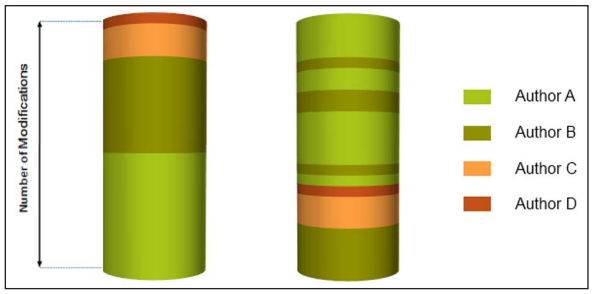

Figure 11. Authorship towers.

Understanding authorship

Another set of questions is addressed by the *authorship map*. Again, a panorama map is used as the basis. Modules are represented by so-called *authorship towers* (see Figure 11). Authorship towers show module modifications made by a selected set of authors. For this purpose, they consist of several colored segments, each of which indicates an author-specific modification of the respective module. The color of each segment is used to encode the author who made the respective modification. Authors can be selected interactively; modifications by authors not contained in this selected author set are represented by transparent segments. Thus, the total height of each authorship tower always represents the number of all modifications applied to the corresponding module.

There are two possibilities for ordering the segments. First, they are sorted by author, i.e. all segments for modifications performed by the same author are stacked to form one composite segment for that author (Figure 11(a)). Second, segments are sorted in ascending temporal modification order and displayed bottom up, i.e. segments for old modifications are displayed at the bottom whereas new modifications are displayed on top of the tower (Figure 11(b)). Whereas the first kind of ordering supports to rate the familiarity of authors with particular modules, the second kind of ordering is particularly useful to detect drifts in code ownership.

Figure 12 shows the authorship map for our example system. The contributions of the main developer are shown in dark green. From the early days of the project until now he is the author and main contributor of many classes throughout the system. In contrast, the dark-brown and the light-brown developers are authors of more recent extensions of the system. Both are specialists for local and clearly separated parts of the system. The yellow developer was the author of an early extension of the system, which also required some adaptations in many existing classes. This developer was active for only a certain period of the development history.

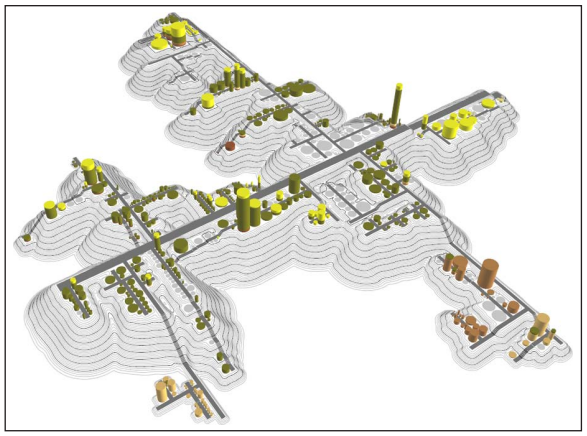

Figure 12. Authorship map for CrocoCosmos.

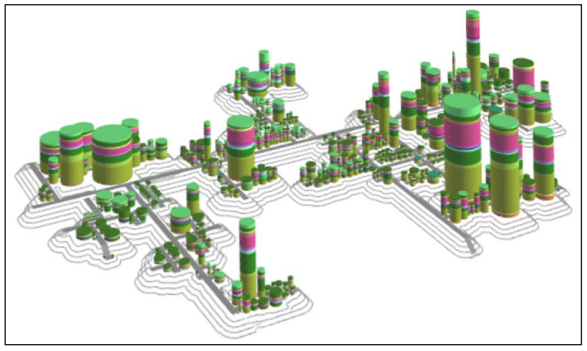

Figure 13. Authorship map for jMonkeyEngine.

The authorship map for a part of the 3D graphics library jMonkeyEngine in Figure 13 shows a different situation with respect to code ownership and the contributions of different developers. Six developers were modifying almost all classes of the system. There seem to be no clear responsibilities of developers for particular parts of the system.

Besides analyzing code ownership, authorship maps can be used to detect implicit dependencies among subsystems, i.e. dependencies that are not explicitly encoded as structural dependencies. Figure 14 shows all structural module dependencies for those two subsystems that were developed by the light-brown developer. Obviously there is no direct structural dependency between these subsystems. Instead they are related to some old modules developed by the dark-green developer.

The reason for this is that the light-brown developer first developed a new graph layout approach and added this implementation to the layout subsystem at the lower-right region of Figure 14(a). The results of this computation are stored in the central data structure. Later on he developed an analysis subsystem to evaluate layout properties such as stability and compactness of his new approach and added this implementation to

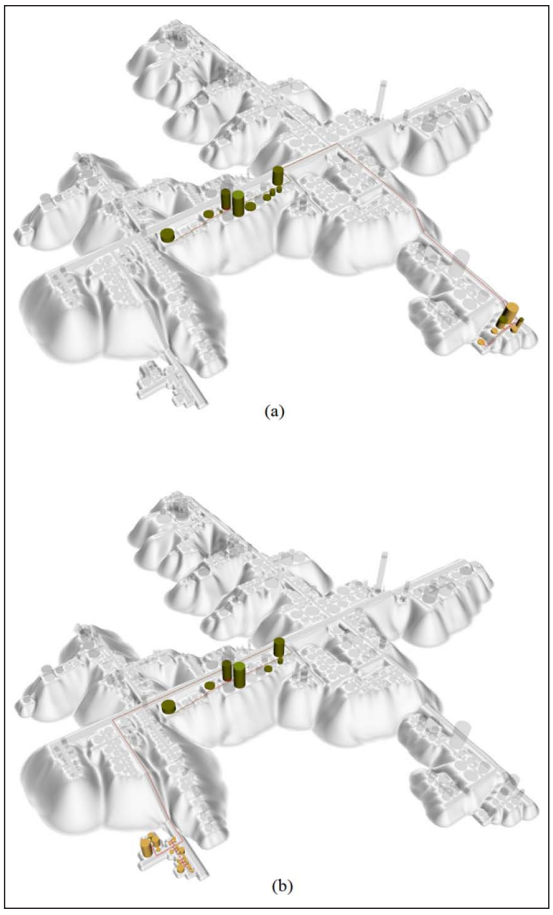

Figure 14. Structural dependencies between (a) layout subsystem and central data structure. (b) central data structure and analysis subsystem.

the analysis subsystem at the lower-left region of Figure 14(b). For this purpose all layout results must be read from the central data structure again. This created an indirect data dependency between the computation subsystem and the analysis subsystem.

While this example very clearly points to a non-syntactical data dependency among two subsystems hidden dependencies certainly cannot be detected as clearly as shown above for the general case. This is especially true because programmers usually work on several distinct and independent parts of the software. Thus, when studying authorships to detect such dependencies it might be reasonable to consider only those modifications of a particular author that were performed during a relatively short period of time.

Discussion

Complex comprehension scenarios

Each of the comprehension scenarios described above concentrates on a specific aspect of the underlying software system (such as authorship, modifications, module coupling, etc.). Many different approaches for the analysis of several aspects of software systems such as quality and team organization are described in the literature. The unique strength of the EvoStreets approach is that it allows for visualizing all of these aspects in an explicit relation to the systems evolution. The uniform way of representing these aspects in thematic maps coherently derived from a common geographical base model supports the cognitive process for integrating formerly separated aspects consistently into more complex mental maps. This provides deeper insights and allows for mastering more complex comprehension tasks.

Examples of such complex analysis scenarios are, for instance, the identification of correlations (such as test coverage with authorship) or the identification of critical system components (e.g. old parts of the software system with strong incoming dependencies that were not modified over a long period of time and for which the responsible authors are no longer available). All this information can be obtained by flipping between maps. Besides that, the approach allows for filtering the visualization for each of the data types mentioned above, e.g. to show only modules modified by a given author or in the recent past or to show modules that were executed by a given test suite. For this purpose test-related data are represented by two specific property towers. For each module test coverage can be represented by two segments indicating the number of lines of code covered and uncovered, respectively, by all test suites. Test suites, i.e. modules that implement test cases, are represented by mapping the number of successful and failed test cases onto both segments in a similar way. To distinguish modules and test suites different forms of towers (cuboid and cylinder) are used.

Filtering the set of represented modules based on test data within other thematic maps helps to focus on particular parts of the system for understanding.

In the same way, run-time data collected during the execution of particular system functions (e.g. execution time, memory usage) can be used as filters for thematic maps. This would, for instance, allow easily combining the information on error-prone executions with information on the modification history or authorship. The CrocoCosmos[22] tool, which implements the EvoStreets approach, supports such a dynamic analysis for Java systems. It uses online code instrumentation to record execution data and integrates these data into the primary model during program runtime.

Re-engineering and refactoring

Primary models capture the structure of software systems in several versions. They do not, however, include restructuring information such as moving modules among subsystems. Thus, none of our models allows to

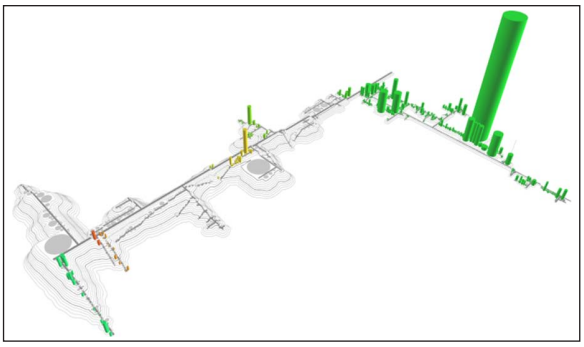

Figure 15. Size-based evolution history map for a heavily restructured system.

distinguish between whether a given system element is removed from the system or whether the system was restructured, i.e. the element moved to another subsystem. As a consequence, visualizations of systems which are often and heavily restructured tend to contain many disused plots.

An example of such a system is the PMD analysis tool. Figure 15 shows the evolution history map for PMD at its SVN repository revision 7000. PMD is written in Java; thus, streets represent Java packages and plots represent the 619 Java classes. During its development the system has been heavily restructured. Many of the elements originally positioned on the now-disused plots actually moved to the upper-right subsystem. An example is the huge tower at the upper right, which originally was positioned on the large empty plot at the lower left.

After such major restructurings, it could make sense to also restructure the secondary model by releasing the disused plots and creating a new layout starting from this point in development history.

Layout compactness and stability

In some situations the current layout approach produces very long streets, which make the layout very spacious. The compactness of layouts (i.e. the area actually used for building plots in relation to the total area spanned by the city) is an important factor for readability and orientation. There are two main reasons for very long streets. First, a subsystem contains many elements that are sequentially arranged along its street representation. An example for such a layout is shown in Figure 17. Second, during system evolution elements are frequently added to and removed from a subsystem. Adding a new element increases the length of its subsystem street by extending it and attaching the new element at its end. On the other hand, removing the element from the subsystem does not decrease its street length. Thus, even small subsystems with a changeful history might be represented by long streets, which have negative impact on the

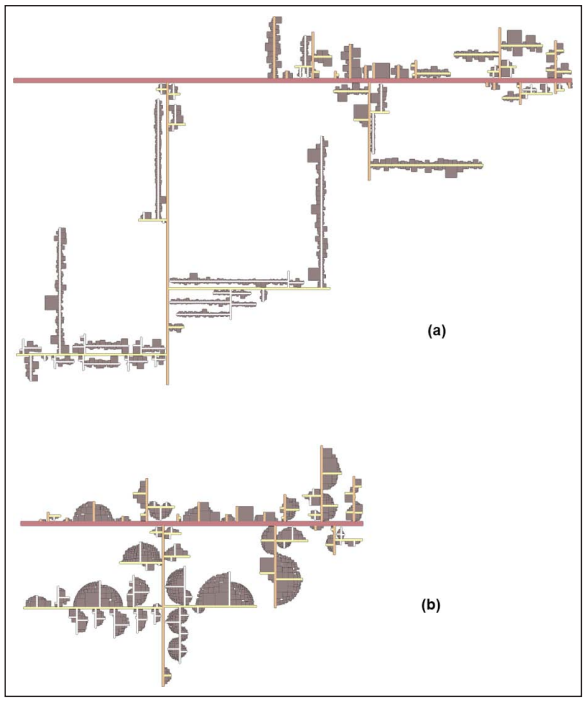

Figure 16. ArgoUML (1585 Java classes). (a) Long streets causing low layout compactness. (b) Compact layout based on a shortest-distance packing strategy.

layout compactness. An example for such a layout is shown in Figure 15.

To overcome these compactness problems we modified our layout algorithm by combining it with a packing-based layout in such a way that neighboring building plots on the same elevation level are arranged as rectangular districts along the street. Figure 16(a) shows the EvoStreets layout for one revision of the ArgoUML software system. Owing to some very long streets this layout creates a lot of empty space. In contrast, the lower part of Figure 16(b) shows a layout of the same system that was obtained by placing buildings as close as possible to the street origin, which yields circular districts on both sides of each street. As this example shows, reducing the length of streets significantly reduces the overall layout size and yields more compact layouts, possibly at the cost of more occlusion and less stability.

As neighboring buildings can be grouped together into the same district only if they are positioned on the same elevation level, this effect strongly depends on the temporal resolution of the primary model. If the primary model is populated on a daily basis, then in general only a few new modules are added to the system during this short period of time. If, however, the primary model is populated for each major project milestone only, then presumably many more modules will be added to the system during this long period of time. In the first case, the visualization contains many elevation levels and relatively small districts. In the latter case, the visualization contains relatively few elevation

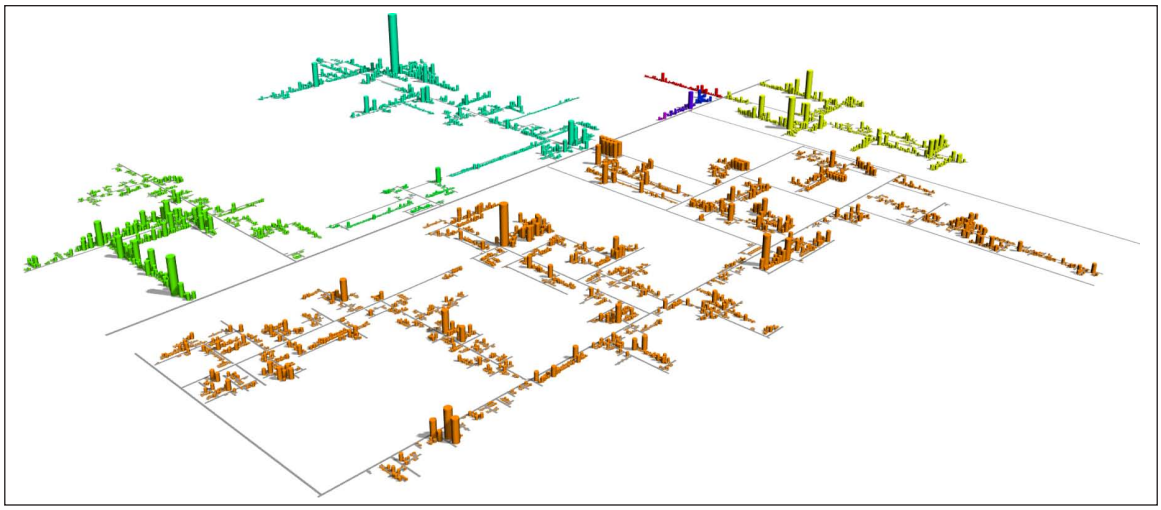

Figure 17. Java Development Kit (JDK 6), ca. 14.000 Java classes.

levels and larger districts. Thus, the use of districts has a larger effect on layout compactness for coarse-grained temporal resolution of the primary model.

Fields of application

The examples in this paper represent software systems on a relatively fine-grained abstraction level, i.e. the module level. These modules (classes for Java systems and files for C/C++ systems) are, however, mainly the domain of programmers. In contrast, software architects or project managers are more interested in architectural units such as components and component hierarchies. The EvoStreets approach is essentially built on general graph-based primary models, which are independent from the programming language and abstraction level. It can easily be adapted to represent multi-language software systems on an architectural level just by populating the primary model with architectural units instead of design-level modules. The derived secondary and tertiary models visualize the system from an architecture perspective. This, however, requires the architectural components to be organized in a strict hierarchy tree so they can be mapped onto the hierarchical street system. Consequently, shared components would have to be represented redundantly.

The use of a general graph-based primary model basically allows for visualizing any kind of hierarchically structured and evolving data, for example organizational structures in companies. Thus, the approach presented in this article is not restricted to the visualization of software systems.

Tool support

The EvoStreets approach is completely tool supported: primary models are automatically extracted from project repositories using the Eclipse IDE or the Sotograph analysis tool. Secondary and tertiary models are created using our visualization tool CrocoCosmos.[22,20] It implements all visualization concepts presented in this paper and provides rich interactive parameterization capabilities for the generation of the secondary and tertiary models.

Each of the representation elements can be configured interactively. In authorship towers, for example, the authors to be visualized can be selected interactively. For the interaction with and navigation in the 3D scene standard navigation techniques, coordinated views, stereo projection, and the use of special input devices (e.g. 3D mouse) are provided. These interaction and navigation capabilities are crucial in 3D scenes to overcome occlusion problems and to fully explore the visualization. Additionally, the tool features interactive animations for stepping forward and backward through the development history to study a software systems evolution in detail.

The visualization part of CrocoCosmos is build on top of the jMonkeyEngine, an open-source graphics library written in Java. It proved to be very capable of handling the visualization of large real-world systems of more than 10,000 classes (Figure 17).

Experiences

We conducted an exploratory study to determine the strengths and weaknesses of the city layouts. For this purpose we visualized 21 open-source systems with sizes ranging from only a few hundred classes to approximately 14,000 classes in the JDK 6 system shown in Figure 17.

An important finding of this study is that low compactness is in fact a rather serious problem that often appears, especially in larger systems. As described

above, this problem can significantly be reduced by arranging neighboring buildings on the same elevation level in rectangular districts.

The landscape patterns described above are not specific phenomena of our example systems. We found all the patterns during this exploratory study, although not every pattern appeared in each system. There were strongly restructured systems as the PMD system described above. Also, we found systems that continuously grew without any major restructurings.

For systems where much of the growth takes place at the beginning, most of the city structure is placed on high elevation levels; only a few new structures are added on lower elevation levels. For such situations we observed that evolution became rather difficult to see as the larger older structures dominate the visualization. Generally speaking, this problem arises whenever relatively small modifications are applied to relatively large systems. On the one hand, this behavior might be desirable as small changes do not disrupt the overall city structure. On the other hand, we conclude a need for additional means supporting the readability of particularly smaller software modifications expressed in the layout.

We conducted a second study with an industrial partner, a large logistics company. On the company side, the study involved the quality assurance team, which is responsible for monitoring the quality of externally developed code. The study included two iterations.

During the first iteration we looked at a medium-sized Java system with 354 Java classes in five major project releases. We produced static visualizations for several scenarios and returned them together with a brief textual description for each. We discussed these visualizations with the quality assurance team and obtained the following results.

From a representational point of view, the team pointed out the aesthetics and readability of the visualizations, though the latter still should be improved. On the one hand, the readability of the hierarchical system decomposition represented by the city layout was highly appreciated. On the other hand, it would be rather difficult to read the elevation information of the landscape provided via contour lines. In part this may be due to the small number of versions we looked at, but nonetheless it also suggests that another additional support mechanism, such as the use of cartographic height coloring or labeling, is necessary.

When inspecting the first system, we identified many areas of zero-height buildings distributed throughout the city for those scenarios where typical-quality data such as bug numbers were mapped onto the building heights. At first we supposed this to be a blind spot caused by a faulty configuration of the quality system and asked for the reasons. But, in fact, this blind spot is intentional as it points to test code and generated code that are not monitored with analysis tools. We offered to remove this code from the visualization, but the team explicitly rejected this idea as it would be important to know about this code and how it is distributed in the system; however, it would be rather helpful for distinguishing among hand-written code, generated code, and test code representations by using additional means such as shape or surrounding ground color.

One of the responsibilities of the quality assurance team is to monitor the evolution of so-called risky code and unfinished code. Both qualities can numerically be quantified using specific metrics. We obtained the respective data, integrated them into our primary models, and derived respective tertiary models to support this customer-requested scenario during the second iteration.

For the second iteration we used a larger system with about 1500 classes again in five major project releases. Again we provided static visualizations for several scenarios including the additional risky code and unfinished code scenarios and discussed them in an informal interview session.

The avoidance of risky code has explicitly been stated as a quality requirement, and the number of occurrences of unfinished code should decline to zero at least until the final project revision is reached. The visualizations largely matched both expectations: risky code has been very rare throughout the whole evolution, and the unfinished code continuously decreased until the final version has been reached. The quality assurance team explicitly pointed out that the major advantage of using these visualizations is that it takes only a few moments to obtain an overview of risky and unfinished code parts, their distribution throughout the system, and their development compared with the previous revision. It would be very much like watching a movie when clicking along the versions and seeing, for example, unfinished code continuously decrease to zero. This is, in fact, a quality that results from the visualization's stability against structural changes.

We provided both 2D and 3D maps and asked for an assessment and preference. Because building height is not visible in 2D maps, we included building shadows to indicate high buildings. Interestingly, 3D maps were assessed as more valuable as they would allow for identifying correlations between quality attributes more easily. Occlusion occurred, but it would not be problematic for two reasons: first, from the quality assurance perspective, interesting parts are usually represented as higher buildings; second, if the visualization can interactively be viewed from different angles then occluded buildings could easily be detected by rotating the scene once.

Summary and outlook

In this article we described a three-staged visualization approach adopted from cartography for the systematic visualization of large software systems which fosters a systematic use of map-making techniques. Each of these stages deals with a specific model. The primary model captures the structure of software systems and product- and process-related data, and their evolution over time. Primary models are the logical base model of all visualizations.

The secondary model adds geometric information to the primary model. In this context, the main contribution of this paper, the novel EvoStreets layout approach for software cities based on the hierarchical system, decomposition and component ages were presented. This approach produces geometrically stable city layouts for evolving software systems, i.e. systems that are still being developed. It uses landscape elevations to directly represent a software systems development history in the layout. Owing to this design decision and the addition of a 2.5D terrain model underlying the hierarchical street system, novel thematic maps can be designed.

The layout approach, however, has some shortcomings. Visualizations of systems that are often restructured or that have large subsystems tend to be not very compact. Therefore, we are currently studying modifications of our layout algorithm to achieve both stable and compact layouts. Furthermore, the stability and compactness of EvoStreets layouts has to be compared with other approaches proposed in the literature. An initial study[23] is being extended to cover more layout approaches and a larger set of real-world examples.

Tertiary models are derived from secondary models by transformations such as projections, coloring, or superposition of symbols and diagrams. Such thematic maps are designed to support specific application scenarios. To represent various module properties in a uniform way we designed so-called property towers. We explored several such specific application scenarios and demonstrated how our specific maps can support these scenarios. Such application or comprehension scenarios were (and still are) derived and discussed in cooperation with several industrial[31] partners.

Funding

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

Note

* This article is an extended and revised version of Ref. 1

References

1. Steinbrückner F and Lewerentz C. Representing development history in software cities. In: *Proceedings of the 5th international symposium on software visualization (SOFTVIS '10)*, 2010, Salt Lake City, USA, 25–26 October 2010, pp.193–202. New York: ACM Press.
2. Panas T, Epperly T, Quinlan D, et al. Communicating software architecture using a unified single-view visualization. In: *ICECCS, 12th IEEE international conference on engineering complex computer systems (ICECCS 2007)*, Auckland, New Zealand, 11–14 July 2007, pp.217–228. DC, USA: IEEE Computer Society Washington.
3. Kuhn A, Loretan P and Nierstrasz O. Consistent layout for thematic software maps. In: *Proceedings of 15th working conference on reverse engineering*, Antwerp, Belgium, 15–18 October 2008, pp.209–218. Los Alamitos, CA: IEEE Computer Society Press.
4. Balzer M, Noack A, Deussen O, et al. Software landscapes: visualizing the structure of large software systems. In: *VisSym 2004, symposium on visualization*, 2004, pp.261–266. Eurographics Association.
5. Knight C and Munro M. Comprehension with[in] virtual environment visualisations. In: *Proceedings of the IEEE 7th international workshop on program comprehension*, Pittsburgh, PA, 5 to 7 May 1999, pp.4–11. Washington, DC: IEEE Computer Society.
6. Wettel R and Lanza M. Visualizing software systems as cities. In: *Proceedings of VISSOFT 2007 (4th IEEE international workshop on visualizing software for understanding and analysis)*, Banff, Canada, 24–25 June 2007, pp.92–99. Los Alamitos, CA: IEEE Computer Society.
7. Knight C and Munro MC. Virtual but visible software. In: *International conference on information visualization*, London, UK, 19–21 July 2000, pp. 198–205. Los Alamitos, CA: IEEE Computer Society.
8. Charters SM, Knight C, Thomas N, et al. Visualisation for informed decision making: from code to components. In: *International conference on software engineering and knowledge engineering (SEKE '02)*, 2002, pp.765–772. New York: ACM Press.
9. Panas T, Berrigan R and Grundy J. A 3D metaphor for software production visualization. In: *International conference on information visualization*, London, England, 16–18 July 2003, pp.314–319. Los Alamitos, CA: IEEE Computer Society.
10. Alam S and Dugerdil P. EvoSpaces visualization tool: exploring software architecture in 3D. In: *Proceedings of 14th working conference on reverse engineering (WCRE 2007)*, Vancouver, Canada, 28–31 October 2007, pp.269–270. Washington, DC: IEEE Computer Society.
11. Alam S and Dugerdil P. EvoSpaces: 3D visualization of software architecture. In *Proceedings of SEKE 2007 (the 19th international conference on software engineering and knowledge engineering)*, 2007, pp.500–505. Washington, DC: IEEE Computer Society.
12. Dugerdil P and Alam S. Execution trace visualization in a 3D space. In: *Proceedings of ITNG 2008 (5th international conference on information technology: new generations) Third international conference on information technology:*

new generations, Las Vegas, NV, 7–9 April 2008, pp. 38–43. Los Alamitos, CA: IEEE Computer Society.
13. Langelier G, Sahraoui H and Poulin P. Visualization-based analysis of quality for large-scale software systems. In: *Proceedings of the 20th IEEE/ACM international conference on automated software engineering (ASE '05)*, 2005, Long Beach, CA, pp.214–223. New York: ACM Press.
14. Wettel R and Lanza M. Program comprehension through software habitability. In: *Proceedings of ICPC 2007 (15th international conference on program comprehension)*, 2007, Banff, Alberta, pp.231–240. Washington, DC: IEEE Computer Society.
15. Wettel R and Lanza M. Visually localizing design problems with disharmony maps. In: *Proceedings of the 4th ACM symposium on software visualization*, 2008, pp. 155–164. New York: ACM Press.
16. Wettel R and Lanza M. Visual exploration of large-scale system evolution. In: *Proceedings of the 15th working conference on reverse engineering (WCRE '08)*, Antwerp, Belgium, 5–18 October 2008, pp.219–228. Washington, DC: IEEE Computer Society.
17. Langelier G, Sahraoui H and Poulin P. Exploring the evolution of software quality with animated visualization. In: *Proceedings of the 2008 IEEE symposium on visual languages and human-centric computing (VLHCC '08)*, Herrsching am Ammersee, Germany, 15–19 September 2008, pp.13–20. Washington, DC: IEEE Computer Society.
18. Diehl S and Görg C. Graphs, they are changing. In: *Revised papers from the 10th international symposium on graph drawing* (eds SG Kobourov and MT Goodrich), *Lecture Notes in Computer Science*, 2002, vol. 2528, pp.23–30. London: Springer-Verlag.
19. Collberg C, Kobourov S, Nagra J, et al. A system for graph-based visualization of the evolution of software. In: *Proceedings of the 2003 ACM symposium on software visualization (SoftVis '03)*, 2003, Las Vegas, Nevada, pp.77–ff. New York: ACM Press.
20. Noack A. *Unified quality measures for clusterings, layouts, and orderings of graphs, and their application as software design criteria*. PhD Thesis, Brandenburg University of Technology at Cottbus, Cottbus, Germany, 2007 (URN: urn:nbn:de:kobv:co1-opus-4046).
21. Slocum TA, McMaster RB, Kessler FC, et al. *Thematic cartography and geovisualization*. 3rd ed. Upper Saddle River, NJ and London: Pearson Prentice Hall, 2010, p.561.
22. Lewerentz C and Noack A. CrocoCosmos—3D visualization of large object-oriented programs. In: Jünger M and Mutzel P (eds) *Graph drawing software*, 2003, pp. 279–297. Springer-Verlag.
23. Mannl U. *Evaluation of layout stability of software cities*. Master Thesis, University of Technology, Cottbus, Germany, 2010.
24. Bridgeman S and Tamassia R. Difference metrics for interactive orthogonal graph drawing algorithms. In: *Graph drawing, volume 1547, Lecture Notes in Computer Science*, pp.57–71. Berlin/Heidelberg: Springer, 1998.
25. Lyons KA, Meijer H and Rappaport D. Algorithms for cluster busting in anchored graph drawing. *J Graph Algorithm Appl* 1998; 2: 7–17.
26. Balzer M and Deussen O. Level-of-detail visualization of clustered graph layouts. In: *APVIS 07, 6th international Asia-Pacific symposium on visualization*, Sydney, Australia, 5–7 February 2007, pp.133–140. IEEE.
27. Lanza M. The evolution matrix: recovering software evolution using software visualization techniques. In: *Proceedings of the 4th international workshop on principles of software evolution (IWPSE '01)*, 2001, Vienna University of Technology, Austria, pp.37–42. New York: ACM Press.
28. Holten D. Hierarchical edge bundles: visualization of adjacency relations in hierarchical data. *IEEE Trans Visual Comput Graphics* 2006; 12(5): 741–748.
29. Fritz T and Murphy GC. Using information fragments to answer the questions developers ask. In: *Proceedings ICSE 2010 (32th international conference on software engineering)*, Cape Town, South Africa, 2–8 May 2010, pp. 175–184. Washington, DC: IEEE Computer Society.
30. DeLine R, Venolia G and LaToza TD. Maintaining mental models: a study of developer work habits. In: *Proceedings of 28th international conference on software engineering (ICSE'06)*, 2006, pp. 492–501.
31. Lewerentz C and Steinbrückner F. SoftUrbs: visualizing software systems as urban structures. Computer Science Reports 02/2009, December 2009. Cottbus: Brandenburg University of Technology at Cottbus.