3-D Visualization of Dynamic Runtime Structures in Applications

Marcus Ciolkowski
Principal IT Consultant
QAware GmbH
Aschauer Str. 32
D-81549, München, Germany
marcus.ciolkowski@qaware.de

Simon Faber
kasai GmbH
An der Stiftsbleiche 11
D-87439 , Kempten, Germany
s.faber@kasasi.de

Sebastian von Mammen
University of Würzburg
Head of Group Games Engineering
D-97074, Würzburg, Germany
sebastian.von.mammen@uni-wuerzburg.de

ABSTRACT

Continued development and maintenance of software requires understanding its design and behavior. Software at runtime creates a complex network of call–callee relationships that are hard to determine but that developers need to understand to optimize software performance. Existing tools typically focus on static aspects (e.g., Structure101 or SonarQube), or they are difficult to use and require high expertise (e.g., software profiling tools).

Unfortunately, these dependencies are hard to derive from static code analysis: For one, static analysis will reveal potential call–callee relationships not actual ones. Second, they are often difficult to detect, since information systems today increasingly use abstraction patterns and code injection, which obscures runtime behavior.

In this paper, we present our efforts towards accessible and informative means of visualizing software runtime processes. We designed a novel visualization approach that utilizes a hierarchical and interactive 3-D city layout based on force-directed graphs to display the runtime structure of an application. This promises to reduce the time and effort invested in debugging programming errors or in finding bottlenecks of software performance.

Our approach extends the city metaphor for translating programmatic relationships into accessible 3D visualizations. With the identified goals and constraints in mind, we designed a novel visual debugging system, which maps programming code structures to 3D city layouts based on force-directed graphs. Exploration of the animated visualization allows the user to investigate not only the static relationships of large software projects but also its dynamic runtime behavior.

We conducted a formative evaluation of the approach with a preliminary version of a prototype. In a series of six interviews with experts in software development and dynamic analysis, we were able to confirm that the approach is useful and supports identifying bottlenecks. The interviews raised and prioritized potential future improvements, several of which we implemented into the final version of our prototype.

IWSM Mensura '17, October 2017, Gothenburg, Sweden

ACM ISBN .....
https://doi.org/....

CCS CONCEPTS

• **Human-centered computing** → **Graph drawings**; **Visual analytics**; Empirical studies in visualization; • **Software and its engineering** → **Extra-functional properties**; *Software usability*; *Agile software development*; • **Social and professional topics** → *Software management*; • **Information systems** → Enterprise information systems;

KEYWORDS

software quality, runtime metrics, call structure, 3D visualization

ACM Reference format:
Marcus Ciolkowski, Simon Faber, and Sebastian von Mammen. 2017. 3-D Visualization of Dynamic Runtime Structures in Applications. In *Proceedings of International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement, Gothenburg, Sweden, October 2017 (IWSM Mensura '17),* 10 pages.
https://doi.org/....

1 INTRODUCTION

Continued development and maintenance of software requires understanding its design and behavior. Without additional efforts, software processes are both complex and invisible. Therefore, understanding software design and behavior can be a time-consuming and challenging task. Mastering this task is a key competency in the contested software development market and accounts for 40 to 60% of the time invested in software maintenance jobs [1]. There are several tools that help in shedding light on specific aspects of software. Yet, not every facet can be covered. The "broken window" metaphor [18], for instance, stands for an important aspect in software quality management. Similar to a broken window in buildings that sends signs of neglect and invites vandalism, finding and fixing even small and trivial defects as soon as they are detected prevents their potentially far-reaching repercussions. Therefore, it is mandatory to measure and collect indications of quality deficits continuously. Tools such as SonarCube [5] support continuous analysis but focus mostly on static aspects of software quality. However, dynamic behavior of software is often neglected. For instance, call–callee dependencies between structural artifacts, such as packets or methods, and drops in performance caused by inappropriate dependencies are difficult to spot. This is aggravated by the intensive use of abstraction patterns and code-injection techniques in modern information systems. Analytics tools that can reveal such relationships are complex, have a steep learning curve and are typically only used by software analytics experts. The majority of developers relies on reading source code, which is time-consuming and error-prone.

The field of software visualization addresses the outlined problem of understanding and improving software by generating visualizations that convey the required information about designs and processes [11, 21]. In 2002, Charters et al. proposed mapping software to cities [8]. Cities host static elements including buildings and traffic infrastructure. Wettel et al. focused on this aspect of cityscapes to visualize static elements of object-oriented software designs. For instance, in Code City [37], buildings represent classes, quarters represent packages. In principle, all those software elements can be considered static that are accessible by merely browsing the program's source code and without running the software program. Complementing static city elements, dynamic processes such as method calls could be animated as vehicles driving from one building to another. The majority of people is familiar with the city metaphor [36], which promotes timely and effective orientation of developers [15]. The city metaphor naturally constraints visualization in three-dimensional space. For instance, buildings do not hover above the ground (yet), which mitigates issues of accessibility that may arise in 3D visualizations that extensively utilize the third dimension as well [10, 12].

In this paper, we present our efforts towards developing a prototype for visualizing the dynamic structure of software runtime processes. Software cities have been used highlighting code quality [35], visualizing engineering progress [33], shedding light on system behaviors for the purpose of reverse engineering [2], educating about software design [3], and for presenting the coverage of tests [32]. In addition, there have been preceding works on visualizing dynamic call–callee relationships and showing performance bottlenecks (e.g., [16, 17]). We have systematically reviewed their contributions with respect to our goals, combined them and extended them, where necessary. In particular, we started with the definition of three different use cases from which particular system requirements emerged. This process and its results are detailed in Section 2. Next, in Section 3 we stress the contributions of related work that we improved upon. In Section 4 we present our approach. Section 5 provides details about our evaluation methodology and the evaluation's results. We conclude with a summary and an outlook on potential future work in Section 6.

2 USE CASES AND REQUIREMENTS

The application logic of an object-oriented software system is distributed across numerous methods. During runtime, the position of individual processing threads leaps from one of these methods to the next. These events are referred to as method calls. One of them can lead to any number of subsequent method calls. We consider runtime behavior the timed succession of method calls during the runtime of a software. Our visualization of runtime behavior aims at serving three use cases:

- (U1) Software developers are supported in understanding the runtime behavior.
- (U2) Software developers are supported in localizing potential performance issues or bottlenecks.
- (U3) Experts for dynamic analysis of applications are supported in gaining an overview of potential bottlenecks.

Software developers, the target group of U1, often do not have a comprehensive knowledge about a software's runtime behavior. Reasons for this include the system's overall size, the fact that the source code presents all possible call–callee interdependencies (and not only those executed at runtime), or the lack of an up-to-date specification of the project. This is additionally obscured by extensive use of code injection techniques and abstraction in modern information systems. The software developer adjusts, corrects or extends a specific aspect or functionality of the software. Use case U1 aims at supporting the software developer in understanding the runtime behavior of the code block he works on, so that unwanted side-effects can be avoided. Typical questions by software developers include [29, 31]: From which origin is a specific method called? How do method calls propagate through the system? Which classes communicate with each other? Which components call each other? In order to answer these questions, a visualization is effective if it shows call–callee interdependencies but does not overwhelm the developer by showing too many elements at once (which is true in all three use cases). In fact, the developer wants to adjust the level of detail of the visualization in accordance with his current focus.

Use case U2 again focuses on software developers as target group and refines U1. It assumes that they are not experts in runtime analysis, and hence they have little knowledge about using analysis tools such as profilers. When correcting, adjusting or extending existing code, software developers run the risk of introducing new bottlenecks which may cause long latencies or even failures during the use of the software. Therefore, U2 aims at supporting developers in identifying and localizing potential bottlenecks during development. Typical questions of a developer with respect to software performance include [9]: Which methods consume most of the time? Are there method calls that take unusually long to execute? Where do more than the average of method calls originate from? Where are queries stalled when propagated through the system? Accordingly, the visualization has to provide fine-grained insights in time consumption and the respective, involved method calls. It is effective, if it supports the developer in quickly spotting unusual behaviors such as short bursts of heavy computational loads or long computing periods and associated pieces of code. Individualized filtering mechanisms should be in place to allow for more elaborate inquiries. The visualization has to maintain the relationship to the structural artifacts so that performance bottlenecks can be directly related to specific pieces of code.

The task of dynamic analysis experts consists of identifying performance bottlenecks and of planning the implementation of countermeasures. To this end, they rely heavily on profiler tools, they analyze log files of the applications, and they use low-level metrics such as stack trace dumps or heap dumps (where runtime memory contents are written out for inspection). These tools and approaches generate large amounts of fine-grained data, which results in long analysis procedures. This analytical process can be supported by quick identification of bottlenecks that need to be investigated further. Typical questions by analytics experts include [9]: Are there clusters of methods that together consume a lot of time? Where do queries get stalled on their way through the system? Accordingly, the visualization should communicate the big picture of interdependencies and time consumption. It should further allow the analytics expert to specify various filtering criteria that allow him to direct his search for performance bottlenecks.

Based on the three use cases the basic requirements of expressiveness and effectiveness [25], the following, more detailed requirements can be inferred:

- R1 Identify static structures: The approach maintains the relationships between runtime behavior and structural artifacts of the code.
- R2 Visualize call-callee interdependencies: The approach provides a clear view on calls between the structural artifacts.
- R3 Visualize performance aspects: The approach provides various performance aspects associated with the structural artifacts.
- R4 Support a drill-down-principle: The approach reveals/hides details based on demand.
- R5 Support directed search: The approach supports the directed search for hot spots.

3 RELATED WORK

In order to identify those aspects of existing software city approaches that work well and those that need improvement, we systematically surveyed the literature following the SLR-guidelines after Kitchenham et al. [20], which loosely foresee the following steps: (1) Specify the research questions, (2) determine a search strategy, (3) determine a criterion for exclusion, (4) conduct selection process. More specifically, we first searched for relevant works that address requirements R1 to R4. We identified 14 related approaches in this step. Next, we excluded software city approaches that do not visualize any runtime information [3, 7, 8, 22–24, 32, 33, 36], that do not provide overviews of the runtime behavior [30, 34], and that do not visualize performance bottlenecks [2, 13]. Finally, ExplorViz [15, 16] and ThreadCity [17] were identified as most closely aligned with our goals. Details of paper selection criteria are documented in [14].

3.1 ExplorViz

In the application visualization of ExplorViz packages and classes are translated to city quarters or buildings which are depicted as boxes based on a static tree-map layout. Differences in type (package/class) are encoded in the boxes' colors. Quarters represented as flat, wide boxes reveal enclosed structural artifacts, whereas buildings hide them. The user can toggle between the two views in order to reach the desired level of detail. Nested dependencies in code are mapped to stacked elements in the cityscape of decreasing size. Accordingly, city quarters represent packets, whereas layers on top (i.e., buildings) represent hierarchically nested packages or classes. The height of buildings represents the number of instances of an element. Streets between buildings depict call-callee relationships, whereas the width of the streets indicates the number of calls and arrow depictions visualize the direction of the call. Upon selection of an individual building, only relations pertaining to the respective element are visualized. A modified tree-map algorithm [19] layouts the positioning of the buildings. The user can navigate freely in the city and zoom into places of interest. Runtime information can be navigated by means of a timeline, or be traced step by step. ExplorViz also empowers the user to filter method calls with respect to their consumed amount of time. Figure 1 gives an impression of ExplorViz' layout approach.

Aligned with (R1), ExplorViz visualizes the static structure of the targeted software, yet limited to packages and classes. Call–callee relations (R2) are depicted as streets between elements of potentially differing types. Replay of stack traces makes it possible to analyze the call dynamics over time. However, no effort is made towards clear and transparent visualization of these relationships—streets can overlap and intersect, which results in cluttered graphs, difficult to interpret [7]. Furthermore, sequences or more complex call dynamics are not visualized but need to be investigated step-by-step. Considering (R3), ExplorViz provides access to several performance measures encoded in street width and building dimensions. Unfortunately, these measures are fixed and means of extensions, for instance to integrate the execution time, are not provided. (R4) is realized as the user can navigate freely and focus on details of a specific level of the code hierarchy. However, there is no functionality to blend out arbitrary structural elements but descending on a specific hierarchy level requires that all the nodes above in the tree-map are unfolded as well. Directed search (R5) is fulfilled in that different metrics are displayed concurrently. However, specific search targets cannot be specified.

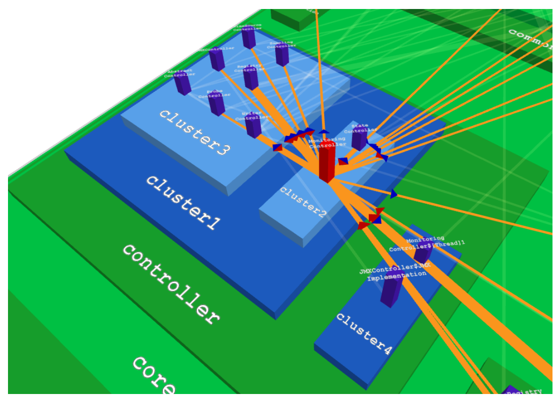

Figure 1: ExplorViz: Layout and display of dynamic relations as overlay over a static TreeMap layout.

3.2 ThreadCity

ThreadCity aims at the interactive exploration of multi-threaded systems [17]. The concrete goals are understanding the program structure and conducting performance analysis. The software is built on a 2D birds-eye city visualization, where gray streets represent packages and orthogonally extending, slightly smaller alleys depict (recursively) nested packages. Blue buildings represent classes that are part of the respective packages and are therefore aligned with the streets. This layout method is also referred to as the EvoStreets approach [33]. Differently colored lines of traffic flow from one building to another (and sometimes alongside each other) visualize call–callee relationships. Circle and bar diagrams are directly embedded in the city visualization and reveal the relative computational load caused by the respective packages and classes. The user can choose to visualize specific system threads, move the

2D camera and zoom into relevant areas. Streets together with all alleys and buildings can be faded out. Selecting individual streets or buildings reveals additional information and, in the latter case, blends in all traffic to and from the specific building. A timeline allows the user to leap to time intervals he is interested in. Figure 2 gives an impression of the layout approach of ThreadCity.

(R1), the need to visualize the static structure, is fulfilled by showing buildings and streets, depicting packages and classes. However, there is no option to visualize additional layers of the static hierarchy. The topology of the hierarchy is mapped to the topology of streets and alleys. Considering (R2), call-callee relationships are depicted as traffic flow lines among potentially different types of structural elements. Traffic animations on the respective sides on the road visualize directionality. The overview is quickly lost due to parallel flow lines and frequent overlaps at intersections. (R3), the visualization of performance measures, is addressed by projecting informative charts into the city visualization. However, these data are strictly limited to the amount of method calls and not configurable to reveal other metrics such as execution times. Due to the tight relationship between the code hierarchy and the visualization, arbitrary artifacts cannot be filtered out. In order to find a specific search target, as in (R5), ThreadCity provides overview diagrams at different levels of the hierarchy and the user can narrow down the search step-by-step. However, the metric cannot be adapted to the user’s search goals.

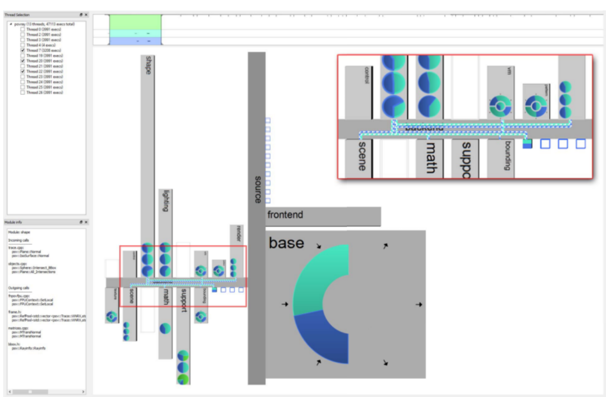

Figure 2: ThreadCity: EvoSreet layout and information display.

4 VISUALIZATION APPROACH

As detailed in the previous section, ExplorViz and ThreadCity support many of the identified requirements. Therefore, we adopt the metaphors of buildings representing static structural artifacts and streets/traffic depicting call–callee relationships. Variable dimensions of buildings and streets intuitively correspond to their respective importance, which is why we adopted these performance cues, together with the 3D perspective from ExplorViz. In both approaches, the user can drill down the code hierarchy to interactively find a balance between visual clutter and data coverage—we also adopted this mechanism. To overcome the shortcomings of both approaches, we additionally addressed the following issues.

(1) We introduced novel ways to comprehensively visualize the progression of call events.
(2) We allow the user to visualize structural artifacts at more fine-grained levels than packages and classes.
(3) We provide a flexible interface for defining additional performance measures.
(4) We allow the user to effectively direct the search for relevant system dynamics.
(5) We changed the city map layout to improve clarity and scalability.
(6) We made sure that arbitrary sets of visual elements can be hidden.

4.1 Data Collection

We generate the interactive software city from runtime information of executed Java byte code utilizing an existing software tool [28]. It uses a byte-code injection mechanism to implement “instrumentation” [38], which means that additional information is added to the program code at the beginning and end of a method definition to capture accurate information about the arising call dynamics. This tool collects the following information about each method call:

- caller and callee of the method call
- the fully qualifying reference for each (package, class and method names)
- the time consumed by the method call
- call path (i.e., caller sequence) for this method call

We store the runtime information in a Neo4j graph database and interactively visualize it by means of Unity3D (a widely-spread engine for developing interactive simulations and game) after program execution. We implemented this sequential workflow to increase the flexibility of our prototypic design. Once the design will have outgrown the prototype status, it will be be feasible to run analyses at real-time by switching to the paradigm of first-in/first-out data stream processing by eliminating the intermediate csv log step. Figure 3 gives an overview of data collection and processing.

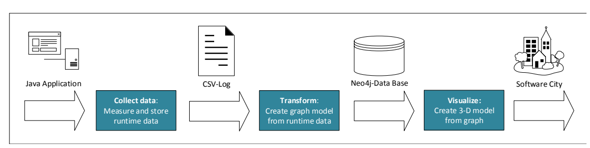

Figure 3: Overview of data collection and processing

4.2 Primary Visual Elements

We decorated the buildings with simple shapes to quickly discern their types (see Figure 4): Buildings with spheres on the roof represent packages, whereas an additional white box at the highest floor represents a class. All other buildings depict methods—boxes monochromatically shaded based on their owning artifacts.

Streets between buildings represent call–callee relations, whereas colored, moving vehicles indicate the direction of these relationships. Sequences of method calls are represented as streams of vehicles of a particular color, which fulfills an important aspect of (R2). In analogy to building dimensions, streets assume one of three dimensions depending on the relative numbers of calls they represent, again relating to (R3). In addition, streets with high traffic volumes pull their attached buildings closer together, which results in clearly visible clusters.

The user can focus on a specific element by selecting it (e.g., a package). This will depict only on incoming/outgoing traffic of this element. Thus, it is possible to investigate potential bottlenecks in detail.

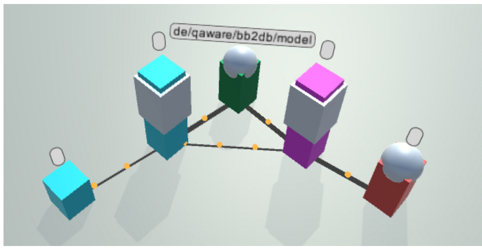

Figure 4: Buildings and streets in our software city visualization.

4.3 Basic Layout

We have decided to implement a force-directed layout, as it is able to reflect the hierarchical relationships among structural code artifacts (R1), and as it is the only layout approach that we could identify that provides a clear visualization of dynamic relations such as call–callee relationships (R2). Figure 5 shows the resulting layout.

We have examined several layout alternatives: Next to the aforementioned tree-map and ThreadCity's EvoStreet layouts, which map the code hierarchy to recursively generated geometries [36], several other layout methods for software visualization have been proposed. For instance, circular layouts or lattice arrangements efficiently use space and facilitate the ordering of classes based on metrics such as lines-of-code or age [2, 24].

Force-directed layouts treat graphs as physical systems with repelling and attracting forces between their elements. Typically, nodes repel each other (like charged particles), while edges draw nodes together (like springs). This results in organic, non-overlapping, aesthetically attractive layouts that are particularly suited for visualizing networks [4, 27].

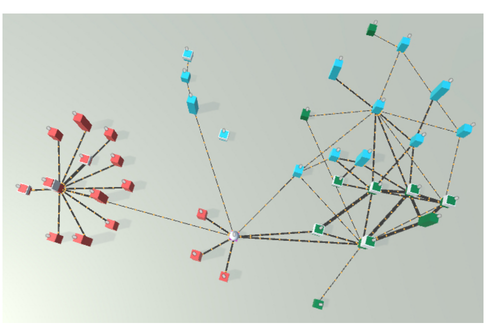

Figure 5: The city layout as seen from above focuses on actual run-time dependencies.

Our layout approach represents structural artifacts such as packages, classes and methods as easily distinguishable buildings. In terms of metrics visualization, (R3), users can switch between sum, maximum or average processing time spent on the artifact (see Figure 6).

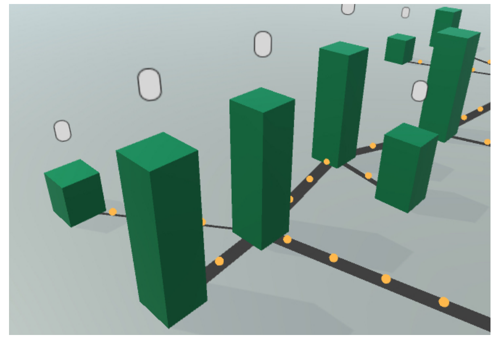

Figure 6: The elements' relative dimensions visualize performance values of the respective artifacts.

4.4 Hierarchies and Artifact Ownership

As data are collected at a fine level of granularity (i.e., methods), the visualization allows focusing on arbitrary levels of the code hierarchy (see Figure 7). Hierarchical relationships between structural artifacts are conveyed by allowing abstraction (i.e., bottom-up substitution of lower-level elements) as well as drill-down (i.e., breaking down higher-level elements). For instance, a "class building" may be replaced by its hosted methods' buildings. The other way round, if a lower-level artifact is substituted, the respective building as well as all associated streets are removed. After introducing or removing one or more artifacts, the force-directed layouting algorithm re-arranges the layout accordingly. In order to communicate the relationship between the previous and the new layout, the transition between the two is animated. Our approach implies that only one level in the code hierarchy is visualized for each structural artifact at the same time; that is, either children or parents in the code hierarchy are visualized, never both.

We provide three different cues to ensure that the different levels of ownership of the artifacts are transparent to the user: (1) Text fields reveal links to the parents. (2) Siblings in the hierarchy are highlighted when an artifact is selected. (3) The colors assigned to different packages are chosen from a palette of contrasting hues and they are inherited by their enclosed classes and methods. For as long as building types are distinguishable this approach conveys the static structural hierarchy (R1). The interactive substitution hierarchy also allows the user to drill down from the highest level of structural artifacts to those most interesting to him, aligned with (R4). The tandem of filtering out irrelevant data and zooming in on important details in combination with the free movement in 3D space effectively supports well-directed user-driven searches (R5). This functionality is furthered by aggregating performance measures at nodes of higher levels. For instance, a class building's height represents, by default, the total amount of processing time used up by all its methods. Alternative performance measures can be easily added at the code-level of our software and, if desired, mapped to the building heights or other visual cues.

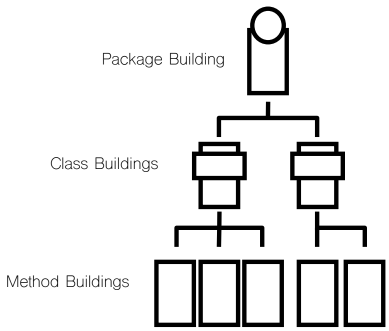

Figure 7: Schematic substitution of buildings.

4.5 Interactions with the Software City

Next to direct interactions with the buildings and streets, we offer a heads-up display (HUD) interface to the user, shown in Figure 8. It overlays the view and provides additional information and configuration options. The legend (element A) helps interpreting the basic visual elements and relates them to individual colors. Additional information on selected elements, such as name, type and location in the code hierarchy as well as various associated performance measures, is provided through the inspector (element B). The inspector also offers buttons to break down high-level artifacts in the code hierarchy. When the user hovers over a building with the mouse pointer, text field (C) shows its name and type, providing quick high-level information for orientation and browsing. Button (D.1) opens the configuration menu (D.2) that provides configuration options of the view in accordance with the user's goals. Here, one can change the mapping of building dimensions to represent absolute or relative execution times. In addition, one can exclusively choose packages, classes or methods to be visualized. These functionalities emerged from expert interviews as explained in the next section. (E) provides a text field for setting filters to prune the visualization in accordance with one's search goals that allows to filter artifacts based on substrings in their names. This functionality, too, is an outcome from the expert interview we conducted.

Some elements of the HUD provide feedback and further options, when the user directly interacts with individual artifacts in the visualized scene. When the mouse pointer hovers over an artifact, it highlights to signal its readiness for interaction. (C) provides high-level information. When clicked, the artifact is selected, highlighted with even brighter color, it is mounted on a yellow circle, its siblings are mounted on white circles to understand the hierarchical context (Figure 9) and further information is shown in the inspector (B).

5 EVALUATION

We conducted six expert interviews to evaluate the expressiveness and effectiveness of our approach in the context of the use cases that motivated our work as outlined in Section 2. Our main goal was to confirm that the approach and prototype are relevant and useful, to improve them where necessary and to provide guidance for future enhancements. To this end, we conducted a simple formative evaluation (i.e., an evaluation during a project's implementation to improve its design and performance) with an early version of our prototype. The final prototype presented in this paper already contains improvements that were prioritized in the evaluation. In particular, we formulated the following four evaluation goals (EG):

- EG1 *Relevance*: Are our goals and the use cases U1–U3 relevant to daily work within the company?
- EG2 *Expressiveness*: A visualization is expressive if it encodes all the information intended and no other information [6, 26]. In our case, we phrase this as: Are the visualization metaphors understandable, and do they encode all and only the required information?
- EG3 *Effectiveness*: A visualization is effective if it presents all information clearly and allows quick understanding [6, 26]. In our case, we phrase this as: Is the visualization able to support U1–U3 in a cost-effective manner; in particular, does it help to quickly identify bottlenecks?
- EG4 *Future improvements*: One main goal of the interviews was to suggest and prioritize potential enhancements to the visualization approach.

Data collection and analysis procedure: We chose to conduct open, semi-structured interviews. This leaves a lot of space for detailed feedback also about future enhancements. We asked the interviewees to verbalize their thoughts (i.e., to follow a think-aloud protocol). We used notes taken by two recorders of the interview for analysis. One person coded the transcripts, and one person verified the coding. In addition, we used closed questions on a 5-point Likert scale to back up the qualitative statements with quantitative ratings and to quantify agreement for EG1–EG3.

Threats to validity: The main threat to validity is external: The work place of the interviewed experts (QAware GmbH) strongly focuses on software quality, which might have biased the evaluation results towards this particular domain. Regarding internal validity, coding and its verification were done by two researchers, and both were involved in prototype development. We believe these threats to validity to be acceptable, since the main goal of this evaluation was formative in nature: to confirm whether the prototype proceeded in a relevant direction for software analysts in general, for the company's purposes in particular, and to provide guidance for future development. However, although the prototype and the evaluation were targeted towards a specific company, we believe that the use cases themselves are relevant for other companies, too.

5.1 Subjects and Context

The prototype development and evaluation were done within the context of QAware GmbH, and all interviewees are employees of QAware. QAware develops software applications for its customers, whereby all projects have to fulfill a quality contract—an integrated quality measurement and assessment based on SonarQube. Apart from software development and maintenance projects, one important business field is analysis and renovation of legacy systems. Here, dynamic analysis of runtime behavior is a crucial task and QAware's experts have a high degree of experience and expertise.

Altogether, we conducted six interviews—representing about 10% of QAware's employees. Four interviewees were experts in

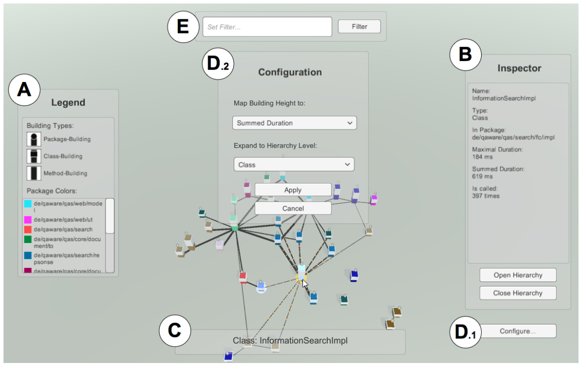

Figure 8: Heads-up display for our software city.

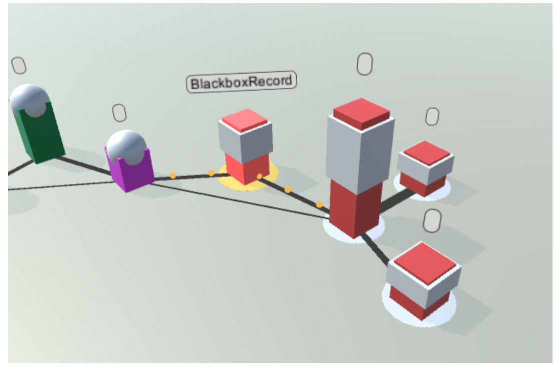

Figure 9: A selected building is shown on a yellow circle, its code hierarchy siblings on white ones.

the analysis of runtime behavior of software, two of them chief technologists with more than 15 years of experience; the other two of them lead technologists with 8–10 years of development and runtime analysis experience. Two additional interviewees were software developers—with 2–4 years of development experience, but with less experience in runtime analysis. The analysts’ feedback would ensure that all the data is provided by our visualization that is relevant for inspecting runtime processes, whereas the developers can test the support received during design and implementation.

5.2 Interview Execution

The six interviews were conducted with a single interviewee each. We targeted a one hour time frame. We gave an introduction (5 min) to inform about the prototype’s goals and the interview procedure. This was followed by discussing the use cases (15 min), using the prototype (15 min), and discussing it (25 min).

Regarding **EG1** (Relevance), we asked to rate the use cases’ importance as well as that of improving current workflows on a 5-point Likert scale. We additionally asked which tools the interviewee currently used for analyzing the runtime behavior, and how they tackled the use cases. To address **EG2** (expressiveness), we asked which information they considered crucial for the use cases. We then presented our prototype and asked the interviewees to verbalize how they interpreted the visualization’s elements and interactions. We let them explore the prototype and provided explanations when appropriate. This allowed exposing obstacles to understandability. Further, we addressed **EG3** (effectiveness) by asking what hindered quick understanding; particularly, whether the force-directed graph layout supported analyzing runtime behavior. Lastly, we addressed **EG4** explicitly (future improvements) by asking about missing information, and by inviting ideas for possible extensions of the approach.

5.3 Results for EG1: Relevance of the Use Cases

The interviewees characterized the *current situation* to address the use cases by the need to use several expert tools. One expert mentioned a tool that visualizes graphs of all possible call–callee relationships (Structure101). Generally, these relationships are tracked by navigating through the program’s source code. UML diagrams that accompany code bases also show call–callee relationships but are not a reliable source as the actual implementation might deviate, as one interviewee explained. Runtime performance of applications are currently evaluated by means of profiler tools, log files, application metric frameworks such as JMX (e.g., active threads, pools, stack traces, heap sizes), monitoring tools provided by the operating system (e.g., dtrace), or by measuring execution times. These are expert tools that are difficult to apply by even senior developers. One expert described the current paradigm in performance analysis as: Design first, trouble-shoot when facing problems. All interviewees agreed that the current situation should be improved on.

Regarding *relevance of the use cases*, all six interviewees agreed that they were relevant to their daily work. Further, five out of six interviewees thought that improvements to the current implementation of use cases are important or very important. One of them explained that there were professional tools such as profilers, yet it was important to make runtime analysis more accessible (U1, U2). Another one pointed out that addressing the use cases was very important as current visualizations did not convey relationships in an intuitive manner. This impression was confirmed by a third expert who also described the current view on software analysis tools as “very deep-down”. As a result, developers need to individually acquire an overview, often unassisted by tools. This perspective was shared by a fourth interviewee who foresaw great utility of the visualization approach for unexperienced developers. A fifth interviewee emphasized the great importance of U2 and U3 (supporting the performance analysis by non-experts and experts), but considered U1 (understanding call–callee relationships) only a weak use case. A sixth interviewee, one of the two analysis experts, said that he sees the value in the prototype but that he has not really needed it, so far. That is, the prototype’s utility for expert analysts may be improved.

To summarize, there are tools that address use cases U1–U3. However, these are expert tools that are difficult to apply and that provide a very detailed view on the software without giving an overview. As they are known only to analysis experts, visualizations of process graphs seem to not have been widely adopted yet. In addition, no expert knew a tool that combined the visualization of performance bottlenecks and call–callee relationships. The experts considered new solutions to the use cases as necessary. However, they set their priorities on different use cases, which may stem from their different work specializations. Overall, the interviewees confirmed the relevance of our use cases, and thereby the relevance of our approach.

5.4 Results for EG2: Expressiveness

To evaluate expressiveness (EG2), we wanted to determine whether the visualization is understandable and useful, and whether it provides sufficient information for the use cases.

Regarding *information need*, the interviewees voiced the following information need for the use cases: hierarchy levels of the software; call–callee relationships; and local resource demand, including CPU cycles, network input/output and memory load. Further, one should be able to track individual requests or calls being propagated through the software. Hotspots, e.g., methods or classes that prolong execution times or which are frequently called, should be discernible. At any time, the global context should be maintained.

The interviewees confirmed that the information provided by the visualization addresses all their information needed and at the same time does not provide unnecessary information with few exceptions: The prototype provides CPU time (not cycles) but no information on other local resources.

Regarding *understandability*, five out of six interviewees fully agreed that the city metaphor is highly understandable and useful, and one abstained from rating. In more detail, two said it provided a good perspective to visualize dynamics in 3D space, two praised the visualization of dynamic method calls. One expert stressed that the metaphor offered a solid foundation to talk about and discuss software systems. Besides, he remarked, it provided more fun than studying software architecture diagrams and IT concepts. Five out of six interviewees confirmed the *prototype's usefulness* in addressing the three use cases. One expert did not commit to any ratings. Four interviewees stated that it was helpful for understanding runtime behavior and identifying performance bottlenecks (U1 and U2). Three interviewees said it aided analysis experts (U3). One analysis expert suggested that the visualization should be embedded in existing profiler tools, and be made available on demand. Two analysis experts asked for the integration of more performance metrics. Four interviewees explicitly stated that the drill-down approach to traverse several layers of the code hierarchy was useful.

To conclude the survey regarding expressiveness, the majority of experts considered the prototype a useful solution to the use cases. We notice, however, that only one analysis and two developer experts found the prototype useful for use case (U3). This limitation might be overcome by integrating the visualization into established tools such as profilers and by visualizing more metrics such as performance measures.

5.5 Results for EG3: Effectiveness

In terms of effectiveness (EG3), we wanted to determine how well the prototype supported cost-effective support of use cases, in particular how well it allowed quick identification of bottlenecks.

Regarding effectiveness of the prototype in general, all interviewees confirmed that the approach supports the use cases and helps to quickly identify bottlenecks. One expert explained that the prototype helped in focusing on a specific aspect without losing its context. Another one noted that the visualization was intuitive and that its interpretations came naturally. Two more experts pointed out that the prototype provided the most important information at a glance. Four experts emphasized that the drill-down concept was useful. One pointed out that the prototype made call cycles visible, even violations of the architectural specification could be seen. Similarly, another expert praised that communication partners were clustered together. Another advantage mentioned by one expert was the ability to present all communication paths and not to lose any information.

Regarding the force-directed layout, five out of six interviewees stated that it was useful, since it is driven by the connections among the buildings. The sixth interviewee did not comment on the layout. Four experts described the layout as clear. One of them emphasized that it was interesting to him to search within the given scene, relying on the given layout. One interviewee said that it was challenging yet learnable not to lose the global static context when drilling down the hierarchy. In contrast, three interviewees explicitly said that they did not miss hierarchical structural information.

In summary, the interviewees confirmed that our approach supports the user by providing a clear and intuitive 3-D visualization of software structures and runtime dynamics. Drilling down the code hierarchy helps to quickly approach places of interest; HUD elements such as legend and inspector have been intensively used by the interviewees. The proposed layout supports use cases U1–U3.

5.6 Results for EG4: Future Improvements

When asked about future improvements, the interviewees described additional use cases for our prototype. For instance, two experts explained that the visualization could be utilized for evaluating software architectures. For example, the prototype can easily be extended to visualize the test coverage of software or to detect orphaned code. Other potential use cases include to trace system or user behavior such as navigation paths of user interfaces. One expert even envisioned the visualization to become the basis of a comprehensive refactoring tool. The expert feedback on potential future extensions makes clear that our approach does not only support the originally expressed problem statement but that it could be tailored to various additional use cases.

During the refinement process of our prototype, we gave those ideas higher priorities that were mentioned by numerous experts during the interviews.

5.6.1 Expressiveness improvement. Table 1 summarizes the information the interviewees felt they still needed in order to fully address use cases U1 to U3. The call for the visualization of additional/higher levels of the code hierarchy sticks out. This is feasible but requires extending the data collection pipeline. As pointed out in Section 4, we have already introduced small extensions the experts asked for, such as the adaptability of the visualized attributes, or additional building dimensions. In addition, we already implemented several aspects: (1) Include forces along edges that are proportional to the propagated amount of data. (2) Highlight outliers in terms of resource usage by mapping maximal execution times to building heights. (3) Provide the frequency of method calls in the inspector.

# mentions	Information lacking for Expressiveness
5	Visualize additional hierarchy layers, e.g. components or distributed systems.
3	Provide configuration of the city visualization, e.g., building heights encode different metrics.
2	- Provide a time window of observed events. - Visualize static grouping of buildings. - Represent differences in software versions. - Display memory consumption. - Display the software's communication with the environment. - Visualize an application during runtime.

Table 1: Interview feedback on the missing information to support the captured use cases.

5.6.2 Effectiveness improvement. Table 2 lists those ideas that were expressed by several experts. Four experts asked for information about the artifacts (e.g.,their names), to be projected into the 3D scene. Again four of them said that filters, for instance by execution times, would help reduce the number of visible artifacts and thereby support the user in identifying hotspots. Three interviewees stated that visualizing all artifacts of a specific hierarchical level at once (e.g., all classes) would be beneficial. The potential use of the streets' color attribute was mentioned three times as well. There were several additional suggestions such as the visualization of the point of exit of an application, but they were rather specific and only mentioned by one expert each.

# mentions	Improvements for Effectiveness
4	- Show labels for artifacts within the 3D scene. - Provide filter for artifacts, e.g., filter those of a specific type (e.g., class), those that exceed a given execution time or number of execution.
3	- Visualize all buildings of one hierarchical layer, e.g. all classes. - Utilize street colors.

Table 2: Interview feedback on the potential improvements to increase the effectiveness of our prototype.

6 SUMMARY & FUTURE WORK

Based on three use cases we motivated the problem statement of our work and we inferred requirements for a visualization prototype we designed and implemented. An in-depth analysis of existing work revealed that some requirements were covered quite well but no existing approach addressed R2 (readable layout for dynamic structures such as call graphs). We extended core ideas of two approaches (ThreadCity and ExplorViz) to build a novel approach that mitigates various obstacles to their practical application.

The key elements of our visualization approach comprise (1) visualizing call events and the resulting call–callee graph, (2) utilizing fine-grained levels of structural artifacts (namely: method instead of package or class), (3) enhancing the visualization with additional performance measures, and (4) adopting a dynamic layout and means of filtering data to increase clarity.

Based on the preliminary evaluation using expert interviews and our own insights from developing the prototype, we believe the following steps would promote the utilization of software cities for practical development and analysis work the most: Our prototype relies on an aggregated form of the runtime model (Section 4.1). In order to achieve greater practical value and empower developers, a refined approach should also record methods calling themselves, and the data pipeline should be re-organized to support real-time analysis. The hierarchy levels that we currently visualize should be extended to incorporate distributed system components. This would ensure scalability to systems where issues emerge from networking technologies, problems of synchronicity, or from mutual access to shared data. Moreover, we are currently investigating whether moving this approach to virtual reality adds value to the visualization and usability.

In addition to functional extensions, we feel that especially the following two questions should be addressed in the scope of future research efforts, since they were raised and emphasized numerous times: (1) "How can the static structure of software be better visualized?"—our expert interviews emphasized that maintaining static grouping might be challenging when drilling down the code hierarchy. At the same time, the independence of the visualization from the hierarchy allowed us to adapt the view to the user's search targets. Potentially, this discrepancy could be addressed by secondary visualizations blended in on demand. (2) "How can the existing prototype be effectively integrated in existing workflows?"—as the experts mentioned during the interviews, one could embed it into existing profiler tools. However, one should also consider the other way round, along with visual programming, and envision the potential benefits to programming environments and profiler tools being embedded in software visualizations.

Following the agile principle of avoiding broken windows, it is important to detect potential quality problems as early as possible. Consequently, a system's quality needs to be analyzed regularly, and it is crucial to make quality assessment part of a common development routine. Regarding static code metrics, it is common to integrate a quality analysis within the standard build chain (e.g., using SonarQube): Every build triggers quality measurement. Regarding the visualization prototype presented in this paper, we envision that it can be employed at regular intervals such as sprint or release gateways, where it is usual to do performance tests. Additionally, data collection for the visualization can be automated and integrated into nightly builds with automated system tests.

ACKNOWLEDGMENTS

We would like to thank all members of QAware GmbH who participated in the requirements elicitation and the empirical evaluation. Parts of this work have been supported by the German Ministry of Education and Research under grant no. 01IS15008D.

REFERENCES

[1] Alain Abran, James W. Moore, Robert Dupuis, Pierre Bourque, and Leonard L. Tripp. 2004. *Guide to the Software Engineering Body of Knowledge (SWEBOK)*. Angela Burgess. 204 pages.

[2] Sazzadul Alam and Philippe Dugerdil. 2007. Evospaces visualization tool: Exploring software architecture in 3d. In *14th Working Conference on Reverse Engineering (WCRE 2007)*. IEEE, 269–270.

[3] Gergő Balogh, Attila Szabolics, and Arpád Beszédes. 2015. CodeMetropolis: Eclipse over the city of source code. In *Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on*. IEEE, 271–276.

[4] Michael Balzer and Oliver Deussen. 2005. Exploring relations within software systems using treemap enhanced hierarchical graphs. In *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*. IEEE, 89–94.

[5] G Campbell and Patroklos P Papapetrou. 2013. *SonarQube in Action*. Manning Publications Co.

[6] Stuart Card. 2012. Information visualization. In *Human computer interaction handbook: Fundamentals, evolving technologies, and emerging applications*, Julie A Jacko (Ed.). CRC press, 515–549.

[7] Pierre Caserta, Olivier Zendra, and Damien Bodénes. 2011. 3D hierarchical edge bundles to visualize relations in a software city metaphor. In *Visualizing Software for Understanding and Analysis (VISSOFT), 2011 6th IEEE International Workshop on*. IEEE, 1–8.

[8] Stuart M Charters, Claire Knight, Nigel Thomas, and Malcolm Munro. 2002. Visualisation for informed decision making; from code to components. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*. ACM, 765–772.

[9] Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John Vlissides, and Jeaha Yang. 2002. Visualizing the execution of Java programs. In *Software Visualization*. Springer, 151–162.

[10] Andreas Dieberger and Andrew U Frank. 1998. A city metaphor to support navigation in complex information spaces. *Journal of Visual Languages & Computing* 9, 6 (1998), 597–622.

[11] Stephan Diehl. 2007. *Software visualization: visualizing the structure, behaviour, and evolution of software* (1 ed.). Springer Science & Business Media. 187 pages.

[12] C Russo Dos Santos, Pascal Gros, Pierre Abel, Didier Loisel, Nicolas Trichaud, and Jean-Pierre Paris. 2000. Metaphor-aware 3d navigation. In *Information Visualization, 2000. InfoVis 2000. IEEE Symposium on*. IEEE, 155–165.

[13] Philippe Dugerdil and Sazzadul Alam. 2008. Execution trace visualization in a 3D space. In *Information Technology: New Generations, 2008. ITNG 2008. Fifth International Conference on*. IEEE, 38–43.

[14] Simon Faber. 2016. *3D Visualisierung des Laufzeitverhaltens von Software*. Master Thesis. University of Augsburg.

[15] Florian Fittkau, Santje Finke, Wilhelm Hasselbring, and Jan Waller. 2015. Comparing Trace Visualizations for Program Comprehension through Controlled Experiments. *2015 IEEE 23rd International Conference on Program Comprehension* (2015), 266–276.

[16] Florian Fittkau, Jan Waller, Christian Wulf, and Wilhelm Hasselbring. 2013. Live trace visualization for comprehending large software landscapes: The ExplorViz approach. In *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on*. IEEE, 1–4.

[17] Sebastian Hahn, Matthias Trapp, Nikolai Wuttke, and Jürgen Döllner. 2015. Thread City: Combined Visualization of Structure and Activity for the Exploration of Multi-threaded Software Systems. In *2015 19th International Conference on Information Visualisation*. IEEE, 101–106.

[18] Andy Hunt and Dave Thomas. 2002. Zero-tolerance construction [software development]. *IEEE Software* 19, 5 (2002), 100–102.

[19] Brian Johnson and Ben Shneiderman. 1991. Tree-maps: A space-filling approach to the visualization of hierarchical information structures. In *Visualization, 1991. Visualization'91, Proceedings., IEEE Conference on*. IEEE, 284–291.

[20] Barbara Kitchenham and Stuart Charters. 2007. *Guidelines for performing Systematic Literature Reviews in Software Engineering*. Technical Report. Technical report, EBSE Technical Report EBSE-2007-01. 65 pages.

[21] Claire Knight and Malcolm Munro. 1999. Comprehension with [in] virtual environment visualisations. In *Program Comprehension, 1999. Proceedings. Seventh International Workshop on*. IEEE, 4–11.

[22] Kenichi Kobayashi, Manabu Kamimura, Keisuke Yano, Koki Kato, and Akihiko Matsuo. 2013. SArF map: Visualizing software architecture from feature and layer viewpoints. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 43–52.

[23] Christian FJ Lange, Martijn AM Wijns, and Michel RV Chaudron. 2007. A visualization framework for task-oriented modeling using UML. In *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*. IEEE, 289a–289a.

[24] Guillaume Langelier, Houari Sahraoui, and Pierre Poulin. 2005. Visualization-based analysis of quality for large-scale software systems. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. ACM, 214–223.

[25] Jock Mackinlay. 1986. Automating the design of graphical presentations of relational information. *Acm Transactions On Graphics (Tog)* 5, 2 (1986), 110–141.

[26] Jock Mackinlay. 1986. Automating the Design of Graphical Presentations of Relational Information. *ACM Trans. Graph.* 5, 2 (April 1986), 110–141. https://doi.org/10.1145/22949.22950

[27] Andreas Noack and Claus Lewerentz. 2005. A space of layout styles for hierarchical graph models of software systems. In *Proceedings of the 2005 ACM symposium on Software visualization*. ACM, 155–164.

[28] Christoph Noetzel. 2013. *Messung des Laufzeitverhaltens von Software-Systemen mit Hilfe einer Software-Blackbox*. Masterthesis. Hochschule für angewandte Wissenschaften München.

[29] Michael J Pacione, Marc Roper, and Murray Wood. 2004. A novel software visualisation model to support software comprehension. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*. IEEE, 70–79.

[30] Thomas Panas, Thomas Epperly, Daniel Quinlan, Andreas Saebjornsen, and Richard Vuduc. 2007. Communicating software architecture using a unified single-view visualization. In *12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*. IEEE, 217–228.

[31] Jonathan Sillito, Gail C Murphy, and Kris De Volder. 2006. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 23–34.

[32] Artur Sosnówka. 2013. Test City metaphor as support for visual testcase analysis within integration test domain. In *Computer Science and Information Systems (FedCSIS), 2013 Federated Conference on*. IEEE, 1365–1370.

[33] Frank Steinbrückner. 2010. Coherent software cities. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*. IEEE, 1–2.

[34] Jan Waller, Christian Wulf, Florian Fittkau, Philipp Döhring, and Wilhelm Hasselbring. 2013. Synchrovis: 3d visualization of monitoring traces in the city metaphor for analyzing concurrency. In *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on*. IEEE, 1–4.

[35] Richard Wettel. 2009. Visual exploration of large-scale evolving software. In *Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*. IEEE, 391–394.

[36] Richard Wettel and Michele Lanza. 2007. Visualizing software systems as cities. *Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis* (2007), 92–99.

[37] Richard Wettel and Michele Lanza. 2008. Code City. *Proceedings of WASDeTT 2008 (1st International Workshop on Advanced Software Development Tools and Techniques)* (2008), 1–13.

[38] John Whaley. 2000. A portable sampling-based profiler for Java virtual machines. In *Proceedings of the ACM 2000 conference on Java Grande*. ACM, 78–87.