Interactive Visualization for OSGi-based Projects

Niklas Rentz, Reinhard von Hanxleden
Department of Computer Science
Kiel University, Kiel, Germany
{nre, rvh}@informatik.uni-kiel.de

Christian Dams
Scheidt & Bachmann System Technik GmbH
Melsdorf, Germany
dams.christian@scheidt-bachmann-st.de

***Abstract*—Big software projects often use architectural frameworks for a consistent structure. OSGi is such a framework to create modular Java applications. The architecture of individual projects, however, is often hidden in configuration files.**

We propose to visualize projects in a modular framework such as OSGi with an approach to allow users to comprehend the connections within a system. We assist this comprehension using filtering and automatically generated, interactive views of the project. We extend the notion of interactive views with a concept to reproduce configured views for arbitrary system revisions to enhance up-to-date documentation.

We have implemented this proposal in the publicly available KIELER project, and have validated it with a large software project in the railway domain.

I. Introduction

Development for software projects requires knowledge about these projects to efficiently find problems and to implement new features. Having such knowledge, in turn, requires to read documentation, time, and, before that, to create that documentation in the first place. Early studies such as by Lientz et al. [1] have shown that the maintenance cost of software ranges as high as 80 percent for some projects, showing that good and up-to-date documentation is important. Much time is needed just to understand the code and documentation compared to handling problems or implementing new features. The need for documentation is quite natural when projects age, and the knowledge about the code disappears because project personnel is changing with time, as noted by Ball and Eick [2].

Visual code comprehension tools can help to solve these problems when they contain visuals generated automatically from the underlying project. This frees users from reading through specific implementation details just to find what can be extracted from the project files directly. For the best usability for architects and the developers, the views should be close to the development environment, or even with interactivity with the source model to allow for roundtrip visualizations, as suggested by Charters et al. [3].

The work presented here is motivated by needs identified for the development of complex software systems for the railway domain. The development and maintenance productivity and quality for such systems are the leading goals of the project. We therefore aim to provide solutions for more visible code for developers and architects, where existing solutions did not answer their specific questions yet.

This work has been supported by the project *Visible Code*, a cooperation between Kiel University and Scheidt & Bachmann System Technik GmbH.

A. Related Work

Seider et al. [4] already proposed automatically generated views for OSGi[1]-based applications. Their views show different metrics of bundles, packages, services, as well as their dependencies. They present a browser-based viewer and a Virtual Reality (VR) environment. As their views focus on graphical aspects to be further inspected via interaction in these tools, the usability of their tools as static documentation is constrained. Furthermore, they mention that "the benefit of the service graph is limited. . . . [It] lacks in supporting the comprehension of their dependencies." We here propose solutions to these problems, with a focus on persisting reproducible views.

The work by Seider et al. was also built upon further, mainly with VR tools [5] and using speech to interact with those [6]. While that is a good way to present the architecture, it is not wide spread to have the hardware available for VR or the knowledge to ask specific questions about the architecture. Therefore we present a tool accompanying the development environment and not using VR to be more accessible.

Petre [7] investigated what expert programmers want from visualizations. Some of the key points they mention to be critical are *insight*, *selectivity*, and *domain knowledge*. Programmers want *insight* to be able to seek information about otherwise obscure data and to be able to identify the key information about a system. The *selectivity* should provide a useful focus on parts of the system, and the *domain knowledge* states that the tool needs to be specific enough to provide useful information about the system at hand. With the visuals we present, we follow these recommendations and provide a tool to give good architectural insight.

Gallagher et al. [8] present seven key areas to improve software architecture visualization practice, such as *static representation* and *task support*. We use these areas to validate our work and provide more details on that in Section IV.

B. Contribution and Outline

- We propose the use of intermediate models to browse projects and to allow reproducible views for documentation purposes on the example of OSGi in Section II.
- We contribute a new interactive visualization for projects using the OSGi technology, including a dependency hierarchy view for bundles and packages, as well as a view for the cooperation of services, see Section III.

[1] OSGi™ is a trademark of the OSGi Alliance in the US and other countries.

978-1-7281-9914-6/20/$31.00 ©2020 IEEE
DOI 10.1109/VISSOFT51673.2020.00013

- We implemented and validated the proposed visualization within the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) project harnessing the *modeling pragmatics* approach by Fuhrmann and von Hanxleden [9] to allow interactivity with automatically laid out diagrams, see Section IV.

Section V concludes the paper and summarizes future improvements and ideas on the topic.

II. Models for the Visualization

To create an insight into projects using some architectural framework, we first have to look at which parts of their specification suggest visualizable components. We present our proposed visualization concepts for the Java framework OSGi, which covers a variety of architectural ideas. The models, visualizations, and interactions are in principle applicable to other architectural frameworks as well.

A. The OSGi Model

OSGi is a Java framework to allow an easy deployment of big applications and extends the architecture of Java projects with multiple levels of functionality. Of particular interest for the visualization in the OSGi specification [10] are the *module layer*, the *life cycle layer*, and the *service layer*. Modularity in OSGi is handled via *bundles* in the *module layer*. They combine classes and packages of the Java world and structure the dependencies between each other as well as requirements for packages provided by unspecified other bundles. For structuring, these bundles can be put into *bundle categories*. As mentioned by Bosschaert [11], this modularity will be extended by *features* in the next release of the OSGi specification, as implementations of that need further reusable components. Some implementations already use their own solutions, there are for example Apache Karaf Features, Eclipse features, and others. This structure is reflected in our proposed *OSGi model* (OM) we use as a basis for creating views for projects using OSGi, as shown in Figure 1. In addition to this generic structure, the OM keeps everything in a top-level *OsgiProject* object and allows features to be grouped in *products*, akin to Eclipse products, to allow the visualization of product-specific configurations.

Connections between parts as dependencies and structuring into higher-level components are not concepts that are specific to OSGi. Dependencies of some kind are included in almost all programming languages and architectures for them. Furthermore, most of them use some kind of file, module, or package equivalent that define some hierarchical structure, and they define some product equivalent for executing an application, so the concepts described here for the module layer are also applicable to other architectures.

More specific to OSGi again is the *life cycle layer*. It includes possibilities to visualize runtime data of bundles and services. We focus on the static views here and do not include such views in this paper yet.

Finally, the *service layer* can be visualized as one of the core pieces of OSGi. Bundles cooperate via *services* that are declared by *service interfaces* and implemented in *service objects*. In addition to this service communication, using Eclipse e4 allows to use injection to inject service objects into specific classes. As that directly extends the OSGi service concept, we include it in the OM.

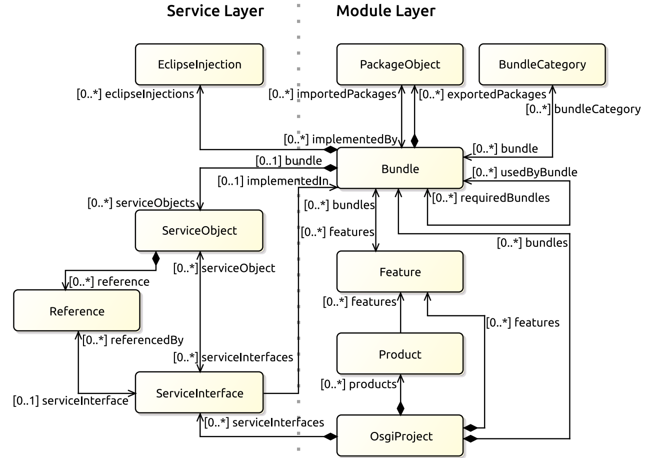

Fig. 1. Class structure of our OSGi model abstraction.

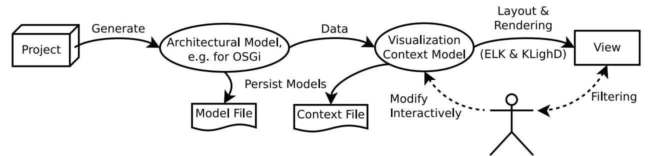

Fig. 2. The architectural model and the visualization context model and their possible interactions.

B. The Visualization Context Model

We propose to have an interactive view that does not limit the user to filter through a global view showing every containment and other dependency between objects. Instead, we let the user choose freely which elements and dependencies to view, with the ability to generate and persist specific views of the project. Therefore we use a second model that contains the current context of the visualization, the *visualization context model* (VCM). An impression of the model interacting with an architectural model such as the OM is shown in Figure 2.

Per default, the VCM describes a generic view on the structure of projects. During interaction, this model can be modified to indicate which parts of the connected model are currently shown. The view then has to ensure to show hints to what is not shown in the current context, which is further explained in Section III.

To use the models and the views as documentation, both intermediate models can be persisted. Persisting the architectural model, for example, can be done as a nightly job or immediately during development of the project that the model is for. Persisting the VCM is important for documentation purposes. The model describes an abstract way on which parts should be shown in the view, such as an overview of all

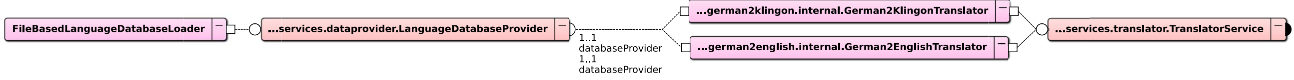

Fig. 3. Hierarchy of service interfaces and service objects providing them. Service objects implementing or requiring interfaces are presented with connections similar to the ball-and-socket notation of the UML specification [12] for a familiar look and quick understanding.

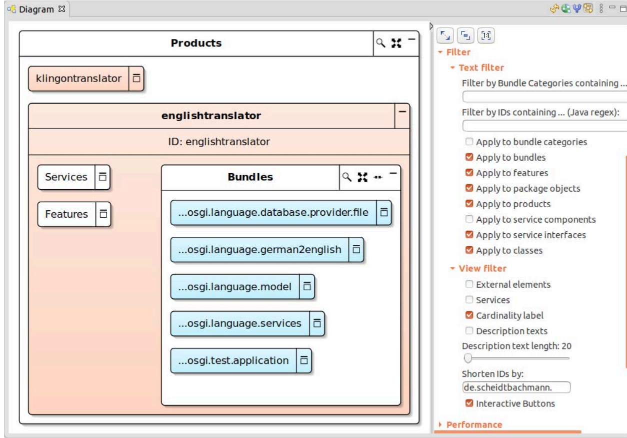

Fig. 4. Overviews of products and bundles within a product using containment. Model generated from example project for this visualization. View presented in the KLighD viewer next to part of the actions and diagram options.

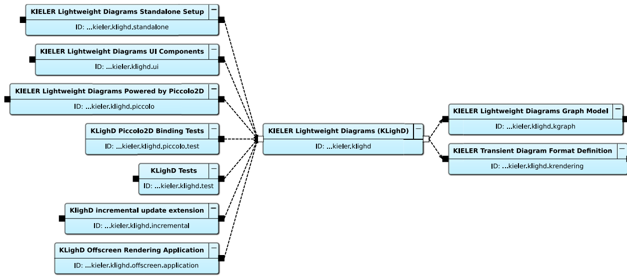

Fig. 5. Dependency hierarchy of the central bundle of KLighD, showing required and requiring bundles. Model generated from the KLighD framework found at https://github.com/kieler/KLighD.

bundles with dependencies to some central UI plugin, or all bundles that use a specific service interface and the service objects providing that. The idea is that no matter what revision of the architectural model is used, the viewer can use the VCM and that architectural model to provide the same view on the wanted revision of the project. For the documentation there is the option to either integrate an interactive viewer for the model or generated static images.

III. Views and Interaction

In consultation with potential users from our industry partner, we identified three view concepts that we want to visualize, namely *containment*, *connection*, and *context*. Note that even though the work presented here is carried out in the OSGi context, we consider these view concepts as rather general. Thus, we also hope that the approach presented here should be of interest beyond OSGi.

The *containment* can be handled in UML style as used in the class diagram in Figure 1, but for specific instances this can get confusing quickly if a single element contains more than a few other elements. Therefore we propose to visualize containment by physical containment of nested element representations, such as shown in Figure 4. Dependencies by elements on the same hierarchical level can be visualized via *connections*, similar to the connections in Figure 1. When viewing dependency hierarchies, we provide a comprehensible model of which elements are on the top or the bottom of the hierarchy as shown in Figures 3 and 5. The advantages of the used layout approach are further explained in Section IV. Finally, we visualize the *context* by various filtering and interaction techniques that allow the user to keep the view at a manageable level, as described below.

The initial view is the same for any project and allows the user to navigate to the context they want to investigate. The navigation allows for three main types of views:

- product, feature, bundle, or service overviews that show those elements and possible connections within some sub-context via containment (see Figure 4),
- a bundle and package dependency hierarchy that uses connections to show the global hierarchy or the hierarchy within some sub-context (see Figure 5), and
- a service dependency hierarchy that uses connections similar to the bundle dependencies (see Figure 3).

The buttons and diagram elements in the figures also show some of the possible interactions. We provide to

- show more or less details for all elements to see the element names, IDs, and descriptive texts or only the ID,
- highlight direct connections of focused elements,
- connect the required or requiring bundles and packages for bundles via clicking the ports to their side. Black ports indicate that not all connections are shown, while white ports indicate complete connections for that element. This is visible in Figure 5 where only all connections for the central bundle are shown. Hovering over the ports hints the user how many elements will be connected by clicking them via a tool tip. The overview also provides the functionality to show all connections at once. Furthermore, a user can
- connect references and implementations of services similarly to bundles. This view can also be configured to show all service objects contained in representations of the bundles that define them. The tool also allows to
- connect representations for classes that inject service interfaces via Eclipse injection, to

- filter shown elements by their IDs or names, and to
- undo and redo past interactions.

With these interactive features we let the user dive into the domain specific parts of the architecture to gain insight about the whole structure or selected parts of it.

IV. Implementation and Validation

A. Implementation—KIELER, KLighD, and ELK

We implemented the OM analysis and extraction as well as the visualization as part of the open source project KIELER[2]. The visualization is split into two individual parts.

The first part is the project analysis tool to generate the OM from the sources of an OSGi project using Eclipse Plug-in Development Environment (PDE) build infrastructure. This tool can be executed as a command line Java application, integrated into a Maven build as a *mojo* or from within the developer's Eclipse IDE.

The second part is the visualization of that model. We use KIELER Lightweight Diagrams (KLighD) as the visualization framework, which is presented by Schneider et al. [13]. KLighD allows to view and interact with node-link diagrams generated from arbitrary models. We use the framework to generate views from the OM, or, more specifically, from the VCM and use the OM as its data source. The interactions described above change the VCM and therefore update the view, while the OM remains unchanged. KLighD uses automatic layout using sophisticated layout algorithms with the Eclipse Layout Kernel (ELK)[3] and the concept of *modeling pragmatics*, which describes customizable views to aid documentation and navigation, as described by Fuhrmann and von Hanxleden [9]. For the dependency hierarchies shown above we utilize the ELK Layered layout algorithm, which is based on the layered approach by Sugiyama et al. [14], for a natural and consistent reading direction of dependencies and requirements. In our experience this approach has shown to be helpful for understanding connections and hierarchies in graphs. Many other program and system comprehension tools use other layout algorithms, such as a force-based layout algorithm, however, that lacks persisting layout directions and therefore does not render hierarchies as well.

The initial implementation of KLighD is based on a Piccolo2D view that can be integrated into Eclipse as a plug-in. That view allows to browse the model and interact with it in real-time. Changes are integrated into and animated in the view to enable a mental model of the user. Further actions and filters can be applied using a sidebar next to the view. Figure 4 shows this view within Eclipse together with some of the buttons and options for interaction with the diagram.

In addition to the Eclipse view, a web view to be plugged into websites for documentation purposes or a standalone website is current work in progress. The master's thesis of Rentz [15] is a possible basis for that, as it describes the migration of the KLighD framework to web technologies using the Sprotty[4] viewer in the web-based Theia IDE[5]. Currently, the same diagrams can be shown in a web browser, but only within Theia and not as a standalone browser plugin yet.

B. Validation—Visualization Practice and Industrial Users

The results presented in this paper were incrementally proposed to the industry partner and revised according to the needs for the data visualization and the feedback on the current progress. To further validate the presented visualization, we compare its possibilities to the seven key areas for software architecture visualization practice proposed to characterize and improve such visualizations by Gallagher et al. [8].

Their first area is the *static representation*. This area characterizes the accessibility of architectural information from architectural and non-architectural sources. This is supported by our tool, as it was the primary goal to support a view on the static architectural data in the system.

Opposing that, the second area describing a *dynamic representation* of runtime data is not supported. It is part of future work to visualize the service and bundle runtime data of OSGi.

Next, the *view* area asks for multiple (simultaneous) perspectives on the architecture. It is a key part that we allow different views to highlight different parts of the architecture. KLighD allows to open multiple views showing different perspectives of the same model, therefore this is also supported.

The fourth area of *navigation and interaction* comes naturally by using the KLighD framework. Navigation via zooming and panning is built into its core. Furthermore, as we support filters on elements and element types, as well as interaction to modify the view, we consider this area to be supported.

The *task support* area cannot be classified as easily, as the authors describe a wide variety of tasks that users might need. Some are supported, such as the comprehension of the architecture, representing anomalies in dependency hierarchies by utilizing the layout, or showing the evolution of software via using the same VCM for multiple revisions of an OM instance. Some are not supported and also not planned, such as the support to construct software architectures or plan and develop them from the visualization directly. There are other tools specifically designed for this that we recommend to use rather than to implement such a feature as well.

Finally, the *implementation* and *representation quality* areas highlight the ease of use, automatic generation of the views, and accurate information within the views. This is also given for the static representation by extracting the data from the sources directly. Overall, our approach already supports several of the critical areas, while some are part of future work.

Also important but not mentioned by Gallagher et al. is the reactivity of the tool. We measured the time needed for the OM generation and for viewing the entire bundle and package dependency hierarchy of a project with 147 bundles. The initial OM generation is done in 13 seconds and the view for the dependency hierarchy is generated in less than a second,

[2]https://github.com/kieler/osgiviz

[3]https://www.eclipse.org/elk/

[4]https://eclipse.org/sprotty

[5]https://theia-ide.org/

allowing for interactive browsing even with bigger projects and unfiltered views.

We presented this work at the Nordic Coding Symposium in Kiel in December 2019 to about 100 participants. We also conducted a workshop and presented this tool to about 20 programmers and architects from our industry partner. The feedback from these events so far was positive. Especially the textual context filtering and the bundle dependency hierarchy were anonymously reviewed to be very helpful by programmers, the automatic layout also got a special mention. We also asked architects for use cases. So far, they created architectural overviews or detail views manually by some generic UML tool. For OSGi they used UML component diagrams to depict relations between OSGi artifacts. One experiment showed that diagrams created by this automatic visualization revealed results that were likewise usable for the architects to document their system. Our approach also helped to communicate the structure and some degree of data and event-flow to stakeholders, e.g. fresh developers in the team. Overall, the tool seems to be helping the comprehension of projects already. A thorough experiment is part of future work.

V. Conclusion and Future Work

In this paper we presented an approach to visualize the architecture of applications that use a modular architecture, on the example of OSGi. Even for visual representations we follow an approach to not only use graphical elements, but also to use textual elements to represent parts of the system. This combination allows for views that can be viewed interactively for browsing or as graphics accompanying the documentation to make the architecture tangible.

With this visualization, new developers can find their way into the architecture quicker, architects can use it to verify the integrity of architectural specifications, and other users can get a better impression on how the application works.

Some improvements can make the concepts presented in this paper usable for more users, more use cases, and in new environments. Differences in distinct versions in time of the architecture could be shown in the same view to communicate the changes in the system on a proper level, for example to communicate refactoring initiatives or other improvements of the system. An extended use case is to compare a manually set architecture to the actual architecture of the implementation. Differences of architecture visualizations could gain many valuable benefits for the architects to keep implementation in the architectural track.

As to future work, embedding the visualization into web views or other textual documentation should help comprehend that documentation further by exploring it directly. Also, visualizing run-time data will help find problems during the startup of OSGi applications and with resolving service requirements during runtime. This could, for example, be built into the Apache Felix Web Console to allow for additional graphical support during the inspection of running OSGi applications.

When developers use Eclipse to develop the OSGi application and use the tool described here to analyze the architecture, further interaction from the diagram to the source files that were used to generate the model would also be useful. For example, the user could interact with the diagram representation for a bundle in such a way that they are guided to the files defining that bundle, the same for other elements.

The OM generation tool could also be extended to allow to analyze projects using Bndtools[6] for their setup or some other non-Eclipse way of defining OSGi projects.

Finally, an area for future work is to generalize the concepts presented here in a way to visualize arbitrary static architectural connections with a meta model for such architectures, so that the tool presented here forms a first instance of such a generic visualization for big software systems.

References

[1] B. P. Lientz, E. B. Swanson, and G. E. Tompkins, "Characteristics of application software maintenance," *Communications of the ACM*, vol. 21, no. 6, pp. 466–471, Jun. 1978.

[2] T. Ball and S. G. Eick, "Software visualization in the large," *IEEE Computer*, vol. 29, no. 4, pp. 33–43, 1996.

[3] S. M. Charters, N. Thomas, and M. Munro, "The end of the line for software visualisation?" in *Proc. 2nd IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, Amsterdam, The Netherlands, Sep. 2003, pp. 110–112.

[4] D. Seider, T. Marquardt, M. Brüggemann, and A. Schreiber, "Visualizing modules and dependencies of OSGi-based applications," in *2016 IEEE Working Conf. on Software Visualization (VISSOFT)*, Raleigh, NC, USA, 2016, pp. 96–100.

[5] M. Misiak, A. Schreiber, A. Fuhrmann, S. Zur, D. Seider, and L. Nafeie, "IslandViz: A tool for visualizing modular software systems in virtual reality," in *2018 IEEE Working Conf. on Software Visualization (VISSOFT)*, Madrid, Spain, 2018, pp. 112–116.

[6] P. Seipel, A. Stock, S. Santhanam, A. Baranowski, N. Hochgeschwender, and A. Schreiber, "Speak to your software visualization—exploring component-based software architectures in augmented reality with a conversational interface," in *2019 IEEE Working Conf. on Software Visualization (VISSOFT)*, Cleveland, OH, USA, 2019, pp. 78–82.

[7] M. Petre, "Mental imagery, visualisation tools and team work," in *Second Program Visualization Workshop*, M. Ben-Ari, Ed. Aarhus, Denmark: University of Aarhus, Dec. 2002, pp. 2–13.

[8] K. Gallagher, A. Hatch, and M. Munro, "Software architecture visualization: An evaluation framework and its application," *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 260–270, Mar. 2008.

[9] H. Fuhrmann and R. von Hanxleden, "On the pragmatics of model-based design," in *Proceedings of the 15th Monterey Workshop 2008 on the Foundations of Computer Software. Future Trends and Techniques for Development, Revised Selected Papers*, ser. LNCS, vol. 6028. Budapest, HR: Springer, 2010, pp. 116–140.

[10] *OSGi Core Release 7 Specification*, The OSGi Alliance, Apr. 2018, Accessed: Jun. 03, 2020. [Online]. Available: https://docs.osgi.org/download/r7/osgi.core-7.0.0.pdf.

[11] D. Bosschaert, "New OSGi work - features," Jul. 2019, Accessed: Jun. 03, 2020. [Online]. Available: https://blog.osgi.org/2019/07/new-osgi-work-features.html.

[12] Object Management Group, "OMG Unified Modeling Language Specification, Version 2.5.1," Dec. 2017, Accessed: Jun. 03, 2020. [Online]. Available: https://www.omg.org/spec/UML/2.5.1/.

[13] C. Schneider, M. Spönemann, and R. von Hanxleden, "Just model! – Putting automatic synthesis of node-link-diagrams into practice," in *Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '13)*, San Jose, CA, USA, Sep. 2013, pp. 75–82.

[14] K. Sugiyama, S. Tagawa, and M. Toda, "Methods for visual understanding of hierarchical system structures," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 11, no. 2, pp. 109–125, Feb. 1981.

[15] N. Rentz, "Moving transient views from Eclipse to web technologies," Master's thesis, Kiel University, Department of Computer Science, Nov. 2018, https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/nir-mt.pdf.

[6]https://bndtools.org/