# Polyptychon: A Hierarchically-constrained Classified Dependencies Visualization

Donny T. Daniel*, Egon Wuchner*, Konstantin Sokolov*, Michael Stal*, Peter Liggesmeyer†
* Architecture Definition and Management, CT RTC SAD, Siemens AG, Munich, Germany
{donny-thomas.daniel, egon.wuchner, konstantin.sokolov.ext, michael.stal}@siemens.com
†AG Software Engineering: Dependability, TU Kaiserslautern, Kaiserslautern, Germany
liggesmeyer@cs.uni-kl.de

***Abstract*—Architects and developers are often tasked with evaluating or maintaining unfamiliar software systems. Reverse engineering tools help extract relationships between the system parts as they exist instead of as documented. Though node-link diagrams have a straightforward correspondence with the graph-representable data generated, the scale and complexity of real-world data sets prevent efficient comprehension.**

**This paper presents Polyptychon, an interactive node-link visualization designed for incremental exploration of dependency information. Given a hierarchical information space of software artifacts, Polyptychon constrains the visible dependencies to be related to the child nodes of a specified artifact node, called a *view root*. It then classifies these *siblings* as levelized, tangled and independent. It also includes *context* nodes, which are a filtered set of nodes elsewhere in the hierarchy that are related to the siblings. The context nodes are further grouped based on a project-specific partition function.**

**The hierarchical constraints and partition function provide means to control the number of nodes displayed, while the dependency classification allows users to form a qualitative impression of the dependency structure. We demonstrate with examples from the Netty open source project. We conclude with areas of future work, in particular, as a basis of evolutionary dependency analysis.**

## I. Introduction

Architecture evaluations are periodically undertaken to ensure the sustainability of a system [1]. Developers assigned to new projects are required to become productive as soon as possible. A typical use case confronted in these scenarios is to understand how the system is actually organized (as opposed to how it is documented). Numerous dependency relations exist between system parts [2] and there are various methods [3] to extract them from a codebase. Visual representations of this dependency information allow the user to get a better understanding of the state of a system in terms of its dependencies.

Node-link diagrams correspond well with the graph-representable data type of dependency information [4]. However, once the dependency graph grows in size, it is very difficult to maintain explorable representations that are useful. Matrix-based representations (e.g. Dependency Structure Matrix (DSM)) have better scalability characteristics when representing large hierarchical dependency graphs [5], and form the critical visual component of many software analysis tools [5],[6]. However, DSMs have a learning curve and are weak in path-finding operations [7].

The motivation for this work was the need of a familiar, interactive visualization for the dependency information aggregated by a software analysis platform in development. In order to provide a good *overview* [4], we wanted to help users get "oriented", i.e., identify what are the top-level or bottom-level parts. At the same time, the user should be given an impression of how "good" the dependency structure is. The number of nodes displayed had to be controlled to reduce cognitive load, while not losing information of how an area of interest is related to the rest of the system.

## II. Visualization Design

Polyptychon is designed for top-down, incremental exploration of compound graphs [8]. The raw dependency information at the source code level is recovered by the analysis tool using the Understand[1] library. The tool then aggregates the dependencies based on the directory structure, forming a compound graph.

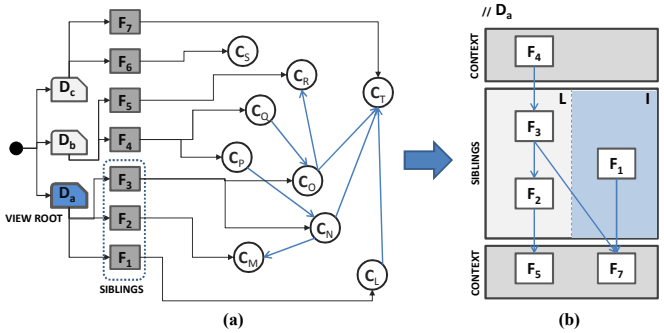



Fig. 1: Constrained, classified dependencies at a view root.(a) shows a compound graph consisting of a hierarchy of directories, files and classes; along with a dependency graph between the classes. (b) shows the aggregated dependency model calculated when $D_a$ is specified as the view root. The calculation constrains the sub-graph to only contain the sibling nodes, and directly related files outside the view root. The sibling relations are classified as *levelized* (L) and *independent* (I) in this example.

### A. Hierarchical constraints

From the global compound graph, a data model is computed on demand based on a node of interest, called a *view root* (by default, the root of the hierarchy). The dependencies visualized

[1]http://www.scitools.com/

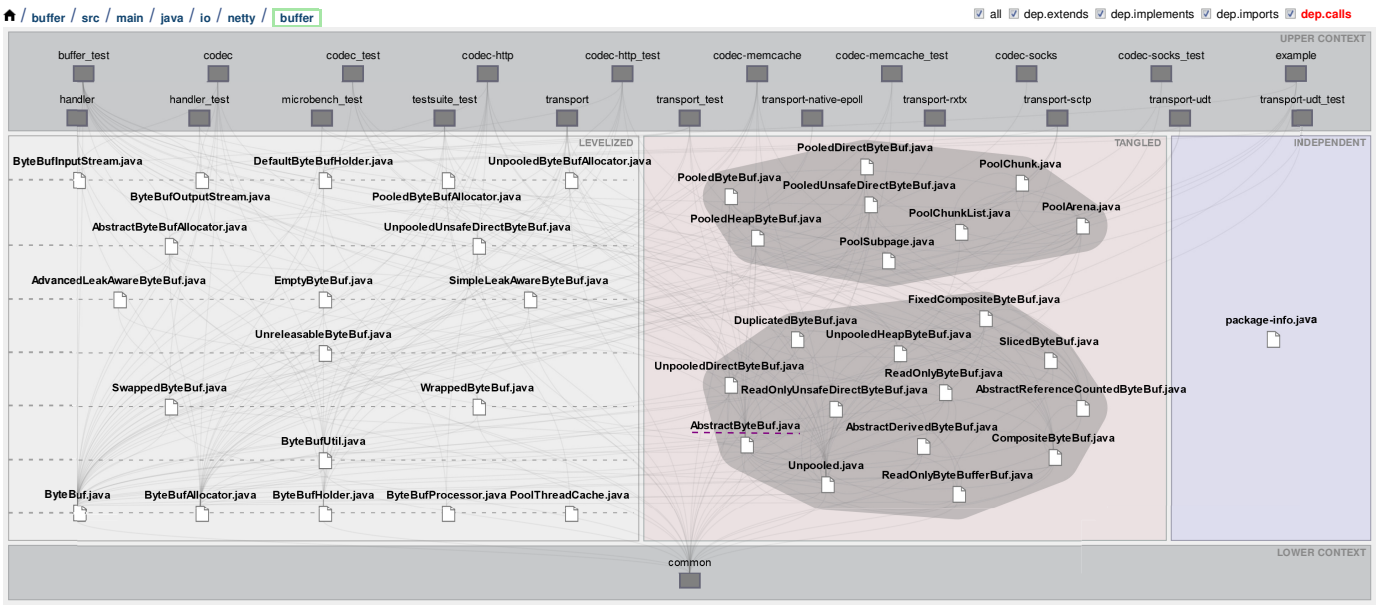



Fig. 2: Polyptychon visualization of the dependencies among the child nodes (siblings) of a directory in the Netty project.The key features are: **Node Classification**: The partitions in the middle contain the siblings classified as levelizable, tangled and independent. **Hierarchical constraints:** The dependencies outside this hierarchy location (context nodes) are laid out in the UPPER CONTEXT (incoming dependencies) and LOWER CONTEXT (outgoing dependencies) partitions. **Context partition function:** The context nodes are grouped, via a project-specific `ctxPartFunc()`, by top-level project folders with separate code and test groups per project (e.g. `buffer_test`, `codec` in the UPPER CONTEXT partition). **Controls:** The breadcrumb controls on the top-left indicate the current hierarchy location. The top-right has controls showing the different kinds of dependencies involved, and can be used to filter the underlying relation type. The call relation between files (*dep.calls*) is indicated as the relation causing tangles. **Drill-down:** Clicking on the underlined file, AbstractByteBuf.java, will lead to the visualization in Fig. 5.

are in terms of the child nodes of the view root (Fig. 1). A sub-graph that consists of these *siblings*, along with directly related *context* nodes (those outside the view root) is extracted. Context nodes are filtered appropriate to the level, e.g. if the siblings contain files and directories, the context nodes will have the same types. Any links between the context nodes are also filtered to reduce clutter.

### B. Classification

Given a view root, we now have a scoped dependency graph. We then perform additional classifications to aid the user in getting oriented and obtaining qualitative insights. To identify "potentially problematic" dependencies, we detect the presence of strongly connected components (SCCs). In a directed graph, an SCC is a sub-graph where each constituent node is reachable from any other node. We classify such nodes as *tangled* nodes.

The next classification is to isolate non-tangled nodes that don't depend on other sibling nodes, called *independent* nodes. Note that they can have dependencies with context nodes. If the independent and tangled nodes are removed from the sibling set, the remaining nodes form an acyclic graph, amenable to a levelization algorithm, e.g. by Lakos [9]. Hence, these nodes are classified as *levelized* nodes. Our levelization algorithm essentially performs Lakos in reverse. We assign the top-level nodes (no incoming dependencies) as level 0, and assign their dependencies increasing level numbers. The additional step is to assign the maximum $n$ value to all the bottom nodes (no outgoing dependencies).

In addition to hierarchical constraints, the number of nodes can also be controlled by a user-defined context partition function, `ctxPartFunc()`. This is required since the number of context nodes pulled into the data model can still be large. `ctxPartFunc()` adds *meta-nodes* and *meta-edges* [8] to the sub-graph that aggregate the dependencies of the nodes included in the partition. The aggregated nodes and edges are removed from the sub-graph. Fig. 2 shows an example of project-specific grouping (see caption for details).

### C. Partitions and Composite Layout

Before performing node layout, we first demarcate exclusive areas for each classification of nodes, namely Levelized, Tangled, Independent and Context. In particular, there are two areas used for Context: i) Lower Context, holding context nodes depended on by the siblings and ii) Upper Context, having the opposite relation.

The visual partitioning enables the user to get a quick picture of the kinds of dependencies at the selected root node. Each enclosure is visible only if the corresponding partition is non-empty. Each partition can now follow its own layout rules to aid in comprehension. The independent partitions do not have any inherent ordering, so we use a force-directed layout. The SCCs detected are also laid out similarly, but the bounds for each are determined by a single-level treemap subdivision of the tangled partition. The area of the treemap is proportional to the SCC size. Convex hulls enclose each SCC for better visual discrimination. For the levelized nodes, each level is rendered in its own row. The context nodes also follow a similar row layout. The row layouts try to fit the node labels

with an alternating (observe the top level in Fig. 2) or slanted arrangement (Fig. 6).

The distinct sub-layouts are packed together to form a *composite layout* of the graph relevant to the view root, emulating a BorderLayout found in GUI toolkits like Java Swing. Fig. 2 shows a typical visualization[2] with all five partitions.

### D. Interactions

When a user hovers over a node (*hover query*), only directly related nodes and edges retain their opacity, while the other nodes are faded out. Incoming and outgoing links are emphasized with different colours for better visual discrimination (Fig. 3 (b), Fig. 4). Clicking a node initiates a *drill-down* operation, loading a new model with the node serving as the view root. Navigating back up the hierarchy is made possible by a breadcrumb control (Fig. 2). *Details-on-demand* are provided when clicking on links (details of the nodes and dependency types), and the convex hulls (list of cycles in the SCC). The virtual hierarchy formed by `ctxPartFunc()` can be expanded to incrementally reveal the elided nodes. The user can switch the underlying relations to create different classifications of the sibling set (described in Fig. 2). Zoom and pan interactions are also supported.

## III. Case Study : Netty

We implemented Polyptychon as a web-based visualization using the d3[3] library. The dependency data is pulled on demand from the analysis tool via a REST API. We now discuss some visualizations generated from a revision of the Netty[4] codebase, a high-performance networking library consisting about 134,500 lines of Java code.

Fig. 3 (a) shows the root level of this project. The root of the project has no tangles, indicating a good dependency structure. We can also find a dependency pattern in the project where a number of higher level directories depend on the `common`, `buffer` and `transport` folders, emphasized by hovering over the nodes (Fig. 3 (b)).

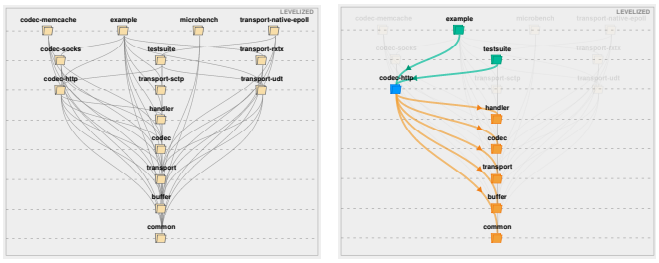
Fig. 2 shows an interesting location in the Netty project hierarchy, in the `buffer` sub-project. The most immediately grabbing feature is the presence of two large strongly connected components. We see that they do not disturb the layout of the parts of the dependency structure that are levelizable. The dependency graph originally has 416 nodes and 863 links. The custom context node grouping reduces it to 63 nodes and 265 links. For large graphs, the edges are deliberately de-emphasized with low opacity to reduce visual clutter.

Fig. 5 shows the result of drilling down on one of the "problematic" nodes. Fig. 4 shows how important nodes can be emphasized when interacting via hover queries. A more complex view root is shown in Fig. 6, with 98 nodes and 458 links, summarized from an original extraction of 591 nodes and 1843 links.

[2]The partitions together resemble a *polyptch*, a multi-paneled style of painting. Our visualization gets its name from the German word for the same.

[3]*D3: Data Driven Documents*, http://d3js.org/

[4]*Netty*, https://github.com/netty/netty/




(a) Default (b) On hover

Fig. 3: Visualization of Netty root level

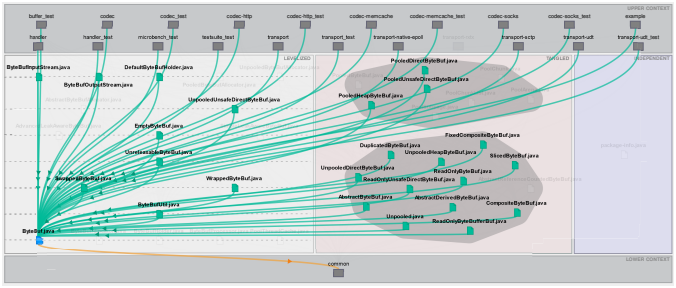


Fig. 4: Highlights on hover emphasize dependency patterns. Incoming dependencies (green) are observed from siblings as well as context nodes, indicating that the highlighted class `io.netty.buffer.ByteBuf` is a critical one. The view root is the same as Fig. 2

## IV. Related Work

Polyptychon can be characterized as an application-specific graph drawing algorithm, using composable layouts [10]. Sugiyama-based [11] layouts break cycles in directed graphs in order to get a layered layout. Dig-CoLa [12] presents a constraint-based layout that is able to take the level relations between nodes into account. Polyptychon removes cycle-inducing nodes before level calculation via SCC detection.

Nested graphs visualizations in tools like SHriMP [13] and Structure101 [6] allow incremental exploration of compound graphs. However, navigating deep into the hierarchy often increases the displayed graph area substantially, such that the context information is lost. Polyptychon provides con-

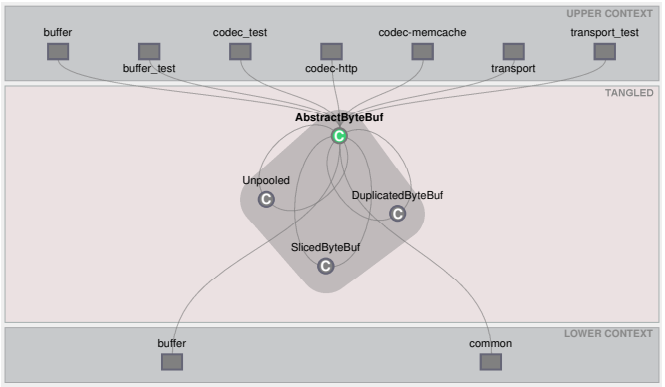



Fig. 5: AbstractByteBuf.java as view root. This file is involved in an SCC with its siblings (Fig. 2). When viewing the associated class (green node), we notice a smaller SCC that gives a scoped list of classes for investigation. Note that the other classes (grey nodes) are considered context nodes.

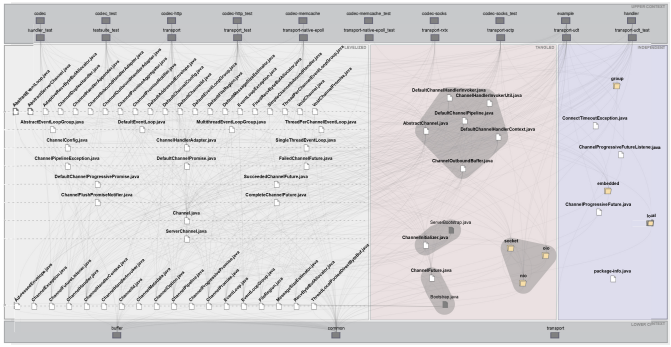


Fig. 6: Folder corresponding to the `io.netty.channel` package. This example shows that even at a reduced scale, patterns regarding top and bottom level nodes and SCCs can be distinguished. The levelized layout slants the text-labels to save on the width of the sub-layout, at the expense of the height.

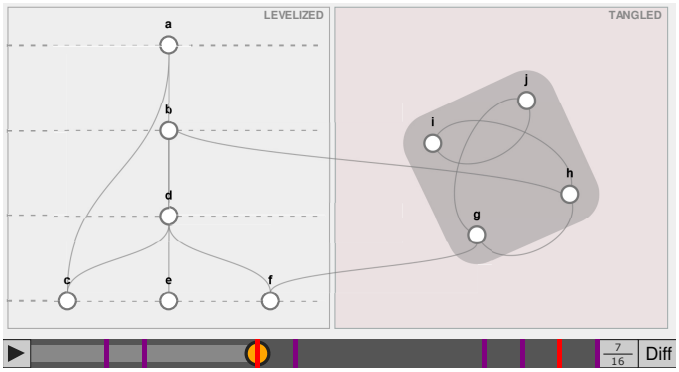



Fig. 7: Extending Polyptychon to an evolution view. The various revisions are represented in a time-bar, with markers to denote changes. Significant changes like introduction of tangles are highlighted.

sistent partitions given a rendering area, but is dependent on the number of nodes per view root and an appropriate `ctxPartFunc()` to avoid over-crowded layouts.

Hierarchical Edge-Bundling [14] reveals areas of interest in compound graphs by bundling dependency edges along the hierarchy. Grouse [15] presents a steerable graph hierarchy exploration tool that selects different layouts based on detected topological features (e.g. trees, connected components). Balloon Treemaps [16] combine circular treemaps and balloon tree layouts to visualize hierarchical information flows in social networks but do not address compound dependency graphs.

## V. Future Work

Polyptychon was developed using a *viewpoint-driven* approach, where we first specify the goal and concerns in an architectural viewpoint [17]. Visualization tasks are then derived from these concerns. Finally, the layouts and interactions are designed to accomplish these tasks effectively. We plan to conduct user studies to evaluate the effectiveness of the Polyptychon representation for the tasks identified, especially in comparison to other relevant visualizations.

Polyptychon currently does not support the display of metrics like lines of code or quality measures. Additional channels like node size, decorations and patterns could be employed to give more information to first-time code explorers.

We are also exploring the use of the Polyptychon representation as a way to make more effective use of dependency evolution information. A *time bar* with player controls is the main addition to the Polyptychon visualization (Fig. 7). Comparing two revisions, we can determine the differences and record them as "events", simple ones being the addition and removal of nodes and edges. However, the Polyptychon model can additionally determine higher level events like the addition and removal of tangles and context nodes. We mark the events on the time bar so that the user is given an impression of the amount of change in the inspected revision range. Higher-level events can have stronger colours to indicate the importance of the change detected, allowing users to quickly focus on the most the important revisions.

## References


[1] P. Avgeriou, M. Stal, and R. Hilliard, "Architecture Sustainability," *IEEE Software*, vol. 30, no. 6, pp. 0040–44, 2013.

[2] T. B. C. Arias, P. van der Spek, and P. Avgeriou, "A practice-driven systematic review of dependency analysis solutions," *Empirical Softw. Eng.*, vol. 16, no. 5, pp. 544–586, 2011.

[3] S. Ducasse and D. Pollet, "Software Architecture Reconstruction: A Process-Oriented Taxonomy," *IEEE Trans. Softw. Eng.*, vol. 35, no. 4, pp. 573–591, 2009.

[4] B. Shneiderman, "The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations," in *1996. Proc. IEEE Symp. on Visual Languages*, 1996, pp. 336–343.

[5] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Using Dependency Models to Manage Complex Software Architecture," in *ACM Sigplan Notices*, vol. 40, no. 10. ACM, 2005, pp. 167–176.

[6] "Structure101, Headway Software," http://structure101.com, 2014, [Online; accessed 25-May-2014].

[7] M. Ghoniem, J.-D. Fekete, and P. Castagliola, "On the readability of graphs using node-link and matrix-based representations: a controlled experiment and statistical analysis," *Information Visualization*, vol. 4, no. 2, pp. 114–135, 2005.

[8] T. Von Landesberger *et al.*, "Visual Analysis of Large Graphs: State-of-the-Art and Future Research Challenges," in *Computer graphics forum*, vol. 30, no. 6. Wiley Online Library, 2011, pp. 1719–1749.

[9] J. Lakos, *Large-scale C++ software design*. Addison-Wesley Professional, 1996, pp. 312–324.

[10] T. Pattison, R. Vernik, and M. Phillips, "Information Visualisation using Composable Layouts and Visual Sets," in *Proc. 2001 Asia-Pacific symposium on Information visualisation-Volume 9*. Australian Computer Society, Inc., 2001, pp. 1–10.

[11] K. Sugiyama, S. Tagawa, and M. Toda, "Methods for Visual Understanding of Hierarchical System Structures," *IEEE Trans. on Systems, Man and Cybernetics*, vol. 11, no. 2, pp. 109–125, 1981.

[12] T. Dwyer and Y. Koren, "Dig-CoLa: Directed Graph Layout through Constrained Energy Minimization," in *IEEE Symp. on Information Visualization*, 2005, pp. 65–72.

[13] M. Storey, C. Best, and J. Michand, "SHriMP views: An interactive environment for exploring Java programs," in *Proc. 9th Intl. Workshop on Program Comprehension*, 2001, pp. 111–112.

[14] D. Holten, "Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data," *IEEE Trans. on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 741–748, 2006.

[15] D. Archambault, T. Munzner, and D. Auber, "Grouse: Feature-based, steerable graph hierarchy exploration," in *Proc. 9th Joint Eurographics/IEEE VGTC conference on Visualization*. Eurographics Association, 2007, pp. 67–74.

[16] F. Viégas *et al.*, "Google+ Ripples: A Native Visualization of Information Flow," in *Proc. 22nd Intl. Conference on World Wide Web*. Intl. World Wide Web Conferences Steering Committee, 2013, pp. 1389–1398.

[17] ISO, "ISO/IEC 42010 Systems and software engineering – Architecture description," 2011.