Visualizing Work Processes in Software Engineering with Developer Rivers

Michael Burch, Tanja Munz, Fabian Beck, and Daniel Weiskopf
VISUS, University of Stuttgart, Germany
Email: {michael.burch,fabian.beck,weiskopf}@visus.uni-stuttgart.de

Abstract—Work processes involving dozens or hundreds of collaborators are complex and difficult to manage. Problems within the process may have severe organizational and financial consequences. Visualization helps monitor and analyze those processes. In this paper, we study the development of large software systems as an example of a complex work process. We introduce Developer Rivers, a timeline-based visualization technique that shows how developers work on software modules. The flow of developers’ activity is visualized by a river metaphor: activities are transferred between modules represented as rivers. Interactively switching between hierarchically organized modules and workload metrics allows for exploring multiple facets of the work process. We study typical development patterns by applying our visualization to Python and the Linux kernel.

I. INTRODUCTION

Work processes in general, and software development in particular, involve a product that is built and people building the product. Conway [9] states in his well-known law that these two aspects interact: “[...] organizations which design systems [...] are constrained to produce designs which are copies of the communication structures of these organizations.” This duality between people and systems needs to be considered for understanding work processes. Previous visualizations on work processes and software evolution, however, either showed one of the two aspects only or did not scale well to large systems and long histories.

With the visualization technique presented in this paper, we intend to bring both aspects of work processes closer together to help researchers, analysts, and project managers better understand, monitor, and control the development process. Our main goal was to build a system that visually and technically scales to large projects (e.g., Python, Linux kernel), where monitoring the work activity is most crucial and difficult to manage without tool support. Since software development is an important, cost-intensive process, we focus on visualizing the evolution of software projects. These need to be studied at a high level of abstraction and over long spans of time. However, we should as well be able to drill down to smaller subsystems, individual developers, and specific events. Aspects that might be explored with the help of the visualization are, for instance, main activities of developers, responsibilities for a module, individual developer roles, special evolutionary events, the flow of development activity, or general trends.

We designed a visualization technique called Developer Rivers, which connects developer activity with modules of a software system. Our technique represents development activity (i.e., developers modifying files) as river-like flows on a timeline, shown in Figure 1 for the evolution of Python. Each of the colored rivers encodes the developer activity within a user-defined module; rivers branch and merge according to the flow of activity between modules. A hierarchical representation of the project on the left is used to interactively browse and select modules. In a case study, we apply the technique to Python and the Linux kernel using multiple configurations of the visualization. We report specific findings for these software systems and discuss insights gained on the qualities and limitations of Developer Rivers.

II. RELATED WORK

Previous work related to Developer Rivers comes from several fields: software visualization [11], time-oriented data visualization [1], and the visualization of graphs [38] and hierarchies [16]. In general, our visualization combines a Sankey diagram [33], [27] (also referred to as flow map [25]) with a timeline. An additional hierarchy representation is used for navigation. While several hierarchy visualization techniques exist [16], we decided to use icicle plots [18] because they are compact as well as easy to label and colorize.

A Sankey diagram is a node-link diagram that encodes branching flows or quantities in the width of links. Those have already been mapped to a timeline, to encode splitting and merging of objects [28], the evolution of character interaction in stories [20], or the temporal development of topics in document collections [10]. In particular, our work is visually similar to the diagrams by Burch et al. [4] for visualizing eye movements of participants of an eye-tracking study. Using smooth bends and stacking the flows on top of each other without margins creates the impression of rivers. It looks similar to multivariate time-series visualizations such as ThemeRiver [17] or Streamgraphs [6]; these, however, do not show transitions between different rivers or streams. RankExplorer [30] uses flow-in and flow-out color bars as an alternative to links for encoding transitions in river-like categorical timeseries representations.

A common visualization for general work processes is the Gantt chart [13], which maps hierarchically organized tasks to a timeline and visualizes dependencies between the tasks. Also a number of approaches already focus on visualizing the evolution of software. Taking a rather technical perspective, releases or commits are visualized on a timeline [3], [12], [32], [36], [37], [35]; among these, the Code Flows approach [32]
also uses river-like streams—including splits, merges, and swaps—but represents flow of code fragments rather than developer activity. The technical evolution of software systems can also be visualized as sequence of coupling graphs [2], [5], [8] or a set of time-dependent software metrics [19], [26].

Considering social aspects of the development process, Storey et al. [31] survey approaches up to 2005. New techniques have been introduced since then: Timeline-based visualizations of software evolution, for instance, show evolving communities of developers [24], [23]. These approaches are similar to ours but do not directly relate developers with the modified code artifacts within the timeline-based visualization; at least, Software Evolution Storylines [23] are prepared to indirectly relate files and developers by mapping the most frequently changed file type of a developer to the color of line representing the developer. CodeSaw [14], in contrast, shows the activity of core developers, however, without encoding relations between developers. Ownership Maps [15] encode changing files on a timeline and color each of these according to the author who currently ‘owns’ the file; this approach only allows for discerning few developers. When color coding developers as well, similar limitations hold for the approach by Weißgerber et al. [39] and XFlow [29].

Animation was also applied visualizing the role of developers in software evolution. StarGate [21] shows the project hierarchy as a radial icicle plot and depicts an animated collaboration graph in its center; developers are placed close to the files that they changed recently. In code_swarm [22] and Gource [7], the developers and changed files are directly connected in an animated node-link graph. While code_swarm organizes files as ‘satellites’ of developers, Gource emphasizes the project hierarchy in the layout. In contrast, we do not use animation because static representations on a timeline have clear advantages for representing complex information [34]: in static diagrams, usually a good overview is provided for time sequences, whereas animation lacks this overview.

### III. Visualization Technique

The visualization combines two separate components: the hierarchy view for the module group selection and the rivers view for showing the time-varying developer activities.

#### A. Developer Activity Model

For a given software repository, we model a commit $c \in C$ as a triple $c = (t, d, f)$ consisting of a timestamp $t$, a developer name $d \in D$ and a set $f \subseteq F$ of involved files. The set $C$ denotes all commits, the set $D$ all developers, and the set $F$ all files involved in all commits. The organization of files $F$ into modules is defined as a hierarchy $H$ (i.e., a tree). Any subhierarchy of $H$ is called module, any set of disjoint modules is called module group.

To analyze the sequence of time, we partition the sequence of commits into intervals, either equally-sized with respect to the frame of time or with respect to the number of commits. For every interval of commits and every module, we calculate the individual activity of each developer as the number of files of the module edited by the developer within the given sequence of commits. By summing all individual developer activities of one module, a total developer activity of the module can be derived. Hence, each interval of commits can be described as a vector of module-specific developer activity, the full evolution of the module group as a sequence of such vectors.

To explicitly represent the changes between the consecutive commit intervals, we calculate weighted transition matrices $M_i \in \text{Mat}((l+1) \times (l+1), \mathbb{R}_+^\times)$. They describe the transition of developer activity between two commit intervals divided by the modules in the module group. Diagonal cells of the matrix represent a constant activity within the same module, off-diagonal cells a transition of activity from one module to another. An $l + 1$-th module group has to be added to model influents and effluents, i.e., developers joining or leaving the project. To prevent that activity is transferred from
one developer to the other, the transition matrices are first computed for each developer individually, before building the final matrices \( M_i \) by summing up all developer-specific ones.

\[ x_{S_1} = x_{P_1}, \quad x_{S_2} = x_{P_2}, \]
\[ y_{S_1} = y_{S_2} = y_{P_3} - r, \]
\[ y_{S_3} = y_{P_3} - y_{P_4}, \]
\[ x_{S_4} = x_{S_4} = x_{P_4} + r. \]

The segments \( P_1S_1, P_2S_2, P_3S_3, \) and \( P_4S_4 \) are straight lines. \( S_1C_1 \) and \( S_3C_1 \) are responsible for the radius of the outermost circle and \( S_2C_2 \) and \( S_4C_2 \) for the innermost one. Circle segments are placed between \( S_1 \) and \( S_3 \) as well as between \( S_2 \) and \( S_4 \). The value of \( v \) expresses the ratio for the circle radius. In the visualizations we choose \( v = 0.7 \) to reduce the effect of large bends, i.e., to produce aesthetically appealing diagrams. We apply a fade-out visual appearance at the top and bottom respectively to prevent influences and effluents dominating the visualization.

2) Adjustment of the Transition Matrices: Since developers might work in several modules before doing a commit, we have to adjust the transition matrix concept introduced in the work of Burch et al. [4]. For this reason, transitions between two subsequent time steps or time intervals cannot be regarded as a single transition between exactly two module groups. In general, developers’ amount of work has to be added/subtracted to/from the weights of several rivers, also when they enter or leave certain modules of interest. Another difference is that each developer typically has varying amounts of work in different work spaces. This aspect is also taken into account by the layout of Developer Rivers.

3) Vertical Order and Label Placements: In the vertical order of the subrivers we first render the influents, then the transitions, and finally the effluents. We first draw those transitions to the same module group and as a second step those which lead to the remaining module groups. All transitions between two subsequent time intervals are ordered by their

---

Fig. 2. A Developer River represented as (a) traditional ThemeRiver and (b) as an expanded representation showing the weighted transitions.

Fig. 3. Weighted transitions between (a) the same and (b) two different module groups, (c) an influent, (d) an effluent.
height, i.e., thinner subrivers are drawn in the foreground to make crossing transitions more readable.

We additionally annotate each time step by labels expressing the developers working most actively in the corresponding module groups, scaling the font size according to activity. However, we cannot show all developer names due to space limitations. To discern labels from colored rivers, labels are displayed as semi-transparent rounded rectangles on which the corresponding developer name is drawn. The most active developer is centered, followed to the top and bottom by other developers in order of decreasing activity.

IV. Usage Strategies

When applying Developer Rivers in practice, certain strategies help efficiently use the tool. On the one hand, we introduce visual patterns that have a specific meaning in the application domain; finding those in the visualization already reveals some of the relevant information. On the other hand, we describe application scenarios for our approach: each of them specifies a configuration and a form of application we identified as particularly useful.

A. Visual Patterns

Certain frequent patterns can be obtained from the river trajectories. These are simple geometric structures that can be detected by looking only at two neighboring periods and the transitions in between. To judge their relevance, not only the topology of the transitions needs to be considered but also their strength and context. If the patterns are not symmetric, a mirrored complement exists:

- **Inflow/Outflow**: A transition from or to the outside of the diagram identifies a developer entering or leaving the project, i.e., one who is not active in the previous period or in the following one.
- **Constant Flow**: An intra-transition with a constant width indicates a group of developers continuing to work on a module with steady effort. While the sum of activity does not change considerably, the activities of individual developers of this group might vary.
- **Growth/Decline**: An intra-transition with an increasing or decreasing strength hints at a group of developers that keep working on a module but with changing total effort.
- **Split/Merge**: A module that is split into or merged from multiple flows shows a qualitative change of developer activity (i.e., developers’ relative focus switches between modules). While at least one inter-transition is required for this pattern, one of the flows can be an intra-transition.
- **Exchange**: A pair of intra-transitions connecting two modules in opposite directions at the same time is a specific qualitative change of activity: some developers move between the two modules in both directions.

While these patterns only provide basic findings, the interplay or sequence of those standard patterns might create more complex patterns that lead to more advanced insights. Composites of the patterns can be formed through temporal succession (i.e., one pattern follows the other), through repetition (i.e., a pattern is repeated constantly or cyclically), or through co-occurrence (i.e., two patterns appear in the same timestep).

B. Application Scenarios

Developer Rivers are interactively configurable and allow, for instance, to select modules, to vary time frame and resolution, to select developers, or to change coloring and labels. The following scenarios are configurations we consider useful and apply throughout the case studies in Section V:

- **Main Module Overview**: The main modules of the system are selected, which usually consist of the main directories. While it is possible to simply select all directories of a level, it often is advisable to do a manual selection in order to obtain a better selection of modules.
- **File Type Overview**: The tool enables the automatic definition of modules by file types. We recommend that one does not use all file types existing in the project as modules but restricts the selection to the most frequent ones.
- **Developer Sparklines**: Selecting a single developer, a visualization is created that reveals specific characteristics of the individual activity of the developer. The reduced complexity of the visualization allows to down-scale the size of the visualization and create a kind of sparkline (i.e., a word-sized graphic). Several of these developer sparklines can be shown as juxtaposed small multiples.
- **Subsystems Details**: Selecting modules in a subdirectory of the system shows details of the specific subsystem. During the analysis of the subsystem, the hierarchy can be switched to only showing the subsystem.

We might want to focus on a specific time frame (e.g., only recent development), which is orthogonal to the scenarios described above and, hence, can be combined with these.

V. Case Studies

We applied Developer Rivers to two reasonably large open source software projects with a considerable development history: Python and the Linux kernel. The case study takes the perspective of an analyst or researcher trying to figure out how the development processes of the projects are organized, how responsibilities are distributed among core developers, and what subsystems have been and currently are the main focus of development. It follows a top-down approach, first starting with high level observations of the history, then investigating the role of main developers, and analyzing examples of subsystems.

A. Python

As first example, we investigated the evolution of the reference implementation of the programming language Python. While the development of Python started in 19891, we retrieved exactly 21 years of development history from January

1http://python-history.blogspot.de/2009/01/brief-timeline-of-python.html
1, 1991 to December 31, 2011 from the Mercurial repository of the project. This dataset consists of 70,813 commits by 159 developers. This project forms a suitable example for demonstrating our approach as it has reasonable size and history. Since the same dataset has already been visualized with code_swarm [22], Gource [7], and Software Evolution Storylines [23], it might be considered a form of benchmark.

1) Project Overview: Figure 1 gives an overview of the project. For sake of readability and simplicity, we cut the development history in clean periods covering only full years. In particular, we split the history of the project into seven periods each showing three years of development. Based on the project’s documentation, we selected four of the most important directories of the project as modules:

- **Lib:** “The part of the standard library implemented in pure Python.” (blue)
- **Modules:** “The part of the standard library (plus some other code) that is implemented in C.” (green)
- **Doc:** “The official documentation.” (red)
- **Tools:** “Various tools that are (or have been) used to maintain Python.” (pink)

One of the most obvious observations in Figure 1, which shows the main module overview, is that the project underwent a considerable growth with respect to numbers of changed files during the studied period and modules. The growth was quite constant, only with a slight decrease of changes during 2003–2005. The activity in the individual modules roughly follows the size of the modules in the project hierarchy; most commits covered the Lib directory, followed by Doc and Modules; Tools played only a minor role. This relation is mainly preserved over the years, only with an exception in the years 1997–1999, where Doc had the highest activity. This seems to be a particular phase of documentation, which might have been neglected during the early years of development.

Figure 4 shows the same time division as in Figure 1 but divides the project by the six most frequent file types into rivers (file type overview). Since file types largely conform as the modules selected before (Doc → .TEX, .RST; Lib → .PY; Modules → .C, .H; Tools → [diverse]), most of the above observation can be confirmed, such as the specific phase of documentation in the third period. Additional to that, Figure 4 provides some new insights: Visually quite apparent, during the last two periods, from 2006 and onward, the documentation format has changed from .TEX (LaTeX) to .RST (reStructuredText), not instantly but gradually (two main transitions between the orange and red river). As a posting on the Python developer mailing list tells\(^3\), the idea of switching the documentation had already been discussed since 2002; Guido van Rossum, the initiator of Python, reacted positively on the idea but was not directly convinced\(^4\). Another observation from Figure 4 is that the role of Python files (.PY) in relation to C code (.C, .H) is slowly increasing over the development history: at the beginning, the number of changes with respect to these two groups of files were nearly balanced, while at the end of the studied period, more than twice the number of changes referred to Python files than to C files. Hence, the library code seems to have gained importance in relation to the core implementation.

2) Developers: The labeled rivers of Figure 1 show who was working on which part of Python. In general, the fluctuation of developers’ activity was quite high because the main developers of a river rarely stayed the same from one time period to the next. Within a time period, different developer names for the selected modules indicate a certain division of responsibilities. Nevertheless, some developers (e.g., Guido van Rossum, Fred Drake, or Georg Brandl) appear as main ones of different modules. For the Doc directory, this is quite natural because implemented functionality (in one of the other directories) might need to be documented specifically.

As Figure 1 also shows, only few developers worked on the project at the beginning with minimal fluctuation (at most 9 developers during 1991–1999). In the period of 2000–2002, the project attracted many new developers and the number of them jumped to 50 (maybe in the context of the release of Python 2.0 in October 2000). Most of them kept working on the project throughout 2003–2005, although the number of commits declined slightly. From 2006 and onward, however, the fluctuation of developers increased: Many developers that were active between 2006–2008 did not continue working on the project during 2009–2011.

In general, developers do not often seem to switch their focus of work activity between the main parts: only few stronger transitions can be observed in Figure 1. The few existing transitions, however, may point to special events in the history of the project. While transitions can be explored interactively by retrieving additional information in tooltips, restricting the time frame increases the temporal resolution. We demonstrate this for the early years of Python, 1991–2001, in Figure 5. The higher temporal resolution (now 2 years instead of 3 years) and the vertical zoom much better reveal the activities and transitions in the focused period. For instance, we see that Guido van Rossum stayed the main developer of Python for a long time. A first interesting pair of transitions from Lib and Modules to Doc between the second (1992–1993) and third (1994-1995) period was caused by Guido

---

\(^{2}\)http://docs.python.org/devguide/setup.html

\(^{3}\)https://mail.python.org/pipermail/python-dev/2002-April/022131.html

\(^{4}\)https://mail.python.org/pipermail/python-dev/2002-April/022099.html
van Rossum editing more documentation files (as confirmed by using details on demand). These transitions fall together with Fred Drake entering development and changing Doc files. For the period of 1996–1997, the documentation effort was considerably intensified, which can be traced back largely to Fred Drake, who then became the most active developer of Doc. After that, during 1999–2000, Fred Drake started also to working in Lib and Modules and Guido van Rossum changed his focus back to these parts—this causes the strong transitions from Doc to the blue and green river between the last two periods.

Generating developer sparklines for the top 5 developers of Python allows to confirm most of these observations. The Developer Rivers as shown in Figure 6 encode again the full period of 1991–2011; since a stretched aspect ratio matches the character of a sparkline, the temporal resolution can be increased to one year per period. Studying the sparklines of Guido van Rossum and Fred Drake, we see the increasing effort in documentation and the subsequent switch of focus towards implementation. But much more insight can be gained based on these diagrams; just to name a few examples: Guido van Rossum steadily worked less on the project during 1998–2004 (with respect to changed files) followed by a sudden peak in 2007. Georg Brandl joined the project late (2005); his efforts switch back and forth between Lib and Doc. Benjamin Peterson, in contrast to the others, started to contribute heavily already in his first year. Jack Jansen had two unconnected periods of high activity, while Fred Drake particularly focused on Doc.

3) Subsystems: With these observations as a background, it is now interesting to go into the details of the module structure. As an example, we chose the Tools directory and mark some of its main subdirectories as new modules. Figure 7 depicts the subsystem details; due to limited space, we here restricted the studied period to 1997–2011, divided into 5-years intervals. The resulting diagram is quite different from those discussed before: the overall number of changes is not increasing, but roughly stays constant; there is a large variety of dominance comparing two time periods; and there are many transition connections between the different modules.

In the first period (1997–2001), scripts (a collection of scripts for various purposes), freeze (a Python compiler for Unix), pynche (a color editor), idle (a Python code editor), and compiler (a Python bytecode compiler) assembled the main activity. In the transition to the next period (2002–2006), there is no considerable outflow—nearly all Tools developers continued to work on the project, however, with overall less activity while some other developers joined. Within the period, scripts (see above), bgen (a source code generator), idle (see above), and pybench (a benchmark suite) were most active, following the strongest transitions between the two first periods, considerable developer activity moved from idle and compiler to scripts (split in idle and compiler, merge in scripts). In the last period (2007–2011), we again find a
different collection of modules being most active; in particular, buildbot (a continuous integration framework) draws considerable attention of developers. Only the scripts directory has major activity across all considered periods, but the developers are largely changing (considerable inflows and outflows).

**B. Linux Kernel**

The second project for this case study is the Linux kernel. Although we were only able to retrieve a part of the project’s history, this part is already at least one order of magnitude larger than the Python project: its Git repository for 8 years from January 1, 2006 to December 31, 2013 contains 408,555 commits by 11,048 developers.

1) **Project Overview:** Figure 8 provides the project overview for the following directories selected as modules (descriptions according to the Linux Documentation Project):

- **drivers:** “the system’s device drivers” (red)
- **fs:** “file system code” (blue)
- **arch:** “architecture specific kernel code” (green)
- **kernel:** “main kernel code” (pink)
- **net:** “kernel’s networking code” (cyan)
- **Documentation:** documentation files (orange)

In contrast to Python, the Linux kernel did not undergo an extensive growth of changes in the studied period (however, we also study a shorter period), but just a slight growth. A minor exception is the year 2013, where development activity slightly declined with respect to the previous year. It is further interesting to note that the overall pattern created by the flows is quite similar across all periods: a small inflow distributed among all selected modules (relative to their size in the period), a similar but even smaller outflow, large constant flows for all modules, and only small inter-transitions; only between drivers and arch, there are considerable inter-transitions showing an exchange pattern—this might be partly explained through their size, but could also mean that these two modules are related so that developers naturally switch between them. In general, the development of the Linux kernel seems to be an established work process with low amount of variance in the developer activity.

2) **Developers:** Although the overall development activity is quite stable, the most active developers within the modules selected in Figure 8 change quite often over the full time period. To further investigate the stability of developer roles, Figure 9 shows developer sparklines for the top 5 most active ones. We find, for instance, that some developers have clear responsibilities that stay constant over time (e.g., Greg

---

3http://www.tldp.org/LDP/itlk/sources/sources.html
Kroah-Hartman \rightarrow drivers or Russell King \rightarrow arch). Other developers, in contrast, switch their focus of activity (Al Viro: arch, drivers, fs; David Howells: drivers, arch). The overall activity of them is considerably varying over time, but none of the main developers left the project in this period.

3) Subsystems: Maybe the most important subsystem of the project is the kernel module. Figure 10 shows this directory as subsystem details with the most active subdirectory selected as modules. A first surprising observation is that this subsystem is not as stable as the overall system, even though 2-years intervals were selected to increase the readability of the figure. The overall activity as well as the relative activity in the different modules changes. In the first period (2006–2007), there is only little activity, while in the second period (2008–2009) much work is done in trace (the kernel tracing systems), to a large extent by new developers but also by developers previously working on time (time keeping functionality) and irq (handling interrupt request). Only some of this activity is preserved throughout 2010–2011, but irq got more active again. During the last period (2012–2013), previously quite inactive modules attract activity: sched (kernel scheduler) and events (performance events). Hence, the stability of the overall system and the relative stability of the main developers are not preserved in this subsystem, which is a particularly important one but also small in relation to the complete project.

VI. Discussion

The case study shows that Developer Rivers scale well to large systems and long evolution histories. Although the rivers highly aggregate the data, still interesting observations can be made at this highest level of abstraction. Interactively selecting smaller subsystems or choosing a narrower window of time further allows for investigating details. Selecting individual developers creates quite characteristic lines that can be interpreted even in small size like sparklines.

Nevertheless, our approach has a number of limitations, one of the main ones being the constraints in the number of modules and time periods that can be visualized: Since only a small number of colors can be visually discerned and vertical space is required for labeling the rivers, only up to about 10–12 rivers can be easily discerned, similar as it is the case for ThemeRiver [17]. Cushion effects, like used in Code Flows [32], could improve the discernibility of rivers somewhat. The limitation with respect to the number of time periods is caused by the transitions that connect the modules across time periods. Like in other approaches showing transitions on a timeline, the transition links require some horizontal space to be readable.

Like in most visualizations, visual and data artifacts may sometimes mislead the analyst. For Developer Rivers, the selected time granularity could make a difference: for instance, an entering developer who starts to heavily contribute to the project might either appear as a strong inflow or by a weaker inflow directly followed by a growth. Also, it cannot be visually discerned between developers not active only during one period of time and people leaving the project permanently. Knowing these potential problems, they can be mitigated by using the interaction techniques of Developer Rivers, such as retrieving details on demand.

A general limitation is that we tested the approach only with two open source systems. Development activity might be different in other projects, in particular, commercial ones. Hence, we do not draw any generalizing conclusions on development patterns based on our case study. Another problem that our work shares with other studies on development activity is that choosing a metric to estimate work effort is difficult. We decided to use the number of changed files as a simple metric that is easy to interpret, but this metric is replaceable by any other metric of development activity. Also developers using multiple alias names could be a problem; integrating a name disambiguation approach could mitigate this issue.

VII. Conclusion and Future Work

In this paper, we introduced Developer Rivers, a visualization approach for understanding development activity in large-scale software systems. Based on the committed code changes, we relate the software system in the form of hierarchically organized modules to developers. Good overview of the software evolution is provided through the timeline-based visualization technique. A flexible configuration and the interaction techniques allow the application of the visualization in different scenarios and on different levels of abstractions. Described visual patterns help quickly gain insights from the visualization. The extensive case studies provide examples that show the usefulness of the scenarios and patterns; they also demonstrate that our approach scales to large systems.

For future application of Developer Rivers, we see two main directions: First, we want to make the tool ready for easy application by practitioners, for instance, software consultants, project managers, and lead open source developers. For this, mainly technical issues still need to be solved (e.g., bundling preprocessing scripts and visualization tool or integration into existing development environments), but also the features of the tool need to be restricted to the most central ones to increase acceptance. Second, we see the visualization as a potentially helpful tool for software engineering researchers to derive hypotheses about development patterns and developer...
roles. These hypotheses might act as starting points of new research projects and experiments studying the socio-technical aspects of software projects.

ACKNOWLEDGMENTS

Fabian Beck is indebted to the Baden-Württemberg Stiftung for the financial support of this research project within the Postdoctoral Fellowship for Leading Early Career Researchers.

REFERENCES