Visual software analytics for the build optimization of large-scale software systems

Alexandru Telea & Lucian Voinea

Computational Statistics

ISSN 0943-4062
Volume 26
Number 4

Comput Stat (2011) 26:635-654
DOI 10.1007/s00180-011-0248-2

Springer

Comput Stat (2011) 26:635–654
DOI 10.1007/s00180-011-0248-2

ORIGINAL PAPER

Visual software analytics for the build optimization of large-scale software systems

Alexandru Telea · Lucian Voinea

Received: 21 September 2008 / Accepted: 11 February 2011 / Published online: 25 March 2011

Abstract Visual analytics is the science of analytical reasoning facilitated by interactive visual interfaces. In this paper, we present an adaptation of the visual analytics framework to the context of software understanding for maintenance. We discuss the similarities and differences of the general visual analytics context with the software maintenance context, and present in detail an instance of a visual software analytics application for the build optimization of large-scale code bases. Our application combines and adapts several data mining and information visualization techniques in answering several questions that help developers in assessing and reducing the build cost of such code bases by means of user-driven, interactive analysis techniques.

Keywords Software visualization · Visual analytics · Static analysis

1 Introduction

Software is everywhere. It is continuously being developed by an estimated 15 million engineers worldwide (Booch 2006), in a hierarchy of activities, ranging from requirement gathering, specification, and design, to implementation, debugging, testing, and maintenance. Understanding software is a towering task. Nowadays software systems are huge: The Mozilla browser has over 2 million lines of code (MLOC) in over 5,000 files (Mozilla Inc. 2008). Banking, telecom, and industrial applications are an order of

A. Telea (✉)
Institute for Mathematics and Computer Science,
University of Groningen, Groningen, The Netherlands
e-mail: a.c.telea@rug.nl

L. Voinea
SolidSource BV, Eindhoven, The Netherlands
e-mail: lucian.voinea@solidsource.nl

magnitude larger. Software code is structured in many ways, e.g., as a file hierarchy; a network of components or packages; a set of design patterns (Gamma et al. 1995), or aspects (Elrad et al. 2003). No single hierarchy suffices for understanding, and the inter-hierarchy relations are complex. If we add design, architecture, documentation, dynamic and profiling data to source code, the understanding challenge explodes. Finally, software continuously evolves, which only increases complexity, as described by the so-called laws of software evolution (Belady and Lehman 1976; Godfrey and Tu 2000). Overall, understanding software is hard, as it is large, complex, abstract, and changing (Klemola and Rilling 2000).

Given the large amount of complex legacy software, maintenance is the most effort-consuming activity in the software life-cycle. Studies over 15 years, from Standish (1984) to Corbi (1999), estimate that over 80% of the cost spent in the software life-cycle goes into maintenance. A significant component (40%) of this cost represents software understanding. Hence, it is of crucial importance for software professionals to be empowered with tools that enable them to reduce the understanding cost efficiently and effectively. In the following, we shall focus on understanding static software source code, which is a major component of the maintenance process.

From a data modelling perspective, software code is similar to a database: it consists of a set of entities ranging from code lines to functions, classes, files, and components; and relationships, such as containment, data, call, and build dependencies. Entities and relationships have multiple attributes of numerical, ordinal, or textual type, e.g., quality and complexity metrics, types of data access, and the source code itself.

Understanding large relational databases involves activities such as data mining, exploration, and presentation. A rapid growing field addressing this goal is *visual analytics*, which combines data mining and information visualization techniques to help users extract and reason about the information contained in such data collections (Wong and Thomas 2004; Thomas and Cook 2005). Visual analytics has been successfully applied in several domains such as network monitoring, banking, traffic control, and homeland security. Central to the application of visual analytics in a particular domain is the customized design of tools and techniques to reflect the questions to be answered about the data at hand. To be time-effective, such tools should reflect the way their intended users reason about their data *and* questions, and also be scalable, integrated, and interactive.

Although many visual methods and tools have been created for software understanding and maintenance, few of them have gained wide acceptance in the software industry. On the other hand, many data mining methods exist and are used in software engineering, but few support the visual analytical reasoning advocated above. This opens new opportunities but also poses several questions and challenges. In this paper, we explore the application of visual analytics principles and techniques to software maintenance, in what we call *software visual analytics*. First, we analyze the specific requirements and constraints of software maintenance. Next, we detail how the principles of visual analytics can be best put to use in light of these requirements. Finally, we demonstrate our model by an application of software visual analytics in solving a concrete problem on industrial software systems: the optimization of build performance of large code bases. Our application of visual software

analytics demonstrates the high applicability of visual analytics principles to software understanding, from data collection and mining to hypothesis forming, validation, and presentation.

This paper is structured as follows. In Sect. 2 we briefly overview the basic principles of visual analytics. Section 3 details the specific requirements and challenges of source code understanding in software maintenance. Section 4 presents our model for a visual software analytics framework, outlining the elements needed for its success. Section 5 presents an instance of a visual software analytics framework for a concrete problem from the software industry, the build analysis and refactoring of a large code base. Section 6 discusses our results, based on actual feedback from users of our systems. Section 7 concludes the paper.

2 Visual analytics: an overview

Visual analytics is defined as "the science of analytical reasoning facilitated by interactive visual interfaces" (Wong and Thomas 2004). Its main ingredients are a tight combination of data mining and visualization techniques aiming at supporting the reasoning about phenomena captured in a given set of data. Visual analytics differs from pure data mining, as the involved reasoning cannot be captured in simple data queries. Also, visual analytics is more than data visualization, as the questions asked often require reinterpretation of the data at hand and the generation of multiple visualizations showing different aspects.

Operationally, visual analytics involves a pipeline of activities that refine and enrich basic data with semantics related to the questions to be answered (Thomas and Cook

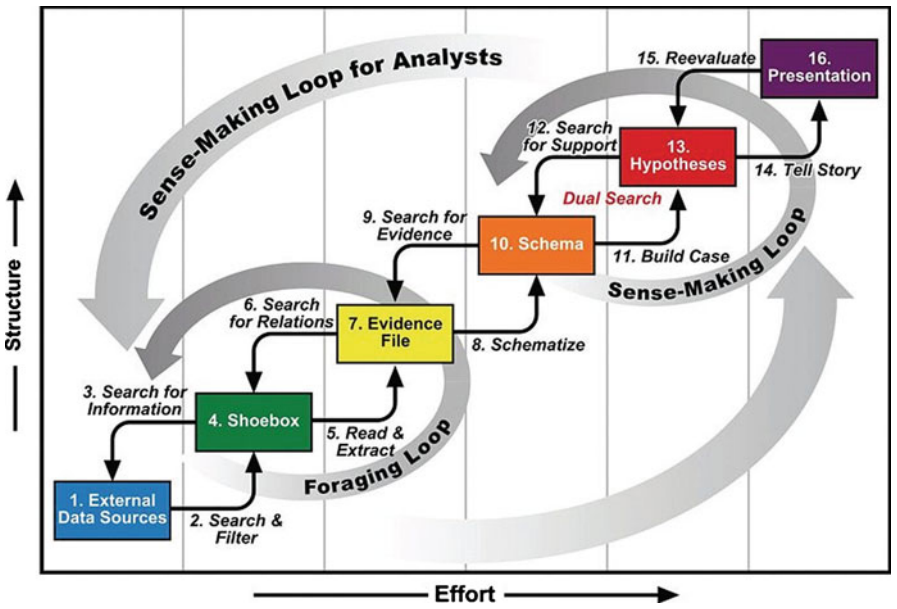

Fig. 1 The visual analytics sensemaking process seen as a set of iterative activities (from Thomas and Cook 2005)

2005) (see Fig. 1). First, data is searched and filtered and elements of interest are extracted in the so-called *data foraging* loop. This is mainly a data mining step, e.g., "extract all modules and module dependencies in a software code base". Secondly, a hypothesis is formed. A refined data schema is structured to reflect the hypothesis, and the basic data is fit to it to (in)validate the hypothesis, in the so-called *sense-making* loop. This is mainly an interactive visualization step, as sense-making cannot be encoded into queries, e.g., "find the modularity problems of a software system". The two loops are repeated several times during an interactive hypothesis formation-and-testing process. The challenge is to find the right methods and tool design to effectively support this.

3 Software maintenance and program understanding

Software maintenance follows the actual development of a software product (Fig. 2), and focuses on correcting faults (corrective maintenance), adding new features (perfective maintenance), and adapting the software to a new environment (adaptive maintenance, see Pigorsky (1996)). As already mentioned, maintenance is by far the most expensive part of the software life-cycle. A main component of maintenance is *understanding* the software at hand, which is time-consuming as software systems are large (millions of lines-of-code) and complex (highly inter-related).

Two types of tools are prevalent in software understanding for maintenance. *Reverse-engineering* tools extract various types of facts from the source code, such as annotated syntax trees, and module dependency and call graphs (Balanyi and Ferenc 2003; Telea and Voinea 2008; Lin et al. 2003; Baxter et al. 2004). Atop of such facts, various quality metrics can be computed, such as lines-of-code, comment density, complexity, cohesion, and coupling (Lanza and Marinescu 2006; Littlefair 2007). The basic facts are further refined by task-specific data aggregation and clustering techniques, *e.g.* grouping functions and classes based on maintainability metrics. *Software*

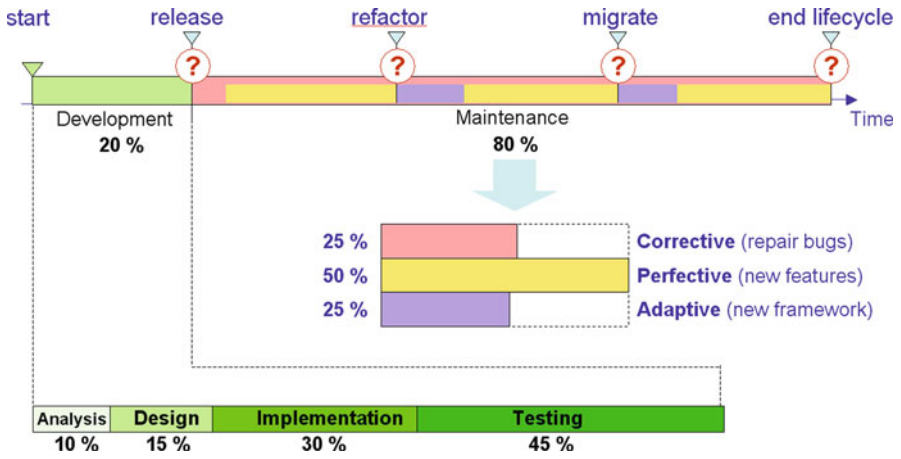

Fig. 2 The software life-cycle: Maintenance is the dominant cost component

visualization tools present the extracted facts using various metaphors such as data-annotated graphs (Tilley et al. 1994; Lanza 2004; Lienhardt et al. 2007; Telea et al. 2002), data tables (Telea and Voinea 2008), and metric-annotated code text (Eick et al. 1992; Lommerse et al. 2005). An overview of software visualization techniques and tools is given by Diehl (2007).

4 Software visual analytics

There is currently a large gap between software visualization and software data mining and metric tools. In terms of the visual analytics pipeline shown in Fig. 1, the data foraging part, consisting of the software fact mining and metrics computation, and the sensemaking part, consisting of the interactive software visualization, are typically separated in practice by being supported by different tools. The overall visual analytics loop is broken.

However, we argue that software understanding in maintenance perfectly fits the aim of visual analytics methods. We designate the combination of software fact mining, metrics computation, and presentation by the term *visual software analytics*. Just as in classical visual analytics, software data is multivariate, relational, large, and abstract. Also, the general building blocks of visual software analytics are instantiations of the generic visual analytics pipeline steps (Fig. 1), as follows:

- *External data sources:* software static artifacts such as source code, UML documents, and binaries; dynamic artifacts, such as log files; and evolution artifacts, such as change data from code repositories;
- *Shoebox:* contains the basic data mining tools, such as parsers and fact extractors (for static data); profilers and loggers (for dynamic data); and repository analyzers (for evolution data);
- *Schema:* contains the refined mined data, such as call or dependency graphs (for static data); aggregated performance or code coverage statistics (for dynamic data); and change sets (for evolution data);
- *Hypotheses:* holds the use-case-specific hypotheses in program understanding. These are typically related to the reasons for low values for a given software quality aspect under optimization, such as modularity and coding standards compliance (for static data), performance (for dynamic data), and decrease of software entropy during evolution (for evolution data, see Belady and Lehman (1976)).
- *Presentation:* contains software visualization views, such as graphs, trees, and pixel-oriented plots (Eick et al. 1992; Lommerse et al. 2005) for static data, performance graphs for dynamic data, and trends visualizations for evolution data.

Moreover, just in classical visual analytics, a successful software visual analytics method must be:

- *scalable:* the method must handle code bases of millions of lines-of-code;
- *interactive:* the method must allow users to formulate "what if" queries on-the-fly;
- *integrated:* the data mining and visualization operations must be tightly integrated to facilitate analysis and exploration.
- *simple:* simplicity of use is essential for the time-constrained software industry;

However, we argue that this particular application of visual analytics to software maintenance poses particular problems and challenges, and requires specific solutions, as compared to visual analytics applied in classical domains such as banking, document analysis, network monitoring, and surveillance. These specific aspects of visual software analytics are discussed below.

4.1 Highly structured text

Software code is highly structured as opposed to general text documents. Parsing tools can transform source code to annotated syntax trees where each word in each line of code has a specific semantics and relation with the rest of the program. The questions asked by programmers at the code level intimately relate to this structure, e.g., "which functions are called by a given code fragment?" or "which are the code fragments that affect the value of a given variable?". Programmers work both with source code, and also higher-level abstractions such as UML diagrams (OMG 2008). Hence, a visual software analysis solution must support posing such queries *and* displaying their results directly on both the code itself and higher-level diagrams.

4.2 Precise semantics

Software artifacts have a precise meaning. For example, a function or module is designed to fulfill a given task, and has well-specified dependencies with other functions. Whereas classical visual analytics methods deal with uncertain data and use statistical methods to reason about the similarity of data items, maintenance activities such as refactoring must analyze, manipulate and present the software in an exact way. One of the fundamental additional elements present in software understanding, as compared e.g., to understanding large sets of documents or numerical databases, is the presence of complex *type system*. Each software element has a type, which has complex structural and functional relationships with other types. This poses a challenge when visually analyzing large-scale systems. The problem comes from the difficulty of aggregating type (semantics). In text or numerical databases, this is typically done by computing statistics or feature vectors that capture the 'average' of a set of elements. However, in software, it is usually not clear how to average a set of software artifacts such as lines of code or functions. Aggregation offered by the natural software hierarchy (functions, modules, files) can be used, but is limited to the few such levels explicitly present in a given architecture. Pixel-filling methods enhance scalability by using (almost) every pixel to show a different software artifact (Eick et al. 1992; Lanza 2004; Lommerse et al. 2005). However, the classical display of relations using node-link diagrams has limited scalability.

4.3 Constrained layouts

Software engineers have well-defined, well-perfected, ways of working with software, e.g., text editors for source code, diagram viewers for architectural models, and tables

for metrics. In particular, 3D visualizations and unconstrained graph layouts have not been so far adopted by typical software developers and have remained mainly in the research realm (Teyseyre and Campo 2009). In contrast, visual analytics methods implemented in successful tools for general datasets have used a wealth of different layouts such as general graphs (PNNL 2008), treemaps (Shneiderman et al. 2008), and flows (Havre et al. 2002). The key difference here is that such datasets are not created by humans in an editing process (as opposed to software code or diagrams), so they do not have a preferred layout or predefined mental map. Given a constrained 2D layout, an additional challenge is to map several software attributes, e.g., metrics, atop of such a representation.

The success of a visual software analytics method or tool is largely determined by design. In the following, we present two instances of visual analysis applications in the software maintenance process. In particular, we detail the rationale behind the design decisions, and the way in which several data mining and visualization techniques were adapted and combined to support hypothesis forming and validation.

5 Application: build process optimization and refactoring

Systems of millions of lines of C (or C++) code, developed by teams of hundreds of people for several platforms, are commonplace in embedded, automotive, and electronics industries. Although modular, such systems face a scale problem: whenever a header file is modified, all the source, library, and executable files that depend, directly or not, on it, must be rebuilt (recompiled). Hence, even small changes to certain files can cause huge rebuild times. This slows down development, debugging, and testing speed for systems maintained by large teams working worldwide around the clock, causing non-negligible financial losses.

In this context, a project manager or team leader is interested in answering several questions:

- What is the exact model of the build *cost*, i.e., how can one predict the build cost of a system given a specified set of changes to its source files? Understanding this can help scheduling file modifications or preventing modifications of certain high-impact files at undesired times, to ensure minimal build bottlenecks.
- How is the build cost *spread* over the system, i.e., which are the files that have the highest impact on the build time, and why? Understanding this can help architects in planning a reorganization of the softwae system to isolate problematic files.
- Which are the changes (*refactoring*) that one could apply to the header files in order to reduce the build time with minimal modifications? An automatic system that suggests changes can be of great help for an architect, since manual refactoring is highly complex for a system of thousands of files.

We have studied such a concrete case in the industry and developed a visual software analysis system to address the above questions.

To answer these questions, we execute several steps, as follows. First, information on actual build times is gathered and analyzed to detect the potential build slowdown causes (Sect. 5.1). This is the basic data gathering step in the visual analytics pipeline. Second, a build cost model is constructed to describe the actual build process

(Sect. 5.2). This is the hypothesis forming phase in the visual analytics pipeline. Third, the build cost model is tested against the measured data, to validate the hypothesis (Sect. 5.3). Fourth, additional information is mined from the system and visually presented to correlate the build cost distribution with the system structure (Sect. 5.4). This is the presentation phase in the visual analysis pipeline. Finally, a method is built to assist in automatic refactoring, based on the system structure and cost model. These steps are described next.

5.1 Information gathering

Before we proceed, let us define some notations. The entire code base, stored in a software configuration management system (CM/Synergy in our case) consists of several thousands of files f_i. These files are further differentiated in platform files that come with the C compiler system (e.g., `stdio.h`) and client files, part of the application proper. Client sources and headers are used to build binary objects, which are further assembled into libraries and executables. The build *cost* $BC(f)$ of a file f is the effective time needed to build that file when any of its dependencies change. Headers have a zero BC, since they are not compiled themselves. Sources, objects and libraries have non-zero build costs. The build *impact* $BI(f)$ of a file is the effective time needed to build the entire system when f changes. We are primarily interested in the BI of headers, since these are the main points of change propagation. Hence, the goal is to predict and isolate high-impact headers and reduce this impact by header refactoring.

In the information gathering phase, we measured the build cost of the entire system upon modification of each client header, i.e., each header's build impact, using the system `time` function. Figure 3 (top) shows this cost. We see that about 80% of the headers have a quite small impact, whereas the remainder (located at the extreme right) have a very high impact. This supports the hypothesis that a refactoring is possible, since the high costs are relatively concentrated.

5.2 Build impact model

In the second step, we proceed to the creation of a build impact model. This will support our "what if" analysis, i.e., show what the build cost would be if a file were modified. As a first approximation, we hypothesize that the build impact of a header equals the number of sources it is used by, directly or indirectly. We call this the *simple* build impact model. Header-header and header-source dependencies are extracted by parsing their C code. For this, we used several tools: the CScout parser (Spinellis 2007), the `gcc -M` compiler option, and the SolidFX C parser (Telea and Voinea 2008). We also tried other tools, e.g., Makepp (Holt 2007), CScope (Bell Labs 2007), and Ctags (Ctags Team 2007), but none provided the complete set of qualified file dependencies we need. CScout and SolidFX have the additional advantage that they also provide header symbol information, i.e., all global functions, data types, and macros, which we use next in our refactoring analysis (Sects. 5.5 and 5.6). Among the two, SolidFX

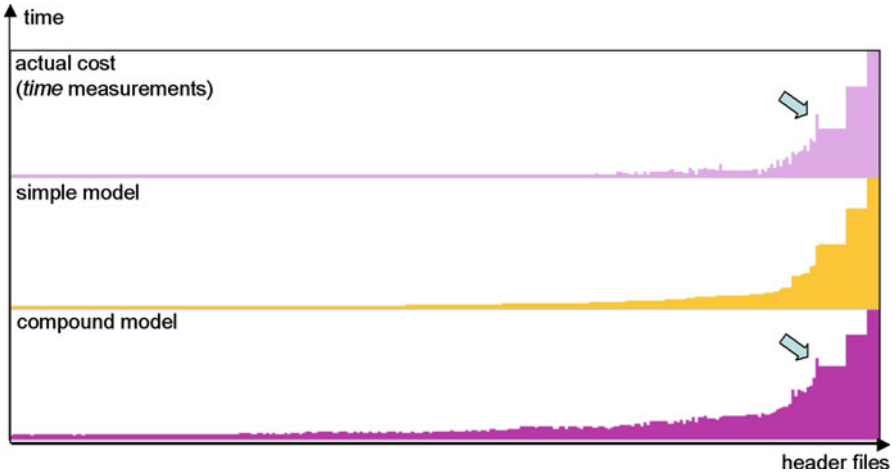

Fig. 3 Build impact overview. *Top* actual measurements. *Middle*: simple cost model. *Bottom* compound cost model. *Arrows* indicate the better match between the actual measurements and the compound model. Entries on the horizontal axis are header files. The y axis indicates build time. Headers are sorted on increasing build impact in the simple model (see Sect. 5.1)

provides more accurate information and is also faster than CScout, which makes it our tool of choice.

Figure 3 middle shows the build impact model described above. We see that the predicted values match quite well the measured values, and the model is attractive since it is simple to compute. However, there are some outliers, one of which is indicated by the arrows. This header has a clearly larger measured impact value than predicted by the model. Since its impact is high, an underestimation of this value is not desirable.

Upon detailed examination, the model's problem became apparent: The impact of a header h should equal the sum of the build costs $BC(s_i)$ of all sources s_i using h directly or indirectly. However, these costs are in general not equal, since different sources s_i can, in turn, have different build costs depending on which *other* headers they include besides h. For instance, a header can have a huge impact even if used in few sources, if those sources include many other headers. To validate this assumption, we measured the build time (build cost) for individual sources. Figure 4 shows the results, sorted by build cost. The time is broken into total translation cost (compilation + preprocessing) and preprocessing. We notice that, on the average, 60% of the cost is preprocessing. Further analysis (not shown here) has shown that this cost is almost linear in the number of included headers, and not dependent on the headers' sizes. This can be explained by the very high I/O time needed by the CM/Synergy filesystem.

5.3 Hypothesis validation

Figure 3 bottom shows the predictions of our refined build impact model described above, which we call the *compound* model. We see that this model accurately captures the outliers, as indicated by the arrow in the figure. Hence, the hypothesis established

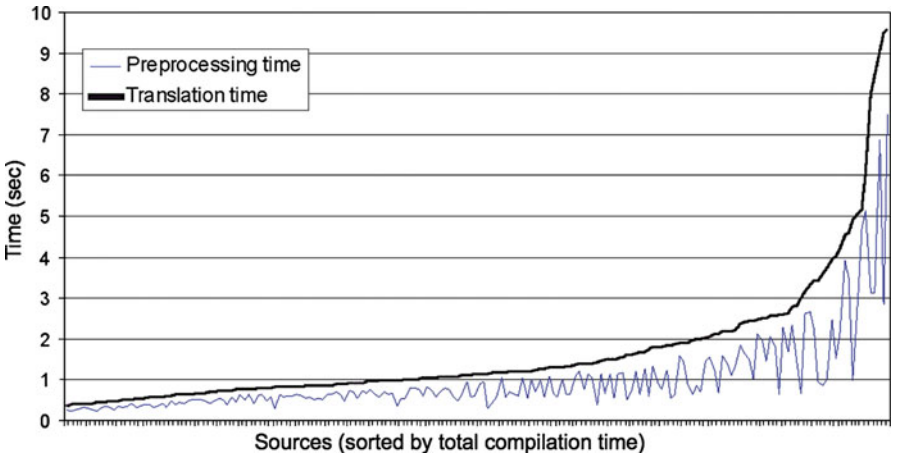

Fig. 4 Build cost overview for source files. Time is broken into total compilation or translation (*thick graph line*) and preprocessing (*thin graph line*). Each *tick mark* on the horizontal axis represents a different source file

above that the build costs essentially depend on the total number of opened files in the build process is validated. We shall use this model in the remainder of the analysis.

5.4 Cost analysis tool

In the next step, we try to answer the question concerning the spread of the build costs and impacts over the system structure. To support this analysis, we implemented an interactive visual analysis tool, as follows. Figure 5 shows a snapshot of the tool, which mimics the appearance of a classical development environment. The *architecture view* shows the tree-like structure of the system under analysis, consisting of systems, components, and files. Each item is colored according to its total build cost, using a blue-to-red colormap. This lets users quickly locate high-cost subsystems. Upon selection of an item in the architecture view, all sources and headers contained in that item are displayed in a second view: the *cost/impact view*. This view is structured as a table: each file is a row, and each column indicates a different attribute. Several attributes are displayed here: the file name (A), impact (B), simple cost (C), and compound cost (D).

This view uses the well-known table-lens technique (Rao and Card 1994). Each cell shows, besides the numerical value, a bar colored and scaled to indicate that value. The table can be zoomed out, thereby reducing each row to a pixel row, and each column to a colored bar graph, similar to the ones shown in Fig. 3. Clicking the columns sorts the table on the underlying metric. Reducing tables to sets of graphs allows outliers and value distributions to be easily seen. For example, in Fig. 5, which is sorted ascendingly on the impact, we can quickly see that only approximately 10% of the files have high impact (the red bars at the bottom of the cost/impact view). The numerical values, also available in a tooltip, indicate the actual impact values predicted

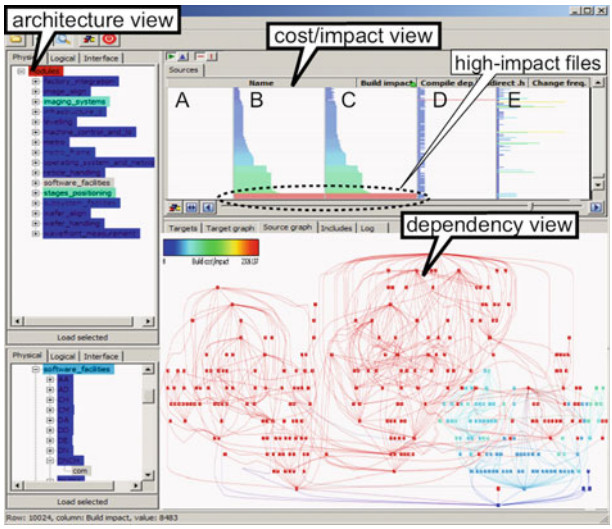

Fig. 5 The different views of the build cost analysis tool (see Sect. 5.5)

using the compound model from Sect. 5.2. This answers the question "which are the most costly files to be modified?".

However, the fact that a header has high impact does not mean necessarily it is a build bottleneck. Some files are changed very rarely during the lifetime of the project, whereas other files undergo intense development by many developers. To capture this aspect, we introduced a new metric: the time-impact. This is equal to the build impact times the number of times the file changed per year, or change frequency. This metric is depicted in column E in Fig. 5, and enables users to pinpoint those files that have a high impact *and* are often changed, thus are the real build bottlenecks during the daily development activities. These files are the primary candidates for refactoring, which is discussed next.

5.5 Refactoring analysis

After the developers agree on the bottleneck headers, as discussed above, it remains to see how easy it is to change, or refactor, the system to remove such bottlenecks. A first view that assists this process is the *dependency view* (Fig. 5 bottom). When the user selects a bottleneck header h in the cost/impact view, the dependency view shows the entire dependency graph having h at top and all files that depend on it, i.e., which need rebuilding when h is changed, below. These files can be colored by various metrics, e.g., the build cost, using a blue-to-red colormap. The graph shows how the build cost is actually spread through the system. Small graphs, or graphs with few hot colors, indicate easy refactoring situations. In such cases, the costs are localized in a few files and refactoring can be performed by removing a few *#include* relations, i.e., deleting a few graph edges, and patching the system by adding the symbols imported

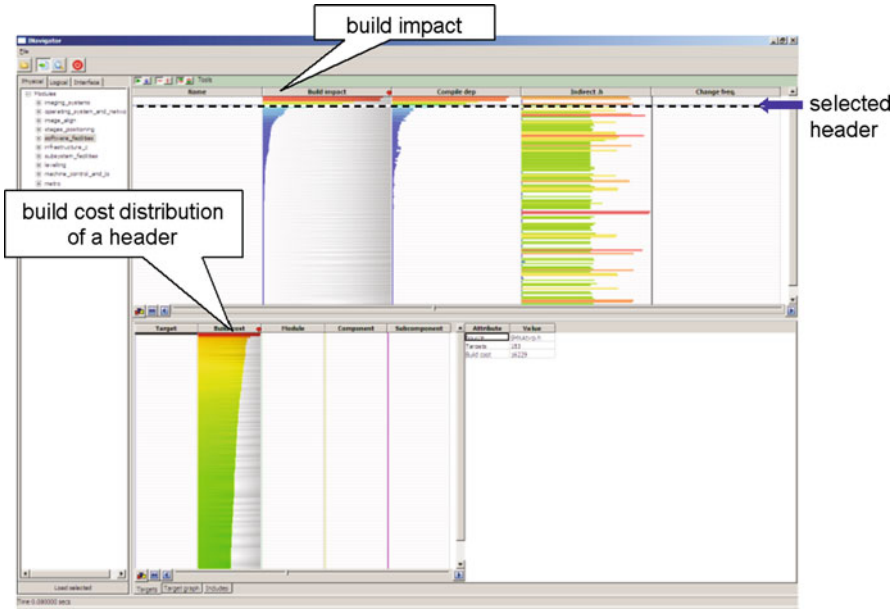

Fig. 6 Cost distribution of a selected bottleneck header (indicated by the *dotted line* in the table lens view). If the distribution is flat, refactoring is difficult

from these headers by hand to their clients. If the dependency graph is large and the cost is relatively uniformly spread, manual refactoring is difficult.

To simplify this assessment, we added a supplementary view: the *cost distribution* view (Fig. 6). This view shows, for a header selected in the cost/impact view, all build costs of all the sources it is used in, directly or indirectly, using the table-lens technique. For example, Figure 6 displays a relatively flat distribution, indicating that the build cost of the selected bottleneck header are spread quite evenly over all the sources that use it. Hence, this header must be removed from *all* the sources that use it, in order to significantly decrease the build cost. When there are many such sources, i.e., when the table has many entries, this is clearly a difficult process to be executed manually, so this header is not a good candidate for refactoring. Conversely, when the distribution is exponential, i.e., the header's impact is concentrated in a few sources only, manual refactoring is a good option.

5.6 Automatic refactoring

As discussed above, manual refactoring of the header dependencies may not be applicable in several cases. Moreover, for a large system of thousands of files, even if this were possible, it would not be practical. To support such cases, we investigated the option of providing automatic refactoring.

In the general refactoring case, headers must be split by grouping symbols which are used together, i.e., by several client files, in smaller headers. We proceed as follows. First, the user finds a bottleneck header using the analysis described in Sect. 5.5. We

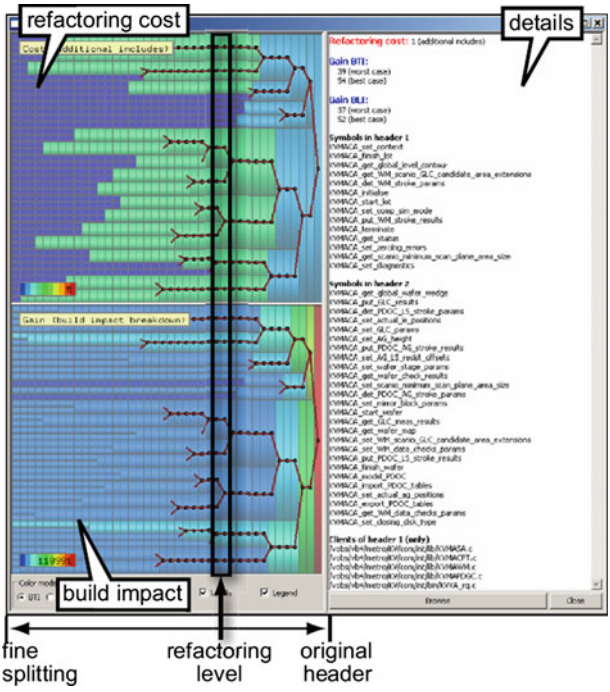

Fig. 7 Automatic refactoring view for the splitting of headers (see Sect. 5.6)

leave this step to the user rather than automating it, since the actual choice for which header to split typically depends on several non-quantifiable, project-specific aspects. Second, we extract all symbols declared by this header and locate which other files actually make use of these symbols, using our C parser (Telea and Voinea 2008). The header h is next split into two headers h_1 and h_2, so that to minimize the number of source files that use symbols from *both* h_1 and h_2. Recursively applying the above process yields a binary tree with h as root and each level l as a possible refactoring, or split, of h into several headers h_i^l. Including all headers from any level l instead of the original h in their respective client sources decreases build costs by decreasing the amount of included code and also the build impact. We call this the *refactoring benefit*. However, sources using large parts of our original header h must now include more headers after refactoring. As described in Sect. 5.2, including more files increases the build cost. We call this the refactoring cost.

The refactoring view has three windows (Fig. 7). The left windows show the refactoring tree. Each tree level l is a vertical strip, with the original header h at the right and the deepest splitting level $l_{\max}$ at the left. In practice, we use values for $l_{\max}$ in the range of 10–20, as finer levels would simply generate too fine-grained headers, which are not meaningful from a logical point of view. Each header h_i^l, i.e., each cell in the tree, is colored by the benefit and cost, respectively, using a blue-to-red colormap. Finding a good refactoring level amounts now to looking for headers having low refactoring cost, high build-impact parents and low build-impact children. The right view details

how the symbols will be split for the level chosen in the left window, thus providing additional detail to the developer.

It is important to stress that the automatic refactoring view is only a hint to the developer as to what refactoring possibilities exist. The actual decision for which header to refactor, and how finely to split it, should be taken involving several other aspects such as the logical coherence of the split headers. However, once the user has found a good refactoring level, the entire code base can be automatically processed to perform the refactoring.

6 Discussion

In this section, we present our insights in the utilization of our visual software analytics system in practice. Several observations can be made, which reflect on the more general topic of applying visual analytics in software maintenance. We structure the discussion along the software visual analytics aspects outlined in Sect. 4, preceded first by a general discussion of user feedback (Sect. 6.1).

6.1 User feedback

The visual build analysis and refactoring application presented in the previous section has been used in practice on an industrial embedded software C code base of about 17.5 million lines of code. Its design has been iteratively constructed and tested in several discussions with the architects of the system. The actual user experience can be divided in two phases: usage of the system during its iterative incremental design (2 months) and usage of the final tools presented here (2 months).

The system has been used by professional architects involved in the maintenance of the above-mentioned C code. Their overall goal is to minimize build times upon code changes, further refined as explained in Sect. 5. A full build of the code base takes around 9 h using a state-of-the-art so-called build farm, or cluster of Solaris machines compiling code in parallel using `gcc`. As the code is continuously modified by around 600 programmers worldwide, optimizing build times, or at least getting insight in build impacts, is desirable for planning development activities.

The users had previously tried to utilize CScout to get insight into the file dependencies. This attempt was already tried before our work started. The outcome was as follows:

- CScout is relatively good at extracting the file-to-file dependencies. This information can be used further to estimate build costs and impacts (Sect. 5.2). However, CScout does not report symbol-to-file relations accurately, due to its preprocessor and parser limitations. Hence, an accurate refactoring analysis like the one presented in Sect. 5.6 was not possible.
- The file-to-file dependency information was originally visualized as a graph (constructed using GraphViz), much like the dependency view in Fig. 5. This graph was useful in providing *local* insight on the dependencies of a file which was already known to create build problems. However, the entire build cost and build

impact analysis and visualizations were not present. As such, it was very hard for the users to actually detect build bottlenecks.

The cost/impact view and related architecture view (Fig. 5) were immediately appreciated by the users. For example, one user had informally maintained a collection of around 20 headers, which he inferred to be the 'main culprit' build bottlenecks from his actual experience with rebuilding the system. Within around 30 minutes of using our tool, this user could already find some high impact headers that he was unaware of (thus, not in his collection), and confirm that the headers in his collection were indeed among the highest build impact headers. The user stated that having a complete set of impact values for all system headers was very useful for planning modification activities.

The automated refactoring analysis (Sec: 5.6) supports a different use-case: When the architect decides that some header *must* be refactored, it provides the refactoring benefit, or build speed increase obtained, if a given header is refactored i.e., split into smaller headers. Using this tool, the architects detected that some high-impact headers can actually provide system-wide build speed increases up to 30%. However, they decided that those specific headers cannot be refactored due to other constraints, such as software packaging issues (product guidelines enforced that all interfaces related to some functionality should stay in one header). Yet, some lower-impact headers were found to be good refactoring candidates, and provided build speed increases of around 10%.

However, the main benefit reported by users of using our visual analysis tooling was not mainly in discovering 'quick fixes' that they could do to improve the overall build time of their system. During the evolution of such a complex software system, many issues come into play, such as user requirements, platform constraints, delivery dates, and integration with third-party libraries. All these determine architects to plan changes on the software stack. The main benefit for our tool reported by the users was to be able to correlate the planning of such changes with build impact implications, and thereby better plan future developments. Previously, such insight was lacking, i.e., it was not clear whether and how much would a software change that implies header dependency changes impact the build time. This caused significant effort to be lost in lengthy architecture meetings, as one would like to change the software but, in the same time, not increase the overall build time. Using the impact insight, decision-making was sped up as the architects were able to see (a) whether a change affects a high-impact component and (b) what actually causes that impact, thus they could decide whether the change was acceptable or not, or whether refactoring could accomodate the change and also keep the build impact low. Informal estimates of the gained time are in the range of tens of hours spread over several months, which is important given the high hourly tariff of system architects.

6.2 Visual analytics mapping

To better understand how our application is a visual software analytics instance, we outline below the mappings from our application steps to the general software visual analytics steps discussed in Sect. 4 (refer also to Fig. 1).

- *External data sources:* these are source and header files in the code base under build;
- *Shoebox:* contains a single tool, the C parser and fact extractor (Telea and Voinea 2008);
- *Schema:* contains the header dependency graph, the relations between declared and used symbols and files, and the build cost and build impact metrics computed on all these files;
- *Hypotheses:* the main hypothesis is that high build cost metrics, determined by high impact files, cause actual long build times. This hypothesis was empirically verified on our code base (Sect. 5.2). Subsequent hypotheses are that refactoring headers by splitting symbols based on usage patterns decrease build times. This was empirically verified by actually peforming such refactorings (Sect. 6.1);
- *Presentation:* this includes the architecture, dependency, and build cost views, and the automated refactoring view.

6.3 Highly structured text

The build analysis application revolves around three types of information: code structure, contained in the system architecture and build dependencies; code build metrics, and the actual code text. The different views used in the application reflect these data types. The architecture view shows the system's hierarchical structure and code metrics. The cost/impact view shows the various code metrics at file level. The dependency view shows the actual build dependencies and build metrics. Finally, the refactoring view shows the low-level code text (symbols), the proposed refactoring structure, and the cost metric. All views are correlated: a selection of an item in a view either highlights the same item in the other views, or updates them to show deeper-level information concerning the selected item. The different views allow exploring the system on different levels of detail, ranging from an individual symbol declaration (in the refactoring view) to an entire package (in the architecture view).

6.4 Precise semantics

Each software artifact shown in the build analysis tool has a precise semantics. Although some views show aggregate information, such as the total build cost on a subsystem in the architecture view, the meaning of these artifacts still relates one-to-one to *existing* elements in the code base. This design element has proven essential in the ease-of-understanding of the tool, as software engineers are more eager to grasp concepts which directly relate to those with which they concretely work on a daily basis: declarations, files, include relations, and subsystems. Moreover, the chosen cost metrics are all expressed in actual time units (seconds), following the build cost model (Sect. 5.2). This gives a very intuitive understanding of the analysis and can answer precise questions: What is the build cost of this item? What is the build cost of the system if I modify this item?

6.5 Constrained layouts

After a number of design iterations, we have chosen a simple visualization using a small number of elements. The overall view layout follows the traditional development environment which is well-understood by developers. All views use only 2D graphics, with a focus on simple metaphors, such as the table lens. The tree layout used in the automatic refactoring view (Sect. 5.6 is structured along a table: each tree level is a column, and each tree node is a rectangle colored by a cost metric. We superimposed an actual tree rendering over this tabular layout to further emphasize the hierarchical structure. We have observed that this way of rendering a tree was actually easier to understand than a classical pure node-link rendering, due to its space-filling properties. The dependency graph view (Sect. 5.5) is probably the most complex display in the entire system. We also noticed that this view tends to be less used than the simpler tabular cost distribution view.

The scalability of the dependency view depends of its actual use-case, as follows. First, if we use this view to show all headers included by a certain *client*, be it another header or source file, then the dependency graph size is directly dependent on the total number of headers included the client, i.e., the transitive closure of the *#include* relation. For all the C code bases we analyzed, within this project and several other large projects (see Telea and Voinea 2008), this closure is small, under 100 on the average, with outliers around 300–400 headers. Secondly, we can use the dependency view to show all targets depending on an actual header. In this case too, statistics on this code base and other C code bases indicate that the transitive closure of such a dependency graph has a few hundred nodes on the average. The graph visualization we use, based on the directed acyclic graph (DAG) layout provided by GraphViz (AT&T 2007), scales well for such graphs sizes.

Although the cost distribution view does not show actual dependencies, it serves its task well: the assessment of the build cost distribution of a given header over its sources. This underlies our principle of using minimal displays focused on the task at hand rather than overwhelming the user with too much information.

6.6 General remarks

The compact, dense-pixel design of the visualizations, together with the choice of a performant C parser, favors *scalability*. Our system can handle code bases of over 10 million lines of code. All queries can be performed *interactively*, by a small number of operations, essentially table sort and select only. All questions described in the problem statement can be addrssed with a small number of mouse clicks, making the tool *simple* to use. Moreover, the entire system is *integrated*: no additional manual data manipulation or complex tool composition is needed. Again, we noticed that this factor was essential for acceptance in an environment where developers need to get answers to their questions in minutes rather than hours.

The design of the entire analysis follows closely the visual analytics process. First, raw data is collected, allowing us to form a hypothesis on the nature of the build cost. Second, a cost model is constructed and validated with actual time measurements.

Third, a visual tool is constructed to enable more sophisticated analyses that go directly to the main questions the developers are interested in: what is the cost of rebuilding a system when a given file is changed, how are the costs spread over the system structure, and what can be done to reduce these costs. We remark that these questions, and the design of the views which address them, go from simple to complex, following the gradually deeper sensemaking process of visual analytics outlined in Fig. 1.

To our knowledge, the principles of visual analytics, albeit already established for a few years in the research community, have not yet been adapted and applied as such to software understanding applications. Although software data mining and visualization techniques exist, they are rarely tightly integrated in practice. Since visual analytics is a new field, its further application to the field of software understanding should be highly profitable both from the perspective of providing more insight to software practitioners in their source code, but also from the perspective of refining the general principles of visual analytics.

6.7 Future developments

During the actual usage of our visual build analytics tool, the users suggested several directions of future improvement, as follows:

1. so far, our tool is architect-centered. A useful extension would be the creation of a counterpart that would show the impact of a code change to the many *developers* who work in parallel. Ideally, this should take the form of a lightweight visual hint, like a pop-up, which would instantaneously display the build impact of a code change right before the user commits the changed code in the central code repository. Based on this information, users can decide if their changes create (un)acceptable build impacts in actual contexts, and proceed with or refrain from the commiting.
2. besides the actual build impact pattern of a code base, managers and architects are interested to see how this impact *evolves* in time over long periods. Detecting trends like steady increases in build costs is useful for estimating the future costs and speed of maintaining a given code base. As one architect put it: “We already know that our system builds slowly, but what we want is see what to do so that this does not get worse in the near future”.

Conceptually, both above types of enhancements could be done relatively easily with the information present so far from our code analysis. The difficulty lies in the actual creation of a distributed visualization and analysis tool that can be used by several hundreds of users in parallel on the same code base (1) and the efficient management of large amounts of information generated by successive build analyses (2). We plan to investigate both these issues in the near future.

7 Conclusions

In this paper, we have presented the application of visual analytics to the process of sofware understanding for maintenance. We outlined a number of similarities and

differences between the general visual analytics process and its application to software maintenance. Next, we presented an instance of a visual software analytics application to the build optimization of industry-size code bases, that combines software data mining and visualization in an integrated set of interactive views. This application combines a number of design principles such as tight integration between program analysis (data mining) and presentation, simple and dense-pixel 2D views, and a user interface which allows users to answer several questions by a minimal number of operations such as clicks and selections. Our application has been actually used by software architects on an industrial C code base of 17.5 million lines of code. The users reported being able to quickly find insight pertaining to the build bottlenecks which they were not aware of, and the added value of the automated refactoring scenarios proposed by our tool in speeding up the decision making process during their development planning.

We believe that visual software analytics, the application of visual analytics principles to software engineering activities, is a rich field with many perspectives, both theoretical and practical. We are currently working on several applications in software engineering, such as understanding large sets of testing data, exploring the evolution of software archives, and correlating software architectures with source code.

References

AT&T (2007) The GraphViz package. http://www.graphviz.org

Balanyi Z, Ferenc R (2003) Mining design patterns from C++ source code. In: Proceedings of ICSM, IEEE. pp 305–314

Baxter I, Pidgeon C, Mehlich M (2004) DMS: program transformations for practical scalable software evolution. In: Proceedings of ICSE, IEEE, pp 625–634

Belady LA, Lehman MM (1976) A model of large program development. IBM Syst J 15(3):225–252

Bell Labs (2007) The CScope code browser. http://cscope.sourceforge.net

Booch G (2006) On architecture. IEEE Softw 23(2):16–18

Corbi T (1999) Program understanding: challenge for the 1990s. IBM Syst J 28(2):294–306

Ctags Team (2007) Ctags home page. http://ctags.sourceforge.net

Diehl S (2007) Software visualization visualizing the structure, behaviour, and evolution of software. Springer, Berlin

Eick S, Steffen S, Sumner E (1992) Seesoft-a tool for visualizing line oriented software statistics. IEEE Trans Soft Eng 18(11):957–968

Elrad T, Aksit M, Kiczales G, Lieberherr K, Osher H (2003) Discussing aspects of AOP. Commun ACM 44(10):33–38

Gamma E, Helm R, Johnson R, Vlissides J (1995) Design Patterns: Elements of reusable object-oriented software. Addison-Wesley, Reading

Godfrey MW, Tu Q (2000) Evolution in open source software: a case study. In: Proceedings of international conference on software maintenance (ICSM), IEEE. pp 131–142

Havre S, Hetzler E, Whitney P, Nowell L (2002) ThemeRiver: visualizing thematic changes in large document collections. IEEE TVCG 8:9–20

Holt G (2007) Makepp home page. http://makepp.sourceforge.net

Klemola T, Rilling J (2000) Modelling comprehension processes in software development. In: Proceedings of international Conference on cognitive informatics (ICCI), IEEE, pp 329–337

Lanza M (2004) CodeCrawler—polymetric views in action. In: Proceedings of ASE, IEEE. pp 394–395

Lanza M, Marinescu R (2006) Object-oriented metrics in practice—using software metrics to characterize, evaluate, and improve the design of object-oriented systems. Springer, Berlin
Lienhardt A, Kuhn A, Greevy O (2007) Rapid prototyping of visualizations using mondrian. In: Proceedings of VISSOFT, IEEE. pp 67–70
Lin Y, Holt RC, Malton AJ (2003) Completeness of a fact extractor. In: Proceedings of WCRE, IEEE. pp 196–204
Littlefair T (2007) C and C++ code counter. http://sourceforge.net/projects/cccc
Lommerse G, Nossin F, Voinea L, Telea A (2005) The visual code navigator: an interactive toolset for source code investigation. In: Proceedings of infoVis, IEEE, pp 24–31
Mozilla Inc (2008) The Mozilla browser. http://www.mozilla.org
OMG (2008) The unified modeling language. http://www.uml.omg
Pigorsky TM (1996) Practical software maintenance. Wiley, London
PNNL (2008) The IN-SPIRE visualization system. http://in-spire.pnl.gov
Rao R, Card S (1994) The table lens: merging graphical and symbolic representations in an interactive focus+context visualization for tabular information. In: Proceedings of CHI, ACM, pp 222–230
Shneiderman B, Bederson B, Wattenberg M (2008) The Treemap 4.0 visualization system. http://www.cs.umd.edu/hcil/treemap
Spinellis D (2007) The CScout extractor. http://www.spinellis.gr
Standish TA (1984) An essay on software reuse. IEEE Trans Softw Eng 10(5):494–497
Telea A, Voinea L (2008) An interactive reverse-engineering environment for large-scale C++ code. In: Proceedings of ACM SOFTVIS, pp 67–76
Telea A, Maccari A, Riva C (2002) An open toolkit for prototyping reverse engineering visualizations. In: Proceedings of data visualization (VisSym), IEEE, pp 56–64
Teyseyre A, Campo M (2009) An overview of 3D software visualization. IEEE Trans Vis Comput Graph 15(8):87–105
Thomas J, Cook KA (2005) Illuminating the path: the research and development agenda for visual analytics. National Visualization and Analytics Center, IEEE CS Press, Richland
Tilley S, Wong K, Storey M, Müller H (1994) Programmable reverse engineering. Int J Softw Eng Knowl Eng 4(4):501–520
Wong PC, Thomas J (2004) Visual analytics. IEEE Comput Graph Appl 24(5):20–21