Chapter IX

An Open Architecture for Visual Reverse Engineering

Alexandru C. Telea, Eindhoven University of Technology, The Netherlands

ABSTRACT

Tool support for program understanding becomes increasingly important in the software evolution cycle, and it has become an integral part of managing systems evolution and maintenance. Using interactive visual tools for getting insight into large evolving legacy information systems has gained popularity. Although several such tools exist, few of them have the flexibility and retargetability needed for easy deployment outside the contexts they were initially built for. The lack of flexibility and limitations for customizability is a management as well as a technical problem in software evolution and maintenance. This chapter discusses the requirements of an open architecture for software visualization tools, implementation details of such an architecture, and examples using some specific software system analysis cases. The focus is primarily on reverse engineering, although the proposed tool architecture is equally applicable to forward engineering activities. This material serves the software architects and system managers as well as the tool designers.

INTRODUCTION

Businesses of many organizations heavily depend on effective maintenance of increasingly aging software. As software ages, the task of managing to maintain it becomes more complex and more expensive. Poor design, unstructured programming methods, and crisis-driven maintenance can contribute to poor code quality, which in turn affects understanding of the system properties. Program understanding (Tilley, 1998; Muller et al., 1993; Tilley et al., 1998) is a relatively young and evolving field concerned with identifying artifacts and their relationships and understanding their

structure and semantics. The essence of this process is essentially pattern matching at different abstraction levels. These levels induce in turn different representations of the candidate system. Overall, the aim is to aggregate these artifacts in a hierarchical representation in order to achieve a more refined and abstract understanding of the original system. Low-level representations, such as call or module dependency graphs, help the developers to grasp the system properties. More abstract and higher-level representations, such as simplified functional, task, or architectural diagrams may be used by the management to succinctly overview the status and evolution of a given software project.

Program understanding uses several information sources, such as direct source code examination, leveraging corporate knowledge, and computer-assisted methods. In this chapter, we focus on reverse engineering (RE) methods that address the process of understanding existing (large) software systems. However, note that the analysis and results presented in this work are also useful for the forward engineering activity.

Furthermore, we shall focus on computer-assisted RE methods, which have a number of important advantages. Firstly, they represent a deterministic representation of a software system, as compared to subjective interpretations. Secondly, they are used to analyze large systems, whereas direct source code examination fails for systems larger than approximately 50000 lines of code (Stasko et al., 1998). Thirdly, they require, in virtually all cases, less time to learn and apply. Finally, automated methods are the only ones applicable in the vast majority of the cases, given the size of the systems at hand. Managing the evolution of large software systems thus requires automated support for their understanding, which implies, at some point, the need for flexible RE tools.

Reverse engineering provides a conceptual framework for describing the process of software understanding and conceptual abstraction. This framework is supported by several RE tools. In the recent past, an impressive number of such RE tools has emerged. However, finding the "right" tool for a given application domain remains a challenging problem. This is mainly due to the fact that application systems vary from systems to systems, and thus may spawn different, often divergent requirements.

Given the above, practitioners in the RE field are left with two main choices: either pick one of the available RE tools and adapt it to one's specific data and requirements or create a new RE tool from scratch. In most cases, the solution of choice falls somewhere between the above two scenarios. If tool adaptation or design is required, it is thus of great importance for the RE practitioner to:

- understand the often subtle trade-offs the existing tools make in their implementation
- be able to predict the limitations before adopting a given tool
- avail a framework for designing a customized RE tool, in case adapting an existing one is too difficult for a particular application.

Overall, these often require a detailed analysis of the *architecture* of the RE tools. Based on such an analysis, the RE practitioner can compare different tools to a set of requirements, estimate the customizability of a tool of choice, or estimate the effort and way to design a custom RE tool. In absence of this analysis, tool evaluation is a time-consuming trial-and-error procedure that is not often feasible in most situations due to various constraints.

We continue our analysis by first noting that most RE tools provide two main features:

- *construction* of a layered program representation by automatic and user-driven (interactive) operations
- *visualization* of this representation in various ways, such as navigable graphical views at different levels of detail.

As mentioned, most RE tools differ in the way and extent they address the above two requirements. Some tools focus on program analysis and domain modelling, and thus on the program representation construction, but provide little for the visual examination and editing of the constructed representation. Other tools focus on data visualization but do not perform program analysis and are hard to integrate with tools that support this task. Overall, one may conclude that most existing RE tools are based on internal architectures that seriously limit the options for customization of several RE tasks such

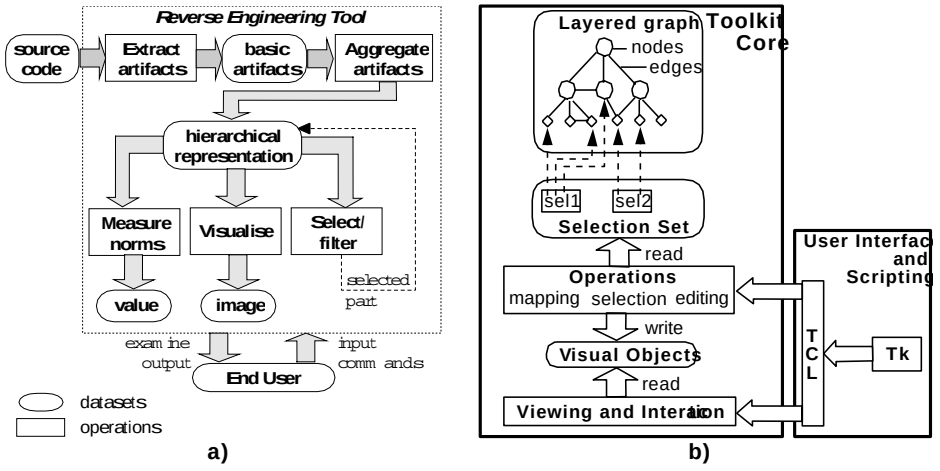
Several attempts have been made to design generic RE tools in the form of software frameworks allowing users to define and customize operations for their specific tasks. Ideally, such frameworks would minimize the time needed by the software engineer to adapt them to specific application requirements. However, the RE framework tools we are aware of are still too rigid to be easily reusable out of the context for which they were initially designed.

We propose here a software architecture for reverse engineering tools that tries to capture most of the concepts presented in the abstract RE framework. We next propose in detail how such an architecture can be implemented. Special attention is paid to the visual aspect of the reverse engineering process. Our first objective is to build a simple prototype of the RE data exploration scenarios by combining and customizing existing software components. We compare various aspects of our proposed architecture with existing RE tools and outline the differences. Finally, we present a number of RE applications in which we used the proposed architecture.

BACKGROUND

Several studies (Tilley, 1998; Telea et al., 2002; Riva et al., 2002) in the past have identified five major tasks that an RE tool should support. These tasks are defined at various abstraction levels of the hierarchy: *program analysis*, *plan recognition*, *concept assignment*, *redocumentation*, and *architecture recovery*. *Program analysis* is the basic task that any RE tool should support and consists of two services: *construction* of a layered program model and *presentation* of this model to the user, that is, via graphical navigable views at different levels (Eick & Wills, 1999; Stasko et al., 1998). *Plan recognition* aims at finding certain *design patterns* in the software (Gamma et al., 1995; Mendelzon & Sametinger, 1997). These design patterns form the so-called *domain model*, that is, the concept group describing a particular application field. A first attempt for plan recognition would be an editor for manual assignment of design patterns to elements obtained from program analysis and the visualization thereof, for example, UML diagrams. *Concept assignment* (Biggerstaff et al., 1994) is the task of discovering concepts and assigning them to their implementation counterparts. RE tools might

Figure 1: ***Reverse Engineering Pipeline (a). Toolkit Architecture Overview (b).***



support concept assignment by annotating the software artifacts with *concepts* retrieved from a domain-specific concept database and visualising this annotation. *Redocumentation* (Tilley et al., 1998) is the task of retroactively providing documentation for existing software systems. Since redocumentation spans three tasks discussed so far, an RE tool could support it by the mechanisms outlined so far. *Architecture recovery* (Wong et al., 1995) focuses on the recovery of architectural aspects of large systems.

The five mentioned RE tasks concur, and not compete, to the overall RE goal, that is, *extracting low-level code information* and *enriching it with information from other sources*. Since we are interested in RE tool support, we shall refine the above RE tasks into the following generic steps that an RE tool should implement (see Figure 1a) (Wong et al., 1995; Young, 1997; Wong, 1999):

1. *extract* the low-level artifacts from the source code.
2. *aggregate* the extracted artifacts into a hierarchical model.
3. *measure* the model's quality using computed norms; if needed, re-execute the aggregation differently.
4. *select* a sub-hierarchy to examine, if the whole is too large, complex, or unclear to display.
5. *visualize* the data, for example by producing a graph layout, followed by drawing the selected data (Telea et al., 2002).

In other words, for an RE tool to address the tasks mentioned previously, it has to implement the above five operations. Steps 2 to 5 can occur in any order – one may, for example, first visualize the whole model produced by Step 1, then apply some user- or system-driven aggregation (Step 2), measure the result's quality (Step 3), select a feature to look at (Step 4), and then repeat from Step 2. This matches the program understanding cognitive model (Young, 1997) that consists of alternate top-down and bottom-up

passes. Step 5 may provide different visualizations besides graph drawing. However, in most cases we are aware of, RE users desire to focus on the specific relations between (groups of) software components, and so graph visualization is their first choice.

THE NEED FOR INTEGRATION AND GENERICITY

Numerous papers address the conceptual problems of reverse engineering sketched in the previous sections. Ample material has been written over various RE tool implementations. However, it seems in practice that every attempt to reverse engineer a large system reaches some functional limitation of the existing RE tools. Concretely, such tools may fail at providing, or allowing customization of one (or several) of the RE pipeline steps described in the section "Background". For example, many tools emerging from the program analysis and formal method community fail at providing interactive means for visual program inspection. At the other extreme, there exist many tools providing extensive, sometimes exotic visualization metaphors for program data, but little in program analysis (Stasko et al., 1998). One reason for this situation is that building a good RE tool spans two traditionally different fields: software engineering and information visualization. Another reason is that information visualization, the discipline that analyzes how detail program information could be conveyed in abstract relational data via images, is a relatively new field. We believe that a successful RE tool should provide a flexible architecture encompassing all the five pipeline steps discussed previously, as well as a generic way to customize and extend these for particular domain models.

ARCHITECTURE PROPOSAL

We propose here a novel architecture for an RE tool that is consistent with the genericity and flexibility requirements detailed in the previous section. This architecture borrows some ideas from the scientific visualization systems community (Upson et al., 1989; Schroeder et al., 1995). In this sense, the proposed architecture consists of a number of *operations* performed on a number of *datasets*. To this model, we add specific operations and structure of the RE pipeline as outlined in the section "Background". Given the increasing importance of providing effective RE tools and the lack of detailed RE tool architecture presentation in the literature, we believe that this will help practitioners in the field needing to assess, develop, adapt, or extend RE tools.

The proposed architecture comes as a layered system consisting of a compiled core and an interpreted based user interface (UI) and scripting front-end, as shown in Figure 1b. Our implementation used C++ for the core and Tcl/Tk (Harrison & McLennan, 1997) for the UI and scripting. However, as detailed later, other implementations of the same architecture could be easily achieved. The core is responsible for the RE data and operation implementation, whereas the front-end provides customization and interaction mechanisms. Virtually all RE tools we are aware of follow this pattern. We describe the data and operation model implemented by the core in the next section.

Data Model

Our data model consists of two main elements: *a layered graph* and *selections*, as depicted in Figure 1b. The layered graph consists of structure and attributes.

Structure

Although many work on RE tools use different terminologies (Wong, 1999; Card et al., 1999; Stasko et al., 1998), virtually all models represent the basic RE data as a *hierarchical* (layered) *attributed graph*. In the graph's nodes model, software artifacts are created from program analysis, for example, source code parsing. In the graph's layers model, node aggregations (clusterings) are done during plan assignment throughout architecture recovery. In the edges model, both relational and containment information is used. In contrast to research work reported by others, we do not impose any restrictions on the graph topology, but rather it is determined by the user-driven RE aggregation process. In other words, we model all data as a graph whose nodes represent software artifacts (e.g., classes, files, methods, packages, tasks) and edges represent relationships (e.g., containment, calls, dependencies).

Attributes

Both nodes and edges may have key-value pair attributes. These represent both the acquired data, for example, number of lines of code of a module or number of bugs, or data derived during the RE process itself, for example, via software metrics. The keys are used as data identifiers. We implement keys as string literals and values as primitive types (integer, floating-point, pointer, or string). In particular, each node and edge has a set of attributes with distinct keys, managed in a hash-table-like fashion. Attributes automatically change type if written with a value of another type. Several *attribute plans* can thus coexist in the graph. An attribute plan is defined implicitly as all attributes of a given set of nodes/edges for a given key, for example, all values of the "number of bugs" key. Our attribute model differs from the one used by most RE applications (Eick & Wills, 1999; Wong, 1999; Kazman & Carriere, 1996) which choose a fixed set of attributes of fixed types for all nodes/edges. Our choice is more flexible, since: a) certain attributes may not be defined for all nodes; and b) attribute plans are frequently added and removed in a typical RE session. See section "Attribute Editing" for more details. Moreover, both memory and access time for attributes are kept low in this way, which is essential for coping with the graphs of tens of thousands of elements. In this sense, our model resembles the one used by the GVF (Marshall et al., 2001) and Visage (Kolojechich & Roth, 1997) tools.

Selections

Selections, defined as sets of nodes and edges, allow executing toolkit operations on a *specific subset* of the whole graph. To make the toolkit flexible, we decouple the subset

specification (*which* are the nodes and edges to work on) from the operations' definitions (*what* to do with the selected data), similarly to the dataset-algorithm decoupling in scientific visualization. Selections are named, and play the role of variables in a usual program. Our graphs are *structurally* equivalent to the node-and-cell dataset model in scientific visualization (SciViz) frameworks, whereas our selections do not have

a direct structural equivalent. Selections are *functionally* equivalent to SciViz datasets, since they are the operations' inputs and outputs. This is one of the main differences between SciViz and software visualization tools, which leads to different architectures for the two. In other words, our architecture is more data-centric than classical SciViz frameworks, as data elements are explicitly addressable via selections. However, our architecture is also operation-centric. More is available from Kolojechich and Roth (1997) for a comparison of the data and operation-centric models, since operations can be explicitly specified, as described in the following texts.

Selections have functional equivalents in some software visualization tools. In GVF (Marshall et al., 2001), they are represented by group nodes. In Rigi (Wong, 1999), they are implicitly represented by the slicing or filtering of operation output. See the section "Selection Operations" for more details. However, neither GVF nor Rigi has an explicit structure similar to selections. Visage's collections (Kolojechich & Roth, 1997) come closest to our selection concept, both structurally and functionally.

OPERATION MODEL

Operations have three types of inputs and outputs: *selections* that specify on which nodes and edges to operate; *attribute keys* that specify on which attribute plan(s) of the selection to work; and operation-specific *parameters*, such as thresholds or factors. We distinguish four operation types, based on their read/write data access:

- Selection operations create selection objects.
- Structure editing modifies the graph topology.
- Attribute editing modifies the graph attributes.
- Mapping maps the graph data to visual objects.

The above data-operation interface allows the core to automatically update all components that depend on the modified data after an operation's execution, using a simple Observer design pattern (Gamma et al., 1995). For example, the selections are automatically updated after a structure editing operation that deletes selected nodes or edges. Similarly, the data viewers (see the section "Visualization") are updated when the monitored selections change. This operation model based on observers monitoring a fixed number of operation types is a simplification of the more general idea of dataflow pipelines widely used by SciViz tools (Upson et al., 1989; Schroeder et al., 1995). The dataflow pipeline advantage is that it allows automatic update of more complex data dependencies. However, the practical experience suggests that constructing and maintaining an explicit dataflow pipeline is not a simple task for average non-programmer users. The simple structure of the RE pipeline as depicted in Figure 1a suggests our operation model serves the purpose.

Selection Operations

Selection operations add nodes and edges to selection objects. Several such operations can be implemented as follows. *Level selections* (called "horizontal slices" in the RE literature (Wong, 1999)) gather all nodes and association edges on a certain aggregation level in the layered graph, and are useful for visualizing the software at a

given level of detail. *Tree selections* (called "vertical slices" in Wong 1999) gather all nodes and containment edges reachable from nodes in an input selection, and are useful in for example, visualizing sub-system structures or change propagation (Marshall et al., 2001). *Conditional selections* (called "filters" in most RE papers) gather all elements in an input selection that obey some attribute-based condition, and are useful in queries such as "show all nodes where the cost attribute is higher than some threshold". Finally, *boolean selections* allow combining existing selections via intersection, union, and so forth, and are useful as part of more complex activities. As compared to many software visualization tools (e.g., Wong, 1999; Marshall et al., 2001), performing a horizontal or vertical slice or a filtering, in our case, does not alter the graph data – it just creates some new selection objects.

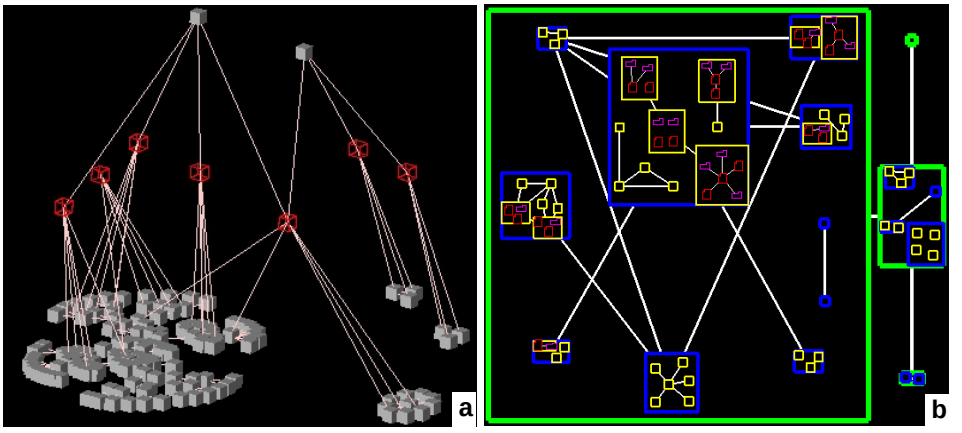
Structure Editing

Structure editing operations construct and modify the graph. Such operations include the standard node and edge addition and removal, as well as reading several graph formats such as file formats like RSF (Wong, 1999), GraphEd, DOT (Koutsoufios & North, 1996), and GXL (Marshall et al., 2001). Aggregation operations usually take the nodes in an input selection and produce a unique parent node. The input selection can either be programmatically constructed, such as automatic clustering methods, or can be the output of user interaction (section "Visualization").

Attribute Editing

Attribute editing operations take a selection as input and compute one or several attributes on the selection's nodes and/or edges (see the section "Attributes"). We found this system much more flexible than, for example, strong-typed designs, which associate a fixed set of typed attributes with a node or edge. Attribute editing operations can be further classified on their function, from a user perspective. In most software analysis scenarios, we have encountered two types of attribute editing: *metric computation* and *layout computation*. These are discussed later. However, note that this

Figure 2: ***Custom Layouts: 3D Stacked Layout (a) and 2D Nested Layout (b).***



classification is one of the many possibilities - other attribute editing operation classes may be created, depending on the specific application domain at hand.

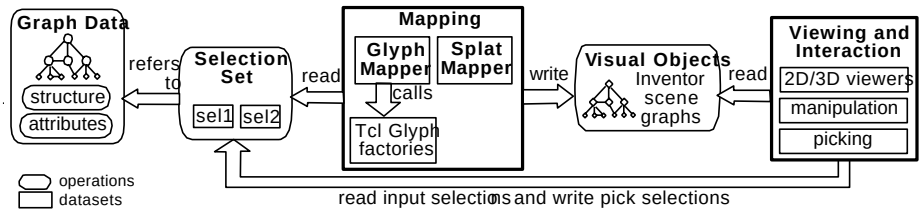
Metrics computation

We treat RE metrics as attribute editing operations. Examples of RE metrics are component coupling strength, the number of provisions, requirements, and internalizations (Wong, 1999; Tilley, 1998). Metrics may produce new attribute-plans, as the above metrics do, or single values, for example, a sub-graph's cyclomatic number or size. Decoupling the metric's selection input from the selection operation allows applying *any* metric on *any* sub-graph, which is not the case in other RE tools (Wong, 1999; Kazman & Carriere, 1996). Moreover, explicitly specifying the input and output attribute-plan names allows easy run-time prototyping of various combinations of metrics, similarly to the way one works with function or matrix objects in systems such as Matlab of Mathematica. Finally, the above decoupling allows the metrics, attributes, and selections to be coded independently in our architecture.

Layouts computation

In contrast to most RE systems (e.g., Wong, 1999; Kazman & Carriere, 1996; Marshall et al., 2001), we treat graph layouts simply as attribute editing operations and thus decouple them *completely* from mapping and visualization (see the section "Visualization"). This has several benefits. Firstly, we can layout different sub-graphs separately; for example, using spring embedders for call graphs and tree layouts for containment hierarchies (see Figure 2). Secondly, we can precompute several layouts, for example, to quickly switch between them. Finally, we can cascade different layouts on the same position attributes, for example, to apply a fish-eye distortion or refine an existing layout. We have implemented several custom layouts by cascading simpler ones, as follows. *Stacked layouts* (see Figure 2a) provide a selection spanning several layers of a graph by applying a given 2D layout (e.g., spring embedder) per layer and then stacking the layers in 3D. The layers are computed as horizontal slices (see the section "Selection Operations"). Stacked layouts visualize effectively both containment (vertical) and association (horizontal) relations in a software system. *Nested layouts* (see Figure 2b) provide a similar selection as above, by recursively laying out the contents of every node separately and then laying out the bounding boxes of the containing nodes. Nested layouts produce images similar to package UML diagrams and have proven to be very helpful in RE applications, as they are quite familiar to software engineers. Users can easily combine any 2D layouts as the building bricks for the stacked and nested layouts. In the example cited in Figure 2a, we use a tree layout, whereas in Figure 2b we use a spring embedder as basic layout. Adding new layouts to the toolkit is reasonably simple. The implementation of the generic spring embedder and tree layout we added to the framework of Koutsoufios and North (1996) exceeds 50,000 lines in C programming language. Adding them in a black-box fashion required less than 100 C++ lines each, whereas our custom layouts have each under 200 C++ lines. In this respect, our layout composition is very similar to the "power tool" design in Visage (Kolojechich & Roth, 1997).

*Figure 3**: Software Components of the Mapping Operation.***



VISUALIZATION

Mapping and visualization operations enable users to see and interact with the RE data. These operations have four sub-components: *mappers, viewers, glyph factories*, and *glyphs* (see Figure 3).

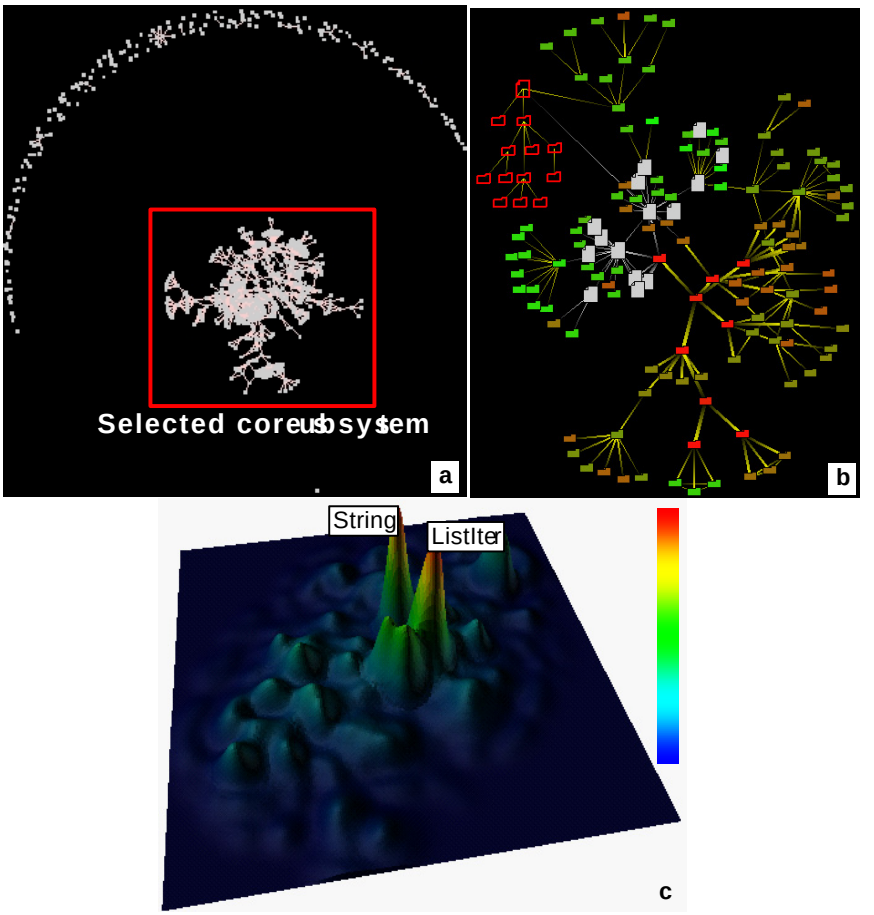
These operations are implemented using the Open Inventor C++ toolkit (Wernecke, 1993), which offers sophisticated mechanisms for object direct manipulation, picking, and rendering. If deemed necessary, the architecture allows using other similar toolkits, for example, Java 3D.

The central visualization component is the mapper, which creates 2D and 3D drawable representations out of the core graph data. We have implemented several mappers, as follows. The *glyph mapper* creates a glyph, that is, an iconic symbol, for each node and edge in the input selection, and positions these glyphs at the 2D or 3D coordinates provided by a node/edge attribute plane, computed by a layout operation (see the section "Layout Operations").

The *splat mapper* produces a height map, coloured by mapping the height attribute via a red-to-blue colormap. The height map shows the variation of node density per unit area, weighted by a given attribute, but does not draw the edges explicitly. This effectively visualizes large software structures exhibiting local node agglomerations, for example, corresponding to highly coupled clusters (see van Liere & de Leeuw, 2003 for details). For example, Figure 4c shows a height field weighted by the package provision metric. The String and ListIter implementation classes show up clearly, as they are used by most other system components. In the previous example, one sees that the system heavily depends on String and ListIter, so changes to these two components potentially have a strong impact on the system stability.

A *glyph* is a 2D or 3D graphical object that visualizes a node or edge. The glyph mapper calls, for every node and edge it maps, a Tcl script, called a *glyph factory*, which builds the desired glyph as an Inventor node. The script sets the glyph's graphical properties (color, shape, size, annotation, and so on) from the attributes of the input node or edge. Users may edit these scripts at run-time, so it is very easy to customize the visualization. Figure 4 shows a glyph-based visualization of the software of a program analysis system developed at Nokia. Figure 4a shows all 1200 software artifacts (methods, classes, packages, and files) extracted from the code. Figure 4b shows a simplified view of the system's core, after several graph-editing operations have been applied to filter the less important artifacts and cluster the remaining ones into higher-level units. Different glyphs have been used to show the different unit types, whereas the sub-

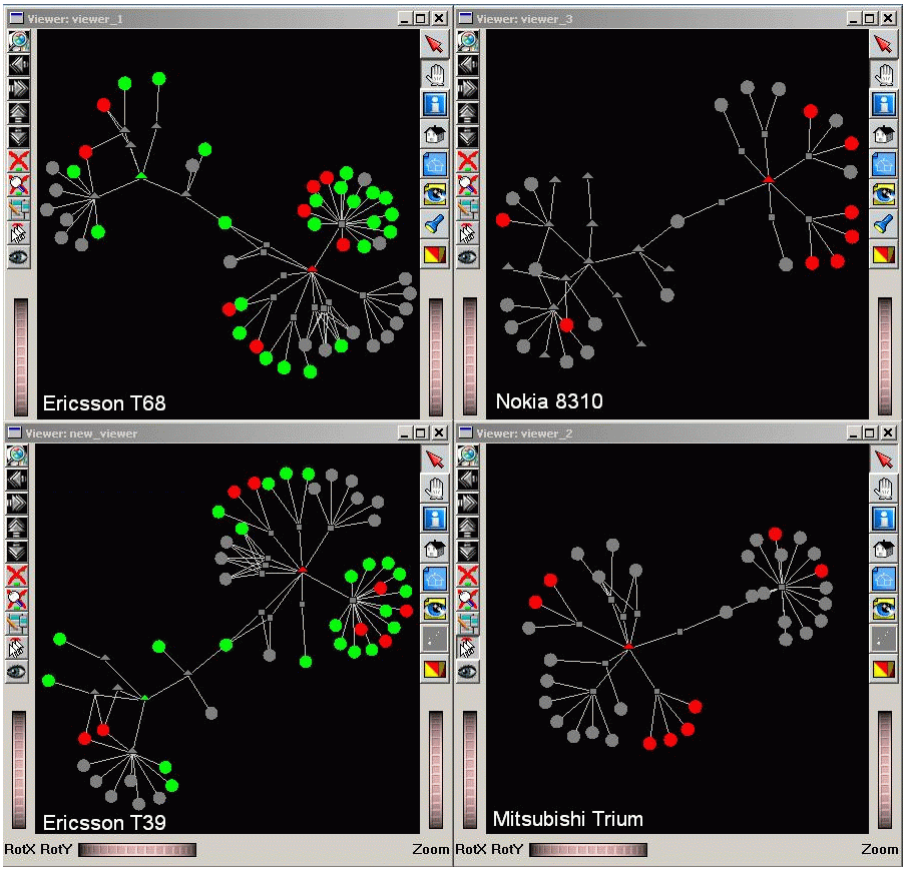
Figure 4: ***Visualization of Program Analysis Tool (a). Glyphs Showing Clustered Core Detail (b). Splat Mapper Visualizing Class Provisions (c).***



system coupling strength (Tilley, 1998) is visualized by edge glyphs of different thicknesses. The separation of the glyph placement done in the layout phase, and the glyph construction done in the mapping phase, is a simple but powerful way of specifying the visualization. New glyph factories can be developed without being concerned by the layout, whereas new layout tools can be added to operate on existing glyphs.

We show next a second example that illustrates the comparison of related software product families shown in Figure 5. For a detailed treatment on this, see Telea et al. (2003). Here, we show the RDF schema instances describing the UAProf notation (WAP Forum, 2000), a language for modeling device capabilities and user preferences for mobile phones. We used the glyph shapes to indicate various types of the UAProf language elements: Triangles are named resources, literals are circles, and rectangles are anonymous resources. The coloring scheme allows comparing different software products: nodes specific to a single instance are green; nodes shared by the two Ericsson products

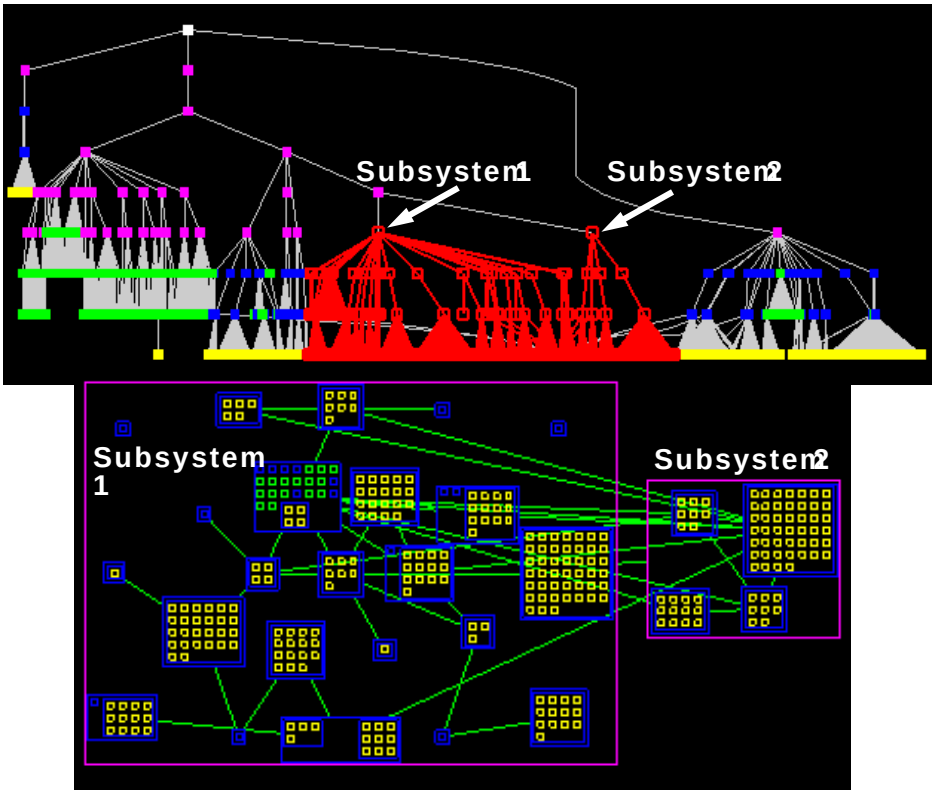
Figure 5: ***Visual Comparison of Four Mobile phone Specifications (RDF Schema Instances).***



are green; and nodes common to all instances are red. This simple visualization already allows one to see that all four products have roughly the same structure and identify the potentially important components shared by two, all, or just one product.

In a final example, we show a component-based architecture of a Nokia mobile phone software system, as shown in Figure 6. The top image shows the system containment tree, in which the user has selected two sub-system sub-trees, denoted by Sub-system 1 and Sub-system 2 (drawn in red). The bottom image shows, using a nested layout (see the section "Layout Operations"), the function call relations between the components contained in the two selected sub-systems. By interactively selecting different sub-system tree via clicking the tree root icons in the first view, architects get, in the second view, insight into the sub-system call interdependencies on each architectural layer. In the previous example, we immediately see which are the "interface" components, that is, the components through which Sub-systems 1 and 2 communicate. Secondly, we see that lower level components (the small, light colored boxes on the

Figure 6: ***Mobile phone Layered Architecture (top). Sub-system Call Dependencies for User-Selected Components (bottom).***



lowest containment level) do not make cross-system calls, a desired property of many architectures.

Concluding this section on operations, the properties of the selection, structure and attribute editing, and mapping operations discussed here make our operation concept quite similar to Visage's "primitive tools" (Kolojechich & Roth, 1997). The main difference between the two concepts is the way of classifying operations. We group operations *structurally*, depending on what data are read and/or written. Visage (and several other tools) groups operations *functionally* into primitive tools, power tools, and appliances. Both classification models have their own merits. Our proposed framework helps the tool designer implement related operations easily by sharing code and behavior. Visage's model helps the user find related operations in the same class.

EVALUATION

We have used the RE tool presented so far for reverse engineering and exploring several software systems. In all cases, parsing the data (source code or RDF schemas) to produce the attributed graph input data has been done by external parser tools that

the users already had. The input source code ranged between a few hundred lines up to 600,000 lines of code. Depending on the concrete application, RDF, C, C++, or Java code was parsed. Specific user interfaces (GUIs) were constructed in Tcl and added to the RE tool core described in this chapter in order to provide application-specific functionality atop the generic core. Such functionality included specific queries and filtering; for example, "show all system components programmed after a given date", or "eliminate all system components having less than 200 lines of code". Custom user interaction, such as constructing a nested layout of an interactively selected sub-graph shown in Figure 6, is also programmed via small Tcl scripts.

An important question is what the costs associated with adding visual reverse engineering support to an existing software workflow are. The above cost has three components:

1. learnin*g to use the visual RE tool*
2. adapting *the tool by writing custom Tcl scripts*
3. abstracting *the domain-specific questions into RE operations*

All three cost components are essentially determined by the *abstraction level* required by the end users. Low-level visual RE, for example, getting insight into the call or structure relationships of a software system, is the easiest to add. The main reason is that the end users, software developers in this case, are quite familiar with call graphs and thus they can easily adapt to the visual metaphors we use. Secondly, writing small Tcl scripts to customize the tool's operation, usually done by adapting existing examples, was quickly learned by end users different from the tool developers. So far, most Tcl-based custom operations we have seen written range from 20 to 150 Tcl script lines. Learning to use the tool took one to two hours. Adding custom operations, starting from the existing examples, took another one to two hours for programmers familiar with Tcl or similar scripting languages but not with our tool. All examples presented in this chapter fall into this category. When users were not familiar with RE and/or tool scripting, we took the path of first programming a "visualization scenario" (consisting of a custom Tcl scripting of queries, filtering, visual mapping, and interaction), and then letting the users use and/or modify this scenario. In all cases, users were able to reuse and modify these scenarios after a number of hours. Once this one-time cost was paid, users reported being able to apply a given visualization scenario to new datasets in a matter of minutes, more or less as reusing document templates in word processors.

The last cost component, abstracting domain-specific questions into (tool supported) RE operations, is in our experience the largest and most unpredictable cost component. Again, this heavily depends on the abstraction level of the application at hand. Low-level (code level) RE tasks easily map to the often one-to-one operations supported by our framework, as explained in the previous sections. Higher level questions, such as, "what is the impact on the system of removing a certain component?" or, "how do two architecture instances of the same system at different time instants or of two (similar) systems compare?", are much more difficult to support by an automatic tool. However, these were the typical questions we encountered when advocating the use of our tool to different end users from both academia and industry. Two statements should be made here. First, these questions can be supported by our (or a similar) RE visual framework, if one invests the effort to *translate* the questions into a number of:

a) familiar visual representations, and b) queries and filters that analyze the data at hand to produce a quantitative to answer the qualitative questions. As stated already, the visual representations we advocate here are easily learnt and understood even by people unexposed to information visualization metaphors. Designing *appropriate queries and filters for domain-specific questions* is, in our view, by far the costliest component of adopting (visual) reverse engineering. However, this is a generic, and so far unanswered, problem of automated program analysis.

Overall, the resulting end-user applications are functionally very similar to other RE tools such as Rigi (Wong, 1999), VANISH (Kazman & Carriere, 1996), Visage (Kolojechich & Roth, 1997) or relational data exploration tools such as Royere (Marshall et al., 2001). However, several differences must be mentioned. The main difference is our toolkit's core architecture, which is based on a few loosely coupled, orthogonal concepts: graph and selection data objects, operations, mappers, glyphs, and viewers. The data-operation with loose coupling via selections encourages developers to write small, independent operations. As stated, all our operations range from 20 to 150 C++ or Tcl lines. In contrast, Rigi (Wong, 1999) uses a monolithic core architecture. Although somewhat adaptable via Tcl scripts, this architecture offers no sub-classing or composition mechanisms *for the core itself*. It is not possible, for example, to change the graphic glyphs, the interactive selection policy, or to add a new mapper without recoding the core. Similarly, adding a new layout, selection operation, or metric involves a low level API to access nodes and edges, as Rigi has no notion of manipulating these as selections. VANISH (Kazman & Carriere, 1996) provides a way to build custom glyphs very similar to our glyph factories. However, VANISH uses node and edge attributes based on compiled C++ classes, which prove inflexible for our targeted RE scenarios. Visage (Kolojechich & Roth, 1997), used in a larger context than software visualization, is the system that shares the most architectural traits with our framework. The main limitation we could name for Visage is its limited customizability of the rendering (layouts and glyphs) it offers.

Finally, we briefly discuss the large class of library-level toolkits, such as GVF (Marshall et al., 2001), GTL, or Graphlet (Himsolt, 2000). These toolkits provide basic graph data manipulation and usually do not address visualization, interaction, and RE-specific operations together. From these, our toolkit resembles GVF the most. However, we found GVF's Java-based API rather complex to understand and use, especially for non object-oriented expert end users, which led us to our choice for a light Tcl customization layer to a C++ core.

CONCLUSIONS

The work reported in this chapter suggests that getting insight into large relational datasets is an intrinsic part of managing the evolution of complex software systems. Interactive, user-driven visualization tools come here as an indispensable aid. In this chapter, we have provided a top-down view of the path leading from the software comprehension goal and its requirements to the design of an open, flexible architecture for a software visualization tool and its applications. Reverse engineering was our main and most challenging application area, due to the size of the involved software systems, although we show examples from different fields engineering (RE) activity. Analyzing the tasks that RE provides, we distilled them into a set of operations and requirements that

any RE tool should provide, from program analysis to the final data visualization phase. Since an essential problem in applying RE to industrial software systems is the lack of flexible, generic RE tools, we have presented the architecture of such a tool implemented by ourselves and outlined the various aspects in which it performs better than other similar RE tools. We did not focus on the concrete implementation of an RE visual tool here for two main reasons. First, several such tools are already available. Second, what is obviously less treated in the literature is the description of what such tools are built upon and how to build or adapt them to the new, changing requirements imposed by system evolution.

The RE tool architecture presented here can be used in various ways. Firstly, it conveys insight into the various features and *flexibility* such tools should offer. This should help practitioners in the field judge whether existing tools present potential inflexibilities for a given task. Secondly, it discusses the *mechanisms* needed to obtain a certain degree of flexibility of a software visualization tool. This should help one judge whether an existing tool can be easily adapted to new requirements. Finally, given the novelty of the software visualization field, this can serve as a blueprint or starting point for implementing new tools that help in examining, understanding, and maintaining concrete software systems.

REFERENCES

Biggerstaff, T., Mittbrander, B., & Webster, D. (1994). The concept assignment in program understanding. *Communications of the ACM, 37*(5), 72-83.

Card, S., Mackinlay, J., & Shneiderman, B. (1999). *Readings in information visualization: Using vision to think.* Morgan Kaufmann Inc.

Eick, S., & Wills, G. (1999). *Navigating large networks with hierarchies.* In S. Card, J. Mackinlay & B. Shneiderman, (Eds.), *Readings in information visualization.* Morgan Kaufmann Inc.

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

Harrison, M., & McLennan, M. (1997). *Effective Tcl/Tk programming: Writing better programs with Tcl and Tk.* Reading, MA: Addison-Wesley.

Himsolt, M. (2000). Graphlet: Design and implementation of a graph editor. *Software: Practice and Experience, 30*(4), 1303-1324.

Kazman, R., & Carriere, J. (1995). Rapid prototyping of information visualizations using VANISH. *Proceedings of 1996 Symposium on Information Visualization (InfoVis '96)*. IEEE CS Press.

Kolojechich, J., & Roth, S.F. (1997). Information appliances and tools: Simplicity and power tradeoffs in the Visage exploration environment. *IEEE Compute Graphics & Applications, 17*(4), 32-41.

Koutsoufios, E., & North, S. (1996) *GraphViz, DOT and NEATO documentation* (technical report). AT&T Research Center. Available: *http://www.research.att.com*

Marshall, M.S., Herman, I., & Melançon, G. (2001). An object-oriented design for graph visualization. *Software: Practice and Experience, 31*(8), 739-756.

Mendelzon, A., & Sametinger, J. (1997). Reverse engineering by visualizing and querying. *Software – Concepts and Tools, 16*(4), 170-182.

Muller, H., Orgun, M., Tilley, S., & Uhl, J. (1993). A reverse engineering approach to subsystem structure identification. *Software Maintenance: Research and Practice, 5*(4), 181-204.

Riva, C., Maccari, A., & Telea, A. (2002). An open visualisation tool for reverse architecting. *International Workshop of Program Comprehension (IWPC 2002)*, 3-10. IEEE CS Press.

Schroeder, W., Martin, K., & Lorensen, B. (1995). *The visualization toolkit: An object-oriented approach to 3D graphics.* PrenticeHall.

Stasko, J., Domongue, J., Brown, M.H., & Price, B.A. (1998). *Software visualization: Programming as a multimedia experience.* MIT Press.

Telea, A., Frasincar, F., & Houben, G.J. (2003). Visualizing RDF(S)-based information. *7th International Conference on Information Visualization (IV'03)*, 294-299. IEEE CS Press.

Telea, A., Maccari, A., & Riva, C. (2002). An open toolkit for prototyping reverse engineering visualizations. *Data visualisation 2002*, Eurographics / IEEE TVCG Symposium Proceedings, The Eurographics Association, Aire-la-Ville, Switzerland, 241-250.

WAP Forum. (2000). *Wireless Application Group (WAP) forum: User agent profile*. Available: *http://www.wapforum.org*