Visualizing protected variations in evolving software designs

Sébastien Rufiange\textsuperscript{a}, Christopher P. Fuhrman\textsuperscript{a}

\textsuperscript{a}ÉTS, 1100 rue Notre-Dame Ouest, Montréal, Québec, H3C 1K3

Abstract

Identifying and tracking evolving software structures at a design level is a challenging task. Although there are ways to visualize this information statically, there is a need for methods that help analyzing the evolution of software design elements. In this paper, we present a new visual approach to identify variability zones in software designs and explore how they evolve over time. To verify the usefulness of our approach, we did a user study in which participants had to browse software histories and find visual patterns. Most participants were able to find interesting observations and found our approach intuitive and useful. We present a number of design aspects that were observed by participants and the authors using our IHVis tool on four open-source projects.

Keywords: software visualization, software design, information hiding, software evolution

1. Introduction

Changes in a software project happen for different reasons. There can be bug-fixes to respect quality attributes such as reliability, security or performance. There can be changes due to the uncertainty that is inherent to early phases of risk-driven iterative projects. There can be additional functional requirements. There could be a need to run the software on a different platform or on a different operating system. An important characteristic of an enduring software design is its ability to handle change over time.

Information hiding (Parnas, 1971) is a core principle in structured and object-oriented design. Designs that apply information hiding aim to hide parts of the software that are likely to change in order to reduce the impact of that potential change on other modules. Information hiding favors designs that have loose coupling to the elements that are potentially unstable.

There are different strategies to achieve information hiding. Encapsulation has been defined as “building a capsule, [...] a conceptual barrier around some collection of things” (Wirfs-Brock et al., 1990). Encapsulation implies that a designer explicitly specifies the boundary and what is visible to the outside. Programming environments typically offer mechanisms of access control of capsules to specify and enforce what is hidden and what is visible. A simple example is a Java class that has private members with public methods. Access-control encapsulation can also be used at the package level in Java, such that classes in one package are invisible to classes outside of that package.

A related idea is the open/closed principle (Meyer, 1988), which suggest that new features in a software should be implemented by extensions (e.g., adding new classes and methods to a subclass) rather than modifications, to reduce the impacts on client modules that depend on existing features. Similarly, the idea of protected variations (Cockburn, 1996; Larman, 2001, 2005) seeks to isolate what is change-prone (or unstable) behind an intentionally stable interface. Polymorphism or composition can then be used to define varying implementations while the clients only access the interface. Larman (2001) mentioned that the open/closed principle and protected variations are essentially equivalent to the more generic and fun-
damental information hiding principle. McConnell (2004) proposed the *iceberg metaphor* as shown in Figure 1. In terms of the open/closed principle, a module is said to be “closed to modification” yet “open to extension”. These relationships are always with respect to a client (or set of clients) that should not be affected by the extensions. This defines a frame of reference relative to client classes.

In practice, popular object-oriented design patterns (Gamma et al., 1994) strive to make designs more tolerant to changes. Many of these patterns make use of protected variations to protect respective client classes from extensions or modifications to the software. These dimensions of extension are intentional, with structures that use polymorphism (e.g., in Strategy and Observer) or composition (e.g., in Facade, Iterator and Proxy). However, despite the intentional dimensions for extension, these patterns are not explicit in their definitions about what is hidden or visible to the clients. Protected variations implies that clients only use the stable interface; the information hiding principle implies they should not know or see the extension classes. Ideally, a designer could specify this with access control. However, this is impractical in some environments with traditional access control mechanisms. Extension (hidden) classes could be in various different packages, scattered somewhat arbitrarily throughout a design. It is therefore challenging to determine the boundaries of the *capsule* (i.e., the iceberg).

### 1.1. Controlling access to unstable elements

Grand (2002) proposed the Interface pattern which is indeed a realization of protected variations. The pattern proposes a solution that keeps “client classes independent of specific data-and-service-providing classes” such that changes to the latter classes will not affect the clients. Figure 2 illustrates the pattern implemented in the context of a package, such that package access control prevents clients from seeing the implementation (Service) classes. This access control strategy works well provided all the implementations can be constrained within a package. Consider, however, an extension to this design where some NewService is added outside the package. There is no standard way of preventing clients from accessing this NewService directly.

Since strict access control beyond the class level is not commonly used by developers, the principles of “structure hiding” and “information restriction” are not always respected in practice. In fact, the *Law of Demeter* (LoD) essentially says that developers of client classes should be more conservative about using the information they can get from a class (Lieberherr et al., 1988). Also known as “Don’t talk to strangers,” the LoD could be thought of as follows: every class that a client class accesses could be a Facade. By not accessing the “strangers” behind the Facade, a client class is protecting itself from potential variations of those classes.

---

1or the client-facing class of any of the patterns that hide change-prone or complex structures, e.g., Proxy, Iterator, Factory.
Because traditional private-public access control mechanisms in languages are often not applied at architectural levels, there exist ad-hoc solutions. One such solution is the “Explicit Extension Rule” in Java Eclipse plug-ins, which is a convention to name hidden, change-prone packages as “internal.” (Gamma and Beck, 2003). The access control to these packages is not enforced by a compiler, but environments such as Eclipse give warnings or errors when code is built if these access control conventions are not followed. Similarly, manifests in OSGi bundles (Hall et al., 2011) can allow defining modular components with access control enforced within the Eclipse Equinox reference implementation.

Some language-specific encapsulation mechanisms exist beyond the class and package level. The “internal” access modifier in C# (Microsoft, 2012) limits visibility of members to files in the same assembly, and there is a planned extension to Java to support modules (Reinhold, 2011). However, even with a broader scope of access control, it remains a challenge to assure that the type of extension proposed outside the scope of an arbitrary “module”, as depicted by the NewService class in Figure 2, is visible to the Client. The problem stems from the fact that modules are specified in a top-down way, rather than being identified dynamically based on coupling.

1.2. Tracking protected variations over time

Differences in the software development process affect the way that designs evolve, and in turn how encapsulation might be specified during the process. As stated by Parnas (1971), it is the designer who decides where the boundary is between what is hidden and what is private. This is traditionally a top-down strategy of design, because important decisions are made before client programmers are exposed to the programming interfaces. At the class and package level, things are straightforward: a designer proposes a software class and specifies which parts will be visible to the outside and which parts will be hidden. However, designs involving a lot of composition of reusable libraries are said to be emergent (McConnell, 2004). It may not be clear at the higher levels where to apply information hiding; if it is applied too conservatively, it might restrict freedom in bottom-up design. Furthermore, design patterns are often applied in groups (Buschmann et al., 1996). The unstable elements of one pattern might be clients of stable elements of other patterns.

Therefore, specifying access control at the architectural level is arguably more challenging. Again, some solutions exist: the Layers pattern and the Model-View-Controller pattern (Reenskaug, 1979) are both examples of where there is a convention of access control. That is, certain lower-level layers (i.e., the Model) should not be coupled to (“see”) upper-level layers (i.e., the View).

Keeping track of coupling over the evolution of a project is an important concern for software architects, but so is the tracking of the dimensions of variability. Once a design with protected variations is specified (regardless of access control conventions), several questions remain: (1) does the protected-variations dimension of the design serve a purpose (are new variations implemented)? (2) do the “stable” parts get re-used by more and more clients (are they “popular” or “useful” abstractions)? and (3) are the coupling conventions respected (are the clients decoupled from the change-prone parts “below the water” of the iceberg metaphor)?

As software grows in complexity, the changes to it become more difficult and can cause architectural erosion and drift (Perry and Wolf, 1992). In terms of protected variations, this implies that as software is added, the protected variations structures could no longer be valid. An example (Figure 2) of this theoretical drift is that a new class is added which is both a client to the stable part of the protected variations but for some reason is also coupled to its extensions.

1.3. Visualizing changes in software designs

Interactive visualization techniques and visual analytics can help users discover insights in (possibly dynamic) data, such as unexpected patterns, anomalies or relationships (Thomas and Cook, 2006). Because of software’s intangible nature and its tendency to be complex, graphic visualizations have been proposed to better understand software design. The dimensions of software designs, which are changing over time, are also invisible, except for when they are visualized in a tool such as ours. By showing these dimensions to developers, they can better understand the design of software in terms of its extendability. Automated approaches may be difficult to use in software engineering, since we do
not know exactly all factors that could influence the quality of a design over time. However, visualizations can represent software data in a more convenient way that facilitates decision making and analysis (Langelier et al., 2008). Software visualization can cover three main aspects: structure, behavior and evolution (Diehl, 2007). Visualization of evolution is essentially doing a visual analysis of software histories, of which there are three kinds: visualization of metrics, visualization of structural information, and discovery of recurring patterns from the software history using data-mining and visual data-mining techniques. Our approach combines aspects of all of these kinds to analyze the evolution of structures, particularly the structures associated with micro-architectural protected variations, where it is more difficult to specify and enforce encapsulation.

Despite a good visualization, a software project’s complexity can still overwhelm the observer. Many visualization techniques tend to display the entire structure of the software, rather than the perspectives derived from encapsulation (i.e., the iceberg metaphor doesn’t apply). By displaying only coupling related to protected variations, for example, a tool could potentially help a developer focus on those aspects, without being distracted by coupling that is unrelated. Visual tools have the potential to help discover unknown interesting patterns by engaging a user in an interactive exploration of a software. To achieve this, the “details on demand” principle in information visualization (Shneiderman, 1996) can be applied, to allow the user to get an overview first, and filter the data based on some classes, types of object, and instances of particular interest or evolving in certain ways.

We propose in this article a visualization tool, applied in the software engineering domain, that presents the elements of a software design to the developer in terms of the frames of reference established by the open/closed principle. To account for evolution in the design, our interface allows visualizing multiple versions of a software in a Subversion repository. We illustrate how our novel approach can help to (1) find periods of interest in a software change history, (2) find design structures in real projects and (3) track and characterize evolving design structures.

The rest of this article is organized as follows. Section 2 presents the variability concepts on which our visualization tool is designed. Section 3 presents the visualization tool itself, and then authors’ observations regarding protected variations in open-source projects. Section 4 presents the task-based user study that was done to show the usefulness of the tool in the software design community. Section 5 discusses all of our results and limitations before Section 6 explores similar work, comparing and contrasting it with this article’s contributions. Finally, Section 7 draws conclusions and discusses ideas for future work.

2. Variability Zones

This section describes the key concept of variability zones as well as how they evolve over time in a software project. As mentioned in the introduction, there are several mechanisms commonly used to achieve protected variations. One popular strategy in the Java language is the Interface pattern (Grand, 2002), and it illustrates well the elements of the variability zone concept that are fundamental to our approach.

Assuming that a Java interface is an information hiding mechanism that can be used to hide variations from some external clients, protected variations can only be achieved if the clients are coupled only to the interface and are decoupled from concrete implementations. As shown in Figure 2, a static package (or arbitrary capsule) could be insufficient to capture the future extensions of implementations of a service, since a developer could create a NewService implementing the public IndirectionIF interface.

A stability point (mapped to a Java interface) thus defines a variability zone, which is the set of elements (e.g., abstract or concrete classes) implementing or extending the interface (e.g., derived interfaces), as illustrated in Figure 3. This definition can also cover one or several levels, to include the concrete elements implementing derived interfaces of the stability point. At level 0, the variability zone only includes the concrete elements (i.e., implementations and extensions) of the stability point’s interface itself. At level 1, the variability zone also covers the concrete elements of the implementations and extensions of the stability point’s interface, and so on.

Variability zones also have properties that we can collect over time. N refers to the actual number of elements inside a variability zone (e.g., classes), and these are extending a stability point in some way. We also refer to the distinct clients coupled to
the stability point as “stable couplings” (counted by S). Alternatively, instances where clients depend on concrete implementations defined inside a variability zone represent “unstable couplings” (counted by U). Unstable couplings implies that clients are not protected from changes inside the variability zone. Property N is similar but more specific than the NOC (number of children) metric from (Chidamber and Kemerer, 1994). In theory, N is the number of children that implement a stability point, where each child is part of the hierarchy of the stability point. In practice, the depth of the hierarchy for this counting (i.e., the number of levels to include in a variability zone) can be changed in the counting algorithm.

2.1. Evolution of Variability Zones

Variability zones can evolve over time in different ways during the history of a software project. Figure 4 illustrates several hypothetical possibilities considered in our approach, and how they would be observed in terms of structure and the properties of N, S and U.

The fluctuations of a variability zone can also be viewed graphically in terms of the N, S and U counts. They can be interpreted in the context of the information hiding principle. For example, if the boundaries of a variability zone increase significantly while the level of unstable relationships remains constant, it suggests that the instabilities of this part of the design do not increase around the stability point. Indeed, no new external clients are coupled (for the period of time analyzed) to the concrete implementations that were added. Figure 5 illustrates a few other possible cases in terms of the evolution of a variability zone. We now describe an hypothetical but realistic example of variability zone evolution illustrated in Figure 5. At step 1, new variations are being added, and thus N increases. However, S remains low, which suggest that the stability point is not coupled to a lot of other elements (i.e., low degree of usage). At step 2, an increasing number of elements refer to the stability point directly. One potentially interesting observation is the fact that this increase in usage does not lead to an increase in unstable coupling in this case, which can be a good sign initially. At step 3, although the number of variations decreases, an increasing number of clients know their concrete implementations and should be verified to check for opportunities to reduce U (e.g., introducing a Factory could help decrease it if the clients are instantiating the implementations themselves). Finally, at step 4, several new variations are being added, and more couplings are toward the stability point interface than toward the variability zone (i.e., S is higher than U).

Views in Figures 4 and 5 complement each other. The first one (Figure 4) is a more detailed view, and depicts both the evolving software design, its structural changes over time (e.g., modules or links added or removed), and variability zones (e.g., added or changed). The second view (Figure 5) focuses on showing the evolution of one or several variability zones, in terms of their properties.

The diagram presented in Figure 5 is useful to track evolving variability in a software design because it makes it easier (1) to find which variability zones are evolving in a software history, (2) to locate the time periods where properties increase or decrease. For example, for a specific stability point, it is possible to see whether it is being used by many clients (a greater value of S implies a popular abstraction), if it has been extended over time (an increase of N signifies a useful variability zone), or
if it has only one implementation (a value of N=1 implies a design of questionable usefulness).

The concepts we presented in this section inspired analytical tasks we asked participants to perform using our prototype.

3. Visualization Tool

IHVis allows users to examine source code repositories of Java projects to find and analyze variability zones. To facilitate the exploration of variability in evolving software, our prototype includes features and interaction techniques to visualize data using multiple views. In this section, we will first explain how data is mapped to software designs (modeled as graphs, or networks), followed by an overview of features, and examples of practical uses of the tool.

3.1. Modeling source code histories as dynamic graphs

IHVis reads the change history of source code in repositories (such as Subversion) that include a list of transactions in which files are modified by developers. A network representation can then be built for each transaction (numbered with a unique revision number) by parsing changed files, as shown in Figure 6.

The parsing process allows our prototype to detect changes at different levels of abstraction. Coarse-grained changes (such as classes or interfaces added, modified or removed), and lower level changes (such as modifications of attributes, method signatures or code implementations) are thus collected for each revision. A network representation can naturally represent this software data in terms of nodes and edges. The nodes include classes, interfaces and abstract classes (all of which can have methods), while edges represent the couplings, inheritance and interface extensions. Furthermore, following our definition of variability zones (see Section 2), the implementations and extensions of a stability point (a Java interface), form network clusters.

A typical example is illustrated in Figure 6. At revision 2, classes are added (FileParser.java and CSVParser.java), while others are modified (FileOpener.java) or removed (FileStub.java). The interface (FileParser) is a new stability point to encapsulate the reading and writing of files, and CSVParser is the first element part of its variability zone at this point in time, created to handle CSV files (Comma-Separated Values). Also, the class FileOpener is modified to use this new stability point to read files. The result of the data collection process, repeated for all revisions of a software, is a dynamic network model that can be visualized (and whose layout can be modified by the user) in IHVis.

3.2. Visualization prototype

IHVis shows the software elements or nodes (interfaces, classes) and the types of relationships (extends, implements or imports) using a UML-like notation shown in Figure 7. This representation is used in several of the figures presented in this paper. Variability zones are shown as blue dashed areas associated with a respective stability point. The
Figure 6: Illustration of the data collection and visualization processes in IHVis. Source code repositories store copies of the files as they are changing over time. In a revision, files can be added ([a]), modified ([m]) or removed ([r]). Based on these changes, our tool builds networks to visualize the evolving software design.
dashed lines are easier to identify in complex visualizations. Since standard UML contains dashed lines (e.g., to indicate interface implementation), they detract from the pre-attentive processing benefit for identifying variability zones. This is the main reason for a notation that is different from UML. Also, when a node (or a cluster of nodes) is hovered, its neighbors are highlighted. This is useful to see more clearly the nodes that are coupled (stably or unstably) to a variability zone.

We also illustrate in Figure 8 how our prototype depicts changes between revisions. The initial network representation of the source code at revision 2 is shown in Figure 8a, and is composed of a FileParser stability point, one concrete implementation (CSVParser), and one stable client (FileOpener). At revision 3 (see Figure 8b), an application class is added (XMLViewer), which makes use of a new concrete strategy (XMLParser) to parse XML files. New nodes are shown as fading in to green. The size of the variability zone thus increases, and new important links to XMLParser and CSVParser are shown as fading in to their final color (red). In the same revision, a method in FileOpener is removed (shown in dark red), while a method in CSVParser is modified (colored in orange). When a method is changed, a small green square appears to indicate the nature of the change (e.g., method signatures, access modifiers or code

Figure 7: Notation within IHVis’ structural view. Couplings toward a stability point or its associated variability zone are red-filled (a). Common couplings (i.e., import and dependency couplings) are also shown in red, but not using filled color (b). Interface implementation relationships are shown in blue (c), while inheritance couplings are colored in green (d). Also, elements in focus are colored in magenta, and hovered elements are color-filled.

Figure 8: Schematic illustration of how evolving software designs are visualized in IHVis. New nodes are colored in green, while modified and removed nodes are shown in orange and red, respectively. Colors of links and nodes fade in or out over time using an animation, but at any given point it’s possible to stop the animation to examine a frame. An element that has been removed within a span of revisions shows as fading out (it becomes more transparent), whereas an element that has been added starts out as transparent and becomes more opaque.
implementations). In the next revision (Figure 8c), a Factory is introduced, to reduce unstable couplings toward the variability zone. Now only the FileParserFactory class links to XMLParser and CSVParser classes, in addition to being coupled to the FileParser stable interface (these added edges are shown as fading in). The other links (e.g., between XMLViewer and XMLParser, and between FileOpener and CSVParser) are removed (and thus, shown as fading out).

The workspace of IHVis is composed of multiple views, each focusing on a different perspective of evolving stability in designs, as shown in Figure 9. A video demonstrating the tool is also available online.

The user can interact with two views that are displayed at the same time: a zoomable network view (Figure 9A,B) and a time-line view, showing either the evolution of variability zone properties using line charts (Figure 9C) or the evolution of stability and project metrics using bar charts (Figure 9D).

To simplify visualizing complex projects with lots of classes, our tool supports panning and zooming in all views as well as highlighting and filtering of classes. Classes can be filtered by name or object type using the options in the upper right part of the interface, and by making rectangular or manual selections in the network view. Users can browse the list of stability points that were found using the filters, and focus on only one of them as desired. If a filter is applied, only the elements in focus and their related elements (neighbors) are displayed. A blank workspace is created when a user imports a new dataset from a source code repository. In this case, node positions are computed using a force-directed layout (Fruchterman and Reingold, 1991), for each time step of the dynamic graph. However, graphical elements can be freely moved around by dragging, and workspace sessions can also be saved and restored later on. To help preserve familiarity with UML, known to most software engineers, child elements (such as internal classes or methods of classes) are shown using nested enclosures and indentation in the node-link view.

The network view can be used to show the software design at a specific revision (i.e., a moment in time of the project’s history) or to compare multiple revisions. If a single revision is selected, the data are visualized using a small multiple. If several time steps are selected, their differences are highlighted, and animated transitions can be used to show the changes. The visualization of these changes may help reveal observations about a design (such as the removal or addition of software elements and couplings, the nature of the changes and the affected variability zones). Also, in a zoomed-in view, the user can see the details of the methods of the visible classes and interfaces including access-control information (visibility).

Although the network-based visualizations are helpful to show evolving fine-grained changes in software designs, other support is needed to help the user locate interesting revisions in a project with a long history. To facilitate this higher-level exploration of change histories, IHVis implements time-line views (shown at the bottom of the workspace). The selection of time steps is done by interacting with the time-line view. The user can scroll forward (or backward) in time by clicking the mouse. Users can visualize a specific revision shown in the time-line view by clicking on it, or a range of revisions can be selected by holding shift and choosing the starting and ending time steps. When a range of revisions has been selected, a user can start an animation of the changes over the range of revisions. The user may control the speed at which the animated transitions are shown by scrolling in time manually using a scrubbing technique or automatically by pressing a play/pause button.

This interactive and filterable time-line view has two possible configurations. First, it can display line charts (Figure 9C), showing the evolution of variability zone properties. A user can track fluctuations in terms of size of variability zones (N) or its stable and unstable couplings (S and U, respectively). A second possible visualization for the time-line view is bar charts, that display the evolution of project metrics (Figure 9D). Bar charts thus displays other information about the changing designs over time, such as Lines of Code (LOC), number of changed classes, and the count of occurrences of certain keywords in commit messages (that might indicate important revisions such as bug fixes or refactorings), for specific revisions. These views support the user in browsing (possibly large) change histories and looking for potentially interesting indications of changes in stability.
Figure 9: IHVis displays a network view (e.g., A or B), along with a timeline view (e.g., C or D) in a single workspace. 

A. Force-directed layout view of a specific revision of a software design. The user can zoom and pan around the view to examine relationships between elements, and reposition elements. 

B. Zoomed-in view showing more detail. 

C. Visualization of the evolution of the variability zone properties (internal size, stable and unstable couplings). 

D. Bar charts showing metrics of revisions (e.g., LOC, classes modified, keywords found in commit messages).

Table 1: Projects that were explored using our approach. JHotDraw and Buddi were also part of the user study.

<table>
<thead>
<tr>
<th>Project</th>
<th># Rev.</th>
<th># Files</th>
<th>Size</th>
<th>Dates</th>
</tr>
</thead>
<tbody>
<tr>
<td>JHotDraw</td>
<td>781</td>
<td>484</td>
<td>3.1 MB</td>
<td>2000-2013</td>
</tr>
<tr>
<td>Buddi</td>
<td>1234</td>
<td>253</td>
<td>1.6 MB</td>
<td>2006-2013</td>
</tr>
<tr>
<td>JabRef</td>
<td>3746</td>
<td>581</td>
<td>5.8 MB</td>
<td>2003-2013</td>
</tr>
<tr>
<td>Violet</td>
<td>284</td>
<td>290</td>
<td>1.9 MB</td>
<td>2010-2013</td>
</tr>
</tbody>
</table>
3.3. Stability Points in the Wild

This section presents results of noteworthy observations made by the authors, using IHVis on different open-source projects mentioned in Table 1. JabRef (http://jabref.sf.net) is a bibliography reference manager and BibTeX editor, and Violet (http://violet.sf.net) is an application for drawing UML diagrams, whose architecture is documented in (Chansler et al., 2011). JHotDraw (http://www.jhotdraw.org) is a project that uses a large number of design patterns and, as such, can be seen as both a graph-drawing framework and application. Buddi (http://buddi.digitalcave.ca), on the other hand, is a less complex project to manage personal finances and budget of households. The focus here is on the early iterations of the software projects, with the assumption that iterative designs tend to be less stable at early phases (Jacobson et al., 1999).

3.3.1. Buddi

Using our tool, we discovered that there were several interfaces with a small number of extensions (e.g., ModelFactoryImpl, SourceImpl, TransactionImpl) across the Buddi change history, as shown in Figure 10.

At first, we could conclude that it might be a case of adding unnecessary dimensions of extensibility or over-designing. However, upon further investigation in the source code and using the time-line view, it appears that some of these implementation classes are in fact models in a data-model framework, as they are extending the Eclipse Modeling Framework classes (e.g., org.eclipse.emf.ecore.impl.EFactoryImpl and org.eclipse.emf.ecore.impl.EObjectImpl).

At some point after revision 225, the usage of several stability points (e.g., Transaction, Category, PrefsPackage) does not change for a long period of time, suggesting the stabilization (or maturity) of a part of the design (see Figure 11A). In another

Figure 11: Illustration of IHVis with the Buddi project. A. The stable coupling of the Transaction stability point, among others, remains constant after revision 225. B. Replacing a class (JTextFieldHint) with a new one (JHintAutoCompleteTextField) in the AutoComplete variability zone at revisions 111 (on the left) and 112 (on the right).

view, the AutoComplete variability zone was modified around revisions 108 and 112 in order to (citing the revision comment) “rearrange internal text component classes and move Hint code from the container to the components”. In fact, Figure 11B shows this small refactoring visually, as the developers introduced a new class (JHintAutoCompleteTextField) to handle the auto-completion of text fields that replaces the older class JTextFieldHint.

Another example of a design improvement is with the BuddiReportPlugin (revisions 261-265) in which the developers worked on the “dynamic loading of reports, ...custom reports options ...within the new plug-in architecture” (Figure 12). Before the introduction of that stability point, the code to generate reports was located inside ReportFrameLayout. However, as a new kind of report is added...
(IncomeExpenseReportByDescription), the code to actually perform the function (e.g., getTreeCell) is moved to nested classes. A possible consequence of this refactoring is that external clients have less access to nested classes.

Figure 12: A new stability point, BuddiReportPlugin, is added to the Buddi project at revision 262. The changes between revisions 262 and 266 are shown using an animation with IHVis, which is partially evident in this single image where methods are in green or red. The variability zone, increasing in size, is also shown as fading in to orange. Implementations (methods) defined by the interface (A) are moved from the ReportFrameLayout class (B) into new classes (C), to support a new plug-in architecture.

3.3.2. JHotDraw

In the JHotDraw project, a Painter describes different strategies to draw a scene, either by simply looping through all objects or using some caching mechanism to only redraw drawing areas if required. Since drawing views are directly coupled to the concrete update strategies, the unstable coupling fluctuates accordingly (see Figure 13A). In addition, a few application classes (e.g., DrawApplet) also contribute to the unstable coupling. Of course, the introduction of a Factory could have reduced the degree of unstable couplings and prevent possible future unwanted relationships, although the complexity seems fairly manageable at this point in time. By examining the bar charts of the various revisions in JHotDraw (not shown here), we identified that new update algorithms, such as ways to draw scaled objects (e.g., ZoomUpdateStrategy), were added gradually over time (Figure 13B).

Similarly, a Layouter’s implementor defines how to position objects in space (e.g., SimpleLayouter, HTMLLayouter). At revision 45 the developers...
refactor and migrate code from StandardLayouter to SimpleLayouter, and reverse the “extends” relationship (see Figure 14). Also, at revision 149, there is a significant increase in unstable couplings, which is essentially caused by the addition of test cases (e.g., SimpleLayouterTest) (see Figure 13C). Interestingly, further observations in the source code suggest that the designer seems to use a top-down methodology in this case. In addition to the use of test cases, the code for HTMLLayouter, for example, remains defined only at a higher level and is left not fully implemented for some period of time.

3.3.3. JabRef

The PrefsTab interface in the JabRef project has an increasing number of concrete implementors (starting from revisions 31, shown in Figure 15B) which define types of Tab (e.g., GeneralTab, ExternalProgramsTab, etc.) in the user interface (see Figure 15A). A closer look at the time-line view shows that the number of unstable couplings matches the size of the variability zone (the corresponding lines are drawn on top of each other), since the PrefsDialog2 client class creates all the elements inside the variability zone (see Figure 15B).

Figure 14: Refactorings in the JHotDraw project shown as a snapshot in an animation. A variability zone evolves and two implementations are also switching their inheritance relationships. (A) Some methods defined in the stability point (such as setInsets) are redefined and several methods from StandardLayouter are migrated to SimpleLayouter. (B) SimpleLayouter was extending StandardLayouter in the past (the inheritance is shown in light green); it is now the opposite, i.e., SimpleLayouter is the default base class that StandardLayouter extends (this new refactored inheritance is shown in dark green).

Figure 15: Illustration of observations in the JabRef project. At revision 43, several new variations of Tab are implemented (A). Over time, we can see that unstable coupling seems to increase dramatically, but in fact always matches the size of the variability zone (N). Thus, in the figure, the lines for N (size) and U (unstable) properties are superposed (B). The LayoutFormatter stability point increases in size (C), although there are no apparent unstable couplings in the time line (D).

This is noteworthy because increases in unstable couplings are generally to be avoided. However, they might be acceptable depending on other factors, such as expected minimal reuse and/or short-term design strategy. A visual approach can help investigate various scenarios. In fact, in this case, each Tab is a different group of settings that can be edited in a panel interface, and the PrefsTab stability point helps define how to store the settings of each panel. We discovered this case using IHVis, by noticing the fluctuations in terms of the variability zone properties, and also by inspecting the repository log view. In particular, with regards to this stability point, the developer commented in the source code repository log, “With this design, it should be very easy to add new tabs later”, which confirms the intent to improve the design at that moment in time.

A designer might wonder, based on this level of unstable coupling, if introducing a Factory would have been useful in this case, because the client (PrefsDialog2) would no longer be coupled to the implementations. Indeed, in this hypothetical scenario, the unstable couplings from PrefsDialog2 to the variability zone would go from 3 to 0, and the Factory would manage the panel object creations.
However, since the complexity of the constructions of these objects seems rather small, it might be acceptable not to use a Factory.

Another interesting stability point that we discovered is LayoutFormatter, introduced at revision 143 (not shown in the figure). The timeline view shows increases in variability types, especially between revisions 160 and 190, shown in Figure 15D. IHVis clearly shows no unstable couplings to the implementations in Figure 15C, which means that these internal elements are isolated and virtually unknown to other clients, limiting the possible impact if they are modified (see Figure 15C). In fact, the LayoutEntry class acts as a reflection-based Factory, which means that client classes can create a type of layout by specifying its name only, using the following code: `formatter=(LayoutFormatter)Class.forName(formatterName).newInstance()`.

3.3.4. Violet

The UML Editor Violet has also some interesting patterns that can be examined with IHVis. Variations of the IGraph stability point are quickly added at the beginning of the history, as new type of diagrams are implemented. For instance, extensions to AbstractGraph such as ClassDiagramGraph, SequenceDiagramGraph and ActivityDiagramGraph are being added within revisions 4-12 (not shown in any figure). Several of these extensions are then disconnected from the diagram. AbstractGraph class and now extend diagram.abstacts.AbstractClass. This transfer of variability (and partial relocation) is depicted visually as crossing lines in the time-line view, as extensions are moved from one variability zone to another one (at revisions 46-60 in Figure 16).

As new tools are implemented with Violet, new implementations (variations) of the IEditorPartBehavior interface are added between revisions 129 and 145. In that same period of time, a few classes are coupled to the interface, and the unstable couplings follow a similar tendency as the increase in variation types, as shown in Figure 17B. This phenomenon is caused by classes (such as the Workspace class shown in Figure 17A), which are usually coupled to all tools that are supported over time. This can also be spotted visually by the symmetry of the N (Size of variability zone) and U (Unstable coupling) lines in the figure. A few other classes contribute to the stable couplings (e.g., IEditorPartBehaviorManager).

Figure 16: Illustration of a case of variability transfer occurring in the Violet project between the revisions 47 and 53. The time-line view (bottom) shows the decreasing size of the IGraph variability zone (blue line) in the framework.diagram package. Over the same time, there is an increase in the IGraph variability zone (green line) in the product.diagram.abstacts package. Parts (A) and (B) show these respective variability zones at revision 52.

Figure 17: Illustration of observations in the IEditorPortBehavior variability zone of the Violet project. The unstable couplings shown by the green line follow the same tendency, in part because of the Workspace class which has knowledge of all these implementations (A). Several new variations of the IEditorPortBehavior stability point are added over time between revisions 129 and 145, evidenced by the plain blue line that is rising in the time-line view (B).
The developers of Violet also defined the abstraction of a theme, which is implemented in varying classes, e.g., EclipseTheme, DarkTheme, VistaBlueTheme, as shown in the Figure 18A. We can observe that the N property of the stability point corresponding to this notion (ITheme) increases over time, since variations were added incrementally. Looking at the time-line view in Figure 18B, we can also see that the unstable coupling fluctuates but remains below the level of stable couplings between the revisions 39-66. Figure 18A shows that some GUI classes (e.g., UMLEditorApplet, UMLEditorApplication) are directly coupled (shown by the dark red arrows) to the concrete themes, while a larger number of classes (e.g., VioletUMLEditor, PrintPanel) are only coupled to the stability point.

We presented in this section our analysis of variability zones discovered in open-source software designs using IHVis. In the next section, we discuss a user study in which we asked participants to find observations about stability in evolving software designs.

4. User study

This section presents an investigation of the use of IHVis with human participants. The objective of our study was to see how the participants were able to use our visual tool to find structures and remarkable changes in software histories, with a focus on variability zones.

We asked twelve (12) participants to perform predetermined tasks (with expected outcomes) using the IHVis tool. All the tasks were evaluated by having participants answer multiple-choice questions concerning a priori observations identified by the authors. A multiple-choice question could have one or more correct answers, as well as wrong answers and an “I don’t know” answer. We allowed participants to document their own observations, in addition to confirming the ones that were expected. Previous work has referred to recording observations as insights, where an insight is an “individual observation about the data by a participant” (Saraiya et al., 2005) that is noteworthy in the context of the task at hand.

In the study, we first presented to participants a general overview of our approach, in which we explained how to perform tasks (using a separate dataset excluded from the full study). Participants could then try our tool until they felt comfortable with it, and we answered any of their questions. Participants then performed each of the experimental tasks using IHVis, as we recorded the task durations with a software. We compared their answers with our own, to evaluate the error rates. Participants were encouraged to talk aloud during the experiment.

4.1. Tasks

We asked participants to perform three tasks to evaluate if our tool could support them in finding insights in evolving software. The tasks were chosen as part of different levels in a drill-down, exploratory work-flow, with each task representing progressively finer-grained exploration of a software’s design. Tasks 1 and 2 involved the user’s ability to locate potentially temporal aspects of a project’s history (revisions or a sequence of revisions) by using the Metrics and Time-line displays in IHVis respectively (Figure 9C,D). IHVis complements functionalities in these tasks by allowing users to search the revision comments, since information about significant changes is often documented by the developers when they submit code. We considered events to be potentially significant if they involve important changes in the design, which could be reflected by evolving data relating to variability zones. Task 3 involved exploring the evolving characteristics of a design within a focused...
sequence of revisions, primarily by using the Network (graph) view of IHVis. Our motivation with this final task was to consider what fine-grain insights about variability zones and stability points a user can derive within the context of a significant revision sequence.

In the first task, participants were asked to browse a software history to find potentially interesting periods of time, by dragging the mouse sideways in the Metrics view, and locating revisions where particular changes of importance occur. Such changes were shown in the Metrics view, e.g., lines of code, bug fixes, and refactorings. Specifically, participants had to find the revisions with the biggest changes, by analyzing bar charts in the Metrics view of the tool and also by filtering commit logs based on keywords (e.g., "bugfix", "refactor"). Examples of possible observations for this task are illustrated in Figure 19A.

Figure 19: Illustration of tasks performed by participants. In tasks 1 and 2, users had to browse a software history and find fluctuations in terms of metrics (A) and variability zone properties N, S, and U (B), respectively. In task 3, participants had to analyze stability points and report their findings (C), e.g., cases where changes in variability zones did not affect stable elements (left) or propagated to directly coupled clients (right).

For instance, significant increases in LOC (lines of code) may indicate that new stability points were added, or that extensions of stability points were added, or simply that code not related to stability points was added. This task is a less discriminant way to locate architectural variability. However, when combined with a search of mature developer documentation of design changes in revision logs, this task could be useful in finding refactoring indicating the creation of new stability points, which could be inspected further in a separate step.

In the second task, participants also had to find possibly important revisions in the change history, but with respect to variability zones using the timeline view of IHVis. Using filtering and keyword searches, participants had to report significant fluctuations in terms of the variability zone properties presented earlier in this paper. Typical findings for this task are illustrated in Figure 19B. For example, if a variability zone increases (N property) suddenly over some series of revisions, it indicates that new concrete implementations, under the same abstract concept represented by the stability point, were added during those revisions. Increases in stable couplings (S) suggest that more client classes are actually using a stability point over time, while decreases in unstable couplings (U) could have been caused by a refactoring such as the application of the Factory (Gamma et al., 1994) pattern.

Task 3 explores in more detail the information regarding specific cases identified in the previous tasks. We thus asked participants questions about two instances of stability points, each visualized in an interactive structural view (Figure 9A,B). Specifically, we asked participants to inspect two stability points in each open-source project (Buddi and JHotDraw). In the first part of this task (3a), participants had to indicate how the stability point evolved in terms of the N, U, S properties. This part was thus similar to task 2, but it was restricted to a specific stability point.

In the second part of the third task (3b), participants had to inspect how the stability point evolved over time (using static and animated views) and report their findings. By analyzing and interacting with the visualization, a user could try to understand the evolution pattern of a stability point. For instance, creating stability points can facilitate future changes through a plug-in architecture, or changes inside a variability zone could indicate that new algorithms to parse files were added. Furthermore, this analysis may also suggest areas for improvement (refactorings). For example, a Factory pattern might be useful to better manage the instabilities of object constructions. We illustrate typical changes in variability zones in Figure 19C. Examples of high-level insights were explained to participants, and they had to try to find similar cases
in the datasets (or report that they could not find any).

4.2. Results

The results of our user study are presented in Tables 2 and 3 (task durations are shown in minutes). Distributions of the performances and errors of participants are also shown in Figure 20. To calculate the percentage of correct answers found and errors, we compared their answers with the expected ones (determined by the researchers). For instance, if the answers determined by the authors were “a, b” and the participant answered “b, c”, it would have resulted in one correct answer, one missed, and another one incorrect (out of two answers).

In terms of task performance, participants were generally able to perform tasks in a few minutes (less than 3 minutes on average). Task 3b not surprisingly took more time than the other tasks (approximately 6 minutes on average), because it involved a more complex analysis, such as identifying evolution patterns.

Performance of the participants in finding correct observations varied more with the Buddi dataset, in comparison with JHotDraw (Figure 20B). Participants had more difficulties concurring with the authors about the findings of the Buddi dataset for tasks 3a and 3b, and found ∼75% of the observations, compared to ∼90% for the JHotDraw project. In addition, they made more mistakes (Figure 20C), suggesting that Buddi’s design was harder to understand (25% of errors vs. 2.8% for JHotDraw for task 3b). The fact that JHotDraw integrated several design patterns possibly resulted in a more decoupled design that participants found easier to grasp.

We also asked participants in our user study to qualitatively evaluate techniques using a five-point rating scale. Participants were fairly confident on average with their analysis (3.9) and they liked the interface of IHVis (4.2), declaring it very useful (4.5) to find observations, even though they never saw or used our tool beforehand.

Generally, we believe these results suggest that novice software engineers were able to use our visual tool to find observations about information hiding evolution within a reasonable time frame. Furthermore, participants performed similarly with both datasets, which may indicate that the tool could lead to comparable performances for different datasets. However, there were more variability in results for task 3, probably because participants had different personal experiences in software design.

We also received general suggestions from participants. They reported that the similarity with UML made it easier to understand the visualization. Several users liked and preferred to use the slider to move in time in an animation, although a few others preferred to use the automated play/pause button instead. They suggested we could add other alternative automatic layout algorithms and a more adjustable scale in the bar charts metric view.

5. Discussion and Limitations

Leveraging the iceberg metaphor, which is analogous to the open/closed principle, IHVis presents stability points and variability zones respectively, facilitating the inspection of client couplings within a specific frame of reference. Specifically, IHVis helps (1) finding and tracking design structures aimed to support protected variations over time, (2) managing coupling between client classes, the stability points and the implementations, and (3) finding important events about protected variations in an evolving design.

IHVis in its current form has some limitations. Only Java is parsed, and the histories only come from Subversion repositories, although support for GIT is possible by using a data import script (based
Table 2: Results of the participants for the Buddi open-source project.

<table>
<thead>
<tr>
<th>Task</th>
<th>$\overline{\tau}_{\text{Time}}$</th>
<th># Obs.</th>
<th>$\overline{\tau}_{\text{Found}}$</th>
<th>$\overline{\tau}_{\text{Missed}}$</th>
<th>$\overline{\tau}_{\text{Error}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1:28</td>
<td>4</td>
<td>95.8</td>
<td>4.2</td>
<td>2.1</td>
</tr>
<tr>
<td>2</td>
<td>3:08</td>
<td>5</td>
<td>84.7</td>
<td>15.3</td>
<td>8.3</td>
</tr>
<tr>
<td>3a</td>
<td>1:52</td>
<td>6</td>
<td>69.4</td>
<td>30.6</td>
<td>0.0</td>
</tr>
<tr>
<td>3b</td>
<td>6:05</td>
<td>4</td>
<td>77.8</td>
<td>22.2</td>
<td>25.0</td>
</tr>
</tbody>
</table>

Table 3: Results of the participants for the JHotDraw open-source project.

<table>
<thead>
<tr>
<th>Task</th>
<th>$\overline{\tau}_{\text{Time}}$</th>
<th># Obs.</th>
<th>$\overline{\tau}_{\text{Found}}$</th>
<th>$\overline{\tau}_{\text{Missed}}$</th>
<th>$\overline{\tau}_{\text{Error}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1:28</td>
<td>4</td>
<td>89.6</td>
<td>10.4</td>
<td>10.4</td>
</tr>
<tr>
<td>2</td>
<td>2:54</td>
<td>6</td>
<td>90.3</td>
<td>9.7</td>
<td>1.4</td>
</tr>
<tr>
<td>3a</td>
<td>2:06</td>
<td>5</td>
<td>93.8</td>
<td>6.3</td>
<td>2.1</td>
</tr>
<tr>
<td>3b</td>
<td>6:08</td>
<td>7</td>
<td>89.2</td>
<td>10.8</td>
<td>2.8</td>
</tr>
</tbody>
</table>

on the subgit\(^5\) tool). The code must compile successfully if a given revision is to be analyzed (and visualized). The algorithms used to calculate the variability zones (and the respective counts of N, S and U) were kept simple in our tool. For instance, IHVis does not capture sub-classes beyond the first hierarchical level by default, although this can be modified in the settings. We found that the simplified calculations were enough to show significant trends in the data of the projects we studied, but perhaps deeper analyses would produce more subtle or detailed events.

Our approach only considers stability points as Java interfaces. This could be generalized to similar concepts in Java and in other languages, such as abstract classes or C++ classes with all virtual methods. Other motives that exercise the open/closed principle but do not involve class hierarchy abstractions could also be stability points. For example, a Facade, Iterator and Proxy provide limited access to elements that could change. These classes may not be defined by an abstraction such as an interface in Java. In such cases, if a designer annotated them as stability points, our parser could be modified to recognize them. It might also be necessary to annotate the implementations with respect to the stability point, as they may not be as easily deducible.

On the other hand, unlike the problem with adding new extensions to public interfaces anywhere in the design, annotated stability points (that by definition do not involve inheritance) might be more easily maintained using traditional encapsulation (e.g., with packages and access control). The annotation idea could apply to Factories who instantiate implementations, as they are also seen as clients to stability points. Google Guice (Vanbrabant, 2008) uses Java annotations and makes use of dependency injection to deal with this kind of problem.

Our visual approach uses a node-link view, already familiar to most software engineers knowledgeable in UML, combined with other visualizations and filtering to facilitate the exploration of data. Considering our observations presented in section 3.3, we believe that our interactive and visual approach, contrary to purely numerical techniques, naturally supports exploratory and qualitative analysis. These features help the designer flexibly discard outliers and focus on interesting cases.

As object-oriented designs are trade-offs involving complexity related both to implementation decisions and the problem domain, our approach does not focus on diagnosing or identifying higher level structures such as design patterns or anti-patterns. Our visual method aims at helping a designer identify and understand how variability zones evolve over time.

As discovered in our observations, IHVis revealed the presence of unstable couplings. Some of these are due to test cases, and they might be more tolerable than unstable couplings coming from clients of a framework or API. Our study was observational and not corrective, but we can imagine that in some cases the burden is on the test designers to accept the risk of having their code change if the implementations change. Black-box tests might imply less unstable couplings than white-box tests, but

\(^5\)http://subgit.com
we did not explore such possibilities.

There are also trade-offs to consider when deciding whether to hide implementations to improve a design. Implementations might be relatively unconnected to other elements, thus limiting the negative impacts of their changes. Also, stable implementations do not necessarily need to be wrapped under an interface.

Design strategies being a set of heuristics, the motivation to enforce rules about stable or unstable coupling or introduce abstractions will vary depending on the actual project constraints, priorities and objectives. But a “strict” application could mean that an interface should be used whenever the need is apparent to isolate the changes into separate variability zones and limit the propagation to several, possibly increasing, external elements. In this case, a possible guideline would be to use an intermediate class, such as a Factory, that returns an interface to hide the concrete implementations to clients whenever the changes in these implementations start to affect too many external and unrelated clients in other subsystems or packages.

Stability points and the study of the corresponding variability zones are possibly a way to validate appropriateness of design patterns (such as GoF (Gamma et al., 1994)) in a project. Many of these patterns make use of stable interfaces to hide possible variations of objects to external clients. If the N, S and U counts with respect to a design pattern’s stability point(s) have anomalies in a project’s history (e.g., the Observer pattern is introduced, but N and S remain small, meaning no future implementations or clients are provided), that could indicate a bad design choice (over complex structures with no need). Conversely, if N and S continue to grow over time in a project, it could mean that the pattern is fulfilling a need within the design space.

5.1. Threats to validity

The user study that we performed aimed at evaluating the usefulness of our novel approach to explore variability zones in software. We could not find an alternative approach against which to compare quantitatively our tool.

In our user study, we asked participants to use our tool to analyze software designs, following a within-subject design (they thus had to perform all tasks in sequence). To reduce the confounding effects due to task ordering, a Latin-square determined the order of tasks and a random number generator determined the order of the open-source projects among participants. We did not test the visual acuity of participants, and users with abnormal visions may have performed worse than others.

We measured the task durations and error rates using a software program. However, the correctness measurements were based on the participants’ ability to find the same insights as the authors, which were not verified with the original open-source software developers. Furthermore, multiple-choice questions were used in this part of the evaluation, which could pose a threat to internal validity. In computing the depth of variability zones (e.g., the N property), the number of intermediary levels was fixed to 1.

With respect to external validity, users that participated in our study only analyzed a limited history of two open-source Java projects in Subversion repositories. Mean times are for participants in our study, who were volunteers within an academic community and may not represent a general software engineering community. In an effort to recruit a representative sample of human subjects with experience in object-oriented software, we verified that participants had completed at least one course in object-oriented software design or had equivalent practical experience. External validity is also threatened by a small number of projects and did not cover all revisions of those projects.

We also demonstrated how our tool can be used in practice with the case study shown in Section 3.3. As with any case study, this evaluation remains qualitative, and essentially shows that our novel approach is working, and can lead to discovering insights about software designs.

6. Related Work

The introduction of this article discussed already much related work concerning the principles of information and software design, which we do not repeat here. The focus of this section is on tools that visualize software designs. We also discuss previous work in contrast to our own whenever possible, although we did not find any work that takes the same approach as we. Comparing novel techniques with non-existent alternatives is challenging for researchers in software visualization (Beck and Diehl (2010); Langelier et al. (2008)).

The approach that we presented in this paper is an applied method, exploiting visualization and interactivity, to allow users to explore dynamic software data (modeled as networks). In information
visualization, common ways to visualize dynamic networks include small multiples (Tufte, 1983), difference maps (Archambault et al., 2011b), and animation (Erten et al., 2004). Small multiples essentially juxtapose static representations of the graph at different time slices, while a static difference map also highlights differences between two time slices. An animation smoothly interpolates changes between two time slices. In previous work, researchers in visualization have found benefits of using static representations (Archambault et al., 2011a; Farrugia and Quigley, 2011) and animation (Zaman et al., 2011). Because of the possibly complex tradeoffs between the techniques, IHVis allows the user to decide which approach is best, as all are supported in the visualization.

The software visualization community aims at finding valuable observations concerning a software project and convey how it is constructed and how it is evolving over time. Previous work has focused on visualizing different aspects of software and its evolution (see surveys (Diehl, 2007; Caserta and Zendra, 2011; Khan et al., 2012)). There are also non-visual approaches to studying software design structures and their evolution.

6.1. Visualizing static aspects of software

Different approaches have been used in the literature to explore a single version of software, and some of them can be used to compare two versions of a software by juxtaposing static representations. Lattix (Sangal et al., 2005) makes use of matrices to reduce the problem of occlusion for software with a lot of dependencies, but does not support the browsing of software histories. Abdeen et al. (2010) proposed a visualization in a matrix-inspired form to see more detail about package relationships, particularly raw size of a package, the incoming and outgoing coupling, as well as cohesion. They consider in their visualizations the visibility of classes (encapsulation with packages) as well as client coupling to packages. Their approach is limited to packages as a unit of modularity, and only applies to single versions of a software project. Matrices, although scalable, can be complex when the software they represent is complex. To help reduce this issue, TreeMatrix (Rufiange et al., 2012) proposed using them only when requested, in combination with node-link diagrams to facilitate the understanding of higher level software relationships. Holten and Van Wijk (2008) proposed a technique to compare hierarchies of two versions of the same software using a cable-and-plug wiring metaphor. Classes and hierarchies that are the same in both versions are connected by wires. Hierarchies in both versions are placed opposite each other to make it easier to compare. The notion of a module is limited to the scope of a package.

6.2. Visualizing software evolution

Visual approaches have also been explored to study the evolution of software. CodeFlows (Telea and Auber, 2008) allows comparing hierarchies of classes of several file versions to identity changes in code such as merges and splits. Voinea and Telea (2007) implemented a visual data-mining tool to show the changes in source code repositories. Their approach allows different levels of detail in a condensed view to show evolving metrics (lines of code and number of modifications). McNair et al. (2007) proposed a technique to filter the evolution data based on change sets, allowing designers to see the changes that were required to fix a bug, for example. However, all of these visual techniques ignore the evolution of coupling and especially in a context of encapsulation.

The Evolution Radar (D’Ambros et al., 2009) focused on visualizing the evolutionary coupling, collected from source code repositories. Wettel and Lanza (2008) used a 3D building metaphor to represent the activities occurring in a software change history and explored real projects. Langelier et al. (2008) used an animated tree-map layout to visualize changes in package structure and metrics, and they identified some cases of responsibility transfer or growing classes. Other work (Denier and Sahraoui, 2009) has found visual patterns of interest in source code, but cannot be used to show evolving design structures. Although these contributions organize the classes in packages and can convey useful information about structure, they do not support the visualization of arbitrary encapsulation. Hindle et al. (2007) used a graph-based layout and colors to show evolution of couplings between packages. These interesting approaches were not formally evaluated in user studies.

6.3. Other approaches

Other non-visual approaches have also been used to explore software histories or identify design structures, such as metric-based empirical approaches (Aversano et al., 2007; Alshayeb and Li,
or change-impact models (German et al., 2009). Contrary to these, visual approaches such as ours have the potential benefit of exploiting the human visual system to facilitate the interpretation of complex data. Furthermore, visual approaches can lead to unexpected discovery through interactive explorations (Ware, 2000) and can be used to convey changes over time.

Concerning the problem of arbitrary encapsulation (which appear at the micro-architectural design level or higher, beyond the traditional package level), some elements in pattern description languages are related to our work. eLePUS (Taibi, 2007) defines several relations that apply to information hiding. The Encapsulation relation addresses hiding internal details of a class or group of classes. The Decouple relation specifies the notion that two elements are independent of each other such that changes in one do not affect the other. Bayley and Zhu (2010) proposed the Client depends on Root (CDR) property, which specifies that if a message is sent from a class that is not explicitly mentioned in a pattern, then the operation must be declared in the root (of a hierarchy) class where that property is specified. This applies to our work because it specifies a kind of encapsulation – clients generally access the elements of a pattern through a stable interface and do not interact with other elements. The concept of a blackbox framework (Fayad and Schmidt, 1997) is also related.

7. Conclusions and Future Work

Interactive visualizations, such as the one presented in this paper, can naturally support explorations of structures and convey evolution tendencies. Our visual tool (named IHVis) allows exploring software designs that evolve over time. The principal novelty is that it helps designers find and track structures used in the protected variations pattern. By recognizing stability points and variability zones with our tool, a designer can inspect, understand and possibly correct the application of information hiding.

Analyzing evolution of variability in software designs is still hard to do in current tools, and our main contribution is to provide a method to explore software stability over time. We presented practical use cases of the visual tool to uncover software elements in real software, such as plug-in architectures, variability transfer, and changes in terms of hierarchies of classes as part of refactorings. This could help finding good applications of patterns that should be repeated for more projects, or ones that should be avoided. For example, a user might discover that a Factory pattern would have helped isolate external clients from changes in concrete implementations of parsing strategies.

The results of a user study show that the tool is useful and that users were able to confirm insights found by the authors in open-source Java projects stored in Subversion repositories. Participants were first asked to browse large change histories and report noteworthy revisions in terms of evolution in variability (e.g., increased usage of stable interfaces, reduced couplings to concrete implementations of interfaces). Software engineers involved in our study were then able to find interesting changes such as the introduction of interfaces or other software elements, renaming of classes, and increased couplings to elements that might need to be hidden. This further encourages the development of new tools to explore visually and check applications of information hiding concepts and design patterns in evolving software.

In the future, we plan on exploring other information hiding mechanisms in more detail (e.g., intermediary classes), and making incremental improvements to our tool (e.g., supporting different automatic layout algorithms and other types of couplings). Other research directions include the exploration of insights in evolving designs, such as the evolution concerning the Law of Demeter, as well as visual and empirical approaches to study the application and usefulness of design structures and patterns in software.

Acknowledgments

We would like to thank the participants in our study and the anonymous reviewers for their insightful comments. Thanks to the open-source organizations for developing and giving access to their source code repositories. This research was funded in part by FRQ (Fonds de recherche du Québec) and ÉTS.

References

Sébastien Rufiange received his B.Eng. degree in software engineering in 2008, and a Ph.D. degree at ÉTS (Montréal, Canada) in 2013. He has taught courses on software design, re-engineering and distributed architectures. His research interests include the visualization of information and networks, software visualization, software design and evolution. He is currently a postdoctoral researcher at INRIA Bordeaux Sud-Ouest (Bordeaux, France), working on network visualizations.

Christopher P. Fuhrman is a professor in software engineering at ÉTS (Montréal, Canada) since 2001 where he teaches undergraduate and graduate courses on object-oriented software analysis and design with agile processes. In 1988 he earned a bachelor’s degree in computer science at West Virginia University (West Virginia, USA) and a Ph.D. at the Swiss Federal Institute of Technology (Lausanne, Switzerland) in 1995. He has worked as a software engineer in corporate and government environments, including Apple Computer and NASA Software Independent Verification & Validation.