Towards Mentoring Object-Oriented Evolutionary Development

Zhenchang Xing and Eleni Stroulia
Computing Science Department
University of Alberta
Edmonton AB, T6G 2H1, Canada
{xing, stroulia}@cs.ualberta.ca

Abstract
Object-oriented software is increasingly developed using an evolutionary development process model. Therefore, capturing and understanding the evolution that the system’s logical design has gone through can provide valuable insights in support of consistently maintaining and evolving the system, without compromising the integrity and stability of its architecture. In this paper, we present a method for capturing and analyzing the design evolution of object-oriented software systems. This method relies on UMLDiff, a heuristic UML-structure differencing algorithm, which, given a sequence of UML class models corresponding to the logical design of a sequence of system code releases, produces a sequence of “change records” that describe the design-level structural changes between subsequent system releases. Various design evolution patterns are then analyzed to reveal the rationale underlying design decisions that may affect the software system. The recovered knowledge about the system’s logic design evolution enables the overall understanding of system design evolution and provides the basis for mentoring the developers on future maintainance activities. We report on one real world case study evaluating our approach.

1 Introduction
Today, most object-oriented software systems are developed using an evolutionary development process model. Most software-development lifecycle models widely-adopted today (for example, Rational Unified Process and the various variants of agile development) advocate that the design and coding activities should be interleaved and, as a result, design and implementation should evolve in tandem. It is generally understood that to consistently evolve the design of an artifact one must have a good understanding of the current design of the artifact and the rationale that led to it. The implication, then, is that design understanding is critical to evolutionary development.

The research question then becomes: “How can one effectively support evolutionary development by enabling the developers to understand the software design rationale and its evolution? And how can one offer advice regarding how to further evolve the system and its design based on this understanding?”

There already has been a substantial amount of research on the subject of understanding the evolution of software. In general, these related research efforts are aimed at understanding system evolution based on the change documentation and code-level metrics [4,7,11,12], understanding the evolving system entropy as a measure of how fast the system changes and how stable the system is [9], understanding co-evolution of design elements [7,17,20], and understanding the nature of individual developer’s contributions to the system evolution [23]. This work has not quite bridged the gap from “understanding the past” to “advising for the future”.

On the other hand, there has been work aimed directly at offering contextual advice during development by extracting similar situations in the past history of the system lifecycle. Hippikat [2] and Strathcona [10] extend the functionalities of the Eclipse IDE by enabling the developers to query about system elements. These tools then respond with relevant segments of documentation and/or similar code snippets from which the developers can draw analogies towards solving their current problem.

In our work with JDevAn [22], we have taken a broader stance to the problem of advising software developers. We have developed a system that is capable of understanding (a) the design of the software system under development at the logical-design level and (b) the evolution history of the design both in the short- and in the long-term [20,21]. In addition to providing “contextual” project-specific advice—like Hipikat and Strathcona do—based on its understanding of the system design evolution history, JDevAn is also endowed with knowledge of abstract object-oriented design principles and more specific design evolution patterns and refactorings. Thus, it can relate this knowledge to its understanding of the system under development to offer advice on how to improve the system design based on “absolute” (i.e., project independent) terms.

The rest of the paper is organized as follows. Section 2 reviews the work related to our work with JDevAn. Section 3 discusses in detail the JDevAn methodology and architecture. Section 4 details an in-depth qualitative case study we conducted to evaluate JDevAn’s advice-offering capabilities. Finally, in section 5 we conclude with a review of the lessons we have learned from our experience to date and our plans for the future.
2 Related research

**GNU diff**, a text-comparison tool, is commonly used to reconstruct the changes between the subsequent versions of the software system. Its output, CVS-like deltas, together with modification requests and bug reports, have been substantially used for aiding software evolution and maintenance activities [4,7,9,11,12,17]. However, since tools like **GNU diff** consider a software system as a set of files containing lines of text, they report changes at the textual level and ignore the high-level logic structural changes of software system.

However, software systems are better understood in terms of structural and behavioral models, such as UML class and sequence diagrams. Unfortunately, representation-specific structural differencing algorithms [28] report changes in terms of their own primitives – XML differencing algorithms report changes to XML elements and attributes instead of XMI entities - ignoring the semantics of the elements these primitives represent, and thus generally do not correspond to the developer’s intuition very well. There are CASE-specific of UML-differencing methods [13] that detect differences between UML diagrams, assuming the existence of persistent element identifiers for the objects in the UML models. This capability is obviously irrelevant when the source code is not developed in its entirety within the CASE tools.

In contrast, our design-evolution analyses rely on a heuristic structural differencing algorithm, **UMLDiff**, that is aware of UML semantics, and thus is able to infer renamings and moves of various design entities based on their identifier similarity and relationship similarity.

There has been some work at investigating the use of comparative analysis of different snapshots of a software system for drawing inferences regarding its evolution. Demeyer et al. [3] define four heuristics based on the comparison of source code metrics of two subsequent snapshots to identify refactorings of three general categories. Filip et al. [15] investigated the use of clone-detection to identify “move methods” refactoring. In contrast, our method depends on the structural modifications identified by a design-level structural differencing algorithm, which enables richer and more accurate analyses. BEAGLE [19] aims at analyzing the structural evolution of software system, using origin analysis to determine the “origin” of “new” files and the merging and splitting of files. In BEAGLE, origin analysis works at the file-structure level and old files are detected as the “origin” of new files, based on source code-level metrics and clone detection. In contrast, in our work we have developed a domain specific structural differencing algorithm that is capable to capture the structural evolution of the logic design that a software system has gone through over its lifespan.

Ryder and her colleagues also worked on comparative analysis of sequences of system versions [14]. They define a set of atomic changes derived from the comparison of the abstract syntax trees of classes in two versions of a project, such as add an empty class, modify body of a method, etc. The major objective of their work is to analyze the impact of changes on test cases, while our work is aimed at capturing higher-level design evolution knowledge.

Recommendation systems, such as Hipikat [2] and Strathcona [10], use information sources associated with the software development to present relevant software artifacts to the developer’s evolution task on hand. The objective of these methods is to facilitate the developer’s tasks, such as locating a component that could be reused, suggesting a potential solution to a particular type of bug, etc. In contrast, the proposed software design-evolution agent works at the granularity of higher abstraction, providing supports on monitoring and mentoring object-oriented software design and its evolution.

There has been some work at analyzing the changes of software at the design level. Egyed [5] has investigated a suite of rule- and constraint-based and transformational comparative methods for checking the consistency of the evolving UML diagrams of a software system. Selonen et al. [16] have also developed a method for UML transformations, including differencing. However, they cannot surface the specific types of changes as reported by **UMLDiff** and these projects have not explored the product of their analyses in service of evolution understanding and mentoring.

3 Java Design-Evolution Analysis

The long-term objective of our design-evolution analysis work is to develop a suite of methods for recovering a detailed and precise model of the design-level structural changes that have occurred in a long-lived evolving software system. The JDEvAn - “Java Design Evolution and Analysis” - tool, developed as an Eclipse [27] plugin, implements and evaluates our methodology. JDEvAn analyzes the evolving structure of design-level software artifacts and automatically produces reports of interesting properties they may exhibit and interesting events or trends during their evolution, such as for example, design-level structural change patterns discussed in this paper. The results of the JDEvAn analysis provide an informative overview of the system’s and its components’ design-evolution history and can be valuable to developers who, based on it, may reach informed decisions on future evolution and maintenance activities, such as for example, mentoring on a practical type of change based on past cases, or advising when and where to apply refactorings.

3.1 Meta-model: Design recovery

JDEvAn focuses on the logical view of object-oriented systems as the first design artifact to analyze. Its primary input is the system’s source code, residing in a versioning system. JDEvAn’s fact extractor recovers a data model of the subject system’s class design. Formally, the meta-model of the data is a graph, $G_D(V_D, E_D)$. The nodes set, $V_D$, contains program entities of the types supported by the Java programming language, i.e., package, (anonymous)
class, interface, field and method (including constructor and initializer). Each entity type is associated with its corresponding set of attributes, such as identifier, visibility, modifiers, etc. The edge set, \( E_p \), contains tuples of the form \((relation, v_1, v_2)\), where \(v_1\) and \(v_2\) are nodes and \(relation\) is a UML dependency between them. The supported core relations are containment, declaration, inheritance, interface implementation, field read/write and method call (including this, super, and constructor calls), class creation, field data type, method return type, parameter type, and exceptions thrown by a method.

The extracted model facts are stored in a relational database. Most derived facts are defined as database views, based on the original facts about ground entities and relations. However, in order to improve JDEvAn’s performance, some frequently used derived relations, such as method implementations, method overrides and class usage, are stored into the database tables after a single query execution after these facts are requested. To address the lack of recursive computation capability of the database, which is essential to computing the transitive closure of various relations among entities, Simon’s transitive closure algorithm [18] has been implemented as a database server-side extension to compute, at the end of the fact-extraction process, the transitive closure of the containment hierarchy, inheritance hierarchy, field read/write, method call, and class-usage relations.

The JDEvAn’s reverse-engineered data models capture the logic design of object-oriented software system and are the primary input to the subsequent analysis processes.

### 3.2 UMLDiff: Design differencing

The crux of JDEvAn’s analysis capabilities is its structural differencing algorithm, UMLDiff, which analyzes how the logical design of the system changes over time. UMLDiff is a domain-specific structural-differencing algorithm, aware of the UML semantics. UMLDiff establishes that two entities of the same UML type in two different versions are the same, based on the two heuristics below.

**Identifier similarity:** Similarity of identifiers can serve as the first indicator for matching two entities of the same type. Of course, a developer can remove a class, and then add a new class with the same name and different functionality. However, this case should rather be a rare exception.

**Relationship similarity:** When an entity is renamed or moved, its relationships to other entities, such as the members it contains, fields it reads/writes, methods it calls or is called by, etc., tend to remain the same. Therefore, by comparing the relationships between two entities -- one belonging in the former of two subsequent versions and the other belonging in the latter -- renamings or moves can be inferred. UMLDiff uses a threshold value for the percentage of the relationships that have to match for two entities to be considered as the “same” entity renamed or moved. If, for a given entity in the “before” version, there are several potential matches above the user-specified threshold in the “after” version, the one with the highest similarity score is chosen.

The UMLDiff matching process assumes that software changes are frequently saved back to the versioning system. If many changes are made without saving the intermediate versions back to the repository, the accuracy of UMLDiff may suffer.

UMLDiff takes as input two graphs, \( G_{v_1} \) and \( G_{v_2} \), corresponding to the design facts extracted by two subsequent versions of the subject system. It traverses the containment spanning trees of the two data models implicit in the two graphs and identifies corresponding entities based on the above heuristics.

![Figure 1: An example of change tree](image)

The comparison result is represented as a change tree, summarizing the modifications to the various design entities of the system and their dependencies. Change trees are visualized to the user as shown in Figure 1. The different icons to the left of each node represent the different object-oriented entities: “class”, “interface”, “method”, and “field”. The top-right adornments show the modifiers of the object, for example, “abstract”, “static”, etc. The bottom-left adornments represent the status of a particular object. It can be plus sign for “add”, minus sign for “remove”, triangle for “rename” and/or “change signature”, arrow with minus sign for “move out from source”, arrow with plug sign for “move into target”.

Figure 1 shows that a new abstract class, `Statement`, was created with three newly created abstract methods, `eachRentalString()`, `footerString()`, and `headerString()`. The `value()` methods of its two subclasses, `HTMLStatement` and `PlainStatement`, were pulled up into the new class `Statement`. This change tree corresponds to the differences between version 27 and 28 of the extended refactoring sample from Fowler’s book [6] as found in [25]. It represents the modifications to the class model after a “Form Template Method” refactoring [6].
UMLDiff builds a detailed and accurate picture of the evolving software system and its components’ design evolution without requiring explicit evolution documentation in the form of consistent change logs. This first design-differencing step is crucial to the recognition of structural change patterns applied in the subsequent stage of the process, and to the analysis of design rationale behind these changes.

3.3 Design evolution patterns
Change is certainly an essential ingredient in object-oriented software systems with their emphasis on evolutionary development, which implies that change itself is the key to understanding how and why the design of a system has evolved the way it has. The fundamental intuition of our design evolution analysis work is that the important design decisions will be reflected in the source code and in the way the code has changed over time, and such decisions can be recognized from their effects (design-level structural modifications) on the class models reverse-engineered from source code.

JDEvAn, through its fact extraction capabilities, captures the logical design of object-oriented software system, which involves all the relevant object-oriented design entities and the important relationships (including appropriate transitive closure) between them. The queries are then defined to elicit the potential problematic designs, such as classes with many same name fields and methods, methods declared as public but not being accessed outside its containers, fields declared as concrete class that implements some general interface, and so on.

JDEvAn, through its structural differencing capabilities, captures the evolution of the system’s logic design. Overall, there are five types of elementary structural modifications identified by UMLDiff:

1. Additions of new packages, classes, interfaces, methods, and fields;
2. Removals of packages, classes, interfaces, methods, and fields;
3. Movements of methods and fields from one class to another, or movement of classes and interfaces from one package to another, or movement of packages from one subsystem to another;
4. Renamings of packages, classes, interfaces, methods and fields;
5. Modifications of signature, such as visibility, modifiers, data type, and parameter list, of classes, interface, methods and fields, and modifications of class inheritance and interface implementation.

These elementary structural modifications are the direct output of our structural differencing algorithm, and they serve as the building blocks for recognizing more complicated structural change patterns, such as refactorings [6].

Various types of elementary structural modifications are combined to define the characteristic of more complicated structural change patterns. For example, “Extract Super-class” (see section 4.2.3) can be characterized as a change involving (a) the addition of a new class, (b) the modification of one or more classes from extending java.lang.Object to extending the newly added class, and (c) the moving of one or more fields and methods from these existing classes to their new superclass. Such characteristic structural change patterns are implemented as database server-side extensions, which can be called to query the instances of well-defined design-level structural change patterns.

A single structural modification may not be especially interesting. However, typically a large number of same or similar changes may occur over a limited time span, which provides a good indicator that the system is going through fundamental and rapid design changes, which may affect a subsystem or even the whole system. For example, classes and interfaces with similar names were removed but their field and members were moved to their superclass. This may be the result of merging parallel inheritance and collapsing hierarchies as discussed in section 4.2.5.

For an evolving software system with \( N \) successive versions, UMLDiff can be applied \( N \) times to generate the differences between the \((I+1)^{th}\) and \( I^{th} \) versions, where \( 0 \leq I < N \) (supposing there is a virtual version 0 with no entities), resulting in a sequence of \( N \) change trees that records the structural modifications of the logic design of the subject system over time. This change-tree sequence provides an audit trail of the design-level structural changes that the system and its components have suffered throughout their evolutionary lifecycle. This trail is analyzed to produce a system-evolution profile and class-evolution profile for each individual system class/interface, which are then analyzed to discern interesting system and class evolution phases and styles and the pairs of co-evolving classes [20, 21].

3.4 Design mentoring
JDEvAn is able to bring to surface the potential problems, through its various querying and analysis methods discussed above, that the system’s logic design has suffered over its lifespan. Associated with each of these problems are potential design modifications that may be applied to remedy the discovered problems or general advice on how the design process could potentially proceed. Clearly, the final arbitrators of whether or not to follow this advice are the developers themselves. However, we believe that the very process of recognizing and reflecting upon specific interesting design and design-evolution examples helps developers acquire valuable design experiences. By capturing and analyzing the instances of recurring structural evolution patterns and the overall system and class evolution profiles, the underlying design rationale may be better understood.

Design evolution has to be guided by high-level object-oriented design principles, such as:

- Program to an interface, not an implementation
- Favor object composition over class inheritance
• Adhere to consistent meaningful names
• Do not unnecessarily expose fields and methods
• Comply with the Law of Demeter [26]

At the same time, it is also informed by state-of-the-art practices, such as
• Extract interface, superclass or class
• Collapse hierarchies
• Form template method
• Use typesafe-enum objects instead of numeric type codes

The instances of these design-evolution patterns, when discovered in the evolution of the software system, indicate that the developers are trying to comply with the well formulated object-oriented design principles, and they can serve as the concrete design and design evolution examples for developers to learn how to design and evolve the object-oriented software in general.

On the other hand, JDevAn is able to discern any general design evolution patterns of its user’s interest, for example, the same or similar structural changes made in a large amount over a short period of time. There may or may not exist the systematic theories behind them, but they represent the project-specific evolution knowledge, which can not be learnt from the textbook. More often than not, they are not recorded in the development log; they usually just exist in the developers’ minds, as part of their overall software-engineering experience. However, JDevAn is able to recover them and present them to developers as a set of contextual advices (see section 4.3), which may be valuable to guide future development and maintenance activities.

4 The JFreeChart Case Study

JFreeChart is an open-source Java class library for generating various types of charts. It has been developed for more than 4 years; there were 31 major releases between the first version 0.5.6, released on December 1 2000, and the last version 1.0.0, which was released on November 29 2004. We used JFreeChart as the subject of an extensive case study to evaluate JDevAn’s ability (a) to uncover modifications aimed towards improving the object-oriented design of software, (b) to detect opportunities for such changes and (c) to mentor on how to perform such changes.

To that end, based on our own software design and development experience, we defined a set of queries on the JDevAn fact base to recognize potentially problematic patterns before or in version $i$. The queries refer to the facts extracted by the JDevAn parser regarding the extracted logic design entities and their relationships, the structural modifications reported by UMLDiff when comparing version $i$ to the version $i-1$, and the results of the various subsequent analyses [20,21]. Then, we examined the changes reported by UMLDiff when comparing version $j$ to version $j-1$ ($j>i$) to see if the changes implied by the problematic patterns discovered by our queries were actually made. When this was the case, we recorded the corresponding queries as valid heuristic mentors that advise developers on how to maintain/evolve their system based on their design knowledge and their understanding of the software system status.

Applying UMLDiff to pair-wise compare the 31 releases of JFreeChart uncovered a substantial number of changes, summarized in Tables 1. Its intensive and varied design-evolution history make JFreeChart an appropriate test-bed for the evaluation of JDevAn.

<table>
<thead>
<tr>
<th>Table 1. JFreeChart’s design changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity renaming</td>
</tr>
<tr>
<td>Entity move</td>
</tr>
<tr>
<td>Entity addition</td>
</tr>
<tr>
<td>Entity removal</td>
</tr>
<tr>
<td>Class inheritance change</td>
</tr>
<tr>
<td>Interface implementation change</td>
</tr>
<tr>
<td>Class usage change</td>
</tr>
<tr>
<td>Datatype change</td>
</tr>
<tr>
<td>Modifier change</td>
</tr>
<tr>
<td>Visibility change</td>
</tr>
</tbody>
</table>

4.1 Adherence to “first principles” of object-oriented development

In this section, we discuss design changes motivated by high-level principles of object-oriented design.

4.1.1 Adopting a consistent, meaningful naming scheme

The adoption of a consistent and meaningful naming scheme is very important in object-oriented design. A case in point is the Abbot method that advocates the discovery of the system classes from the nouns in the textual requirements specification of the system, and the various “renaming” refactorings aimed at improving code readability and understandability by alluding to the functions of the design elements.

In JFreeChart, many renamings were discovered. For example, DEFAULT_COLORBAR_THICKNESS_PCT was renamed into the more meaningful DEFAULT_COLORBAR_THICKNESS_PERCENT. In version 0.6.0, fields categoryGap, introGap and trailGap of class BarPlot were renamed to categoryGapPercent, introGapPercent and trailGapPercent respectively, to reflect the change of these fields’ type from type int to type double. Similarly, in version 0.9.3, method getNormalColor() of class MeterPlot was renamed to getNormalPaint() to reflect the fact that its return type was changed from java.awt.Color to java.awt.Paint. Essentially all types of design entities, including packages and interfaces, were renamed, frequently to coincide with more “substantial” changes in their function in the context of the overall system.

Note that renamings are among the basic fact types discovered by UMLDiff. Advice on when to rename and how
has to be application-domain specific and is, therefore, impractical to automate. One could imagine however that “renaming advice” could be offered when other similar interesting design changes happen, such as changes to the data types of entities or the signatures of methods.

4.1.2 Programming to Interfaces

Programming to interfaces and not to implementations is another important tenet of object-oriented development [8]. When the client is implemented to be unaware of the internal implementation of the server class, assuming only the specification of its public behavior interface, enables flexibility in the evolution of the server: as long as the public interface remains the same, modifications to each implementation will not break its clients.

The design advice here is to not declare fields and methods with particular concrete classes but rather to commit only to interfaces and abstract classes. The more abstractions introduced, the more flexibly can the system implementation evolve. The meta-model of JDevAn captures the class inheritance and interface implementation relationships, including their transitive closure, in its fact extraction process. Thus, we have defined queries to obtain fields and methods declared as:

- Concrete classes that implement interfaces
- Specialized interfaces that extend general ones
- Subclasses that extend abstract super classes

These fields and methods are the candidates that should be examined to see if the general interfaces or abstract classes could be used.

The developers of JFreeChart made efforts to comply with this principle. For example, before version 0.7.3, the class XYPlot declared four fields (horizontalColors, horizontalLines, verticalColors, verticalLines) of the concrete java collection class java.util.ArrayList, which were changed to the interface java.util.List in version 0.7.3. In the same version, the LinePlot’s method getValueAxis() was changed to return the abstract class ValueAxis instead of its subclass VerticalNumberAxis. Furthermore, in version 0.9.19, the return type of several methods of CategoryPlot and XYPlot was changed from the interface java.util.List to the more general interface java.util.Collection.

It is interesting to note that a super interface (or class) sometimes needs first to be extracted so that the clients can then start using it instead of its implementations (concrete subclasses). An example will be discussed in section 4.2.1.

Furthermore, it is also possible for JDevAn to remember the specific instances of programming-to-interface modifications it has seen in a project and can offer contextual advice in later phases when the same “implementations” are used.

4.1.3 Favoring composition over inheritance

Object-oriented software engineering enables white-box (design) reuse through class inheritance and interface implementation and black-box (implementation) reuse through object composition. Frequently, software teams make the “stronger” commitment to white-box reuse when they only need black-box reuse. The result is high coupling among the classes in the inheritance hierarchy, brittleness in the evolution of the base class, and overriding of unwanted features by the subclasses. Object composition enables classes to reuse objects in terms of their well-defined interfaces, with limited implementation coupling and increased flexibility. This is the intent behind the “favor object composition over class inheritance” tenet [6,8].

Let us look at a concrete example in JFreeChart. In version 0.9.5, four classes HorizontalColorBarAxis, VerticalColorBarAxis, HorizontalLogarithmicColorBarAxis, and VerticalLogarithmicColorBarAxis were introduced into system. Their internal structure was the same, except for the fact that they extended HorizontalNumberAxis, VerticalNumberAxis, HorizontalLogarithmicAxis, and VerticalLogarithmicAxis respectively. In version 0.9.9, all except HorizontalColorBarAxis were removed and this was renamed to ColorBar. This class did not extend ?Axis class; instead it simply contained a field of type ValueAxis, the ancestor abstract class of all ?Axis classes. Its clients, such as ContourPlot, can call ColorBar.setValueAxis(ValueAxis) to pass any axis objects, conforming to the interfaces defined by the ValueAxis abstract class, and the color bar is drawn in the form of the given axis parameter. This example illustrates a poor choice of inheritance vs. composition at the time the four ?ColorBarAxis classes were introduced, which was subsequently amended by the introduction of the ColorBar class. The intent of the original four classes was to add a color bar within ContourPlot; this color bar may be horizontal or vertical and may be in a linear or a logarithmic axis. This common feature, available across all four different ?Axis classes is better accommodated in a new class since its implementation is independent of the other ?Axis classes’ features.

In the above example, the evidence for the need to replace inheritance with composition is the simultaneous development of “parallel inheritance hierarchies”; this change is easily recognizable by the co-evolution detection [20] capability of JDevAn. Of course, there are many other types of situations where composition could be advocated instead of inheritance, such as subclass inherits many features that are not used at all by its client, we are working on developing a more complete catalog of “symptoms” that can be addressed by this type of modification.

4.1.4 Law of Demeter

The “Law of Demeter (LoD)” [26] – “only talk to your friends” - is a simple style rule for object-oriented design. It advocates that the methods of a class should only manipulate the class’ own fields and should call methods defined in the class or the classes whose instances it contains. It is essentially an object-oriented formulation of the general “low coupling” software-engineering principle.

Computing the object form of LoD requires the dynamic analysis of software system. However, there are
some symptoms that can be easily detected in JDEvAn’s data models in terms of “high coupling” and “low cohesion”. For example, we have defined queries that return the fields and methods defined in one class but are mostly used in other classes. Such fields and methods often need to be moved in order to enhance encapsulation and reduce coupling.

In our case study, the JFreeChart class coordinates such objects as `legend`, `plot`, `axis` and `dataset` in order to draw a chart on a Java 2D graphics device. In the early versions of the system, it used to delegate the actual drawing to the `Plot` object it contained. In version 0.5.6, it had four fields, `seriesPaint`, `seriesStroke`, `seriesOutlinePaint`, `seriesOutlineStroke` that were representing properties of the plot being drawn, since they were only accessed by plot classes. They should, therefore, be accessed from within the `Plot` object according to LoD. Indeed, in version 0.6.0, these four fields were moved to the `Plot` class.

### 4.1.5 Information hiding

Object-oriented languages provide explicit support for defining the scope of the various design elements of a system. Frequently, developers start off with making elements “too accessible”; as the picture of the scope of the valid clients of each element becomes clearer, the element’s visibility can be restricted.

JDEvAn queries are defined to return design elements, such as fields, methods, nested classes and interfaces, which are not declared as private but have not been accessed outside their containing elements. Furthermore, visibility changes are one of the five basic types of structural changes reported by UMLDiff.

For example, 519 (about 60% of all the visibility changes, see Table 1) fields and methods changed their visibility to private in release version 0.9.4, which clearly indicates that JFreeChart underwent an information-hiding restructuring, an observation validated by the CVS log statement “fix errors reported by CheckStyle”. Checkstyle is a tool to help programmers write Java code that adheres to the coding standard, such as Sun Java Specification.

### 4.2 Refactorings

Refactoring, i.e., behavior-preserving structural modification is one of the most important practices in the agile software-development process.

#### 4.2.1 Extracting interfaces

A corollary of the programming-to-interfaces principle is the “Extract Interface” refactoring. If two or more classes have some common behaviors, an interface should be extracted to include the methods delivering the shared behaviors. In this manner, the clients of the refactored classes that are interested in their common behaviors can start depending on the extracted interface, get decoupled from the classes’ implementation and become able to use all implemented classes interchangeably.

We have defined queries that return the classes that declare enough (by enough, we mean over user-specific threshold) same signature fields and/or methods. For some of the returned classes, the number of same signature fields and/or methods remains the same or changes a little, which indicates that these classes share the stable common interfaces. Among them, we identified instances of “Extract Interface” in order to comply with programming-to-interface principle.

For example, classes `HorizontalBarRenderer` and `VerticalBarRenderer` declared five same name fields and methods before version 0.8.0, while in that version, a common interface `CategoryItemRenderer` was extracted; the above classes were modified to implement the new interface and the field `renderers` of class `HorizontalCategoryPlot` and `VerticalCategoryPlot` were pulled up into superclass `CategoryPlot` that declared it as the type of interface `CategoryItemRenderer` instead of specialized `HorizontalBarRenderer` or `VerticalBarRenderer`.

We also discovered some other cases of interface extraction. In version 0.7.4, seven constants of the `Axis` class were extracted into the newly added interface `AxisConstants` that was then implemented by `Axis` and its subclasses. Similarly, `JFreeChartConstants` was extracted from `JFreeChart`, `ChartPanelConstants` from `ChartPanel` and `CategoryPlotConstants` from `CategoryPlot`. The intent for all these changes must have been to enable the use of the constants by classes other than their original containers. However, further development, it turned out that these constants were only accessed by the classes that originally contained them and the developers decided to move them back from the interfaces to the corresponding classes and remove the corresponding interfaces. We call such classes, whose features remain in the system even after the classes themselves are removed, die-hard classes [20].

#### 4.2.2 Forming template methods

The template method design pattern [8] is applicable in situations where an algorithm is defined in a superclass, with its overall process and some of its steps being shared by the subclasses as-is, some of the steps being used as defaults when the subclasses do not override them while yet others being overridden or extended by the various subclasses according to their needs. “Form template method” (see Figure 1) is one of the complex refactorings identified in the Fowler catalog [6] to get template method pattern.

We have defined queries that search for the classes that declare methods with enough (again, user-specific threshold) same field reads/writes and method calls, which indicates that these methods do their job in a similar way. Therefore, they are the candidates for further examination of forming template methods.

For example, in version 0.9.19, an superclass `AbstractCategoryItemLabelGenerator` was extracted from class `StandardCategoryItemLabelGenerator`, in which `generateLabelString()` was defined as a template method that called the default tooltip and label implementation defined
in method `createItemArray()`. The subclasses, StandardCategoryLabelGenerator and StandardCategoryToolTipGenerator, implemented the interfaces CategoryLabelGenerator and CategoryToolTipGenerator respectively and called `generateLabelString()`. The other subclasses, such as IntervalCategoryLabelGenerator, overrode `createItemArray()` to provide their specific behaviour.

The class `StandardPietitemLabelGenerator` had the similar condition to StandardCategoryItemLabelGenerator. If the JFreeChart developers wanted to restructure StandardPietitemLabelGenerator later on, the changes made to StandardCategoryItemLabelGenerator as reported by UMLDiff constituted the contextual advice on how to accomplish the task.

### 4.2.3 Extracting superclasses

The “Extract Superclass” refactoring is advisable when two (or more) classes share a substantial part of their behaviors, which also seem to be modified together over time. We ask JDEvAn again for the instances of classes that share the enough same name fields and/or methods. But this time we are more interested in those classes that show the similar evolution profiles [20], such as for example, the same name fields and/or methods are often added to those classes in the same version, which results in the number of same features increasing over time. This is a good indicator ofshot-gun surgery [6], which can be fixed by such refactorings as “Extract Superclass”.

For example, before version 0.9.9, there are three classes, `PaintTable`, `StrokeTable` and `ShapeTable` that share the same interface and behavior. The only difference among them is that they dealt with `Paint`, `Stroke`, and `Shape` objects respectively. In version 0.9.9 the JFreeChart developers were faced with the need to introduce three more similar classes: `FontTable`, `NumberTable` and `TextAnchorTable`. At this point, however, they must have noticed the commonalities between them and the three existing `Table` classes. Thus, instead of duplicating the existing code, they extracted the `ObjectTable` superclass and made all `Table` classes extend it, overriding the default behavior when necessary.

### 4.2.4 Extracting classes

Complex classes are sometimes incohesive because they are responsible for delivering many responsibilities. Such classes should be simplified by extracting some of their features into other classes, created for exactly that purpose. The simplified class can then delegate to the newly created class to deliver its responsibilities.

For example, in version 0.9.14, a new class `RendererState` was created. The field info of type `PlotRendererInfo` and the method `getInfo()` were extracted from the `AbstractRenderer` to the `RendererState` class. A similar refactoring was also applied to `Axis` to extract a new `AxisState` class. Such state classes were designed to hold state information for `renderer` and `axis` objects during the drawing process, which enable multiple threads to draw the same axis to different targets, since each drawing thread maintains its own separate state object.

The symptom motivating this refactoring is high class complexity, which is directly analogous to the number of facts extracted by the JDEvAn parser. In such cases, querying the JDEvAn database for the method sets used by the incohesive class clients may reveal subsets of methods used together which are candidates to become methods of a new extracted class.

### 4.2.5 Collapsing Hierarchies

Collapsing hierarchies is another important refactoring that deals with generalization. Refactoring hierarchies often involves moving fields and methods or pulling them up into a newly added or an existing superclass, which, frequently, results in the classes that do little job or subclasses that are not that different from its superclass. In such cases, the (sub)classes should be merge to superclass.

For example, in version 0.9.9 JFreeChart was overhauled substantially. UMLDiff reported by far the largest number of changes to the system design between any two subsequent versions.

Several inheritance hierarchies were collapsed: `HorizontalCategoryAxis` and `VerticalCategoryAxis` were removed and most of their members were pulled up into their common superclass `CategoryAxis`; `HorizontalDateAxis` and `VerticalDateAxis` were removed and their superclass `DateAxis` received most of their members; `HorizontalNumberAxis` and `VerticalNumberAxis` and their superclass `NumberAxis` were also analogously modified; finally, `HorizontalBarRenderer` and `VerticalBarRenderer` were removed and left some of their members into their superclass `BarRenderer`.

Class `VerticalLogarithmicAxis` was renamed to `LogarithmicAxis`, and `VerticalLogarithmicAxis` was removed. The similar changes were made to several pairs of classes: `Horizontal-`, `Vertical-` and `Stacked-` symbolic axes, `Horizontal-` and `Vertical-` number axes, `Horizontal-` and `Vertical-` interval bar renderers, and `Horizontal-` and `Vertical-` statistics bar renderers. These classes were direct subclasses of the corresponding horizontal or vertical classes discussed in the above paragraph before version 0.9.9, which were all removed from system. Thus, in version 0.9.9, they collapsed to extend `NumberAxis` and `CategoryAxis`, and `BarRenderer` respectively.

`HorizontalColorBarAxis` was renamed to `ColorBar`, and `VerticalColorBarAxis`, `Horizontal-`, and `Vertical-` logarithmic `ColorBarAxis` were all removed (see the detailed discussion on `ColorBar` in section 4.1.3). Finally, `HorizontalMarkerAxis` was renamed to `MarkerAxisBand` (there is no `VerticalMarkerAxisBand`) to keep the naming convention consistent.
All these recovered design changes indicate that the substantial changes were made to Plot, Axis, and Renderer classes in version 0.9.9. It seems like the JFreeChart developers made a great effort in that release to reorganize the Plot, Axis, and Renderer hierarchies and to eliminate the parallel inheritance. This observation is validated by the change log file shipped with release 0.9.9.

Actually, there exist three parallel inheritance hierarchies (Horizontal- and Vertical-plot, axis, and renderer) in the JFreeChart system before version 0.9.9. The horizontal class and its corresponding vertical one are very similar (or sometimes identical). The only major difference is, one set was used for horizontal drawing, the other for vertical. Such parallel hierarchies make the subsequent changes difficult, since when it comes time to modify something, you have to change more than one place. This also results in a large amount of code duplication. The JFreeChart developers became, at some point, aware of the existence of parallel inheritance. This observation is validated by the change log file shipped with release 0.9.9. They made a great effort to redevelop the Plot, Axis, and Renderer hierarchies.

Parallel hierarchies, symptomatic of strong design interdependencies, are discovered in two ways by JDEvAn: first, through their parallel evolution history [20] and second by querying the various branches with high-degree of similarity.

4.2.6 Replacing type code with typesafe-enum object

Numeric type codes are a common feature of procedural programming languages like C. Frequently, they are assigned as values to named constants to make them more readable. However, the compiler still sees the underlying number and, for optimization purposes, it may alias it to any other number with no restrictions to its value range. In this case, there is nothing to force the named constants to be used; any arbitrary nonsense number can be passed in. A better alternative in object-oriented software is the typesafe-enum class [1]. The idea is to replace numeric type code with a class with private constructors; use factory methods to make sure only valid instances are created and passed around.

Monitoring the use of type code may depend on dynamic analysis, but a simple query to JDEvAn’s database that returns all the constant fields of type int that are declared as static and final provides the developers a good start point to investigate the type code fields.

For example, in the case of JFreeChart, before version 0.9.9, the interface AxisConstants defined four constant int fields BOTTOM, TOP, RIGHT, LEFT, which were used to determine the axis location when drawing axis on plot. A new class AxisLocation was added to system in version 0.9.9. The above four constant fields were recognized by UMLDiff as moving from interface AxisConstants to the new class AxisLocation. However, the type of these four fields changed from int to AxisLocation. At the same time, the relevant method parameters and field data types, such as the int parameter of the Axis.draw() method and the int domainAxisLocation field of the ?Plot classes, changed in version 0.9.9 to be declared as of type AxisLocation. AxisLocation is an instance of the typesafe-enum pattern. It was introduced to replace type an int code with a typesafe-enum object. The same type refactoring was applied several times to produce such typesafe-enum classes as HistogramType, RangeType, HorizontalAlignment and VerticalAlignment, etc.

4.3 Contextual programming style hints

In this section, we discuss some of JFreeChart specific design changes that could be valuable as contextual evolution knowledge, if properly recorded.

4.3.1 Splitting a package

We investigated in detail the three package-splitting instances in JFreeChart. In version 0.9.4, the package com.jrefinery.chart contained 111 classes and interfaces. In version 0.9.5, 75 of them were moved into three new packages com.jrefinery.chart.plot, com.jrefinery.chart.axis, and com.jrefinery.chart.renderer. In version 0.9.7, 17 classes and interfaces were split out from package com.jrefinery.chart.data to a new package com.jrefinery.chart.data.time, but over 90 classes and interfaces were still left in it, until it was split again in version 0.9.21 to 8 new or existing packages. In version 0.9.21, 47 of 62 classes and interfaces were split out from org.jfree.chart.renderer to org.jfree.chart.renderer.category and org.jfree.chart.renderer.xy respectively.

These package-splitting activities generally reduced the number of classes and interfaces contained in each package to about 20-30. This project-specific behavior essentially constitutes a piece of contextual advice – although by no means definitive – that indeed, as a general rule:

- It is ok for a package to have ~30 classes and interfaces;
- A package should be split into subpackages when it has ~100 classes and interfaces;
- Increasing the size of a package over 60 can be flagged as a potential problem, because it reaches the range of complexity that makes it a splitting candidate.

4.3.2 More contextual advice

Due to the space limit, we can not get into details of more instances of project-specific advices that JDEvAn recorded by analyzing the evolution history of JFreeChart, but we list some of them briefly as follows:

- Avoid introducing parallel horizontal and vertical Plot, Axis, or Renderer classes when working on these hierarchies, since a great effort was made in the past to eliminate them;
- If the constant fields are only used by a single class, do not separate them out;
- When adding new type of plot class, let it handle its own corresponding dataset. Avoid putting the dataset
in the superclass Plot and let the subclasses do the downcasting:

- New classes should implement interface Serializable and Cloneable;
- Two sets of methods should be provided in the dataset classes. One set for efficient access, the other for convenience.

5 Conclusion and future work
JDEvAn is a tool, seamlessly integrated within the Eclipse IDE, that provides a wide range of design-understanding functionalities. These functionalities are designed to support developers in their design- and code-evolution decision making. As our JFreeChart case study demonstrates, JDEvAn
- recovers a logical-design model of the system from its source code;
- identifies changes that the design entities of the system have suffered since its last release using UMLDiff;
- and recognizes patterns in the current design of the system, such as parallel inheritance, template method; and in the evolution history of the system and its individual design entities, such as the recurring similar design changes, refactorings and class co-evolution.

Based on its multi-perspective understanding of the system design and its evolution, JDEvAn offers advice regarding potential modifications that may improve the system design. This advice is grounded in JDEvAn’s knowledge of object-oriented principles, design and refactoring patterns and programming-style hints previously adopted by the system under inspection.

To our knowledge JDEvAn’s ability to use its design-understanding towards such broad-scope design mentoring is novel. We are currently working on more extensive and more quantitative evaluation of JDEvAn’s design-mentoring capabilities.

References
8. E. Gamma, R. Helm, R. Johnson and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.