Understanding the Use of Inheritance with Visual Patterns

Accepted to ESEM'09

Simon Denier
RMOD Project-Team
INRIA Lille-Nord Europe, France
Simon.Denier@inria.fr

Houari Sahraoui
Geodes Team, DIRO
Université de Montréal, Canada
sahraouh@iro.umontreal.ca

Abstract

The goal of this work is to visualize inheritance in object-oriented programs to help its comprehension. We propose a single, compact view of all class hierarchies at once using a custom Sunburst layout. It enables to quickly discover interesting facts across classes while preserving the essential relationship between parent and children classes. We explain how standard inheritance metrics are mapped into our visualization. Additionally, we define a new metric characterizing similar children classes. Using these metrics and the proposed layout, a set of common visual patterns is derived. These patterns allow the programmer to quickly understand how inheritance is used and provide answers to some essential questions when performing program comprehension tasks. Our approach is evaluated through a case study that involves examples from large programs, demonstrating its scalability.

1. Introduction

Inheritance in object-oriented programming is an important mechanism with multiple usage. It enables subtyping [1], polymorphism [5], and code reuse [18]. In the past decades, inheritance has been put forward [4] as a means to enhance reusability, to provide specialization and generalization, and to define hierarchical taxonomies of real world objects.

Confronted with the task of understanding an object-oriented program, it is useful to quickly identify the different parts of the program and their use of inheritance. The range of this task can be illustrated by the following questions, which one could start answering to get the big picture of inheritance in a program. How many class hierarchies are there? How many classes are involved? How deep and how large can they be? What are the important classes in each hierarchy? What is the relationship between classes in a hierarchy: how do they behave and how similar are they?

Yet it is difficult to understand at first glance the usage of inheritance in any given class hierarchy. There is no single fashion to characterize the usage of inheritance [15, 2, 16]. Two properties of inheritance makes it hard. Inheritance is transitive, so one must have a look at the whole hierarchy to understand a part of it. Inheritance is oblivious, so it is easy to bypass the relationship of a class with its parent and its children.

Our goal is to improve the initial understanding of a program by providing a large overview of class hierarchies and of their main properties. We propose a suite of metrics and properties to characterize inheritance among its main properties. We use software visualization to compact the large amount of data into a manageable space. Specifically we use the VERSO framework for visualization: VERSO allows one to interactively browse entities organized in a layout, showing its scalability on large programs [14].

Our contribution includes: a custom Sunburst layout preserving a visual link between parents and children; a mapping of standard inheritance metrics and characteristics onto visual properties; a new metric to characterize the similarity between children, bringing a new dimension in inheritance characterization; visual patterns of interest using this visualization.

We present three case studies corresponding to three programs with different sizes. We show how our representation scales up. For each program, we analyze its representation to characterize its inheritance usage, highlight particular hierarchies using visual patterns, and identify some singularities which pop up.

Section 2 discusses related work. Section 3 introduces the VERSO visualization framework, while Section 4 presents the adaptation we made in VERSO for inheritance visualization. Section 5 describes metrics used to characterize inheritance and their mapping on visual properties. Section 6 defines interesting visual patterns and Section 7 illustrates our approach on three well-known programs. Section 8 discusses future work and concludes.

It is advised to view an electronic version of this article.

Figures are best understood with colors and can be zoomed in for full details.

2. Related Work

Usual visualization for inheritance is based on graph layout where various node shapes represent classes and some data while edges stand for relationship between parent and subclasses. Typical graph layout algorithms will optimize the positions of parent nodes and children nodes following a particular constraint, like preserving the hierarchical order or minimizing length or crossing of edges [7]. A problem with graph layouts is the visualization in extreme cases, like nesting of multiple levels of nodes, or a large number of children nodes. Yet those cases are often the most interesting in inheritance and all too common. Depending on the layout, the visualization often displays edge overlapping, or nodes pushed too far apart, which reduce readability. Space-filling layouts suffer much less from the positional constraint of graph layouts. Instead, they define custom regions in space in which positions are less constrained, allowing for more compact layouts.

Different visualization frameworks have been proposed to display metrics instead of raw and transformed data extracted from the source code. Holten et al. [10] use *Treemaps* with the help of bump maps and textures to display two metrics at the same time at the method level. In the same context, Balzer and Deussen [3] developed another layout technique based on the *Treemap* and Voronoi tessellation to display metrics associated to a color code. Similarly, Graham et al. [8] exploit the solar system metaphor to display metrics of classes and packages. On the other side, Lange et al. [13] enrich UML diagrams with visualization technics so as to keep a traditional frame of reference, well known by software developers.

Lanza and Ducasse [15] presents Polymetric Views, a generic and multi-purpose software visualization technique, developed for reverse engineering. Each Polymetric View is customized for a specific task by combining the following elements: a layout, a set of entities to be represented, a set of metrics to be mapped onto visual characteristics of the entities. In particular, Lanza and Ducasse define three Polymetric Views dedicated to inheritance: System Complexity, Inheritance Classification, and Inheritance Carrier. All three uses a classical tree layout, which compacts all children at the same level on the same line: as a consequence, children are not aligned vertically with their parent, which makes it difficult to distinguish the extent of families and hierarchies. The three views together convey many details on inheritance, such as the Number Of Attributes or the Number of overRiden Methods. However, each one individually does not display as much. Three different views means three different mappings of metrics to visual characteristics, as well as specific sets of patterns, which makes the learning curve substantial. Besides, there is no visual mean to follow a class between Polymetric Views, which makes the task of understanding inheritance more cumbersome.

Stasko and Zhang [17] introduces the Sunburst layout, a radial space-filling technique, as an alternative to Treemap. Their application was visualization of large filesystems. Stasko and Zhang perform a study to compare Treemap and Sunburst in terms of task operation. The Sunburst layout appears to offer a better initial intuition of a hierarchical structure.

3. An Overview of VERSO

Software artifacts, being abstract notions, do not have concrete visual shapes. A common strategy in software visualization defines a mapping between design/programming artifacts and concrete graphical elements. This process is usually known in the literature as the application of a metaphor [12]. In this context, our framework uses 3D graphical elements distributed over a 2D plane to represent an object-oriented program. This 2D-3D compromise offers the best of both worlds. Navigation is intuitive because the analyst always looks toward the plane and occlusion is greatly reduced compared to a fully 3D layout. Moreover, the third dimension extends the potential characteristics with which one can display metrics. The problem of size estimation of 3D objects with a perspective camera is less of an issue in our configuration because 3D objects raising from a 2D plane are very similar to real-world objects. Moreover, according to Healey and Enns [9], humans correctly interpret the sizes and distances of objects in perspective views.

3.1. Visual Representation of Classes

The graphical representation for a basic element of an OO program (class) is a 3D box. A 3D box has a number of interesting features: it is simple, it is familiar for human perception and analysis, and it possesses a number of non-interfering characteristics such as color, twist, height, etc. (see [9]).

Class properties, captured by metrics, are mapped to graphical attributes of the 3D box : height, color ranging from blue to red, and twist ranging from 0 to 90 degrees (starting from the vertical). The mapping depends on the analysis to be performed. In the context of inheritance understanding, the mapped metrics are those described in section 5.

3.2. Layouts

The visualization framework VERSO proposes two layout algorithms for the 3D boxes, discrete versions of *Treemap* [11] and *Sunburst* [17]. Both algorithm place boxes following a hierarchical organization by dividing a plan into regions. More specifically, classes (3D boxes) are placed according to their logical architecture (packages, sub-packages and classes). Packages are not associated with graphical objects. They are represented by regions of the plan. More details on the two different layout algorithms as well as on the framework in general can be found in [14].

For our work, we decided to reuse VERSO with the Sunburst layout but with the hierarchy relationship instead of the package inclusion relationship. Both relationships define hierarchical organization. The difference, however, is that for inheritance, all the levels represent classes. Unlike packages, parent nodes (super-classes) must be rendered in the same manner as child nodes. Our adaptation to Sunburst/VERSO is detailed in Section 4.

3.3. User Interactions

The user controls a VERSO visualization interactively by navigating along multiple dimensions: position, rotation, zoom. It allows the programmer to change its point of view, avoid occlusions, and focus on a particular region of the layout for more details. The user can also request more information about a particular entity such as class name, properties, or source code. A range of filters representing relationships between classes (children, descendants, uses, used_by, etc.) is available to selectively highlight entities in relationships without modifying the layout.

4. Adaptation to Inheritance Visualisation

To consider the inheritance relationship, we adapt the layout algorithm Sunburst/VERSO that was defined primarily to represent the logical package architecture of object-oriented programs. The basic idea is to divide the plan into regions representing the class hierarchies. Each slice will contain the subclasses of a parent class. We focus on single inheritance and, in the case of Java, do not look into Java interfaces.

4.1 Definitions

Before presenting the adaptation details, let us start by giving some definitions of the inheritance notion which will be visualized.

Parent class a class which has subclasses;

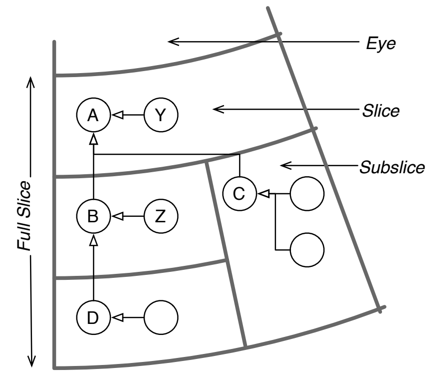

Figure 1. Mapping a Class Hierarchy in a Sunburst Layout

Childless class a class which has no subclasses;

Family the set of direct subclasses of a parent, including childless and parent subclasses;

Hierarchy the full set of subclasses of a parent, including children of subclasses.

Hierarchy root the parent class of a hierarchy, whose parent is `Object`.

Ghost class a (super) class which is defined outside the project, such as in a library or a framework.

4.2 Sunburst for Inheritance

The Sunburst layout is a radial space-filling layout which can be tailored to display the different levels of a hierarchy. According to Stasko [17], the Sunburst layout offers a better intuitive overview of a hierarchy, compared to a Treemap.

Figure 1 shows the correspondence between a classic tree layout and the regions of our custom Sunburst layout. In addition, Figure 2 shows the full sample for the same hierarchy as it appears in VERSO. We introduce a visual vocabulary to describe our adaptation then detail the rationale for the positioning of classes in regions.

Slice a part of the visualization delimited by edges;

Subslice a slice which resides on the outer edge of the current slice;

Full slice a slice and all its subslices down to periphery;

Eye the innermost slice of the visualization.

As a first interpretation, visual entities represent classes and reside in slices; a slice represents a family and a full

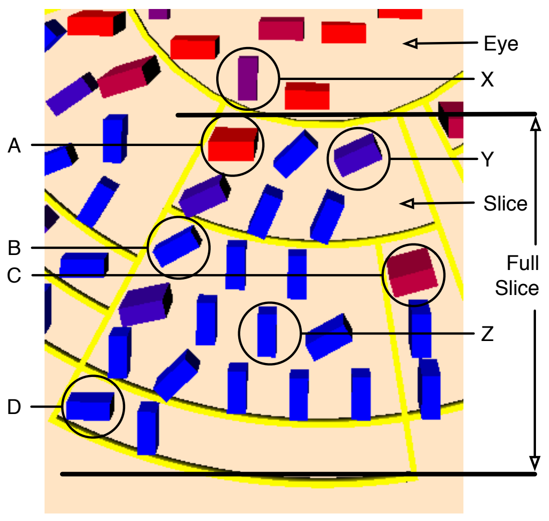

Figure 2. Sample Details

slice, a hierarchy. The Sunburst layout gives a visual account of how many entities reside per slice as well as how many subhierarchies stem from the slice parent.

The inner circle contains direct subclasses of `Object` without children. Thus the eye of the layout shows classes which are not interested in inheritance, for example class X in Figure 2.

An intuitive choice for the Sunburst layout is to place each class in the slice of its parent. As a consequence, a parent class should be placed in the slice of its own parent. However, looking at an arbitrary slice, it becomes difficult to track its parent in the upper slice as it is mixed with other siblings. This makes it impossible to look for all parent classes at once. Yet it is interesting to compare the properties of a parent with that of its children as well as to compare parents between them.

We resolve this problem by having a different position rule for parent classes than for childless classes. We push down parent classes with their own childless subclasses and assign them a special position to be able to identify them visually among their children. The parent class of a family always sits in the inner (i.e. near center) clockwise corner of its slice.

The discrimination rule is:

- if an entity sits in the inner clockwise corner, it is the parent class of the slice (classes A, B, C, and D in Figure 1);
- otherwise it is a childless subclass in the slice (classes X, Y, and Z in Figure 1).

This choice implies that to have a complete picture of a family, one should look at the slice as well as all parents of the subslices, since they are pushed down.

Inheritance of classes outside the project, such as classes in library or framework, is important to identify as it has a deep impact on the classes. Yet we do not want such classes, which can be arbitrarily complex, to disturb the visual characteristics of program classes. We call ghost classes such classes outside the scope of the program, yet still referred to (transitively) through inheritance. Since our approach deals only with single inheritance and that any class has `Object` as its root, ghost classes fit nicely in the Sunburst layout as any other classes.

We choose to display ghost classes in order to preserve the structure of the Sunburst layout. Yet we use a dedicated visual entity, which conveys no information about its class, to not disturb program classes with external characteristics. Ghost entities all use the same shape, a cylinder of small standard height and of purple color. There is no visual distinction except for their position in the layout. By construction, all visible ghost classes are parents of some classes in the program. Figure 3 shows many ghost classes in its top part.

5. Mapping Class Properties on Visual Entities

Each class can be characterized with respect to inheritance by a small set of properties, including metrics values and symbolic information. We detail the different metrics and information used as well as how the properties of each class are mapped on characteristics of its visual entity. We are currently able to display five properties per class.

5.1. Basic Metrics

In [6], two basic metrics are proposed to characterize classes with respect to inheritance: Depth of Inheritance Tree and Number Of Children. We use another common class metric which is the Number Of Methods.

DIT is the Depth of Inheritance Tree, i.e. the number of parent classes up to the root. The greater DIT is, the farther the class is in the hierarchy: then it is more specialized but can also be more complex because of the chain of inheritance.

In the Sunburst layout, DIT is the number of slices up to the inner circle, including the starting slice. There is an exception to the rule for parent classes, which are push down with their children in their slice, so their DIT is the number of slices minus one. Thus a class in the inner circle has a DIT of 1, except for `Object`.

In Figure 2, the hierarchy of class A has a maximum DIT of four. Class B, which is a parent, has $DIT = 2$ while its child Z has $DIT = 3$.

NOC is the Number Of Children of a class. The greater NOC is, the more responsibility the class has because of all its subclasses. However, the majority of classes in a program has no child: it is also interesting to see those childless classes where the hierarchy stops.

Each slice in the visualization displays a family with its parent and its childless classes. Adding to that each subslice counts as one child, since their own parent is a child in the slice. This allows one to quickly weigh the difference between the number of children without child and children which are themselves parent.

In Figure 2, class A has five childless subclasses and two parent subclasses (B and C), thus $NOC = 7$.

A class having only one child can have multiple grandchildren through this child. NOC is computed on a single slice of the hierarchy and can not account for such a case. On the contrary, the display of children in concentric slices can give a visual account of the number of children at different levels.

NOM is the Number Of Methods in the class. The greater NOM is, the more services the class can offer to the program, the more interesting it is to investigate for itself. In the context of inheritance, a parent class with a high NOM define many methods, which subclasses then inherit and can reuse.

We map NOM measure of a class to the height of its visual entity: the greater NOM is, the taller the entity. A class with an unusual NOM compared to its neighbourhood will stand out because of the difference of height, making it easy to spot and a target for investigation.

In Figure 2, classes A, C, and the child under B are definitely taller than their neighbourhood, although the perspective of the screenshot flattens the heights.

5.2. Subclassing Behavior

The hierarchical relationship between a class and its parent can be characterized by its subclassing behavior, i.e. whether the class has a tendency to add new methods or to override methods from its parents. We define a nominal scale with five categories, from a *pure extender* class which only adds methods, to a *pure overrider* class which only overrides methods. We map this scale to a colour scale of five colours (Table 1).

5.3. Children Similarity

In addition to the hierarchical characterization of a class with respect to its parent, we define a new "horizontal" characterisation of a class with respect to its siblings, i.e. the children of the same parent. Knowing that a class shares strong similarity with its siblings is a good indication that the family can be understood as a whole. Each child defines a small variation around common characteristics, for example by defining the same set of methods. On the contrary dissimilar children is an indication that each child defines its own behavior and must be understood independently from the others.

We define a new measure of similarity between siblings by comparing their interface. The more method signatures they have in common, the more similar they are. More specifically, the computation of the similarity metric is a two-step process.

1. A prototype of common interface is computed with all interfaces of siblings.
2. For each sibling, the similarity of its interface to the prototype is computed.

The computation of the interface prototype is a unique step. It uses a majority rule to select methods from the children interface. To appear in the prototype, a method must be declared by more than the half of children. Table 2 illustrates this rule on a simple example with three classes and five methods.

Class	m_1	m_2	m_3	m_4	m_5	sim
C_1	x	x	x	x		$2/5$
C_2		x	x		x	1
C_3		x			x	$2/3$
Prototype		x	x		x	-

Table 2. Prototype building by majority rule. x means class C_i declares method m_j. Last column shows resulting similarity measures.

The similarity metric for each child is defined as the Jaccard index between the interface of the child and the interface prototype. Considering that an interface is a set of methods, the formula for the Jaccard index between interfaces i_1 and i_2 is:

$$\begin{aligned} sim(i_1, i_2) &= \frac{|i_1 \cap i_2|}{|i_1 \cup i_2|} \\ &= 1, \text{ if } i_1 = i_2 = \emptyset \end{aligned}$$

We map the similarity measure of a class to the twist of the visual entity. Similar children have a null, vertical twist. Dissimilar children are twisted towards the horizontal axis.

In Figure 2, the family of class B includes many similar children, such as Z. On the contrary, children of class A show a medium similarity between them. Class A is completely dissimilar, which is not surprising given that there is no similarity between the many subclasses of `Object`.

Category	Definition	Colour	Sample Class
Pure Extender	Adding only new methods	Bright red	A
Extender	Adding more methods than overriding	Red	C
Other	No method defined	Purple	X
Overrider	Overriding more methods than adding	Blue	Y
Pure Overrider	Only overriding methods	Bright blue	B, D, Z

Table 1. Subclassing behaviors and mapping to colour scale (see sample in Figure 2).

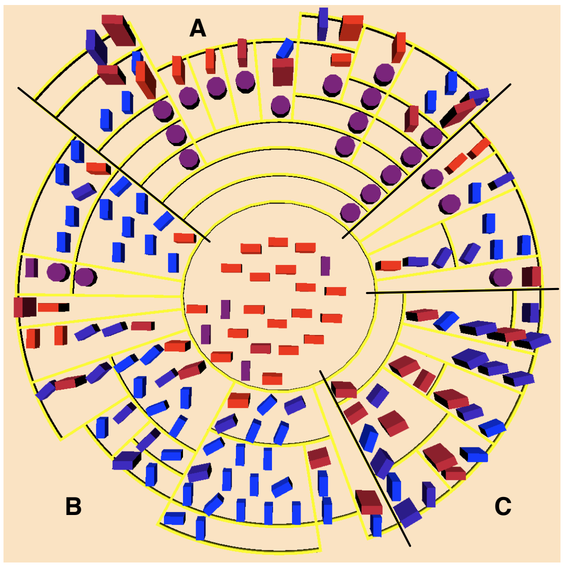

Figure 3. Overview of JHotDraw

5.4. First Example

Figure 3 shows the VERSO visualization of JHotDraw 5.2. The 150 classes are all organized according to their inheritance hierarchies in a compact view. This overview allows one to distinguish the extension of the Swing framework in JHotDraw (slice *A*), the high use of overriding for polymorphism in three class hierarchies (slices in *B* quarter with many blue entities) as well as the main hierarchy of `Figure` with many large classes (slice *C*).

6. Visual Patterns

We identify visual patterns of interest. Such patterns stand out in the visualization as a particular combination of visual properties. They are interesting because their discovery imply interpretation or investigation of class hierarchies.

In this section we only present common, recurrent patterns of general interpretation. Next section will provide illustrations of such patterns on real examples as well as particular patterns unique to their context.

Name: Large root.

Visual properties: Root slice with a large radius.

Interpretation: This pattern reveals a root class which has many children or many grandchildren in subhierarchies. This is a basic pattern which can be spotted in the Framework Extension and Polymorphic Hierarchy patterns.

Name: Big Hierarchy.

Visual properties: Full slice with deeply nested slices and slices with many entities or subslices.

Interpretation: A big hierarchy is a basic pattern easy to spot due to its relative size. As it involves many classes, it is often an important part of the program.

Name: Common Family.

Visual properties: Red parent entity and mostly blue children.

Interpretation: This is a common case of family seen in visualization. It suggests a typical behavior where the parent adds new methods and where the children inherit such methods and override some to specialize their parent.

The Big Hierarchy pattern combined with the Common Family pattern often appear with variations, which we group under the name Polymorphic Family or Polymorphic Hierarchy. We describe those variations in the following patterns. They are strong signs that classes in the family or hierarchy share a common purpose with polymorphic adaptation. The more patterns are spotted together, the more polymorphic the family or hierarchy is.

Name: Large Family.

Visual properties: Slice with many entities, often childless.

Interpretation: Large Family with Common Family indicates that the parent defines a common behavior for multiple children.

Name: Similar Children.

Visual properties: Slice with many vertical entities.

Interpretation: This is a strong indication that children specialize the same set of methods, each adding its own variation. Similar children are often small blue classes, because they override the same small set of methods.

Name: Nested Families.

Visual properties: Nesting of previous patterns.

Interpretation: The polymorphic behaviour spans multiple levels of inheritance, an indication that the hierarchy defines a specialized taxonomy. In nested families, it is common to find blue parents.

We also identify patterns involving ghost classes. Those patterns point to part of the program with a special purpose related to external classes.

Name: Library/Framework Extension.

Visual properties: Full slice with nested ghost parents, parenting entities of mixed properties (i.e. red or blue, tall or small, twisted or not).

Interpretation: Identifying which part of the program inherit from external classes in libraries enable the identification of dependencies and their respective importance. If it appears as a Big Hierarchy, it involves a framework such as Swing: the presence of large root slices with ghost parents makes it easy to spot.

Name: Exception Hierarchy.

Visual properties: Full slice with nested ghost parents and a majority of purple entities.

Interpretation: Purple classes define no instance methods. The only reason to create hierarchies with void classes is to create a subtyping hierarchy. More than often such hierarchies are in fact derived from the `Throwable` class in Java to create various kinds of exceptions.

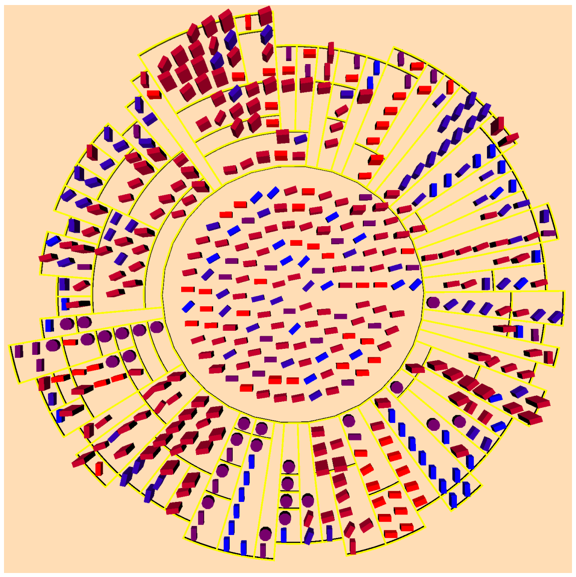

Figure 4. Overview of JFreeChart

7. Case Studies

The analysis of inheritance in a program using our adaptation of VERSO follows a three-step process. First, an overview of inheritance usage is provided which allows answering general questions: How many class hierarchies are there? How many classes are involved? How deep and how large can they be? Is there a general tendency in the behavior and similarity of classes? Then one can focus on particular hierarchies and classes and look for visual patterns as well as singular visual entities. For each identified case, it is possible to interactively request VERSO for the name, properties, and source code of interesting classes. This additional information allows to refine the analysis and to provide detailed reports.

In the remainder of this section, we describe occurrences of the previous patterns found in three systems of different sizes, as well as singularities of inheritance in those systems. Due to lack of space, we only focus on the most interesting visual features of each case.

7.1. JFreeChart

JFreeChart is a library for building and rendering charts. It contains around 500 classes in version 1.0.

The overview of JFreeChart in Figure 4 shows many slices, many of them being nested. There exists three big hierarchies, with large roots and deep nesting. However, nesting is shallow for most slices, involving many classes in small hierarchies. Many entities are of the red colour, which shows a mixed behaviour of adding methods in subclasses while still overriding some. This overview indicates a fair use of inheritance to build classes in layers, both extending and specializing their parents. It means for the programmer that the resulting interaction between the multiple levels of inheritance can be difficult to understand.

There are three big hierarchies (Figure 5). However, none of them exhibits a polymorphic pattern. Entities display mixed properties, red or blue, tall or small, twisted or not. So each class derives its hierarchy for reuse and extension. Two big hierarchies are particularly interesting by their size: `AbstractDataSet` (slice *A* in Figure 5) and `AbstractRenderer` (slice *B*) define the main domain objects in a JFreeChart project. The fact that there is many tall classes in `AbstractDataSet` hierarchy reinforces its importance. The third big hierarchy (slice *C*) is an occurrence of Framework Extension involving Swing.

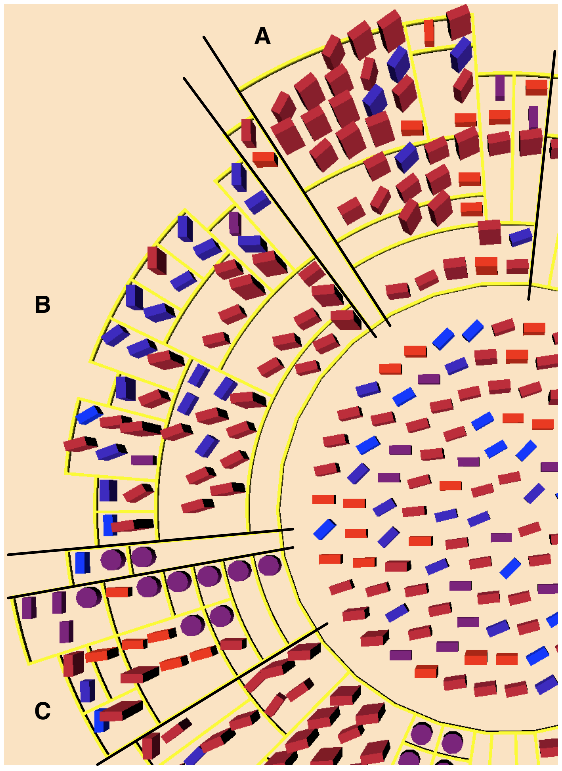

Figure 5. Three Big Hierarchies in JFreeChart

A singular feature of the JFreeChart overview is the horizontal twist of purple entities in the eye (seen in Figures 4 and 5). Indeed, such classes are normally vertical as they perfectly match with the empty prototype of the eye. This implies that the eye prototype is not empty. Code examination confirms that most eye classes override the `equals(Object)` method.

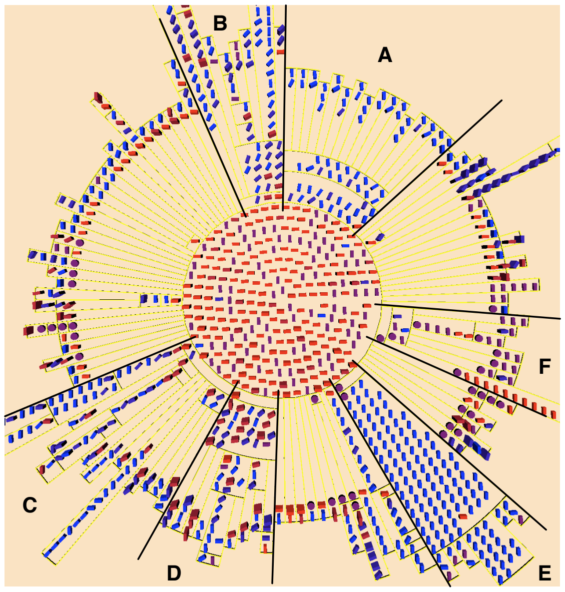

Figure 6. Overview of Xalan (cropped for readability)

7.2. Xalan

Xalan is an XSLT processor for transforming XML documents into HTML, text, or other XML document types. It contains around $1,000$ classes in version 2.7.

The overview of Xalan classes and hierarchies given in Figure 6 shows many slices. The eye looks comparatively small. This overview tells that many classes in Xalan are involved in inheritance. Some slices are deeply nested and large root slices are visible, which indicates the presence of big hierarchies of classes. Entities in slices are mostly of the bright blue color, so classes are mostly of the overriding behaviour. Overall, this shows that Xalan makes a large use of inheritance in a specialization manner. Thus, the programmer can focus its understanding effort on the parent classes, which define the common behavior, and on how children override their parent.

The most visually interesting parts are the six big hierarchies spotted with the large root pattern (Figure 6, slices *A* through *F*). They have strikingly different properties.

Two big hierarchies, `DTMAxisIteratorBase` (slice *A* in Figure 7) and `ResourceBundle` (slice *E*), show strong polymorphic patterns with bright blue, large, nested families with many similar children. The first one is obviously devoted to the definition of an hierarchy of iterators while the second one deals with internationalization support. The `DTMAxisIteratorBase` hierarchy is interesting in particular as it contains only blue classes (except for the root), and many subslices at the third level.

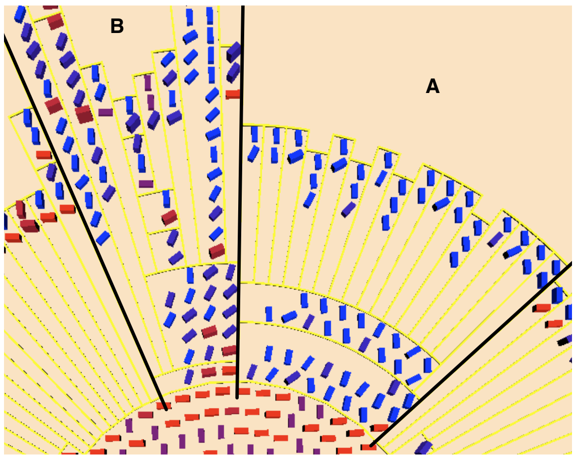

Figure 7. Detail of Xalan with two Polymorphic Hierarchies side by side

Three big hierarchies display mixed properties of polymorphic hierarchies, in decreasing order: `SyntaxTreeNode` (slice *B* in Figure 7), `Expression` (slice *C*), and `UnImplNode` (slice *D*) with very mixed properties and tall red entities. All three define domain classes of Xalan, respectively its AST classes, its interpreted language, and the template-based generator.

The last big hierarchy matches with the Exception Hierarchy pattern (slice *F*). A singular feature in this occurrence is the red family, yet many entities in the sibling slices are purple.

There is a few occurrences of nested single children, i.e. of a chain of slices with a single (non-ghost) entity in each slice. It means that each parent class is also the unique child of its own parent, defying the purpose of inheritance for managing multiple children. It reveals to be different specializations for graph traversal strategies.

7.3. Azureus

Azureus (nowadays Vuze) is an extensible peer-to-peer client, comprising around $1,600$ classes in version 2.4.

The overview of Azureus gives a very different picture of inheritance use compared to Xalan. The most striking feature of the view is the big eye, which implies that most classes in Azureus are not concerned by inheritance. With respect to the size of Azureus, there is few hierarchies and in particular few big hierarchies. Almost all hierarchies are shallow. Finally, blue and red entities are mixed between hierarchies. This overview shows that Azureus uses inheritance sparingly and for different purposes, mixing overriding and extension of superclasses. However, this picture does not account for interface implementation which might be used in place of inheritance.

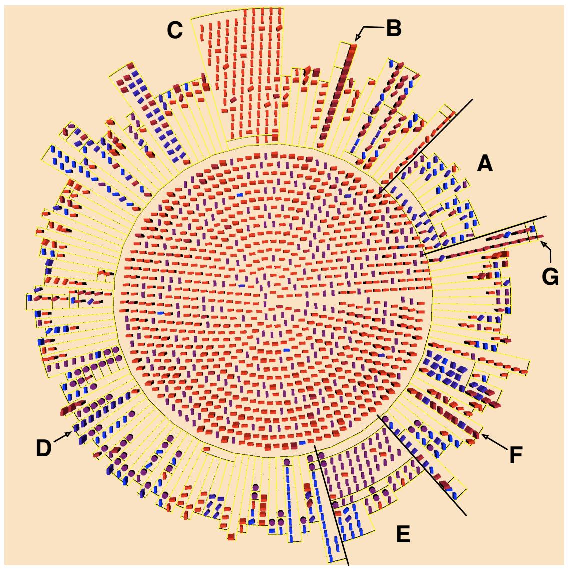

Figure 8. Overview of Azureus

One big hierarchy displaying polymorphic properties is that of `DERObject` (slice *A*) for encoding basic datatypes. The other big hierarchy is an occurrence of Exception Hierarchy pattern (slice *E*).

Azureus is still interesting because unusual patterns show up. The most striking pattern is the large red family beneath `TableColumnImpl` with high similarity of children (slice *C*). Such visual properties are a strong indication that the classes implement the same interface (`TableCellRefreshListener` in this case) independently of their parent.

Also two families display medium to tall classes with good similarity: family of `GeneralDigest` (slice *D*) implements different algorithms to compute message digests using the same interface. In the same manner, family of `ResourceDownloaderBaseImpl` (slice *F*) defines different strategies to download resources using a common interface. Thus they are occurrences of Polymorphic Families.

On the contrary, family of the small class `LogRelation` comprises tall red classes, strongly dissimilar (slice *B*). Examination shows that this family exists for reuse of common code in `LogRelation` to define. In the same manner, family of `RPObject` comprises mostly tall red classes, sharing a common ancestor for reuse (slice *G*).

8. Conclusion

We present a scalable inheritance visualization using VERSO. We are able to display many characteristics of inheritance: those account for the use of inheritance at program level as well as at class level. We achieve this result by specifying a custom Sunburst layout, able to visually preserve the link between parents and children. We also develop a new metric for similarity between children, bringing a new dimension to inheritance characterisation. The visual patterns we derive from our visualization highlight common usage of inheritance as well as important structures in program. We demonstrate the power of analysis of our approach on programs of varying size.

The major limitation of our approach is that it can only render single inheritance. Multiple inheritance, in this case Java interface implementation, can not be represented and is thus ignored. Yet interface implementation is a major feature of the Java programming language, taking in charge a large part of the subtyping function of inheritance. Many programs still rely on inheritance to define the major subtyping relationships, using Java interfaces to crosscut the main hierarchy. So our visualization can still give a good preliminary picture of subtyping. Yet we have seen with Azureus a different style in combining use of inheritance and interface implementation: Java interfaces are the primary means of specifying subtyping, inheritance being used more sparingly.

In some cases, the Sunburst layout can display improper effects, such as the deep slices bursting out of the periphery in the overview of Xalan (Figure 6). In large programs, the number of small hierarchies makes the visualization less readable without providing much information. However, such problems are relatively minor. In future work, we plan to enhance VERSO with interactive filtering of classes and hierarchies: it will allow the programmer to switch between a full overview and a focused view on a reduced set of hierarchies, where the recovered space will be used to optimize the layout.

The principles and techniques supporting the similarity metric are not tied to the visualization. We have already begun to explore the implementation, interpretation, and refinement of these techniques, as well as their application in different contexts.

Acknowledgements. This work has been partially funded by the Égide Lavoisier fund (France), NSERC, FQRNT, and CFI (Canada).

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, second edition, 1998.

[2] G. Arévalo, S. Ducasse, and O. Nierstrasz. Discovering unanticipated dependency schemas in class hierarchies. In *Proceedings of 9th European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 62–71. IEEE Computer Society, Mar. 2005.

[3] M. Balzer, O. Deussen, and C. Lewerentz. Voronoi treemaps for the visualization of software metrics. In *Proc. ACM Symposium on Software Visualization*, pages 165–172, 2005.

[4] R. Biddle and E. Tempero. Explaining inheritance: A code reusability perspective. *SIGCSE Bulletin*, 28(1):217–221, March 1996.

[5] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[6] S. R. Chidamber and C. F. Kemerer. A metric suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):293–318, June 1994.

[7] S. Diehl. *Software Visualization*. Springer-Verlag, Berlin Heidelberg, 2007.

[8] H. Graham, H. Y. Yang, and R. Berrigan. A solar system metaphor for 3D visualisation of object oriented software metrics. In *Proc. Australasian Symposium on Information Visualisation*, pages 53–59, 2004.

[9] C. G. Healey and J. T. Enns. Large datasets at a glance: Combining textures and colors in scientific visualization. *IEEE Transactions on Visualization and Computer Graphics*, 5(2):145–167, 1999.

[10] D. Holten, R. Vliegen, and J. J. van Wijk. Visual realism for the visualization of software metrics. In *Proc. International Workshop on Visualizing Software for Understanding and Analysis*, pages 27–32, 2005.

[11] B. Johnson and B. Shneiderman. Treemaps: A space-filling approach to the visualization of hierarchical information structures. In *IEEE Visualization*, pages 284–291, October 1991.

[12] C. Knight and M. Munro. Virtual but visible software. In *Proc. International Conference on Information Visualisation*, pages 198–205, July 2000.

[13] C. Lange, M. Wijns, and M. Chaudron. Supporting task-oriented modeling using interactive uml views. *J. Vis. Lang. Comput.*, 18(4):399–419, 2007.

[14] G. Langelier, H. A. Sahraoui, and P. Poulin. Visualization-based analysis of quality for large-scale software systems. In *IEEE/ACM International Conference on Automated Software Engineering 2005*, pages 214–223, Nov. 2005.

[15] M. Lanza and S. Ducasse. Polymetric views – a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, September 2003.

[16] P. F. Mihancea. Towards a client driven characterization of class hierarchies. In J. Ebert and P. Linos, editors, *Proceedings of the* 14^{th} *International Conference on Program Comprehension*, pages 285–294. IEEE Computer Society Press, June 2006.

[17] J. Stasko and E. Zhang. Focus+context display and navigation techniques for enhancing radial, space-filling hierarchy visualizations. In *Proc. IEEE Symposium on Information Vizualization*, pages 57–65, 2000.

[18] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In J. Coplien, editor, *Proceedings of the* 11^{th} *conference on Object-Oriented Programming Systems, Languages and Applications*, pages 268–285. ACM Press, October 1996.