Visualising Class Cohesion with Virtual Worlds

Neville Churcher Warwick Irwin Ron Kriz

Software Visualistion Group, Department of Computer Science, University of Canterbury, Private Bag 4800, Christchurch, New Zealand

VT-CAVE, Virginia Tech, Blacksburg VA24061

E-mail: {neville,wal}@cosc.canterbury.ac.nz, rkriz@vt.edu

Abstract

An understanding of cohesion is an important factor in software design. However, cohesion is difficult to quantify, particularly for OO, and attempts to develop metrics have had limited success. We advocate the use of visualisation techniques to provide a richer view of cohesion than is possible with a single numeric value. In this paper we describe the application of ANGLE *for 3D graph layout and the use of XSLT transformations both to select the ingredients for visualisations and to determine their presentation details. We discuss our experiences with the use of virtual worlds as a presentation medium both on the desktop and in immersive environments and report early results from ongoing empirical work.*

Key words: software visualisation, cohesion, VRML, virtual reality, software engineering

1. Introduction

Decomposition is one of the oldest and most fundamental techniques employed in software engineering. It results in a "good" system design which consists of relatively independent components which only interact in order to achieve system goals.

During analysis we model a real-world problem by breaking it up into successively smaller, more manageable, sub-problems until these are small enough to be represented accurately in our preferred notation. The problem is then defined by the representations of the sub-problems and the relationships between them. Similarly, our design and subsequent implementation will also consist of a number of related parts.

Although the terms used to denote the parts (subsystems, modules, classes, components, ...) and relationships have changed over the years the benefits of a modular system have endured. The basic idea is simple: put things that belong together in one place. This leads to independent 'pluggable' modules which are easy to develop, maintain and replace and which present simple interfaces to their clients.

As always, there is a catch. There is no single correct way to perform decomposition and different choices can lead to different designs with very different properties. For example, techniques based on decomposition with respect to data flow tend to produce different results to those of techniques based on decomposition with respect to control.

Concepts such as information hiding [25] help address the problem of deciding what should be in a module. Good choices lead not only to 'better' modules but also to simpler relationships between them.

A related problem is assessing the degree to which each module's contents 'belong together' and the relative independence of modules. Cohesion is a measure of internal module strength while coupling describes the strength of inter-module relationships. They are not independent—coupling is lowered when communicating activities are located in the same module. Cohesion and coupling are an ingredient in many design techniques [29, 24].

The advent of object-oriented software development has brought many benefits but has added further complexity to the concepts of cohesion and coupling by increasing the number of available component and relationship types. For example, in conventional languages such as Pascal or C the dominant relationship is *procedure invokes procedure*. In OO languages such as Java or C++ components may be methods, classes or packages and relationships include inheritance and containment in addition to invocation.

Coupling and cohesion for OO systems have been discussed extensively [1, for example] and underlie design heuristics such as the Law of Demeter [20].

Only for very small systems is a subjective assessment of the levels of cohesion possible and sufficient. Measures which are more objective and amenable to automated collection are required for systems of realistic size. Cohesion is multi-faceted, involving complex interactions between

many components, and it is not possible to represent it adequately with quantitative metrics without significant loss of information.

In our experience, software visualisation is a better way to handle problem domains having many variables, complex interactions and large data volumes. Software visualisation involves applying information visualisation principles to the software engineering domain. In previous work [7, 14, 6, 18] we have explored the use of 3D virtual worlds to deliver visualisations to the desk top via VRML [5]. We are primarily interested in "just in time" visualisations, requested and configured under the control of the user, rather than professional quality visualisations requiring time and expertise not normally available.

In this paper we develop 3D graph-based visualisations which expose cohesion. The bindings which give rise to cohesion are edges and aspects of cohesion may be inferred directly from the graph layout. The ANGLE layout engine is used and XSLT transformations [19] describe the mappings from the software system to the graph model and from the graph model to the final visualisation.

The remainder of this paper is structured as follows. In the next section we discuss previous cohesion metrics. Our visualisation strategy is introduced in section 3 and its application to the visualisation of class cohesion is described in section 4. Some results are presented in section 5 and in section 6 we discuss the use of immersive environments to go beyond desk-top presentation. Finally, our conclusions and future directions are presented in section 7.

2. Measuring coupling and cohesion

Software metrics have been developed for measuring many aspects of software artifacts and processes [15, 13] including coupling and cohesion.

Measures of coupling typically involve inter-module quantities such as fan-in and fan-out while measures of cohesion are generally derived from intra-module quantities. Ideally, metrics should be developed in the context of a framework which embodies the conceptual model of the paradigm [9, 4, for example] but this has proved to be an elusive goal.

Yourdon and Constantine identified several kinds of cohesion and coupling and used numeric values to indicate the desirability of each [29]. Strictly speaking, this scale is ordinal even though the values are numeric. Cohesion measures based on program slices have been developed [23, for example].

The most commonly used suite of OO metrics, proposed by Chidamber and Kemerer, includes three relevant metrics: RFC (method invocation), CBO (data and object use) and LCOM (lack of cohesion of methods). Precise definition and interpretation of these metrics has proved not to be straightforward [8]. LCOM has been particularly problematic [4, 16, 15] and a number of alternatives have been proposed.

These include LCOM* which ranges from 0 for perfect cohesion (each method uses all properties) to 1 for the case where each method accesses only one property [16].

An alternative approach was suggested by Hutchens and Basili [17]. They were interested in the large-scale morphology of systems written in conventional languages. By considering data bindings (triples (f_i, d_k, f_j) where data item d_k is accessible to both functions f_i and f_j) they were able to define a dissimilarity measure between functions.

This allowed the application of statistical clustering techniques [12, for example]. The resulting dendrograms are trees showing how the most closely related (i.e. least dissimilar) functions form clusters which in turn interact with other clusters to form larger clusters until finally the entire system becomes a single cluster. Measures of cluster strength and distinctness are available.

The interpretation of the initial clusters, near the leaves of the dendrogram, suggests features such as ADTs where a group of functions interacts strongly with core data structures. Towards the middle of the dendrogram the coarser system structure is revealed while at the root all systems appear as a single blob.

Hutchens and Basili identified three characteristic 'fingerprints' from their experimental work.

planetary several subsystems 'orbiting' a central core.

black hole dominant cluster gradually accretes elements.

gas cloud shows no tendency to cluster.

Similar ideas have been explored for object oriented systems. Durnota and Mingins proposed a coherence metric from which dendrograms could be constructed [11]. Another application is the analysis of legacy code in order to identify structures which might become classes when migration to an object oriented version occurs [27]. Although these ideas are intuitively appealing they appear not to have been widely adopted.

In many ways, a class is a conventional program in microcosm—with properties corresponding to global data items and methods corresponding to functions which use global data and invoke each other.

Defining and measuring class cohesion involves multiple aspects. The internal cohesion of methods is analogous to cohesion of functions in conventional languages and affects the coupling via invocation of other methods. Similarly, the use of class properties by methods is analogous to the access of global data by functions in conventional languages. Although encapsulation and inheritance ensure that there is no truly global data, the basic concept is the same. Indirect coupling arises between methods which access the

same properties. Additionally, the invocation of methods within the same class lowers the coupling to other classes.

Our motivation stems from Hutchens and Basili's approach of looking at a system as a whole and interpreting the complex picture which results. The alternative is to define specific metrics, with corresponding information loss, and attempt to find empirical evidence for their relevance. Rather than using statistical clustering, we apply visualisation techniques in order to highlight the cohesion of classes. If we can develop OO counterparts of fingerprints then these may be useful to software engineers.

3. Visualisation with ANGLE

The conventional visualisation pipeline involves several stages [26]. Data capture and transformation are followed by the computation of geometry and finally the rendering of the images. In our previous work we have shown how this pipeline approach may be extended in the software visualisation domain [7, 14, 6, 18]. In particular, we have explored the potential uses of XML [21] both for representing the products at each stage of the pipeline and for specifying the sequence of transformations which occur.

The pipeline used in our current work is shown in Figure 1. A central component is the ANGLE graph layout engine [6] which provides reliably "nice" layout of 3D graphs using a force-based approach. Nodes repel each other as if they were charged particles while edges act like springs which pull connected nodes towards each other. Parameters selected by the user include the functional form and strength of the forces which, in turn, influence the equilibrium state which constitutes a "good" layout [10].

The current version of ANGLE is strongly based on XML technology. Its input and output files are in attributed NGML format—an XML vocabulary in which the graph structure is represented by node and edge elements—with the output files augmented with layout information. More expressive XML-based vocabularies for graph representation and exchange are being developed [28, 3]. However, NGML is currently sufficient for our needs in this domain and we can make use of XSLT transformations in order to translate to and from such alternative representations. We believe that the separation of a model from its many possible views is at least as important in visualisation as it is in other areas of software engineering.

Designing a visualisation is not an exact science and there are generally competing solutions. The key activity is the establishment of mappings between features in the system and elements of the display presented to the user.

Software visualisation requires the construction of artificial geometry, nodes and edges in our case, based on semantics extracted from the source data by appropriate mappings. ANGLE provides a flexible and extensible approach to specifying and applying such mappings. A given data source may be passed through any number of pre-filters, each of which is an XSLT transformation determining which elements and relationships will be included in the structure and semantics of the graph which underlies the visualisation. Similarly, post-filters transform the graph layout into the desired form for presentation. Figure 1 shows a single data source with 2 pre-filter and 4 post-filter transformations (i.e. two different graphs, each presented in 2 different ways).

4. Visualising cohesion

It is perhaps not surprising that such a multi-faceted concept as cohesion has resisted representation by a single numeric metric. Loss of information is inevitable when multi-faceted data is aggregated into a single value.

We contend that it is preferable to deliver a holistic "big picture" representation of the class structure which can be readily interpreted by software engineers. Since cohesion arises from the bindings among the class components we should try to visualise these explicitly and use them to assess cohesion levels.

Of course, quantitative metrics may be included in such views. The use of 3D virtual worlds enables more variables to be included in our visualisations. Thus, the size, shape or colour of symbols can be used to represent the values of numeric metrics.

The basic steps in visualisation construction are:

1. Form graph model by extracting node and edge data from source data
2. Compute 3D graph layout
3. Use semantic information to add presentation detail

corresponding respectively to the pre-layout filter, layout and post-layout filter stages in Figure 1.

The source data for the pipeline is represented by the symbol at the left of the figure. In our work, this typically arises from the use of yakyacc-based static analysis tools [18]. The details of the data format are defined by an appropriate Document Type Definition (DTD). For example, the data used to produce the worlds shown in this paper is expressed in terms of classes, which have properties and methods, which in turn use properties (Figure 2).

XSLT stylesheets are used in the transformation from the specific source data format to the more general attributed NGML input for ANGLE. One considerable advantage of this approach is that a single source data set may be used unchanged in multiple visualisation pipelines, each of which will emphasise different aspects of the data.

Table 1 shows some typical mappings between the elements of the source data and those of the corresponding

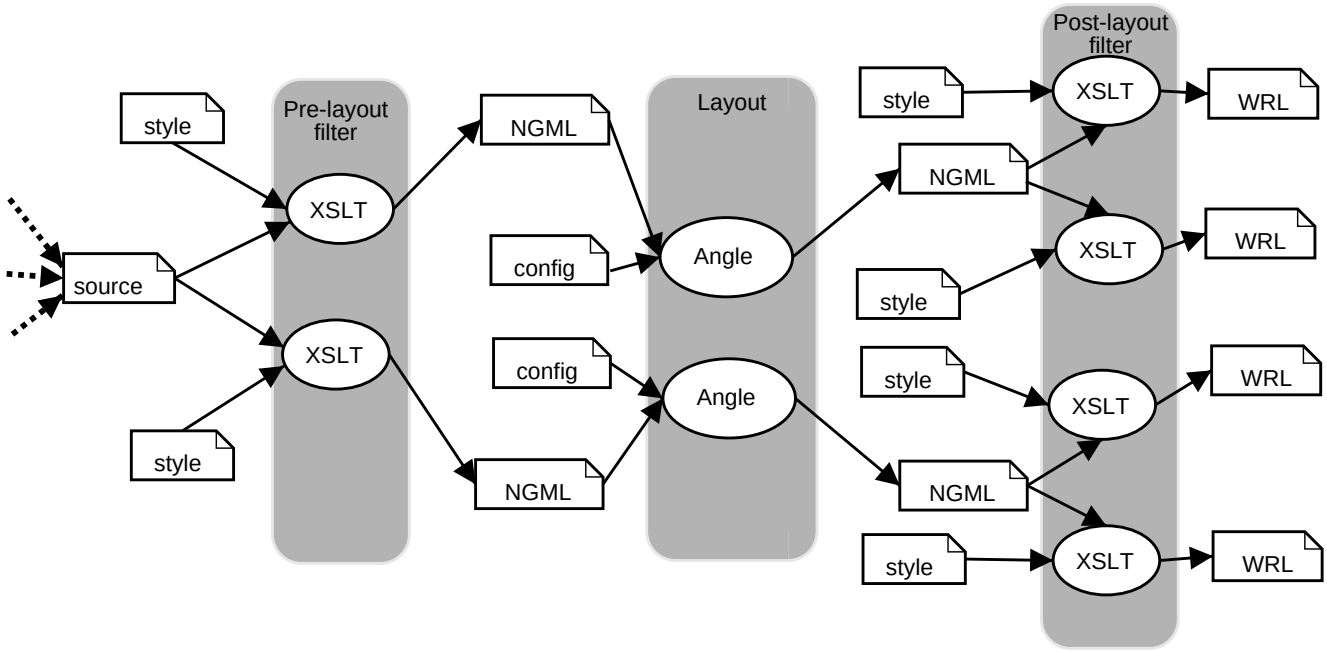


Figure 1. ANGLE **in the visualisation pipeline**

```
...
<class>
  <name>Customer</name>
  <property name="address"/>
  <method signature="getAddress" accessMode="public">
    <name>getAddress</name>
    <use property="address"/>
  </method>
..
</class>
```

Figure 2. Class description format

	source	**NGML**
γ_c	class	node
γ_p	property	node
γ_m	method	node
γ_{cp}	class-has-property	edge
γ_{cm}	class-has-method	edge
γ_{mp}	method-uses-property	edge

Table 1. Pre-filter mappings

	NGML	**VRML**
δ_c	node (classNode)	purple cylinder
δ_p	node (propertyNode)	green cube
δ_m	node (methodNode)	red sphere
δ_{cp}	propertyEdge	yellow cylinder
δ_{cm}	methodEdge	purple cylinder
δ_{mp}	useEdge	cyan cylinder

Table 2. Post-filter mappings

attributed NGML file. The nodes and edges resulting from the selected mappings will participate in the layout computation and be available to subsequent pipeline stages. The worlds shown in this paper represent uses as edges (γ_{mp}) between method (γ_m) and property (γ_p) nodes. Some include nodes representing the class itself (γ_c) and edges connecting it to properties (γ_{cp}) and methods (γ_{cm}).

Applying the XSLT file corresponding to some pre-filter mapping Γ yields an NGML description of the graph, together with various other data which have been attached as attributes of node and edge elements. Such attributes are ignored by ANGLE and simply pass out the other end where they are available to transformations further down the pipeline. In this example, three separate kinds of node and three kinds of edge are present.

ANGLE then generates a 3D layout consistent with the configuration data (algorithm, terminator and iteration pa-

```
<graphset>
  <graph>
    <title>The Customer Class</title>
    <nodes>
      <node nodetype="classNode">
        <name>Customer</name>
      </node>
      <node nodetype="propertyNode"
            accessmode="private">
         <name>address</name>
      </node>
      <node nodetype="methodNode"
            nid="getAddress"
            accessmode="public">
        <name>getAddress</name>
      </node>
...
    </nodes>
    <edges>
      <edge edgetype="propertyEdge">
        <from>Customer</from>
        <to>address</to>
      </edge>
      <edge edgetype="methodEdge">
        <from>Customer</from>
        <to>getAddress</to>
      </edge>
      <edge edgetype="useEdge"
            usetype="unspecified">
        <from>getAddress</from>
        <to>address</to>
      </edge>
...
    </edges>
  </graph>
</graphset>
```

Figure 3. Attributed NGML

rameters) provided. The format of the output is essentially the same as that of Figure 3 with the addition of coordinate elements for each node.

A given output NGML file may then be further transformed in various ways, as specified by XSLT files corresponding to post-filter mappings, to produce VRML worlds or other formats. For example, Table 2 contains post-filter mappings used to produce worlds shown in this paper. Note that nodes and edges which participate in the layout computation will not necessarily appear in the presentation unless corresponding mappings are used. Although not shown here, it is straightforward to add further mappings (such as *accessmode(public)* $\mapsto$ *transparency 0.5*). In this way, changes to presentation details, such as colour and shape of symbols, are neatly separated from the structural data about the underlying graph.

For convenience, we define some frequently-used sets of mappings (Table 3) used in the pre- and post-filter transformations reported in this paper. Clearly, many other individual mappings and combinations are possible. These are straightforward to generate.

Γ_1	$\{\gamma_c, \gamma_p, \gamma_m, \gamma_{cp}, \gamma_{cm}, \gamma_{mp}\}$
Γ_2	$\{\gamma_c, \gamma_p, \gamma_m, \gamma_{cp}, \gamma_{mp}\}$
Γ_3	$\{\gamma_p, \gamma_m, \gamma_{mp}\}$
Δ_1	$\{\delta_c, \delta_p, \delta_m, \delta_{cp}, \delta_{cm}, \delta_{mp}\}$
Δ_2	$\{\delta_c, \delta_p, \delta_m, \delta_{cp}, \delta_{mp}\}$
Δ_3	$\{\delta_p, \delta_m, \delta_{mp}\}$

Table 3. Sets of mappings used in this paper

5. Results

Figure 4 shows four low cohesion situations for a class (represented by the purple cylinder) with three properties (green cubes) and three methods (red spheres). Edges represent the relationships *class-has-property* (δ_{cp}), *class-has-method* (δ_{cm}) and *method-uses-property* (δ_{mp}). Each figure is a snapshot of a VRML world and is labelled with the corresponding pre- and post-filter mappings.

Figure 4(a) shows the case where no method uses any property. The graph is symmetric, with the class node in the centre, and is characterised by a radial fingerprint reminiscent of a *kina* (sea urchin). Properties which are never used are likely to be candidates for removal during design review. These will not be highlighted by metrics such as LCOM and it is valuable to be able to see these readily.

Now let us consider the effect of adding *method-uses-property* edges to our visualisations. Figures 4(d), 4(b) and 4(c) show three visualisations of the case where each method uses one property.

In Figure 4(b) all edges both contribute to the layout and are represented explicitly since $\gamma_{cm} \in \Gamma_1$ while in Figure 4(c) all contribute to the layout but *class-has-method* edges are not present in the world since $\delta_{cm} \notin \Delta_2$.

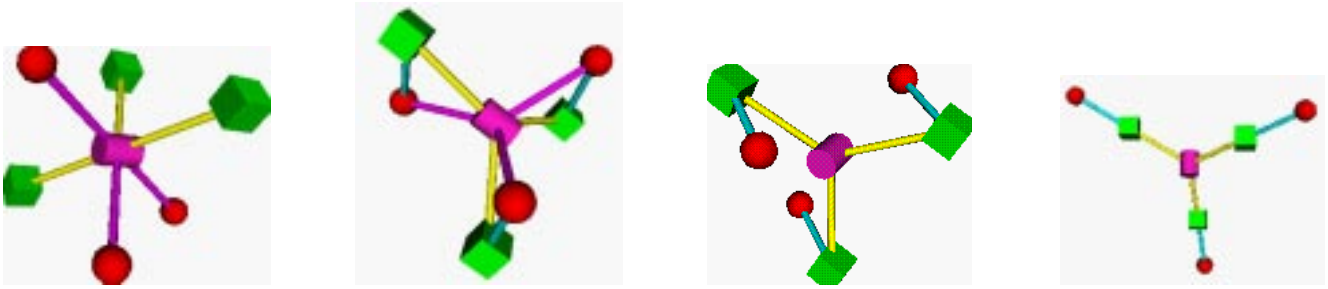
In Figure 4(d) the edges representing *class-has-method* relationships neither contribute to the graph layout nor are they present in the world since $\gamma_{cm} \notin \Gamma_2$.

The effects of more than one method accessing a property and more than one property being accessed by a method are shown in Figure 5. In this example all methods access at least one property.

Figures 5(a) and 5(b) both include γ_{cm} but the corresponding edges are not rendered in Figure 5(b) since $\delta_{cp} \notin \Delta_2$. The 3-fold symmetry arising from the three methods each accessing the same property and the two-fold symmetry arising from the use of two properties by one method are clearly visible in each case. The presence of two clusters of accesses is highlighted in Figure 5(c) where the Δ_3 post-filter has suppressed the class node and all non-use edges.

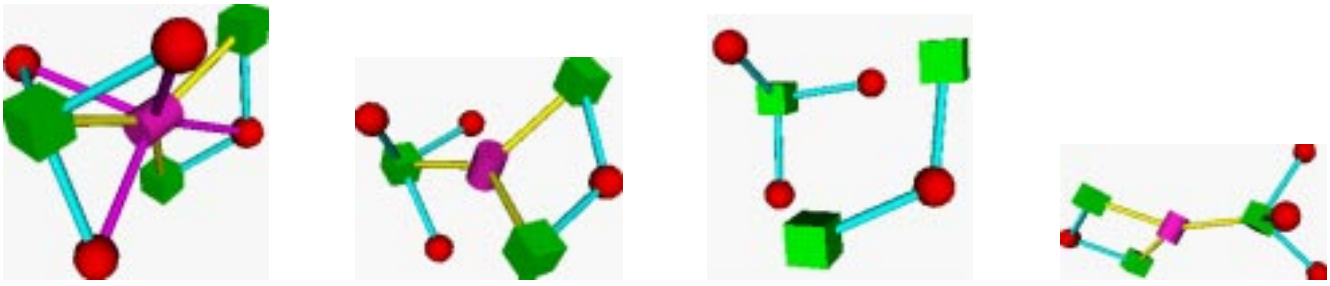
Figure 5(d) results from transformations which exclude γ_{cm}—this generally leads to more "open" layouts.

At the opposite end of the class cohesion spectrum lies perfect cohesion, where each method uses each property.



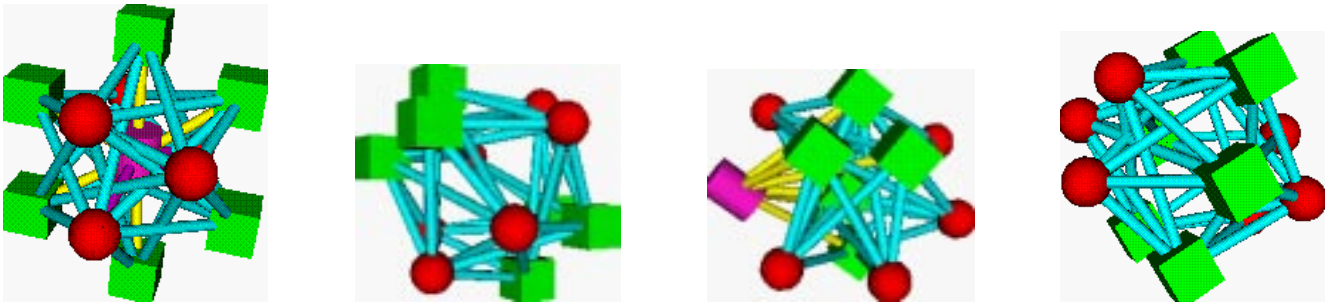
(a) $\Gamma_1\Delta_1$, 0 uses (b) $\Gamma_1\Delta_1$, 3 uses (c) $\Gamma_1\Delta_2$, 3 uses (d) $\Gamma_2\Delta_1$, 3 uses

Figure 4. Low class cohesion



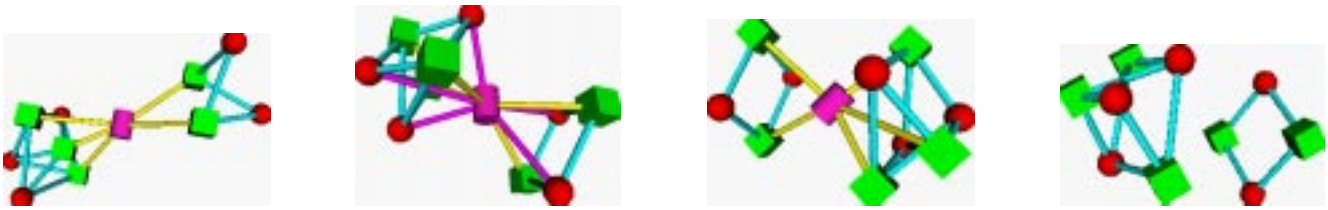
(a) $\Gamma_1\Delta_1$ (b) $\Gamma_1\Delta_2$ (c) $\Gamma_1\Delta_3$ (d) $\Gamma_2\Delta_1$

Figure 5. Multiple use participation



(a) $\Gamma_1\Delta_1$ (b) $\Gamma_1\Delta_3$ (c) $\Gamma_2\Delta_1$ (d) $\Gamma_3\Delta_3$

Figure 6. Perfect cohesion (6 properties, 6 methods)



(a) $\Gamma_2\Delta_1$ (b) $\Gamma_1\Delta_1$ (c) $\Gamma_1\Delta_2$ (d) $\Gamma_1\Delta_3$

Figure 7. Separable class

Figure 6 shows 4 presentations of this situation for a class with 6 properties and 6 methods. In Figure 6(a) Γ_1 has included in the layout the class node (at the centre of the Figure) and all associated edges. The same pre-filter has been used in Figure 6(b), so the overall shape is the same, but Δ_3 has suppressed the class node and non-use edges. This gives rise to a more un-cluttered graph since the shape is the same but fewer edges need be displayed. When γ_{cm} is removed (Figure 6(c)) the class node (at left) moves from the centre but the overall symmetry is maintained. In the perfect cohesion case, symmetry is also maintained if the non-use edges are omitted from the layout (Figure 6(d))

Perfect cohesion results in highly symmetric polyhedral structures characterised by a "gas cloud" fingerprint.

One of the primary aims of cohesion and coupling measures is to guide refactoring activities. In OO this includes activities such as splitting a class into two or more classes, combining classes (the reverse of splitting) and moving properties or methods from one class to another. Figure 7 shows a potentially separable class which has 5 properties and 5 methods. One pair of properties is used by each of a pair of methods while the other three properties are accessed by each of the remaining three methods.

Two clusters are visible—each corresponding to a candidate class after refactoring. In general, separable classes are suggestive of the "planetary system" fingerprint. The class node connects the clusters in Figures 7(a), 7(b) and 7(c).

However, If the pre-filter mappings for a separable class do not include γ_c, γ_{cp} and γ_{cm} then more than one connected component will be present. This situation is not generally welcome in force-based layout as the connected components will separate dramatically under the effect of the repulsive force. It is possible to modify the functional form of the forces to keep separate connected components "nearby" but other aesthetic issues, such as distortion of the components, then arise. A reasonable compromise is illustrated in Figure 7(d) where the mappings used are such that the class node and its associated edges participate in the layout but are omitted in the visualisation.

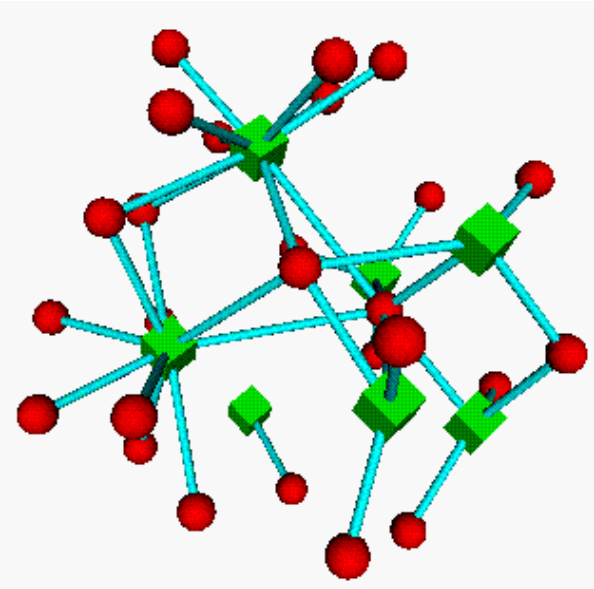
The examples we have given thus far have been somewhat artificial. Classes encountered in the wild are likely to be larger and to exhibit more complex interactions between their properties and methods. Figure 8 shows several visualisations of a class selected from the results of our on-going study of Java software.

Several features are evident. The overall shape is somewhat planetary and open. A number of *kina* fingerprints indicate the presence of several methods which use only a single property. These are connected by a small number of common methods (such as those enclosed by the ellipse in Figure 8(c)) which use multiple properties.

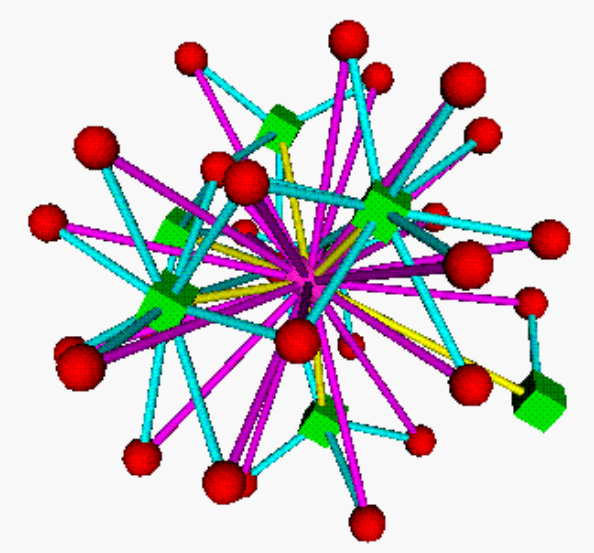
Cohesion is far from "perfect", since the structure is rather open, but it is not clear whether separation is warranted. Design review should concentrate on those *kina*, such as the pair at the upper- and lower-left of Figure 8(c), which are weakly connected (i.e. have few shared methods).

One method (rectangle in Figure 8(c)) uses only one property and is the only user of that property. The radial strand highlights this feature. This is also clearly visible in Figure 8(a) where (since $\delta_{cm} \notin \Delta_3$ and $\delta_{cm} \notin \Delta_3$) it appears as a separate cluster. A separate cluster also appears, for the same reason, in Figure 8(d)), but is a long way away from the main cluster since non-use edges do not contribute to the layout. However, in Figure 8(b) it is much harder to identify (at the right hand edge) because of the larger number of edges present. Such a feature indicates a possible design anomaly or that an unseen ancestor class contains further users.

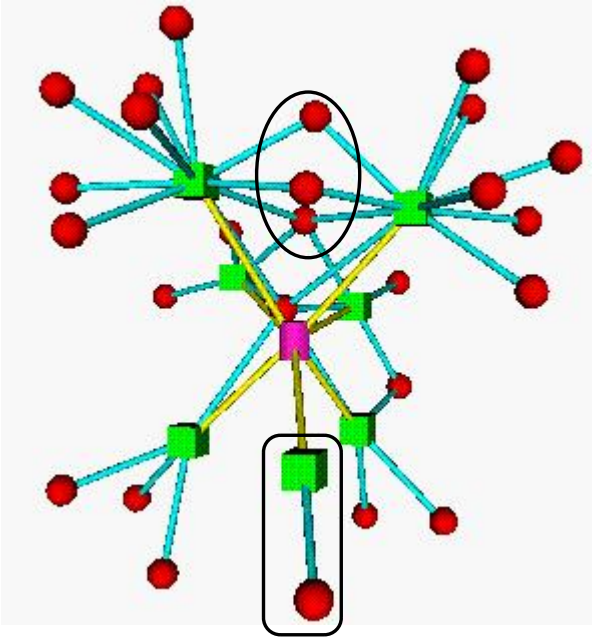
Clearly, different mappings emphasize different features and we do not expect any particular set to be preferable in all situations. For example, the inclusion of γ_{cm} leads to more spherically symmetric layouts (Figures 8(a) and 8(b)) which can make it more difficult to spot weakly connected clusters. However, the flexibility of our approach allows users to develop rapidly custom visualisations and to compare the differences between alternative representations.



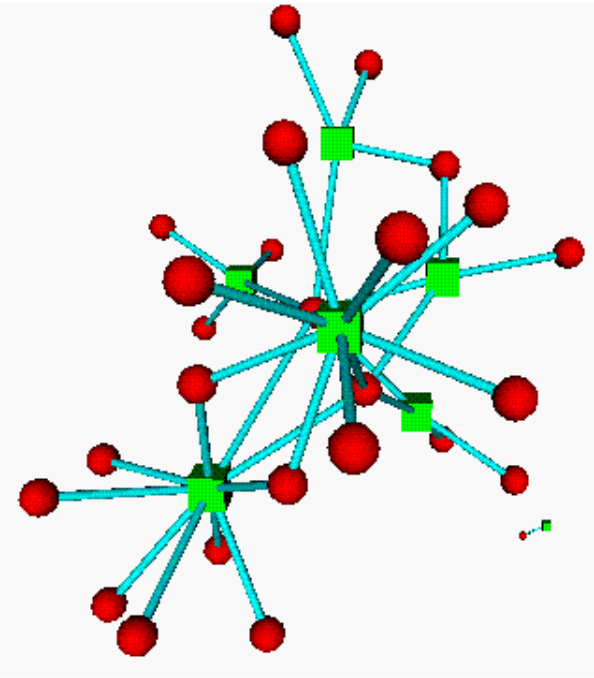
(a) $\Gamma_1\Delta_3$



(b) $\Gamma_1\Delta_1$



(c) $\Gamma_2\Delta_1$



(d) $\Gamma_3\Delta_1$

Figure 8. A “real” class with 7 properties and 26 methods

6. Beyond the desktop

In practice, classes can be very large and complex: one would expect frequently to encounter classes with tens of properties and hundreds of methods, particularly when inheritance is included. Another important aspect is the interaction of software engineers—refactoring is likely to be based on the outcome of collaborative discussions of artifacts such as our visualisations.

When worlds become very large they can become difficult to navigate and interpret with desktop facilities (typically a web browser with VRML plug-in). Similarly, as the number of collaborators increases it becomes more difficult for them to gather around a VDU. Immersive virtual reality environments such as CAVEs utilise stereo projections onto multiple surfaces to give users the impression of being inside the artifact being visualised.

Figure 9(a) shows the basic structure of the CAVE at Virginia Tech (`http://www.cave.vt.edu`). Figure 9(b) shows a snapshot from the CAVE console while the class of Figure 8 is being displayed. The labelled walls correspond to the projection surfaces indicate how much of the world is currently "inside" the CAVE. Figure 9(c) is another snapshot from the CAVE console and illustrates how the world extends beyond the CAVE walls, which are indicated by the grid lines. CAVE users do not see the walls and grid.

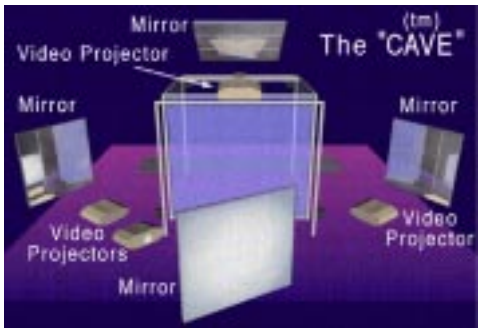
Experiencing these worlds in the CAVE is an impressive experience and leave us in no doubt that comprehension of worlds such as these is greatly enhanced. In particular, the ability to interact with "life-size" artifacts is a significant advantage.

Although CAVE technology is currently rather expensive, lower-cost and portable alternatives and augmented reality systems are becoming more affordable and may be available routinely on the desktop in a few years. Our visualisations have also been used with the MagicBook [2].

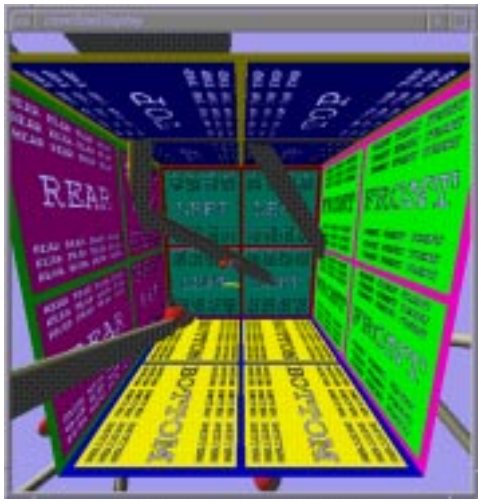
7. Conclusions and future work

Our initial results are encouraging. Our pipeline approach allows us to determine which components and relationships from a data source will be included in a graph structure, to generate a nice 3D layout of the corresponding graph and to control the presentation of the resulting visualisation using semantics derived from the original source data. XSLT transformations combined with ANGLE permit software engineers to explore not only the content of existing visualisations but also the creation of new kinds.

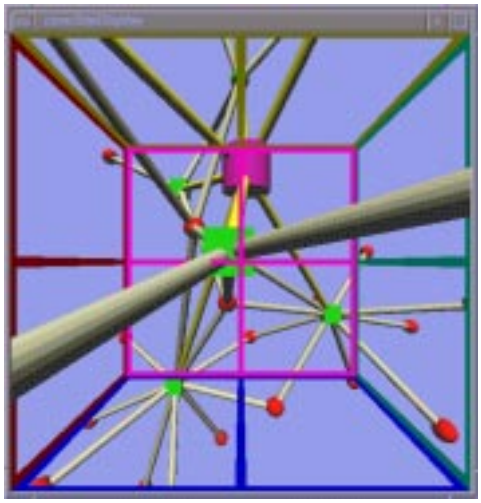
Our approach represents explicitly the bindings from which cohesion arises. A number of characteristic features have been identified and related to specific cohesion situations. We believe that our visualisations of class cohesion complement the conventional metrics-based approach. We



(a) Basic structure



(b) CAVE console snapshot showing walls



(c) CAVE console snapshot

Figure 9. The VT CAVE

hope that data from our on-going empirical work allow us to make more confident predictions about which classes are most likely to exhibit cohesion anomalies.

Only a small set of mappings has been discussed in this paper. We will explore others (such as *method-invokes-method* edges) in order to learn more about what is necessary for a "realistic" visualisation and to build up our library of transformations.

Real systems exhibit many complicating factors. Constructors and methods such as Java's toString tend to access many properties while setters and getters (mutators and accessors) tend to access only one. Inheritance and overloading are important. By providing a range of mappings we enable the user to explore the effects of such factors—we do not expect to find a single form which will be optimal in all situations.

User studies are needed in order to clarify which visualisations are most useful to software engineers.

Virtual and augmented reality environments encourage and support the collaborative analysis of large complex systems and we anticipate their increased adoption as part of the software engineering tool set.

References

[1] E. Berard. *Essays on object-oriented software engineering*, volume 1. Prentice Hall, 1993.

[2] M. Billinghurst, H. Kato, and I. Poupyrev. The MagicBook—moving seamlessly between reality and virtuality. *IEEE Computer Graphics and Applications*, 21(3):6–8, May/June 2001.

[3] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. Marshall. Graphml progress report. In Mutzel et al. [22], pages 501–512.

[4] L. Briand, J. Daly, and J. Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1):65–117, 1998.

[5] R. Carey and G. Bell. *The Annotated VRML 2.0 Reference manual*. Addison-Wesley, 1997.

[6] N. Churcher and A. Creek. Building virtual worlds with the big-bang model. In P. Eades and T. Pattison, editors, *Information Visualisation 2001*, volume 9 of *Conferences in Research and Practice in Information Technology*, Sydney, Australia, Dec. 2001. ACS.

[7] N. Churcher, L. Keown, and W. Irwin. Virtual worlds for software visualisation. In A. Quigley, editor, *SoftVis99 Software Visualisation Workshop*, pages 9–16, University of Technology, Sydney, Australia, Dec. 1999.

[8] N. Churcher and M. Shepperd. Comment on "a metrics suite for object oriented design". *IEEE Trans. Softw. Eng.*, 21(3):263–265, Mar. 1995.

[9] N. Churcher and M. Shepperd. Towards a conceptual framework for oo software metrics. *ACM SIGSOFT Software Engineering Notes*, 20(2):69–75, Apr. 1995.

[10] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

[11] B. Durnota and C. Mingins. Tree-based coherence metrics in object-oriented design. In C. Mingins, B. Haebich, J. Potter, and B. Meyer, editors, *TOOLS12&9*, pages 489–504, Sydney, Australia, 1993. Prentice Hall.

[12] B. Everitt. *Cluster Analysis*. Edward Arnold, 1993.

[13] N. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous & Practical Approach*. International Thompson Computer Press, 2nd edition, 1997.

[14] D. Hartley, N. Churcher, and G. Albertson. Virtual worlds for web site visualisation. In *Proc APSEC2000, 7th Asia Pacific Software Engineering Conference*, pages 448–455, Singapore, Dec. 2000. IEEE Press.

[15] B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice Hall, 1996.

[16] B. Henderson-Sellers, L. Constantine, and I. Graham. Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design). *Object Oriented Systems*, 3:143–158, 1996.

[17] D. Hutchens and V. Basili. System structure analysis: clusterings with data bindings. *IEEE Trans. Softw. Eng.*, 11(8):749–757, 1985.

[18] W. Irwin and N. Churcher. XML in the visualisation pipeline. In D. D. Feng, J. Jin, P. Eades, and H. Yan, editors, *Visualisation 2001*, volume 11 of *Conferences in Research and Practice in Information Technology*, pages 59–68, Sydney, Australia, Apr. 2002. ACS. Selected papers from 2001 Pan-Sydney Workshop on Visual Information Processing.

[19] M. Kay. *XSLT Programmer's Reference*. Wrox, 2000.

[20] K. Lieberherr and I. Holland. Assuring good style for object-oriented programs. *IEEE Software*, pages 38–48, 1989.

[21] D. Martin, M. Birkbeck, M. Kay, B. Loesgen, J. Pinnock, S. Livingstone, P.Stark, K. Williams, R. Anderson, S. Mohr, D. Baliles, B. Peat, and N. Ozu. *Professional XML*. Wrox Press, 2000.

[22] P. Mutzel, M. Jünger, and S. Leipert, editors. *Graph Drawing: 9th International Symposium GD2001*, volume 2265 of *Lecture Notes in Computer Science*, Vienna, Austria, Sept. 2001. Springer-Verlag.

[23] L. Ott and J. Bieman. Program slices as an abstraction for cohesion measurement. *Information & Software Technology*, 40(11–12):691–699, Nov. 1998. Special Issue: Program Slicing.

[24] M. Page-Jones. *The practical guide to structured systems design*. Yourdon Press Computing Series. Prentice Hall, Englewood Cliffs., N.J, 2nd edition, 1988.

[25] D. Parnas. On criteria to be used in decomposing systems into modules. *Commun. ACM*, 14(1):221–227, 1972.

[26] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*. Prentice Hall, 2nd edition, 1998.

[27] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 246–255, Los Angeles, 1999. ACM Press.

[28] A. Winter. Exchanging graphs with GXL. In Mutzel et al. [22], pages 485–500.

[29] E. Yourdon and L. Constantine. *Structured design : fundamentals of a discipline of computer program and systems design*. Prentice Hall, Englewood Cliffs, N.J., 1979.