An Architecture for Multi-View Information Overlays

Derek Weber **Matthew Phillips**
Defence Science and Technology Organisation
Command and Control Division
PO Box 5100, Edinburgh 5111, South Australia
derek.weber@dsto.defence.gov.au matthew.phillips@dsto.defence.gov.au

Abstract

This paper describes an architecture for supporting multi-view information overlays within the InVision visualisation framework. The concept of an information overlay is defined as well its key benefits, including the ability to support coordination of presentation across differing views. We describe how we can then abstract a view to being simply the lowest overlay in a generalised stack of information overlays. A description of the concrete implementation of the overlay architecture that has been developed for InVision is also presented. The InVision technique for supporting multi-view display coordination is then briefly compared to existing techniques supported by other environments.

Keywords: Multi view visualisation, multi view coordination, visual querying, information overlays.

1 Introduction

Research into information visualisation has yielded many innovative methods for presenting complex information and, as these techniques have become mainstream, the benefits of being able to combine different kinds of visualisations have become compelling. Today, many techniques exist for integrating multiple visualisations, including such methods as 'brushing and linking' (Becker 1987), synchronized pan and zoom (Livny 1997) and sophisticated methods of data-level coordination (of which Snap (North 2000) is a good example).

As part of our research into abstract information visualisation we have developed InVision, a software framework for producing multi-view visualisation applications. While InVision supports common techniques for integrating views, a key capability in this area is provided via *information overlays*: visual queries presented as virtual transparencies on a view. Because InVision overlays can be applied to any view and shared across views they provide a flexible and general method of coordinating multi-view display.

In this paper we describe the concept of information overlays, and outline the architecture that we have developed to enable any InVision view to support overlays. We then describe the approach taken to implement this architecture in InVision and compare our approach with that taken by some other multi-view systems.

2 The InVision Visualisation Framework

InVision is a Java-based software framework built with component-based technology for assembling integrated visualisation solutions. It allows multiple views to interact through a view coordination infrastructure and shared data models. Furthermore, it provides a view-independent method of visually querying information through its use of *information overlays*, also known as *visual sets* (Pattison et al, 2001a).

InVision's data models are accessed through a modeling framework that presents information in the form of an extended entity relationship model. The modeling framework allows information to be represented independently of storage format and location. Data is provided by plugins that access specific data sources (e.g. relational databases, XML, or even other visualisation environments).

InVision's visualisation framework defines a set of common interfaces for views, and allows them to be linked together. It also provides components for building and manipulating new views as well as a number of ready-made view components, including 2D and 3D diagramming views, a table view, a tree (outline) view and a chart view. Elements of view coordination are provided via the view coordination architecture described by Pattison et al (2001b).

3 The InVision Modeling Framework

The InVision modeling framework makes it possible for all views to share the same notion of how information is represented. The view-independent information overlay approach presented in this paper depends on concepts from the modeling framework, so this section provides some necessary details of InVision's information models.

InVision's modeling framework provides a high-level information abstraction layer. It represents information independent of the underlying data storage technology being used and, by providing a common language for information exchange, is one of the key aspects of integrating multiple types of visualisation.

In order to be able to represent data ranging in complexity from simple tables to multidimensional graphs, the modeling framework defines its data using a flexible *extended entity relationship* model (it also has many concepts in common with *attributed graphs*). Using this

representation, we have modeled information as varied as research project management structures, earthquake data and the InVision software framework itself.

An InVision system model is a set of three types of entities: *elements*, *relations* and *attributes*. An element represents a logical entity in the model, such as *Person*, *Account* or *Department*, and has an associated set of attributes and relations, which are collectively termed the *features* of the element. The values attached to attributes are 'atomic' values such as strings, numbers, dates or images, while relations hold references to other elements.

3.1 Element Aliases

The modeling framework also defines the concept of an element *alias*, which conforms to the 'Decorator' design pattern described in Gamma et al (1994). An alias of an element can be treated as if it is the element itself, as it presents all the features of the aliased element as its own. Aliases can also have their own extended features, which can be modified independently of the aliased element. They can also override the original feature values of the elements they alias, effectively changing or hiding features.

The original reason for introducing element aliases was to support the InVision query engine, which can create new features or hide existing ones as the result of executing a query. By creating aliases to the original elements and returning them as its result, the query engine can safely modify feature sets without affecting the original elements (and inadvertently making the query results persistent).

As it turns out, the concept of a modifiable alias is also core to our technique for supporting inter-view overlays.

4 Information Overlays

InVision *information overlays* provide a way to interactively query the information on a view and present the result directly within the view by modifying its presentation. Information overlays can be thought of as virtual transparent layers, conceptually similar to real transparencies used to show features on map, although we will see that they extend the transparency concept to provide far more flexibility.

The key benefit of using information overlays for querying is that results are displayed in the same view as the source information, rather than in a separate view. Thus overlays not only show the results in answer to a query, but also how they relate to other information. Results shown in context can also be less distracting than in a separate view. That this is an effective way of presenting contextual information is attested to by the fact that this technique is so widely used – for example, the Periodic Table viewer described in (Ahlberg 1992). However, most systems do not formalize the application of the presentation in a generic way.

An information overlay is essentially a mapping from each model element to a set of logical *visual styles* that determine its display.[1] Examples of visual styles might be 'set background colour to red', 'set shape to ellipse' or 'set tool-tip to be "Squishy"'. When an overlay is applied to a view, these styles are merged with existing styles; each view does its best to display as many types of display style as possible.[2] Styles in overlays higher in the stack override ones lower down so, for example, if two overlays specify different shape styles, the one higher in the stack will be displayed. Some styles can be merged rather than overwritten: for example, tool-tip styles are appended to form multiple lines.

An additional rule is that, if a view element ends up with no display styles at all (e.g. if all overlays it participates in are hidden), it becomes invisible. This provides a way to quickly elide information from a view.

For example, Figure 1 shows an abstract view of a software system, with modules (in this case Java classes) shown as hexagons and stars inside container shapes representing key components of the system. Several overlays have been added, one of which colours classes by their size, and another that causes those created by a particular author to show as stars. This demonstrates the use of two display styles, colour and shape, to highlight overlapping aspects of the information. Note that turning off all other overlays except the 'Author' overlay would result in classes not written by the given author being hidden.

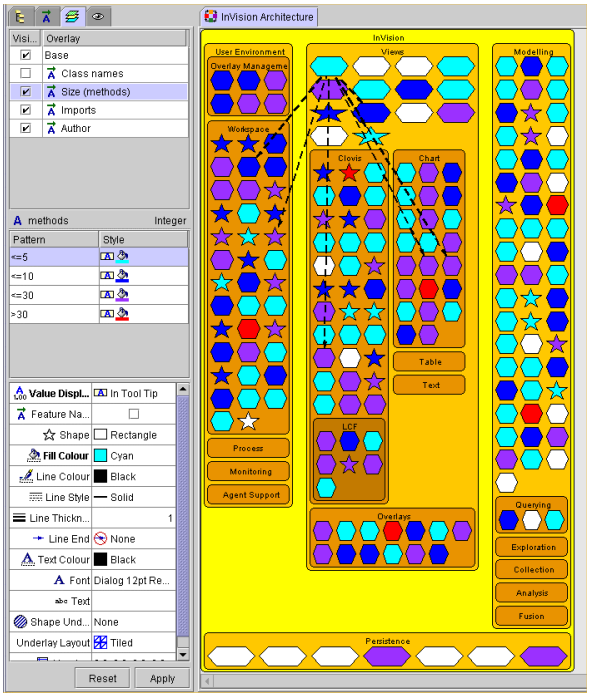

Figure 1: InVision view with multiple overlays highlighting different aspects of the view.

[1] A more rigorous definition of information overlays can be found in Pattison et al, 2001a.

[2] Even apparently view-specific styles, such as 'shape', often have useful representations on other views: for example, a line chart displays shapes as the points on the line, a table displays the shape as an icon.

In the example view, a third overlay is actually visible, showing the dependency of one class to other classes using dotted lines. This is an example of an overlay adding new information, instead of hiding or modifying that which is already there. In this case, the overlay has selected the relations that represent the dependencies between the selected class and others and applied a dotted line style to them. The 2D diagram view then represents the new relation elements as lines connecting the shapes representing their source and target elements.

The example above illustrates that information overlays go beyond the basic 'transparency' model of some visualisation solutions (such as BBN Technology's OpenMap (BBN 2003)) by employing a more sophisticated approach to logically merging display styles that includes the ability to elide elements. In effect, information overlays combine the advantages of the transparency-style overlays used on map views, which only add new visual elements, with the approach of many other multi-view systems, which allow modifications to the presentation of existing information only (section 7 contains a brief comparison of InVision's approach with that taken by some other multi-view systems).

Since information overlays are supported across all views, InVision can provide a single user interface for generating and manipulating them. Implementers of new views within the framework automatically get the benefit of this standard information overlay user interface.

The flexibility of the approach has a downside: because information overlays can affect nearly any aspect of a displayed element, including size and visibility, overlays can add a higher computational cost to rendering a view. Overlays affecting size and visibility of view elements may trigger re-layout and/or zooming and panning of views to ensure that all elements are appropriately displayed, a disruptive process which might override the advantages of showing the information in context.

5 The Common Overlay Architecture

The concept of information overlays is general enough that it can be extended to be the basis for representing all views in InVision. To see how this approach works, it is first necessary to briefly describe how views are generated in InVision.

All InVision views are created from a *view specification* that contains rules for selecting elements from the underlying data model and controlling how they are presented in the view. For example, the specification for a 2D diagram view might be the equivalent of 'select all People elements, display them using a stick figure shape, and lay them out left-to-right'. The rules in the view specification, plus additional mark-up from its information overlays, are then executed against information from the source data model to generate a *view model*, a view-specific representation of the data ready for display by a view. For example, the view model for a table view would be a 2D array of cells. One or more views can then be created to display and manipulate the model. Figure 2 illustrates this process.

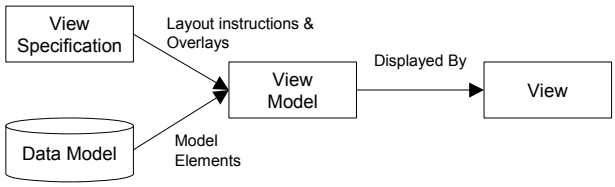

Figure 2: Process for building an InVision view.

The key to providing common support for information overlays is to treat the view specification as defining another kind of information overlay, one which always lies at the bottom of the overlay stack and is thus termed the *base overlay*. We then introduce the concept of *style layers* of displayable *view elements* (described below) and require each overlay to be able to generate a corresponding style layer from a source data set. The resulting stack of style layers is then merged using the rules described in section 4, and the view model is built from the resulting merged layer (see Figure 3). Since both the view specification and its overlays generate style layers, the same merging logic can used across all views, allowing the view implementer to support any sort of information overlay.

The style layer for each overlay consists of a set of view elements, which are aliases to elements in its source data model, with the addition of an attribute specifying a display style. Each view element will end up merged with others aliasing the same underlying model element and displayed in the view according to its merged display style. The nature of the target view will determine how a view model is created from the merged layer and how the resultant style is rendered.

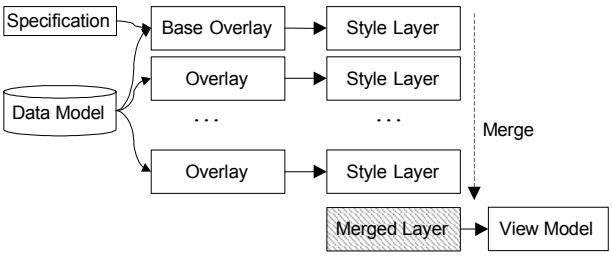

Figure 3: The Common Overlay Architecture.

This approach has the advantage that, while the view elements for the base overlay are generated according to the particular view's specification, the elements from other sorts of overlays can be generated by any technique desired. So while InVision's most commonly used overlay, the *feature overlay*, assigns styles based on the model element's feature values, others that perform a task-specific function can be created and used on any view. For example, an overlay that shows the results of a graph clustering analysis might be developed for a particular application.

6 Implementation

This section describes our approach to implementing the common overlay architecture in InVision. This implementation currently provides overlay support for three very different view types in InVision: a 3D diagram

view, chart view and table view. An example of a 3D view based on the framework is shown in Figure 4 below.

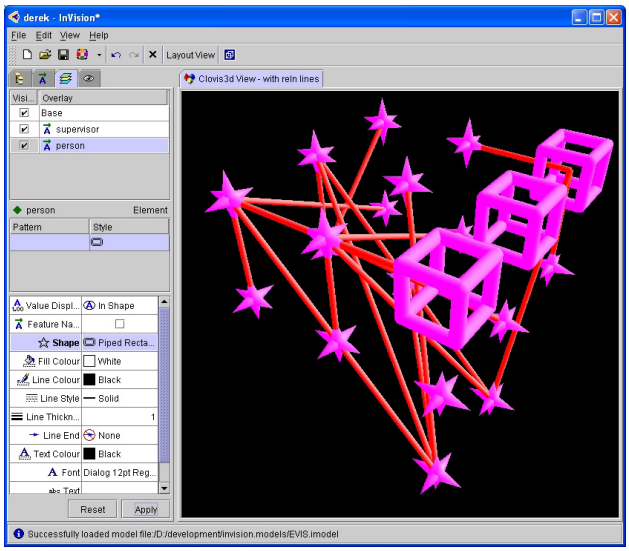

Figure 4: Example of information overlays in an InVision 3D view. The stars and pipes represent people in a research group, the lines represent supervisory relationships.

Our implementation adds two functional units to the architectural diagram in Figure 3 to perform the generation of a merged style layer: (a) a 'glue' component between the overlay and style layers that tracks changes to the overlays and triggers the appropriate changes to the corresponding style layers and (b) a style layer merger that listens to changes in the style layers and manages efficient incremental merging into a single layer (see Figure 5). The view elements from this merged layer are then reflected in the underlying view model, a process that is up to the specific view implementation.

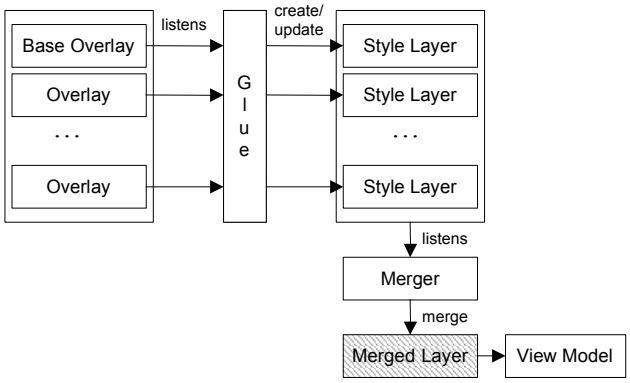

Figure 5: Implementation approach: 'glue' synchronizes overlay and style layer stacks, while 'merger' maintains a merged layer.

Both view specifications and overlays have a defined interface for generating view elements from a source model that the glue component uses to populate the style layers. It is necessary to merge the layers in order from the base layer downwards, as the source elements provided to each new layer consist of the merged view elements of the layers above. In this way each new overlay can influence the appearance of the overlays above it.

Another reason for the layers to be built incrementally is to support two advanced usages: multiple view elements that refer to a single data model element, and elements that do not reference an element in the data model at all (termed *non-model elements)*. Non-model elements may be created by a view for various reasons: for example, the 3D view uses them to represent container shapes for sub-queries.

Figure 6 shows how the stack of style layers is structured internally. The top layer is the data model layer, the merged layer on the bottom is maintained by the style layer merger, and the layers in between represent the style layers corresponding to the overlays.

The first thing to note about the structure shown in Figure 6 is that many view elements do not directly refer to model elements, but refer instead to other view elements in preceding layers. This linkage is maintained by the merger to disambiguate cases where there is more than one view element in a layer for a single model element. For example, $VE_{1.1}$ and $VE_{1.2}$ both refer to E_1. $VE_{2.2}$ in the following layer also ultimately refers to E_1 but was applied by the user to the visual representation of $V_{1.2}$ only, resulting in one occurrence of E_1 remaining green-coloured while the other is red. If view elements were merged only on the basis of their ultimate model element, then the style applied by $VE_{2.2}$ would apply to both occurrences of E_1.

The second point to note is that new view elements can appear in later layers that are not in preceding ones. For example, $VE_{2.3}$ appears in layer 2 and aliases a model element (E_2) that is not referenced in layer 1. Layer 3 then further modifies the appearance of this new element. Also, view elements that do not reference any model element at all are supported. For example, $NME_{1.4}$ (layer 1), may have been created as a 'synthetic' display element by the view specification to represent something not directly connected with the data model, such as a visual grouping. These complications, plus the fact that layers can change once added, mean that an efficient implementation is not straightforward.

The advanced features described above were not necessary for InVision's basic views such as business charts and table views for which the framework was initially designed. However we quickly found that they are essential to support our more sophisticated diagram-style views, since both allow grouping and sub-queries as well as allowing the same model element to appear more than once in a view.

Because overlay manipulations can have large effects on the structure of a view, our implementation ensures that, when the overlays in the stack change, the view will be efficiently updated without having to be fully rebuilt, since only those layers below the changed overlay need be refreshed. When overlay changes are detected by the glue component, it creates new style layers or minimally updates the existing ones accordingly. These changes to style layers result in changes to the layers' view elements, which subsequently affect the alias chains in the style

layer stack. New view elements may be inserted, or replace existing ones in situ, or be removed from the chains. The style layer merger is prompted to remerge only those chains that were altered, and the view model responds accordingly by updating the view, which may be as simple an operation as repainting the few presentation elements that have changed.

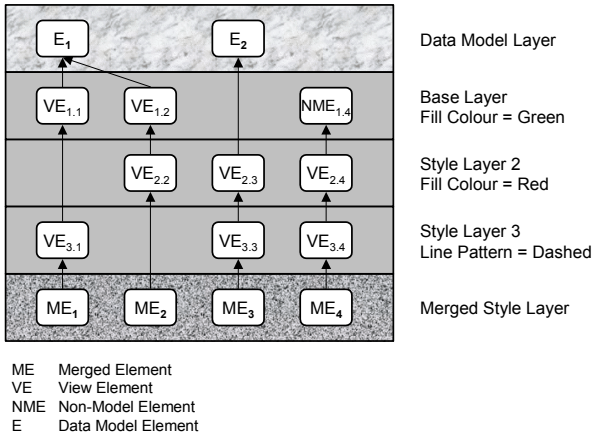

Figure 6: Style layer view elements aliasing structure.

6.1 Development Issues

This section highlights some of the issues encountered while implementing the common overlay architecture, which forced us to reconsider our approach.

The style layer model component, which merges the view elements, was originally constructed for InVision's chart and table views, each of which permits only one view element per model element (e.g. a single a table row or a single chart data point). Style layers were merged sequentially assuming that for each view element in one layer there was exactly one in the next layer down (see Figure 7) and thus merging was a simple iterative process. All view elements directly aliased the data model elements they represented but no view element would ever be confused for another because only one ever aliased a given element. However this approach meant multiple view elements representing a single model element could not be handled.

When work began on constructing a new 3D shape view component it became apparent that the assumption of exactly one view element per model element was no longer valid. The new structure needed to be able to distinguish which visual elements in a style layer corresponded to which other visual elements in the preceding layers in order to avoid ambiguities.

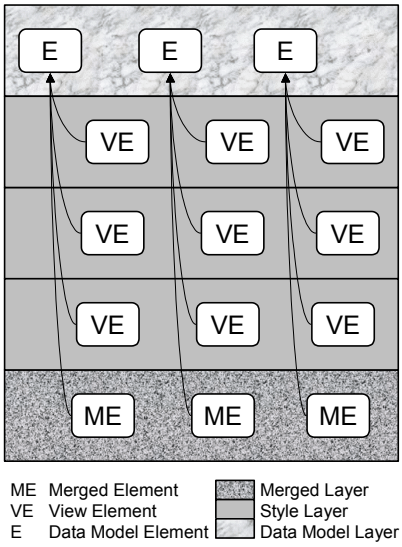

Figure 7: The original style layer view elements aliasing structure was found to be insufficiently flexible.

It was also apparent that we needed to support view elements that did not reference the data model at all. Non-model view elements may be artificially created to represent a group of data model elements or to represent the results of a calculation across a group of elements. The main issue with non-model view elements in a style layer becomes apparent when the layer is moved down the style layer stack. Any alias chains depending upon the non-model elements are broken. It was found, however, that the only place that such elements were actually created was in the base overlay (i.e. by the view itself), which cannot be moved, so no layer movement issues were actually encountered. In future revisions this may need to be readdressed.

As a side note, the logic developed for managing alias chains was so much more complex than that developed for the previous approach that, in order to ensure reliability, we introduced a test-driven development scheme, as proposed by Beck (2002). We developed a full suite of regression tests to ensure correct behaviour is maintained when modifying the existing logic.

We believe the existing implementation will handle most view overlay requirements, being flexible enough to support multiple view elements aliasing a given model element as well as artificially created elements.

7 Comparison With Other Approaches

Most modern generic visualisation systems employ some variant of the basic data element → display style mapping approach described in this paper as an integral part of their operation (eg. DEVise's *visual attributes* (Livny 1997), Polaris' *retinal properties* (Stolte 2002)). However, InVision goes somewhat further than most systems by allowing any presentation option to be represented as a changeable display attribute, including visibility, which can have fundamental effects on the structure of the view.

It is also common for multi-view systems to support view coordination via consistent rendering of display styles across linked views. For example, in Visage (Roth 1996), applying a display style to elements dropped into a view will cause the same style to be applied to the original elements in their source view. The InVision approach essentially formalises and extends this method of visualisation 'mark up' by introducing the concept of information overlays and defining a consistent method applying them across multiple view types. In fact, the InVision approach goes as far as describing *any* view as a view-specific overlay plus generic additional overlays.

As well as being an effective and general method of interactively marking up visualisations, this approach enables a new form of general inter-view coordination via sharing of overlays across views (see Figure 8). A key advantage of this over the implicit linking used in other systems is that, when the same overlay is applied to multiple views, not only will changes to the overlay be reflected consistently, but also the visual link between the views is explicitly represented and able to be manipulated. This allows only selected mark up to be shared, and for the visual link to be modified or broken as needed.

Most visualisation systems have no concept analogous to InVision's view specifications that allow them to regenerate existing visualisations for new or changed data; DEVise and Polaris being notable exceptions. DEVise, for example, allows saving and sharing *visual templates* that support later regeneration and collaborative sharing of visualisations. However no system that we are aware of incorporates the equivalent of information overlay specifications as an integral part of the view's specification.

8 Conclusions and Future Directions

The unified information overlay approach described in this paper has allowed us to consistently support overlays across all current InVision views without significant extra effort from the view developers. Future work in this area will include the development of an interface for showing and manipulating links between views, both between shared overlays and the other kinds of view coordination supported by the framework. We also plan to investigate new kinds of information overlays, including an overlay based on generic queries and smart mapping of display styles.

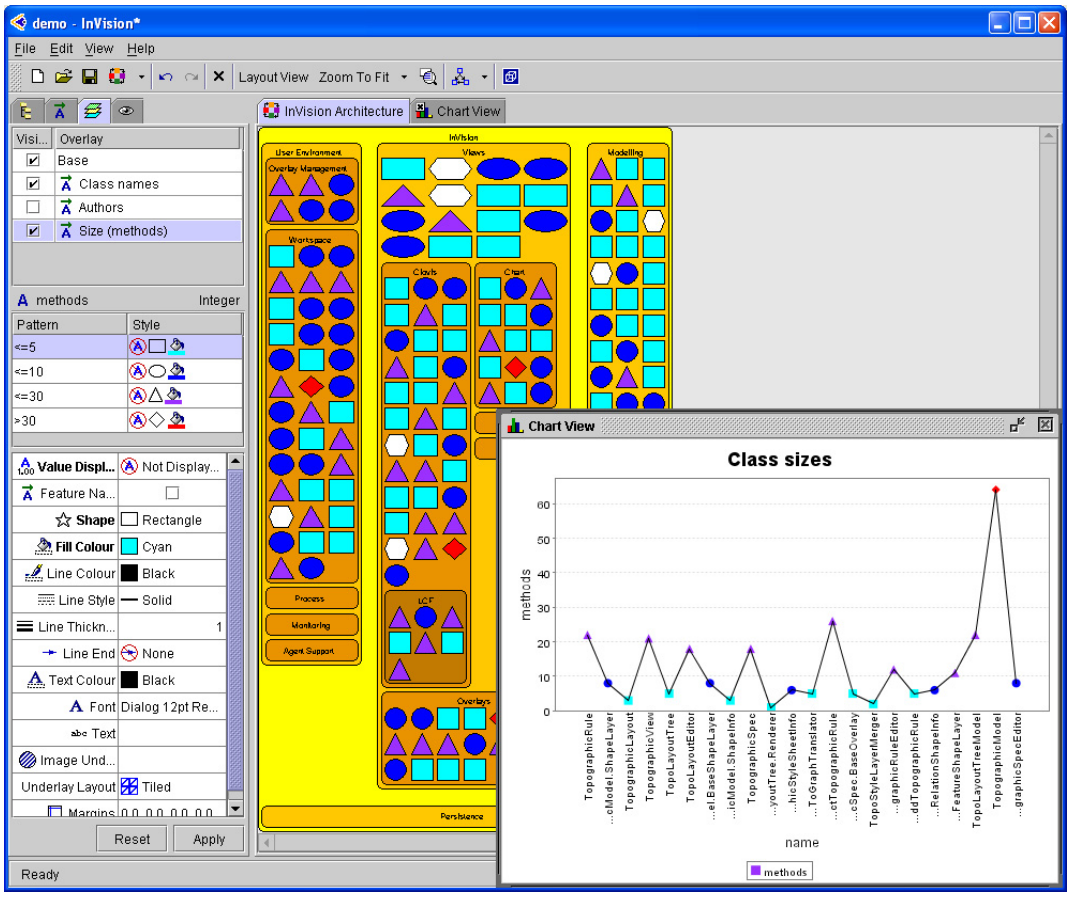

Figure 8: InVision with shared overlays on a 2D Clovis view and a chart view. Class size (in number of methods) is shown in colour, classes written by a selected author are shown as stars. The overlay stack is shown top left.

9 References

Ahlberg, C., Williamson, C., Schneiderman, B. (1992): Dynamic Queries for Information Exploration: An Implementation and Evaluation, *Proc. CHI'92: Human Factors in Comp. Systems Conf.* ACM Press. 213-218.

BBN Technologies: OpenMap(tm), BBNT Solutions LLC. http://openmap.bbn.com/. Accessed 30 Sept 2003.

Beck, K. (2002): *Test Driven Development: By Example*. Addison-Wesley.

Becker, R., Cleveland, W. (1987): Brushing scatterplots, *Technometrics*, 29(2), pp. 127-142.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Wokingham, England.

Livny, M., Ramakrishnan, R., Beyer, K., Chen, G., Donjerkovic D., Lawande, S., Myllymaki, J., Wenger, K. (1997): DEVise: integrated querying and visual exploration of large datasets. *Proc: ACM SIGMOD 1997*, Tucson, Arizona, May 1997, pp. 301-312.

North, C., Shneiderman, B. (2000): Snap-together visualization: A user interface for coordinating visualizations of a relational database, *Proc: 5th International Working Conference on Advanced Visual Interfaces (AVI 2000)*, Palermo, Italy, May 2000.

Olston, C., Woodruff, A., Aiken, A., Chu, M., Ercegovac, V., Lin, M., Spalding, M., Stonebraker, M. (1998): DataSplash. *Proc: SIGMOD 1998*, Seattle, Washington, June 1998

Pattison, T., Vernik, R. and Phillips, M. (2001a): Information Visualisation using Composable Layouts and Visual Sets. *Proc: Australian Symposium on Information Visualisation, (invis.au 2001)*, Sydney, Australia, **9**. Eades, P. and Pattison, T., Eds., ACS. 1-10.

Pattison, T. and Phillips, M. (2001b): View Coordination Architecture for Information Visualisation. *Proc: Australian Symposium on Information Visualisation, (invis.au 2001)*, Sydney, Australia, **9**. Eades, P. and Pattison, T., Eds., ACS. 165-171.

Roth, S. F., Lucas, P., Senn, J. A., Gomberg, C. C., Burks, M. B., Stroffolino, P. J., Kolojejchick, J. A., Dunmire, C. (1996): Visage: A user interface environment for exploring information. *Proc: Information Visualization*, 3--12. San Francisco: IEEE.

Stolte, C, Tang, D., Hanrahan, P (2002): Query, Analysis, and Visualization of Hierarchically Structured Data using Polaris. *Proc: SIGKDD '02* Edmonton, Alberta, Canada: ACM.