Chapter 9

The Affinity Browser

Xavier Pintado

Abstract Large numbers of classes, complex inheritance and containment graphs, and diverse patterns of dynamic interaction all contribute to difficulties in understanding, reusing, debugging, and tuning large object-oriented systems. These difficulties may have a significant impact on the usefulness of such systems. Tools that help in understanding the contents and behaviour of an object-oriented environment should play a major role in reducing such difficulties. Such tools allow for the exploration of different aspects of a software environment such as inheritance structures, part-of relationships, etc. However, object-oriented systems differ in many respects from traditional database systems, and in particular, conventional querying mechanisms used in databases show poor performance when used for the exploration of object-oriented environments. This chapter defines the requirements for effective exploration mechanisms in the realm of object-oriented environments. We propose an approach to browsing based on the notion of *affinity* that satisfies such requirements. Our tool, the affinity browser, provides a visual representation of object relationships presented in terms of affinity. Objects that appear closer in the visual representation are more strongly related than objects lying farther apart. So, the intensity of a relationship is translated into distance in the visual representation that provides the support for user navigation. We provide many examples of metrics defined over the objects of an environment to illustrate how object relationships can be translated in terms of affinity so that they can be used for the exploration of an environment.

9.1 Introduction

Large numbers of classes, complex inheritance and containment graphs, and diverse patterns of dynamic interaction all contribute to difficulties in understanding, reusing, debugging, and tuning large object-oriented systems. From the inception of object-oriented environments, developers and software designers have felt the need for tools that support the process of understanding the objects, the classes and the relationships provided by their environments. For example, reuse of existing software components requires navigation and inspection of classes and how they are related. Inspection and navigation capabilities are also instrumental for the combination of instantiated objects since they allow the

Xavier Pintado, "The Affinity Browser," *Object-Oriented Software Composition*, O. Nierstrasz and D. Tsichritzis (Eds.), pp. 245-272, Prentice Hall, 1995.

user to go back and forth, inspecting objects and combining them. In a similar vein, discerning global and local patterns of interaction among classes and among objects is critical for tuning and debugging.

This chapter proposes an approach to browsing for object-oriented environments based on the notion of affinity. Our tool, the affinity browser, allows for the exploration of collections of objects based on a visual representation of object relationships presented in terms of affinity. Objects that appear closer in the visual representation are more strongly related than objects lying farther apart. So, the intensity of a relationship is translated into distance in the visual representation.

Our approach displays many advantages. First, affinity browsing is not based on point-to-point navigation. The user is provided with the set of objects that lie within a given neighbourhood relative to the object currently being inspecting. The affinity browser promotes, therefore, proximity-based navigation whereby exploration proceeds by exploring first the objects that are close to the current object of interest. Second, the browser allows for the exploration of dynamically evolving relationships. The evolution of such relationships is visualized as an animation where the change in the relative position of objects conveys the change of the underlying relationships expressed in terms of affinity. Third, many different kinds of object relationships can be translated into affinity representations allowing the same exploration paradigm and the same user interface to be used to explore a large spectrum of object relationships.

This chapter is organized as follows. Section 9.1.1 addresses the problem of finding and selecting objects inside an object-oriented environment. It discusses the characteristics of object-oriented systems that may have an impact on the effectiveness of various browsing mechanisms. Section 9.1.2 surveys work related to browsing ranging from traditional graph-based browsing to graphical and spatial browsing. Section 9.2 defines the requirements for effective exploration mechanisms in the realm of object-oriented environments. Section 9.3 presents the affinity browser as a tool that satisfies such requirements. In section 9.4 we provide many examples of metrics defined over the objects of an environment to illustrate how object relationships can be translated in terms of affinity so that they can be used for the exploration of an environment

9.1.1 Object Selection

We address here the issue of selection in the object-oriented realm. Users may want, for instance, to select classes, objects, or functionality. Selection in an object-oriented environment has many problems, however. First, an application designer has only approximate selection criteria to select an appropriate reusable object class for developing his or her application. Second, the object classes and the objects in a running system have relationships that change dynamically. Third, objects are encapsulated and content selection has only very limited use.

Furthermore, object-oriented principles applied to software design seem to promote systems with object relationships that are more complex than in more traditional software

environments. Many authors think that these principles will allow designers and developers to create software environments that are an order of magnitude more complex than existing software systems [19] [4].

A noteworthy supporting reason for such belief is that object-oriented design techniques seem to allow significantly better decomposition of complex problems into units of manageable complexity. First, by the virtue of encapsulation an object conceals its internal complexity and it acquires some level of autonomy. Second, incremental definition through inheritance allows for the endless refinement of object behaviour and functionality without the need to rework the whole hierarchy at each refinement step. These mechanisms, with such desirable features, allow for the implementation of models that integrate much detail both at the object level and at the level of object relationships. This intuition is further supported by experience that shows that it is quite easy to introduce complexity in the design and in the implementation of an object-oriented environment. For instance, object-oriented programming is more an activity of *wiring* together sets of objects. For the programmer or for the designer whose task is to build a system through the composition of objects it might be quite easy to combine them in many different ways — this is the producer's view. On the other hand, for a developer who wants to understand existing functionality for reuse or maintenance, it may be difficult to comprehend the large number of functional relationships that have been created — this might be the consumer's view.

Early experiences with object-oriented environments highlighted the need for tools that allow for the exploration of object relationships. The Smalltalk environment, for instance, already provided a sophisticated integrated browsing tool [12]. Interestingly enough, it has been argued that the Smalltalk browsing tool is one of the most appealing features of that environment and it is often cited as a reference. For sure, almost every programming activity on the environment relies on the browser to support navigation needed for the kind of non-linear programming promoted by object-orientation. The browser is used to code new objects, to find reusable classes and to explore object relationships.

9.1.1.1 Querying and Browsing

The two methods commonly applied for selection are querying and browsing. The methods are usually applied in a complementary manner; we query and browse in alternation, applying which method seems more appropriate at different stages of the selection process.

Querying provides fine selectivity when the structure of the information space is known and when content selection can be used. For instance, querying is the primary selection method in database systems. When querying provides good selectivity, browsing diminishes in importance. Most selected items are appropriate and we only need a crude browsing tool to inspect them.

Querying, however, can have poor results for many reasons. If the selection criteria are ill-defined and fuzzy querying does not work well, e.g. in information retrieval. If the structure of the information space changes dynamically, queries are not easy to formulate, e.g. in financial information systems. Finally, if content selectivity is difficult to exploit,

querying loses a lot of selectivity power, e.g. in multimedia databases. In all these cases powerful browsing capabilities become indispensable.

9.1.1.2 Dynamically Evolving Relationships

As we already mentioned, the analysis of dynamically evolving relationships plays an important role in debugging but can also be of invaluable assistance for reuse since it helps understanding how objects are related in existing applications. However, providing support for the understanding of dynamically evolving relationships is a challenging task. In fact, traditional querying techniques usually assume a user with knowledge of the search structure that supports selection. Such an assumption usually implies structure stability since it seems unrealistic to assume user knowledge of a quickly evolving structure.

With traditional databases it is usually assumed that their information contents changes but not their structure — or at least not frequently. For example, widely used query languages such as SQL provide almost no support for selection in an environment with a changing structure. The stability of database schemes represents an advantage in terms of access to information but it makes traditional databases ill-suited for information with dynamically evolving structures.

The need to cope with dynamically evolving relationships appears in many object selection problems. For example, we may be interested in finding which are the objects that interact most frequently with a given object in order to determine its patterns of interaction. The change in the interaction patterns depending on what activities the system is performing may provide useful information about the intended role of an object. This information can be used, for instance, to assess the potential of reuse for an object in an environment that may or may not provide the same activity context.

The need for more flexibility than that provided by query mechanisms appeared also in databases. For example, Motro [20] [21] [22] describes browsing tools that allow for *navigation* in a semantic network extracted from the internal structure of a relational database, and provide capabilities for fuzzy queries. The approach has been later extended to integrate similar capabilities in an object-oriented environment [23].

9.1.2 Related Work

Because there is an observable trend towards more complex and quickly evolving information systems we need to investigate how to enhance browsing capabilities for the exploration of information systems. In this section we describe previous work related to browsing.

9.1.2.1 The Smalltalk Browser

To the best of our knowledge, the Smalltalk system was the first programming environment where exploration tools played a major role. Furthermore, the browsing concepts and mechanisms have been clearly defined [13] [12] and they are quite often cited as the historical reference to which more recent browsing tools are compared.

The Smalltalk environment provides capabilities to inspect the message interface of objects through a system view called a *browser*. Similarly, the internal state of an object can be inspected through another system view called an *inspector*. Furthermore, it is possible to obtain interface information about sets of objects through another kind of system view called a *message-set browser*. These views are generated as responses to queries such as: which classes implement a given message? Which objects send a particular message?

The main way to find out about classes in the environment is to use a system class browser. The browser presents a hierarchical view of class-related information. It presents *categories* that organize the classes within the environment, and categories that arrange messages within each class. Categories provide essentially a way of grouping classes and messages into meaningful groups.

It should be noted that in the Smalltalk environment the role of the exploration tools is not restricted to inspection. For example, an *inspector* allows users to change interactively the values of instance variables and to send messages to objects. In general, inspection tools are used for both inspection and programming purposes. For instance, the creation of a new class derived from an existing one, and the definition of new methods is also performed through the browser.

Other browsing tools have been described and implemented in various systems. The browsing mechanisms implemented in the Smalltalk environment have been a continuous source of inspiration for new browsing tools. For example, the Trellis programming environment [24] provides browsing capabilities that are quite similar to those of the Smalltalk environment [12].

The great majority of existing browsing tools allow for a *point-to-point* navigation, i.e. the navigation paths are defined by a tree or a network structure. For instance, the tree structure of the Smalltalk browser is based on classification. This approach has proven to be useful for small collections of objects. But when the number of classes becomes large users may feel lost because there is no global view and the structure cannot be rearranged to fit their intuitive perception of the object's space.

Discerning global and local patterns of interaction among classes is critical for tuning and debugging. A few authors have already identified this as an important issue and proposed adequate tools. For example, Böcker and Herczeg [1] introduce a *software oscilloscope* for visually tracking the interactions between objects in a system. The system's dynamic behaviour is inspected by placing obstacles between objects and animating the flow of messages across them. The tool focuses only on microscopic behaviour, however. Brüegge, Gottschalk and Luo [3] describe BEE++, an object-oriented application framework for the analysis of distributed applications. BEE++ is fundamentally an event processing system since it views the execution of distributed activities as streams of events. Event processing is encapsulated in a set of core base classes that are intended to be derived for customization.

Other authors such as Kleyn and Gingrich [17] focus on object behaviour issues. Their tool offers concurrently animated views of the behaviour of an object-oriented system. These views include graphs of invocations between objects. Podgursky and Pierce address the problem [30] of retrieving reusable software components based on sampled behaviour.

Finally, Rubin and Goldberg [31] sketch an object-oriented design approach based on object behaviour analysis and stress the importance of exploration tools to support the design process.

9.1.2.2 Graphical and Spatial Browsing

In the late 1970s Fields and Negroponte, in a visionary paper [10], expressed the need for new clues to find data. Among the many approaches they envisioned for locating information are spatial referencing and proximity. Shortly after, Donelson [7], Bolt [2], and Herot [14] published papers about spatial management of information which apply many techniques for information exploration and inspection that will serve as a basis for future systems. They introduced the *spatial data management system* (SDMS) concept, whereby information is expressed in graphical form and presented in a spatial framework so that the information has a structure that is more obvious than in a conventional database. Herot argues that: "in this way the user can find the information he seeks without having to specify it precisely or know exactly where in the DBMS it is stored."

More recently, Caplinger [5] has described a sophisticated browsing tool with a graphical spatial interface that is, in fact, an evolution of the original SDMS idea. A further elaboration of SDMS is BEAD [6], a system for the visualization of bibliographical data. In BEAD, articles in a bibliography are represented by particles in 3-space. The system uses physically based modelling techniques to take advantage of methods for the approximation of potential fields. Interparticle forces tend to make similar articles move closer to one another and dissimilar ones move apart, so that the relationships between articles are represented by their relative spatial positions. We may also mention the N-Land system [18], which addresses the problem of visualizing higher dimension information spaces.

The growing interest on hypertext systems generalized the use of browsing as a mechanism for information access. Many things have been written recently about hypertext browsing and hypertext navigation, and we will just mention a few works that seem to deserve particular interest in the context of this work. SemNet [8] is a system for the three-dimensional visualization and exploration of large knowledge bases that promotes a hypertext-like navigation paradigm. Feiner's work addresses the problem of how to conveniently display hypertext structures [9] so as to facilitate hypertext navigation.

Another interesting approach is described by Stotts and Furuta [34]. The basic idea is to replace the usual directed graph of an hypertext system by a Petri net. Unlike a directed graph, a Petri net also allows the specification of *browsing semantics*, i.e. the dynamic properties of a reader's experience when browsing a document. So, Petri nets add to the hypertext system access control capabilities based on a formally sound mechanism. The authors describe the α-Trellis system that has been implemented to experiment with the Petri-net-based model. This approach is also discussed in [28] where it is used to explore hypertext systems with an affinity browser.

A sophisticated browsing tool with advanced capabilities for databases has been developed by Stonebraker [33], which combines query refinement techniques and browsing. Jones has described a personal filer with interesting retrieving capabilities [15]. His system, ME, is a database of files connected through links which represent weighted terms. A

retrieval request is a set of terms, and a spreading activation process is used to match the files that are most relevant. Finally we cite a browsing tool for specific databases; Gedye [11] has discussed the problems associated with accessing information related to chip design, and described a browsing tool to inspect the contents of a chip design database.

9.2 Browsing Requirements

To illustrate our browsing requirements we will use a simple paradigm. Suppose we have an information base relative to a city. We need a *city browser* which can guide visitors to plan their stay. For example, suppose we arrive at a hotel and want to go to eat. We would like the city browser to help us choose a restaurant which is geographically close, within an interesting and safe walk (or a place easy to reach and park), with good food, nice surroundings, good service and within our budget.* It is obvious that we have multiple criteria for our choice and it will be very difficult to find a restaurant that is best in all. We need, therefore, to be guided to reach a compromise. We should also be aware that restaurants do not always advertise all their points (especially their shortcomings). They have, therefore — like encapsulated objects — hidden information which we can only get from persons that have been there.

To begin, we should point out that if the number of restaurants is small then we don't need sophisticated browsing tools. We can explore each one of them according to the multiple criteria, while keeping the rest in the back of our mind. This approach, however, breaks down when the number of objects and criteria becomes large.

The first requirement for effective browsing is a notion of locality. The browser should present us first with the choices that are *close*. Close implies a measure of distance which does not necessarily have a single interpretation. For instance it can be geographically close, public-transportation close, etc. Each definition of closeness is within a certain context. The browser should, therefore, be capable of dealing with many contexts. Each context defines a measure of affinity between the objects we are looking for, in this example city locations. We should also be in a position to change contexts in our browsing or combine contexts relating independent selection criteria.

The second requirement is that the measure of distance should be able to change dynamically. For example, time distances between locations can vary with traffic. The browser should be able, therefore, to deal with quickly changing definitions of closeness.

The third requirement is that we need a notion of set-at-a-time navigation. The browser should present us with many choices which could be pursued in the information space. There are two reasons for this requirement. First, the immediately next objects should all be presented to allow other more subjective criteria to be considered. Second, if we insist on point-to-point navigation we may reach many dead-ends and be forced to backtrack. Backtracking is very confusing especially when trying to find an object according to multiple criteria.

---

* Such a system was implemented at Bell Labs for New York city restaurants.

Finally, users should be able to visualize the information space they are searching. We need, therefore, to project a multidimensional information space into a two dimensional screen. This projection should somehow preserve the definition of closeness and give a good user interface for identification of choices.

To summarize, we need a browsing capability which can incorporate:

- a multidimensional space;
- a measure of distance among objects defined according to a certain context;
- a facility for dealing with many contexts independently or in combination;
- a dynamic environment where measures can change;
- a set-of-objects-at-a-time navigation;
- visualization of contexts in two dimensions.

9.3 The Affinity Browser

We describe in this section an approach to browsing based on the concept of *affinity*. Our approach, the *affinity browser*, is a tool for the exploration of object relationships expressed as affinity between objects that fulfils the requirements discussed in section 9.2. The affinity browser is a generic browsing tool for the exploration of information systems. As a generic tool it is meant to be tailored to specific browsing activities. The tailoring is accomplished in essentially two ways. First, by defining the appropriate affinity metrics to describe object relationships of interest among the objects of the system. Second, by adding concepts and visual features that enhance the navigation guidance of the associated search space.

Most of the browsing tools that have been discussed in the previous sections support either point-to-point navigation based on hierarchical structures (e.g. the Smalltalk browser), or they rely on spatial relations for navigation. Our approach is based on the concept of affinity that can be appropriately expressed in visual terms as a spatial relationship: proximity. Objects that appear close in the representation space are more strongly related than objects that lie farther apart. A significant advantage of this approach is that a large spectrum of object relationships can be expressed in terms of affinity provided that we can devise metrics defined on the objects of the system that appropriately portray the relationships in terms of affinity.

The first step for the realization of a visual representation of a relationship among objects portrayed in terms of affinity is the choice of a metric that satisfactorily represents the relationship. The second step is the construction of a multidimensional placement of the objects based on the affinity information. The dimension of the space, the coordinates and the measure of distance are chosen in such a way that the position of each object conveys its relationship to the others. Objects that appear close together should have an affinity to each other. Finally, the object placement needs to be visualized in order to provide navigation support for the user. A detailed discussion of the affinity browser can be found in [28].

Affinity is a powerful conceptual relationship that humans utilize in everyday life to construct a cognitive structure over a generally loosely structured world. One of its important characteristics is that it is highly *context* sensitive. A set of objects that are close in one context can appear quite unrelated in another context. Furthermore, different views of the same set of objects relating to different contexts can be displayed simultaneously and thus complement one another. Adding new views increases, therefore, the user's understanding about these object relationships.

Once affinity is visually represented, users perform proximity-based navigation. Because users can explore different contexts, the browser should allow them to explore the system by choosing, at each step, the context that seems the most appropriate for the next move and update the other views accordingly. The set of coordinated views are called synchronized views. This capability seems convenient since objects that appear close together in one view may lie far apart in another view. Conversely, the user may wish to pursue many explorations concurrently, so the browser should also allow for independent views. These aspects will be discussed in more detail in the next section.

9.3.1 The Affinity Browser Exploration Paradigm

The intended usage of the affinity browser is the exploration of an information space assisted by visual representations of object relationships. Each such affinity can be explored through an affinity browser.

Figure 9.1 represents the typical layout of an affinity browser. Each of the round icons represents an object. The black icon in the centre of the browser is the *marked object*. The marked object is the object around which exploration recurs; users usually select, or *mark* an object, and then explore the objects in its neighbourhood. Eventually, during the exploration they will find an object that appears to be more appropriate, in which case they may select it as the new marked object.

The selection of a new marked object has two main consequences. First, the new marked object is displayed in the centre of the browser. Second, the set of objects that appear in the browser are those that correspond to the new marked object's neighbourhood. As a consequence of marking a new object, some objects may disappear from the representation while others may become visible.

In terms of exploration concepts, marking a new object corresponds to a shift in perspective. The user chooses a new navigation focal point and then explores the neighbourhood of the new marked object.

In a typical browsing session users select either an object they are acquainted with if they already have some knowledge of the information space or they selected one of the entry points that may be provided by the system.

An exploration path can be characterized by the sequence of marked objects. These may act as exploration landmarks and it may be interesting to provide a set of exploration paths that represent relevant guided tours.

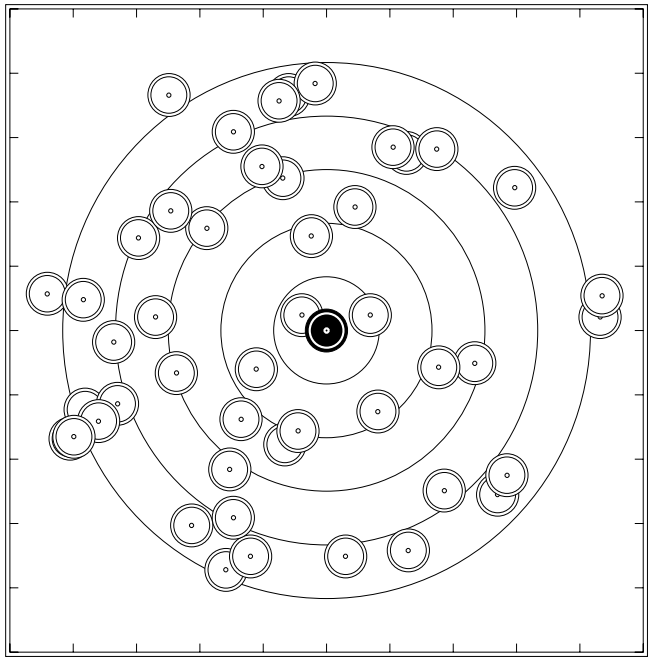

Figure 9.1 *Typical layout of an affinity browser representing an affinity context. The black icon represents the* ***marked object****.*

9.3.1.1 Affinity Neighbourhood

An affinity browser does not usually show all the objects of an affinity context at a time. The displayed objects are those that lie within a user-defined neighbourhood of the marked object. More precisely, the neighbourhood of an object is controlled by a parameter $\varepsilon \in [0, 1]$ which represents a discriminant threshold: only the objects that have an affinity higher than ε relative to the marked object are displayed.

Alternatively, the user may specify the maximum number of objects to appear in the display. In practice this is the most commonly used way of specifying the visual neighbourhood range. The reason is that by keeping the same number of objects during exploration the user avoids situations where the system does not provide enough choices (e.g. few objects displayed), or situations where the browser presents too many choices in a cluttered display.

The notion of set-of-objects-at-a-time navigation results from limiting the displayed objects to those that lie in the specified neighbourhood of an object. This set represents the inspection alternatives that the browser offers concurrently to the user. Although the "radius" of the neighbourhood can be changed at any time, it is an essential assumption of our approach that proximity-based navigation is a convenient exploration paradigm for most exploration or inspection tasks. Further, we see the neighbourhood restriction rather

as a feature than as a limitation. Once users locate a region of interest they should be presented only with the choices that are close in its exploration context.

9.3.1.2 Synchronised Affinity Browsers

The proximity-based navigation provided by an affinity browser is mainly intended for "fine-grained" exploration. That is, once users have identified an interesting region, they explore the alternatives that are close in order to select the most appropriate. However, when users are exploring the information space "at large", local navigation alone is usually not enough.

A powerful mechanism used in human mental processes is association. For example, users proceed by association to recall entities that are close to a given entity. This mental process corresponds, in terms of browsing, to proximity-based navigation. A slightly more elaborate mental process consists of focusing on an object, exploring its neighbours, and investigating how the neighbouring objects in the present context are related in another context, and then exploring the objects that are close in the new context. This is a powerful process since it allows us to reach objects that are not closely related in the first context. Loosely speaking, we may say that exploration is based on transitive association; navigation is proximity-based but by alternating the navigation context the user can reach many other interesting objects. The mechanism that we provide to support this kind of transitive associations is the synchronization of affinity browsers. The synchronization of the affinity browsers implies that the object under inspection in one browser is also highlighted in the others. Users may pursue exploration in any of the browsers and the same path is followed in the others provided the inspected object also belongs to the latter context. We may recall here that two objects that are close in one context might not be close, or may even be unrelated, in another context. Figure 9.2 shows a set of four synchronized browsers. Synchronized views allow users to inspect objects that would otherwise be unreachable if navigation is based on just one exploration context. This stems from the fact that, in one browser objects that are not related to the marked object are normally not displayed. So, to reach non-related objects the user needs to switch to another browser for which the objects are related in the displayed context. This emphasizes the notion of navigation based on the strict neighbourhood of the marked object. However, the browsers allow users to display objects that are not directly related but are related by transitivity.

When objects are transitively related, their affinity is calculated either by a max-min transitivity rule or by a max-product rule. Refer to [28] for a detailed discussion about these operations.

Finally, the user may also explore the information space based on multiple independent browsers or a combination of synchronized and non-synchronized browsers. The synchronization of the browsers is not a symmetric mechanism: saying that browser (a) is synchronized with browser (b) does not imply that browser (b) is synchronized with browser (a). To obtain two-way synchronization the user needs to specify it explicitly.

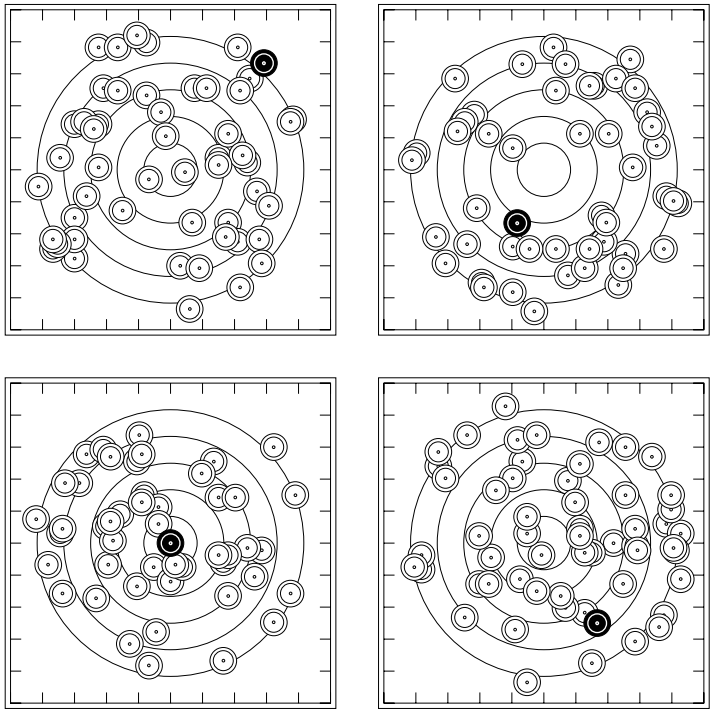

Figure 9.2 *Synchronous affinity browsers. The black icon represents the marked object. The user is performing exploration in the lower left browser where the marked object appears in the centre. Since the browsers are synchronized, the marked object is the same in all the browsers.*

9.3.1.3 Exploration Based on Dynamically Evolving Affinity Contexts

As we stated in our browsing requirements, affinity browsers are intended to provide navigation guidance based on dynamically evolving object relationships. The browser provides such support essentially in two ways. First, it is able to track in a visual way and in interactive time-evolving relationships. Second, the browser provides for a degree of visual feedback where the movement of the visual objects gives the illusion of dynamic motion and dynamic interaction. Both aspects are addressed in more depth in [28] and [25].

One difficulty that users may find with dynamically evolving affinity contexts is that the changes in object relationships may make some objects disappear from the representation and others may show up due to the neighbourhood-restricted display. From our experience, this is quite cumbersome for unstable relationships that evolve at a fast pace.

9.3.2 Architectural Elements of an Affinity Browser

The architectural foundation of the affinity browser relies on an approach to software construction based on the composition of software components. Such an approach emphasizes modularity and careful study of component interfaces in order to achieve reusability and flexibility in software configuration. This flexibility is needed for the affinity browser since the idea is to provide a generic architecture that can be configured to meet the exploration requirements that a specific browser is intended to support.

9.3.2.1 Affinity Engine and View Engine

An affinity browser is comprised of two main units: the *affinity engine* and the *view engine*. The affinity engine is responsible for the management of tasks that are related to the translation of object relationships into a standard form of affinity representation.

The view engine is responsible for display and user interaction management. The affinity engine and the view engine communicate through well-defined protocols. The affinity engine often incorporates application-domain-dependent functionality in order to enhance navigation guidance with domain dependent-features. Similarly, the view engine can also incorporate visual features specific to the application domain and we frequently use this capability, in particular for financial tools.

9.3.2.2 Translucency: One Browser, Multiple Contexts

In our architecture, a browser can display multiple contexts simultaneously. This capability is made available by the view engine that supports a stack of translucent views so that the user can see through the views those that lie behind. The user can specify the desired degree of translucency from completely transparent to completely opaque. In a transparent view, no objects are visible. In an opaque view, objects hidden behind a front view do not show up. The superimposition of views is displayed with a visual effect of depth cueing: views progressively fade away from front to back.

The use of translucency is quite effective because it allows for the simultaneous exploration of many contexts on the same visual space. As a rule of thumb, in order to be useful the number of displayed views should not usually exceed four since the visual fading effect makes some views unreadable. Translucent visual layers are also effective to display domain-dependent information such as names, visual cues, transient information and alarms.

The interaction protocols between the view engine and the visual layers is well-defined, which allows the dynamic insertion of new layers into the view stack. The main advantage of having multiple views displayed in two dimensions is that lengths and distances can be compared visually, which is not usually the case when display relies on three-dimensional techniques since projection distorts distances.

9.3.3 User Interaction and Event Management

In order to conveniently support interaction with multiple superimposed visual layers, the view engine provides an event distribution mechanism through which events from many sources are distributed to the various layers that are responsible for reacting to them. When a new event is queued, it is sent first to the topmost layer, which is asked if it is interested in the event. If the layer is not interested or if the layer does not consume the event, then it is sent to the next layer in the view stack. The operation is applied recursively down the view stack until either the event is consumed or the bottom of the view stack is reached.

The order of the visual layers can be changed interactively by the user. Typically, users bring the layer with which they want to interact to the top of the stack. Furthermore, visual layers can be added to and deleted from the stack. A new visual layer is inserted, by default, at the top of the stack. Object relationships displayed in different visual layers of the same browser can be either synchronized or not, much in the same way as object relationships are displayed in different browsers.

The event distribution mechanism plays an important role in implementing coupled cooperative strategies between the visual layers. In fact, one of our design goals was to define an architecture for the view engine independent of the application domain. To achieve this goal, the interaction between the view engine and the visual layers only supports application-independent operations and not intended to be extended. We decided to provide flexibility in the way cooperation between views can be specified through an extended event distribution mechanism that acts as a messaging backbone.

The event distribution mechanism allows visual layers to communicate spontaneously or in reaction to user-initiated events. Additionally, the browser can be dynamically controlled by other applications that send events through the event distribution mechanism.

We applied the idea of external browser control to a financial application that displays real-time evolving relationships [29]. The application, which runs most of the time without user interaction, implements various display strategies aimed at highlighting important financial instruments relationships. The display and the relative position of the visual layers changes under the control of another application that monitors interesting investment opportunities. This approach to browsing control can be used to provide automatic navigation for dynamically evolving system.

To summarize, the affinity browser architecture has the following desirable characteristics for an exploration tool:

- ***Versatility***. Allows users to inspect the underlying system through object relationships expressed in terms of affinity. The exploration can be based both on static or dynamic relationships, and the exploration perspective can be either local or global.
- ***Composability***. Users can navigate based on multiple object relationships used independently or in combination. Multiple views can be active concurrently.
- ***Extensibility***. New object relationships can be easily added to the exploration tool and combined with previously defined ones.

9.4 The Affinity Browser by Example

An intuitive way to describe the affinity browser approach is to say that we "measure" object relationships in such a way that the measurements translate the relationships into object affinities. Alternatively, we can say that we quantify a relationship in order to express it in terms of object affinity or proximity. For the affinity browser, these measurements are always performed between pairs of objects and are called metrics (refer to [28] for a formal presentation of these concepts).

As we may easily anticipate, one of the critical issues related to affinity browsing is the definition of metrics that portray interesting object relationships. We provide here a few examples of such metrics describing both static and dynamic relationships. Our main goal is to illustrate how the affinity browser can help one to understand particular aspects of an object-oriented environment, and provide typical examples of the kind of information an affinity browser is intended to provide for a system.

We first discuss metrics based on static analysis of class relationships. This kind of analysis is usually important to assess design and to understand architectural articulations; it provides insight into the relationships among classes without actually executing the code. Therefore, the information is primarily extracted by source code analysis.

Next we address the issue of extracting relationships corresponding to the dynamic behaviour of the system. We can identify interesting relationships among both classes and objects. Metrics to portray such relationships are based on dynamic analysis that consists of collecting statistical information, or simply frequency data during a system's execution.

The analysis can be performed either dynamically, in which case the display of the relationship is synchronized with the execution, or it can be off-line based on the information collected. In the latter case, the exploration phase resembles static analysis since the relationships do not evolve dynamically. It is also possible to collect data about the dynamic behaviour of the system and perform the analysis off-line. The advantage is that the analysis can be performed at the user's pace while still allowing for dynamic display.

9.4.1 Class Relationships

We discuss in this section three examples of metrics aimed at revealing class relationships. The first example deals with portraying functional commonality among classes. As a result of inheritance, derived classes inherit functionality from their base classes, and this raises the issue of the extent to which classes differ. The example discusses metrics related to this issue.

The second example deals with class acquaintances. In order to perform their tasks, the methods of a class send messages to other classes to invoke services. Patterns of interaction between a class and its environment may provide useful information about the required working environment for the class. We discuss metrics intended to reveal class acquaintances.

The third example addresses the problem of class relationships related to object *birth* and *death*. More specifically, we are interested in knowing which classes are instantiating and freeing objects. Because we are focusing here on relationships among classes, we consider that two classes are related if one class instantiates or frees objects of the other class.

It should be noted that the extraction of information for building such metrics depends considerably on the environment and on the language used to define the classes. In particular, with strongly typed object-oriented languages such as C++ and Eiffel, relationships like those of the first two examples are usually more accurately portrayed than when metrics are derived from classes implemented with weakly typed languages since, with strongly typed languages, relationships among classes are mostly statically defined.

9.4.1.1 Functional Commonality

In this example we construct a metric aimed at portraying the functional commonality among classes. For the sake of concreteness, the metric construction is illustrated with the set of classes $C = \{C_0, \ldots, C_8\}$ depicted in figure 9.3. Following inheritance rules, classes recursively inherit methods from their superclasses. We further assume that a class can redefine the methods inherited from its superclasses. Let $M(X)$ be a function that returns the set of methods in the interface to class X. For instance, $C_3 = \{a, b, g, h\}$. With this metric we want to convey the extent to which classes provide common functionality. The measure of affinity between two classes can, therefore, be expressed as the proportion of methods that are common to the two classes relative to the total number of the methods defined in both classes. As a candidate measure we define the affinity $A_1(X, Y)$ between class X and class Y by the function:

$$A_1(X, Y) = \frac{\text{card}(M(X) \cap M(Y))}{\text{card}(M(X) \cup M(Y))}$$

where card() is a function that returns the cardinality of a set.

Suppose now that we want to emphasize the fact that redefined functionality might differ from inherited functionality. We can modify slightly the affinity measure for the case of redefined functionality. Let m be the inherited method and m' be its redefinition. In the case where both m and m' appear in $\text{card}(M(X) \cup M(Y))$ then for the affinity calculation we consider $m = m'$ in $\text{card}(M(X) \cap M(Y))$ while in $\text{card}(M(X) \cup M(Y))$ we take $m \neq m'$. This produces a slight reduction of the affinity between classes where one redefines a method from a superclass (such as class C_1). From the affinity function we can derive the table 9.1 of pairwise affinities.

Figure 9.4 shows a view of the affinity browser depicting metric $A_1(X, Y)$ applied to the classes of figure 9.3. In figure 9.4, the highlighted class, C_4, is the *marked* item selected by the user. Therefore, the exploration is centred on it and the browser displays the items that lie inside the neighbourhood of the marked item, where the neighbourhood is defined as the set of objects for which the affinity relative to the current object is higher

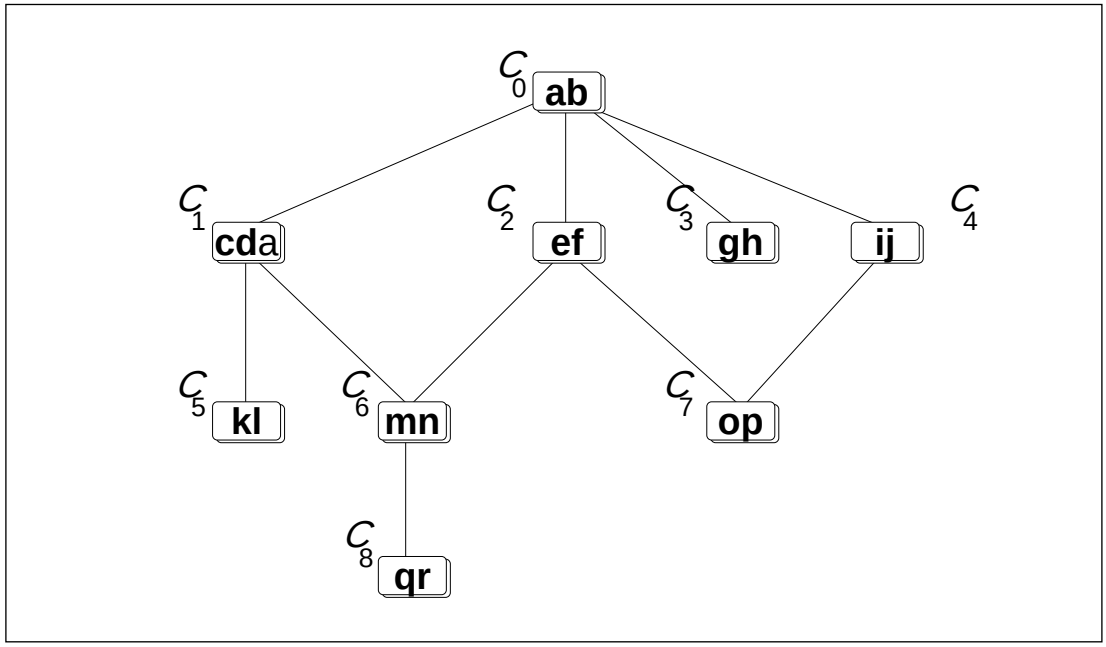

Figure 9.3 *Inheritance structure of a set of classes.*

C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	
2/5	1/2	1/2	1/2	2/7	2/9	2/8	2/11	C_0
	2/7	2/7	2/7	2/3	4/9	2/11	4/11	C_1
		1/3	1/3	2/9	4/9	1/2	4/11	C_2
			1/3	2/9	2/11	2/10	2/13	C_3
				2/9	2/11	1/2	2/13	C_4
					4/11	2/13	4/13	C_5
						4/13	8/11	C_6
							4/15	C_7

Table 9.1 *Functional commonality: pairwise affinity.*

than a chosen value. In this case, however, due to the small number of items, they are all displayed.

9.4.1.2 Metrics Based on Binary Vectors

Many other metrics can be defined to reveal functional commonality. A particularly interesting approach relies on metrics based on binary data. The interest in using binary vectors

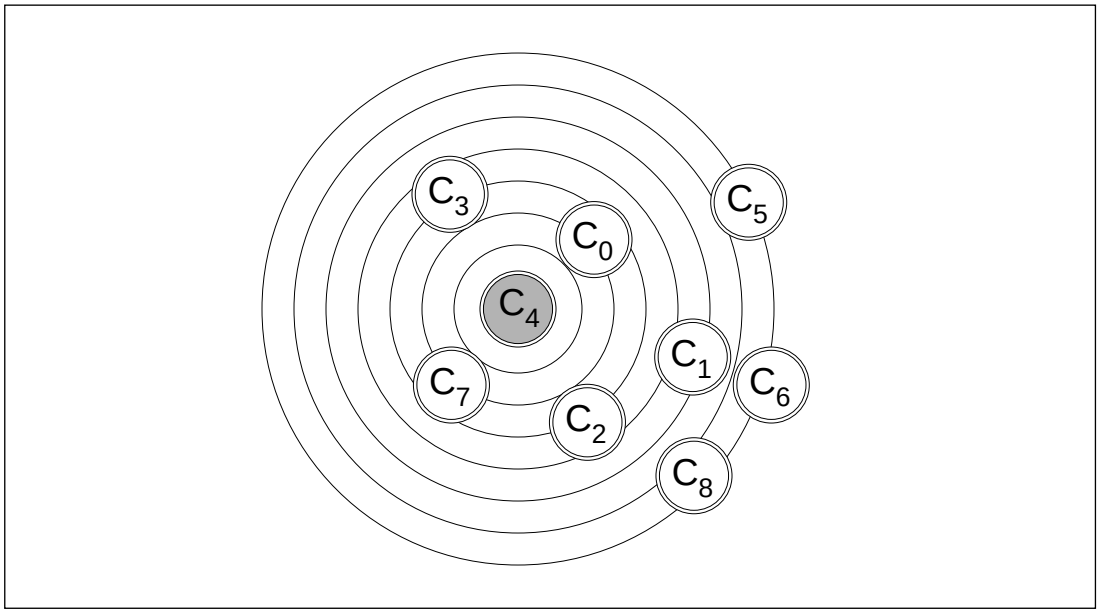

Figure 9.4 *Affinity browser display showing a set of classes.*

to build metrics is that many relationships can be expressed in terms of binary vectors to which we can apply a set of "standard" operations to measure their similarity.

In order to apply these metrics to portray functional commonality we assign to each class a binary vector of length l, where l represents the number of distinct method signatures in the system. Each entry of the vector is associated with a method signature. Referring to the set of classes depicted in figure 9.3, the binary vector takes the form:

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r

Each entry contains a Boolean value that tells if the associated method signature is present or absent in the class. For example, the binary vector associated with class C_0 looks like:

1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

and the vector associated with class C_6:

1	1	1	1	1	1	0	0	0	0	0	0	1	1	0	0	0	0

The construction of an affinity metric from binary vectors consists essentially in measuring to what extent vectors match. These can be defined based on the following auxiliary parameters:

$$\psi_{11} = \sum_{k=1}^{l} \min(x_k, y_k) \qquad \psi_{10} = \sum_{k=1}^{l} x_k - \psi_{11}$$

$$\psi_{01} = \sum_{k=1}^{l} y_k - \psi_{11} \qquad \psi_{00} = l - (\psi_{01} + \psi_{10} + \psi_{11})$$

where x and y represent two binary vectors. Ψ_{11} counts the number of times 1 appears simultaneously in the corresponding entries of x and y; Ψ_{10} counts the number of times 1 appears in x and 0 in y for corresponding entries; Ψ_{01} counts the number of times 0 appears in x and 1 in y for corresponding entries; and Ψ_{00} counts the number of times 0 appears simultaneously in the corresponding entries of x and y. So Ψ_{11} and Ψ_{00} count the number of entries in which x and y agree, while Ψ_{10} and Ψ_{01} count the number of disagreements.

We propose three metrics to portray functional commonality based on the binary vector representation. The first metric is

$$A_2(X, Y) = \frac{\Psi_{11}}{l}$$

where X and Y represent the classes from which the binary vectors x and y are derived. $A_2(X, Y)$ assesses binary vector similarity in terms of 1-consensus relative to the length of the binary vectors.

The second metric is

$$A_3(X, Y) = \frac{\Psi_{11}}{\Psi_{11} + \Psi_{10} + \Psi_{01}}$$

With this metric the proportion of the 1-consensus is evaluated relative to the number of entries of the vectors excluding those that correspond to a 0-consensus; that is, the metric assesses affinity in terms of 1-consensus relative to disagreement. This means that $A_3(X, Y)$ is equivalent to $A_1(X, Y)$.

The third metric

$$A_4(X, Y) = \frac{\Psi_{11}\Psi_{00}}{\sqrt{(\Psi_{11} + \Psi_{10})(\Psi_{11} + \Psi_{01})(\Psi_{00} + \Psi_{10})(\Psi_{00} + \Psi_{01})}}$$

measures binary vector correlation but is not a metric similarity index as are $A_3(X, Y)$ and, consequently, $A_1(X, Y)$.

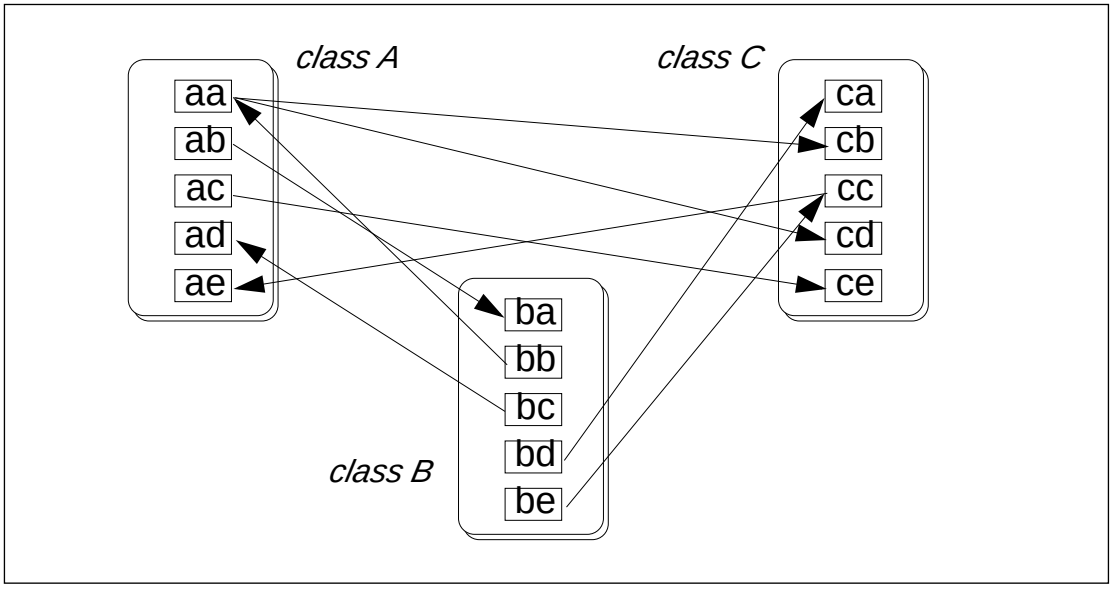

Figure 9.5 *A set of cooperating classes. The three classes cooperate by service exchange. Each slot represents the body of a method and the arrows represent the activation of a method from the body of another method.*

9.4.1.3 Class Acquaintances

The functionality of a class is not usually self-contained. Methods belonging to a class can invoke services from other classes. This perspective corresponds to a commonly accepted view of object-oriented systems as sets of collaborating objects.

We are interested in understanding patterns of collaboration between objects. However, collaboration has many aspects. We can focus, for instance, on the relationships between classes that can be observed by static analysis of the source code. Alternately, we may focus on dynamic acquaintances of classes measured by observing message sending patterns between objects of the classes.

Both perspectives are interesting and are, to a large extent, complementary. The former perspective usually reflects design decisions since “hard coded” relationships usually materialize links defined at the architectural level. Such links represent the required working environment for a class. But this perspective may fall short of providing an accurate picture if we are looking for working acquaintances between classes. In this case the latter perspective may be more helpful.

In practice, the collaboration patterns revealed by the two perspectives usually differ significantly. However, the analysis of the differences might offer useful insight about mismatch between the collaborations that have been foreseen by the designer and those that show up in specific execution contexts. We start with a metric intended to portray static class acquaintances. That is, acquaintances that can be determined without actually executing the methods of the class.

Figure 9.5 represents the analysis context for such a metric. Each class contains a set of methods and the methods activate methods belonging to other classes that, in turn, trigger other methods as well. So the execution of a class's method usually involves the execution of methods from many classes.

Let I_K^J denote the number of times class K invokes methods from class J, and let I_K denote the total number of invocations from class K to any other class. The following is a candidate metric to portray class acquaintances:

$$A_5(K, J) = \max\left(\frac{I_K^J}{I_K}, \frac{I_J^K}{I_J}\right)$$

which means that the acquaintance affinity between two classes is defined as the maximum of relative invocation frequency of both classes. We may notice, however, that many different functions can be used instead of max() to combine the two "one-sided" acquaintances. We can define a more general metric as follows:

$$A_5(K, J) = 1 - \log\left(\frac{\left(\lambda^{1-\frac{I_K^J}{I_K}} - 1\right)\left(\lambda^{1-\frac{I_J^K}{I_J}} - 1\right)}{\lambda - 1} + 1\right)$$

where $\lambda \in (0, 1) \cup (1, \infty)$ and $\log(x)$ has base λ. This metric is inspired from a function proposed by Frank [1] to define the union operation on fuzzy sets. The reader might want to refer to [28] for a detailed discussion about other functions that can be used in this context. This way of doing things may suggest an interpretation where I_K^J / I_K represents the affinity degree of an element J to affinity set K which depicts the unilateral affinity acquaintance between class K and the other classes.

9.4.1.4 Class Acquaintance Similarities

We may also be interested in class acquaintance similarity. In other words, we want to discover to what extent classes match in terms of the services they ask for from other classes. Let $s_{C,m}$ denote a service; that is, s is an association of a class name C and one of its methods represented by its method's signature m. Let f_K^s denote the frequency of invocation of service s from inside the methods of class K. We can associate to each class K a vector v_K with entries containing f_K^s. The dimension d of vector v_K is equal to the number of different services invoked by the classes of the system.

So, the collection of classes can be represented in the d-dimensional space of the services, where each class will appear as a point. The idea is that classes lying close together in this space ask for similar services. We may want to modify slightly the service weight-

ing scheme to improve selectivity. Let i_s denote the number of classes from which service s is invoked, and let n denote the number of classes. We can define

$$L_s = \lceil \log_2 n \rceil - \lceil \log_2 i_s \rceil + 1.$$

A service weighting proportional to $f_K^s \cdot L_s$ will assign larger weights to services which are invoked with high frequency in individual classes, but that are only invoked by a few classes. This type of weighting scheme improves substantially both recall and precision when applied to document retrieval [32]. Finally, we can define a distance metric between two classes K and J as the Euclidean distance between the associated vectors v_K and v_J.

9.4.2 Creation and Destruction Relationships

In an object-oriented environment, objects are usually created and destroyed by other objects. Understanding creation and destruction relationships is important for many reasons. First, it provides essential information about which classes are managing the object population in the system and, in particular, which are the typical procreators of objects that provide specific kinds of services. Second, this understanding is crucial for debugging and, in particular, memory allocation related errors. As a matter of fact, the very nature of object-oriented systems as sets of cooperating agents raises the problem of object cleanup. Designers need to decide who is responsible for freeing the objects. It is often difficult to assign this responsibility to its creator, especially if the creator is not the consumer of the services. The non-destruction of stale objects may become a particularly important issue in the absence of automatic garbage collection.

Creation and destruction relationships can be analyzed either statically or dynamically. Similar to acquaintance relationships, dynamic and static analysis provide different perspectives on the creation and destruction relationship. Static analysis based on source code scanning essentially provides information about the structure of the creation and destruction process. We can learn, for instance, which classes can create and destroy instances of given classes.

Dynamic analysis provides another perspective on the relationship by showing which class instances are actually creating and destroying objects, and also how many objects are created and destroyed. However, the static perspective falls short of portraying an important aspect of software execution: execution phases. A typical software system or subsystem goes through a number of execution phases. It may start with an initialization phase, then alternate through several phases. Different phases become evident by analysis of both interclass acquaintances and creation and destruction relationships. Entering a new phase usually corresponds to a significant modification of interaction patterns and an intense activity of object destruction — for phase cleanup — and creation of new objects for the new execution phase.

The information about the creation relationship can be represented by a matrix like

$$C = \begin{bmatrix} C_{A,A} & C_{A,B} & C_{A,C} \\ C_{B,A} & C_{B,B} & C_{B,C} \\ C_{C,A} & C_{C,B} & C_{C,C} \end{bmatrix}$$

where $C_{X,Y}$ represents the number of times creation of an instance of class Y can be identified inside the source code specifying class X, if we are in the context of static analysis. In the context of dynamic analysis, $C_{X,Y}$ represents the number of times instances of class X create instances of class Y during a given time interval. In order to explore execution phases we can collect data for several time intervals that should reveal the changes in creation patterns. The destruction relationship can be represented by a matrix D that has a similar form to C where entry $D_{X,Y}$ represents the number of times instances of class X destroy instances of class Y during a given time interval.

We can derive a matrix

$$R = \begin{bmatrix} C_{A,A}-D_{A,A} & C_{A,B}-D_{A,B} & C_{A,C}-D_{A,C} \\ C_{B,A}-D_{B,A} & C_{B,B}-D_{B,B} & C_{B,C}-D_{B,C} \\ C_{C,A}-D_{C,A} & C_{C,B}-D_{C,B} & C_{C,C}-D_{C,C} \end{bmatrix}$$

which might represent an acceptable view of the balance between creation and destruction responsibilities. For instance, $R_{X,Y} < 0$ means that class X destroyed more instances of class Y than it created during the time interval under analysis. Many insightful metrics can be derived from the information contained in these matrices.

We convey creation relationships in such a way that classes that are frequently involved in creation (either by creating or by being created) have more affinity and thus cluster together in the representation. A candidate metric is:

$$A_6(X, Y) = \frac{\max(C_{X,Y}, C_{Y,X})}{\max(C)}$$

where $\max(C)$ denotes the maximum value in matrix C. $A_6(X, Y)$ fails to show which one of two classes displaying high affinity is responsible for creation. To obtain such information we may either define a pair of metrics to be used in exploration with synchronized views or create a metric that highlights asymmetry. Both approaches have already been discussed in the context of the formulation of previous metrics.

We provide another metric to convey the balance between creation and destruction. The idea is that instances of a class X that create more instances of another class Y than they destroy, display more affinity while a negative balance in the creation/destruction process reduces the affinity between X and Y.

$$A_7(X, Y) = \frac{\max(R_{X,Y} - \min(R), R_{Y,X} - \min(R))}{\max(R) - \min(R)}$$

9.4.3 Object Relationships

We discussed in the previous section metrics to portray class relationships. The information needed to apply those metrics relies either on static analysis of the class definitions, on dynamic analysis of execution activity, or on both. We may notice, in passing, that many of the metrics discussed could be used to portray object relationships as well. In this section we focus specifically on object relationships that are related to dynamic aspects of the system's execution and, therefore, require dynamic behaviour analysis. Understanding the dynamic behaviour of a set of objects that collaborate to perform a task can provide useful information for reuse and for class management. Dynamic behaviour analysis can be helpful:

- in giving useful hints about the usage a developer intended for a particular class;
- by showing the typical utilization of classes inside an application;
- to tune the performance of classes;
- in providing information for the assessment of class designs;
- in application debugging.

We now discuss candidate metrics intended to portray different aspects of the dynamic behaviour of objects defined in terms of object affinity. In order to perform tasks collectively, objects exchange messages. As a first goal we want to know which objects collaborate closely. Because we are interested in dynamic patterns of collaboration, the information needed to build the metrics is collected by monitoring message passing activity.

Let $O = \{O_1, O_2, \ldots, O_n\}$ denote the set of interacting objects during a given time interval. We may define an affinity metric $A_8(X, Y)$ between object $X \in O$ and object $Y \in O$ by:

$$A_8(X, Y) = \frac{\text{card}(\text{send}(X, Y)) + \text{card}(\text{send}(Y, X))}{\sum_i \text{card}(\text{send}(O_i))}$$

where $\text{send}(X, Y)$ is a function that returns the set of messages sent by object X to object Y, and $\text{card}(x)$ returns the cardinality of a set. So $\Sigma_i \text{card}(\text{send}(O_i))$ represents the total number of messages exchanged during the monitored time interval. With such a metric of affinity, objects that exchange messages frequently will have more affinity and will therefore cluster together in the affinity browser's visual representation.

9.4.3.1 Detecting Object Interaction Asymmetry

However, metric $A_8(X, Y)$ does not show asymmetric interaction patterns. Suppose, for example, that object X sends messages frequently to object Y while object Y seldom

sends messages to object X. Measure $A_8(X, Y)$ will not reveal this fact. In order to expose asymmetry we define two metrics, $A_{9a}(X, Y)$ and $A_{9b}(X, Y)$, that are intended to be used in two synchronized views:

$$A_{9a}(X, Y) = \frac{\text{card}(\text{send}(\text{src}(X, Y), \text{dest}(X, Y)))}{\sum_i \text{card}(\text{send}(O_i))}$$

$$A_{9b}(X, Y) = \frac{\text{card}(\text{send}(\text{dest}(X, Y), \text{src}(X, Y)))}{\sum_i \text{card}(\text{send}(O_i))}$$

We introduce the notion of *source* and *destination* to cope with the asymmetry between the two metrics; the functions src() and dest() return respectively the *source* object and the *destination* object. The role of these functions is to enforce a rule so that an object that plays the source role for a given pair (*X,Y*) in the first metric plays the destination role for the same pair in the other metric. Additionally, the rule copes with pair symmetry so that $A_{9a}(X, Y) = A_{9a}(Y, X)$ and $A_{9b}(X, Y) = A_{9b}(Y, X)$. The rule works as follows: for an affinity context with n objects we generate the $(n(n-1))/2$ pairs that correspond to the upper right half of a matrix which is illustrated for $n = 4$:

$$\begin{bmatrix} (O_1, O_2) & (O_1, O_3) & (O_1, O_4) \\ & (O_2, O_3) & (O_2, O_4) \\ & & (O_3, O_4) \end{bmatrix}$$

The rule specifies that for each pair, the object that appears on the left-hand side plays the role of the source and the object on the right plays the role of the destination. Furthermore, for any such pair (*X,Y*) the roles for the symmetric pair (*Y,X*) are assigned to the same objects. For example, O_1 plays the role of the source and O_3 the role of the destination in both (O_1, O_3) and (O_3, O_1).

When metrics $A_{9a}(X, Y)$ and $A_{9b}(X, Y)$ are represented in synchronized views, the user can spot asymmetry by looking for pairs of objects that appear close in one view and farther apart in the other. These two metrics provide both global information about collaboration as well as information about collaboration asymmetry.

However, if we are mostly interested in detecting asymmetry, it might be more appropriate to emphasize pairwise interaction instead of global interaction affinity. To this end we may consider the replacement of metrics $A_{9a}(X, Y)$ and $A_{9b}(X, Y)$ by $A_{10a}(X, Y)$ and $A_{10b}(X, Y)$. Let

$$\Delta(X, Y) = \text{card}(\text{send}(\text{src}(X, Y), \text{dest}(X, Y)) \cup \text{send}(\text{dest}(X, Y), \text{src}(X, Y)))$$

and

$$A_{10a}(X, Y) = \frac{\text{card}(\text{send}(\text{src}(X, Y), \text{dest}(X, Y)))}{\Delta(X, Y)}$$

$$A_{10b}(X, Y) = \frac{\text{card}(\text{send}(\text{dest}(X, Y), \text{src}(X, Y)))}{\Delta(X, Y)}$$

It is also possible to synthesize information about symmetry of message passing in one metric, although the resulting display might be more difficult to interpret. $A_{11}(X, Y)$ is a candidate measure for such a view. This measure focuses on symmetry of message exchanges and, therefore, suppresses information about frequency of communication.

$$A_{11}(X, Y) = 1 - \frac{\text{abs}(A_{9a}(X, Y) - A_{9b}(X, Y))}{\text{abs}(A_{9a}(X, Y) + A_{9b}(X, Y))}$$

Interaction asymmetry is an important issue at the design level. For instance, Booch [4] identifies three roles for objects in terms of message passing activity:

- **Actor**: an object that operates upon other objects but that is never operated upon by other objects.
- **Server**: an object that never operates upon other objects; it is only operated upon by other objects.
- **Agent**: an object that can both operate upon other objects and be operated upon by other objects; an agent is usually created to do some work on behalf of an actor or another agent.

Such roles can be identified with metric $A_{11}(X, Y)$. The comparison of the roles assigned to objects during the design phase, with the effective role they play in given execution contexts, might be instrumental to assess to what extent reusable software components are used as intended by their designers.

9.5 Conclusion

This chapter dealt with the exploration of object relationships in the context of object-oriented environments. We addressed the important issue of understanding how objects are related because such understanding plays an important role in many key issues related to software engineering such as reuse, debugging and software maintenance. Early object-oriented environment designers have identified these issues and provided browsing tools to help users explore the environment.

We have proposed a new approach to the exploration of an object-oriented environment where object relationships are translated into affinity relations so that the object relationships can be graphically represented in terms of distance: objects that are strongly related appear closer in the representation. The approach has the advantage that many different relationships can be represented and explored with the same tool and with the same explo-

ration paradigm. From the user's perspective, affinity is a very intuitive concept that has the advantage of being easily translated into a visual distance.

References

[1] Hans-Dieter Böcker, Jürgen Herczeg, "What Tracers are Made Of," *Proceedings of OOPSLA/ ECOOP '90, ACM SIGPLAN Notices*, 1990, pp. 89–99.

[2] R. Bolt, "Spatial Data Management," DARPA Report, MIT, Architecture Machine Group, 1979.

[3] Bernd Brüegge, Tim Gottschalk, Bin Luo, "A Framework for Dynamic Program Analysers," *Proceedings of OOPSLA '93, ACM SIGPLAN Notices*, 1993, pp. 65–82.

[4] Grady Booch, *Object Oriented Design With Applications*, Benjamin/Cummings, 1991.

[5] Michael Caplinger, "Graphical Database Browsing," *Proceedings of ACM-SIGOIS, SIGOIS bulletin*, vol. 7, no. 2–3, Oct. 1986.

[6] Matthew Chalmers and Paul Chitson, "Bead: Explorations in Information Visualization," *Proceedings of SIGIR '92*, June 1992, pp. 330–337.

[7] William Donelson, "Spatial Management of Information," *Computer Graphics (Proceedings of SIGGRAPH '78)*, Aug. 1978, pp. 203–209.

[8] Furnas Fairchild, Poltrock, "Semnet:Three-dimensional Graphic Representation of Large Knowledge Bases," in *Cognitive Science and its Applications for Human–Computer Interaction*, ed. R. Guindon, Lawrence Erlbaum, Hillsdale, NJ, 1988.

[9] Steven Feiner, "Seeing the Forest for the Trees: Hierarchical Display of Hypertext Structures," *Proceedings of COIS '88*, Palo Alto, March 1988, pp. 205–212.

[10] C. Fields and N. Negroponte, "Using New Clues to Find Data," Third International Conference on Very Large Data Bases, Tokyo, Oct. 1977, pp. 156–158.

[11] David Gedye and Randy Katz, "Browsing the Chip Design Database," University of California at Berkeley, Computer Science Division, Oct. 1987.

[12] Adele Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading, Mass., 1984.

[13] I. Goldstein and D. Bobrow, "Browsing in a Programming Environment," *Proceedings of the 14th Hawaii International Conference on System Science*, January 1981.

[14] Charles Herot, "Spatial Management of Data," *ACM Transactions on Database Systems*, vol. 5, no. 4, Dec. 1980, pp. 493–513.

[15] William P. Jones, "On the Applied Use of Human Memory Models: The Memory Extender Personal Filing System," International Journal Man–Machine Studies, vol. 25, no. 2, 1986, pp. 191–228.

[16] George Klir, Tina Folger, *Fuzzy Sets, Uncertainty and Information*, Prentice Hall, Englewood Cliffs, NJ, 1988.

[17] M. Kleyn, P. Gingrich, "GraphTrace — Understanding Object-oriented Behaviour Systems Using Concurrently Animated Views," *Proceedings of OOPSLA '88, ACM SIGPLAN Notices*, 1988, pp. 191–205.

[18] Jeffrey T. LeBlanc, "N-Land: A Visualization Tool for N-Dimensional Data," Technical Report Computer Science Department, University of Worcester, May 1991.

[19] Bertrand Meyer, *Object-Oriented Software Construction*, Prentice Hall, 1988.

[20] Amihai Motro, "Browsing in a Loosely Structured Database," *Proceedings of ACM–SIGMOD 1984, International Conference on Management of Data*, 1984, pp. 197–207.

[21] Amihai Motro, "BAROQUE: A Browser for Relational Databases," vol. 4, no. 2, April 1986, pp. 164–181.

[22] Amihai Motro, "VAGUE: A User Interface to Relational Databases that Permits Vague Queries," *ACM Transactions on Office Information Systems*, vol. 6, no. 2, July 1988, pp. 187–214.

[23] Amihai Motro, Alessandro D'Atri, Laura Tarantino, "The Design of KIVIEW: An Object-Oriented Browser," *Proceedings of the Second International Conference on Expert Database Systems*, Virginia, 1988, pp. 17–31.

[24] P. O'Brien and D. Halbert and M. Kilian, "The Trellis Programming Environment," *Proceedings of OOPSLA '87, ACM SIGPLAN Notices*, Oct. 1987.

[25] Xavier Pintado, Eugene Fiume, "Grafields: Field-directed Dynamic Splines for Interactive Motion Control," *Computers & Graphics*, vol. 13, no. 1, Jan. 1989, pp. 77–82.

[26] Xavier Pintado, Dennis Tsichritzis, "An Affinity Browser," Technical Report, Centre Universitaire d'Informatique, University of Geneva, June 1988.

[27] Xavier Pintado, "Selection and Exploration in an Object-oriented Environment: The Affinity Browser," in *Object Management*, ed. D. Tsichritzis, Centre Universitaire d'Informatique, July 1990, pp. 79–88.

[28] Xavier Pintado, "Objects' Relationships," Ph.D. Thesis, Centre Universitaire d'Informatique, University of Geneva, Switzerland, 1994.

[29] Xavier Pintado, "Visualization in the Financial Markets," *VR '94, Proceedings of the Fourth Annual Conference on Virtual Reality*, Mecklermedia, London, 1994, pp. 80–84.

[30] Andy Podgursky, Lynn Pierce, "Retrieving Reusable Software by Sampling Behaviour," *ACM Transactions on Software Engineering and Methodology*, vol. 2, no. 3, July 1993, pp. 286–303.

[31] Kenneth Rubin, Adele Goldberg, "Object Behaviour Aanlysis," *Communications of the ACM*, vol. 35, no. 9, Sept. 1992.

[32] Gerard Salton, *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*, Addison-Wesley, Reading, Mass. 1988.

[33] Michael Stonebraker and J. Kalash, "Timber: a Sophisticated Database Browser," *Proceedings of the 8th International Conference on Very Large Data Bases*, Sept. 1982, pp. 1–10.

[34] David Stotts and Richard Furuta, "Petri-Net-Based Hypertext: Document Structure with Browsing Semantics," *ACM Transactions on Information Systems*, vol. 7, no. 1, Jan. 1989.