Answering Conceptual Queries with Ferret

Brian de Alwis
Dept of Computer Science
University of British Columbia
Vancouver, B.C., Canada
bsd@cs.ubc.ca

Gail C. Murphy
Dept of Computer Science
University of British Columbia
Vancouver, B.C., Canada
murphy@cs.ubc.ca

ABSTRACT

Programmers seek to answer questions as they investigate the functioning of a software system, such as "which execution path is being taken in this case?" Programmers attempt to answer these questions, which we call *conceptual queries*, using a variety of tools. Each type of tool typically highlights one kind of information about the system, such as static structural information or control-flow information. Unfortunately for the programmer, the tools seldom directly answer the programmer's conceptual queries. Instead, the programmer must piece together results from different tools to determine an answer to the initial query. At best, this process is time consuming and at worst, this process can lead to data overload and disorientation.

In this paper, we present a model that supports the integration of different sources of information about a program. This model enables the results of concrete queries in separate tools to be brought together to directly answer many of a programmer's conceptual queries. In addition to presenting this model, we present a tool that implements the model, demonstrate the range of conceptual queries supported by this tool, and present the results of use of the conceptual queries in a small field study.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments; D.2.12 [**Software Engineering**]: Interoperability

General Terms

Algorithms, Theory

Keywords

Software Representation Models, Tool Integration

1. INTRODUCTION

When performing evolution tasks to a software system, a programmer asks many different questions, such as *"where is this method called or type referenced?"* [27], *"which execution path is being taken in this case?"* [27], or *"who last changed this code?"* [31]. We refer to these questions that a programmer *wants* answered as *conceptual queries*.

Programmers use a variety of tools to answer these conceptual queries, such as cross-reference engines (e.g., Masterscope [30]) and dynamic profilers (e.g., gprof [15]). These tools represent different sources of information about the same software system. However, as observed by Sillito *et al.* [27], many of a programmer's conceptual queries are either poorly or only indirectly supported by the *concrete queries* provided by the tools. This mismatch places up to three additional cognitive burdens on the programmer.

First, the programmer becomes responsible for *mapping* a conceptual query onto the available concrete queries and in *scoping* the possible broad results to just those of interest. For example, to search for the program locations where a particular exception is thrown, a programmer using commonly available searching tools in an integrated development environment (IDE) must search for references to the exception type (mapping) and then must manually examine the results (scoping) to find those performing a throw, as compared to those in which a variable is declared of the exception type or some other use.

Second, the programmer may need to *compose* the results of multiple concrete queries to answer a conceptual query. The concrete queries might each need to be performed in a separate tool. Typically, the programmer must then merge the sets of results produced by the different concrete queries to answer the original conceptual query. For example, answering a conceptual query about where instances of a particular Java interface are created requires searching for the instantiators of each of the interface's implementing classes.

Third, answering a conceptual query may require a programmer to *integrate and reason across* several sources of information about the software system. This burden takes two forms. The first form arises when two (or more) information sources have some overlap in the types of information represented. For example, a programmer can choose to consider a program either from its actual runtime behaviour as represented by a dynamic trace, or from the possible behaviours as represented in the source code. In this situation, the programmer must choose between two descriptions of the program's behaviour that differ in the precision provided by their concrete queries [26]. The second form of this burden arises when two (or more) information sources maintain different internal representations for what a programmer would plausibly consider to be the same element. The programmer is then responsible for establishing and carrying this correspondence when making further queries. Establishing such correspondences is trivial for the programmer when there is an exact one-to-one relationship between the tool representations; for instance, it is easy to map between representations of a

ICSE'08, May 10–18, 2008, Leipzig, Germany.

method declaration in two static Java cross-reference tools. Other correspondences, such as when an element bears some similarity to another or when an element corresponds to a group of elements, can be more difficult to make. For example, in examining a dynamic execution trace, a programmer wishing to reason about the callers of a method defined on a Java interface must extend his reasoning to the callers of the implementations of that method.

Each of these cognitive burdens takes the programmer away from her core goal of answering a particular conceptual query. Each of these burdens can be time consuming and distracting. Each of these burdens can lead to data overload [34] and disorientation [8].

At first glance, it is surprising that these problems persist despite significant efforts that have been devoted to tool integration and query languages to support software development. On closer examination, efforts in these two areas have not focused on the core need of composing the results of multiple concrete queries, each of which may return results from a different perspective on the software system. Efforts in tool integration, for example, have typically focused on sharing artifacts between independent tools (e.g., [28]), or triggering transformations upon a change to an artifact (e.g., [35]), rather than on formulating inquiries across tools. Query languages, and tools based on query languages, provide significant power to express and compose queries. But the few implementations that support cross-artifact querying assume an exact correspondence between elements from different artifact domains.

To enable direct support for expressing conceptual queries, this paper introduces a model that supports the composition and integration of different sources of program information to form a single queryable knowledge-base suitable for answering conceptual queries. We call a particular source of information a *sphere*. We provide an implementation of this model in a new software exploration tool, named Ferret. Ferret summarizes how particular program elements relate to the system, structured as answers to different conceptual queries about those elements. We have implemented 36 different conceptual queries. Directly answering conceptual queries means our tool, Ferret, deals with issues of mapping, scoping and composition. Relieving these burdens does not come for free; the spheres over which queries occur must be configured. This configuration is straightforward. We describe the results of a small field study showing promise that the tool will be useful in practice.

We begin in Section 2 where we describe our approach for representing and integrating different sources of program information, the sphere, and describe a framework for establishing the correspondences between elements of different spheres, and how spheres can be combined to produce interesting results. In Section 3, we describe how this model is implemented in Ferret, and discuss how the results may be used to answer the questions. We report on a brief diary study involving four professional programmers in Section 4. We discuss some of the issues that arose from the implementation in Section 5. We examine related work in Section 6, and conclude in Section 7.

2. THE SPHERE MODEL

Core to supporting a useful set of conceptual queries is the need to combine information from different sources about the program. We call different sources of program information *spheres*. For instance, the static Java relationships represented in the source code form a sphere, the revision history recorded in a software repository forms a sphere, and the calls information embodied in a dynamic execution trace forms a sphere. The expression of a conceptual query corresponds to resolving relations against a sphere, which may be a composite sphere (as described in this section).

Table 1: Examples of some possible elements and relations from different spheres (indicated in italics) in a Java-based software system.

Type	Example
Elements	*static*: classes, interfaces, methods, fields, packages *dynamic*: classes, methods *evolution*: programmers, files, revisions, elements, transactions *Eclipse plug-ins*: extension points, extensions, plug-ins, identifiers, Java types
Relations	*static*: implements, extends, declares, calls, instantiates, overrides, returns, has-argument, throws, catches, tests-instanceof, casts, declared-type, gets, sets *dynamic*: calls, instantiates, was-invoked, was-instantiated *evolution*: modified-by, modified-in-transaction *Eclipse plug-ins*: specifies, extends, depends

2.1 Spheres

We define a sphere using the relational algebra. A sphere provides a view of a source providing information about a software system. A sphere S is a tuple

$$S = (E, L, R)$$

where

- E is the set of elements represented in the source;
- L is a set of relation labels describing the types of relationships represented in the source that may exist between the elements;
- $R \subseteq L \times E \times E$ is a set of labelled relations. There may be multiple relations between two elements, but each is treated as a distinct relation with different labels.

Information accessible through existing tools is easily transformed into the relational form of a sphere by expressing the information available in terms of relations between distinct elements. Consider the example of a Java cross-reference tool that displays information based on the static structure of the system. The distinct objects in the tool's domain correspond to the sphere's elements (e.g., the elements in Table 1), and the tool-provided queries correspond to the sphere's relations (e.g., the relations in Table 1). Spheres are used to integrate information from different tools.

Information provided by some tools is best represented by multiple spheres. For example, the results from a dynamic Java tool is highly dependent on the particular dynamic runtime traces that is has been configured to use.

2.2 Composing Sphere Relations

The model supports composing spheres, such as using the results of one sphere to replace the information from another. As a simple example, consider a situation where a programmer has been examining the source code for a Java interface and asks *"which of the implementations of this interface were actually instantiated in this last run?"* Composing the static information about the software system with the dynamic information from a run-time trace allows a tool to answer such a conceptual query. When the composition is over multiple dynamic spheres, this same query can answer the query over several scenarios of use of the program.

There are several ways to compose spheres which we express as using a function f. We express the composition of sphere S_1 by S_2 using function f as:

$$S_2 \circ_f S_1 = (E_1 \cup E_2, L_1 \cup L_2, f(R_1, R_2))$$

We treat the composition of the elements and labels as simple unions of sets as the two spheres may have common elements (i.e., the two

spheres share a common representation, such as many tools use the Java model from the Eclipse Java Development Tools (JDT)), or the two spheres share common labels (i.e., the two spheres provide common operations, such as the static and dynamic spheres both describe calls relations between methods). This definition pushes the effect of composition to the function on the relations.

We have found use cases for four particular composition functions. Composition functions f are not required to be commutative. We adopt the notation and semantics as described by Schmidt & Ströhlein [25]: $\cup$ and $\cap$ are respectively union and intersection of elements; and $\sqcup$, $\sqcap$, $-$ are respectively union, intersection, and difference of relations. We provide simple illustrative examples of each function in terms of the static and dynamic spheres.

Union: Union includes all relations from each sphere involved in the composition:

$$f(R_1, R_2) = R_1 \sqcup R_2$$

The union can be used to combine dynamic profiles resulting from different runs of a program. Union is commutative.

Replacement: Replacement causes those relations of R_1 with a label from R_2 to be removed and replaced with the relations from R_2:

$$\begin{aligned} X &= \{(l,x,y) \in R_1 \,|\, l \notin L_2\} \\ f(R_1, R_2) &= X \sqcup R_2 \end{aligned}$$

Replacement can be used when composing a static sphere with a dynamic sphere, replacing the calls relations with what actually happened during the execution. Replacement is not commutative.

Transformation: Transformation performs a join of the relations of R_1 by a subset of R_2 for a particular label l_r in R_2.

$$f(R_1, R_2, l_r) = \{(l,x,z) \,|\, (l,x,y) \in R_1 \wedge (l_r,y,z) \in R_2\}$$

Transformation can be used to apply blanket restrictions to all relations. For example, the relations of a static sphere (R_1) can be restricted to those methods actually invoked in a dynamic trace (represented in R_2), by defining l_r = was-invoked to be an identity relation where $(x, \text{was-invoked}, x) \in R_2$ if x is either not a method, or x is a method invoked in the trace.

There are other possible composition functions, such as intersection, subtraction, and symmetric difference, but we have not found a use case to consider their inclusion at present.

2.3 Composing Sphere Elements

Our definition of sphere composition defines elements of the sphere resulting from the composition as being the union of the sets of elements from the composing spheres. However, there are occasions where a programmer would consider an element from one sphere to *correspond* to some otherwise seemingly-different elements from the other spheres for the purpose of making a query. Consider the following situations:

1. A class C in the static source code should correspond to the occurrence of the class C' in the dynamic trace, and vice versa. Similarly a method m defined on C in the static source should correspond to the method m' on C' in the dynamic trace.
2. An interface I in the static source has no direct correspondence to any element in a dynamic trace resulting from execution of the program, but it may be considered to have a correspondence to any classes C' in the dynamic trace that correspond to any class C in the source that implements I. Similarly a method m declared by interface I has no direct correspondence to any element of a dynamic trace, but may have an correspondence to a method m' of class C' where $C'.m'$ corresponds to the method $C.m$, where C implements I.
3. The body of a method as defined in a particular file revision as committed to a repository may have a correspondence to the representation for that method in the static source.

In each of these examples, the programmer is mentally converting from one particular representation to another, to treat one object from one representation as one or more equivalent objects in another representations.

We model this process of establishing correspondences using an adaptation of Wing and Ockerbloom's notion of a respectful type converter [33]. They define

> A converter $C : A \rightarrow B$ *respects type* T if the original object of type A and the converted object of type B have the same behavior when both objects are viewed as a type T object. (p. 579)

T represents a set of common properties or behaviours that must be preserved across the conversion. We relax and extend this definition in two ways. First, in their definition, T is considered to be a common ancestor of A and B. We remove this requirement: in our current sphere definition, the correspondences from A to B are often not reflected in the concrete representations. Second, we allow an instance from A to correspond to one or more instances of B: an element from one sphere should be treated as a set of items from a different sphere if a programmer would consider them to be the same.

These three examples illustrate also demonstrate that correspondences may differ in at least two ways:

the valency of the conversion: There may be no single best match for an element: an element in one sphere may correspond to multiple elements in another sphere. This is illustrated with the conversion of a Java interface to its implementing classes in the dynamic run-time.

the fidelity of the conversion: The fidelity describes the degree of information preservation or information loss in the conversion. We provide three levels: EXACT, EQUIVALENT, and APPROXIMATE.

1. EXACT indicates a conversion where there is a perfect one-to-one and onto correspondence. For example, a method defined on a class for a static Java tool will correspond exactly to the same method from a dynamic runtime tool, assuming the dynamic run-time was taken from the same source base.
2. EQUIVALENT indicates a very near match: there is not a perfect conversion, but the elements are felt to be equivalent from a programmer's perspective. For example, a Java interface will have an equivalent correspondence to its implementing classes in in the dynamic run-time, providing the trace was obtained from the same source code.
3. APPROXIMATE indicates an imperfect match, where there is a similarity, but some differences. For example, a method from a particular version of a file as committed to a repository will have only an approximate correspondence to the representation of the method in the source code being edited.

We discuss the use of fidelity in the next section.

Conversion is modelled as a function on the relations defined by a sphere. Imagine relations $R \subseteq L \times E \times E$ as a relational matrix

$M = E \times E$ with each matrix entry holding the applicable set of labels, where $E = E_1 \cup E_2$. Then there are three possibilities for how a converter $C : E_1 \rightarrow E_2$ maps inputs to outputs; each of these conversion possibilities corresponds to row merge or duplication operations on the relational matrix.

$1 \rightarrow 1$: element $e \in E_1$ corresponds to a unique element $e' \in E_2$: the row corresponding to e is replaced by the union of the rows for e and e'.

$1 \rightarrow n$: element $e \in E_1$ corresponds to several elements $e'_i \in E_2$: the row corresponding to e is replaced by the union of the row for e with the rows for e'_i.

$1 \rightarrow 0$: element $e \in E_1$ corresponds to no elements in E_2: the row corresponding to e remains unchanged.

This construction process proceeds similarly for $C : E_2 \rightarrow E_1$. Note that conversion may only rarely be symmetric.

2.4 Fidelity

The concept of *fidelity* is used in Section 2.3 to describe the degree of information preservation or information loss resulting from a conversion. Even when a converted element does not have a perfect correspondence to its source, it can still be used as a source for inquiry.

Consider an example where a programmer is using Ferret over a union of the static Java and evolution spheres (Table 1). We have a Java type T, which is instantiated in methods m_1, m_2 and m_3. We examine the source for an earlier edition of T, an evolution element S, which we consider to have an APPROXIMATE correspondence to T. Now consider projecting the relation instantiators from S. Although the evolution sphere has no such relation, the conversion rules from Section 2.3 cause the projection to be the results of projecting from T, resulting in the methods m_1, m_2 and m_3. But because they were determined from T, which has an APPROXIMATE correspondence to S, then the resulting methods must have fidelity of APPROXIMATE at best. To have a different fidelity would mean that the results of a relation could be more certain than the input.

As it may be useful for a relation to involve some speculation, we cause relations to have an associated fidelity too. The general principle is that fidelity of the results of projecting a relation should be the less of the fidelity of the input to the relation and the fidelity of the relation itself. As a conceptual query may require projecting a sequence of relations, then the fidelity should be monotonically decreasing over the course of the query.

Thus the fidelity provides an assessment of the soundness of a query's results. Our use of fidelity serves a similar purpose to Balzar's pollution markers [2], where computation results that fail to meet specified constraints are marked as being polluted. The markers are propagated through any subsequent computations that use a polluted result, indicating that the computation is polluted as well. We should note that the fidelity only annotates the results of a query, and does not affect the query computation.

3. TOOL

We created Ferret as a proof-of-concept tool to verify that the model can express sufficiently rich conceptual queries, to demonstrate that the conceptual queries can be implemented efficiently, and to support experimentation with programmers to determine which conceptual queries are useful. Currently, Ferret implements 36 conceptual queries across four spheres. We describe the queries supported by Ferret and show how these queries utilize the model. We defer a description of the results of a small field study to examine the usefulness of the queries to the next section of our paper.

3.1 Implementation of the Model

Ferret is a faithful implementation of the model described in Section 2. At a high level, a conceptual query is issued in the context of one or more elements of a program, such as a method declaration, and the results of the query are computed using a set of relations that are resolved by name through a sphere.

3.1.1 Spheres

Spheres are responsible for binding relation implementations to relation names. Ferret currently implements four spheres corresponding to the four sources: static, dynamic, software evolution, and Eclipse PDE (see Table 1). The relations supported by the spheres are implemented using the APIs provided by the sources. The static Java sphere is implemented using the public Core and Search APIs provided by the Eclipse Java Development Tools (JDT). The dynamic Java run-time sphere is implemented using the profile querying facilities of the Eclipse Test and Performance Tools (TPTP). The software evolution sphere is implemented using facilities from Kenyon [5], a source code repository analysis framework supporting systems such as CVS [4], amongst others. The plug-in development sphere is implemented using the internal representation model from Eclipse Plug-in Development Environment (PDE).

3.1.2 Conceptual Queries

The 36 conceptual queries currently implemented in Ferret are listed in Table 2. The queries we have implemented were identified from the literature, blogs, or our own experiences developing Ferret; the motivations for queries is tracked in Table 2. Some of these queries have analogs to existing queries (marked with a dagger or double-dagger in Table 2) but are augmented in their ability to use other sources of program information. Other queries, such as the evolution queries for relating source changes to code, do not have direct analogs in any existing tool. Some conceptual queries are included to serve as a demonstration of cross-artifact queries. For example, the PDE-related queries link declarative information specified in Eclipse plug-in manifests to static source elements.

A conceptual query is specified in terms of relational operators over relation names. The relations named in the definition of a conceptual query are resolved over a composite sphere that serves as a fact database for the program. For example, the query *"what instantiates this type?"* can be expressed as:

implementors ∘ instantiators

The relation implementors implicitly takes the given input, likely to be a Java type, and returns all types that implement the input; the relation instantiators returns all instantiators of a given input. These relations are resolved using the composite sphere.

If a relation cannot be resolved, then the conceptual query fails. In this respect, conceptual queries implement a form of dynamic typing: if all the relations are resolved, then an answer can be computed for the query; if some relation cannot be resolved, then the query was not applicable for its inputs. This approach to implementing the model isolates conceptual queries from the sources of program information used to compute the results of the query.

3.1.3 Composing Spheres and Resolving Relations

Conceptual queries are executed in the context of a sphere, which is used to resolve the relations named in the conceptual query's definition. The composite sphere is an implicit parameter to an installation of Ferret, and is used to resolve the relations named in a conceptual query's definition. The sphere may be a composite sphere specified using the composition functions defined in Section 2.2. We defer an example to Section 3.2. The composition

Table 2: Conceptual queries currently defined for Ferret. '✓' indicates spheres providing relations used in the conceptual query. '*' indicates queries supporting plural inputs; '†' indicates queries that are not available in current tools and '‡' indicates queries that have partial support in current tools. The 'Usage' columns reflects the average usage of the query by the programmers observed for the study described in Section 4.

		Contributing Spheres				Usage	
	Category / Conceptual Queries	**Static**	**Dynamic**	**Evolution**	**Plug-ins**	**#**	**%**
inter-class							
1	†What methods return instances of this type? [16]	✓				1	0.4%
2	*What methods instantiate this type? [27]	✓	✓			8	2.9%
3	*What methods take an argument of this type? [27]	✓				2	0.7%
4	What fields are declared as being of this type?	✓				8	2.9%
5	*What calls this method? [27]	✓	✓			78	28.2%
6	*Where is the value of this field retrieved? [27]	✓				23	8.3%
7	*Where is the value of this field set? [27]	✓				5	1.8%
8	*What tests instanceof against this type? [17]	✓				2	0.7%
9	*Where are casts to this type? [17]	✓				1	0.4%
10	*What throws exceptions of this type? [27]	✓				0	0%
11	*What catches exceptions of this type?	✓				0	0%
12	*What references this type by name? [27]	✓				13	4.7%
13	*Which of the classes in this package were *actually* used?		✓			7	2.5%
14	*Which methods defined by this class were *actually* used?		✓			5	1.8%
intra-class							
15	‡*What fields does this type or method access? [27]	✓				16	5.8%
16	‡*What methods does this type or method call? [27]	✓	✓			23	8.3%
17	†*What types does this {type, method, or extension} reference?	✓			✓	18	6.5%
inheritance							
18	*What interfaces does this type implement? [27]	✓				0	0%
19	What are this class' subclasses? [27]	✓				6	2.2%
20	*What classes implement this interface? [27]	✓				8	2.9%
21	*What interfaces extend this interface? [27]	✓				5	1.8%
22	†What are this type's siblings? [27]	✓				2	0.7%
declarations							
23	What are all the fields are declared by this type? [27]	✓				9	3.2%
24	What class methods implement this interface method? [27]	✓				6	2.2%
25	*What interface methods specify this class method? [27]	✓				2	0.7%
26	What class methods does this method override?	✓				5	1.8%
27	What class methods override this method?	✓				7	2.5%
28	‡What extension points or extensions reference this {type, file, folder}?				✓	6	2.2%
29	†What types is this type adaptable to?	✓			✓	3	1.1%
30	†What types could this type have be adapted from?	✓			✓	2	0.7%
31	What extensions are there of this extension point?				✓	4	1.4%
32	What plug-ins depend on this plug-in?				✓	2	0.7%
evolution							
33	†What transactions changed this element? [21]			✓		N/A	N/A
34	†Who has changed this element, and when? [21, 31]			✓		N/A	N/A
35	†What elements were changed in this transaction? [21]			✓		N/A	N/A
36	†What files were changed in this transaction?			✓		N/A	N/A

functions determine the order in which spheres are consulted for possible candidate relation implementations. The functions union, and transformation cause all spheres to be resolved and to perform some function on the combined results. The function replacement selects an individual sphere.

Relations are always resolved in the context of some input from which the relation is to be projected. This input comes from a selection from some information displayed by a tool that is represented by one of the spheres. A relation implementation is considered resolved only after verifying that it supports the input. For example, the static Java sphere's implementation of implementors requires its input to be a JDT type instance (an IType). This support generally entails that the inputs are or can be converted to an instance of some particular type, and that possible conversions of the inputs meet a minimum required fidelity. If the input cannot be converted, then the relation implementation is rejected and resolution contin-

ues. For example, both the static Java sphere and the dynamic Java sphere provide implementations of a calls relation. Should these sphere be composed using the union composition function, then both spheres will be consulted for a calls relation. Should they be composed using replacement, each sphere is consulted in turn until one successfully resolves the relation. The former case is useful if the conceptual query is intended to include calls implicit in the code, such as calls via the Java reflection mechanism. The latter case is useful if only the calls actually made during execution are of interest.

3.1.4 *Supporting Correspondences*

Key to the relation resolving process is the conversion of inputs. For example, the implementation of the implementors relation in the static Java sphere using JDT requires its input to be a JDT representation of a Java type, an IType. The resolving process will attempt to find any ITypes that corresponds to its input.

Correspondences are implemented in Ferret using a mediating broker architecture [33]. Each broker specifies the source and destination types supported, the maximum possible fidelity of the correspondence, and the maximum possible valency of its correspondence. To reduce the number of brokers necessary, the brokers are assembled into a graph. A correspondence may result in a path with several intermediate correspondences. As there may be multiple candidate paths for converting a source type to some desired target type, we select the best path by (i) considering only paths where the intermediate correspondence steps are univalent so as to avoid any possible ballooning in the results, and (ii) weighting each path by its total fidelity. The fidelity of the resulting correspondence is the minimum fidelity of the correspondence on the path. A correspondence once made is cached based on the source instance.

The set of currently configured correspondences is shown in Figure 1, grouped by related purposes. The types in this graph are grouped into enclosures to indicate types with a related purpose. In our implementation of Ferret, we have created two classes of correspondences. The first class are handle correspondences of elements between different spheres. The JDT ↔ TPTP , and Kenyon ↔ JDT conversions serve as examples of this kind. The second class of correspondences exist to convert types that are conceptually part of one of the provided spheres into a canonical form. For example, the java.reflect and JDT/Debug enclosures represent live Java instances from a runtime, such as in a stack frame of the debugger. These instances have a likely correspondence to types defined in the static Java sphere. These canonical correspondences allow Ferret to be invoked from different views in the UI.

3.2 Example of Relation Resolution

To illustrate the implementation of the model, we trace through the resolution of the returns-type relation when provided a particular JDT interface (type IType). This relation matches a type to the methods declared as returning an object of that type. We use the composite sphere corresponding to the following definition; we often use this definition when trying to understand some particular functionality of a software system.

```
replacement[
  dynamic-java-sphere(tptp-trace-file),
  transformation[
    static-java-sphere,
    dynamic-java-sphere(tptp-trace-file),
    "was-invoked"]]
```

This particular sphere composition uses the replacement function at the top level. Thus function causes returns-type to be resolved against its sub-spheres in succession until one is successful. Since a dynamic Java sphere does not provide an returns-type, the replacement causes resolution to proceed to the next sphere.

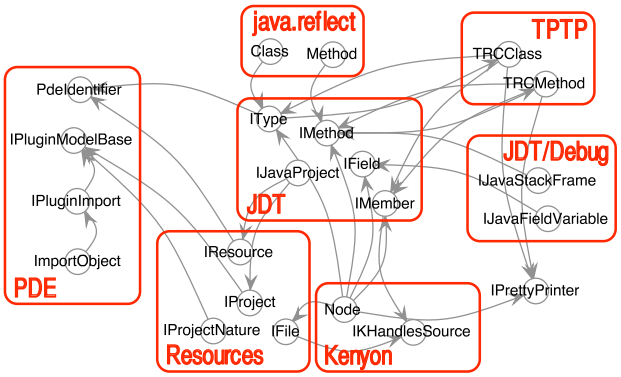

Figure 1: The conversion broker network in Ferret. Arrows indicate possible conversions; rounded enclosures indicate groupings of related elements.

The next sphere is a composite sphere obtained using the transformation function. The transformation function first resolves returns-type against the static Java sphere. This resolution is successful as the sphere provides an implementation for returns-type, and the implementation expects a JDT type instance (type IType) as input. The transformation function then joins the results from projecting returns-type to another relation, was-invoked, which is projected against the dynamic Java sphere.

The dynamic Java sphere provides an implementation of was-invoked. Was-invoked acts as a filter, filtering methods that were never called. Its implementation is as follows:

- if its input is a TPTP method, or can be converted to a TPTP method:
  - if the method was actually recorded as being invoked in the provided dynamic execution trace, then return the input,
  - else the input was not sent, so return nothing;
- else the input is not a method, so return the input.

The implementation of was-invoked must attempt to convert its input to native TPTP methods (of type TRCMethod). The correspondence network, shown in Figure 1, supports converting JDT methods (type IMethod) to the corresponding native TPTP method (TRCMethod). Methods defined on interfaces, which have no executable components, are converted to the TPTP methods corresponding to the class methods that implement the interface methods. Thus each of the overriding methods are checked for was-invoked, which effectively filters the overriding methods by those actually sent. These methods are the result of the transformed returns-type relation and, since the returns-type was successfully resolved, the replacement returns the type-returning methods that were actually sent as the overall results of the projection of returns-type.

Thus with this composite sphere, projecting returns-type from a particular interface *t* results in the methods declared to return type *t* that were actually invoked as recorded in the given dynamic trace file. Performing this query without Ferret would require correlating a list of methods from a JDT view using linear searches through a TPTP view.

3.3 User Interface

Ferret is integrated into the Eclipse IDE as a single view (Figure 2). Upon being invoked for a particular element, Ferret com-

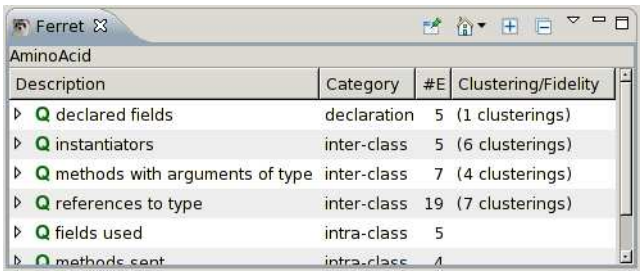

Figure 2: The typical Ferret view in Eclipse.

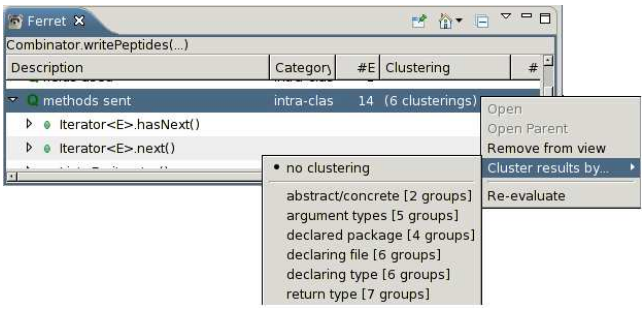

Figure 3: Ferret supports clustering results of conceptual queries.

putes and displays the results of all known conceptual queries in a structured tree view. Queries that are not supported for the particular element from which Ferret is triggered are not displayed; by default, queries with no results are also not displayed. For each query, Ferret displays the number of results for the query, and the number of possible clusterings of the results (discussed below). When displaying the results of the query, Ferret also informs the programmer of the fidelity of the results.

Ferret can be invoked either explicitly or implicitly. Ferret is invoked explicitly either through a keyboard shortcut or from a program element's context menu. Ferret also treats the programmer's selection of program elements in the IDE (except for selections within the Ferret view) as an implicit invocation of Ferret's queries; this style of implicit invocation can be disabled.

As a conceptual query may have many results, Ferret provides means for imposing alternative organizations of results by clustering according to the attributes of the solution elements. Figure 3, for example, shows different possible clusterings for a set of Java methods, which includes the declaring package, or the types of the arguments. A particular conceptual query may provide additional clusterings based on properties calculated during the query. For example, the *"what methods instantiate this type?"* query, which will includes the results of queries on implementors of an interface, adds an additional clustering for the type instantiated. A programmer may manually remove any result from the view should it be deemed to be uninteresting.

Queries can also be *cascaded* from a query result, where the result is used as the source for an invocation and whose results are displayed in-place in the tree. This ability is similar to that offered in JQuery [17].

Some conceptual queries can operate with multiple inputs and correspond to the questions of the form *"what common* [feature] *is supported by these* [elements]*?"*; these queries are indicated in Table 2 by an asterisk.

Table 3: Ferret benchmark timings (seconds, average of five runs) for the two sphere configurations described in Section 3.2.

Row	Element	C1	C2
1	PropPanelCallEvent (class)	0.3	0.3
2	PropPanelEvent (abstract class)	0.6	0.5
3	TargetListener (interface: 232 implementers)	14.3	15.5
4	PropPanelEvent.paramScroll (field)	0.3	0.4
5	TabProps.findPanelFor(Class) (method)	0.5	0.2
6	TargetListener.targetSet(Object) (method)	1.1	1.9

3.4 Benchmarks

Ferret's querying performance is commensurate with the degree of use of the element used to drive the queries: elements that are widely used will require more time to compute the relations. Table 3 presents average timings to perform a Ferret invocation for a selection of types and methods from ARGOUML.[1] The table presents timings for two different sphere configurations. The first configuration (labelled 'C1') used only the static Java sphere and corresponds most closely to invoking a number of queries using a normal static Java tool. This configuration allows comparing the time taken with a static Java tool and, more importantly, calibrating against what a human might accept from a tool. The second configuration (labelled 'C2') used the sphere configuration described in Section 3.2, which combined static Java information and dynamic program information from a dynamic execution trace. This dynamic execution trace had filtered all classes from packages matching java*, com.sun*, com.ibm*, sun*, and org* except for those classes matching org.argouml*. The resulting trace recorded 417 291 events over 2 604 methods defined in 760 classes. Rows 1–3 measure the performance of queries for a Java class or interface, and rows 4–6 measure the performance of queries for Java methods.

The addition of the dynamic Java sphere has a mixed effect, with some types of elements causing longer query times and others leading to shorter query times. The increased times are likely due to occurrences of JDT interface methods: the conversion of JDT methods to TPTP methods required for the was-invoked relation (Section 3.2) requires first expanding the interface method to its implementors, which are then converted to the native TPTP methods, which slows the operation. The decreased times are likely benefiting from the faster projection times from the TPTP queries, which are done from an in-memory representation.

Although these times are not instantaneous, they are generally faster than the times required by a human to visit and combine the results of queries between multiple tools. These times can also be improved by using better-suited query back-ends. For instance, Ferret currently uses the JDT search facilities for static Java searches, which defers most of the processing required for a search to search time. Other search back-ends might optimize search time, which would be better for Ferret since it issues multiple queries at once.

4. FERRET IN USE

Having implemented 36 queries from a variety of sources, we now turn to the question of whether these conceptual queries are useful to programmers.

[1]Timings were performed for ARGOUML 0.13.6 (1605 Java classes), running on a Lenovo ThinkPad T60 2GHz Core Duo with 1GB of RAM using the Sun JDK 1.5.0_11-b03, Ferret 0.3.0.200708261920, Eclipse SDK R3.3.0 with 36 open projects.

In an earlier controlled study, we sought to compare programmer performance using several software exploration tools, including an earlier version of Ferret. In this earlier laboratory study, Ferret was equipped only with the static Java sphere. Thus this study did not support investigation of multiple spheres nor how and when programmers might use the conceptual queries supported by the sphere model and the Ferret tool. We wished to investigate how programmers used Ferret during their day to day work. More specifically, we sought to discover:

- Which conceptual queries implemented in Ferret are useful to programmers?
- Is the composition of static and dynamic program information, which have some overlap in their concrete queries, useful?
- Are there features of Ferret that programmers find particularly useful?

To answer these questions, we undertook a diary study. We recruited four professional Java programmers from a large company to use Ferret during their day-to-day programming activities. The programmers were working on their own familiar codebases. Each programmer was equipped with a version of Ferret instrumented to record the queries used. This version of Ferret included the JDT, PDE, and TPTP spheres; we were unable to use the evolution-related queries as our repository library software was not compatible with their repository software. We provided a 20-minute tutorial on configuring and using Ferret. As we had found in previous studies that programmers often revert to using the tools with which they are familiar, we included some monitoring software to remind the programmers of Ferret should they undertake long periods of searching or exploration without using Ferret. The programmers were asked to maintain a journal, reflecting on their usage, as well as recording any events when Ferret failed to do something or surprised them in some fashion. We interviewed the programmers at the end of their first day, asking about their usage of Ferret and discussing the events recorded in their journal. We retrieved the instrumentation logs and journals at the conclusion of the study.

The diary study was originally scheduled to last two days. We anticipated that programmers would spend the first day learning to use the tool and determining how to incorporate it into their exploration strategies. We anticipated the programmers would be able to use the tool productively on the second day. We staggered the programmers so as to allow one-on-one attention for the first day. We did not initially schedule more time for the diary study because we did not believe we would be able to recruit programmers for use of an experimental tool in their daily work for a longer period of time. Some programmers volunteered to extend the study beyond the two days (P1–P3), and some programmers reported continuing to use the tool beyond the end of the study (P2,P3).

4.1 Results

Three of the programmers used an integration of two spheres (the static Java and PDE spheres), and one programmer used the integration of three different spheres (the static Java, PDE, and dynamic Java spheres). The programmers provided limited journal entries, generally recording questions or small usability or performance issues. The monitor logs provided a record of Eclipse and Ferret use, and the programmers were able to provide feedback during the end-of-day interview.

The programmers reported finding Ferret useful. The programmers used almost all of the conceptual queries provided by Ferret that were available to them. Table 2 reports the average use of each query. Not surprisingly, the most heavily used queries correspond to common queries in Eclipse. For instance *"what calls this method?"* accounted for 28% of the usage. For 3 of the 4 participants, this query would have returned the same results as Eclipse because they were not able to make use of the TPTP integration. However, the data in Table 2 also shows the programmers used some queries more than once that are not supported by JDT. For example, the intra-class queries for *"what {fields, methods, types} does this element access?"* were used surprisingly often.

P4 was the only programmer to use the TPTP integration to incorporate dynamic runtime information.[2] P4 created a trace to examine some anomalous run-time behaviour exhibited by a particular set of unit tests. He reported that having the ability to see only what was *actually* called was useful because it eliminated spurious calls made from other tests.

Table 4 presents the usage statistics over the course of the period of the study. The '# Days Used' and 'Eclipse Usage' columns indicate the total time that each programmer used Eclipse: the days column records workdays, and the usage column as assessed by accumulating the time between events in the monitoring logs, less any gaps of four minutes or more. This table lists the number of explicit invocations of Ferret ('Invoked'); the number of cascading invocations ('Cascaded'), where Ferret was triggered in-place from a previous result; and the number of query expansions ('Expanded'), where a programmer examined the results of one of the queries. Also included are the number of times where a programmer caused the results of a query to be clustered by some attribute ('Clusterings'), and the number of Eclipse-based searches made.

The fact that expansions were used more than invocations and cascades indicates that the programmers generally looked at the results of several queries per invocation.

P3 made particularly heavy use of the clustering, primarily to cluster by package type. He noted that some of his queries had a large number of results, and he was only interested in those from a particular project which were all in a single package.

P1 and P3 both used a small number of Eclipse-based searches during their sessions. The general lack of use of Eclipse-based searches indicates that Ferret seemed to have provided a suitable mix of conceptual queries. P1 explained that he knew precisely what he was looking for, and although Ferret could have returned the information, he did not want to wait for Ferret to find the results. P3's queries were for occurrences of text strings, which does not match Ferret's focus on conceptual queries.

4.2 Validity

We chose to focus on realistic work by real programmers. The use of a small number of programmers from the same company, and over a small period of time, may limit the generalizability of the results. A more extensive discussion of threats to validity are available elsewhere [7].

5. DISCUSSION

5.1 Extending Ferret

The sphere model is designed to be extensible to handle the definition of other conceptual queries. The model can easily accommodate, for instance, the definition of new composition functions.

The Ferret tool is designed to be extensible: spheres, conceptual queries, sphere composition functions, type conversions, and clusterings are all configured through a set of extension points in Eclipse. These elements are assumed to have been configured or implemented by some provider, such as a tools-support group.

[2]P1 and P2 were not able to use the TPTP integration due to problems with their installation which prevented its use; P3 reported not being in a situation where he felt he needed it.

Table 4: Ferret usage over the course of the diary study.

Id	# Days Used	Eclipse Usage (hh:mm)	Ferret Queries				# Eclipse Searches
			Invoked	Cascades	Expanded	Clusterings	
P1	4	20:24	47	43	107	7	9
P2	3	14:20	20	10	50	2	0
P3	3	14:53	31	20	66	38	2
P4	2	9:04	10	9	54	2	0

5.2 Conversions Are Not Symmetric

Some conversions are not as straightforward as might be thought. This is illustrated by the JDT to Kenyon conversions. We initially supposed that the individual JDT element types would correspond to all the editions of that element in the repository (a Kenyon Node). We realized this conversion implied that a programmer would regard *all* editions as being the same — which is not true.

We instead convert the JDT elements to an intermediate *handle* representation. The Kenyon repository history is then queried for the editions of this handle. The individual editions of a handle element can be converted to JDT element.

5.3 Presentation Issues

Much of the feedback from the programmers from our diary study involved improving the presentation of results in Ferret's view. Presentation of data is a subject fraught with cultural and personal preferences. We have deliberately chosen to avoid presentational issues as much as possible by using the standard presentation mechanisms in Eclipse's JFace, such as simple tree views. This is an area for future work.

6. RELATED WORK

In considering previous efforts related to the sphere model and Ferret, we restrict ourselves to efforts that seek to integrate different sources of program information.

6.1 Integrating Dynamic and Static Views

Richner and Ducasse [23] describe an approach to integrate static and dynamic information using logic programming. They use logic queries to drive program visualizations. As their implementation was centred around Smalltalk, which has no reification of interfaces, they did not encounter the valency issues in conversions.

Shimba [29] integrates static with dynamic information to produce UML-style sequence diagrams and state diagrams. Upon the selection of some subset of the classes or methods, Shimba instruments the program to produce either a sequence diagram representing interactions between selected objects, or a state diagram representing the control flow behaviour of the selected object or method. Shimba's focus is on the production of particular views rather than on a model to integrate different kinds of program information.

6.2 Tool Integration

A number of approaches have focussed on integrating tools, such as unifying different user interfaces and data requirements. Several different levels of integration have been recognized, from presentational aspects, to sharing of data formats, to co-operation and notification between tools [32]. These approaches aim to support (i) the automated production of new software artifacts in response to changes to other artifacts [35, 13, 22], (ii) reconciling inconsistencies between artifacts from different stages of the software development life-cycle [6, 3], or (iii) facilitating the sharing of the external artifacts produced by different tools [28, 18]). We take a different approach by focusing on the use of integration of different sources of program information to provide better support to programmers during exploration of the source code.

Other groups have focussed on defining meta-models to express correspondences between software artifacts so as to support reasoning [19, 1, 14]. These approaches generally support only coarse-grained artifacts, and assume a one-to-one mapping.

Garlan's *views* [12] is perhaps closest to our work. Tools within a development environment, for instance, an editor and a compiler, often require different perspectives on similar data to perform a desired action. To avoid duplication of data in such an environment, Garlan proposed the use of views that together formed a logical database over which all tools in the environment could operate. One can ask whether Garlan's approach could be used to implement conceptual queries by considering each query as a tool. We believe this approach is not feasible because Garlan's model does not easily support the transformation and combination of the result sets of queries over the data being represented. Our intent is not to enable tools to work together to achieve their intended goal, but rather to gather from a programmer's perspective the results that such tools produce.

A final set of related work involves type conversions. PTOLEMY [36] and BALBOA [9] provide systems for lossless type conversions when selecting components for use in component-based modelling and design. Wing and Ockerbloom [33] describe *respectful* type conversions, where a conversion $A \rightarrow B$ is characterized by the properties preserved as expressed by another type T. These conversions assume a one-to-one mapping.

6.3 Cross-artifact Search Tools

Both G^{SEE} [11] and SEXTANT [24] support integrating other sources of information other than the static source code. SEXTANT, for instance, supports integration of the relations embedded in Enterprise Java Beans deployment descriptors. However both of these approaches assume that there is a direct correspondence between elements shared between the different artifact domains. In addition, neither tool considers how to reconcile the choice of different implementations of the same query. The sphere model addresses both of these issues.

6.4 Query Languages

There have been a number of languages proposed for querying software. Most have used a standard database language such as SQL or *Datalog* (e.g., CodeQuest [10]), a Prolog-like implementation (e.g., JQuery [17]), or a custom language (e.g., SCA [20]). None of these approaches provide support for the expression of correspondences between elements or for the automatic propagation of queries across corresponding elements. These languages also do not consider how to reconcile the choice of different implementations of the same query. Ferret addresses these issues, but we have not yet determined the range of conceptual queries that the sphere model can support.

7. CONCLUSIONS AND FUTURE WORK

At one time, programmers had to extract information manually from source code to answer questions about a program. As programs grew more complex, tools were developed to help answer some of a programmer's question(s). These tools continue to become more and more sophisticated and able to efficiently answer more and more of a programmer's questions. Despite this progress, many of a programmer's questions about a program still require significant manual work to determine which tools can provide a piece of the answer and how to integrate the results of multiple tools to get closer to a complete answer.

In this paper, we introduce a model that alleviates the burdens programmers face for a significant number of questions, which we call conceptual queries. This model enables different sources of information to be composed from a programmer's perspective. We have shown how this model can implement 36 conceptual queries, many of which are derived from the literature, and some for which there exists no current direct tool support. We have shown in a small field study that practicing software programmers found a number of these queries useful.

This work illustrates that it is possible to achieve deeper integration in an integrated development environment that goes beyond tool to tool interactions, such as invoking one tool from another, to integrating the results of the tools in a comprehensible manner for programmers.

Acknowledgments

Thanks to Chris Dutchyn and to Thomas Fritz for helpful conversations and comments. This research was funded in part by NSERC and in part by IBM.

8. REFERENCES

[1] G. Antoniol, M. Di Penta, H. Gall, and M. Pinzger. Towards the integration of versioning systems, bug reports and source code meta-models. *Electr. Notes in Theor. Comp. Sci.*, 127(3):87–99, Apr. 2005.

[2] R. Balzer. Tolerating inconsistency. In *Proc. ICSE*, pages 158–165, 1991.

[3] S. M. Becker, T. Haase, and B. Westfechtel. Model-based *a-posteriori* integration of engineering tools for incremental development processes. *Softw. Syst. Model*, 4(2):123–140, May 2005.

[4] B. Berliner. CVS II: Parallelizing software development. In *Proc. USENIX Winter 1990 Technical Conference*, pages 341–352, 1990.

[5] J. Bevan, E. J. Whitehead Jr., S. Kim, and M. Godfrey. Facilitating software evolution research with Kenyon. In *Proc. ESEC/FSE*, pages 177–186, 2005.

[6] S. Burmester, H. Giese, J. Niere, M. Tichy, J. P. Wadsack, R. Wagner, L. Wendehals, and A. Zündorf. Tool integration at the meta-model level: the Fujaba approach. *Int. J. Software Tools and Technology Transfer*, 6(3):203–218, Aug. 2004.

[7] B. de Alwis. *Supporting Conceptual Queries Over Integrated Sources of Program Information*. PhD thesis, Dept. of Comp. Sci., University of British Columbia, Apr. 2008.

[8] B. de Alwis and G. C. Murphy. Using visual momentum to explain disorientation in the Eclipse IDE (short paper). In *Proc. IEEE Symp. on Visual Lang. and Human-Centric Comput. (VLHCC)*, pages 51–54, 2006.

[9] F. J. Doucet, S. K. Shukla, and R. K. Gupta. Typing abstractions and management in a component framework. In *Proc. Asia South-Pacific Design Autom. Conf. (ASPDAC)*, pages 115–122, 2003.

[10] M. V. Elnar Hajiyev and O. de Moor. CodeQuest: Scalable source code queries with datalog. In *Proc. European Conf. Object-Oriented Programming (ECOOP)*, pages 2–27, 2006.

[11] J.-M. Favre. G^{SEE}: A generic software exploration environment. In *Proc. Int. Worksh. on Prog. Compr. (IWPC)*, pages 233–244, 2001.

[12] D. Garlan. Views for tools in integrated environments. In *Proc. Int. Worksh. Advanced Programming Environments*, volume 244 of *LNCS*, pages 314–343. Springer-Verlag, 1986.

[13] D. Garlan, G. E. Kaiser, and D. Notkin. Using tool abstraction to compose systems. *IEEE Comput.*, 25(6):30–38, June 1992.

[14] T. Gîrba and S. Ducasse. Modeling history to analyze software evolution. *J. of Softw. Maint. and Evol.*, 18(3):207–236, Mar. 2006.

[15] S. L. Graham, P. B. Kessler, and M. K. McKusick. Gprof: A call graph execution profiler. In *Proc. SIGPLAN Symp. on Compiler Construction*, pages 120–126, 1982.

[16] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proc. ICSE*, pages 117–125, 2005.

[17] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *Proc. Conf. Aspect-Oriented Softw. Dev. (AOSD)*, pages 178–187, 2003.

[18] G. Karsai, A. Lang, and S. Neema. Design patterns for open tool integration. *Softw. Syst. Model*, 4(2):157–170, May 2005.

[19] P. Mi and W. Scacchi. A meta-model for formulating knowledge-based models of software development. *Decision Support Systems*, 17(4):313–330, 1996.

[20] S. Paul and A. Prakash. A query algebra for program databases. *IEEE TSE*, 22(3):202–217, Mar. 1996.

[21] C. M. Pilato. Are detailed log messages really necessary? Personal blog, May 2007.

[22] S. P. Reiss. Simplifying data integration: The design of the desert software development environment. In *Proc. ICSE*, pages 398–407, 1996.

[23] T. Richner and S. Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In *Proc. ICSM*, pages 13–, 1999.

[24] T. Schäfer, M. Eichberg, M. Haupt, and M. Mezini. The SEXTANT software exploration tool. *IEEE TSE*, 32(9):753–768, 2006.

[25] G. Schmidt and T. Ströhlein. *Relations and Graphs: Discrete Mathematics for Computer Scientists*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1993.

[26] J. Sillito. *Asking and Answering Questions During a Programming Change Task*. PhD thesis, Dept. of Comp. Sci., University of British Columbia, Dec. 2006.

[27] J. Sillito, G. C. Murphy, and K. De Volder. Questions programmers ask during software evolution tasks. In *Proc. FSE*, pages 23–34, 2006.

[28] H. Stoeckle, J. Grundy, and J. Hosking. A framework for visual notation exchange. *J. Visual Lang. Comput.*, 16(3):187–212, June 2005.

[29] T. Systä. Understanding the behavior of java programs. In *Proc. WCRE*, pages 214–223, 2000.

[30] W. Teitelman and L. Masinter. The Interlisp programming environment. *IEEE Comput.*, 14(4):25–33, Apr. 1981.

[31] L. Voinea, J. Lukkiena, and A. Telea. Visual assessment of software evolution. *Sci. Comput. Programming*, 65(3):222–248, 2007.

[32] A. I. Wasserman. Tool integration in software engineering environments. In *Proc. Intl. Workshop on Software engineering environments*, volume 467 of *LNCS*, pages 137–149, 1990.

[33] J. M. Wing and J. Ockerbloom. Respectful type converters. *IEEE TSE*, 26(7), July 2000.

[34] D. D. Woods, E. S. Patterson, and E. M. Roth. Can we ever escape from data overload? A cognitive systems diagnosis. *Cogn., Tech. & and Work*, 4(1):22–36, 2002.

[35] R. Wuyts and S. Ducasse. Unanticipated integration of development tools using the classification model. *Comput. Lang., Syst. & Struct.*, 30(1–2):63–77, 2004.

[36] Y. Xiong and E. A. Lee. An extensible type system for component-based design. In *Proc. Int. Conf. on Tools and Alg. for Constr. and Anal. of Syst. (TACAS)*, volume 1785 of *LNCS*, pages 20–37, 2000.