Towards a Simplified Implementation of Object-Oriented Design Metrics

Cristina Marinescu Radu Marinescu
LOOSE Research Group
"Politehnica" University of Timişoara, Romania
{cristina, radum}@cs.utt.ro

Tudor Gîrba
Software Composition Group
University of Berne, Switzerland
girba@iam.unibe.ch

Abstract

In order to compute metrics automatically, these must be implemented as software programs. As metrics become increasingly complex, implementing them using imperative and interrogative programming is oftentimes cumbersome. Consequently, their understanding, testing and reuse are severely hampered. In this paper we identify a set of key mechanisms that are involved in the implementation of design metrics and, more general, of design-related structural analyses: navigation, selection, set arithmetic, filtering and property aggregation. We show that neither of the aforementioned approaches offers a simple support for all these mechanisms and, as a result, an undesirable overhead of complexity is added to the implementation of metrics. The paper introduces SAIL*, a language designed to offer a proper support to a simplified writing of design metrics and similar design-related analyses, with a especial emphasis on object-oriented design. In order to validate the expressiveness of* SAIL *the paper provides a comprehensive comparison with the other two approaches.*

1 Introduction

In the last decade the number and complexity of design-related structural analyses (*e.g.*, design metrics[16], design heuristics[4], quality models[1]) for quality assessment has increased significantly.

When the source code is as small as a few or hundreds of lines of code, these analyses can be applied manually; but as the source code becomes larger, the various analyses that asses the quality of a piece of software must be automated. Consequently, the automation of such analyses requires that they are implemented; thus, analyses become themselves *programs*. Additionally, as the area of quality assessment evolves, analyses become more and more complicated and, consequently, their implementations become more and more obfuscated and thus harder to understand and maintain.

Through the last years we were much involved in the definition and implementation of various metrics (or metrics-related analyses)[15, 17, 18] regarding the quality assessment of object oriented design. This experience has shown us that apart from their intrinsic complexity, there is an additional complexity due to the expressiveness limitations of the programming language used for implementation.

Metrics are defined in terms of an abstract view of the system (*i.e.*, a model). Based on how models are represented we identify two major approaches for their implementations: the *structure based* and the *repository based* approach (Section 2). Additionally, we identify a set of five *key mechanisms* that are the building blocks for the implementation of any structural analysis: *navigation*, *filtering*, *selection*, *set arithmetic* and *property aggregation*. We show that none of the two aforementioned implementation approaches offers a simple support for all key mechanisms and we claim that this adds an undesirable overhead of complexity to the implementation of analyses.

The main goal of this paper is to show how the aforementioned mechanisms could be implemented efficiently in a programming language, so that they support a simplified writing of metrics and other quality assessment analyses. Thus, as a proof of concept, we defined and successfully used a simple interpreted language, called SAIL. In this paper we use SAIL to illustrate our approach towards a simplified implementation of metrics.

The paper is structured as follows: in Section 2 we describe the two aforementioned approaches with a special focus on the model representation. The first part of Section 3 identifies the mechanisms of a structural analysis and discusses how the current approaches support them. In Section 4 we introduce the SAIL language and illustrate how it enhances the support for the five key mechanisms. Next (Section 5) we present a validation of SAIL by comparing the size and complexity of the implementation of a suite of over 40 object-oriented design metrics. The paper concludes with a discussion on related work (Section 6) and some final remarks and hints towards future work (Section 7).

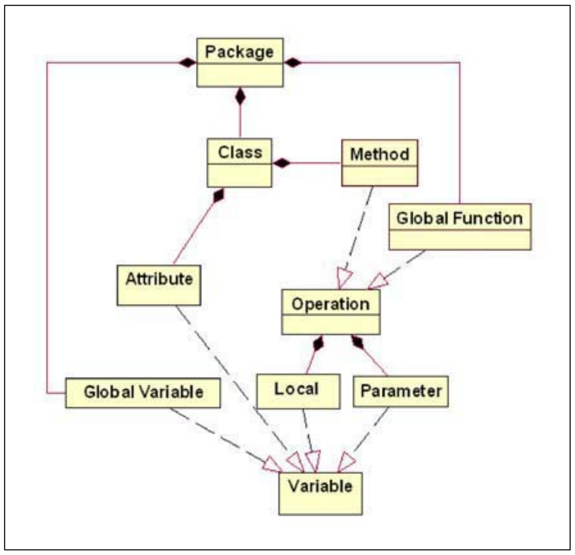


Figure 1. Structure based meta-model

2 Representation of System Models

To compute a metric on a given software system, we need to extract a *model* of the system. This model is extracted and stored based on a *meta-model* that specifies the relevant *entities* (*e.g.*, classes, methods etc.) and their relevant *properties* and *relations* (*e.g.*, inheritance, method calls)[17]. In this context a crucial question is: how is this meta-model represented? There are two "mainstream" family of approaches: *structure based* and *repository based*. Next, we describe them briefly and summarize both the positive and the negative aspects with respect to an example.

2.1 Structure Based Approach

In this approach the meta-model is represented as an interconnected set of data structures, usually one for each type of design entity. The fields are either elementary properties of that design entity or links to other related data structures. For example, a structure that models the Variable design entity is expected to have a field of type Class that establishes its connection to the class that represents the type of a variable. Commonly, in the *structure based approach* the meta-model and the metrics are implemented in a language that support user-defined data structures (*i.e.*, a procedural or object-oriented language).

In[17] we applied this approach, by defining a meta-model called MEMORIA [1] used for the implementation of software metrics and other quality assurance analyses. MEMORIA is concretely implemented in JAVA and, thus, classes were used to implement the data structures. In Figure 1 we show a simplified depiction of this meta-model where only aggregation relations are represented (*e.g.*, a class *has* methods and attributes, a method *has* parameters and local variables). The use of an object-oriented language for implementation allows us to hide the fields that describe the entity and define operations by which an entity communicates with other entities – *e.g.*, every entity has a `getName()` method, the `Class` entity has a `getMethods()` operation that returns a collection with all the `Method` entities defined in the given class.

[1]MEMORIA is similar to the FAMIX[24] meta-model

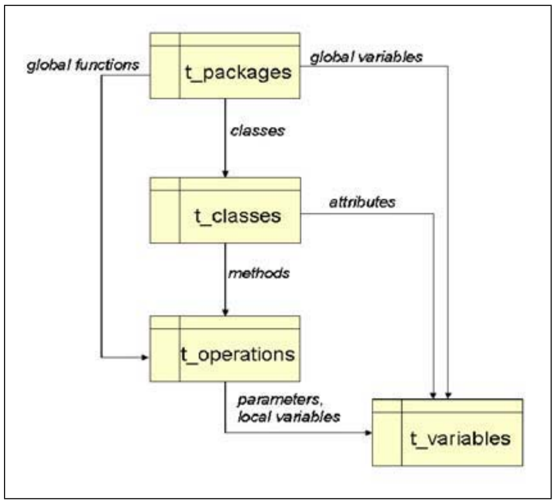


Figure 2. Repository based meta-model

2.2 Repository Based Approach

In this approach the meta-model is represented as a knowledge source that can be queried. Oftentimes this appears physically as a relational database system in which usually one table is defined for each design entity. The fields of the table contain the relevant properties of the entity and its connections to other (related) entities. In a normalized database these are established by means of *foreign keys*. Thus, reusing the example from the previous section, a hypothetic Variable table would contain a field (*i.e.*, a table column) that would point to the Class table, more precisely to that entry that represents the type of the variable.

Figure 2 presents a repository-based implementation of the meta-model already depicted in Figure 1 [2]. Each element in Figure 2 represents a table. Each arrow in the figure indicates a foreign key that relates two tables. As we did in Section 2.1, despite the fact that there are more relations than those depicted in the figure, for the sake of simplicity we decided to visualize only the *has* relations, as they contribute decisively to the navigability of the model.

[2]An earlier version of this was concretely implemented in SQL and is presented in[14]

3 Key Mechanisms

Next we are going to identify the key mechanisms that are involved in the implementations of metrics and see how these are supported by the two approaches discussed above.

3.1 Types of Analyses and Key Mechanisms

In general, metrics (or metrics-related analyses) fall mainly in two big categories: *group building* and *property computing*. The former category is mainly used for understanding a system while the latter is used for the assessment of the system. The implementation of each of these requires a particular set of key mechanisms.

Group Building Analyses construct *collections* of meta-model entities that are associated by a particular rule, described by the analysis itself, with the analyzed entity. Building a group for a meta-model entity requires a set of four elementary mechanisms:

- **Navigation.** All, except trivial metrics, are based on multiple entities so it is necessary to be able to browse through the model, going from the analyzed entity to a related entity (*e.g.*, from a class to its base class) or to a group of related entities from the meta-model (*e.g.*, from a class to the group of its methods).
- **Selection.** Every meta-model entity is described by various fields but only several of these are of interest in the context of a particular analysis. Therefore, the *selection* mechanism enables the definition of a "view of interest" by choosing only a subset of an entity's fields.
- **Set Arithmetic.** Groups of entities are after all built by means of set arithmetic. The most used set operations in analyses are: the addition of an entity to a group (a set) and the reunion of two or more groups[3].
- **Filtering.** An essential mechanism for building a group with a particular property is applying some filtering conditions to an initial larger group. For example, getting the group of public methods of a class requires first a navigation to the group of methods and then it requires also a filtering operation that builds a new group that keeps only methods with the "public" access specifier.

Property Computing Analyses associate a new, non-elementary, *property* to an entity. Usually, computing a property is preceded by the construction of an appropriate group. Thus, we may say that in most of the cases *property computing* analysis imply a *group building* analysis. Therefore, all key mechanisms identified before are, on principle, needed for a *property computing* analysis.

[3]Sometimes the cartesian product between two sets is also needed. For example getting all the method pairs in a class is needed in the context of cohesion metrics (see Section 5)

Example. Computing the FAN-OUT metric[12] for a class is obviously a *property computing analysis*. Yet, it first requires the construction of the group of all the classes on which the analyzed class depends. This operation is in itself a *group building* analysis. Additionally, in order to compute a *property*, usually a numeric or boolean value computed from a group associated with the entity, we need a fifth mechanism:

- **Property Aggregation.** This mechanism allows to compute and associate a single value for a group, a value which is aggregated from the values of each part of the group. Probably the most simple property aggregation is to get the cardinality of a group (used in the computation of most metrics). Depending on the type of properties of the entities belonging to the group more such aggregations can be imagined (*e.g.*, sum or average for numerical properties, maximum length for strings, logical AND for booleans).

3.2 The Structure Based Approach

In this section we are going to present how the key mechanisms identified before are expressed in the *structure based* approach. For each mechanism, we are going to illustrate its expression by using an ongoing example. Every implementation is written in Java and uses heavily the *model* of the system. It is stored is a suite of collections (*e.g.*, `sysPackages`, `sysClasses`) whose elements contain information about the relevant design entities extracted from the source code. The structure of a kept entity is a class (*e.g.*, `Package`, `Class`, `Method`).

Filtering. Example 1. From the analyzed system, we want to obtain *the package named* `my.package`.

In order to obtain from a group the entities that satisfy one or more conditions it is mandatory to iterate through the elements of the group. This operation is done using a *cycling instruction*, usually the `while` (like in 3) or `for` instruction. It is also mandatory to apply the filtering condition to each element, using a branching instruction, usually the `if` statement, like in 5. In Java 1.5, instead of the classic cycling instructions, we can use `foreach`. In our examples we did not used it because not every *structure based* approach provides it and its use only eliminates the need of a `cast` instruction, like the one in 4.

```
1 Package myPack;
2 Iterator it = sysPackages.iterator();
```

```
3 while(it.hasNext()){
4  myPack=(Package)it.next();
5  if(myPack.getName().equals("my.package"))
6   break;}
```

Thus, a simple operation like the one presented in this example requires a significant overhead that reduces the readability of the code and, thus, hampers its maintenance. As we will see next, this complexity increases if we need to navigate deeper in the hierarchy of containment and it gets even more obfuscated if the filtering conditions are "dispersed" through all the navigated entities.

Navigation. Example 2. From the analyzed package, we want to obtain the group of *the defined methods (excluding global functions)*.

As it can be noticed in Figure 2.1, the *defined methods* do not directly belong to a package, they belong to classes. Due to this, in order to obtain the desired group of methods, we need to pass two levels in the hierarchy of containment.

```
1 Collection methods=new ArrayList();
2 core.Class crtClass;
3 it=myPack.getClasses().iterator();
4 while(it.hasNext()){
5  crtClass=(core.Class)it.next();
6  methods.addAll(crtClass.getMethods())};
```

The navigation to the level of classes in the hierarchy whose root is the result of the first example, as you see in 3, is not difficult because passing one level means the access of an aggregated entity (in this case the collection of classes). Moving to the level of methods implies the iteration from 4. The iteration is done, like in the case of the filtering mechanism, using a *cycling instruction*.

What if we needed to navigate to the level of parameters? It is obvious we would perform one more iteration inside the one from 4. As long as we navigate deeper, the obfuscation of the implementation increases.

Selection. Example 3. From the defined methods in the analyzed package, we want to obtain *only their names and signatures*.

The result of this analysis is a collection which contains less information about the *defined methods* in the package called `my.package` than the previous one. Each element from the result contains only the name and signature of a method and is an instance of the `DetailedMethod` class.

```
1 class DetailedMethod{
2  private String name, sign;}
```

For each *defined method* we create an element that has a reduced structure and it is added to the result. Storing information in this new element requires the calls of some accessor methods[4]. Thus, reducing the information defined by the *meta-model* requires a lot of operations which increas the size of the implementation.

Set Arithmetic. Example 4. From the analyzed package, we want to obtain *the defined operations (methods and global functions)*.

We have obtained in the second example all the *methods* from the analyzed package. The only thing that remains is to unify them with the *global functions* defined in the package. Fortunately, the *global functions* are obtained in 1 by passing only one level in the hierarchy of content.

```
1 Collection globals = myPack.getGlobalFunctions();
2 methods.addAll(globals);
```

Due to the library support for manipulation of collections provided by Java, the union operation between the collection of methods and the global functions (see line 2) can be expressed in a straightforward way. But in a *structure based* approach which does not provide the aforementioned support, writing such union would required more operations (*e.g.*, writing iterations, comparisons).

Property Aggregation. Example 5. From the analyzed package, we want to obtain *the number of its classes*.

```
1 int classNo=myPack.getClasses.size();
```

Like in the mechanism presented before, the library support for manipulation of collections allows a simple expression of this mechanism and its absence would increase the complexity of the implementation.

3.3 The Repository Based Approach

In this section we are going to present how the key mechanisms identified in Section 3.1 are expressed in the *repository based* approach. We illustrate for each mechanism its expression by using the same example like the one in Section 3.2. In the following implementations each design entity is stored in a table, the columns whose names end with `ID` are *primary keys, foreign keys* and the interrogation is done using the SQL language.

Filtering. If the group whose cardinality we want to reduce is stored in only one table, the implementation of the filtering mechanism is expressive due to the `where` clause, like below.

```
1 select * from t_packages
2          where f_name="my.package"
```

[4]We did not provide the implementations of the accessor methods defined in the `DetailedMethod` class

But when the group is obtained by interrogating more than one table the expression of the filtering mechanism becomes obfuscated because in the `where` clause we have, beside the conditions for filtering, other conditions that assure the interconnections between tables.

Navigation. The navigation between a design entity from a level to another entity or to a group of entities from a deeper level can be mainly done in two ways.

The first form of navigation we present is the one in which we use only one querying instruction which takes data from all the passed through design entities (in this case, from operations, classes and packages) and assures the interconnection between the elements in the `where` clause (see line 4 and 5). In the `where` clause is also implemented the filtering mechanism (we are interested in the methods which belong to the package called `my.package`). This mixture of purposes from 3 to 5 in the selection condition drastically reduces the understanding of this implementation.

```
1 select F.* from t_packages as P,
2        t_classes as C,t_operations as F,
3 where P.f_name="my.package"
4       and C.f_packageID=P.f_packageID
5       and F.f_classID=C.f_classID
```

The other way in which the navigation can be done is presented below [5]. We use a number of querying instructions which equals the number of levels we pass through (in this case 3) and, beside the querying instruction associated with the reached level, every other is embedded into the one regarding the next level. This way the filtering condition is separated from those referring to the interconnection but having a `select` inside another `select` does not make the implementation more understandable.

```
select * from t_operations as F inner join
 (select f_classID from t_classes as C
  inner join
   (select f_packageID from t_packages
    where f_package="my.package") as P
 on C.f_clasID=P.f_classID)
on F.f_classID=C.f_classID
```

Selection. This mechanism, usually, is not difficult to implement in a *repository based* approach. Establishing the fields of interest associated with an entity requires an enumeration of fields, like the one from 1.

```
1 select f_name, f_signature from t_operations
```

[5]This particular implementation is based on the MYSQL *inner join* mechanism. Most of the the SQL dialects don't support this.

Set Arithmetic. The union of two or more groups can be rapidly done using the `UNION` command. But this command always needs at least two `select` instructions which compute the groups that are going to be unified and we have seen that sometimes the `select` instruction might have an obfuscated expression.

```
select F.* from t_operations as F,
           t_classes as C,t_packages as P
where F.f_classID=C.f_classID
       and P.f_name="my.package"
       and C.f_packageID=P.f_packageID
UNION
select F.* from t_operations as F,
           t_packages as P
where F.f_classID=P.f_globalID
      and P.f_name="my.package"
```

The first operand within the previous `UNION` instruction is the same as the one created in the paragraph regarding the Navigation. But why did not we *reuse* it? *Modularity*, the main mechanism that provides *reusability*, despite the introduction of *user-defined functions*, is still weak in SQL:99[13], a superset of SQL:92. For example, in Microsoft SQL Server 2000 *user-defined functions* have some restrictions placed upon them[10]: not every SQL statement or operation (*e.g.*, statements that update, insert, or delete tables or views) is valid within a function. This weakness of the *modularity* harms the implementation and, obviously, the maintenance.

Property Aggregation. Usually, the *repository based* approach provides a lot of operators for aggregation that reduce the complexity of the implementations.

```
select COUNT(C.*) from t_classes as C,
                  t_packages as P
where P.f_name="my.package"
      and C.f_packageID=P.f_packageID
```

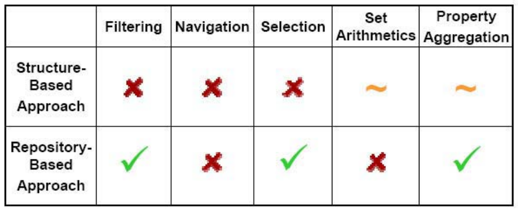
3.4 Comparative Discussion

In this section we are going to compare briefly the implementations of the mechanisms presented in Section 3.1 in a *structure based* approach and in a *repository based* approach. Figure 3 presents the characteristics of the implementations regarding simplicity.

In a *structure based* approach *Filtering* requires an iteration over the elements that are filtered and for each element is mandatory to apply the filtering condition while in a *repository based* approach this operation does not have such complexity.

None of the compared approaches provides support for a simple expression concerning *Navigation* between more than one level.

Selection requires a lot of operations in the *structure based* approach while in the *repository based* approach this



	Filtering	Navigation	Selection	Set Arithmetics	Property Aggregation
Structure-Based Approach	✗	✗	✗	~	~
Repository-Based Approach	✓	✗	✓	✗	✓

Figure 3. Key mechanisms'implementations

mechanism requires only the structure of the remained information.

The implementation of the *Set Arithmetic* mechanism is quite simple in the *structure based* approach but only if the used language provides library support for the manipulation of collections; if not, it requires a lot of iterations. In the other approach, the mechanism requires at least two *Navigation* steps and because the expression of the navigation itself is not simple, the expression of *Set Arithmetic* is obfuscated, too.

The *Property Aggregation* is provided by the *repository based* approach, while in the other one approach it might be missed out (like in the case of *Set Arithmetic*).

Because the *structure based* approach provides *modularity*, it allows us to reuse analyses and compound them into more abstract and complex ones. The weak *modularity* provided by the *repository based* approach severely reduces the aforementioned properties.

4 The SAIL Language

As mentioned in the beginning, the main goal of this paper is to identify the causes of the complexity overhead that appears in the implementation of design metrics and to suggest how programming languages could support in a better, more simplified way, the writing of metrics. In the previous section we identified the key mechanism involved in metrics' implementation and discussed the limitations of current approaches. In this section we will introduce SAIL, a dedicated language that we defined and successfully used for the implementation of metrics and further design related structural analyses. The role of SAIL in this paper is mainly to illustrate how the identified mechanism could be more efficiently implemented at a language level, so that the various complexity overheads identified in Section 3 would be removed.

After the presentation of the *Meta-Model* we are going to present how each key mechanism is implemented in SAIL, using the examples defined in Section 3.2.

4.1 Meta-Model Approach in SAIL

In SAIL the meta-model is represented as an interconnected set of predefined data structures (*e.g.*, `Package`, `Class`, `Method`). A structure groups a number of fields of different types. What types? A structure may contain elementary types (a package has a name), structured types (a variable is an instance of a class) and collections (a packages has classes, a class has methods). The information extracted from the analyzed system is stored and accessed as *predefined* collection variables (*e.g.*, `sysPackages`, `sysClasses`, `sysMethods`). We are not going to discuss how are these predefined collections fulfilled because this is beside the point of this paper.

4.2 Filtering

In the beginning of this section we are going to provide the implementation in SAIL of the first example defined in Section 3.2.

In 1 we declare the package which will store the result. But in order to make it store the information about the package called `my.package` we need the assignation from 2. The value that is assigned to `myPack` is the result of a query.

```
1 Package myPack;
2 myPack = select (*) from sysPackages
3           where name="my.package";
```

On one hand, the efficient querying mechanism introduced in SAIL (*i.e.*, the `select` statement) contributes decisively to reducing the complexity overhead found in *structure based* approaches when complex navigation must be combined with filtering. On the other hand, because query results can be stored in SAIL variables, it becomes possible to "break down" the excessively complex monolithic queries often encountered in the *repository based* approach. Thus, the intelligence of the query can be better modularized, making the analysis easier to understand. The filter condition might be complex but in SAIL we can *encapsulate* its complexity into one or more functions. Mandatory, the return type of the used function in the `where` clause is `boolean`. A called function inside the `select` instruction increases the *readability* of this statement and in the same time it serves the *modularity* criteria by letting those functions be reused in the context of another analysis.

4.3 Navigation

The navigation in SAIL has a facile expression. Passing one level down in the hierarchy of contents means, like in the *structure based* approach, the access of an aggregated entity. Every deeper navigation requires the use of the assignment and the `select` statements. But the difference between the presented approaches and SAIL is the absence of the embedded `select` instructions which creates a clear implementation of the *Navigation* mechanism.

For the exemplification, in 2 we pass to the levels of methods belonging to the package obtained before by using the aforementioned instructions.

```
1 Method[] methods;
2 methods = select (methods) from myPack.classes;
```

If we liked to navigate to the level of defined variables in the methods, we would write one more select like below.

```
3 Variable[] variables;
4 variables = select (variables) from methods;
```

We can also navigate from the entity `myPack` directly to the level of variables, like in 5.

```
5 variables = select(classes.methods.variables)
              from myPack;
```

The collection in the `from` clause of the `select` instruction can be simple (like in the previous examples) or compound (*e.g.*, resulting from the union of two or more collections of the same type, from the call of a function whose result is a collection).

4.4 Selection

In order to perform a selection we need to define a structure which has only fields whose values are interesting for us. In 1 we define a new structure type which contains only the name and the signature of the method.

```
1 struct DetailedMethod{
2  string name;
3  string sign; };
```

Filling a collection of this type, like in the *repository based* approach is done by enumerating the interesting fields within the `select` instruction.

```
4 DetailedMethod[] detailed;
5 detailed = select (name, sign) from methods;
```

4.5 Set Arithmetic. Property Aggregation

SAIL provides support for the manipulation of collections and, consequently, writing an union of two or more collections requires only one operation (like in 3). Beside the union, SAIL brings support for the well-known intersection and difference operations with collections (all of them can be done between collections or between a collection and a structure variable of the same type as the elements of the collection). The collection arithmetic provided by SAIL also includes support for computing the cartesian product between two or more collections.

```
1 Method[] globals = myPack.globalFunctions;
2 methods += globals;
```

In SAIL we have two types of aggregation operators. The first type of aggregation operators use directly a collection (an example of this is provided in 3).

```
3 int classNo = myPack.classes.#;
```

The other category of aggregation operators, instead of a direct use of a collection, embeds a collection within the `select` instruction, like in 4.

```
4 classNo = select count(*) from myPack.classes;
```

5 Comparative Evaluation of Approaches

In this section we propose a comparative evaluation between the implementations of different object-oriented design metrics. We have implemented a suite of over 40 metrics in a *structure based* approach (using Java), a *repository based* approach (using SQL) and SAIL. Table 1 provides some characteristics about the implementation of the aforementioned suite.

	Java	**SQL**	**SAIL**
LOC	1650	1073	1461
Size	37580	48057	31012

Table 1. The implementations'characteristics

The number of lines of code (LOC) of the SQL implementations is smaller than any others. But in this case it does not mean a shorter implementation. If we take a look at the second row of the table, we find the SQL implementations are the largest in terms of bytes. On average, a SQL line of code contains 44.73 bytes while the one written in Java and SAIL have 22.8, resp. 21.2 bytes.

According to Ghinsu's[11] code reduction module, there are some general properties that usually characterize simpler programs: they are shorter in size, have fewer variables and fewer nested constructs[23].

Metrics written in SAIL are shorter in size than any others. We are going to illustrate how the rest of the properties mention before are fulfilled in SAIL by presenting details about the implementations of two metrics, *Tight Class Cohesion* (TCC)[3] and *Changing Methods* (CM)[17].

TCC is defined as the relative number of directly connected methods defined in the analyzed class. Two methods are connected if they access a common instance variable of the class. In Java, beside imports, declarations and assignments, the implementation requires two embedded cycling instructions through the defined methods in the class and

LOC	Java	SQL	SAIL
TCC	53	31	21
CM	89	25	25

Table 2. LOC for TCC and CM

Size	Java	SQL	SAIL
TCC	2016	1346	893
CM	3600	1152	933

Table 3. Size in bytes of TCC and CM

between each member of the pair of methods (inside the cycling instructions) we have to check if there is a common access. Finding the common access requires the use of another cycling instructions. Thus, the Java implementation of TCC has 3 nested cycling instructions and 13 variables. The implementation in SQL consists in the creation of 3 temporary tables that are filled by using 3 `select` instructions. Beside this we also have 3 embedded `select` instructions. In SAIL the expression of TCC has 4 variables and does not contain any nested cycling instructions. The simplicity is provided by the powerful *Navigation* mechanism doubled by the proper operation for manipulating collections (*e.g.*, in this case, the cartesian product between the methods of the analyzed class).

CM measures the dispersion of the changes, (*i.e.*, in how many classes are spread the methods that are potentially affected by a change in the given class). A method depends on a class if invokes methods or redefines methods defined in the class or accesses its attributes. Obviously, the result of this analysis is a union of three collections. In Java, CM contains 19 variables and 3 cycling instructions and each of them embeds at least another cycling instruction. In SQL the implementation consists of a union of the three involved entities and this union is surrounded by 2 `select` statements. In SAIL we have 3 variables. Finding the overridden methods is done using 2 `select` statements and obtaining the others involved collection requires another two `select` statements. But the miss of the embedded instructions brings clarity into the implementation.

6 Related Work

We dedicate this section to a briefing of several representative solutions that fall in (or are closely related with) the two implementation approaches: the *structure based* and the *repository based* one. Eventually, we are going to discuss the relation between SAIL, Embedded SQL, OQL and GOQL approaches as apparently they are similar.

Approaches Based on Structural Models. The approaches based on structural models are used both in the research and the commercial world. One research environment that is based on such approach is *Moose*[6]. This platform uses an object-oriented meta-model, is implemented in *Smalltalk* and is used intensely for various structural analyses in the context of object-oriented re-engineering. Due to the language powerful support for collections and, especially, to the `select` message that can substitute the SQL SELECT, this approach is the only adequate alternative to SAIL that we found so far. On the commercial side a good (and popular) example is *IBM Eclipse*[21] which also defines an object-oriented representation of the meta-model and supports the definition of metrics via a Java API. Thus, metrics must be written as Java programs. As we have seen in Section 3.4 this makes their writings verbose and complex due to an insufficient support for navigation and filtering.

Approaches Based on Repository Models. An interesting one is found in[4], where Ciupke proposes an approach and a tool kit for detecting automatically violation of design heuristics like those defined by Riel[19]. For this, he stores the system model in form of Prolog fact repository, while the heuristics are implemented as queries on this repository. The queries are formulated as Prolog rules that analyze the model and deliver the locations of problems. As the author himself acknowledges, this approach requires logical programming skills and even so, is adequate only for structural analyses of moderate complexity. In spite of its limitations, this approach illustrates the fact that repository-based approaches are not limited to relational databases and SQL programming.

A commercial tool for auditing that takes the repository approach for defining analyses is *Audit C/C++*[5]. The tool was also used in the context of re-engineering research[20, 2] mainly for the implementation of design metrics. In this concrete case the source code model is stored in form of a set of 4-5 plain ASCII data tables with a predefined format. They build the repository. Analyses are then written in CQL, which is a subset of SQL mainly focused on the interrogation instruction (SELECT). All the disadvantages and limitations already discussed in Section 3.4 are totally applicable in this case[14].

Embedded SQL. The term *embedded SQL* refers to SQL statements placed within a procedural (or object-oriented) program. This approach improves the modularity of the code over standard SQL and it does also allow the use of cycling and decisional instructions. For example PL/SQL[9] is a procedural language that was designed to fill the gaps of standard SQL by allowing Oracle database developers to interface with the underlying relational database in an imperative manner. PL/SQL is analogous to the embedded procedural languages for other relational databases (*e.g.*, Transact-SQL, PL/pgSQL etc). What makes embedded SQL different from our approach? Apparently, SAIL is just a procedural language that ”embeds” an SQL-like SELECT statement. The main difference resides in the relation between the data model and the language instructions that manipulate the data. In *embedded SQL* the model of the analyzed code is still stored in a relational database and therefore the ”intelligence” of metrics that need a lot of navigation and filtering tends to be still centralized in complex SELECT statements. Thus, it is hard to distribute their logic between the declarative statements (*i.e.*, SQL’s SELECT) and the imperative statements (*e.g.*, loops, branches etc) because the data model is not directly manipulable in a procedural manner. On the contrary, in SAIL the model, based on data structures, can be used without any overhead both in the imperative statements and in the SELECT instruction. This lack of overhead in accessing the model reduces the complexity when implementing metrics in SAIL.

OQL. Even object-oriented database systems did not gain general acceptance yet[13], we are going to emphasize some similarities and differences between SAIL and OQL[8], an object-oriented query language. Both of the languages have a `select` instruction, similar to the one found in SQL. Within the `select` instruction from OQL the *Navigantion* can not have the facile expression like the one in SAIL. In OQL we can not navigate with only one `select` from an entity two levels down in the hierarchy of contents as long as the first level we pass through is a collection of entities. OQL can be used interactively in the command prompt or as a function called from other languages (*e.g.*, C++, Java, O_2C - a programming language specialized for developing object-oriented database applications[7]). It is obvious the first way of using OQL does not fit ours needs. Concerning the second way of using OQL, we are going to present how OQL is used within O_2C. O_2C has a predefined function `o2query` which has a parameter that embeds the OQL select instruction. In this case, the expression of selection’s criteria is actually a `string` and is not validated by the compiler before passing it to OQL. In addition, in SAIL the mechanism of querying is integrated directly into the language.

GOQL. GOQL is an OQL-like query language, for querying graphs[22]. GOQL has, like OQL, the traditional `select` instruction from SQL. But in the `where` clause of the instruction, beside the conditions for filtering, we have also conditions regarding the interconnection of the elements that, like in SQL, obfuscate the expression of the *Navigation*.

7 Conclusions. Future Work

Structural analyses are getting more and more complex. We have identified a set of key mechanisms based on which code analysis are built: navigation, selection, set arithmetic, filtering and property aggregation. We have analyzed in detail two major analyses’approaches and found out that a considerable complexity overhead is due to insufficient language support for the key mechanisms identified in the beginning. The implementation approaches based on procedural or object-oriented programming languages are especially unsatisfactory for non-trivial combinations of model *navigation* and *filtering*. The approaches based on querying a repository have another major problem: they miss adequate mechanisms that would support a better modularization. This oftentimes leads to analyses that consist of a monolithic query, which is very hard to maintain.

In this paper we introduce SAIL, a language designed to offer a proper support for all the needed key mechanisms. There are three major ideas in *SAIL*:

1. The **integration of query mechanism** (*i.e.*, the `select`) in a very simple structured programming language which is syntactically very close to known programming languages like C and Java. This brings a twofold advantage: the language ”grabs” the key advantages of a query language and it requires almost no learning effort for a programmer, due to its syntactic similarity with C and Java.

2. The **representation of the data model in *SAIL***, although totally based on *data structures*, can be used without any overhead both in imperative statements and in the query mechanism.

3. The **simple manipulation of collections** in SAIL proved to play an important role in simplifying the writing of metrics (or metrics-related analyses). This is mainly because, as we have seen in this paper, set operations are an essential building stone in all non-trivial analyses.

The paper validates the claim that the use of SAIL would lead to a simplified expression of metrics (or metrics-related analyses) by comparing the size and complexity of implementation for a suite of over 40 metrics, most of them quite complex (*e.g.*, TCC and CM which are also discussed in more detail). These metrics were all implemented in Java, SQL and respectively SAIL. One one hand, the comparison has revealed that SQL implementations usually need less lines of code to be implemented than Java or SAIL but the complexity of each line highly exceeds those of found in the approaches based on structural/object-oriented programming. On the other hand, comparing SAIL and Java implementations, metrics written in SAIL prove to be significantly more concise than those implemented in Java. This supports the hypothesis that while keeping the "shape" of a procedural language, SAIL adds to it the conciseness of a query languages.

Acknowledgments This work is supported by the Austrian Ministry BMBWK under Project No. GZ 45.527/1-VI/B/7a/02. We owe a lot to Violeta Voinescu for taking her time to review the paper and to Daniel Cioată for testing intensively the SAIL interpreter. Last but not least we would like to thank the LOOSE Research Group for being such a great and challenging team.

References

[1] J. Bansiya and C.G. Davis. A Hierarchical Model for Object-Oriented Design Quality Assessment. *IEEE Transactions on Software Engineering*, 2002.

[2] H. Bär, M. Bauer, O. Ciupke, S. Demeyer, S. Ducasse, M. Lanza, R. Marinescu, R. Nebbe, O. Nierstrasz, M. Przybilski, T. Richner, M. Rieger, C. Riva, A. Sassen, B. Schulz, P. Steyaert, S. Tichelaar, and J. Weisbrod. *The FAMOOS Object-Oriented Reengineering Handbook*. 1999.

[3] J.M. Bieman and B.K. Kang. Cohesion and Reuse in an Object-Oriented System. *Proc. ACM Symposium on Software Reusability*, 1995.

[4] O. Ciupke. Automatic Detection of Design Problems in Object-Oriented Reengineering. In *Technology of Object-Oriented Languages and Systems*, 1999.

[5] Sema Group Corp. *User Manual Concerto2/Audit-CC++*. Sema Group, France, 1998.

[6] S. Ducasse, M. Lanza, and S. Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. In *Proceedings of the Second International Symposium on Constructing Software Engineering Tools*, 2000.

[7] Deux et al. *The O_2 System*. Communications of ACM, 1991.

[8] R. Cattell et al. *The Object Data Standard*. Morgan Kaufmann, 2000.

[9] S. Feuerstein. *Oracle PL/SQL Programming, 3rd ed.* O'Reilly and Associates, 2002.

[10] K. Gayda. *Introduction to Transact SQL User-Defined Functions*. Jupitermedia Corp, 2000.

[11] P.E. Livadas and P.K. Roy. Program dependence analysis. In *Proc. IEEE Conference on Software Maintenance*, 1992.

[12] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics*. Prentice-Hall, 1994.

[13] W. Mahnke and H. Steiert. *Modularity in ORDBMSs-A new Challenge.* Grundlagen von Datenbanken, 2001.

[14] R. Marinescu. The Use of Software Metrics in the Design of Object-Oriented Systems. Diploma Thesis, "Politehnica" University Timişoara, 1997.

[15] R. Marinescu. Using Object-Oriented Metrics for Automatic Design Flaws in Large Scale Systems. In Serge Demeyer and Jan Bosch, editors, *Object-Oriented Technology (ECOOP'98 Workshop Reader)*. Springer-Verlag, 1998.

[16] R. Marinescu. A Multi-Layered System of Metrics for the Measurement of Reuse by Inheritance. In *Proceedings of TOOLS Asia*, 1999.

[17] R. Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, "Politehnica" University of Timişoara, 2002.

[18] R. Marinescu. Detection strategies:metrics-based rules for detecting design flaws. In *Proc. IEEE International Conference on Software Maintenance*, 2004.

[19] A.J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[20] A.M. Sassen and R. Marinescu. Metrics-Based Problem Detection in Object-Oriented Legacy Systems Using Audit-Reengineer. In *Proceedings of the ECOOP Workshop on Experiences in Object-Oriented Re-Engineering*, 1999.

[21] S. Shavor, J. DAnjou, and J. Kellerman P. McCarthy S. Fairbrother, D. Kehn. *The Java Developer's Guide to Eclipse*. Addison-Wesley, 2003.

[22] Lei Sheng, Z. Meral Özsoyoglu, and Gultekin Özsoyoglu. A graph query language and its query processing. In *Proc. International Conference on Data Engineering*, 1999.

[23] E. Skordalakis and N. Papaspyron. *Ghinsu's Code Reduction Module*. Software Engineering Laboratory, National Technical University, Athens, 1995.

[24] S. Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, Institute of Informatics and Applied Mathematics, University of Bern, 2001.