The Logical Modularity of Programs

Daniel Ratiu
Technische Universität München
Germany
Email: ratiu@in.tum.de

Radu Marinescu
Politehnica University Timisoara
Romania
Email: radum@cs.upt.ro

Jan Jürjens
The Open University
Great Britain
Web: www.jurjens.de/jan

***Abstract*—The principles and best practices of object oriented design require that modules in a program should match logical decomposition of the knowledge that the program implements. The violation of these modularization rules leads to several undesired consequences: (i) non-cohesive modules that mix different kinds of knowledge and (ii) logically coupled modules due to a dispersion of conceptually cohesive knowledge. In this paper, we use domain knowledge driven program analysis to detect and characterize discrepancies between the structural modularization of programs and the conceptual decomposition of the implemented knowledge. We characterize the mismatches at the levels of granularity of packages and classes and present their impact on different maintenance activities. The presented approach includes a technique for automating the recovery of mappings between the different categories of knowledge used in the program and the modules that implement them. We briefly present our experience with analyzing JHotDraw.**

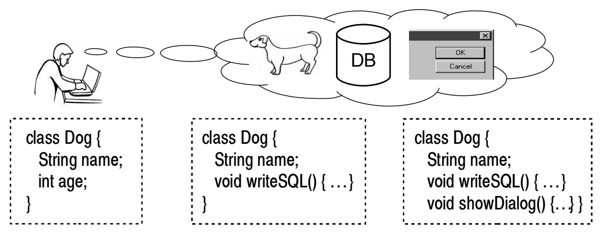
***Keywords*-modularity of programs, domain knowledge driven program analysis, ontologies**

I. Introduction

" [...] the class Dog is functionally cohesive if its semantics embrace the behavior of a dog, the whole dog, and nothing but the dog." [1, p. 113]

Programming is modeling the reality. Heuristics of good object-oriented design advice that concepts of the modeled domain and their relations should be represented by correspondingly related program entities [1], [2], [3]. The situation in the practice is however much different. In Figure 1 we present intuitively how different kinds of knowledge can be implemented in the code. On the leftmost side, we have a good implementation of the abstraction "Dog", but in the center and on the rightmost side the class Dog implements also concepts belonging to other categories of knowledge. This is an example where the structural code modularization does not follow the modularization of the domain knowledge. Due to the different kinds of knowledge the rightmost class is more complicated to understand and has different evolution directions. Inappropriate modularization can lead to different problems: more difficult comprehension, difficult extensibility with other concepts, architectural erosion and/or the occurrence of code smells.

Most object-oriented programming textbooks advise that a class should implement a *single abstraction of the domain*



```
class Dog {
  String name;
  int age;
}
```

```
class Dog {
  String name;
  void writeSQL() { ...}
}
```

```
class Dog {
  String name;
  void writeSQL() { ...}
  void showDialog() {..} }
```

Figure 1. Good and bad decomposition

[4], [3]; it is also recommended that the starting point for identifying classes should be deeply routed in the modeled domain, which is reflected by the nouns identified in the requirements documents [5], [6]. At an architectural level, modularity principles and rules recommend that there is a *Direct Mapping* between the modular structure created in the process of building a software system and any modular structure that results from the process of modeling the problem domain [2]. In conclusion, there is a widespread acceptance that analyzing the modularity of a system by focusing exclusively on its implementation structure is insufficient. In order to assess appropriately the modularity of a system, the relation between a program and its modeled domain must be taken into account. This paper continues our previous research on domain knowledge driven program analysis in the direction of assessing the logical modularity of programs. In Section II we present our framework to assign domain meaning to programs. In Section III we characterize different mismatches between the program modularity at the package and class level and the logical decomposition of the conceptual space implemented in the program. In Section IV we describe our experience with assessing the modularity of JHotDraw. Section V summarizes the related work and Section VI presents our conclusions.

II. Assigning domain meaning to programs

In order to enable the assessment of the logical modularity of programs, we need to map the domain knowledge to the program modules that implement it. Our general framework to give domain meaning to programs [7] is based on three ingredients (Figure 2): *program abstraction*, *semantic domain*, and *interpretation*. In order to instantiate this framework, one has to specify precisely these ingredients as dictated by the specific analysis use-case. In this paper our use-case

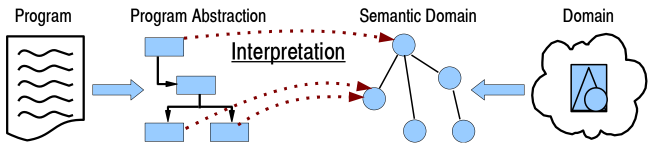


Figure 2. Assigning meaning to programs

is to evaluate a program's architecture with respect to the differences between the conceptual decomposition of the knowledge implemented in the program and the structural decomposition of the code. In order to do this, the basic question to be answered is: *which categories of knowledge (e. g. business domain, GUI, persistency) are contained in a certain module?* To answer this question, we instantiate our framework as presented in the following.

1) Program Abstraction: To evaluate the mismatches between the structural and logical modularization of programs, our program abstraction should capture the structural decomposition.

Definition: *A program consists of a set of packages P. Each package $p \in P$ consists of a set of classes C_p. The set of classes of the system is denoted through C.*

In Figure 3 (bottom) we illustrate an example of a program that contains two packages, the first package containing two classes and the second package containing one class.

2) Semantic Domain: Typically, the knowledge that is implemented in a program can be divided into different categories (e. g. persistency, user interfaces, business domain). These kinds of knowledge usually represent "axes of change" and are most commonly used by the heuristics that characterize the architecture. We thereby partition the conceptual space into "knowledge dimensions". A knowledge dimension is a set of concepts that are characteristic for a certain domain of interest. Since we are interested only to group concepts into knowledge dimensions, we need only a weak ontology that partitions the conceptual space into different dimensions that contain flat-ordered concepts.

Definition: *$\mathcal{D}$ is a well-structured set of knowledge dimensions if the following conditions hold:*

i) each dimension $d \in \mathcal{D}$ has a concept called `d_Thing` *that is the root concept of the dimension,*

ii) `d_Thing` *is a subordinate of the concept* `Thing` *that represents the root of conceptual space,*

iii) the set of concepts that belong to a dimension d are arranged into a flat hierarchy under `d_Thing`*,*

iv) the sets of concepts belonging to two different dimensions are disjunct.

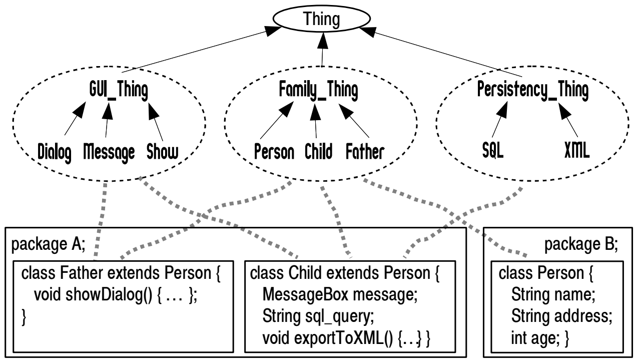
In Figure 3 (top) we present an example of a weak ontology that contains concepts arranged into three knowledge dimensions: Persistency, GUI, or Family. In order to enable the automatic identification of occurences of dimensions in code – based exclusively on the similarities between the concepts and program elements names – we include in our ontology only the concepts with names that are characteristic for a dimension (i. e. whose name is not a polysemous word).

3) Interpretation: As mentioned before, for evaluating the logical modularization of the program, we need only to know that a certain kind of knowledge is referenced from a particular program module.

Definition: *Let C be a set of classes and $\mathcal{D}$ a set of knowledge dimensions. We define $\overrightarrow{Ref_C} : C \rightarrow \wp(\mathcal{D})$ to be a function that given a class associates a set of dimensions whose concepts it references. The function $\overleftarrow{Ref_C} : \mathcal{D} \rightarrow \wp(C)$,*

$$\overleftarrow{Ref_C}(d) = \{c \in C \mid d \in \overrightarrow{Ref_C}(c)\}$$

returns for each knowledge dimension the set of classes that refer at least one of the concepts from this dimension.



```
class Father extends Person {
  void showDialog() { ... };
}
```

```
class Child extends Person {
  MessageBox message;
  String sql_query;
  void exportToXML() {..} }
```

```
class Person {
  String name;
  String address;
  int age; }
```

Figure 3. Knowledge dimensions reference

Definition: *Let P be a set of packages and $\mathcal{D}$ a set of knowledge dimensions. We define $\overrightarrow{Ref_P} : P \rightarrow \wp(\mathcal{D})$ to be a function that given a package associates a set of dimensions with it as defined below:*

$$\overrightarrow{Ref_P}(p) = \bigcup_{c \in C_p} \overrightarrow{Ref_C}(c)$$

where C_p is the set of classes of package p. The function $\overleftarrow{Ref_P} : \mathcal{D} \rightarrow \wp(P)$ with

$$\overleftarrow{Ref_P}(d) = \{p \in P \mid d \in \overrightarrow{Ref_P}(p)\}$$

returns for each knowledge dimension the set of packages which reference it.

For example, in Figure 3 we present the reference function by a dotted line – e. g. $\overrightarrow{Ref_C}(Child) = \{Family, GUI, Persistency\}$, $\overleftarrow{Ref_C}(GUI) = \{Father, Child\}$, $\overrightarrow{Ref_P}(B) = \{Family\}$, $\overleftarrow{Ref_P}(Family) = \{A, B\}$. Since we choose in our knowledge dimensions only concepts whose names do not exhibit polysemy, we can automatically recover the interpretation by identifying the occurences of concepts in modules using only the similarity between the names of concepts and of program elements.

III. Characterizing the logical modularity

We use the functions $\overrightarrow{Ref_C}$, $\overleftarrow{Ref_C}$, $\overrightarrow{Ref_P}$, $\overleftarrow{Ref_P}$ to characterize the modularity from the point of view of delocalization of dimensions (i. e. a dimension is scattered along more modules), interleaving of dimensions (i. e. situations when more knowledge dimensions are referenced in the same module), and misplacements of classes in packages.

Dimension delocalization in packages. *The delocalization degree of a knowledge dimension d in packages is:*

$$Deloc_P(d) = |\overleftarrow{Ref_P}(d)| \ / \ |P|$$

Intuitively, the delocalization of a dimension in packages represents the ratio of packages that reference this knowledge dimension. The higher the $Deloc_P(d)$ the lower the modularity of the system at the package level with respect to the knowledge dimension d.

Dimension delocalization in classes. *The delocalization degree of a knowledge dimension d in classes is:*

$$Deloc_C(d) = |\overleftarrow{Ref_C}(d)| \ / \ |C|$$

Intuitively, the delocalization of a dimension in classes represents the ratio of classes that reference this knowledge dimension. The higher the delocalization degree the lower the modularity of the system at the class level with respect to this knowledge dimension. Ideally, dimensions that are only marginal to the project (e. g. persistency, communication) should have a low delocalization both at a package and at a class level.

If for a given dimension d, a set of packages P and a set of classes C we have $Deloc_C(d) < Deloc_P(d)$ this means that the classes are better modularized than packages and that the de-modularization occurs at the level of packages.

Dominant dimension. *Let P be the set of packages and $\mathcal{D}$ a set of knowledge dimensions. We define the function:*

$$Dom : P \rightarrow \wp(\mathcal{D}), \quad where$$
$$Dom(p) = \{d \in D \mid \forall d' \in D. \ |\overleftarrow{Ref_C}(d')| \leq |\overleftarrow{Ref_C}(d)| \ \}$$

Intuitively, for each package the function Dom associates a set of knowledge dimensions that are most often referenced by the classes from the package. The dominant dimension gives information about the main decomposition direction of the package.

Interleaving degree of package. *For a package $p \in P$ we define the interleaving degree to be*

$$Int_P(p) = |\overrightarrow{Ref_P}(p)|$$

Intuitively, the interleaving degree is the number of dimensions referenced from within the package. In the case of an ideally modularized package $Int_P(p) = 1$. The higher the Int_P, the more dimensions are interleaved and therefore the less cohesive the package is from a conceptual point of view. If $Int_P(p) > 1$ we need to inspect the decomposition at the class level in order to understand the nature of interleaving in this package. For example, if the package contains classes that refer to a single knowledge dimension they can be moved to other packages. In the cases when a package references more knowledge dimensions, one (or more) of these can have a dominant role, while others might be spanned only across a few classes.

Dimension spanning. *The spanning of a dimension $d \in \mathcal{D}$ in a package $p \in P$ that contains the classes C_p is:*

$$Spanning(d) = |\{c \in C_p \mid d \in \overrightarrow{Ref_C}(c)\}| \ / \ |C_p|$$

Intuitively, the spanning of a dimension in a package represents the ratio of classes from the package that reference this dimension. The spanning degree is zero when no class references the dimension. The dominant dimension of a package has the highest spanning degree (ideally it should be one). In the case of dimensions with spanning degree very small, they are due to spurious occurrences of the dimension in a small number of package classes. Such classes might be misplaced (as defined below) and therefore should be moved to other packages.

Misplaced classes. *Let $p \in P$ be a package and C_p the set of classes from p. A class $c \in C_p$ is misplaced iff:*

$$\overrightarrow{Ref_C}(c) \cap Dom(p) = \emptyset$$

Intuitively, a class is misplaced when it does not reference the dominant dimension of the package to which it belongs. Classes that are misplaced make the package less cohesive. In order to increase the logical cohesiveness, we should move the misplaced classes to other packages that contain the same dimensions of knowledge that the misplaced classes themselves.

Interleaving degree of a class. *For a class $c \in C$ we define the interleaving degree to be*

$$Int_C(c) = |\overrightarrow{Ref_C}(c)|$$

Intuitively, the interleaving degree of a class represents the number of knowledge dimensions that are referenced within that class. The higher the interleaving degree, the less modular the class is, the less (logically) cohesive and thereby harder to understand. Furthermore, a class with high interleaving degree has a high conceptual overload since it references more dimensions of change.

IV. Logical modularity of JHotDraw

JHotDraw is a framework for drawing 2D graphics and contains 20 packages, 359 compilation units, and 1369 normalized words. We have chosen JHotDraw because it is a well documented software, has a high quality, and is popular among the reverse engineering community. Before we performed the recovery of $\overrightarrow{Ref_C}$ we firstly used Eclipse for optimizing imports (i. e. eliminate import statements that are not used).

We investigate the measure in which persistency knowledge (IO) is modularized with respect to the graphical user interface knowledge (UI). This is why, we chose two knowledge dimensions 'persistency' and 'UI'. We built the ontology by analyzing the words contained in the identifiers of JHotDraw and manually classifying the words into knowledge dimensions (and thereby lifting words to concepts). The ontology contains only concepts whose names are words that are not polysemous and clearly denote concepts about UI or persistency. For example, the word 'path' is used in a polysemous manner since it refers both to a curve and to the path of a file; the words 'tree' or 'table' are more general since they can refer to UI or to data structures. All these words were left out. In ca. two hours we harvested 73 concepts about GUI and 15 about persistency. Below we present the entire ontology (e. g. the list of concepts assigned to IO and UI dimensions) used for analyzing JHotDraw.

UI (73 concepts): antialias, area, arrow, awt, bezier, border, button, canvas, cascade, circle, color, component, connector, curve, dialog, direction, display, document, drag, drawing, draw, drop, editable, editor, ellipse, figure, focusable, focus, font, frame, geometry, geom, gradient, graphics, gui, height, horizontal, iconifiable, iconified, label, layouter, layout, menu, metric, paint, pane, panel, pixel, point, polygon, radius, rectangle, redo, rgb, scale, screen, shadow, shape, sheet, showing, show, stroke, submenu, swing, triangle, undoable, vertical, view, viewport, visible, width, window, zoom
IO (15 concepts): cdata, css, directory, dom, dtd, file, ixml, nanoxml, parse, parser, read, write, writer, xmldom, xml

Delocalization. In Table I we present an overview over the delocalization of IO and UI knowledge in JHotDraw. In the third column we notice that UI is dominating dimension in 75% of packages, while IO is dominant only in 20% of packages. By looking at $Deloc_C$ and $Deloc_P$, we remark that the UI knowledge is spreaded in most of the classes and packages of the system. In the case of IO, even if only a third of classes reference it, it is spread across 55% of packages. This fact suggests a deficiency of the modularization of IO knowledge also at the package level.

	$Deloc_P$	$Deloc_C$	$Dominance$
UI	0.85	0.89	0.75
IO	0.55	0.32	0.20

Table I
JHOTDRAW DELOCALIZATION

In the table from Figure 4 we present the entire list of packages from JHotDraw and the spanning degrees ($Spanning$) of the IO and UI knowledge dimensions in these packages. We notice that in many packages only a small ratio of classes reference IO knowledge. For example, in the package org.jhotdraw.gui the spanning degree of IO knowledge is very small. Also, the packages org.jhotdraw.application.action and org.jhotdraw.app.action contain IO knowledge in only ca. a third (34% and 37%) of their classes. The package org.jhotdraw.util has dominant knowledge IO, but this spans over only 25% of the classes (packages called "util" usually are collection of utilities with different functionalities).

Package	UI	IO	Package	UI	IO
org.jhotdraw.xml	0.5	1	org.jhotdraw.draw	1	0.25
org.jhotdraw.xml.css	0	1	org.jhotdraw.draw.action	0.97	0
org.jhotdraw.util	0.25	0	org.jhotdraw.gui.datatransfer	1	0
org.jhotdraw.util.prefs	1	0	org.jhotdraw.application	1	0.58
org.jhotdraw.undo	1	0	org.jhotdraw.application.action	0.97	0.34
org.jhotdraw.io	0.33	1	org.jhotdraw.app	1	0.58
org.jhotdraw.gui	1	0.06	org.jhotdraw.app.action	1	0.37
org.jhotdraw.beans	0	0	org.apache.batik.ext.awt	1	0
org.jhotdraw.gui.event	1	0.5	org.apache.batik.ext.awt.image	1	0
org.jhotdraw.geom	1	0	net.n3.nanoxml	0	1

Figure 4. JHotDraw package decomposition

Interleaving. At the package level, the persistency knowledge is interleaved with UI knowledge in 45% of the packages while at the class level the interleaving is much smaller (23% of classes). We also notice that 50% of packages and 75% of classes are ideally modularized with respect to the chosen dimensions. In the table from Figure 4 we can see to which degree does a package reference UI and IO knowledge. We notice that some packages are ideally modularized – e. g. org.jhotdraw.geom or net.n3.nanoxml; while other packages – e. g. org.jhotdraw.draw, org.jhotdraw.io, org.jhotdraw.xml – mix knowledge dimensions. In the following we present several packages that contain interleaved knowledge in more detail.

In package org.jhotdraw.draw the dominant knowledge dimension is UI as we expected (according to the documentation, this package implements the core of the business domain), but the package contains also many references (25% of classes) to persistency knowledge. By analyzing the interleaving degrees at the class level, we found that IO knowledge is referenced from classes from the hierarchies of Figure and Drawing, and from other classes. In Figure 5(a - f) we present code samples from this package. Furthermore, we identified several classes that are badly packaged – e. g. OutputFormat or InputFormat would better fit in another package.

The packages org.jhotdraw.io and org.jhotdraw.xml contain knowledge about the UI, even if their dominant dimension is IO (as their names suggest). As shown in Figure 5h, the class ExtensionFileFilter reference UI knowledge since it inherits javax.swing.filechooser.FileFilter. As shown in Figure 5i, the class JavaxDOMInput references UI knowledge by building objects of type Color from the XML documents.

The package org.jhotdraw.gui.event references the IO knowledge since one of its classes responsible for reacting to events uses a JFileChooser dialog (Figure 5g). The package org.jhotdraw.app references IO through the class Project that knows about the files where the JHotDraw drawings are saved into (Figure 5j). The last two examples are less harmful and represent places in the code which naturally make the integration between the UI and IO functionalities.

```
package org.jhotdraw.draw;
public class AbstractCompositeFigure ... {
   void layout() { . . . }
   void write(DOMOutput out) { . . . }
   void read(DOMInput in) { . . . }
...}                                  a)
```

```
package org.jhotdraw.draw;
public class AbstractDrawing ... {
     int getFigureCount() { . . . }
     void write(DOMOutput out) { . . . }
     void read(DOMInput in) { . . . }
. . .}                                b)
```

```
package org.jhotdraw.draw;
public class ImageFigure ...{
   void drawFill(Graphics2D g) { . . . }
   void write(DOMOutput out) { . . . }
...}                                  c)
```

```
package org.jhotdraw.draw;
public class ImageTool ... {
     void creationFinished( ...) { ...
File file = getFileChooser().getSelectedFile();
 . . .} ... }                         d)
```

```
package org.jhotdraw.draw;
public interface OutputFormat {
   void write(File file, Drawing drawing);
...}                                  e)
```

```
package org.jhotdraw.draw;
public interface InputFormat {
     void read(File file, Drawing drawing);
. . .}                                f)
```

```
package org.jhotdraw.gui.event;
public class SheetEvent ...{
     public SheetEvent(JSheet source,
        g)    JFileChooser fileChooser, ...)
```

```
package org.jhotdraw.io;
public class ExtensionFileFilter extends
         javax.swing.filechooser.FileFilter { ...}
                                      h)
```

```
package org.jhotdraw.xml; . . .
public class JavaxDOMInput ... {
  public Object readObject(int index) ... { ...
    } else if (tagName.equals("color")) {
       o = new Color(getAttribute("rgba",0xff));
    } ...                             i)
```

```
package org.jhotdraw.app;
public interface Project {
     public JComponent getComponent();
     public File getFile(); . . .
}                                     j)
```

Figure 5. Code samples from JHotDraw

V. Related Work

Information retrieval based modularity analyses. Information retrieval techniques like Latent Semantic Indexing are used to characterize the architecture from a logical point of view in terms of logical cohesion [8], coupling [9] or distribution of "linguistic concepts" [10]. Marcus and Poshyvanyk [8] define conceptual cohesion metrics for classes that are based on the lexical similarity between the words used in the implementation of methods in the same class. Similarly, coupling metrics can be defined using the lexical similarity of the words from methods belonging to different classes [9]. The measurements of coupling and cohesion in these papers are based exclusively on statistics. Therefore, if the cohesion metric for a class does not return a high value, it might be not clear what the problem with the class is. In our work we interpret programs via domain ontologies as basis for assessing the logical modularity in terms of the spreading of knowledge in the code, interleaving of different knowledge in single modules or detecting misplaced classes.

Ducasse and his colleagues [10] present a general framework for analyzing the distribution of different properties (such as concepts) on the program structure. An instantiation of the framework is the analysis of the distribution of concepts (extracted by using LSI) over the packages of the system. Beside the fact that be interpret programs via ontologies for linking domain knowledge to programs, we perform more advanced analyses that consider both interleaving and delocalization at the package and class levels.

Cross-cutting concerns. Another branch of research that assesses the logical modularity of programs is based on detecting and measuring cross-cutting concerns [11], [12].The concerns are typically seen as pieces of functionality while our notion of knowledge dimensions is more general. The distribution of concerns can be used for developing metrics that address the modularity of software architectures [12]. Some of the measurements proposed by us are similar to the "concern diffusion", "coupling between components", or "component cohesion". Figueiredo and his colleagues [11] develop a general framework for the characterization of concern measures that assess the maintainability of aspect oriented software. Our work is more general and addresses not only functional concerns but different categories of knowledge. We assess the modularity from a logical, conceptual point of view, while the concern measurements assess the modularization of aspects.

VI. Conclusions

In this paper we continue our work on domain knowledge driven program understanding and propose a method for using the relation between domain knowledge and program modules in order to characterize the modularity of the code. Our method enable us to make well known design heuristics directly and automatically analyzable by capturing the knowledge into a weak ontology that represents a partition of the conceptual space into knowledge dimensions.

References

[1] G. Booch, *Object-Oriented Analysis and Design with Applications (3rd Edition)*. Addison Wesley, 2004.

[2] B. Meyer, *Object-Oriented Software Construction*, 2nd ed. Prentice-Hall, 1997.

[3] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship (Robert C. Martin Series)*, 1st ed. Prentice Hall PTR, August 2008.

[4] B. Eckel, *Thinking in Java*, 3rd ed. Prentice Hall, 2002. [Online]. Available: http://www.mindview.net/Books/TIJ/

[5] R. Wirfs-Brock and A. McKean, *Object Design — Roles, Responsibilities and Collaborations*. Addison-Wesley, 2003.

[6] S. W. Ambler, *The Object Primer : Agile Model-Driven Development with UML 2.0*. Cambridge University Press, March 2004.

[7] D. Ratiu, "Intentional meaning of programs," Ph.D. dissertation, Technische Universität München, 2009.

[8] A. Marcus and D. Poshyvanyk, "The conceptual cohesion of classes," in *ICSM '05*. IEEE CS, 2005, pp. 133–142.

[9] D. Poshyvanyk and A. Marcus, "The conceptual coupling metrics for object-oriented systems," in *ICSM '06*. IEEE CS, 2006, pp. 469–478.

[10] S. Ducasse, T. Girba, and A. Kuhn, "Distribution map," in *ICSM'06*. IEEE CS, 2006.

[11] E. Figueiredo, C. Sant'Anna, A. Garcia, T. T. Bartolomei, W. Cazzola, and A. Marchetto, "On the maintainability of aspect-oriented software: A concern-oriented measurement framework," in *CSMR'08*, 2008, pp. 183–192.

[12] C. Sant'Anna, E. Figueiredo, A. F. Garcia, and C. J. P. de Lucena, "On the modularity of software architectures: A concern-driven measurement framework," in *ECSA'07*, 2007.