A Context-Aware Style of Software Design

Francesca Arcelli Fontana, Pietro Braione, Riccardo Roveda, Marco Zanoni

Dipartimento di Informatica, Sistemistica e Comunicazione (DISCo)
University of Milano-Bicocca
P.zza dell'Ateneo Nuovo, 1
20126, Milano, Italy
Email: [arcelli, braione, zanoni]@disco.unimib.it, r.roveda@campus.unimib.it

***Abstract*—Contemporary large software systems rely on complex software ecosystems for managing infrastructural tasks. While these ecosystems facilitate software development, the software architect must put care in not relying on assumptions on behaviors and policies of the ecosystem that may change with platform evolution. Based on our experience with developing analyses within MARPLE, a framework for software comprehension and architecture reconstruction, we propose an abstract, *context-aware* style for specifying software. In the spirit of decoupling computation from coordination, this style decouples the specification of the computations to be performed from the specification of the contexts where they must take place. Software described in this way exposes its primitives at a level of abstraction closer to that of the framework, enabling better reasoning on the features of the design, easing correct implementation, and fostering a better interaction between the software and the framework it relies on.**

I. Introduction

Contemporary large software systems are not developed in isolation. Instead, they rely on complex software ecosystems for managing many essential but infrastructural tasks. This is testified by the mainstream success of many off-the-shelf frameworks that address the most typical concerns in software development: dependency management, data representation and persistence, deployment, integration, coordination, parallelism, etc. The complexity of these frameworks is reflected on the many requirements they impose on the structure and behavior of the software systems that are developed on them: The more pervasive ("crosscutting") are the concerns a framework addresses, the more these requirements are. A typical framework takes the responsibility of managing the lifecycle of software objects, their location, their persistence, the activation of tasks and activities and their communication, subtracting all or most of these responsibilities from the control of the software developer. For instance most frameworks are based on the inversion of control paradigm, that assigns to the framework the decision on when some user-defined methods or functions are invoked. In other words, frameworks separate the aspects of *coordination*, that is responsibility of the framework, from *computation*, ideally the only concern of software development.

Notwithstanding this situation, software is still developed with the programming languages designed when the development of monolithic, library-based software systems was the norm. Current programming languages assume that the developer will decide how the software is structured at all levels of granularity, when the resources are allocated and freed, how the control flow is structured, when a task must be activated and when it will terminate, how different task communicate and coordinate their operations, and so on. Our opinion is that these languages, although empowering, are not a good match to the above outlined scenario.

A. Motivation

To motivate this stance we will describe our experience with designing and developing software for the MARPLE framework. MARPLE (Metrics and Architecture Reconstruction PLugin for Eclipse) [1] is an Eclipse plugin that provides many functionalities supporting the comprehension and evolution of large codebases. MARPLE strongly relies on the functionalities provided by the Eclipse ecosystem to perform its many analysis tasks, as opposed to being a wrapper allowing Eclipse to interoperate with an external tool. MARPLE is itself an open platform designed to facilitate the development and integrate a heterogeneous set of static analysis algorithms. Some analyses MARPLE supports are design pattern detection through data mining, detection of bad code structures as antipatterns [2] and code smells [3], [4], and software architecture reconstruction, a functionality useful for program comprehension and reverse engineering. Antipattern detection and software architecture reconstruction facilities of MARPLE are described in [5].

The work presented in this paper stems from our experience matured while developing the MARPLE algorithms for code smell detection. Code smells are structural features of code which indicate potential design issues that make software hard to evolve and maintain. Some examples of code smells are Long Method (a method is too long, thus it should be broken in smaller submethods), Speculative Generality (some features of the software are never used, thus it might be desirable to remove them and reduce the complexity of the software) and Intensive Coupling (two classes are strongly dependent one on the other, possibly hinting bad code modularization). Many static analysis tools are available, both in the literature and in the wild, whose purpose is to facilitate the detection of code smells. We cite, as examples, Checkstyle[1],

[1] http://checkstyle.sourceforge.net/

978-1-4673-7037-0/15 $31.00 © 2015 IEEE
DOI 10.1109/CSD.2015.11

```
1: Graph<MethodDeclaration,MethodCall> g ← ∅
2: Set<MethodDeclaration> roots ← ∅
3: for all Class cl in currentProject do
4:   for all MethodDeclaration m in cl do
5:     if m.signature =
       public static void main(String[])
       then
6:       roots.add(m)
7:     end if
8:     for all MethodCall mc in m do
9:       Class target ←
           mc.invokeVar.classDeclaration
10:      MethodDeclaration m' ←
           target.findImpl(mc.signature)
11:      g.add(m → m')
12:    end for
13:  end for
14: end for
15: return g.unreachableFrom(roots)
```

Fig. 1. Speculative Generality Detection Algorithm

PMD[2], JDeodorant[3], and inCode[4]. Our experience in adding code smell detection algorithms to MARPLE highlighted that the interoperation with the platform (Eclipse and MARPLE) makes the link between a smell detection algorithm and its implementation obscure, and, at worst, augment the chance of introducing bugs in code.

Figure 1 reports, as an example, a simplified version of the Speculative Generality code smell detection algorithm implemented in MARPLE. As defined by Fowler and Beck [3], Speculative Generality affects the regions of the code that were included to anticipate future developments, possible changes and for the preparation of additional features of the system. The algorithm builds a call graph g for all the methods in an Eclipse project, and puts in $root$ all the `main` methods, that will be considered the roots of the graph. The algorithm returns all methods that cannot be reached from some root as potentially affected by the Speculative Generality smell. This simplified version of the algorithm assumes (line 9) that a variable will be always assigned an object of its static type, or no object if its static type is not a concrete class. This overly conservative assumptions serves the purpose of making the presentation simpler. The actual algorithm implemented in MARPLE differs from the one reported in Figure 1 only for its more sophisticated type analysis.

Some comments arise. Code smell detectors are implemented as extensions of the *Micro-Structures Detector* (MSD) MARPLE module. Micro-structures [6] can be informally defined as facts or binary relations about code elements that can be consistently detected by static analysis. A simple example of micro-structure is whether a class is declared abstract. Micro-structures are extracted by defining custom operation on a metamodel of the Java codebase under analysis. MARPLE exploits the Eclipse Java Development Tools (JDT) to extract ASTs from Java code and build the metamodel [7], while user-defined extractors are programmed based on the Visitor design pattern [8], where the actions to be performed on each metamodel element relevant for the analysis are defined as methods encapsulated in a same object—the visitor defining the analysis. The MSD module creates and runs each visitor only on the AST nodes that may contain the information they are able to detect. This has two consequences. First, the developer is not allowed to enforce a visit order: The visit order is automatically determined by the MSD layer based on the structure of the visitor itself. Second, the lifecycle of the visitors is entirely managed by the MSD layer, that is free to create and destroy them in a non-prescribed order.

Having to abstract from visit order and from lifecycle of visitor code poses some requirement on software development. Were it possible to enforce a visit order from the root methods along method calls, the Speculative Generality analysis could be done on-the-fly. Instead the programmer must persist the graph g in an independent store. Moreover, the developer cannot rely on the visit order to produce intermediate data that the algorithm depend on: The only form of dependency allowed by MARPLE is defining a sequential order of invocation for *distinct* types of visitors. If intermediate data is necessary to an algorithm, the developer must define another visitor to produce (and persist) the necessary data before the consumer is run. This complex protocol complicates development and potentially hinders performance.

If the software architect break the policies of the ecosystem, either because "cheating" or for mistake, the consequences are potentially dire. Back to our case study, the actual policies of MARPLE on visit order and data persistence match its data management strategy, that may, at a given moment, decide which parts of the metamodel reside in main memory and which are backed up on secondary storage. These strategies are tuned for optimizing performance when MARPLE must handle very large amounts of project data, and are likely to change across versions of the platform. This is the reason why the framework's API does not expose them. If a developer violates the MARPLE protocol, e.g. if she manually extract some data for recursively invoking a visitor method from another one—as it is customary in most incarnations of the Visitor pattern—the resulting MARPLE plugin might work correctly with the current version of the framework and unexpectedly fail with the next ones.

Current general-purpose programming languages are in our opinion at a too low level of abstraction to easily enforce the sophisticated architectural constraints embedded in a framework's design. A language with a lower conceptual gap would facilitate correct software development by preventing trivial errors as the one previously exemplified. But such a language would bring an additional, possibly subtler advantage: A higher degree of abstraction *improves* the ability of reasoning on the features of the software like correctness

[2]http://pmd.sourceforge.net/
[3]http://www.jdeodorant.com/
[4]http://www.intooitus.com/inCode.html

```
1:  agent PRJ:
2:    context: Project
3:    features:
4:      Graph<MethodDeclaration,MethodCall> g,
5:      Set<MethodDeclaration> roots
6:    enter:
7:      g ← ∅
8:      roots ← ∅
9:    exit: return g.unreachableFrom(roots)
10: agent MD:
11:   context: PRJ p :: MethodDeclaration m
12:   enter:
13:     if m.signature = ...main(String[]) then
14:       p.roots.add(m)
15:     end if
16: agent MC:
17:   context: PRJ p :: MD md :: MethodCall mc
18:   enter:
19:     Class target ← mc.invokeVar.classDeclaration
20:     MethodDeclaration m' ←
            target.findImpl(mc.signature)
21:     p.g.add(m → m')
```

Fig. 2. Speculative Generality, Context-Aware Style

and performance, by exposing and untangling the relevant concept that in a tradition language would be scattered and tangled with irrelevant implementation details. This clarifies the interaction patterns between a software and its ecosystem, possibly suggesting ways to improve both.

This paper aims at demonstrating the above ideas by introducing a novel method for designing software based on a *context-aware* style. According to it, the designer explicitly declares the possible contexts where a task operates and the data that task is expecting when active in such contexts. It is up to the runtime support to provide the remaining glue, i.e., to explore the data space, to activate each action in the right context, and to persist the data across context change. We will finally explain why, in our opinion, this method may potentially foster a better design, enable better reasoning about correctness, and lower the burden of correctly implementing the designed functionality.

B. Structure of the Paper

The paper is organized as follows. In Section II we describe a context-aware specification style and apply it to our case study. In Section III we exemplify the kind of reasoning on software properties this style fosters. In Section IV we correlate our contribution with current literature. Finally, Section V outlines some ideas for future research.

II. Context-Aware Specification of Algorithms

Figure 2 reports the same Speculative Generality detection procedure of Figure 1, expressed in our context-aware style. The specification is broken in three parts. Each part is the definition of an *agent type*, i.e., a set of operations to be performed in all the contexts with a given shape. A *context* is the specification of a chain of objects in the Java code metamodel, in a containment relationship. The meant semantics is that, in all the contexts associated to an agent type, the runtime support creates and executes an instance of the agent. The context specification of an agent type definition is indicated by a **context** clause. As an example, the agent type PRJ has, as associated contexts, all Project metamodel elements, each signifying a different Java project managed by Eclipse (and MARPLE). A **context** clause is a regular expression over *atomic contexts*, i.e., metamodel class names or agent type names. Every atomic context in a **context** clause can be associated to a variable name for reference by the agent specification. As an example, the MD agent type must be activated in all the contexts where a Project metamodel object contains a MethodDeclaration object md, and an agent p with type PRJ is active on the Project object[5]. An optional **filter** clause can further refine the context specification by an arbitrary predicate. Only one agent of a given type is active in a same context, but multiple agents with different types can be active in the same context. In this case, no order of execution is guaranteed.

Each agent expects a set of context *features*. A feature is a variable associated to a context and available to all the agents in it and in its subcontexts. Features are declared by a **features** clause in an agent type specification. As an example, every PRJ agent expects in every Project context a graph g to be available, having as nodes MethodDeclaration metamodel objects, and edges labelled by MethodCall metamodel objects (line 4). Note that distinct Project metamodel object are distinct *instances* of the context expected by PRJ, and as such they have *distinct* graphs g. This reflects the fact that every Java project has its call graph. A MD agent can access the graph g of the PRJ agent in its context, that surely exists as its defining **context** requires the existence of an active PRJ agent in all the contexts where a MD agent is active (line 11). Features are only available internally to agents in subcontexts, and cannot be accessed by agents in sibling contexts, or in supercontexts. This is the only way different agent types can share data. The only way an agent can provide data externally is to terminate and return a global result by a **return** statement (e.g., line 9).

Agent behaviors are ordinary blocks of Java statements. These blocks can access the features of the agent, and the features of all the agent types declared in its **context**, via the variable name (if present) associated to the corresponding elementary context specification. As an example, the MC agent accesses the feature g of the PRJ agent p in its context, by indicating it as $p.g$. An agent type definition may contain up to two different behavior specifications: what must be done when the agent *enters* the context (clause **enter**), and what must be done upon *exit* of the agent from the context (clause **exit**).

[5]We use the symbol :: to indicate containment. Note that, in the simplified Java metamodel assumed in this paper, the containment relation between two metamodel elements either does not exist, or exists unambiguous. Were some ambiguity, we should also qualify which containment relation we are referring to.

```
1: agent CLS:
2:    context: Class c ...
3: agent MD′:
4:    context: CLS cls :: MethodDeclaration m
5:    features: int maxNesting, cint, callsDispersed ...
6: agent CS:
7:    context: (MD′ md | CS cs) :: CompositeStatement s
8:    features: int nestingLevel
9:    enter:
10:      match context MD′ md :: CompositeStatement s
11:        nestingLevel ← 0
12:      match context CS cs :: CompositeStatement s
13:        nestingLevel ← cs.nestingLevel + 1
14:      end match
15:      md.maxNesting ←
             max(md.maxNesting, nestingLevel)
16: agent MC1:
17:   context: MD′ md :: (CS)* :: MethodCall mc
18:   enter: md.cint ← md.cint + 1
19: agent MC2:
20:   context: CLS cls :: MD′ :: (CS)* :: MethodCall mc
21:   filter:
        mc.invokeVar.classDeclaration ≠ cls.c
22:   enter:
23:      md.callsDispersed ← md.callsDispersed + 1
```

Fig. 3. Intensive Coupling, Context-Aware Style (Excerpt)

More precisely, **enter** blocks are executed *before* the (**enter** and **exit**) blocks of all the agents active in its subcontexts, while **exit** blocks are executed *after* activation of all the agents in subcontexts. Since agents communicate only via shared context features, **enter** blocks serve the purpose of preparing data to be read by subcontext agents, while **exit** blocks are dually used for using data produced by subcontext agents. As an example, a PRJ agent initializes on entry the graph g with an empty set of nodes and edges (line 7). Then, MC agents active in all MethodCall subcontexts of the PRJ agent populate g. Finally, the PRJ agent reads the complete g graph and returns it as a result (line 9).

This specification style achieves both kinds of abstractions we outlined previously. The **context** clauses abstract from visit order as they only declare "where" an agent expects to be active. The **features** declarations handle declaratively the lifecycle of data necessary to a computation, by associating them to metamodel elements. As an example, the call graph is a feature of a Project relevant only to the computation done by the PRJ agent (and by the MD and MC agents that depend on it). It is up to the framework to decide how to make available the data to the active agents that need them. This specification style also has a nontrivial expressive power, evident by considering the example in Figure 3. This is an excerpt from the Intensive Coupling [4] code smell detection algorithm. The algorithm calculates, for each nonabstract method, the maximum nesting level of CompositeStatements in its body ($maxNesting$), the total number of method calls ($cint$), and the fraction of method calls to methods declared in a different class ($callsDispersed$). The CS agent type calculates $maxNesting$, and has a recursive **context** declaration (line 7), which specifies that a CS agent can be either in a MethodDeclaration context with agent MD′, or in a CompositeStatement context with another CS agent (the vertical bar character in the **context** clause of CS means alternative). In the former case (line 11), the nesting level of the statement is set to 0. In the latter (line 13), it is set to the nesting level of the containing statement plus one. Finally (line 15), the CS agent updates the maximum nesting level feature for the MethodDeclaration, as exported by its contextual MD′ agent. This computation pattern would be awkwardly expressed in the unstructured style of Figure 1.

III. Reasoning on Context-Aware Specifications

In this section we briefly show how, by untangling the dependencies between information providers and consumers, the proposed context-aware style allows to reason more easily on the features of the algorithm. It also has benefits for framework designers, in that it facilitates the detection and application of optimizations. As examples:

- The language is sufficiently flexible to define complex coordination patterns between tasks encapsulated in agents, yet sufficiently abstract to relieve from the burden of explicitly programming them. For example, it is not necessary (nor possible) to explicitly invoke an agent to perform its task. The violation of the MARPLE protocol that is produced when invoking a visitor method from another visitor method here is simply not possible.
- A feature should not be used until its value is ready. In general, whenever an agent reads one of its own features in an **enter** block, and some subcontext agent writes it, we may expect a nondeterminate result, as there is no guarantee that subcontext agent will be executed in some order. Thus, either its declaration or its use is misplaced. A simple static analysis of a context-aware specification can catch this issue. On the contrary, it is quite easy to misplace a variable declaration in the style of Figure 1 and obtain a compiling, but incorrect, algorithm (e.g., by declaring the variable $roots$ after line 3).
- If an agent's feature is not used by any other subcontext agent, then the framework can transform it in a local variable, or simply drop it.
- If two agents for a same context write distinct features, the framework can execute them in parallel or combine them, e.g., agents with type MC1 and MC2 in Figure 3.
- By knowing which contexts are relevant for a computation, the framework can more easily optimize the visiting order and reduce the amount of data in the main memory.

IV. Related Work

The approach proposed in this paper has analogies with attribute grammars, with constructs in aspect- and context-

oriented programming languages, and with some paradigms for modeling context-aware systems and computations.

Attribute grammars [9] are a formalism for describing the semantics of programming languages. Similarly to the approach proposed in this paper, attribute grammars associate data, called attributes, to metamodel elements and express computations by functional relationships between attributes. The main differences are that attribute grammars allow inherited (top-down) attribute dependencies, and that are context-free, i.e., attributes are associated to single nonterminals, where features are associated to contexts, and dependencies can be defined only between the attributes of nonterminals in a same syntax rule. Our proposal has some analogies with *elementary* attribute grammars, in that it allows only synthesized (bottom-up) attributes, but it additionally breaks the context freedom requirement. Since elementary attribute grammars are equivalent in expressive power to general attribute grammars, we conjecture our approach has similar expressive power, but we have no formal proof of this conjecture.

Aspect-oriented languages define constructs for capturing all the join points in the dynamic call stack of a method, e.g., AspectJ [10] `cflow`. This way, one can add context-aware behaviors to method calls, where the context is given by the stack of callers. Our approach is complementary in that it defines contexts based on the data necessary to computation, and abstracts control flow based on them. Recently proposed context-oriented programming languages [11] dispatch behaviors based on an arbitrary programmable definition of context, but require the programmer to manually change context when necessary.

Context-aware systems [12] dynamically change their behavior to match changes in computational resource availability, or in any feature of its environment or user profile. This paper develops some ideas of [13], where a framework is outlined for specifying context-aware systems based on decoupling definition of behaviors from definition of the contexts that enable and disable them. The main differences are that here we do not disable behaviors based on context, we explicit specify the relationship between data and context, and we require to synchronize behaviors with context enter and exit events. More generally, the approach presented in this paper follows the idea of separating (the specification of) *computations* from (that of) *coordination*, an idea proposed and developed by the Coordination Languages community [14].

V. Conclusions and Future Developments

This paper motivates and describes an innovative software specification style that describes computations as a set of tasks, the agents, plus a declarative specification of how they should be activated based on the context of the computation. We applied the specification style to a real-world example and discussed why, in our opinion, it potentially fits the development of software based on complex software ecosystems.

Work is in progress for refining the language and achieve a better separation between data and algorithms and add constructs for modularization and reuse of agent specifications. We are also implementing an automatic compiler to Java code (a prototype partial implementation based on annotations already exists) and static checkers. Finally, we need to validate our hypotheses about the usability, ease of reasoning and performance of the specification language by applying it to a set of case studies.

References

[1] F. Arcelli Fontana and M. Zanoni, "A tool for design pattern detection and software architecture reconstruction," *Information Sciences*, vol. 181, no. 7, pp. 1306–1324, April 2011.

[2] W. H. Brown, R. C. Malveau, H. W. I. McCormick, and T. J. Mowbray, *Antipatterns: Refactoring Software, Architectures and Projects in Crisis*. John Wiley & Sons, 1998.

[3] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co. Inc., 1999, http://www.refactoring.com/.

[4] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[5] F. Arcelli Fontana and S. Maggioni, "Metrics and antipatterns for software quality evaluation," in *Proc. 34th IEEE Software Engineering Workshop (SEW 2011)*. Limerick, Ireland: IEEE, Jun. 2011, pp. 48–56.

[6] F. Arcelli Fontana, S. Maggioni, and C. Raibulet, "Design patterns: A survey on their micro-structures," *Journal of Software Maintenance and Evolution: Research and Practice*, 2011.

[7] F. Arcelli, M. Zanoni, R. Porrini, and M. Vivanti, "A model proposal for program comprehension," in *Proceedings of the 16th International Conference on Distributed Multimedia Systems*, ser. DMS 2010: Globalization and Personalization. Oak Brook, Illinois, USA: Knowledge Systems Institute, October 2010, pp. 23–28. [Online]. Available: http://www.ksi.edu/seke/Proceedings/dms/DMS2010_Proceedings.pdf

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[9] D. E. Knuth, "Semantics of context-free languages," *Mathematical Systems Theory*, vol. 2, pp. 127–145, 1968.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *ECOOP*, 2001, pp. 327–353.

[11] R. Hirschfeld, P. Costanza, and O. Nierstrasz, "Context-oriented programming," *Journal of Object Technology*, vol. 7, no. 3, pp. 125–151, Mar. 2008.

[12] B. Schilit, N. Adams, and R. Want, "Context-aware computing applications," in *First Workshop on Mobile Computing Systems and Applications (WMCSA)*, Dec. 1994, pp. 85–90.

[13] P. Braione and G. P. Picco, "On calculi for context-aware coordination," in *COORDINATION 2004*, ser. Lecture Notes in Computer Science, vol. 2949. Springer, 2004, pp. 38–54.

[14] D. Gelernter and N. Carriero, "Coordination languages and their significance," *CACM*, vol. 35, no. 2, pp. 97–107, Feb. 1992. [Online]. Available: http://doi.acm.org/10.1145/129630.129635