Explicit Support for Software Development Styles throughout the Complete Life Cycle

Roel Wuyts[*], Kim Mens[†] and Theo D'Hondt
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium
e-mail: { rwuyts | kimmens | tjdhondt } @vub.ac.be

April 1st, 1999

Abstract

Throughout its entire life cycle software development is subject to many rules constraining and guiding construction of software systems. Examples are *best-practice patterns*, *idioms*, *coding conventions*, *design guidelines*, *architectural patterns*, etc. Although such regulations are widely used, their usage is currently implicit or ad-hoc, and most software development environments do not explicitly support them. We present an approach to declare explicitly software development styles in an open declarative system that allows querying, conformance checking and enforcement of these declarations on the source code. We validate the approach by expressing and supporting several software development styles in a real-world case.

1 Introduction

The software development process is strongly constrained by all kinds of implementation, design and architectural guidelines. Although the widespread

[*]Research conducted on a doctoral grant from the "Instituut ter bevordering van het Wetenschappelijk en Technologisch onderzoek in de Industrie" (Flanders, Belgium).

[†]Research funded by the Brussels Capital Region (Belgium).

usage of software development styles throughout the life-cycle, general support is almost nonexistent in current day development environments. The few systems that allow the usage of such rules (like [Min96, MP97, Bok99, MDR93]) target one phase only in the development cycle (implementation, design or architectural), as we will see in the next section. While these systems are an important first step, we claim that general support is needed for

- declaring software development styles explicitly in all phases of the life-cycle and on all levels of abstraction in one and the same formalism.
- supporting these software development styles to allow enforcing and compliance checking to the source code.

We propose a declarative formalism to express all software development styles, thereby making them explicit in a medium that allows us to extract source code, to do conformance checking and enforcement. To validate our formalism we use a real-world case study where we express many different software development styles, on different levels of abstraction, and explain their usage in practical development.

The remainder of this paper is structured as follows: next section discusses software development styles in more detail, while the following section will describe examples of expressing and using software development styles throughout development. Then we discuss the results of these experiments. Finally we end with related and future work, and a conclusion.

2 Software Development Styles

We define a *software development style* as a set of rules that constrain the structure a software system according to certain requirements (such as efficiency, extensibility, ...). Literature provides different examples of software development styles, but mostly in specific phases of the life-cycle:

1. *implementation styles* are widely encountered in different programming languages and systems. They include style guides that describe

what a 'well-styled' program in a language looks like, for example *best-practice patterns* [Bec97], general *guidelines for Smalltalk* [Lew95], or *idioms* [Cop98]. Although a lot of these rules are fairly simple to enforce or even check, almost no development environments do so. Exceptions are for example [MDR93] or [Bok99].

2. *design styles*: the best known example in this phase is probably *design patterns* [GHJV94, Pre94]. Support for design patterns, let alone general design style support is virtually nonexistent. As exceptions we note [FMvW97, Bos97]. Another example of a design guideline is *contracts* [HHG90].

3. *architectural styles*: work has been done in the architectural community in constructing *architectural description languages*, *architectural constraint languages* and *architectural patterns* [SG96, SSWA96, Bal96, BMR+96]. The major difference with our approach is that they focus on architecture alone, while we propose a unified formalism that can be used for all kinds of software development styles. Another important example is *law governed architectures (LGA)* [Min96]; we will discuss this in more detail in section 6.

Both this related work and the experiences of our industrial partners motivate the need and demand for expressing and supporting development styles at all levels of the software life-cycle. We were also struck by the commonalities in currently available approaches. Therefore, it would be beneficial to develop a *unified approach* in which all these styles can be expressed. With such an approach, developers only need to use one approach, minimizing the overhead of learning to work with it. Furthermore, all developed support tools are immediately available at many levels of the life cycle. Finally, declaring all styles in the same uniform formalism makes it very easy to define them in terms of more low-level ones. This is a powerful abstraction mechanism typically not provided by most existing approaches.

Our experiments indicate that a *declarative approach* would be very well suited to support software development styles. Another advantage of a declarative approach is that it expresses formal relationships between arguments. This means for example that one and the same relation between two classes can be used in 4 ways: to check whether this relation holds between both, to enumerate all classes if one of the two is given, or to enumerate all couples of classes satisfying the relation. A non-declarative approach would require different functions.

We also want our approach to be as *open* as possible. It should be possible at all times to introduce new kinds of styles, to reason about new kinds of software artifacts, and so on.

Before the next section introduces the declarative medium we used for our experiments, we give two informal examples of software development styles that we will describe explicitly later on in the experiments section. The first is application specific, while the other is an example of a naming convention:

1. A bridge pattern will be used as the core design element in the persistency layer. Every class that needs to be persistent should conform to this pattern.
2. Methods returning booleans should be prefixed with 'is'.

3 Smalltalk Open Unification Language

In the experiments for this paper we have used SOUL (Smalltalk Open Unification Language) [Wuy98] to declare software development styles, and tools based on SOUL for querying source code, for doing conformance checking and for enforcement of styles. SOUL is PROLOG-like, but has an extension that allows unification on user-defined elements. This lets logical clauses directly reference elements from the base language (such as classes, inheritance relations, method bodies, ...), avoiding the overhead of constructing and maintaining a separate repository containing the source code. Another feature are the second-order predicates that allow easy manipulation of lists. While SOUL itself is implemented in Smalltalk, it reasons about programs of general object-oriented programs by representing the programs using a general OO parse tree format. Thus logic reasoning is possible using fine-grained structural information (such as classes, methods, instance variables, message sends, parameter passing, ...).

A *declarative framework* of rules that allows reasoning at the implementation, design and architectural level was implemented in SOUL [Wuy98]. It can be seen as a layered structure, starting from a base-language dependent layer that maps to the basics of an object-oriented language. On top of this layer sits the *basic structural* layer that adds basic predicates directly related to the source code. Then higher level layers are available that describes higher level relationships, such as design patterns. Recently yet another layer was put on top of this that allows reasoning at an architectural level [MW99]. Using the declarative framework implemented in SOUL gives us a powerful medium to reason about structural information in an object-oriented language, while still providing the capability to check such structures against the source code.

4 Using explicit Software Development Styles

In this section we give a concrete feeling on how to express concrete software development styles. We will do this using a real-world case, where we can express a number of software development styles that arose during development, and indicate how to use these styles in the development process. We will assume an incremental development process where we will focus on two cycles:

- a first cycle where the core of the framework is built.
- a second cycle that extends this core towards a full fledged application.

The reason to structure the examples in two cycles is that usage of the software development styles will normally be different. In the first phase a number of software development will typically be introduced, but not directly strongly enforced. Progressively, as the framework matures, conformance to the existing guidelines becomes more and more important. Once

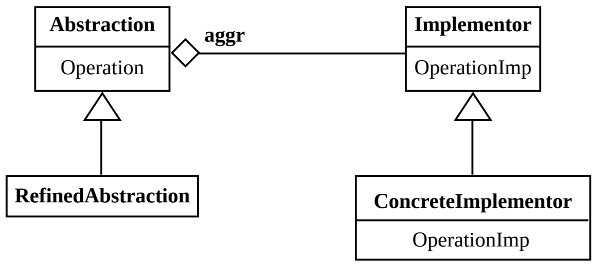


Figure 1: Bridge Pattern Structure

in the maintenance phase, violations of the software development styles are not allowed any longer. We will now start by introducing the case, and afterwards the examples in the two cycles.

4.1 Case: The Persistency Layer

The case we use is the *persistency layer*[1], part of a planning application for broadcasting companies [CHSV97, VV96]. It allows the storage of objects in Smalltalk in a relational database, transparently supports different relational databases that can be changed at runtime, and supports different mappings from objects on relational tables. The persistency layer was developed in house because persistency layers with these features were not available at the time.

The core of the persistency layer is a *bridge architecture* (see picture 1) that permits different useful mapping strategies to map objects on rows and tables in a relational database. The bridge is responsible for mapping the high-level domain objects to the lower-level entity objects that will be stored in the database. The bridge "knows" how to map structures of the domain model to a set of entity classes that respect the constraints of the database layer. The persistency layer gives a lot of freedom in choosing a mapping strategy. Strategies can be reused, enhanced or new strategies can be developed. The same holds for the storage classes themselves. Performance can

[1]Provided by one of our industrial partners, MediaGeniX

be gained by fine-tuning a storage class or creating a very specific one (that could even include hard coded SQL-statements if needed), but this will never have an impact on the conceptual object model. Note also that the same conceptual object class can have multiple storage classes that implement it. This property of the bridge architecture is used to adapt applications to different relational database systems and to integrate applications that use the persistency layer with existing, customer specific tables.

The most important reason for choosing this particular case above others is that it is sufficiently elaborate and general to demonstrate a number of different software development styles, yet fairly easy to explain. It is also part of an application framework that is instantiated and customized for different users, yet it is not too domain specific (the problem of persistency plays an important role in many different domains).

4.2 Cycle 1: Core Framework Development

In this section we assume that the requirements and analysis phase are over and that the initial design of the persistency layer is done. Implementation of the core framework then starts. Some software development styles can be known up-front (for example by experience from other applications) and can be expressed and used directly. Other styles will only become clear during development. We will give two examples: the bridge pattern that forms the core of the persistency layer, and a naming convention rule. These are actually the examples that were given informally in section 2.

4.2.1 The Bridge Pattern

From the analysis it was clear that the core of the persistency layer should be a bridge architecture to decouple the domain and the storage classes. It was also clear that persistent classes need to adhere to this structure at all times. In the original development this could not be expressed, let alone enforced. We will describe and use the software development style expressing this core architectural knowledge of the persistency layer.

We first have to express the bridge pattern [GHJV94] that relates the domain classes and the storage classes. We will do this with a logical rule that describes the four participants of the bridge pattern and their relations (see also figure 1): the *abstraction class*, the *implementor class*, the *refined abstraction* (subclass of the abstraction class) and the *concrete implementor* (subclass of the implementor class). The most important relation is the aggregation relation *?aggr* between the abstraction class *?abs* and the implementor class *?impl*. This aggregation relation is used by the methods on the abstraction classes *?absMethod* to call methods *?impSelector* of the concrete implementation classes. Following rule expresses the bridge pattern relationship between the four participants[2].

```
Rule bridge(?abs, ?refinedAbs, ?impl, ?concreteImpl) if
        hierarchy(?abs, ?refinedAbs),
        hierarchy(?impl, ?concreteImpl),
        aggregationRelation(?abstraction, ?implementor, ?aggr),
        method(?abstraction, ?absMethod),
        classImplements(?implementor, ?impSelector),
        implementedInTermsOf(?absMethod, ?aggr, ?impSelector)
```

Now that we have expressed the software development style describing the bridge pattern relation, we can use SOUL and the SOUL tools for querying or conformance checking. For example, we can check in the source code whether the persistent class *WONPsiUser*[3], conforms to our style expressing the bridge pattern using the following query:

```
Query bridge(?domain, [WONPsiUser], ?storage, ?concrete)
```

[2]Some notes on SOUL syntax:

1. the keywords **Rule** , **Fact** and **Query** denotes logical rules facts and queries
2. variables start with a question mark
3. terms between square brackets contain Smalltalk code, which can be constants, such as strings or symbols, but also complete Smalltalk expressions that reference logic variables from the outer scope.
4. <> is the list notation

[3]This is the actual name of a class used in the persistency layer which explains it somewhat strange name

The result of this query is a number of bindings for the different variables. In this case it will answer that *?domain* will be *WONDomain*, *?storage* will be *WONStorage* and *?concrete* will be *PSIUser*. This means that *WONPsiUser* and *PsiUser* are the domain and storage classes respectively, which was correct according to the developers. Note the usage of the multiway property of declarative languages as introduced in section 2. The query shows how the rule that describes the bridge relation can be used in different ways, just by filling in what is known. The rest will be computed using the relations expressed. This makes it easy for example to bind every variable with concrete classes and check if they conform to the software development style. It makes it easy also to use SOUL as reasoning engine in specialized tools, for example in a 'Bridge Browser' that browses source code as defined by this style.

4.2.2 Prefix Boolean Methods with 'is'

As second example we will express a well known implementation guideline that methods that return booleans should have a name starting with *is*. We can easily write a rule to express this naming convention for a method *?method*. We see if *method* returns a boolean[4]; if so, we check its name *?name* to ensure it starts with *is*:

```
Rule prefixedWithIs(?method) if
        resultType(?method, [Boolean]),
        methodName(?method, ?name),
        ['is*' match: ?name]
```

Again, now that we have expressed the style we can use it for querying, conformance checking or enforcement. Even in this stage of development this rule will typically be strongly enforced. However, if we want to check the source code for violations of the style, we could use a query that checks for each class *?class* in the persistency layer whether its methods *?method* conform to the rule describing the style:

[4]Note that in Smalltalk this requires a type analysis, while in a statically typed language this can be checked easily

Query persistencyClass(?class),
method(?class, ?method),
prefixedWithIs(?method)

4.3 Cycle 2: Extending and Refining the Core Framework

The next phase in the development process is where the core framework is largely completed and stable, and where it is then incrementally refined and extended. In this stage a lot of software development styles are needed to guide this explosion of code and make sure that there is not too much architectural drift, compromising the initial clean structure of the framework. As the framework becomes more stable, the rules will typically be enforced stronger. We will describe some example styles, namely that instance variables can only be accessed using accessor methods, that storage classes should provide default domain classes and vice versa, and that accessors of persistent classes use a very specific form to cross the bridge safely.

4.3.1 Always use Accessor Methods

Because of several reasons, such as hiding the interface from the implementation, there was a decision to use *accessor methods* consistently to access instance variables instead of manipulating them directly [Bec97]. While this is currently implicit, we could again make this software development style explicit and use it in tools so that violations could be tracked down or logged. As an example we will describe that a method *?method* should never sent messages directly to instance variables (so the receivers *?receiver* cannot be instance variables). Note that, in order to retrieve the instance variables of a class, we first ask the method *?method* for its class *?class*.

Rule usesNoInstVars(?method) **if**
methodClass(?method, ?class),
isSendTo(?receiver, ?message, ?method),
not(instVar(?class, ?receiver))

Once we have expressed this style, we can check if the code conforms to this software development style by checking whether for every persistent class *?classToCheck* every method *?methodToCheck* does not use instance variables directly:

```
Query  forall(  and(
                    persistentClass(?classToCheck),
                    method(?classToCheck, ?methodToCheck)),
                usesNoInstVars(?methodToCheck))
```

This query will return true if all methods of persistent classes conform to the style, or false as soon as there is one that does not. While this is already interesting to check, we can provide more feedback to detect where the violations are. Next query for example will actually find the classes and methods that violate the style:

```
Query  persistentClass(?classToCheck),
       method(?classToCheck, ?methodToCheck),
       not(usesNoInstVars(?methodToCheck))
```

4.3.2 Default Domain and Storage Classes

Another important decision was to provide default domain and storage classes using factory methods [GHJV94]. More specifically, with each domain class must have a factory method returning a default storage class and vice versa. We can write a query to check for each class in the persistency layer whether it conforms to this structure. This is indeed very important because the bridge relies on these methods to have a default mapping. For every class *?storage* in the storage hierarchy we check if it has a class factory method *?storMethod* that creates a domain class *?domain*. We then check if this associated domain class has a factory class method *?domMethod* that references back the storage class.

```
Query  forall(
           storageClass(?storage),
           and(
               classFactoryMethod(?storage, ?storMethod, ?domain),
               classFactoryMethod(?domain,?domMethod,?storage)))
```

Because the presence of a default mapping is important for the bridge pattern used in the persistency layer, we chose to refine the definition. We made a *refinedBridge* rule that not only specifies that the general bridge architecture should hold, but also that the factory methods for the default mapping should exist:

```
Rule refinedBridge(?abs, ?refinedAbs, ?impl, ?concreteImpl) if
         classFactoryMethod(?refinedAbs,?m1,?concreteImpl),
         classFactoryMethod(?concreteImpl,?m2,?refinedAbs),
         bridge(?abs,?refinedAbs,?impl,?concreteImpl)
```

Note that this rule illustrates that by having one unified approach we can easily describe styles that cross the boundaries of different levels of abstraction. It also shows how the abstraction mechanism of a rule-based language are used to express specific, case dependent software development styles in function of other styles.

4.3.3 Accessors to cross the Bridge

In a domain class that needs to be persistent, instance variables should not be used. The class should adhere to the bridge pattern instead, and has to provide accessor methods to store and retrieve its instance variables from the relational database. This means that the accessor methods have a specific form, because they are responsible for crossing the bridge. To be able to enforce the correct form of an accessor method in the persistency layer, we will first describe the form of a statement that uses the bridge to fetch the value of an instance variable from its storage class, and turn it into a domain object. This gives following sequence of message sends:

1. send the message *asStorage* to the receiver *self* (*this*) to get the storage class of the receiver;

2. send the message to retrieve the instance variable to that storage class (by convention this method's name is the same as the instance variable's);

3. send the message *asDomain* to get the domain class for the instance variable.

The following fact expresses this sequence, where the variable *?selector* is the message that will be sent to the storage class to fetch the value for the instance variable (note that it works directly on the representation of the parse tree):

```
Fact persistentAccessorStatement(
        return(
            send(
                send(
                    send(variable([#self]),[#asStorage],<>),
                    ?selector,
                    <>),
                [#asDomain],
                <>)),
            ?selector
            )
```

Now that we have a fact that describes the form of a statement to retrieve the value of an instance variable from the storage class, we can write a rule that specifies the structure of an accessor method. We express that an accessor method *?method* for an instance variable *?instVar* has the same name as *?instVar* (in both storage class and domain class), and that the body contains one statement *?statement* that complies to the form given in above fact.

```
Rule persistentAccessor(?method, ?instVar) if
        methodName(?method, ?instVar),
        methodStatements(?method, <?statement>),
        persistentAccessorStatement(?statement, ?instVar).
```

Again, once we have expressed a rule describing the software development style we can use this rule for querying, conformance checking or enforcement.

5 Discussion

In this section we recapitulate some important issues that were already addressed in the case. We will discuss the generality of the approach, the integration in a development environment, performance for different kinds of usage and the need of managerial and developer support when fully embracing software development styles.

5.1 General support throughout development

As motivated in section 2 there is need for an open, general declarative approach to support software development styles throughout development. We have illustrated this in the previous section by expressing expressing and using several software development styles on different levels of abstraction. For example, on implementation level we expressed the naming convention that boolean methods have to start with *is*. At design level we declared the default mapping between domain and storage classes, and that this mapping uses the factory method design pattern. At higher level design we expressed the bridge pattern, the core of the persistency architecture. They also indicated the need for a powerful, rule-based declarative approach. While not always visible at first, most queries actually require unification to be solved.

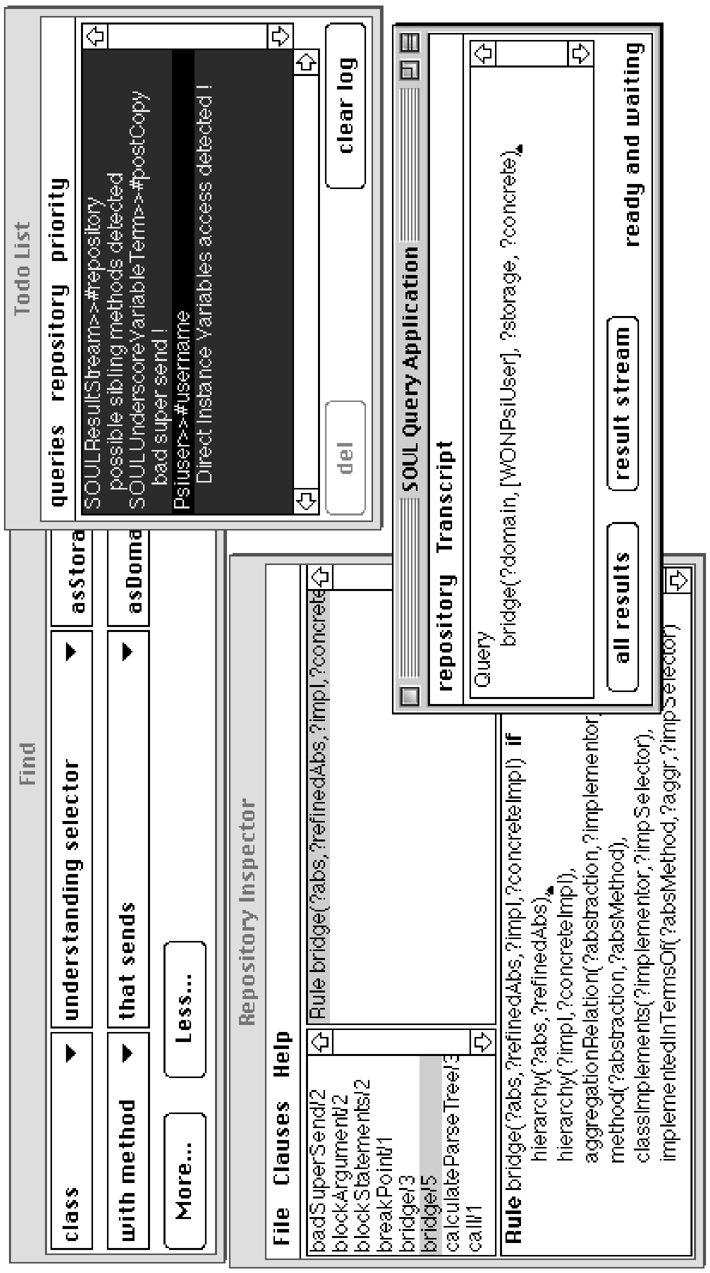
5.2 Integration with the Environment

One of the important aspects to practically use software development styles is the integration with the development environment. Based on our experience gained with the experiments we think an environment supporting software development styles should at least provide the following tools:

- tools to easily express software development styles. More specifically, the developer should not have to interact directly with the underlying reasoning language, but should be able to interact with software development styles using appropriate views.
- tools that take advantage of the knowledge described by the software development styles to browse the system on a high level of abstraction.
- tools for doing conformance checking of software development styles against the code. They should provide overviews what does and what does not conform to the styles.
- tools for enforcement (both strong and weak).

SOUL adheres to some, but not all of these requirements. We have implemented a tool that makes it easy to find source code that conforms to certain criteria and styles, and a tool that allows weak enforcement of styles by doing a conformance check every time a method is accepted in Smalltalk[5]. The violations are logged on a todo-list so development is not hindered while styles are enforced. While this non intrusive approach is necessary in some stages of development there should also be a strong enforcement tool that does not allow the compilation of non-conforming methods. We are experimenting with such tool in order to find a good balance between the kinds of styles we want to enforce strongly versus the performance penalty of performing the checks. Writing software development styles and doing conformance checks is currently done by manually writing SOUL code. Figure 2 shows a screenshot of the tools we used for our experiments.

[5]Smalltalk uses incremental compilation, so this is a natural integration with a Smalltalk development environment.



```
SOULResultStream>>#repository
    possible sibling methods detected
SOULUnderscoreVariableTerm>>#postCopy
    bad super send !
Psiuser>>#username
    Direct Instance Variables access detected !
```

```
badSuperSend/2
blockArgument/2
blockStatements/2
breakPoint/1
bridge/3
bridge/5
calculateParseTree/3
call/1
```

```
Rule bridge(?abs,?refinedAbs,?impl,?concrete
```

```
Rule bridge(?abs,?refinedAbs,?impl,?concreteImpl) if
    hierarchy(?abs,?refinedAbs),
    hierarchy(?impl,?concreteImpl),
    aggregationRelation(?abstraction,?implementor
    method(?abstraction,?absMethod),
    classImplements(?implementor,?impSelector),
    implementedInTermsOf(?absMethod,?aggr,?impSelector)
```

```
Query
    bridge(?domain,[WONPsiUser], ?storage, ?concrete).
```

Figure 2: SOUL Tools: the *Query Application* (for manually invoking queries), the *TODO application* that allows weak enforcement, the *Find Application* and the *Repository Inspector*

5.3 Performance

The need for performance differs differs depending on how software development styles are used. Conformance checking, for example, is typically a bulk operation that is run overnight and produces a report with the violations. Raw performance is then less of an issue. On the other hand, for querying and enforcement of software development styles the performance is important. We find that the performance of SOUL for this kind of usage is acceptable, meaning that it typically is an order of magnitude faster than performing the operation by hand. For example, the slowest query in this paper (doing a conformance check to make sure that domain and storage classes have default methods) takes only slightly more than a minute[6]. Manually checking this relation takes over 20 minutes. This does not mean however that *every* query is fast. The combination of the declarative nature of our approach with high-level predicates implies that solving some queries might take a very long time.

5.4 Managerial issues

A development methodology that fully embraces the usage of software development styles, special phases may be needed to declare and maintain such rules. Dedicated management and developer functions should probably be created in order to maintain the rules. This extra cost should pay of later, because development and maintenance are then guided by the software development styles. We plan to investigate further this issue.

6 Related Work

The main contribution of this paper is the introduction of a formalism that allows the expression of software development styles explicitly and supports querying, conformance checking and enforcement of these styles to the source code. In this section we will summarize some related approaches, and dis-

[6]This was measured for the persistency layer containing 143 classes and 431 methods on a G3-250 Mhz Macintosh and a Pentium-II 300 Mhz PC.

cuss similarities and differences. We have subdivided the approaches in two groups depending on whether or not they use a declarative language.

6.1 Non-declarative Languages

There are a number of approaches that propose non-declarative constraint languages. For example, Coffeestrainer [Bok99] is a Java preprocessor where constraints can be added that reason about an annotated parse tree. The constraints are expressed in Java. While there are arguments in favor of this approach (such as the developers not needing to learn a new language), we feel that there are more drawbacks than advantages, as mentioned in section 2.

Other approaches propose languages that are not as powerful as declarative languages. An example is Smalltalk Lint [RBJO96], a regular expression like language that allows to check for common programming mistakes in Smalltalk (such as its famous C predecessor). While it allows a certain level of reasoning about Smalltalk parse trees, we lack the abstraction facilities of a logic-like language that allows the expression of higher level abstractions. Yet another example is CDL [KKS96], the Constraint Description Language, targeted towards a very specific domain and only allowing fixed parse trees.

Another category of languages are those that propose formalisms to describe constraints or rules about the structure of programs, but that lack a computable medium to check them. OCL [OMG97], the Object Constraint Language used in UML 1.1 is a good example. These languages can be used as a notation, but because they cannot be executed they merely serve as structured comment.

6.2 Declarative Languages

CCEL [MDR93] is a constraint language for use with C++ that allows one to express programmer-defined constraints on the structure. The constraint language however is limited and does not allow to make abstractions. This means we are not able to construct constraints referencing or building on other constraints. We consider this a serious shortcoming, since this abstraction feature allows us to reason about ever higher abstractions in the same medium.

Another approach that closely resembles ours but is focussed on C++ only is ASTLOG [Cre97]. Like SOUL, it is a logic programming language that reasons directly about user-defined objects and provides second order abstraction capabilities. ASTLOG introduces a notion of *current object*. While this is not by itself an increase in expressivity, the authors claim that it allows a distinct style of logic programming suited very well to enumerating tree-like structures. However, for performance reasons, ASTLOG removes the ability to add or remove clauses dynamically, which in turn removes many higher-order logical features. In SOUL we have not made this restriction because we feel that it could limit the expressiveness.

Law Governed Architecture (LGA) [Min96, MP97] expresses and enforces regularities of software architectures using Prolog as reasoning engine. Not only an object-oriented language like Eiffel is supported, but also Prolog itself. Because the focus is explicitly on architectures and collaborations, only exchange of messages is regulated by the laws. This differs from our approach that allows styles to be expressed using the complete parse tree and abstractions thereof. An advantage of LGA however is its support for both static and dynamic enforcement, while our approach is currently limited to static reasoning over the source code. Dynamic reasoning is an important feature that we certainly want to look into.

7 Future Work

One of the interesting topics we are currently working on is the integration with other data sources, such as other base languages, case tools or architectural description languages. This would allow rules that can check not only the source code and abstractions thereof, but also other existing higher level abstractions.

We are also working to integrate other 'rule solvers' in our approach, such as simple but more efficient pattern matching, regular expressions and forward chaining algorithms. Such language extensions guarantee the power and expressiveness one expects from a logic programming language, combined with faster query solving techniques.

Another interesting topic is generation of source code. Amongst others, we would like to investigate how code generation could be used for transforming code. For example, when detecting that a domain class in the persistency layer forgot to implement a default mapping, one could be generated (semi-)automatically based on the software development style used.

8 Conclusion

This paper presents a first step towards explicit support for software development styles throughout the complete life cycle. It proposes not only a formalism to express software development styles, but also claims that these styles should be usable to query the source code, to do conformance checking against the source code and to enforce software development styles in the source code. These claims were validated on a real-world case using an experimental environment integrated in VisualWorks Smalltalk. We hope that this contribution will spark more research in general approaches integrated with development environments to express and support software development styles throughout the software life cycle.

9 Acknowledgments

We like to thank the software engineers at MediaGeniX, especially Koen De Hondt, Mark Plas and Wilfried Verachtert for the case and their patience explaining it. We also take this opportunity to thank our colleagues at the Programming Technology Lab for their support, discussions and proofreading: Wolfgang De Meuter, Kris De Volder, Carine Lucas, Tom Mens, Tom Tourwé and Bart Wouters.

References

[Bal96] R. Balzer. Enforcing architecture constraints. In *Second International Software Architecture Workshop (ISAW-2)*, October 1996.

[Bec97] K. Beck. *Smalltalk Best Practice Patterns.* Prentice Hall, 1997.

[BMR$^+$96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture — A System of Patterns.* John Wiley & Sons, 1996.

[Bok99] B. Bokowski. Coffeestrainer: Statically-checked constraints on the definition and use of types in java. Technical report, Freie Universität Berlin, 1999. submitted to ESSEC'99.

[Bos97] Jan Bosch. Specifying frameworks and design patterns as architectural fragments. Technical report, University of Karlskrona/Ronneby, 1997.

[CHSV97] W. Codenie, K. De Hondt, P. Steyaert, and A. Vercammen. Evolving custom-made applications into domain-specific frameworks. *Communications of the ACM*, 40:71–77, October 1997.

[Cop98] James O. Coplien. C++ idioms. In P. Dyson J. Coldewey, editor, *Proceedings of the Third European Conference on Pattern Languages of Programming*, 1998. To be published.

[Cre97] Roger F. Crew. Astlog: A language for examining abstract syntax trees. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, 1997.

[FMvW97] Gert Florijn, Marco Meijers, and Pieter van Winsen. Tool support for object-oriented patterns. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming, 11th European Conference*, volume 1241 of *Lecture*

Notes in Computer Science, pages 472–495, Jyväskylä, Finland, 9–13 June 1997. Springer.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addisson-Wesley, 1994.

[HHG90] R. Helm, I.M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *ECOOP/OOPSLA'90 Proceedings*. ACM Press, 1990.

[KKS96] N. Klarlund, J. Koistinen, and M. Schwartzbach. Formal design constraints. In *Proceedings OOPSLA '96, ACM SIGPLAN Notices*, pages 370–383, 1996.

[Lew95] S. Lewis. *The art and science of Smalltalk*. Hewlett-Packard professional books, 1995.

[MDR93] Scott Meyers, Carolyn K. Duby, and Steven P. Reiss. Constraining the structure and style of object-oriented programs. Technical Report CS-93-12, Department of Computer Science, Brown University, Box 1910, Providence, RI 02912, April 1993.

[Min96] Naftaly H. Minsky. Law-governed regularities in object systems, part 1: An abstract model. *Theory and Practice of Object Systems*, 2(4):283–301, 1996.

[MP97] Naftaly H. Minsky and Partha Pratim Pal. Law-governed regularities in object systems, part 2: A concrete implementation. *Theory and Practice of Object Systems*, 3(2):87–101, 1997.

[MW99] K. Mens and R. Wuyts. Declaratively codifying software architectures using virtual software classifications. In *Proceedings of TOOLS-Europe 99*, June 1999.

[OMG97] OMG document ad/97-08-08. Object Constraint Language Specifications, version 1.1, September 1997.

[Pre94] W. Pree. *Design Patterns for Object-Oriented Software Development.* Addisson-Wesley, 1994.

[RBJO96] D. Roberts, J. Brant, R. Johnson, and B. Opdyke. An automated refactoring tool. In *Proceedings of ICAST '96, Chicago, IL*, April 1996.

[SG96] M. Shaw and D. Garlan. *Software Architecture — Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[SSWA96] R. W. Schwanke, V. A. Strack, and T. Werthmann-Auzinger. Industrial software architecture with gestalt. In *Proceedings of IWSSD-8.* IEEE, 1996.

[VV96] A. Vercammen and W. Verachtert. Psi: From custom developed application to domain specific framework. In *Addendum to the proceedings of OOPSLA '96*, 1996.

[Wuy98] R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings TOOLS USA'98, IEEE Computer Society Press*, pages 112–124, 1998.