8 Conclusions

In this chapter, we summarize the contributions of this thesis (Section 8.1), discuss the issue of whether constraints are a sign of bad design (Section 8.2), and point out directions for further work (Section 8.3).

8.1 Summary

In this thesis, we have shown that – for its correct functioning – existing object-oriented software relies on application-specific constraints regarding the definition and use of program elements. We have presented a framework, called CoffeeStrainer, which allows to check such programmer-defined constraints for Java. CoffeeStrainer constraints are unique in that they are modular, extensible and composable, and special support is provided for constraints on the usage of program elements. Additionally, CoffeeStrainer constraints can consist of static (compile-time) and dynamic (run-time) parts. CoffeeStrainer has been fully implemented. It supports separate checking of compilation units, and its performance in terms of static checking time is comparable to running a compiler.

Unlike previous work, CoffeeStrainer takes a pragmatic approach and does not define a special-purpose constraint language. Instead, constraints are specified using Java, so that the programmer need not learn new syntax. Constraint code and base-level code share the same structure by embedding constraint code in Javadoc comments, making it easy to find the rules that apply to a given part of the program, and allowing arbitrary compilers and tools to be applied to the source code that contains constraints. When defining a new rule, the programmer has access to a complete abstract syntax tree of the program that is to be checked.

8.2 Constraints – a sign of bad design?

An interesting question is whether the existence of constraints might be a sign of "bad" software design in the sense that, if the design was changed to make better use of existing programming language features, the constraints would not be needed.

For example, in section 2.3.1, we noted that certain sequencing constraints in package `java.sql` could be removed if additional wrapper object types were introduced. One such sequencing constraint is mentioned in the documentation of the method `wasNull` in `java.sql.CallableStatement`: "Note that this method should be called only after calling the `get` method; otherwise, there is no value to use in determining whether it is `null` or not.” As already noted in section 2.3.1, it would be possible to wrap the result of the `get` method in a wrapper object which allows to retrieve the result and to query whether or not the result is the SQL `null` value. In this case, a constraint would not be necessary because the method `wasNull` now could be called on the wrapper object, which makes sure that the SQL result is retrieved before `wasNull` can be called.

In fact, this particular constraint is a sign of bad design. However, most other constraint examples that we have encountered could not be removed either because a language feature for expressing the constraint is missing from the particular programming language used, because non-functional requirements prevent using existing language features, or because the design-specific consistency requirements cannot be expressed using static types.

Missing language features

Some constraints are needed only because the particular language used does not provide a known language feature which would allow expressing the constraint directly in the language. For example, genericity constraints can be expressed directly in a language that supports parameterized types.

Certainly, it is laudable to use modern programming languages which provide modern language features like genericity. However, very often there is no choice of using a different, more advanced language. Moreover, it will always be the case that existing programming languages lack features which have only recently emerged from programming language research. Rather than waiting for the perfect programming language, it is much more realistic to use mechanisms similar to CoffeeStrainer’s to express constraints which cannot be expressed directly using language features. Furthermore, there is a conflict between language simplicity and the support for more advanced language features, which could be avoided by providing mechanisms to add programmer-defined constraints, which can be thought of as programmer-defined language features.

Non-functional requirements

Another reason for constraints can be the existence non-functional requirements like security, performance, distribution etc. which dictate “bad” design.

For example, the constraint in `java.io.Writer`, which disallowed certain ways of synchronizing access to `Writer` objects is due to performance considerations that suggest using only one object for synchronizing access to a whole chain of `Writer` objects instead of synchronizing for each element of the chain separately.

Many example constraints that we have found are due to non-functional requirements which prevent the restructuring of the software towards “better” design.

Design-specific consistency requirements

We believe that the most important reason for constraints is due to design-specific consistency requirements. In complex object-oriented applications and, most notably, in object-oriented frameworks, the programmer defines abstractions which can be regarded as a kind of domain-specific language. The abstractions created by the programmer serve as the "syntax" of this language, and its "semantics" is determined by the implementation of the abstractions. However, the programmer cannot define a domain-specific "type system" which could prevent that the abstractions are used in a meaningless way. This is where constraints come into the picture: Both static and dynamic constraints can be used for expressing such a "type system" with its static and dynamic parts.

Thus, we expect that constraints will remain an important aspect of the development of complex object-oriented software even if the other two reasons for constraints could be removed by coming up with the "ultimate language" which does not lack important language features, and by providing ways of dealing with non-functional requirements which do not lead to constraints on the structure of programs.

8.3 Directions for future work

On the practical side, CoffeeStrainer could be improved in the following ways:

- *Integration of a decompiler*: Integrating a decompiler into CoffeeStrainer would allow to check constraints on program parts which are available in compiled form only. Furthermore, this would allow to check constraints at load-time similar to the byte-code verifier of Java which checks well-formedness and type correctness of byte-code files. Especially in the case of constraints for software security, checking constraints at load-time is a must. An alternative of using a decompiler would be to check and compile source-code at the same time, and using digital signatures to certify that a byte-code file has been checked successfully.

- *Support for advanced dynamic constraints*: As already discussed in Section 3.3.1, it is not possible to implement method postconditions in an empty interface `Programming-ByContract` in a way similar to the general implementation of method preconditions. This is because CoffeeStrainer provides no way of storing values computed at method entry which can be used in the postcondition at method exit. We were reluctant to add this feature in an ad-hoc manner because it might prove useful for other kinds of dynamic constraints as well, and examples for such advanced dynamic constraints did not occur in the Java standard classes.

- *Better reusability for method and field constraints*: The elegant technique of using empty marker interfaces for type constraints cannot be applied as easily for method and field constraints. One example where this deficiency became apparent is the empty marker interface `HasAnonymousMethods` in Section 5.7. The purpose of this interface is to check constraints regarding anonymous methods. It would be better if the tagging (Javadoc comment `/**@anon*/`) already carried the necessary semantics; but we were unable to come up with a solution to this problem which did not compromise modularity.

On the theoretical side, we see the following possibilities for further research.

- As with most new formalisms and languages, we expect that CoffeeStrainer can be used in ways which its designer did not foresee. We believe that the most interesting direction of further research is in the area of programmer-defined constraint systems which check non-trivial but important properties of programs. Confined types as proposed in Chapter 5 are probably just one interesting example of such programmer-defined constraint systems. One possible way of finding areas in which such constraints could be useful seems to be searching for particular keywords in the documentation of existing software.

- Constraints found by searching for keywords in documentation can be used for another purpose as well, namely, to detect areas of possible improvements to a programming language. For example, of the constraints found in the Java standard classes, two interesting subcategories, aliasing constraints and sequencing constraints, deserve further study. Before thinking of language features that support expressing these constraints within a language, it is probably a good idea to first realize them using a framework like CoffeeStrainer. Even if such constraints cannot be implemented easily by a programmer, it is probably useful for a language designer to experiment in the context of an existing language and with existing programs.