# Language Concepts and Design Patterns

Uwe Aßmann, Andreas Heberle, Welf Löwe, Andreas Ludwig, Rainer Neumann

Institut für Programmstrukturen und Datenorganisation
Universität Karlsruhe
Postfach 6980, Zirkel 2, 76128 Karlsruhe, Germany
(assmann|heberle|loewe|ludwig|rneumann)@ipd.info.uni-karlsruhe.de

**Abstract.** Programming languages aim at the construction of simple but expressive programs. To achieve this, plenty of language concepts have arisen over time. Design patterns aim at the solution of common design problems. To achieve this, plenty of approved design concepts have been collected.
We claim that language concepts and design patterns are essentially the same. Indeed, a language may offer a design pattern as a language concept; we call such patterns *language patterns*.
A design pattern can be implemented in terms of other design or language patterns. Since a concrete programming language only supports a subset of language patterns, every other pattern must be expressed in terms of this subset. We call such an implementation a *workaround*.
The specification of a workaround imposes proof obligations: it must be shown that a workaround simulates the pattern. Once proved correct, we can collect patterns and their workarounds in a trustworthy catalogue. This helps software developers to correctly apply patterns in any language and helps the language designer to decide which patterns to put into the language core.
We demonstrate this pattern integration process with well-known design patterns and concepts of object-oriented languages. Additionally, we list important language patterns together with their workarounds.

**Keywords:** decomposition, design pattern, language design, language pattern, meta-programming, programming languages, program transformation

## 1 Introduction

Design patterns have brought software engineering closer to other engineering disciplines, where the reuse of standard solutions to common problems is already state of the art. Many of approved design concepts have been collected in catalogues during the last years [9, 3, 23].

Recently, many language constructs appear as design patterns in these catalogues. This somewhat astounding trend results from the fact that a particular programming language does not support all well-known language constructs; either the integration of a certain pattern into the language is too costly or not desired. To use the concept nevertheless, it is described as a design pattern so

that programmers know how to work around the concept in terms of the language.

On the other hand, more and more design patterns appear as concepts in modern programming languages, in particular in domain-specific languages. While these languages become more expressive and powerful they tend to be more complicated. Extending a language with a new concept is a cost-intensive procedure: a new compiler has to be constructed or an existing one has to be extended. Also, since design patterns often have informal semantics, they cannot directly be mapped to the machine level. To this end, it is required to formalize a design pattern, but this reduces the degree of freedom in application of the pattern.

Hence, a systematic method to integrate design patterns into languages would be highly appreciated. Such a method should provide automation support for patterns; it should rely on a precise semantics for patterns; it should provide a proof methodology such that a language designer or compiler writer can develop a trustworthy implementation. In short: it should provide a *pattern integration process*, i.e. a means to help integrating patterns into languages, together with a quality management method to assure that the automated patterns work correct.

This paper presents such a process. Its underlying insight is that

*Design patterns and language constructs are conceptually equivalent.*

This insight yields a uniform view on design patterns and language constructs as *patterns*. To this end, we define the notion of a *workaround* for patterns in terms of other patterns. Workarounds can be created by a static meta-program in an open language. The meta-program maps a new language construct, namely the formalized pattern, to the constructs of the base language so that the compiler automatically generates code for the pattern. As a functional basis we present a formal syntax and semantics for patterns, based on initial ground term algebras and Abstract State Machines. Using this formalization, it can be proven that the workaround behaves like the designated pattern.

The remainder of this paper is organized as follows: The following section describes work related to ours. In section 3 we discuss the relationship between language concepts and design patterns. Sections 4 and 5 show the equality of language concepts and design patterns. It demonstrates how meta programming can be used to extend languages by patterns. In section 6 we define a methodology for correctness proofs for workarounds, i.e. we explain how to proof that a workaround simulates the expected behavior of the pattern. Section 7 summarizes our results and gives an outlook on forthcoming work.

## 2 Related Work

[9] introduced the first design pattern catalogue. To avoid burdensome definitions of basic language concepts, these patterns are intentionally based on C++ and SmallTalk. This continues in the catalogue [3] where *idioms* are additionally defined as language specific implementation schemes. In the pattern survey in

[23] it can be observed how language concepts begin to arise as design patterns. [10] defines *cadets* and *idioms* in order to classify design patterns with respect to programming languages. In our terminology, an idiom as defined in [10] would roughly translate to a workaround as we will define in a later section.

[2] sees the need to support the evolution of concepts and identifies a "semantic gap" between design patterns and language concepts which leads to problems. We agree on this and bridge that gap by the notion of workarounds. Further, [2] claims that design patterns are conceptually orthogonal to language concepts. A similar claim can be found in [7]: design patterns should be classified as *fundamental* if they do not use language specific concepts. However, we think one should not try to constitute which design patterns may be allowed to become language concepts and what should not. We believe that design pattern hierarchies as described in [25] nicely *extend* to language concepts.

In [15], the author notes that it is necessary to provide easy means to include a pattern into a language and proposes the use of attributed grammars to define the syntax of patterns and to enable static type checks based on patterns. This is reasonable as attributed grammars have proven their worth. The idea fits perfectly in our picture as design patterns obviously do not differ in their statical semantics from common language constructs. What remains is to look at the dynamic semantics and to give a precise formal definition of uses and similar relations, both of which we will provide in this paper as part of a pattern integration process.

## 3 Design Patterns and Programming Languages

This section describes some basic observations on design patterns and programming languages. We show how design patterns are currently related to programming languages. They can depend on their underlying language in several ways:

- Design patterns can be language concepts.
- Design patterns can depend on language concepts.
- Design patterns can ignore "alien" language concepts.

We observed the following phenomena which we think are consequences of these dependencies.

First of all, in certain languages design patterns occur as fully supported language concepts. Some of these concepts have already invaded the domain of design patterns, such as Try-and-Catch, see [23].

*Example 1.* In prototype-based languages [19, 4] a Prototype is the only way to create new objects. An Iterator is a common part of database query languages and languages incorporating data streams such as Sather [17, 12].

Other design pattern sometimes solve problems that stem from the lack of support for a language concept.

*Example 2.* A CONVENIENCE CLASS as depicted in [23] can bundle parameters to reduce long signatures, either to gain expressiveness or to reduce parameter transfer costs. However, a CONVENIENCE CLASS sometimes arises from the need for multiple output parameters.

Sometimes, only a part of the functionality of a design pattern deals with the replacement of missing language concepts.

*Example 3.* An OBJECT ADAPTER as depicted in [9] is a general workaround for the lack of multiple inheritance. If a specific language does not support multiple inheritance or if multiple inheritance does not seem appropriate for any particular reason, a workaround using delegations is possible. This workaround does not depend on the problem context ADAPTER and hence should not be coupled with it.

Last not least, there are language concepts that could have an impact on design patterns but are largely ignored by now. A good programmer can use the idea behind the concept in an "alien" language and transfer it as a workaround to the language in use. If patterns are written without a full awareness of language concepts, this is not possible.

*Example 4.* A couple of pattern variants can be formed using genericity, such as the STATIC BRIDGE [8]. This idea extends nicely to other design patterns such as DECORATOR or PROXY.

How do these observations match with the notion of language indepedent design patterns? Obviously, design patterns are currently tailored to a specific language model that due to [9] is roughly equivalent to a subset of C++. This clearly stems from the use of C++ for implementation examples.

Our idea is that language concepts can be regarded as design patterns and that design patterns can be systematically incorporated as new language concepts. We show that both directions are valid.

## 4 From Language Concepts to Design Patterns

We now present arguments why it is desirable to describe language concepts as design patterns. Informally, a *design pattern* is a schematic documentation of an approved solution for a specific design problem which incorporates reuseable expert knowledge. The most important part of a pattern is its name and the list of aliae defining a common terminology. A pattern also includes a motivation, a purpose, a discussion of the applicability including its relations to related patterns, and the limitations or side-effects of an application. The pattern gives hints for possible implementations and illustrates them by examples.

Interestingly, all these requirements for documentation could be repeated for language constructs. Language manuals, language rationals, language lecture note books, and language tutorials all try to fulfil these requirements in order to specify the semantics of the concepts precisely although often informally.

Additionally, in the same way as design patterns are assembled in catalogues, language lecture notes try to collect language concepts systematically in order to help teaching students. A catalogue for language concepts could look very similar to the currently used design pattern catalogues. The multiple variants would require an extra section "known variants" which would compare variants in different languages including source code examples and refer to corresponding literature. The implementation part would then describe and exemplify workarounds.

Most design patterns provide a couple of variants and factorize their common parts. Language concepts also differ in fine, but important details between the languages (see example 5). Thus, the conceptual idea and possible variations must be pointed out thoroughly. If done properly, this would allow a detailed configuration of design patterns derived from those language concepts.

*Example 5.* GENERIC CLASSes in Eiffel [16] form a subtype hierarchy that is parallel to the hierarchy of their generic parameters, while they do not in C++ [21]. EXCEPTIONs have different catch, propagation and resume strategies in different languages. Multiple inheritance comes in different flavors such as interface inheritance or mixin-based inheritance with early or late conflict resolution. ASSERTIONs in Eiffel and C++ are supported at different degrees. Perl [20] provides associative arrays as standard COLLECTIONs while most other languages only support standard arrays.

Design patterns usually assume that there is no predefined solution that matches the problem; hence an implementation by hand must be provided. A language concept is defined in a programming language and therefore does not have to be implemented. However, in a language that does not support the concept, an implementation becomes necessary. This is the reason why language concepts are being described as design patterns: to let a programmer implement the pattern in a language which does not directly support it.

This leads to two definitions. If a design pattern is supported in a given language we call it a *language pattern* wrt. the given language. An implementation of a non-supported pattern is called a *workaround*. A workaround is similar to an *idiom* [3] but not the same. While an idiom characterizes a certain coding pattern in a particular programming language, a workaround realizes a pattern in terms of other design or language patterns.

An example for such a workaround follows in the next section.

## 5 From Design Patterns to Language Concepts

We have demonstrated that language concepts can be specified as design patterns. It remains to show that the opposite is also true: design patterns can be incorporated into programming languages.

### 5.1 Applying Workarounds by Program Transformations

[26] showed that the application of a design pattern can be regarded as a program transformation. In the very same way, we can use dedicated transformations to implement language concepts:

*Example 6 (*VISITOR*).* Imagine you want to apply a VISITOR pattern. In Cecil [4], multimethods are available and the structure of a VISITOR using a double-dispatched method looks rather different than that depicted in [9]. The following program transformation scheme replaces a multi-dispatched polymorphic call[1] by a cascaded sequence of single-dispatched calls:

$$\begin{aligned} f(x_1^*, x_2^*, \ldots, x_n^*)\{\mathcal{F}(x_1, \ldots, x_n)\} \rightarrow\ & f_1(x_1^*, x_2, \ldots, x_n)\{f_2(x_1, \ldots, x_n)\}, \\ & f_2(x_1, x_2^*, x_3, \ldots, x_n)\{f_3(x_1, \ldots, x_n)\}, \\ & \ldots \\ & f_{n-1}(x_1, \ldots, x_{n-2}, x_{n-1}^*, x_n)\{f_n(x_1, \ldots, x_n)\}, \\ & f_n(x_1, \ldots, x_{n-1}, x_n^*)\{\mathcal{F}(x_1, \ldots, x_n)\} \end{aligned}$$

The VISITOR problem can be handled by a multimethod

*visit*($e^*$: *Element*, $a^*$: *Action*) { do something with $e$ and $a$ }.

The transformation yields the following function declarations:

*visit*$_e$($e^*$: *Element*, $a$: *Action*) { *visit*$_a$($e, a$) }
*visit*$_a$($e$: *Element*, $a^*$: *Action*) { do something with $e$ and $a$ }

After some renaming and with the common notation for the single dispatch parameter, we receive the well-known pattern from [9]:

*Element*::*accept*($a$: *Action*) { *a.visit*(*this*); }
*Action*::*visit*($e$: *Element*) { do something with $e$ and *this* }

Obviously, program transformations are capable to implement design patterns. Unfortunately, the definition of a transformation does not provide an automated solution per se; it would be desirable to integrate such a transformation directly into a language.

### 5.2 Patterns as Language Extensions

Since the sixties, researchers investigate extensible programming languages which can be extended with new syntax constructs [5]. Recently, it has been discovered that this is possible by *static meta-programming*. An *open* language offers its abstract syntax tree as a library, i.e. as a meta-object protocol allowing the user to annotate a new keyword or syntax construct to a meta-program. Each time the compiler processes the keyword, it starts the related meta-program; it

[1] The dispatch parameters are marked by an asterics*.

is able to introspect and to modify the abstract syntax trees of the currently translated classes, giving a semantics to the new language construct. The later phases of the compiler – semantical analysis as well as code generation – do not need to be modified. Since all meta-programs are evaluated at compile-time, no run-time overhead is created. Two modern open languages are OpenC++ [6] and OpenJava [22].

*Example 7.* The OpenJava distribution contains an example workaround for the language pattern GENERIC CLASS: an instantiated template class is flattened to an ordinary class by heterogeneous template parameter expansion [11].

In consequence, design patterns can be implemented by hand, integrated into a language as a construct, or implemented by a static meta-program in an open language. Of course, meta-programs for language extensions can be developed much faster than new compilers, but they allow only simple optimizations.

### 5.3 Pattern-based Language Design

These arguments lead to an evolutionary process in language design, the *pattern integration process*. When design patterns are implemented as meta-programmed language extensions language designers can early gain experience with new language concepts. When a pattern has proven to be valuable and manageable, it may be integrated as a native language construct (Fig. 5.3). In this way, more and more abstract design concepts enrich programming languages and a next generation of more expressive languages results. Of course, a language designer will try to define a small language core incorporating only the most important concepts. To find an acceptable compromise, he must evaluate the cost of a workaround and decide whether to support it as a language pattern, as a language pattern in an open language, or leave it to the programmer as a design pattern.

Consider the example of Java. This language provides only a comparatively small set of language patterns but this did not prevent its success. The designers obviously decided that certain language patterns could be worked around easily and omitted those from the language. Some of the omissions are heavily debated meanwhile, especially ASSERTION, CLOSURE, and GENERIC CLASS. These discussions have lead to the development of several language dialects which integrate these concepts as language patterns [18] [11]. Based on an open language, this process would be greatly facilitated.

### 5.4 Workarounds and Non-functional Aspects

A general-purpose language allows to work around all functional parts of a foreign language pattern.[2] However, while a workaround might provide a semantically equivalent solution, it is usually not as sophisticated as a native language

[2] To support concurrency, synchronization as provided by a SEMAPHORE requires a non-interruptable test-and-set operation.

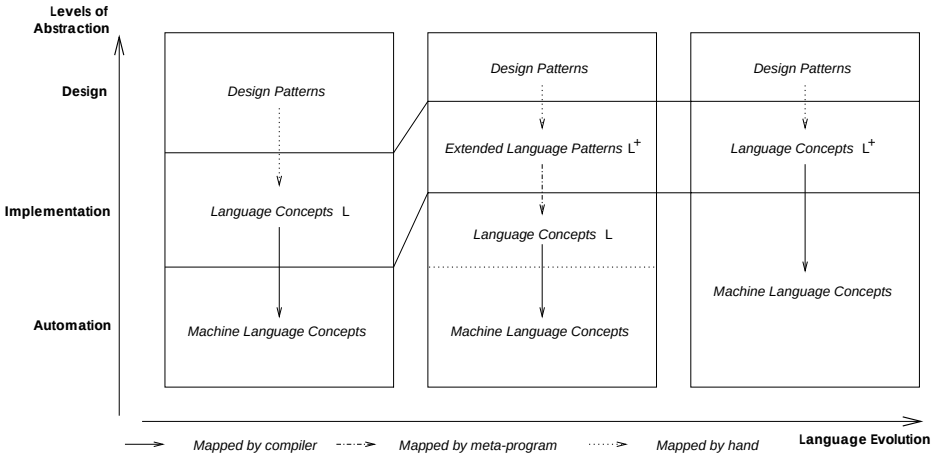


**Fig. 1.** The pattern integration process: design patterns are mapped by hand to a language $L$, then they are implemented with an extensible language yielding an extended language $L^+$, and finally compilers integrate them as native language constructs.

construct. This is obviously the case for certain non-functional properties such as the syntactical representation or the efficiency of the compiled code.

**Loss of syntactical expressiveness** A language pattern has a certain syntactical structure. A workaround may come with an "uglier" syntactical code structure, the source code grows, becomes more complicated, and more difficult to maintain. Especially, this is the case when other aspects are interwoven with the workaround code so that its purpose is no longer visible. These effects have been noted by [2] as traceability and reusability issues.

**Loss of efficiency** A workaround usually does not match the performance of a supported language pattern. Often the code pattern of a workaround has to be matched by an optimizer in order to create target code that is tailored to the pattern. However, the optimizer is not aware of this information.

It is possible that a workaround has a worse asymptotic behavior. As [28] has shown, pure functional languages would implement array accesses in $\Omega(\log n)$. Luckily, most workarounds produce only a slight overhead that does not change complexity.

*Example 8.* A typical workaround in C is to provide Polymorphic Features using functions pointer arrays and structures with a type tag. However, for a non-trivial subtype graph only a compiler can provide an enumeration for an optimal dispatch [24].

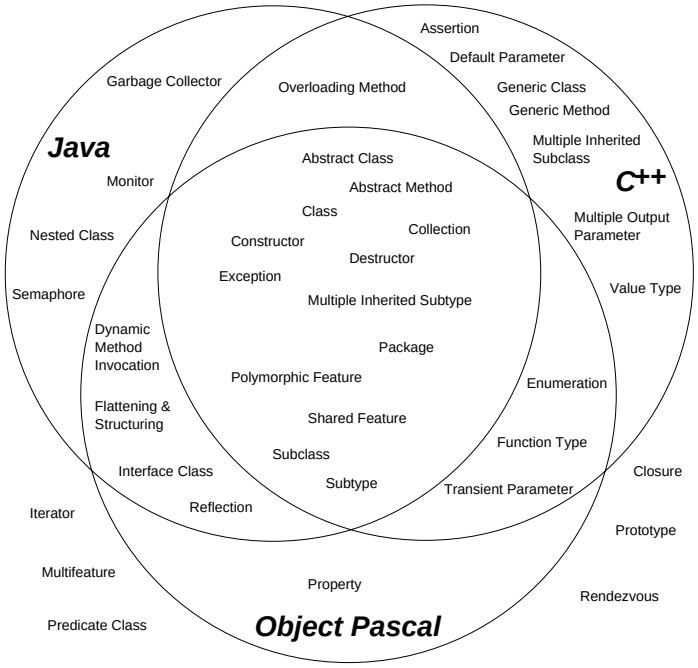


**Fig. 2.** Concepts in different programming languages

**Loss of statical type and access safety** An ordinary language does not provide access to the static types of a program. When a workaround must grant more access rights to certain modules or must weaken types, additional statical type or access checks would be required. This results in a loss of safety since a workaround by hand must also perform these checks by hand.

*Example 9.* A Generic Class can be worked around by heterogeneous template parameter expansion. While this workaround can be made type-safe, it enlarges the amount of code enourmously, since the code is replicated. The other workaround, *homogeneous parameter expansion* compiles the code only once, but is not type-safe since the parameters are replaced by the most general type [18].

**Loss of memory control** Certain operations might require control over the storage location of a particular data object, e.g. not being stored in a register. A lack of these concepts cannot be worked around.

### 5.5 Proof obligations

When a design pattern is implemented by a workaround correctness is a critical issue. This does not only hold for hand implementations, but is very impor-

tant for developing correct meta-programmed workarounds in the the pattern integration process.

In order to prove a mapping of a pattern to a workaround correct, the programmer has to select specific specification methods for syntax and semantics of patterns and workarounds. Using these specification methods, it has to be proved that the semantics of the pattern is the same as the semantics of the workaround. To simplify the proofs, syntax as well as semantic specification mechanisms should be the same for patterns and workarounds.

In the next section we choose ground-term algebras as syntactical and Abstract State Machines as semantical mechanism. This is motivated from verification in compiler construction [14] and leads to a simple and powerful proof methodology for the pattern integration process.

## 6 A Formal Base for Pattern Decomposition

First of all we define the notions of *design and language pattern* as they are used in the context of this work. Second, we define *syntax and semantics of patterns.* Therefore, we introduce a specification language suitable for syntax and semantics definitions. We will see that patterns, design as well as language patterns, are in general defined in terms of other patterns. We define this relation as the *use* relation over patterns. A workaround for a pattern *simulates* this pattern which is another relation over patterns that we have to define formally. The simulation relation bases on the operational semantics of the pattern and its workaround. We use Abstract State Machines (ASM) to define semantics operationally. ASMs have successfully been used for the specification of various programming languages, e. g. C [13] or Java [1]. In compiler construction correctness of translations is established by proving a simulation relation on the semantics of source and target code, see e. g. [27]. We adopt the technique in order to proof correctness of workarounds. The simulation relation on patterns and workarounds implies some proof obligations which are discussed at the end of this section. Figure 3 sketches the correctness requirements. $ps'$ is a workaround of a pattern $ps$ if the semantics defined by $ps'$ simulates the semantics defined by $ps$.

$$\begin{array}{ccc} ps[\![t]\!] & = & ASM(t) \\ & & \uparrow \\ \sqsubseteq & & simulates \\ & & | \\ ps'[\![t']\!] & = & ASM(t') \end{array}$$

**Fig. 3.** Correctness of Workarounds

### 6.1 Design and Language Patterns

We begin with an example in order to make the definitions and proof more descriptive.

*Example 10.* A CLOSURE is an object containing a function $f$ with formal parameters $X = x_1, \ldots, x_n$ of $f$ and result $x_r$. A possibly empty subset of the formal parameters $B \subseteq X$ can be bound to some actual parameters $o_i, \ldots o_k$. The closure can be executed, i.e. $f$ is executed, if all formal parameters are bound.

In this informal definition, we identify other patterns like FUNCTION, FORMAL PARAMETER, ACTUAL PARAMETER, and RESULT. Furthermore, we defined functions over closures as bind and execute.

With respect to concrete languages , CLOSURE may be design or a language pattern. This leads us to a formal definition of design and language patterns.

**Definition 1 (Design Pattern).** *A* design pattern *$d$ is a many sorted algebra $d = (A, \Sigma, Q)$, with sorts $A = D \cup L \cup \mathbb{B}$ where $D$ is a set of design patterns, $L$ is a set of language patterns, and $\mathbb{B}$ is the Boolean algebra.*

**Definition 2 (Language Pattern).** *A design pattern $l = (A, \Sigma, Q)$ is a* language pattern *wrt. a programming language $\mathcal{L}$ iff $l$ is defined in $\mathcal{L}$.*

Obviously, it holds the following

**Theorem 1 (Closedness of Language Patterns).** *If $l = (A, \Sigma, Q)$ is a language pattern wrt. a programming language $\mathcal{L}$ then for all patterns $l' \in A$ it holds that $l'$ is a language pattern wrt. $\mathcal{L}$.*

*Example 11.* The CLOSURE can be defined by the following algebra:

$$closure = (A_{closure}, \Sigma_{closure}, Q_{closure}), \text{ where}$$

$$\begin{aligned}
A_{closure} &= \{closure, formal_parameter, actual_parameter, function, result\} \\
\Sigma_{closure} &= \{ \\
&\quad create: \ function \to \{bound_parameter\} \to closure \\
&\quad bind: \ closure \to actual_parameter \to formal_parameter \to closure \\
&\quad eval: \ closure \to result \ \} \\
Q_{closure} &= \{ \\
&\quad bind(create(f, bps), ap, fp) = create(f, bps - \{(ap, _)\} \cup \{(ap, fp)\}) \\
&\qquad eval(create(f, bps)) = eval(f(bps)) \ \}
\end{aligned}$$

For some languages $\mathcal{L}$, CLOSURE can be directly used in programs of $\mathcal{L}$, i.e. is defined in that language $\mathcal{L}$. For those languages, the design pattern is a language pattern.

### 6.2 Syntax and Semantics of Patterns

For language patterns (language constructs), it turned out helpful to distinguish syntax of the language from the semantics[3]. This distinction is common sense since it allows the construction of modular compilers, i.e. compilers checking for the static correctness of a language pattern before generating code. It also is advantageous with extended language patterns, since it allows to prove the workaround mapper – the meta-program expressing the semantics of the extended language pattern – correct. As we will see, a workaround for a design pattern $d$ is nothing but a correct transformation of $d$ to some language patterns $L$.

**Definition 3 (Syntax of Patterns).** *Let $d = (A, \Sigma, Q)$ be a design pattern. The* syntax *of $d$ is the initial ground term algebra $d^s = (A^s, \Sigma^s)$ where*

$$\sigma : d_1^s \to \cdots \to d_n^s \in \Sigma^s \Leftrightarrow \sigma : d_1 \to \cdots \to d_n \in \Sigma \wedge d_n = d$$
$$d_a^s \in A^s \Leftrightarrow d_a \in A \wedge \exists \sigma : d_1^s \to \cdots \to d_n^s \in \Sigma^s : d_i^s = d_a^s, i = 1..n$$

Note that for language patterns, Definition 3 is just another notation of abstract syntax trees which extends the idea of abstract syntax trees nicely to design patterns.

From the syntax we distinguish the syntactic structure of design patterns. The syntax defines correct programs creating design patterns and operations over them. The structure defines the syntax of instances of a certain design pattern.

**Definition 4 (Structure of Patterns).** *Let $d = (A, \Sigma, Q)$ be a design pattern with $d^s = (A^s, \Sigma^s)$. The* structure *of $d$ is an initial ground term algebra $d^{struct} = (A^{struct} \subseteq A^s, \Sigma^{struct})$ where*

$$(\sigma_i^{struct} : d^{struct} \to d_i^{struct}) \in \Sigma^{struct}$$

*for each argument $d_i, i = 1..n$ of a signature*

$$\sigma : d_1^s \to \cdots \to d_n^s \to d^s \in \Sigma^s$$

*in the syntax $d^s$.*

*Example 12.* The syntax $closure^s$ of the CLOSURE is described by the following initial ground term algebra:
$closure^s = (A^s_{closure}, \Sigma^s_{closure})$, where

$A^s_{closure} = \{\ closure^s, bound\_parameter^s, function^s\ \}$
$\Sigma^s_{closure} = \{$

[3] Although this is not necessarily required, cf. the $\lambda-$ and $\pi-$ calculus.

$create : function^s \rightarrow \{bound\_parameter^s\} \rightarrow closure^s$
$bind : closure^s \rightarrow actual\_parameter^s \rightarrow formal\_parameter^s \rightarrow closure^s$
$eval : closure^s \rightarrow result^s$ }
The structure of CLOSURE includes functions

$$bound\_parameter : closure^{struct} \rightarrow bound\_parameter^{struct}$$
$$bound\_function : closure^{struct} \rightarrow function^{struct}$$

The semantics of a pattern could be defined axiomatic by its algebras. However, our intension is to apply these definitions to imperative languages. Therefore we choose a more operational style of semantic definitions and use Abstract State Machines (ASMs). ASMs describe mathematical machines operating on algebras which model the states. ASMs introduce the notion of universes as unary predicates which represent the sorts. Each state of the ASM has the same signature together with an interpretation. The modification of these interpretations define state transition. In ASM, updates of function interpretations are specified by update rules. A brief introduction is contained in the Appendix B. We define three universes: $Objects$ to define state, $Tasks$ to define the program that in turn defines state transitions, and $Value$ to define basic values. For simplicity, the latter is assumed to be specified in Boolean and Integer algebras, respectively, and is not defined here. Objects may be structured, i.e. there are functions

$$\sigma : Object \rightarrow \{Value, Object\}.$$

In general, each design pattern defines a part of the initial state of a program, they define some objects $o$ and some functions $\sigma(o)$ and state transitions. The state transitions change this initial state, i.e they may

1. change the interpretation of some existing function $\sigma(o)$ or
2. define a new, yet undefined function $\sigma(o)$ or
3. create a new object $o$

How this interpretation changes is defined by the tasks: for each specific task type we define corresponding updates of functions. A global program counter, modeled by a 0-ary function $ct \rightarrow Task$ defines the current task and, thus the updates to execute. The current task is initially set to the first task of a program and is modified in the updates of the tasks. Finally, we have to model call stacks, data and sequential control flow. The call stack is a stack of common objects. We therefore only define a function

$$current\_stack\_frame :\rightarrow Object$$

which defines the current top element of the call stack. The data flow is modeled by a function

$$value : Task \times \mathbb{N} \rightarrow \{Value, Object\}.$$

The value of a task depends on the current recursion level of the execution which is modeled by

$$reclevel :\rightarrow \mathbb{N}.$$

The control flow is defined by a function

$$next : Task \to Task.$$

The update of the function $value(t, reclevel)$ in the context of a task $t$ defines the value that $t$ computes. The update $ct := next(ct)$ defines the next task to compute after $t$. For non sequential control flow, $ct$ must be updated alternatively according to some Boolean value. W.l.o.g. our transition rules have the following form

**if** $ct = \sigma(x_1, \ldots, x_n)$ **then**
  *updates*
  $value(ct, reclevel) := \ldots$
  $ct := \ldots$
**endif**

We sometimes skip the definition of *value* if it is not required.

Given a term of the syntax of patterns, the semantics ought to define an initial state $I$ of an ASM, i.e. some objects, functions over these objects and a set of tasks possibly updating these function.

For each function $\sigma$ in the signature of a design pattern, we define a task. The result of $\sigma$ is encoded by the *value* function of the corresponding task and is defined in an update performed by that task.

For each function

$$\sigma : d^{struct} \to d_i^{struct}$$

in the structure of a pattern $d$, updates of the corresponding task create an object $d$. Additionally it initially defines functions

$$\sigma : d \to d_i$$

and, hence, the initial state of pattern $d$.

The definition of semantics of a pattern merges ASMs, see appendix B for a more detailed definition[4].

**Definition 5 (Semantics of a Pattern).** *Let $d = (A, \Sigma, Q)$ be a design pattern with syntax $d^s = (A^s, \Sigma^s)$. The semantics of $d$ is a mapping of each program term $p \in d^s$ to an Abstract State Machine $d[\![p]\!] = (\Sigma^{[\![]}, S, A^{[\![]} \cup Task, \to, I)$. Let $d_1[\![x_1]\!], \ldots, d_n[\![x_n]\!]$ be the semantics, i.e. the corresponding ASMs, of terms $x_1, \ldots, x_n$ of patterns $d_1, \ldots, d_n$. Let $t(x_1), \ldots, t(x_n)$ be the tasks of $x_1, \ldots, x_n$ in the respective ASMs. Let $X = \biguplus_{i=1}^{n} d_i[\![x_i]\!]$.*

$$d[\![\sigma(x_1, \cdots, x_n)]\!] = (\Sigma^{[\![]}, S, A^{[\![]} \cup Task \cup Object, \to, I) \uplus X$$

*where*

$$\Sigma^{[\![]} = \{next : Task \to Task$$
$$value : Task \times \mathbb{N} \to d_n\}$$

[4] Here, we assume merged ASMs to be valid.

$$A^{[]} = \{d, d_1, \ldots, d_n\}$$
$$Task = \{\sigma\}$$

*Initially, only the next function is defined according to the intended control flow. $S$ is inductively defined by $I$ and $\rightarrow$. If $\sigma$ is a constructor of $d$ then $\rightarrow$ is defined by the transition rule:*

***if*** $ct = \sigma$ ***then***
  ***extend*** $d$ ***with*** $x$
    $\sigma_1^{struct}(x) := value(t(x_1), reclevel)$
    $\vdots$
    $\sigma_n^{struct}(x) := value(t(x_n), reclevel)$
    $value(ct, reclevel) := x$
  ***endextend***
  $ct := next(ct)$
***endif***

*where $\sigma_i^{struct}$ is in the signature of the structure of $d_i$. If $\sigma$ is not a constructor of $d$ then $\rightarrow$ is defined by the transition rule:*

***if*** $ct = \sigma$ ***then***
  $updates(x_1, \ldots, x_n)$
  $ct := next(ct)$
***endif***

*and the updates of the functions in the structure of $d$ conform to the axioms $Q$.*

The definition of a semantics of a design pattern is generic in the sense that it requires semantics definitions of other patterns for completeness. Additionally, it poses some restrictions on these other semantics definitions. We do not further discuss the correctness of the definitions of syntax and semantics of a pattern $d$ wrt. its algebra. Instead, we assume syntax and semantics of a pattern $d$ to be the definition of $d$. Hence, we do not further define the notion of conformance of updates in an ASM of a pattern and the axioms of its algebra. Our example may give some more insights:

*Example 13.* Let $f$ and $bp$ be terms of the patterns FUNCTION and BOUND_PARAMETER, respectively. The semantics $closure[\![create(f, bp)]\!]$ adds a task *create* to the task universe and extends the transition rules by the following rule:

**if** $ct = create$ **then**
  **extend** $Closure$ **with** $c$
    $function(c) := value(t(f), reclevel)$
    $bound\_parameters(c) := value(t(bps), reclevel)$
    $value(ct, reclevel) := c$
  **endextend**
  $ct := next(ct)$
**endif**

Let $ap$ and $o$ be terms of the patterns ACTUAL_PARAMETER and OBJECT, respectively, and $c$ an term of CLOSURE. The semantics $closure[\![bind(c, ap, o)]\!]$ adds a task $bind$ to the task universe and extends the transition rules by:

**if** $ct = bind$ **then**
  **let** $bp = bound\_parameters(value(t(c), reclevel)$**in**
  $bp := bp - \{value(t(ap), reclevel), \_\} \cup$
    $\{value(t(ap), reclevel), value(t(o), reclevel)\}$
  $ct := next(ct)$
  **endlet**
**endif**

Let $c$ be a term of $closure$. The semantics $closure[\![eval(c)]\!]$ adds a task $eval$ to the task universe and extends the transition rules by:

**if** $ct = eval$ **then**
  **if** $visited(ct, reclevel)$ **then**
    $visited(ct, reclevel) := false$
    $value(ct, reclevel - 1) :=$
      $value(lasttask(bound\_function(value(t(c), reclevel))), reclevel)$
    $reclevel := reclevel - 1$
    $ct := next(ct)$
  **else**
    $visited(ct, reclevel) := true$
    **extend** $Object$ **with** $o$
      **do forall** $(ap, o') : (ap, o') \in bound\_parameters(value(t(c)))$
        $ap(o) := o'$
      **endforall**
      $dynamic\_predecessor(o) := current\_stack\_frame$
      $current\_stack\_frame := o$
    **endextend**
    $reclevel := reclevel + 1$
    $ct := firsttask(bound\_function(value(t(c), reclevel)))$
  **endif**
**endif**

$visited$ depends on the current recursion level. In the beginning $visited$ is set to $false$. Evaluation of a function implies an incrementation of the recursion level.

The semantics of closure is complete only with the definition of the semantics of the patterns FUNCTION and BOUND PARAMETER. It requires that the pattern FUNCTION defines a $firsttask$ and a $lasttask$ function.

### 6.3 Correctness of Workarounds

Having defined syntax and semantics of patterns, we are now able to define relations between patterns.

**Definition 6 (Use Relation of Patterns).** *Let $d = (d^s, d[\![])$ be a design pattern. $d$ uses all patterns $d_i$ with $d_i[\![] \in A^{[\![]}$ of $d$. We denote this by $use(d, d_i)$.*

The transitive closure of the use relation of a certain design pattern is required to identify the design and language patterns that are necessary to implement that design pattern.

In order to prove the correctness of a workaround we have to establish a simulation relation on the design pattern implemented by the workaround. This relation is defined in terms of the simulation relations over the semantics of design patterns, i.e. in terms of simulation relations over ASMs. Therefore we discuss the notion of simulation of ASMs.

In general, not all state transitions of an ASM are observable from outside. An observer can not distinguish runs of two different programs as long as they show the same input/output behavior. For our purposes it is sufficient to assume that only events are observable which read an input of the environment or write an output to the environment. We model these events by input and output streams. Thus, observable behavior can be modeled by merging all states where the following state transition does not change the interpretation of the input or output stream, see figure 4. Simulation of two ASMs $a$ and $b$ is now

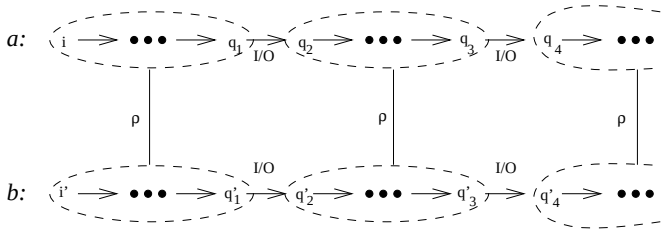


**Fig. 4.** Simulation of observable behavior

defined similar to the simulation notion in complexity and computability theory. $\rho$ relates states of ASM $b$ and ASM $a$ showing the same observable behavior[5]. ASM $b$ simulates ASM $a$ if for every observable behavior of the $b$ there exists a corresponding behavior of $a$. A detailed discussion of observable behavior and our notion of simulation can be found in [27].

**Definition 7 (Simulation Relation of Patterns).** *Let $d = (d^s, d[\![])$ be a design pattern. $d$ is simulated by a set of patterns $D = \bigcup_{i=1}^{n} d_i, d_i = (d_i^s, d_i[\![])$ iff $d[\![]$ is simulated by $\biguplus_{i=1}^{n} d_i[\![]$. We denote this by $D \sqsubseteq d$.*

[5] This is a slight simplification since in general it is possible that the observable behavior of $a$ and $b$ is modeled by different functions. Then we have to find an additional function $\overline{rho}$ which defines an injective mapping from observable functions of $a$ to the observable functions of $b$.

$\sqsubseteq$ can be interpreted as a contra-covariant subtype relation over patterns. Note, that no update in the semantics of a pattern in $D$ can update functions of the signature in the pattern $d$. Hence, in order to fulfill $D \sqsubseteq d$, the signature of $D$ needs not necessarily contain the signature of $d$.

*Example 14.* The design pattern CLOSURE uses the patterns FUNCTION and BOUND PARAMETER. The CLOSURE can be simulated by a class (as usually available in object oriented languages). This class must implement parameter binding and currying.

The transitive closure of $\sqsubseteq$ for a certain design pattern is required to translate this pattern to a concrete programming language. We now formally define the notion of a workaround:

**Definition 8 (Workaround of Patterns).** *Let $d = (A, \Sigma, Q)$ be a design pattern. A workaround of $d$ is a set of patterns $D$ with*

$$D \sqsubseteq d, \text{ and}$$
$$\forall d_i \in D : use^+(d_i, d') \Rightarrow d' \text{ is a language pattern,}$$

*where $use^+$ is the transitive closure of the use relation over patterns.*

A special form of the simulation relation plays an important role for design patterns and their efficient implementation:

**Definition 9 (Extension of Patterns).** *A design pattern $d$* extends *a design pattern $e$ iff $d$ simulates $e$ but $e$ does not simulate $d$.*

*Example 15.* Figure 5 shows the relationships between CALLBACKS, CLOSURES, FUNCTION TYPES and classes: A CALLBACK can e.g. be simulated by FUNCTION TYPES. Since a CLOSURE is an extension to a FUNCTION TYPE, the workaround with FUNCTION TYPES can also be used with CLOSURES.

A pattern that is extended by another pattern can often be implemented more efficiently. The combination of a language pattern and a more powerful language pattern is only sensible if it cannot be decided in general whether or not a use of the simpler pattern would be sufficient, since then the task to produce highly efficient code could be left to the optimizer.

*Example 16.* A VALUE TYPE could be applied automatically if no references must be manipulated. A monomorphic call can replace a polymorphical one if only one type is possible during the dispatch. A CLOSURE can be replaced by a FUNCTION TYPE if no parameters are bound.

A workaround defines the implementation of a design pattern $d = (d^s, d[\![])$ in terms of language patterns of a language $\mathcal{L}$. For the definition of a workaround we have to proof $D \sqsubseteq d$. $d[\![]$ is defined in terms of $d_1[\![], \ldots, d_n[\![] \in A^{[\![]}$ whose semantics is defined in terms of the semantics of their components and so on. In our example the semantics of CLOSURE is defined on the semantics of FUNCTION

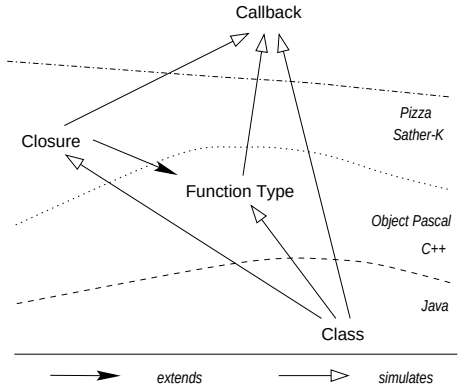


**Fig. 5.** Implementation of CALLBACKS based on other concepts

which is build up merging the semantics of PARAMETERS and STATS and so on. This results in huge ASMs on which we have to proof a simulation relation. We rather want a structuring of this proof.

As already mentioned, the semantics of a design pattern $d = (d^s, d[\![])$ is defined using $d_i[\![] \in A^{[\![]}$. Nevertheless, in order to define the semantics of $d$ it is not necessary to know the concrete semantics $d_i[\![]$. Instead, we define a design pattern generically and assume some minimal properties $min\_d_i$ of the $d_i$. These minimal properties are defined by ASMs. This leads to the definition of generic design patterns.

**Definition 10 (Generic Design Pattern).** *A generic design pattern $d(min\_d_1, \ldots, min\_d_n)$ is a design pattern $d = (A, \Sigma, Q)$ where $min\_d_1, \ldots, min\_d_n$ describe minimal requirements on the patterns $d_i$ used by $d$.*

This means that the semantics of $d$ is not defined based on the concrete semantics of the $d_i$ but defines requirements the $d_i$ have to fulfill. The minimal requirements $min\_d_i$ specify the *roles* of the pattern $d$. The bounds can be used to detect errors in the instantiation of a design pattern and to check the consistency of workarounds.

**Definition 11 (Sound and Correct Instantiation).** *An instantiation $d(d_1, \ldots, d_n)$ of a generic design pattern $d(min\_d_1, \ldots, min\_d_n)$ is* sound *if for all $d_i$ used by $d$: $\Sigma_{d_i}^{[\![]} \subseteq \Sigma_{min\_d_i}$. The instantiation is* correct *if it is sound and for all $d_i$ used by $d$: $d_i[\![] \sqsubseteq min\_d_i$.*

Generic patterns $d(min\_d_1, \ldots, min\_d_n)$ can be used to define generic workarounds $d'(min\_d_1, \ldots, min\_d_n)$. The correctness of such implementations can be proven assuming minimal requirements $min\_d_1, \ldots, min\_d_n$.

The following shows the benefit of this construction.

**Theorem 2 (Correctness of Workaround Instantiations).** *If* $d(d_1, \ldots, d_n)$ *is a correct instantiation of the generic pattern* $d(min\_d_1, \ldots, min\_d_n)$ *and* $d'(min\_d_1, \ldots, min\_d_n)$ *is a generic workaround of* $d$ *then* $d'(d_1, \ldots, d_n)$ *is a correct instantiation of* $d'$*.*

*Proof.* Let $d'$ be a generic workaround of $d$. Then the bounds of $d'$ and $d$ are equal. Hence, an instantiation of $d'$ with parameters which define a correct instantiation of $d$ is also correct. □

**Theorem 3 (Simulation of Pattern Instantiations).** *A correct instantiation* $d(d_1, \ldots, d_n)$ *of* $d(min\_d_1, \ldots, min\_d_n)$ *is simulated by* $d'(d_1, \ldots, d_n)$ *if* $d'(min\_d_1, \ldots, min\_d_n)$ *is a generic workaround of* $d(min\_d_1, \ldots, min\_d_n)$*.*

*Proof.* By theorem 2 the instantiation of $d'$ is correct if the instantiation of $d$ was correct. A generic workaround $d'$ simulates a generic pattern $d$ if the components of $d'$ fulfil the minimal requirements of $d$. Therefore, for the same instantiation $d_1, \ldots, d_n$, $d'(d_1, \ldots, d_n)$ simulates $d(d_1, \ldots, d_n)$. □

Generic design patterns together with generic implementations define a structure on the correctness proof for workarounds. Existing correct generic implementations can be reused to define different concrete workarounds for a particular design pattern. In addition, we are able to define and verify intermediate forms where parts of the parameters are instantiated while others are left abstract.

## 7 Conclusions and Outlook

> All animals are equal, but some animals are more equal than others.
>
> George Orwell, Animal Farm

The concepts of design and language patterns are equal. Design patterns can be supported as language concepts directly, whereas language concepts appear as design patterns for other languages which do not directly support them. Therefore, language concepts often start off as design patterns but are later integrated into a programming language as *language patterns*.

This paper introduced the term *workaround* to describe how a pattern that is not directly supported by a language can be implemented on top of provided concepts. In contrast to [9], workarounds describe implementation of higher level concepts in terms of lower level concepts rather than in terms of a concrete programming language.

Additionally, we describe an evolutionary methodology for pattern based language design. The idea of that methodology is to use the workarounds as meta operators in extendable programming languages. This is especially useful to easily integrate new concepts into a language and to experience the advantages and disadvantages of the new features before integrating them into a compiler.

Finally, we defined the notion of syntax and semantics of patterns using abstract syntax trees and *Abstract State Machines (ASM)*. We introduced relationships between patterns – *use*, *simulation* and *extension*. These allow us to define the notion of correctness of workarounds and to build the base for a process of formal language extension.

The next step is to collect a catalogue of language patterns based on top of the listing we provided in the appendix and include their formal semantics and proofed workarounds. Language pattern catalogues would broaden the understanding of programming language concepts and help programmers to adopt these ideas for software design as well as implementation.

Based on the workarounds described within the catalogue, libraries of meta-operations should be implemented. Such libraries could start a new software market for language extensions and help programmers increase their expressiveness in programming. The libraries should be tailored to a specific implementation language and cover a wide range of abstract design patterns and language constructs. All that is needed to make the extension libraries work is a compiler for an open language; the presented proof methodology will provide a formal means to verify them.

Our goal is to provide a simple process for the description of patterns and their integration into programming languages. This may also lead to new compiler architectures with respect to a cost effective language extensibility.

## A Listing of Language Patterns

This listing of language patterns gives an overview over important concepts and sketches the purpose (▷), the central idea of a workaround (↻) and the disadvantages (↯) of the workaround. The very brief description scheme is by no means a complete pattern documentation, but a detailed discussion of every pattern would go far beyond the scope of this paper.

As a basic programming model, we assume a structured, procedural, modular imperative language with user-defined data types. We focus on object-oriented concepts that seem essential to the authors.

Abstract Class (Deferred Class)
- ▷ Model an abstraction of similar objects and define their the common features.
- ↻ Implement Abstract Methods and hide the Constructors to prevent instantiation.

Abstract Method (Deferred Method, Pure Method)
- ▷ Define a method signature and make the implementation a task of Subclasses.
- ↻ Document the abstract method as such and throw an Exception when the method is called.
- ↯ Loss of robustness since exception appears only at runtime.

Assertion (Runtime Check)

▷ Increase robustness and facilitate testing by checking whether the application is in an expected state.

↻ Insert check code and throw an Exception in case of failure. Guard the checks with a debug flag (to be removed by the optimizer) or add them in Decorator classes.

Class

▷ Encapsulate data with operations to modularize a software system.

↻ Define a record with Function Types corresponding to the operations. Add the record as first parameter where needed.

↯ Usually there is no notion of access privacy for structured data types.

Closure (Bound Method)

▷ Enable currying for a Function Type by partially binding parameters changing the function signature.

↻ Apply Function Type, add attributes for bound parameters and Overloading Methods for different signatures.

↯ The bound parameters produce an additional overhead.

Collection (Container)

▷ Model a dynamic 1:N association.

↻ Implement the appropriate abstract datatype using built-in collections such as array types.

Constructor

▷ Ensure that an allocated object becomes initialized immediately.

↻ Apply a monomorphic Factory Method as a Shared Feature and forbid pure allocations (as a convention). Call the appropriate superclass constructor to obtain a pre-initialized object.

Default Parameter

▷ Omit arguments in calls and have them replaced by default values.

↻ Add Convenience Methods defining sensible combinations of the parameters and provide default values in the call to the hook method.

Destructor

▷ Define actions that should take place right before deallocating an object.

↻ Define a method for all objects that is called by the instance responsible for deallocating the object, e.g. a Garbage Collector.

↯ Pure deallocations might still be possible.

Dynamic Method Invocation

▷ Call a method at runtime knowing the name and parameter types but not the type of the object.

↻ Implement an interpreter that performs the method call according to dynamically provided names and types.

↯ The interpreter must be updated when the system changes.

Enumeration

▷ Define a data type from a set of constants.

↻ Define the constants in an Interface Class or provide a special Iterator.

↯ Range checking might be lost.

Exception (Try-and-catch)

▷ Check whether a method execution – or a set of such – succeeded and provide alternative code for the exceptional state.

↻ Pass error codes as additional output parameter, e.g. Multiple Output Parameters, or define global result variables for groups of methods.

↯ If exception handling is not enforced it is often not done.

Flattening and Structuring

▷ Write/read an object structure to/from a file or stream in order to send the structure over a network or to make it persistent.

↻ Define flattener and structurer classes or methods, e.g. pretty printer and parser for abstract syntax trees.

Function Type (Delegate, Higher-Order Function)

▷ Treat functions as types to pass them as parameters.

↻ Use an (explicit) Interface Class to define the signature(s) and pass an object of a concrete subclass.

↯ Possible loss of efficiency due to indirection and memory overhead.

Garbage Collector

▷ Automatically deallocate unneeded objects to provide robust applications.

↻ Implement a memory manager and create a thread to find and deallocate dead data objects.

Generic Class (Template Class, Virtual Class)

▷ Parameterize a class with types to avoid unsafe downcasts.

↻ Use the most general abstraction and create Subclasses that cast to the more special types.

↯ Access to more general types is still possible.

Generic Method

▷ Parameterize a method with types to avoid multiple structurally equivalent declarations.

↻ Expand the types manually, use Overloading Methods if possible.

↯ The workaround does not quite meet the purpose.

Interface Class

▷ Define a behavioral specification for Classes.

↻ Create an Abstract Class with Abstract Methods only.

Interface Implementation

▷ Provide an implementation of an Interface Class.

↻ Declare a Subtype relationship with the Interface Class.

Iterator (Cursor, Stream)

▷ Navigate through the elements of a possibly hidden data structure.

↻ Define operations that have access to the internals of the data structure, e.g. by a Nested Class.

↯ Concurrent iterators may not be possible or encapsulation is broken.

Monitor

▷ Synchronize access to features of a class.

↻ Use Semaphores.

Multi Feature (Multimethod, Multiple Dispatch)
▷ Chose the right implementation of a feature depending on the actual types of the parameters.
↻ Add the feature to all parents and for multimethods, make the other features accessible. Resolve polymorphic calls by cascading single dispatches.

Multiple Inherited Subclass (Multiple Inheritance)
▷ Apply Subclass for a set of superclasses.
↻ Use explicite delegation instead of inheritance.
↯ Loss of performance due to indirection.

Multiple Inherited Subtype (Multiple Inheritance)
▷ Apply Subtype for a set of supertypes.
↯ No workaround possible if Subtype defines an order relation.

Multiple Output Parameters (Multiple Return Parameters)
▷ Implement a multi-valued function.
↻ Define a Convenience Class to bundle the parameter set.
↯ Possible loss of efficiency due to the indirection.

Nested Class (Inner Class)
▷ Give a class opaque access to private features of the outer class.
↻ Define a separate class with appropriate access rights and hold a reference to the outer class when needed.
↯ Appropriate access rights might not be available.

Overloading Method
▷ Allow methods with disjoint signatures to carry the same name.
↻ Make single parameters Subtypes of a new supertype and make the method a Polymorphic Feature. Add dummy parameters or combine parameters to new types to match the number of parameters.
↯ Dynamical dispatch is less efficient than statical.

Package (Class Subsystem, Group, Namespace)
▷ Localize the definitions of a class system and provide a unique name space.
↻ Group the compilation units into different subdirectories and add the package name to each entity name to resolve possible name clashes.

Polymorphic Feature
▷ Access a feature of a set of types instead of a particular type.
↻ Add type tags and dispatch using a function pointer table.
↯ Possible loss of performance due to inefficient dispatch.

Predicate Class
▷ Determine class membership by predicate evaluation at runtime so that objects may migrate through various classes. Method availability and selection depends on the determined actual class.
↻ Define predicates as methods that each object must provide and use the predicates to dynamically check availability or to dispatch methods.
↯ Type safety is lost if predicates can be checked statically.

Property (Virtual Attribute)
▷ Control access to a possibly virtual attribute, e.g. add range checks or notification.
↻ Implement an attribute with access methods and hide the set-method if the attribute should be read-only.

Prototype (Clone)
▷ Avoid expensive re-initialization of new objects and create a type given at runtime only.
↻ Implement a (deep) copy function that can be used to clone an object.

Reflection (Introspection)
▷ Retrieve meta information on an object at runtime, usually for a Dynamic Method Invocation.
↻ Implement a defined set of reflection methods for each class.

Rendezvous
▷ Synchronize two methods at a certain point, e.g. to exchange data.
↻ Use two Semaphores to synchronize the methods in contrary order.

Semaphore
▷ Protect a system from data inconsistency by providing exclusive locks.
↻ Use a lock variable and suspend the current thread if needed.
↯ The implementation depends on a non-interrupteable test-and-set operation.

Shared Feature (Static Feature, Class Feature)
▷ Share common state (shared attribute) or common behaviour (shared method) through all instances of a class.
↻ Apply State for all instances that require access.

Subclass
▷ Reuse the implementation of a class and optionally extend it.
↻ Delegate all the methods to be inherited to a private instance of that class.

Subtype
▷ Make a type a substitute for a (super-)type indicating that it behaves like the supertype, usually for Polymorphic Features.
↻ Often already induced by Subclass or Interface Implementation.
↯ There might be no way to hinder a Subclass to define a Subtype.

Transient Parameter (In/Out Parameter, Var Parameter)
▷ Let a method change the value of an object given as parameter.
↻ Introduce a separated in- and output parameter or define a Convenience Class with a pair of Access Methods.
↯ Possible loss of efficiency due to the indirection.

Value Type (Composite)
▷ Gain efficiency by passing or storing values instead of references.
↻ Extract primitive value types from the compound reference types and resolve name clashes.
↯ The former compound type can no longer be referenced.

## B Introduction to Abstract State Machines

An *abstract state machine* (short: ASM) is a tuple $\mathcal{A} = (\Sigma, Q, S, \rightarrow, I)$, where $\Sigma$ is a signature, $Q$ is a set of $\Sigma$-algebras (the *states*) with the same carrier set, $S$ is a set of sorts (the *super-universe*), $\rightarrow \subset Q \times Q$ is the *transition relation*, and $I \subset Q$ is the set of *initial states*.

$f_q$ denotes the interpretation of $f \in \Sigma$ in state $q \in Q$. Interpretations on $S$ of function names in $\Sigma$ are called *basic functions*. The super universe does not change when the state of $\mathcal{A}$ changes, the basic functions may. The super universe contains distinct elements *true*, *false* and *undef* ($\perp$) that allow to deal with binary relations and partial functions. They do not appear in the signature.

A *universe* $U$ is a special type of basic function: a unary relation identified with the set $\{x : U(x)\}$. Any sort $U \in S$ denotes a universe. The universe $BOOL$ is defined as $BOOL = \{true, false\}$. A function $f : U \rightarrow V$ from an universe $U$ to an universe $V$ is an unary operation on the super universe such that $f(a) \in V$ for all $a \in U$ and $f(a) = \perp$ otherwise. In the ASM model there exists a special universe *reserve* which can be used as a source for new elements.

A term over the signature $\Sigma$ is defined as usual. $\mathcal{T}(\Sigma)$ denotes the set of terms over the signature $\Sigma$. The interpretation of a term $t \in \mathcal{T}(\Sigma)$ in state $q$ is denoted by $[\![t]\!]_q$. The relation $\rightarrow$ is defined by a finite collection of transition rules of the form:

| | | |
|---|---|---|
| **if** `Condition` **then** | | **if** $t_0$ **then** |
| `Updates` | for example | $f(t_1, \ldots, t_n) := t_{n+1}$ |
| **endif** | | **endif** |

where $t_0, t_1, \ldots t_{n+1} \in \mathcal{T}(\Sigma)$ is a transition rule. Let $q$ be the state before and $q'$ be the state after applying the rule. The meaning of the rule is: If $[\![t_0]\!]_q = true$ then for all $g \in \Sigma \setminus f$ $g_{q'} = g_q$, and $f_{q'}$ is defined as follows:

$$f_{q'}(x_1, \ldots, x_n) = \begin{cases} [\![t_{n+1}]\!]_q & \text{if for all } i,\ 1 \leq i \leq n,\ [\![t_i]\!]_q = x_i \\ f_q(x_1, \ldots, x_n) & \text{otherwise} \end{cases}$$

If $[\![t_0]\!]_q = false$ then $f_q = f_{q'}$ for any $f \in \Sigma$. Thus the interpretation changes the value of the basic function $f$ at the value of the tuple $(t_1, \ldots, t_n)$ to the value $t_{n+1}$, provided that $[\![t_0]\!]_q = true$. If several updates contradict then one update is chosen nondeterministically.

In our ASM specification we use the data types *SET*, *LIST*, and natural numbers $\mathbb{N}$ with the usual operations. These data types are assumed to be defined by term algebras. The carrier set of a term algebra represents the corresponding sort of $S$. Additionally, we use some extensions of the basic ASM model.

| Extension | Informal meaning |
|---|---|
| **do forall** $v : g(v)$<br>$R(v)$<br>**enddo** | Let $q$ be the actual state. This rule executes $R$ for each element $v$ with $g_q(v) = true$ in parallel. |
| **extend** $U$ **with** $u$<br>$R$<br>**endextend** | Take an element $u$ of the reserve universe, add $u$ to the universe $U$, and execute the rule $R$. This means that before the execution of the rule $reserve(u) = true$ and $U(u) = false$ and then $reserve(u) = false$ and $U(u) = true$. |
| **let** $x = t$ **in**<br>$R$<br>**endlet** | Bind the term $t$ to the name $x$ in $R$. |

A merge of two ASMs is defined as follows:

**Definition 12 (Merge of ASMs).** *The merge of two ASMs* $a = (\Sigma^a, S^a, A^a, \rightarrow^a, I^a)$ *and* $b = (\Sigma^b, S^b, A^b, \rightarrow^b, I^b)$ *is the ASM* $a \uplus b = (\Sigma^a \cup \Sigma^b, S^a \uplus S^b, A^a \cup A^b, \rightarrow^{a,b}, I^a \uplus I^b)$ *where the merge* $S^a \uplus S^b$ *of a* $\Sigma^a$*-algebra* $S^a = (A, \Sigma^a, Q^a)$ *and a* $\Sigma^b$*-algebra* $S^b = (B, \Sigma^b, Q^b)$ *is defined by* $(A \cup B, \Sigma^a \cup \Sigma^b, Q^a \cup Q^b)$. *The transition relation* $\rightarrow^{a,b} \subseteq (S^a \cup S^b) \times (S^a \cup S^b)$ *is defined by the union of the sets of transition rules defining* $\rightarrow^a$ *and* $\rightarrow^b$*, respectively.*
*The merge of two ASMs* $a$ *and* $b$ *is* valid *if the interpretations of all function symbols defined by* $I^a$ *and* $I^b$ *are consistent. Interpretations of a function* $f : X_1 \times \ldots \times X_k \rightarrow X$ *are consistent in a state* $q$ *if* $\forall\ \overline{x} \in X_1 \times \ldots \times X_k$ : $f_q^a(\overline{x}) \neq \bot \wedge f_q^b(\overline{x}) \neq \bot \Rightarrow f_q^a(\overline{x}) = f_q^b(\overline{x})$.

## References


1. E. Börger and W. Schulte. Programmer Friendly Modular Definition of the Semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS. Springer, 1998.
2. J. Bosch. Design patterns and frameworks: On the issue of language support. *Lecture Notes in Computer Science*, 1357, 1998.
3. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stad. *Pattern-Oriented Software Architecture – A System of Patterns.* John Wiley, 1996.
4. Craig Chambers. The Cecil language specification and rationale: Version 2.0. Available from http://www.cs.washington.edu/research/projects/cecil/www/Papers/cecil-spec.html, December 1995.
5. T. E. Cheatham, Jr., Alice Fischer, and P. Jorrand. On the basis for ELF — an extensible language facility. In *1968 Fall Joint Computer Conference*, volume 33, part two of *AFIPS Conference Proceedings*, pages 937–948, Washington, D. C., 1968. Thompson Book Company.
6. Shigeru Chiba and Takashi Masuda. Designing an Extensible Distributed Language with a Meta-Level Architecture. In O. Nierstrasz, editor, *Proceedings of the ECOOP '93 European Conference on Object-oriented Programming*, LNCS 707, pages 483–502, Kaiserslautern, Germany, July 1993. Springer-Verlag.

7. Ellen Agerbo and Aino Cornils. How to Preserve the Benefits of Design Patterns. *ACM SIGPLAN Notices*, 33(10):134–143, October 1998.
8. Arne K. Frick, Walter Zimmer, and Wolf Zimmermann. On the design of reliable libraries. In R. Ege, M. Singh, and B. Meyer, editors, *TOOLS 17 — Technology of Object-Oriented Programming*, pages 13–23. Prentice Hall, August 1995.
9. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.
10. J. Gil and D. H. Lorenz. Design patterns vs. language design. *Lecture Notes in Computer Science*, 1357, 1998.
11. Gilad Bracha, Martin Odersky, David Stoutamire and Philip Wadler. Making the future safe for the past: Adding genericity to the java programming language. In *Object-Oriented Programming: Systems, Languages, Applications (OOPSLA)*, Vancouver, October 1998. ACM.
12. Gerhard Goos. Sather-k – the language. *Software – Concepts and Tools*, 18:91–109, 1997.
13. Y. Gurevich and J. Huggins. The Semantics of the C Programming Language. In *CSL '92*, volume 702 of *LNCS*, pages 274–308. Springer, 1993.
14. Andreas Heberle, Welf Löwe, and Martin Trapp. Safe Reuse of Source to Intermediate Language Compilations. In Ram Chillarege, editor, *Proceedings of the Ninth International Symposium on Software Reliability Engineering, Fast Abstracts and Industrial Tracks*, 1998.
15. G. Hedin. Language support for design patterns using attribute extension. *Lecture Notes in Computer Science*, 1357, 1998.
16. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, 2 edition, January 1997.
17. Stephan Murer, Stephen Omohundro, David Stoutamire, and Clemens Szyperski. Iteration abstraction in sather. *TOPLAS*, 18(1):1–15, 1996.
18. Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 146–159, Paris, France, 15–17 January 1997.
19. Randall B. Smith and David Ungar. Programming as an experience: The inspiration for self. In Walter Olthoff, editor, *ECOOP '95 - Object-Oriented Programming 9th European Conference, Aarhus, Denmark*, number 952 in Lecture Notes in Computer Science, pages 303–330. Springer-Verlag, New York, N.Y., 1995.
20. Stephen Spainhour, Ellen Siever, and Nathan Patwardhan. *Perl in a Nutshell*. O'Reilly, 1 edition, July 1998. estimated.
21. Bjarne Stroustrup, editor. *The C++ Programming Language*. Addison Wesley, second edition, 93.
22. Michiaki Tatsubori. OpenJava language manual, version 0.2.3, January 1998. http://www.softlab.is.tsukuba.ac.jp/~mich/openjava/.
23. Walter F. Tichy. A catalogue of general-purpose design patterns. In *Proc. Technology of Object-Oriented Languages and Systems (TOOLS 23)*. IEEE Computer Society, 1998.
24. Jan Vitek and R. Nigel Horspool. Compact dispatch tables for dynamically typed object oriented languages. In Tibor Gyimothy, editor, *Compiler Construction, 6th International Conference*, volume 1060 of *Lecture Notes in Computer Science*, pages 309–325, Linköping, Sweden, 24–26 April 1996. Springer.

25. Walter Zimmer. Relationships between Design Patterns. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design*. Addison-Wesley, 1995.
26. Walter Zimmer. *Frameworks und Entwurfsmuster*. PhD thesis, Universität Karlsruhe, February 1997.
27. W. Zimmermann and T. Gaul. On the Construction of Correct Compiler Back-Ends: An ASM Approach. *Journal of Universal Computer Science*, 3(5):504–567, 1997.
28. Wolf Zimmermann. Complexity issues in the design of functional languages. In *Proceedings, International Conference on Computer Languages*, pages 34–43, Oakland, 1992. IEEE Computer Society Press.