On the Design of Reliable Libraries

Arne Frick
Universität Karlsruhe
Institut für Programmstrukturen
und Datenorganisation

Walter Zimmer
Forschungszentrum Informatik
Gruppe Programmstrukturen

Wolf Zimmermann
Universität Karlsruhe
Institut für Programmstrukturen
und Datenorganisation

Abstract

Software libraries are an important instrument for achieving reuse of both designs and programs. This goal is supported by several non-functional properties of libraries. Specifically, a library should be *flexible* by allowing components to be reused in different contexts with possibly changing requirements. However, reuse must not introduce new errors: the library components must remain correct in unknown contexts (*robustness*). Most common object-oriented libraries focus unilaterally on flexibility, neglecting robustness requirements.

This paper investigates the trade-off between flexibility and robustness. We show that the desire for both flexible reuse and robustness can be met. *Design patterns* is identified as an important tool for structuring large object-oriented class libraries.

The concepts discussed in this article have been realized in KARLA, an object-oriented library of algorithms and data structures, that currently contains more than 200 classes, and that is continuously being developed.

Keywords: object-oriented class libraries, robustness, flexibility, efficiency, design patterns

1 Introduction

Software components should be reusable in order to improve development speed and reduce maintenance cost. Most commonly, this is done using *libraries* of components. The focus of this paper is *object-oriented class libraries* [Meyer 88], supposedly providing the largest potential for reuse.

The library designer should consider the functional correctness of its components as well as non-functional properties such as *robustness*, *flexibility*, and *efficiency*. These terms will be defined in section 3. As the focus of this paper is the construction of robust libraries, we propose methods and techniques to achieve flexibility and efficiency while maintaining the robustness property.

The remainder of this paper is organized as follows: section 4 discusses several definitions of inheritance based on the behavioral specification of classes with respect to robustness. Section 5 considers instantiations of generic classes. We show that for the constructions of libraries the possible instantiations must be restricted in order to avoid errors. The motivation for section 6 is the exchange of class implementations and method implementations at run-time. We show that this goal can be achieved by using and extending adequate *design patterns*. These are discussed from the viewpoint of library users as well as library designers. In section 7, other libraries are evaluated according to the requirements of robustness, flexibility, and efficiency. We conclude with our experience on using the techniques and methods of this article for the construction of the library of algorithms and data structures KARLA. The appendix contains an example used throughout the paper. This example is defined in the programming language SATHER-K, whose most important features are also explained in the appendix.

2 Notation

An *abstract class* is a class that cannot be instantiated, because for at least one method, there exists only an interface description. Every non-abstract class is called *concrete*.

A class B inherits from class A iff definitions of A are visible in B. In this case, B is a *subclass* of A. By *polymorphism* we mean that at a program point requiring an object of type A, an object of any of its subclasses may be used instead. Following the semantics of SATHER-K, we identify the notion of class and the notion of type. SATHER-K specific notation will be explained in the appendix.

3 Robustness, Flexibility, and Efficiency

In this section, we define the notions of robustness, flexibility and efficiency. We assume that a class is completely specified by both an invariant and the pre- and postconditions for each of its methods. A class A is called *locally correct* iff

(i) after creation of an object of class A, the invariant Inv_A holds, and

(ii) after calling a method m of an object of class A, the invariant Inv_A and the postcondition $Post_{m,A}$ of m hold, provided that the invariant Inv_A and the precondition $Pre_{m,A}$ of m are satisfied when calling m.

Any object of class A must satisfy Inv_A at any time except possibly during execution of a method of A. The local correctness is the only notion of correctness that can be provided by library designers. We now extend the notion of local correctness by the object-oriented concepts of inheritance, polymorphism, and genericity.

A class library is *correct* iff each class is locally correct – even if it is inherited by other classes, used polymorphically, or used as an instantiation of a generic parameter.[1] These cases are discussed in sections 4 and 5, where we first analyze different notions of inheritance according to their robustness under polymorphism, and then discuss how robustness can be maintained when using generic parameters.

Even if the correctness of the library is proven, programmers might still use it in a wrong way. For example, they might call a method before establishing its precondition, or that the requirements introduced in sections 4 and 5 are not satisfied. In this case, the worst thing that could happen is that an error message points deeply in the calling hierarchy of the library, or even that the running program is interrupted without any error message. Instead, programmers must be made aware of their mistake. Therefore, an error message should contain information on the violated precondition. Of course, this means that checks and balances have to be built into the library to be able to identify the cause of the misuse. A class library with these properties is called *robust*. A library that is both robust and correct is called *reliable*. Reliability is the most important property of a library, because an unreliable library is simply not usable. The high reliability of FORTRAN libraries such as LAPACK or BLAS may well have been the determining factor for their success.

It should be possible for the programmer to exchange implementations behind interfaces easily when the requirements (especially the non-functional ones) change.[2] This should not cause a complete redesign. We analyze the conditions under which this *flexibility* can be provided by a class library without violating its correctness and robustness properties.

Finally, implementations of classes should be as *efficient* as possible with respect to time and space. We will show that requiring maximal flexibility of a robust library may allow only inefficient implementations for a class or method.[3] Hence, the library should provide flexible, as well as efficient, implementations for a class. The decision of using the flexible or efficient implementation variant is then left to the user.

4 Inheritance Relations

The notion of inheritance may be defined in several ways. Using a formal class specification, there are four different possibilities. This section explores how polymorphism, based on the different possibilities for defining the notion of subclass, affects the robustness requirement. There is only minimal related work [Liskov 94, Schmidt 92], and they only consider conformance[4].

The four possible inheritance relations are shown in figure 1. We discuss these in turn w.r.t. the criteria robustness, encapsulation, flexibility, and practical use.

More formally, we say a class B *conforms to* a class A iff the following implications hold (figure 1a):

(i) $Inv_B \Rightarrow Inv_A$

(ii) For each method m of A: $Pre_{m,A} \Rightarrow Pre_{m,B}$ and $Post_{m,B} \Rightarrow Post_{m,A}$.

A class B is *more special than* a class A iff the following implications hold (figure 1b):

(i) $Inv_B \Rightarrow Inv_A$

(ii) For each method m of A: $Pre_{m,B} \Rightarrow Pre_{m,A}$ and $Post_{m,A} \Rightarrow Post_{m,B}$.

A class B is *covariant to* a class A iff the following implications hold (figure 1c):

(i) $Inv_B \Rightarrow Inv_A$

(ii) For each method m of A: $Pre_{m,B} \Rightarrow Pre_{m,A}$ and $Post_{m,B} \Rightarrow Post_{m,A}$.

The specialization in [Liskov 94] requires the equivalence of the preconditions. Therefore, their notion is a special case of both conformance and covariance.

A class B is *contravariant to* a class A iff the following implications hold (figure 1d):

(i) $Inv_B \Rightarrow Inv_A$

[1] The question of correctness under overloading is not discussed here for the sake of brevity.

[2] Note that the FORTRAN libraries just mentioned are *not* flexible in this sense, as a library routine can only be used for one specific problem.

[3] In the extreme case, this might go as far as to only allow for exponential-time implementations of methods.

[4] [Liskov 94] additionally defines a "specialization". Thorough analysis reveals that this is but a special case of conformance. Furthermore, their correctness proofs require the existence of execution histories of objects, leading to rather complicated proofs. This article waives this requirement.

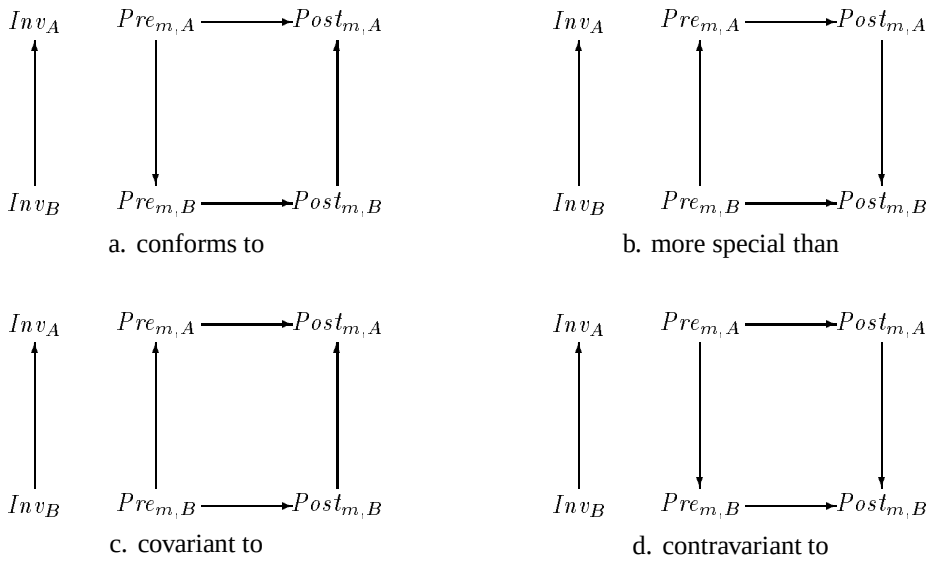

Figure 1: Different Inheritance Relations

(ii) For each method m of A: $Pre_{m,A} \Rightarrow Pre_{m,B}$ and $Post_{m,A} \Rightarrow Post_{m,B}$.

The actual type of polymorphic objects may be determined as late as run-time, which might lead to violations of robustness if the actual type does not match the declared type. Robustness for polymorphic objects can only be guaranteed if the definition of inheritance is based on conformance, as the following argument shows. The definition of conformance assures the correctness of polymorphic variable use, because pre- and postconditions fit correctly. Assume that method m of a polymorphic object of type A is called, and that the actual type of the object is B. We need to assure that before the call, $pre_{m,B}$ holds, which is in fact the case, assuming that $pre_{m,A}$ was true. Therefore, we may safely assume that the call to m is legal. After the call, we know that $post_{m,B}$ holds due to the assumed correctness of the implementation of m in B. By definition, this logically implies $post_{m,A}$, which is what we needed to show.

The remaining definitions of inheritance are not robust against polymorphism. In order to ensure correctness, the actual type of an object must be known. This naturally leads to the question of whether the other definitions of inheritance are irrelevant. It turns out that this is not the case, because they potentially allow for the reuse of code. We demonstrate this for the case of specialization.

Example 1: Consider the example in the appendix. The robust modeling of directed graphs requires that directed acyclic graphs (*dag*s) be a specialization of directed graphs. It is easy to see that not all edges which can be added to a graph can also be added to a dag. Only such edges are allowed which do not violate the acyclicity constraint. Other examples are data structures (such as queues, lists, stacks, etc.) of bounded size, which are more special than their unbounded versions, because the insertion of items may raise an overflow on the size. A third example comes from relational databases. Integrity constraints and normal forms of relations lead to specializations.

These examples show the usefulness of the specialization. A robust class library should therefore provide at least conformance and specialization. From a library designer's viewpoint, specializations can be only considered in terms of their potential for reuse, excluding polymorphism for robustness reasons. However, the information on the inheritance relation between classes may help library users to design their programs. As we already saw, only conformance leads to a robust notion of polymorphism. We now discuss the remaining three inheritance relations from a library user's point of view.

The following scenario is assumed. The library user designed a program where they use an object x of class A. Suppose now that, during the design, they discover that x would be better an object of class B instead (possibly due to incomplete requirement specifications). In this case, knowledge of the kind of inheritance relationship between A and B can help to avoid a complete redesign.

1. A is more special than B. In this case, reuse is possible if the correctness proof can be done with the weaker invariant Inv_B.

2. B is more special than A. It suffices to check whether the correctness proof can be done with the stronger preconditions $Pre_{m,B}$ and the weaker postconditions $Post_{m,B}$.

3. B is covariant to A. Reuse is only possible if the correctness proof can be done with the stronger preconditions $Pre_{m,B}$.

4. B is contravariant to A. Reuse is only possible if the correctness proof can be done with the weaker postconditions $Post_{m,B}$.

For the first case, all locations must be considered where x is used. The remaining cases require only the consideration of method calls on x. If the correctness proof cannot be performed at some location, additional code has to be inserted at that location in order to enable the proof to obtain. This may include checks for illegal conditions. The last two cases (A contravariant to B and A covariant to B) require the analysis of locations where x is just used *and* of locations where a method of x is called. This usually leads to a complete redesign.

Summarizing the results of this section, we observed that the conformance relation is the only notion of inheritance which is robust under polymorphism. The other three possible inheritance relations allow for the reuse of program code and design. A robust class library should therefore support each of the four inheritance relations to exploit the full potential of reuse. In practice, however, we have only observed conformance and specialization relationships.

5 Generic Classes

Genericity is mainly used for the construction of *container classes* $C(T)$. When instantiating a container class, the *type parameter* T is replaced by an arbitrary type (*unbounded genericity*). The definition of $C(T)$ is independent of T. Under robustness requirements, neither the definition of $C(T)$ nor any implementation of $C(T)$ is allowed to make any assumptions on the structure of T. Otherwise, the programmer might instantiate T with a type violating the assumptions of the library designer. This faulty instantiation leads to compile-time or run-time errors, a situation to be avoided.

Requiring robust unbounded genericity restricts reuse potential. For example, it is impossible to sort lists unless it is assumed that any instance of the type parameter T has a relation $<$ which totally orders the objects of T. A robust library should therefore in addition offer such restricted instantiations (*bounded genericity*).

Example 2: The parameter *EDGE* in the generic class of graphs is restricted to instantiations which conform to *EDGETYPE*.

A restriction on the instantiations of a generic parameter T of a container class $C(T)$ is defined by a type bound B. The definition of $C(T)$ and any implementation of $C(T)$ is allowed to make use of the properties specified in B. T can only be instantiated with types having at least the properties of B. From the discussion in the last section we get an elegant way to express type bounds. The restriction B must contain minimum properties to be satisfied by any instance of T and must provide some methods including pre- and postconditions. The most easy way to do this is to define the type bound B itself as a class and restrict the possible instances of T to classes which conform to B. We denote this restriction by $C(T < B)$. The correctness proof of an implementation of $C(T < B)$ can then use Inv_B, $Pre_{m,B}$, and $Post_{m,B}$ for any method m of B.

6 Design Patterns

An *object-oriented design pattern* typically consists of several collaborating classes which represent a reusable, well-proven solution scheme to a recurring object-oriented design problem. The first significant collection of object-oriented design patterns has recently been published [Gamma 94].

This section investigates how some of the object-oriented design patterns defined in [Gamma 94] may be used to address the different issues of reliability, flexibility, and efficiency. As there are trade-offs between these requirements, and as [Gamma 94] focuses just on flexibility, the design patterns are extended such that library users can make their own choice between flexible and efficient implementations. These extensions and their usage are exemplified for the design pattern *bridge*.

6.1 Strategy

According to [Gamma 94], a *strategy* defines a family of algorithms solving the same problem and makes them interchangeable. This can be achieved by defining an abstract class P for the problem solved by the different algorithms and by defining for each algorithm a its own class A which must be a conformant subclass of P. If one has now a polymorphic *algorithm object* x of type P, then an algorithm can be exchanged by creating an object of the desired *algorithm class* and assigning it to x. If these algorithm classes are generic parameters of a data structure, then we obtain a static variant of a strategy. As we combine strategies with bridges, we refer to the next subsection for more concrete examples.

6.2 Bridge

The design pattern *bridge* separates the interface of a class from its implementation. A bridge for an abstract class A allows the dynamic exchange of the implementation.

Example 3: Consider the example in the appendix. A bridge for the class $D\_GRAPH(VERTEX, EDGE)$ allows the replacement of a set implementation of an object $x : \$D\_GRAPH(VERTEX, EDGE)$ by an implementation based on adjacency lists or adjacency matrices.

A bridge for A can be realized by a concrete class A_b, which must be a conformant subclass of A. A_b contains an attribute $repr : \$A$ which refers to the current implementation. Method calls m on objects of class A_b are delegated to $repr$. Replacing the current representation by a new implementation A_i of A is just an assignment of the A_i object to $repr$. In order to maintain the abstract state of

A-objects, a method $change\_repr$ is provided, which performs a consistent representation change not visible to the user of A_b.

Observe that this representation change exchanges a class – including all its methods. For library users this may not be sufficient. A more flexible policy would allow to exchange not only implementations of abstract classes as a whole, but also particular methods of abstract classes. The dynamic exchange of algorithms can be realized by combining the design pattern *bridge* with *Strategy* into a so-called *dynamic bridge*. Dynamic Bridges have, in addition to the attribute $repr$, an algorithm attribute $m_a : \$M_a$ for each method m of A. Any call of m is delegated to m_a. The class M_a defines an arbitrary algorithm implementing m, e.g. M_a might delegate to the method m in the current implementation of $repr$.

This solution is very general. In fact, in order to construct robust libraries it is even too general, as demonstrated by the following example:

Example 4: In our example in the appendix, methods such as *are_connected* and *is_acyclic* can be implemented in the abstract class $D\_GRAPH(VERTEX, EDGE)$. We call this kind of method *derived*. According to the definitions of section 2 this is not possible for each method of an abstract class, otherwise it would not be abstract. Examples are the methods *init*, V, E, *addvertex*, *addedge*, *delvertex*, and *deledge*. These methods are connected to the concrete implementations of $D\_GRAPH(VERTEX, EDGE)$, such that they cannot be implemented there. We call such methods *kernel methods*.

Since abstract implementations of methods cannot make use of the concrete representations of objects, they are usually inefficient. Hence, changing the implementation of such methods at run-time is important, especially under efficiency criteria. This is possible with dynamic bridges.

Figure 2 realizes such a bridge for an abstract class A. It shows the treatment of a kernel method $k : SAME$ and a derived method $m : T$. Furthermore, the most *general* strategy M_a for m is shown there. Suppose now that the current implementation of m has to be replaced at run-time by a new algorithm implementing m. This algorithm must be defined in a subclass N of M_a. The replacement can then be realized by the assignment $m_a := \#N$, which assigns a new object of class N to m_a. Observe that the initialization $init$ of an object of A_b just chooses an arbitrary implementation of A and the algorithm for m in that implementation. If m has parameters, they are added in the definition of m in M_a. If A has generic parameters, then A_b and M_a have the same parameters (including type bounds).

For robustness reasons, in a dynamic bridge not all possible algorithms implementing m can be used.

Example 5: An algorithm for finding all shortest paths of a graph is based on successively squaring its adjacency matrix [Cormen 89, Chapter 26.1]. This algorithm must not be used if the current implementation of graphs is based on the set implementation in the appendix, because the algorithm relies on the adjacency matrix implementation of graphs.

```
(1)  class A_b is
(2)  subtype of A
(3)  private repr : $A
(4)  m_a : $M_a;
(5)  ...
(6)  init : SAME is
(7)      repr := #A_i.init;
(8)      m_a := #M_a;
(9)      ...
(10)     res := self;
(11) end;
(12) ...
(13) k : SAME is
(14)     repr := repr.k;
(15) end;
(16) ...
(17) m : T is
(18)     res := m_a.m(repr);
(19) end;
(20) ...
(21) end;
(22)
(23) class M_a is
(24)     m(repr : $A) : T is
(25)         res := repr.m;
(26)     end;
(27) end;
```

Figure 2: Realization of a Dynamic Bridge of Class A

We therefore distinguish *representation-dependent* from *representation-independent* algorithms. In general, representation-dependent algorithms must not be used if the current representation is the wrong one. As the library designer cannot predict anything about the current representation of an abstract class in a dynamic bridge (it is possible to change the representation at any time), we allow only use of representation-independent algorithms. In figure 2 this is achieved by the fact that only representation-independent algorithms can be defined by algorithm classes which are subtypes of M_a.

Unfortunately, the most efficient algorithms implementing a method are often representation-dependent. In order to use these algorithms, the concrete representation must be known statically. We define therefore a *static bridge*. In contrast to the dynamic bridge, the representation now becomes a generic parameter of the bridge. It is also a generic parameter of the class M_a. Lines (1)–(11) in figure 2 are therefore replaced by

```
(1)  class A_b(REPR < A) is
(2)  subtype of A
(3)  private repr : REPR
(4)  m_a : $M_a(REPR;
(5)  ...
(6)  init : SAME is
(7)      repr := #REPR.init;
(8)      m_a := #M_a(REPR);
(9)      ...
(10)     res := self;
(11) end;
```

A static bridge differs from a dynamic bridge only in the class definition, in the definition of the bridge related attributes, and in the initialization.

If an algorithm for m is based on a specific implementation A_i of A, then the corresponding algorithm class is defined as a subtype of $M_a(A_i)$. We demonstrate this by continuing the example of the shortest paths:

Example 6:
A static bridge for $D\_GRAPH(VERTEX, EDGE)$ defines an attribute

$$s\_path : \$S\_PATH(VERTEX, EDGE, REPR).$$

The call of s_path for the computation of all shortest path is delegated to the method s_path in this object, which might be representation-dependent. If for example the representation is based on adjacency matrices, then the above implementation for computing all shortest paths must be a subclass of

S_PATH(VERTEX, EDGE,
REPR<D_GRAPH_AM(VERTEX,EDGE))

where $D\_GRAPH\_AM(VERTEX, EDGE)$ is the implementation class of $D\_GRAPH(VERTEX, EDGE)$ based on adjacency matrices.

To summarize the observations of this subsection, dynamic bridges for a class A allow the dynamic exchange of implementations of A as well as algorithms implementing particular methods. These algorithms, however have to be representation-independent. In contrast, a static bridge for A also allows representation-dependent algorithms. This however is payed for by the non-interchangeability of the implementation of A.

6.3 Decorator

The design pattern *Decorator* can be used to dynamically extend the functionality or properties of an object.

Technically, these goals are achieved by "decorator objects" (class D) which enclose an object (class O) in order to extend its properties. The decorator object delegates requests to the original object and may perform its own functions before or after delegation. Thereby, the decorator class D and the class O are subclasses of a common abstraction (class C). As in [Gamma 94] shown, the interfaces of D and O must conform to that of C. But concerning reliability and flexibility, there are more points to take into account.

Conformance between classes O, D, and C on the semantic level (conformance as defined in section 4), i.e. not only at the interface level, is essential, as illustrated by the following example.

Example 7: A bounded graph is a graph with a bound on the number of vertices. A decorator for this extension is the class $BOUNDED$, which must be a subclass of D_GRAPH. It decorates any implementation of D_GRAPH. The interface does not change at all. According to [Gamma 94] this should be a valid decoration. However, bounded graphs are specializations of unbounded graphs. This can be seen for example at the method $addvertex$. The precondition is strengthened, because if the bounded graph already has the maximum number of vertices an overflow may occur. A decorated object cannot always be used in place of an undecorated object.

A static version of decorators can be realized with inheritance. The disadvantage is an explosion of classes with less flexibility but high reliability. Currently we are investigating for KARLA an automatic generation of decorated classes. A configuration management system should help the user of class libraries to find the appropriate classes. Therefore, disadvantages such as the class explosion could be removed.

6.4 Adapter

Adapter is a design pattern for adapting a given interface I_1 to another interface I_2. Adapters can be used for integrating class libraries: the interface of one library is adapted to interfaces of abstract classes of another library. This allows the use and a common view of many complex algorithms and data structures of other libraries.

Furthermore, adapters can be used for the implementation of bridges. A specific implementation can be used in different bridges, if the interface of this implementation is adapted to the interface of the abstract class of the bridge. For example, a list implementation can be used for abstract data structures such as *List*, *Mapping*, *Set*, *Bag*, etc. If the implementation variants conform to the abstract class then this combination of adapters and bridges is reliable.

6.5 Summary and Evaluation

A flexible library should provide mechanisms for combining its classes. An important prerequisite for this is the provision well-defined interfaces for the abstract classes. If the subclasses conform to the abstract classes, then they may be combined in the same way that the abstract classes can be combined. This principle is applied in many design patterns.

It is difficult to find such standard interfaces for the abstract classes. Sometimes, concrete classes support the interface of the abstract class, but they can not provide the functionality behind each method interface, or the behavior is implemented but it does not conform to the pre- and postcondition given in the abstract class. In this case,

the conformance relationship between a concrete class and an abstract class is violated and interchangeability is no longer given. But this means that the prerequisite for combining these concrete classes is not satisfied and that the combination of components may cause errors, i.e. the standardization process has been performed too rigorously, thus breaking reliability in support of flexibility.

This trade-off between flexibility and few standardized interfaces on the one side, and reliability on the other side, should be always considered when designing a library. The priority (flexibility or reliability) depends on the type of application, how often it is reused, how much flexibility or reliability is needed, and so on.

The static variants of design patterns introduced in this paper allow algorithms to exploit representation features in order to gain efficiency, while retaining reliability. The flexibility of the combination of classes at run-time, for example, is partially lost, but it is possible to regain large parts of this flexibility by an appropriate combination of static and dynamic bridges.

The design patterns in [Gamma 94] focus on flexibility, losing some of the reliability. KARLA was designed with much more attention on reliability.

7 Other Class Libraries

This section investigates existing object-oriented libraries with regard to the following questions:

- How are the often conflicting issues – reliability vs. flexibility vs. efficiency – addressed? What concepts and techniques are provided to support the different aspects?

- What role does inheritance play in the structure of the library? What purposes is it used for and what are the consequences of this use?

Eiffel

The Eiffel language [Meyer 92a, Meyer 92 b] supports reliability by allowing for pre- and post-conditions of methods and for class invariants.

As Casais [Casais 92a] points out, (multiple) inheritance is used for a variety of purposes throughout the Eiffel library: equality and extension (both are very closely related to subtyping), restriction and code sharing (close to specialization), and a mix of these inheritance relationships.

For this reason, there is no clear separation between interface and implementation, and the exchange of the implementation of a data structure or algorithm is not easy. The resulting library structure is quite complex, so that the advantages of standardizing interfaces seem to be lost to a great degree.

The goal of the refactoring operations (decomposing and factorization) performed in [Casais 92a] was to use inheritance only for specialization and refinement relationships. Similar to the discussion in section 4, specialization and refinement are considered as the two main applications of inheritance (in Eiffel they cannot be distinguished explicitly).

LEDA

LEDA [Näher 92, Mehlhorn 95] is short for *Library of Efficient Data Structures and Algorithms*. It is implemented in C++ and focuses on data structures (basic data types, dictionaries, priority queues, graphs, computational geometry) and efficient algorithms.

Efficiency is addressed by the fact that all data types are implemented by the asymptotically most efficient implementation known. The parameterization of a data type (interface) with an implementation variant is possible, but only at compile-time, because the parameterization is performed by multiple inheritance, a static relationship (static bridge). LEDA supports the so-called *item-concept*, i.e. it provides the user with a safe abstraction of a position in a data structure to gain efficient access to such a position.

LEDA's inheritance structure is flat, as inheritance is not used to internally structure the library. Therefore, reuse by derived features cannot be achieved. Furthermore, the comprehensibility is affected. When parameterizing data types such as dictionaries with key and information types, there are no restrictions on these types (e.g. KARLA achieves this by bounded genericity), thus violating robustness of the library structure.

Although pre- and postconditions are used for documentation purposes, they are not checked at run-time, another violation of robustness. The latest version of LEDA provides several implementation variants for some of its data types, but the needed implementation variant can only be chosen at compile-time (static bridge), not at run-time.

Overall, LEDA provides implementations for the most efficient data types currently known, but it lacks reliability and flexibility in several points.

Smalltalk

As Smalltalk [Goldberg 83] is a dynamically-typed language, it naturally supports flexibility, but there is a loss in efficiency and especially in reliability. In [Cook 92], the Smalltalk-80 collection class library was analyzed, and a *clean* hierarchy of interfaces was developed from the inheritance structure. Several weaknesses of the original library were revealed as a result of this analysis.

In particular, inheritance is used for reuse of interfaces, of code, or a mix of both. Methods are often deleted by subclasses, or methods of subclasses violate the invariants given by superclasses. The need for specifications and interfaces when designing libraries was clearly demonstrated by Cook. Furthermore, he proposed that inheritance should be used only for conformance (refinement) and other techniques (e.g. multiple inheritance) should be provided to reuse code.

The flexibility of Smalltalk often leads to an undisciplined use of mechanisms such as inheritance, which also

affects the reliability of applications. Efficiency is still a problem for time-critical applications.

Coherent Design

[Rüping 93] discusses how the different requirements of extensibility, comprehensibility, and efficiency can be reconciled. That paper develops a data structure catalogue and applies the design principles *separation* of design and implementation, *imitation* relationships between classes (close to conformance), and *conceptual abstraction* to support these requirements.

Thus, the clean design principles, especially the disciplined usage of inheritance for imitation, and techniques such as kernel/derived features, items (see the discussion of LEDA), pre- and postconditions and class invariants lead to a high rate of internal and external reuse and to good comprehensibility.

The concept of *imitation* employed by the authors is a weaker form of compatibility than conformance. Imitation allows for the substitution of *all* occurences of B by C in a class definition A iff C imitates B. The Coherent Design methodology has no notion of specialization, and it is not possible to substitute only selected occurences of B. As we have shown in this paper, both are important concepts to achieve a high degree of flexibility and reuse. Our work can therefore be viewed as an extension of Coherent Design.

If the user does not comply with this guideline then type errors can occur, implying that the reliability criterion is violated. The massive use of multiple inheritance affects the flexibility of a library (no flexible bridges) and is considered as a major weakness by many.

OBST

The object-oriented database system OBST [Casais 92b] comes with a small, but useful library of data structures such as sets, bags, mappings, and the like. The clear separation between interface and implementation allows for the interchange of implementation variants.

By introducing the *bridge* design pattern, the two implementation variants *List* and *Hashtable* of the data type *Mapping* can be interchanged at run-time, either automatically (controlled by the data type itself) or under user control. The flexibility of choosing appropriate implementation variants for different applications resulted in a gain of efficiency in time and space. Of course, some efficiency is lost by having an additional indirection inside method calls on a container object.

Karla

During the design of KARLA, the *Kar*lsruhe *L*ibrary of *A*lgorithms and Data Structures, written in SATHER-K, major effort was put into the support of reliability. To this end, both conformance and specialization relationships are supported by the library.

As the programming language does not provide both relations, user support is given by means of a configuration and information tool called *skit*, that allows for the automatic generation of static and dynamic bridges, given an arbitrary abstract class. In addition, property classes are used as type bounds for the element type of container classes, effectively supporting compile-time checks on the validity of the element type.

Summary

Summarizing the discussion, we find that each of the above libraries has a characteristic way of supporting the design goals of reliability, flexibility, and efficiency.

LEDA focuses on very efficient components; Smalltalk offers great flexibility; the OBST data structure library is small, but flexible; the Eiffel libraries offer a large functionality; the data structure catalogue, built according to design principles of *coherent design*, strongly supports reuse and extensibility. For KARLA, the reliability of its components is very important.

Unfortunately, most libraries neglect the reliability requirement for the most part. With the growing number of safety-critical software, the significance of this aspect will steadily increase, thus requiring more effort to address it. By continuing to develop KARLA, we hope to contribute to this effort.

8 Conclusions

We discussed several techniques for the construction of reliable class libraries. To this end, we first distinguished between abstract and concrete classes, leading to a separation of interface and implementation. We showed that besides the *use*-relation, at least two kinds of relationship between classes are necessary to achieve both flexibility and efficiency. We defined these relationships – *conformance* and *specialization* – formally in terms of class invariants and pre- and postconditions. Conformance allows for maximal flexibility in the use of classes (i.e. any class conforming to a class A can be used at any time instead of A). Specialization allows for the reuse of code and opens up possibilities for efficiency gains by using representation-dependent algorithms. We showed that in existing class libraries, these relations are either mixed up, or only one of them is supported. One of the major contributions of this paper is that these libraries are inherently unreliable, or they single-sidedly support either flexibility or efficiency.

The tradeoff between flexibility and efficiency was examined in order to come up with conditions for the reliable, yet flexible and efficient use of a library. The goal was to enable the exchange of the representation of an abstract class A as well as the change of any algorithm implementing features of A, even at run-time. To this end, we extended the *dynamic bridge* pattern by combining it with *Adapter* and *Strategy*. We showed that a reliable use of dynamic bridges requires that only algorithms can be used which do not rely on a particular representation. On the other hand, efficient algorithms often rely on a certain representation be used. This is supported by the static version

of *Bridge*, in which representation-dependent algorithms may be used. The existence of both bridge types gives the library user the choice between flexibility and efficiency, depending on the type of bridge he uses.

This paper builds on our experience with building KARLA, the KARlsruhe Library of Algorithms and data structures. The first version of KARLA provided only for conformance between classes. Consequently, programmers started to use conformance to reuse code. With the lack of specialization, this led to an incomprehensible and unnecessarily complicated library structure. After realizing this situation, we had to rebuild large parts of KARLA. The good news is that large-scale reorganization has not been necessary since we introduced specialization relationships. KARLA currently contains over 200 classes. Among them there are approximately 80 abstract classes that are related by specialization and conformance. Decorators allow for the addition of properties to objects of a class. In container types, these additional properties may also be properties of the element type, being expressed as type bounds using so-called *property classes*. We observed for KARLA that, until now, the former leads to specialization, the latter to conformance relationships.

Our experiments with KARLA started a year ago, when we let students use and enhance the library. We found that the students were quickly able to determine conformance and specialization relationships between classes. Furthermore, they use this information to find the library classes they need, and to construct bridges using the *skit* configuration tool. This shows that the concepts of conformance and specialization are quite natural and useful.

Our experience showed the usefulness of the concepts proposed in this paper. An important open question is whether purely *covariant* or *contravariant* relationships of section 4 arise in practice.

Acknowledgements

The authors thank Gerhard Goos for the inspiration to this work and the anonymous referees for their helpful comments. Richard Walker and Achim Weisbrod proof-read previous versions of this paper. Of course, all remaining errors our purely ours.

References

[Casais 92a] Eduardo Casais. An incremental class reorganization approach. In O. Lehrmann Madsen, editor, *Proceedings of ECOOP '92*, LNCS 615, pages 114–132, Utrecht, The Netherlands, July 1992. Springer-Verlag.

[Casais 92b] Eduardo Casais, Michael Ranft, Bernhard Schiefer, Dietmar Theobald, and Walter Zimmer. OBST - An Overview. Technical report, Forschungszentrum Informatik (FZI), Karlsruhe, Germany, June 1992. FZI.039.1.

[Cook 92] William R. Cook. Interfaces and Specifications for the Smalltalk-80 Collection Classes. In *Proceedings of OOPSLA '92, ACM SIGPLAN Notices*, pages 1–15, October 1992. Published as Proceedings of OOPSLA '92, ACM SIGPLAN Notices, volume 27, number 10.

[Cormen 89] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1989.

[Gamma 94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Software Components*. Addison-Wesley, 1994.

[Goldberg 83] A. Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation*. Addison-Wesley, May 1983.

[Näher 92] S. Näher and K. Mehlhorn. *Algorithms, Software, Architectures, Information Processing 92*, volume 1, chapter Algorithm Design and Software Libraries: Recent Developments in the LEDA Project. Elsevier Science Publishers B.V., 1992.

[Liskov 94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.

[Mehlhorn 95] Kurt Mehlhorn and Stefan Näher. LEDA – A Platform for Combinatorial and Geometric Computing. *Communications of the ACM*, 38(1):96–102, January 1995.

[Meyer 88] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

[Meyer 92a] Bertrand Meyer. Applying "Design by Contract". *IEEE Computer (Special Issue on Inheritance & Classification)*, 25(10):40–52, October 1992.

[Meyer 92 b] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[Rüping 93] Andreas Rüping, Franz Weber, and Walter Zimmer. Demonstrating coherent design: A data structure catalogue. In Raimund Ege, Madhu Singh, and Bertrand Meyer, editors, *Proceedings of the 11th TOOLS Conference*, pages 363–377. Prentice Hall, 1993.

[Schmidt 92] Heinz W. Schmidt and Richard Walker. TOF: An efficient type system for objects and functions. Technical Report TR-CS-92-17. Department of Computer Science, The Australian National University, November 1992.

A An Example: Graphs

This appendix contains the *graph* example we use throughout this paper. It illustrates our ideas and concepts. Although the example is written in the imperative object-oriented programming language SATHER-K, it suffices to know that it is an EIFFEL-like language. The main difference between EIFFEL and SATHER-K is the separation of polymorphic and monomorphic types, denoted by a '$' prefix for polymorphic types in SATHER-K. The **subtype of** keyword denotes conformant subtyping, and polymorphism is only possible for subclasses. The **specialization of** keyword denotes a specialization relationship.[5] SATHER-K has type bounds. Class definitions beginning with the keyword **abstract** are abstract in the sense of section 2 and serve solely as interface descriptions. The class invariant has to hold after object creation and calling a special $init$ method which every class has to support.

We assume that a class $SET(T)$ is given, which contains the usual methods for set creation, element insertion, deletion, and membership ($\in$). In addition, we assume that there are methods for set intersection ($\cap$), union ($\cup$), and difference ($\setminus$). The state of an object after the call of a method is denoted by the suffix ''. Edges are assumed to have two attributes $from$ and to to characterize their incident vertices. This gives the following type-bound for edges, implying that any edge type has to define at least these two attributes.

```
(1) class EDGETYPE(VERTEX) is
(2)     from, to : VERTEX;
(3) end
```

Equality of edges is defined as the equality of the incident vertices. Instead of $e.from$ and $e.to$, we write e_1 and e_2 for simplicity.

```
(1)  abstract class D_GRAPH(
(2)      VERTEX,
(3)      EDGE < EDGETYPE(VERTEX)) is
(4)  -- defines directed graphs;
(5)  -- Invariant: ∀e ∈ E : e1, e2 ∈ V;
(6)
(7)  V : $SET(VERTEX) is deferred;
(8)      -- returns the set of vertices
(9)
(10) E : $SET(EDGE) is deferred;
(11)     -- returns the set of edges
(12)
(13) init : SAME is deferred
(14)     -- returns an empty graph;
(15)     -- Precondition: true
(16)     -- Postcondition: res.V.empty ∧ res.E.empty
(17)
(18) addvertex(v : VERTEX) : SAME is deferred
(19)     -- inserts a vertex;
(20)     -- Precondition: true
(21)     -- Postcondition: res.V = V.ins(v) ∧ res.E = E
(22)
(23) addedge(e : EDGE) : SAME is deferred
(24)     -- inserts an edge;
(25)     -- Precondition: e1 ∈ V ∧ e2 ∈ V
(26)     -- Postcondition: res.V = V ∧ res.E = E.ins(e)
(27)
(28) delvertex(v : VERTEX) : SAME is deferred
(29)     -- deletes a vertex together with its incident edges;
(30)     -- Precondition: true
(31)     -- Postcondition: res.V = V.del(v)∧
(32)     --          res.E = E \ {e ∈ E : e1 = v ∨ e2 = v}
(33)
(34) deledge(e : EDGE) : SAME is deferred
(35)     -- deletes an edge;
(36)     -- Precondition: true
(37)     -- Postcondition: res.V = V ∧ res.E = E.del(e)
(38)
(39) are_connected(v, w : VERTEX) : BOOL is
(40)     -- result is true iff there is a path from v to w;
(41)     -- Short: v →* w
(42)     -- Precondition: v ∈ V ∧ w ∈ V
(43)     -- Postcondition: res = true ↔ w = v∨
(44)     --          ∀e ∈ E : e1 = v ⇒ e2 →* w
(45)     ...
(46) end;
(47)
(48) is_acyclic : BOOL is
(49)     -- result is true iff self is acyclic;
(50)     -- Precondition: true
(51)     -- Postcondition: res = true ⇔
(52)     --          ∄v, w ∈ V : v ≠ w ∧ v →* w ∧ w →* v
(53)     ...
(54) end;
(55)
(56) ...;
(57) end;
```

The '...' in line (56) stands for arbitrarily many further methods, e.g. the computation of shortest paths, strongly connected components, etc. All methods except those marked **deferred** are derived and may therefore be implemented without knowing the actual representation. Any actual implementation must provide at least the **deferred** methods. The following piece of code gives a simple implementation based on the set definition of a graph.

```
(1)  class D_GRAPH_SET(
(2)      VERTEX,
(3)      EDGE < EDGETYPE(VERTEX)) is
(4)  -- implements directed graphs directly by sets
(5)  -- Invariant: ∀e ∈ EE : e1, e2 ∈ VV;
(6)
(7)  subtype of D_GRAPH(VERTEX, EDGE);
(8)
(9)  VV : $SET(VERTEX);
(10) EE : $SET(EDGE);
(11)
(12) V : $SET(VERTEX) is ;
(13)     -- returns the set of vertices
(14)     res := VV;
(15) end;
(16)
```

[5] The current version of SATHER-K does not yet have this keyword. Instead, there is an **include** clause, by means of which only source code can be reused.

(17) $E : \$SET(EDGE)$ **is**;
(18) −− returns the set of edges
(19) $res := EE$;
(20) **end**;
(21)
(22) $init : SAME$ **is**
(23) −− returns an empty graph;
(24) −− Precondition: *true*
(25) −− Postcondition: $res.VV.empty \wedge res.EE.empty$
(26) −− $\wedge self' = res$
(27) $VV := \#SET\_LIST(VERTEX).init$;
(28) $EE := \#SET\_LIST(EDGE).init$;
(29) $res := self$;
(30) **end**;
(31)
(32) $addvertex(v : VERTEX) : SAME$ **is**
(33) −− inserts a vertex;
(34) −− Precondition: *true*
(35) −− Postcondition: $res.VV = VV.ins(v) \wedge res.EE = E$
(36) −− $\wedge self' = res$
(37) $\cdots$;
(38) **end**;
(39)
(40) $addedge(e : EDGE) : SAME$ **is**
(41) −− inserts an edge;
(42) −− Precondition: $e_1 \in V \wedge e_2 \in V$
(43) −− Postcondition: $res.VV = VV \wedge res.EE = EE.ins(e)$
(44) −− $\wedge self' = res$
(45) $\cdots$;
(46) **end**;
(47)
(48) $delvertex(v : VERTEX) : SAME$ **is**
(49) −− deletes a vertex together with its incident edges;
(50) −− Precondition: *true*
(51) −− Postcondition: $res.V = V.del(v) \wedge$
(52) −− $res.E = E \setminus \{e \in E : e_1 = v \vee e_2 = v\} \wedge self' = res$
(53) $\cdots$;
(54) **end**;
(55)
(56) $deledge(e : EDGE) : SAME$ **is**
(57) −− deletes an edge;
(58) −− Precondition: true
(59) −− Postcondition: $res.V = V \wedge res.E = E.del(e)$
(60) −− $\wedge self' = res$
(61) $\cdots$;
(62) **end**;
(63)
(64) **end**;

The notation $\#A$ means that an object of type A is created. Finally, we define a type for directed acyclic graphs.

(1) **abstract class** $DAG($
(2) $VERTEX,$
(3) $EDGE < EDGETYPE(VERTEX))$ **is**
(4) −− defines directed acyclic graphs;
(5) −− Invariant: $\forall e \in E : e_1, e_2 \in V \wedge is\_acyclic$;
(6)
(7) **specialization of** $D\_GRAPH(VERTEX, EDGE)$;
(8)
(9) $addedge(e : EDGE) : SAME$ **is deferred**
(10) −− inserts an edge;
(11) −− Precondition: $e_1 \in V \wedge e_2 \in V \wedge \neg e_2 \xrightarrow{*} e_1$
(12) −− Postcondition: $res.V = V \wedge res.E = E.ins(e)$
(13)
(14) $topsort : \$LIST(VERTEX$ **is**
(15) −− sorts *self* topologically;
(16) $\cdots$;
(17) **end**;
(18)
(19) **end**;

Observe that the definition of a method *topsort* to compute a topological order on the vertices exists only for *dag*s.