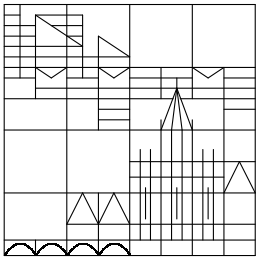

Universität Konstanz

Using Design Patterns for Reusable, Efficient Implementations of Graph Algorithms — Working Paper —

Dietmar Kühl
Karsten Weihe

Konstanzer Schriften in Mathematik und Informatik
Nr. 1, Januar 1996

ISSN 1430–3558

Using Design Patterns for Reusable, Efficient Implementations of Graph Algorithms — Working Paper —

Dietmar Kühl[1] Karsten Weihe[1]

Lehrstuhl für praktische Informatik I
(Algorithmen und Datenstrukturen)

1/1996
Universität Konstanz
Fakultät für Mathematik und Informatik

Abstract

Software reusability is an important and difficult problem in general, and this is in particular true for graph algorithms. The usual way to address reusability of graph algorithms is to provide a standard–setting library of data structures (incl. various kinds of graphs), on which graph algorithms may be implemented.

In this working paper, we discuss the disadvantages to this approach, and we propose a couple of *domain design patterns* to overcome these disadvantages. To apply design patterns, we shift the focus from data structures as the primary units of reuse to algorithms themselves.

So far, our concept is highly immature, and so is this paper. In fact, this paper merely reflects the state of our discussion and is intended to serve as a base for further dicussions.

We are currently implementing a case study (maximum flow problem) in C++ to evaluate our ideas empirically.

1 Introduction

Efficiency vs. reusability. Besides mere correctness, there are several design criteria for good software, which contradict each other to some extent.

On the one hand, run–time efficiency is crucial in many situations. This means the time needed for execution and (typically less important) the amount of machine resources required. The dominant measure of run–time efficiency in the literature is the asymptotic worst–case complexity. However, in practice the "big–oh factor" is equally important. In particular, in many situations the factor imposed by complicated operations (e.g., virtual storage management) may be prohibitive.

On the other hand, *reusability* becomes more and more important, because this determines the cost of the software development process. Reusability concerns data structures and algorithms that are potentially useful in many different contexts, possibly even in contexts that are not yet discovered. The aim is to implement these data structures and algorithms such that the resulting code can be reused in new, unforeseen contexts with little additional effort for adaptation.

Graph algorithms. Reusable implementations of graph algorithms are particularly desirable, since there is an overwhelming wealth of interrelations between different algorithms in the literature, and this wealth is still dramatically growing. Algorithms for basic tasks (e.g., distances, connected components, blocks, paths, spanning

[1]Universität Konstanz, Fakultät für Mathematik und Informatik, Postfach 5560/D 188, 78434 Konstanz, Germany, {dietmar.kuehl,karsten.weihe}@uni–konstanz.de, http://informatik.uni–konstanz.de/~{kuehl,weihe}

trees) are used in literally thousands of more advanced algorithms. The latter algorithms are, in turn, used for even more advanced algorithms, and so on.

Not all of these interrelations are addressed explicitly in original work and textbooks. Just to mention an obvious example: For many algorithms it is assumed without loss of generality that the underlying graph be connected or even biconnected. In many contexts, this means that an implementation of the algorithm must invoke a procedure that partitions the graph into connected (resp., biconnected) components.

Reusability is in general a difficult task. It is particularly difficult for graph algorithms, for the following reasons.

- Sophisticated performance tuning is very often a prerequisite for implementations of graph algorithms to be acceptable. These two goals—performance and reusability—contradict each other and are, hence, hard to achieve simultaneously.
- A key problem to writing reusable code is dependency on the underlying data structures. Many graph algorithms depend on highly sophisticated data structures. In particular, many algorithms depend on fine–tuned, tricky representations of graphs, which differ significantly from algorithm to algorithm, even for algorithms that work on the same type of graphs.

Typically, most details of this fine–tuning are not explicitly mentioned in original work and textbooks. In fact, more often than not the "big–oh" factor is simply ignored, and the details necessary to achieve the desired asymptotic bound must be concluded from the high–level description of the algorithm.

However, it is exactly these details which make writing reusable code for graph algorithms so hard, because every algorithm and every context requires another fine–tune.

There is a large number of papers on implementations of particular graph algorithms, and it is absolutely impossible to give a comprehensive survey. However, it seems that the overwhelming majority of these papers focuses exclusively on performance issues and does not address reusability at all. Consequently, most work does not apply to situations where reusability matters.

Focus on data structures. To our knowledge, all approaches to reusable implementations of graph algorithms in the literature focus on the underlying data structures as the primary units of reuse. This means that a library of data structures is implemented, and all algorithms are implemented on this common basis. These data structures are intended to set a standard. The aim is to formulate every context in terms of this standard and to make all algorithms thus compatible with different contexts and with each other. (In contrast, we set no standard and, hence, our approach is compatible with *every* standard.)

There are many standard–setting libraries for different languages, which typically provide basic data structures such as lists, stacks, queues, search trees, and hash tables, just to mention a few. Some of them also provide data structures for graphs and basic algorithms on graphs.

On the other hand, there are a few libraries especially tailored to graph algorithms and related problems such as computational geometry. It seems to us that LEDA [MN95] has become the most popular library in the algorithm research community and in various application domains.

The design of the more recent libraries is based on object–oriented principles [Boo91, GHJV94, Mey94], and most of them are implemented in languages that provide object–oriented features. This is not surprising, since object–oriented programming is a major step towards reusable implementations of data structures. A brief introduction can be found in the glossary at the end of this working paper.

Focus on algorithms. In contrast, we focus on algorithms as the primary units of reuse, and instead of a standardized set of data structures, we propose a couple of *domain design patterns* for implementing algorithms such that they may work on every set of underlying data structures (e.g., a standard library or a performance–tuned implementation tailored to a specific algorithm). Moreover, we discuss a few typical scenarios, and we provide some code that supports our concepts

in these scenarios.

Design patterns were developed very recently as a concept of their own. The book by Gamma et al. gives an introduction and lists the most fundamental design patterns [GHJV94], and the glossary at the end of this paper gives a brief survey. Our proposal was also inspired by the *Standard Template Library (STL)*, which is part of the prospective ANSI/ISO standard for C++.

Like us, Gallo and Scutella [GS93] focused on algorithms as the primary units. They, too, propose a couple of general design decisions and implement various maximum flow algorithms in a case study, which evaluates these decisions empirically. (In terms of Gamma et al. [GHJV94], they apply the pattern "Strategy" rigorously.) However, their goal is completely different, namely

- To find a joint implementation of all algorithms such that they share as much code as possible.
- To implement different algorithms such that they are all tailored to one specific context, but in this context they are interchangeable at run time (dynamic polymorphism; see the glossary).

Organization of the paper. In Sect. 2, we review the usual approach in greater detail, that is, libraries of standardized data structures as the basis for graph algorithms. Then, in Sect. 3, we introduce and discuss, on a high level, our concepts for achieving high performance and reusability simultaneously. In Sect. 4, we analyse several concrete scenarios. Finally, in Sect. 5, we outline our case study (maximum flow problem). The glossary at the end of the paper reviews all basic terminology.

A description of our concepts using the style introduced by Gamma et al. [GHJV94] follows soon.

Note. We tried to give examples for all our concepts. Some of them are very concrete, whereas other examples are rather generic, maybe even a bit "academic." However, we think that the latter examples are important, too, because they demonstrate the scope—and the limits—of a concept much better than concrete, real–life examples.

Special Acknowledgement

The discussion in the following section is mainly based on our experiences with LEDA [MN95, Näh95]. In fact, we used LEDA in several projects, but sooner or later, we removed LEDA from all algorithmic parts and used it only for modeling in non–algorithmic parts. Moreover, we know several groups where things like that happened, too. In Sect. 2, we will try to work out our main problems with LEDA.

Nevertheless, among all libraries we know, LEDA is the most advanced one. Our point is to discuss libraries as a general approach, not a particular library.

Moreover, we do not intend to replace libraries, but to add a framework which may help overcome the problems we had with LEDA. LEDA seems to fit perfectly into this framework.

2 Standardized Data Structures

All containers in LEDA and similar libraries are template classes such that the item data types are template parameters. In particular, graphs are container classes, too, and implemented as classes with two template parameters: the node data type and the edge data type. Both types may be compound types (*records*), so any set of parameters for nodes and edges may be modeled.

In each library, the graph classes provide methods for inserting and removing items and for accessing the associated data. Moreover, there are methods to iterate over the set of all items in a linear order. There are also methods to iterate over all edges incident to a given node. The latter methods can be used to navigate through the structure of the graph, for example, in a depth–first search. In each library, iteration and navigation are realized by an abstraction of pointers to nodes and edges. Some libraries also integrate a few basic graph algorithms (e.g., graph searches) in the graph classes themselves.

To give a concrete example: LEDA [MN95] offers a set of basic data structures such as lists, stacks, different search trees, and hash tables. It offers data structures for directed and undirected graphs, which may or may not be parameterized.

For every data structure, a sophisticated implementation with provably good worst–case complexity is chosen. To some extent, polymorphism is realized.

In the rest of this section, we will discuss some problems of the general library approach.

Temptation for re–implementation. There is always a temptation to re–implement algorithms and data structures from scratch instead of reusing existing software. In fact, reusing data structures and algorithms causes programming overhead both in development and maintenance, because the rest of the program must be written so that all pieces fit perfectly together. The necessary workarounds may ruin the internal architecture of the program or cause a significant run–time overhead. The temptation for re–implementation is especially high when

- The interfaces of the modules to be reused do not fit smoothly into the new context.
- A re–implementation which *does* fit into the new context is not too difficult and does not require special expertise.
- There is a simpler algorithm or data structure which provides the same functionality, is easy to implement, and has acceptable performance.

General–purpose implementations of graphs and similar data structures are easy to re–implement. On the other hand, for most sophisticated data structures there are simple alternatives, which may have acceptable performance in many contexts. This all the more so, because the theoretical worst–case complexity is only loosely coupled with the practical performance.

Therefore, the temptation to circumvent the standards of a low–level library is very high when the basic data structures are the units of reuse.

See the remark on full logical transparency in Sect. 3.1 for a (we think) generally interesting situation where we could not resist this temptation in our work with LEDA.

The most drastic example for the significance of this argument is LEDA itself. For example, the algorithms for connected and biconnected components and several other algorithms re–implement depth–first search, several algorithms re–implement breadth–first search, and even Dijkstra's algorithm is re–implemented to fit in another context.

Integration of different standards. Implementing all algorithms on a couple of standard data structures is often not possible. Instead, algorithms implemented on different standards must work together. There are several strategies to attack this problem.

Conversion: Clearly, this problem can be completely solved by conversions, which convert all relevant data back and forth, whenever an algorithm is invoked that is implemented on another standard than the calling algorithm.

However, conversions take linear time. For invoked algorithms for which linear run time can be proved theoretically or at least observed empirically, the increase of the "big–oh" factor may not be acceptable (even for many superlinear algorithms). Even worse, if the invoked algorithm requires sublinear time, the conversion may increase the asymptotic complexity of the calling algorithm.

For example, this happens if the invoked algorithm manipulates the underlying graph only locally, but the size of the manipulated subgraph is not foreseeable for a single invocation and can be bounded only by an amortized analysis over all invocations.

Moreover, conversions are often a non–trivial task of their own, because there may or may not be a simple mapping from one representation to the other one, especially if the representations are fine–tuned.

Parallel maintenance: As an alternative to casual conversions back and forth on demand, an algorithm could maintain a copy of the underlying data, which is represented according to the requirements of the invoked subroutine. However, if an algorithm invokes several subroutines that are implemented on different representations, this slows down the algorithm drastically. Even worse, these representations must be hard–wired in the algorithm, which either restricts reusability to trivial modifications of the context or is highly error–prone (namely when being hard–wired as generic pointers or as polymorphic classes using

run–time type information; see the glossary).

Dynamic polymorphism: Here we define the standardized graph interface only as a pure base class and derive concrete classes from it, which serve as Adapters (cf. glossary) for fine–tuned representations and for representations according to other library standards.

However, the functionality of this pure base class must not exceed the intersection of the functionalities of all derived classes. Even more, it must not exceed the functionality of any representation possibly to be integrated in the future. Therefore, this functionality is unreasonably weak.

To some extent, this problem may be overcome in the following way: The concrete representations are not immediately derived from one common base class. Instead, an inheritance hierarchy is constructed where every path from the overall base class to some concrete representation class contains several intermediate pure classes, each of which adds some piece of functionality to its ancestors. Every algorithm is implemented on as high a class as possible in this hierarchy to achieve maximum reusability.

However, it is well known that such a large hierarchy causes severe design problems, which may even be prohibitive. In fact, both authors were involved in projects whose implementations should, originally, be based on large inheritance hierarchies for basic data structures such as graphs. However, in no project could this ambitious aim be matched satisfactorily, notably the ADLIPS project [KLM+93] and the PlaNet project (http://winnie.math.tu-berlin.de/~neyer/planet/demo.html).

One of the major problems is the following. On the one hand, it is practically impossible to implement the potential lattice of all intermediate classes as a whole. On the other hand, it is impossible to identify in advance the classes that will actually be needed in the future. Therefore, intermediate classes must be inserted afterwards "on demand." Since it is even impossible to forecast this lattice, it is impossible to leave the "right" gaps in the hierarchy. Therefore, chances are high that an initial hierarchy must be rearranged sooner or later, which is well known to be difficult and error–prone.

Flexibility of standards. To be useful, a standard must be flexible enough to serve all purposes in a reasonable manner. Reasonable means that the workarounds necessary to circumvent inflexibilities do not require too much effort. However, this flexibility causes severe efficiency problems.

For example, to achieve the asymptotic worst–case complexity stated in the literature, many graph algorithms require that testing adjacency of two given nodes takes only constant time. This feature can be supported only by additional random–access information, say an adjacency matrix or a sophisticated hash table.

Unfortunately, no algorithm may run in $\mathcal{O}(|V| + |E|)$ time anymore, if the underlying graph representation provides such a support. For algorithms on very large, sparse graphs (e.g., large–scale grids in VLSI design) this overhead may take several years, even if the algorithm itself is real time.

The following strategies may be used to attack this problem.

Essential algorithms: Only an "essential" subset of all algorithms is supported. However, the last example is so basic (and it is by far not the only one) that this approach seems hopeless.

Multiple standards: Another way is to support a small number of standardized representations for each logical data structure such as a directed graph. However, this variant relies either on conversion routines or on parallel maintenance, which again drives the run time of many algorithms beyond an acceptable amount.

Dynamic polymorphism: This technique may also be used to add more functionality to a standardized graph class on demand. Unfortunately, the design problems explained above reappear here as well.

Organization of the data. Consider the problem of organizing all parametric data associated with nodes and edges on a background device. To this end, think of all node (resp., edge) parameters as a table with one row for each node and one column for each parameter. This table can be stored in two ways: either row–wise or

column–wise (or combinations thereof).

In other words, we may either hold one record for each node (resp., edge), which contains all data for this particular node (edge); or we may hold one dictionary for each parameter, wherein the values of this parameter are stored for all nodes (edges). In the latter case, a particular value must nonetheless be accessible in constant time, given the corresponding node (edge). Therefore, an array with dynamic index range seems the only reasonable implementation for such a dictionary.

In the row–wise case, the set of all parameters—and their names—must be hard–wired in every algorithm. This makes reuse impossible except for trivial cases, because otherwise the node (edge) data types mismatch. Therefore, algorithms in libraries such as LEDA rely on a column–wise organization (using arrays with *static* index range, by the way).

However, a column–wise organization may cause significant run–time overhead, namely whenever an algorithm manipulates the underlying graph in tricky ways. For example, whenever nodes and edges are added beyond a variable upper bound, all arrays must be reorganized on the free store.

To give a more subtle example: Consider an algorithm that partitions the graph into subgraphs and calls itself recursively with each of these subgraphs. Moreover, suppose the latter algorithm needs to scan a certain node parameter in a linear order. In this case, the ordering of the array elements must be rearranged on each level of the recursion—either in–place or by copy—so that the indexes of each partition set appear consecutively.

3 High–Level Concepts

In this section, we propose a way to use design patterns for reusable, efficient implementations of algorithms. The key insight is that algorithms should be reused, not the underlying data structures.

3.1 Organization of Algorithms

Algorithms as classes. We implement algorithms not as functions or procedures, but as classes. Such a class has a specific execution method, which executes the algorithm. This greatly simplifies the handling of auxiliary data for algorithms and supports the concepts introduced in the rest of this paper.

If the algorithm consists of several major steps, it might be reasonable to provide an additional couple of methods, each of which performs exactly one major step (partial execution methods). An advanced example of this decomposition is the design pattern Loop Kernel in Sect. 3.3.

Each algorithm object is initialized before the whole algorithm starts, that is, before the execution method of the root of the algorithm–subalgorithm hierarchie is invoked. This separates the execution of algorithms from all organizational stuff. This is essential for reuse of algorithms because the organizational stuff is typically context–dependent.

Calling subroutines. For many problems, more than one algorithm is proposed in the literature. Each algorithm has its own strengths and weaknesses, and in each concrete context another algorithm may be favorable. Hence, for an implementation of an algorithm to be reusable, it is crucial that the subroutines invoked by the algorithm are not hard–wired, because otherwise they cannot be chosen according to the given context.

We realize all subalgorithms that are directly called by an algorithm as private data members of the calling algorithm's class, and we make these subalgorithms exchangeable by means of polymorphism.

However, we prefer dynamic to static polymorphism, for the following reasons.

- The list of template parameters becomes much too large, at least for high–level algorithms.
- The subroutines invoked directly determine which subroutines may be invoked indirectly. This cannot be expressed by means of templates in a convenient, clean manner.

Therefore, static polymorphism is used only

where these two problems do not occur. Typically, this includes all algorithms for which inlining is profitable. Since invocations of other subalgorithms are not too frequent, the run–time overhead caused here by dynamic polymorphism might be negligible.

Before running an algorithm, all subalgorithms of the algorithm object are instantiated. If a subalgorithm is not instantiated explicitly, a default algorithm is chosen. If the same subalgorithm is used for two or more tasks which are logically different, it may be preferable to store two different objects for these two tasks, which may or may not represent different algorithms.

Note that keeping subalgorithms as private members of the algorithm itself is crucial for exchangeability. In fact, two subalgorithms solving the same problem may call, in turn, a completely different couple of subalgorithms of their own. Therefore, an algorithm must not "know" the number and functionalities of its "subsubalgorithms."

Clearly, this can be realized by unsafe constructs like void–pointers or run–time type information, which are passed downwards in the algorithm–subalgorithm hierarchie from algorithm to algorithm. In contrast, subalgorithms as private data members are an absolutely safe way to avoid upwards propagation of knowledge about subalgorithms, because it strictly realizes the general principle of locality, that is, no piece of code should have to manage data that are not relevant for this piece.

Full logical transparency. Like every other class, an algorithm class may be assigned a *logical state*, which describes the state of an object independently of the implementation of the class (see the glossary). An algorithm class should provide methods to retrieve the logical state of an object completely, and it should also provide methods to change the logical state to some extent (e.g. to replace subalgorithms between two calls to execution methods).

This concept is important for algorithms that offer not only one single execution method but a couple of partial execution methods, An algorithm that invokes a subalgorithm step by step might make decisions after each step based on the current logical state of the subalgorithm object. For a concrete example, see the last example for the Loop Kernel in Sect. 3.3.

Remark: In our opinion, "normal" classes, i.e. classes representing data structures, should be logically transparent as well. For example, for a stack this means that there is not only a method "top" to retrieve the uppermost element, but also a way to iterate over *all* elements, because all elements together constitute the logical state of a stack object.

An example where full logical transparency is useful is debugging a program that contains classes from a foreign library, e.g. a stack. In this situation, one might be interested in the contents of the stack at some state of the program, even when the bug is not in the stack itself.

Unfortunately,we know of no library wherein full logical transparency is realized; typically, only the basic functionality that defines an abstract data structure is realized (e.g., "push," "pop," and "top" for stacks). In our practice, this simple problem forced us time and again to replace existing implementations of basic data structures by self–written code, since any workaround seems to be more difficult to implement and to mess up the program.

In the context of graph algorithms, full logical transparency is particularly important for polymorphic data structures. In fact, in our opinion, the logical state of a concrete representation class should also contain information about this concrete representation. For example, if a pure base class represents a dictionary and is implemented by an AVL tree in a derived, concrete class, the latter class should provide additional methods for retrieving information about the current tree structure, for example, a way to iterate over all data items in a tree–like fashion.

At first glance, this seems to contradict the principle of encapsulating all implementation–specific details and to hide them from the client code. However, this is really no contradiction, since a class "dictionary as AVL tree" is logically an AVL tree, not a generic dictionary.

A concrete example from our practice is the dynamic tree variant on the push–relabel algorithm [GT88], which is the currently best maxi-

mum flow algorithm with respect to asymptotic complexity. This algorithm relies on a specific implementation of dynamic trees and needs information that are specific for this implementation. Therefore, this algorithm cannot be implemented on a pure base class for dynamic trees, but only on a derived concrete class, and the latter class must provide additional functionality, which is specific for this implementation of dynamic trees.

Within the ADLIPS project [KLM+93], the second author had to implement a maximum flow algorithm and originally planned to implement the push–relabel algorithm based on the dynamic tree class in LEDA. However, this class provides only the functionality of general dynamic trees, so he got stuck and decided to implement a variant with worse asymptotic complexity.

Exchangeable data types. Many algorithms may in principle work on different data types. For example, lengths, capacities, flows, costs and similar edge data types may be of any numeric type (integral or real). In some contexts, it is even necessary to use a self–defined type.

For instance, in some contexts where numeric data (integral or real) are involved, it may be necessary to "perturb" all data slightly. One way of doing so is the following: Instead of values $a_1, \ldots, a_n$, we consider $a_1 + \varepsilon \cdot b_1, \ldots, a_n + \varepsilon \cdot b_n$. The arithmetics of these pairs (a_i, b_i) conforms to the informal assumption that $\varepsilon > 0$ be "infinitesimally" small.

As far as this makes sense, we leave the actual type of the data open, that is, the data types on which an algorithm works are template parameters of this algorithm class. (For technical reasons, we use so–called *traits*; this design pattern is extensively used in the prospecitive ANSI/ISO–standard for the C++ library.)

In order to avoid propagation of template parameters from a subalgorithm to a calling algorithm, each algorithm is represented by *two* classes. The first class is a pure base class, and only the types of input/output parameters of its execution method appear in its definition (as statically polymorphic parameters in fact). The other class is derived from this base class. The derived class overrides the base class' execution method with the actual implementation of the algorithm. All data types on which the algorithm works are template parameters of the derived class.

If another algorithm uses this one as a subroutine, it sees only an object of the pure base class. Therefore, a calling algorithm need not be parameterized by the types of the data that subalgorithms use for their internal purposes. In other words, information about the internals of a subalgorithm is not propagated upwards.

In contrast, LEDA provides two versions of several algorithms, one for integer weights and one for real (double precision) weights (e.g., shortest paths, maximum flows). Moreover, for the minimum cut algorithm, only an integer version is provided, although the algorithm is potentially useful for other data types, too. Therefore, the LEDA algorithms cannot even be used for all built–in types of C++. For example, in some situations it may make sense to use data type *float* instead of *double*, because this type requires much less space, and the additional accuracy provided by the double precision type is not necessary for most graph algorithms.

Special graph categories. Many algorithms are guaranteed to work only if the underlying graph falls into some category, for example, grid, planar graph, interval graph, chordal graph, perfect graph, bipartite graph, just to mention a few.

At first glance, subtype modeling by inheritance seems the method of choice. However, here the subtype does not only extend, but also restrict the functionality of the base type.

For example, a planar graph may safely offer all methods that a general graph class offers. However, if an algorithm tries to insert an edge that would destroy planarity, insertion is denied by the called method. On the other hand, a special class for interval graphs might even restrict the set of methods offered. In fact, in interval graphs the set of all edges is completely determined by the underlying interval structure. Therefore, any attempt to insert or remove a single edge must be denied, so it makes no sense to offer these methods.

Restricting functionality of subtypes handicaps reuse severely, for the following reasons.

- Algorithms that are implemented so as to run on general graphs cannot be reused for

a special class, no matter whether the set of methods or only their functionality is restricted. Therefore, such an algorithm must be re–implemented for every category where it is useful.

- Even worse, many algorithms cannot even be re–implemented this way, although they guarantee that the graph still falls into its special category after execution of the algorithm. In fact, modifications of the graph structure are usually done step by step, and there is a huge number of algorithms where the intermediate results fail to fall into this category.

Another problem is that there is not always a "natural" inheritance relation. For example, in LEDA undirected graphs are derived from directed graphs such that, roughly speaking, directions of edges are dropped. However, for many algorithms (e.g., flow algorithms) it is natural to regard an undirected graph as equivalent to the directed symmetric graph where each undirected edge is replaced by the two corresponding directed edges.

In summary, subtype modeling is of limited use for reusable implementations of graph algorithms. To cope with special categories of graphs, we use two mutually converse strategies, which are derived from the design patterns Adapter and Decorator, respectively.

Subtype modeling by Adapters. For the representation of the graph, we choose a data structure that models general graphs. Whenever an algorithm assumes that the graph falls into a special category, it gets a partial view of the graph, which is restricted to the features of this category. This is easily done using the concepts in Sect. 3.2 below.

The calling algorithm must ensure that the graph falls into the required category. Sometimes, this can additionally be ensured by checks of preconditions. However, the run–time overhead for a complete check may not be acceptable, so we must live with a security gap.

Subtype modeling by Decorators. Sometimes it is not reasonable to choose a data structure for general graphs, even if algorithms for general graphs are invoked as subroutines.

For example, consider a grid. In order not to waste run time and space, it is certainly reasonable to store a grid as a dynamic matrix rather than as a set of adjacency lists. In contexts related to VLSI design, it is often useful to apply algorithms that work on more general graphs (e.g., planar graphs) and should, hence, be implemented only once, namely for the most general case.

Typically, these algorithms change the structure of the graph only slightly. For instance, they may introduce temporary, auxiliary nodes and edges or remove a few edges in order to partition the grid.

For cases like this, we reverse the strategy. The grid is actually realized as a matrix. Whenever an algorithm is called that might change the structure of the graph in a non–grid–like fashion, it does not get the naked grid as input, but a Decorator, which provides the functionality needed by the algorithm. This Decorator adds a small, auxiliary data structure to the grid, in which all violations of the grid structure are recorded and managed separately of the grid structure itself.

3.2 Partial View of Data Structures

In the libraries mentioned in Sect. 2, the interface of a data structure serves as a pipe, which separates the concrete representation of the graph from the abstract algorithm and controls all accesses of the algorithm to the concrete representation. Every algorithm has a complete view of this interface, although it may not need all its features.

We implement, instead of a standardized interface, a tool box of basic interface classes, each of which controls the access to one specific feature of the underlying representation. This means a single node or edge data item or a particular aspect of the structure of the graph.

An algorithm gets only a partial view of the underlying data structure, namely only the information that it definitely needs to do its job. This partial view is composed of a set of basic interface classes from the tool box. In other words, instead of a graph object, an algorithm receives a couple of these basic classes as input.

Essentially, the tool box consists of two kinds of classes, named *Structural Iterators* and *Data Accessors*, which are derived from the design patterns Iterator and Command, respectively.

Each algorithm is polymorphic with all Structural Iterators and Data Accessors being polymorphic parameters (in addition to subalgorithms). For reasons of efficiency, we make them statically polymorphic (in contrast to subalgorithms).

If dynamic polymorphism of some Structural Iterator or Data Accessor class is reasonable in a specific context, the corresponding class contains a pointer to a pure base class (cf. glossary) as a data member. Therefore, the run–time overhead of dynamic polymorphism is completely avoided when dynamic polymorphism is not required.

Structural Iterator. Remember the two types of iteration mentioned in Sect. 2: passing all nodes (resp., edges) in a linear order or navigating through the structure of the graph. Other types of iteration are often required, too, for instance, passing only a subset of all nodes and edges. Other, more advanced, examples are (i) navigating over nodes and faces of a planar graph and (ii) navigating through an interval graph along a chain of the corresponding interval order.

To realize iteration, we propose a generalization of the Linear Iterator design pattern (cf. glossary), which we call *Structural Iterator*. A Structural Iterator class is tailored to one concrete representation of the graph. An object of such a class serves as a pointer to a single node or edge of the graph. However, an object of a Structural Iterator class may iterate over the graph, that is, the node or edge associated with such an object may change over time. Each such class realizes exactly one kind of iteration. For example, a Structural Iterator for navigating through the structure of the graph in the ususal manner could maintain a current node and a current incident edge at each stage of its life time.

The following differences to "normal" Iterators are crucial.

- The features of Linear Iterators and their semantics may be standardized to a large extent, because all possible iterations over a linear structure may be composed of a small number of basic features such as starting at the first item and switching over to the next item. See the STL library for a rigorous (yet small and highly flexible) taxonomy of Linear Iterators.

  In contrast, there is a potentially infinite number of ways to iterate over graphs, especially since many special categories of graphs provide a much richer structure than general graphs. Even more, different algorithms working on the same special category may require different ways of iteration, so it is not clear whether there is a reasonable standard even for special graph categories.

  Therefore, in case of doubts it might be preferable to define Structural Iterators according to the actual requirements of an algorithm and its subalgorithm, not according to a pre–designed taxonomy.

- An object of a Structural Iterator class provides no methods to access the data associated with nodes and edges. This is the Data Accessors' job (see below).

  As a consequence, data organization and structural organization are separated from each other and can thus be combined orthogonally. Since many algorithms access the underlying graph through several different Structural Iterator classes simultaneously, this is a real advantage.

Example: Find a shortest path from a node s to a node t with respect to the number of edges, using a breadth–first search. The implementation of this algorithm gets as input two objects of a Structural Iterator class, which allows navigation through the graph. These objects refer to s and t, respectively. The algorithm works on a queue of objects of this Iterator class. To produce these objects, the algorithm's template requirements for this Structural Iterator class include a clone method.

An object of another Structural Iterator class receives all output from the algorithm. Therefore, in the terminology of STL, this is a pure Output Iterator. The output of the algorithm

is neither the distance function nor a shortest path tree, nor whatsoever. Instead, the algorithm hands over to this Iterator information about its essential actions, for example, the edge scanned last, whenever a node's distance label becomes permanent. The (statically polymorphic) type of the Output Iterator decides upon the actual representation of the solution, and the information passed over to the Output Iterator might suffice to construct any reasonable representation with only negligible overhead. Such an Output Iterator may be seen as a special case of the design pattern "Builder," see the glossary.

To give a contrasting example, the breadth–first search implementation in LEDA returns only the set of all nodes reached and their distances. For an application where a breadth–first *tree* is required, this implementation is simply worthless: The necessary conversion is more difficult to implement and more error–prone than a re–implementation from scratch.

Example: A more subtle example is a situation where the algorithm constructs a graph structure itself, for instance, an algorithm that decomposes a graph into its biconnected components, which are then passed to another algorithm one after another. The latter algorithm (the *client*) may need a special representation of these components, and this special representation may vary from client to client. Structural Iterators save the conversion that is otherwise necessary. Again this may be seen as a special case of the Builder design pattern.

Example: A graph contains primary and secondary edges, and an algorithm invokes the same subalgorithm at least once for the entire graph and at least once for the subgraph induced by all primary edges. A concrete scenario is given in [Sou94], Chapter 2.6.

In our approach, the algorithm requires two different Structural Iterator classes: one class that iterates over primary edges only, and one class that does not distinguish between primary and secondary edges. If primary and secondary edges are strictly separated from each other in the underlying graph representation (e.g. two adjacency lists for each node), the former Structural Iterator restricts attention to the parts representing the primary edges, and the latter Structural Iterator performs a combined iteration. Otherwise, primary and secondary edges are distinguished from each other only by some kind of label, which is checked by the former Iterator and ignored by the latter one.

In our opinion, this approach is much more flexible than the approach presented in [Sou94]. For example, the author discusses the problem that a subalgorithm may be designed either so that it ignores the differences between primary and secondary edges *or* so that it runs either on primary edges only or on secondary edges only. No such restriction is necessary in our approach.

Data Accessor. More often than not, algorithms require additional parameters for nodes and edges. We propose a variant on the *Command* design pattern, which we call a *Data Accessor*.

An object of a Data Accessor class controls access to one single node or edge parameter. That is, given a node or edge (by an Iterator object), an object of such a class is able to read and write this parameter for this particular node or edge. Such a class is tailored to one specific way of managing this parameter.

The Data Accessors needed by an algorithm are private members of this algorithm class. Each algorithm class may provide for each Data Accessor a separate initialization method.

Example: Find a shortest path from a node s to a node t with respect to given edge lengths, using Dijkstra's algorithm [AMO93]. This algorithm requires one Data Accessor for the edge lengths. (Recall that in our concept, shortest path algorithms do not manage distance labels at the nodes, but hand over all necessary information to a Builder class; therefore, no Data Accessor for node distances is required.)

Example: An algorithm for the minimum–cost flow problem calls a shortest path algorithm. The former algorithm keeps a Data Accessor referring to the residual capacities of the edges, the latter one, a Data Accessor referring to the edge lengths. Both Data Accessors refer to the same logical item, because the residual capacity of the flow network is to be interpreted as the edge length.

Therefore, the latter Data Accessor is a copy of the former one. Note that this circumvents the problem of type mismatches in a row–wise organization of all data (cf. Sect. 2).

Example: An algorithm works on a graph on which several edge length parameters are defined, and the edge length used for a call to a shortest path algorithm is not known at compile time. In this case, the algorithm that decides upon which edge length to choose invokes the method of the shortest path algorithm object which initializes the Data Accessor for the edge lengths. This is an example of the only case where an algorithm needs to see a Data Accessor used by a subalgorithm, and this cannot be avoided by a more sophisticated design, because it is intrinsic in the algorithm. Beyond that, the interdependencies between different Data Accessors and Structural Iterators do not appear in the implementations of the algorithms.

3.3 Decomposing Algorithms

In the above–mentioned libraries, each algorithm is implemented as a function or procedure, which encapsulates all steps of the algorithm and serves as a "black box." We do just the contrary.

Each algorithm is decomposed into its components, and these components are further decomposed into even smaller components. Each component at any level is a unit of reuse of its own, as far as this is reasonable. That is, such a component is a module that may be used in another context, independently of the rest of the algorithm for which it was originally implemented. The template requirements on the input of such a component reflect the needs of this single component, not of the whole algorithm.

Next we identify three situations where this general idea is particularly useful: initialization, loops, and error detection.

External initialization. Many graph algorithms consist of an overall initialization phase and a core routine. We separate both parts strictly from each other.

Example: Most shortest path algorithms first set each node distance parameter to an initial value and compute the actual distances afterwards. The first phase requires a Structural Iterator that passes all nodes in a linear order; the second phase requires a Structural Iterator that enables navigation through the structure of the graph. Therefore, the algorithm is partitioned into an initialization algorithm and a core algorithm. Each of them works on only one Structural Iterator class.

Many graph algorithms invoke only the second phase of a shortest path algorithm and provide their own initial values for all node distances. A concrete example is [Fre87]. The core of the shortest path algorithm can be reused for all these algorithms only if the initialization is strictly separated from the core, because otherwise the initial values provided by the calling algorithm are immediately overridden. This happens, for example, when the shortest path implementation in lEDA are used.

Loop Kernel. Most cores of algorithms are loops. The same loop may appear in several algorithms, however, with a few minor details varying from algorithm to algorithm. We propose a domain design pattern, which we call *Loop Kernel*. This is a generalization of the built–in stream concept in the Sather programming language [Omo91]. See [Fla95] for an application of the stream concept to algorithmic software.

Consider an algorithm that consists of one single loop, and assume one step per iteration (the *kernel step*) is common to many other algorithms. In our concept, this algorithm installs an object of a Loop Kernel class. That class provides a partial access method to perform this kernel step once. The algorithm itself is still implemented as a loop. The algorithm–specific operations are done explicitly, and the kernel step is delegated to the Loop Kernel object.

If the kernel step performed by the Loop Kernel can be reasonably divided into a sequence of even more basic steps, another couple of partial access methods is provided by the Loop Kernel class, which perform these basic steps, one after another.

Example: A branch–and–bound algorithm for an optimization problem may be seen as a routine

to traverse a large tree (too large to be stored explicitly), where every leaf corresponds to one possible solution to the problem, the root to the set of all solutions, and any other internal node to a subset of all solutions. The set corresponding to an internal node is the union of the immediate descendants' sets.

In our concept, this traversal is realiazed by a Loop Kernel class. (A prototype has been implemented by the second author within the ADLIPS project [KLM+93].) An object of this class stores, internally, a representation of the current tree node and the corresponding set of solutions (e.g., as a set of additional restrictions of the solution space). It provides one method to go forward in one particular direction, and one method to go backward. A particular branch–and–bound algorithm is a loop where each iteration first invokes a subroutine, which tries to prove that there is no optimum solution in the current set. If succeeding, the loop invokes the backward method of the Loop Kernel, otherwise it chooses one subtree rooted at the current tree node and invokes the forward method with this branch. The break condition of the loop is the break condition of the algorithm.

Example: The Loop Kernel pattern is especially useful, whenever a loop combines several kernel steps. This may happen in two different ways (and combinations thereof).

- Several kernel steps of different kinds are combined into one loop. For example, every primal–dual algorithm combines one primal step with one dual step in each iteration.
- A variable number of kernel steps of the same kind are combined into one loop. For example, several instances of an evolution strategy or a simulated annealing algorithm may run simultaneously. The overall loop performs one step for each instance, and then the current results are compared in order to decide which instances "survive" this round.

Using domain design pattern Loop Kernel, we get all these combinations "for free."

The latter example is an example where full logical transparency of algorithm classes is crucial. In fact, *full* transparency is crucial, because the designer of, say, an evolution strategy cannot forecast what kind of state information a client algorithm needs in order to decide upon death and survival of populations.

Error detection. A subroutine is said to be *robust* if it checks that the input parameters fulfill all preconditions, and that its output parameters and return values fulfill all postconditions, and if it reacts on failure in an adequate fashion. Clearly, besides efficiency and reusability, robustness is desirable as well. For example, the *Karla* library is rigorously designed to be robust, though at the cost of performance and reusability [FZZ94a, FZZ94b].

The code for error detection can either be integrated in the subroutine itself or form two separate subroutines, which may be optionally called immediately before and after the subroutine itself. There are good reasons for the integrated solution.

- Integration of error detecting code in the subroutine itself simplifies the client code, that is, the piece of code from which the subroutine is invoked.
- The error detection routines are invoked automatically for every client, so there is no chance to leave them out by mistake.
- Modern languages typically provide means for turning error detection on and off. Therefore, in any situation where error detection is not efficient enough, the code for that (or the most time–consuming parts thereof) may simply be turned off.

For these reasons, an integrated solution might be preferable in many situations. Nonetheless, we think that for graph algorithms—and for many other mathematical algorithms—separating the error detection code from the algorithm itself is generally preferable. The reason is that there is not necessarily a natural set of preconditions, and since the postconditions usually depend on the preconditions, no natural set of postconditions either.

For example, it is known that the general Newton method to find a zero of a function is correct if

the function is convex. A few further non–trivial cases are known. However, the Newton method is successfully applied to many situations where no satisfying condition is known at all or where such a condition is specific for the given context. Therefore, any set of preconditions fixed in the desing phase of the library might result in checkers that are turned off in the overwhelming number of all contexts (maybe with an additional, more appropriate checker embedded in the client code). This complicates the program and introduces a bias towards one single special case without an adequate benefit.

4 Concrete Scenarios

In this section, we identify some key scenarios and show how to realize the high–level concepts from Sect. 3. More precisely, we focus on the concepts introduced in Sect. 3.2, because getting the Structural Iterators and Data Accessors right in every possible context seems crucial for the value of our ideas.

4.1 Optimize!

Consider the case that full emphasis is put on efficiency.

In this case, the graph might be fine–tuned with respect to performance. Since this fine–tune absolutely depends on the specific context, we cannot say anything about that. We think that only case analyses can approach this scenario. However, in principle our concept is suitable for this case, because it imposes almost no overhead. Moreover, the concept of Structural Iterators and Data Accessors should be flexible enough to model any performance–tuned access to the graph.

In fact, especially a graph representation tuned with respect to a specific algorithm might perfectly fit into this algorithm's template requirements, so the design of the Structural Iterators and Data Accessors, which link the algorithm to the graph representation, should essentially amount to a one–to–one mapping, which means that the overhead with respect to development time and efficiency may be expected to be small.

When efficiency is the primary goal, Structural Iterators are typically mere pointers. If node and edge parameters are organized in the straightforward way (row–wise as records or column–wise as dynamic arrays), a Data Accessor stores either the offset of its component in the record (row–wise case) or is a pointer to an array (column–wise case). Since all Structural Iterators and Data Accessors are statically polymorphic, the (trivial) methods of these classes can be inlined by the compiler.

To improve efficiency, sometimes parameters are not explicitly stored at all and computed only on demand. In this case, the library approach in Sect. 2 simply does not work. Realistic examples are:

- A graph data structure for a flow problem provides edge slots for the flow value and for the capacities, but not for the residual capacities, because these values are easily computed given the current flow values and the capacities. In our framework, this computation is done by the responsible Data Accessor, so for the algorithm itself it makes no difference whether the data are simply retrieved or computed on demand.
- The value of the parameter is constant, for example, the length function in a shortest path problem if the length of a path is defined to be the number of edges on the path. The responsible Data Accessor simply returns this constant on every query, and it does not provide a method for changing the edge lengths. (Clearly, in this scenario an algorithm must be chosen whose template requirements do not include modifications of edge lengths, or else we must apply subtype modeling by a decorator, see Sect. 3.1.)

A more drastical example is the following. Consider an imaginary complete graph K_n whose nodes are strings, and each edge is weighted by the Hamming distance of the two incident strings. If the dictionary of strings is not tiny, the graph cannot be represented explicitly. In our framework, applying algorithms to such a "graph" amounts to designing a Data Accessor that does not look up the Hamming distance of an edge, but computes it "on the fly."

If such an algorithm needs to change the weights of a few edges temporarily, these changes are recorded by the Data Accessor, for example in a hash table, and when computing the weight of an edge, the Data Accessor takes these recordings into account. (If the algorithm needs to change a significant percentage of all edge weights, it might be preferable to store the graph explicitly or to design another compact representation of the problem, if either way is feasible at all.)

4.2 Integration of Standardized Libraries

Consider a graph class like the one in LEDA; that is, a standardized representation of graphs, such that each node and edge parameter may be stored independently either row–wise or column–wise.

It is very easy to adapt an algorithm to such a library if the algorithm is written according to our concepts, namely as follows.

Structural Iterators: A Structural Iterator that realizes a standard way of iterating or navigating simply delegates each request to the corresponding access method of the underlying representation. Clearly, the situation may be very difficult, if the underlying representation is not suitable for a certain way of iterating over the graph. However, such a difficulty is intrinsic to the underlying library, not to our concept.

Row–wise data: We have written a general template class for row–wise Data Accessors, which is independent of the actual library to be integrated. To integrate a new library, all one has to do is writing a (usually very small) function which gets a Structural Iterator and returns the address of the parameter record associated with the node or edge to which the Structural Iterator points.

Once this is done, no further type definitions are necessary; Data Accessor objects may be immediately declared to be of this template class, with this function being one template parameter and the name of the data item being the other template parameter.

Column–wise data: We are designing another template class, which needs a function that gets a reference to an array and a Structural Iterator as input and sets or returns the corresponding value. Once this is done, no more type definitions are necessary like in the row–wise case, no matter how many libraries are to be integrated in the future.

Variations of the scenario: A special case of this scenario is a library for visualizing graph algorithms. In this case the Structural Iterators and Data Accessors form a bridge between the algorithm and the graphical user interface.

Of course, it is possible to integrate several standards simultaneously, for example, a foreign library *and* a graphical user interface. In this case, each Structural Iterator and Data Accessor is a mere combination of several ones.

4.3 Statistical Data

Another variation is the collection of statistical data at run time. For example, operation counts are a good means to estimate the asymptotic complexity of an algorithm empirically ([AMO93], Ch. 18). This means that a couple of operations are counted (separately), whose asymptotic behavior is representative for the asymptotic behavior of the whole algorithm. In all examples given in [AMO93], Ch. 18, it suffices to extend the Structural Iterators and Data Accessors slightly in order to collect these statistical data. Therefore, the algorithm itself need not be changed.

It seems to us that this observation is true for most algorithms in the literature. When this observation is not true for an algorithm, the same method of a Structural Iterator of Data Accessor object's is called in two or more different situations such that these situations must be distinguished with respect to operation counts, but this Iterator/Accessor cannot distinguish these situations from each other by the input parameters of these calls.

This is yet another example where full logical transparency of all algorithm classes is useful. In fact, let us call an algorithm object *active* if one of its methods is currently invoked, that is, this method is somewhere on the run–time stack, but not necessarily top of stack. in some way or other, the current logical states of all active

algorithm objects together contain all information necessary to distinguish situations from each other that must be treated differently with respect to operation counts.

So we can implement a small couple of algorithm classes, each of which stores references to all relevant active algorithm objects as private data members and evaluates their logical states in order to distinguish between these situations.

This algorithm class is "smuggled" into the algorithm–subalgorithm hierarchy as a "Trojan horse." Essentially, there are two possibilities for that.

1. The Structural Iterator or Data Accessor responsible for counting is replaced by a Decorator, which maintains an internal counter and calls the Trojan horse before each incrementation of the counter.
2. A certain subalgorithm is replaced by a Decorator, which calls the Trojan horse once in the beginning. In this case, the Structural Iterator or Data Accessor responsible for counting must provide methods for turning the internal counter on and off, which are called by this Decorator.

Since Structural Iterators, Data Accessors, and subalgorithms are polymorphic for each algorithm, inserting a Trojan horse is trivial in both cases.

The latter strategy might be preferable if there are certain "milestones" during the execution of an algorithm such that all situations between two subsequent milestones are equivalent with respect to counting. Possibly such milestones are the only states of execution where distinguishing between different situations is easy. In this case, the former strategy is even impossible.

4.4 Rapid Prototyping

One major trend in practical software development is to replace classical, cascade–like development cycles by purely incremental ones. This means the following.

We start with a prototype, which is implemented in a rather simple way (e.g., no performance considerations) and based on a rather rough problem specification. This prototype is analyzed experimentally in cooperation with end-users. This yields a more concrete problem specification and a performance profile. Based on these results, the prototype is refined so as to match the latter problem specification and to improve the performance profile. This process is usually repeated several times.

Clearly, rapid prototyping is a difficult reusability problem. Our concepts support incremental development of algorithmic software as follows.

- Since subalgorithms are polymorphic to the calling algorithm, they may first be implemented in a simple way and easily replaced by more sophisticated implementations later on.
- Since algorithms are absolutely independent of the underlying data structures (first of all, by means of Structural Iterators and Data Accessors), refining the implementation of a data structure step by step does not impose too much overhead. In fact, only the internals of a couple of Structural Iterators and Data Accessors must be adapted to such a refined version.
- Even more, algorithms may first be implemented on a library of data structures which is available only on the developer's platform, not on the platform where the algorithms shall run eventually. Again, only the internals of Structural Iterators and Data Accessors must be adapted.

4.5 Managing Algorithm Specific Data Dynamically

Consider the situation that the algorithm and its subalgorithms shall be chosen only at run time. From Sect. 2, remember the disadvantages to a column–wise organization of all node and edge parameters. Therefore, it is often desirable to organize all data row–wise.

To avoid type mismatches in the row–wise case, often the following strategy is applied: Only the problem specific parameters are hard–wired as slots of the node/edge record; all algorithm specific data are managed separately (ab-)using the gaps of the language's type checking system.

We have implemented a pure base class which we call a *Memory Dispatcher*. An object of this class simulates a logical record of algorithm specific data. One Memory Dispatcher object realizes access to the node data, and another one, access to the edge data. Moreover, we have implemented a template class for Data Accessors which cooperates perfectly with this class.

Once an algorithm or subalgorithm is definitely chosen, the Data Accessors required by this algorithm are initialized, and the Memory Dispatcher object is a parameter of their initializing functions (usually a constructor of the Data Accessor class). This initialization function of the Data Accessor notifies the Memory Dispatcher of the amount of space required by the data managed by this Data Accessor. This notification is done using a method of the Memory Dispatcher's, which returns an integer ID for later requests from this Data Accessor.

After this initialization, another method of the Memory Dispatcher's is called once, which makes the Dispatcher ready for work. Afterwards, each access to a data item of a node or edge is as follows: The corresponding Data Accessor calls a third method of the Memory Dispatcher's, the data access method, which receives the ID from the Data Accessor and returns a pointer to the corresponding data item.

In summary, to integrate a new underlying representation of the graph and all associated data, it is only necessary to design a new Memory Dispather class, which is tailored to the concrete representation. No Data Accessor classes need to be defined, since we can reuse the general Data Accessor class template mentioned above, which gets the Memory Manager class as a template parameter.

Subscenarios. We may identify three realistic subscenarios:

Chunk of space: Associated with every node (resp., edge), there is an additional typeless, fixed–size array of machine words, which we call a *chunk* in what follows. The size of this chunk is an upper bound on all possible requirements. In this case, the Memeory Manager classe manages a *current offset* in the chunk, and the notification method of the Memory Dispatcher simply returns this current offset. The current offset is increased by each call to the notification method such that any two data items are disjoint. The initialization method is void.

Like in Sect. 4.2, the Memory Manager needs a function that, given a Structural Iterator, returns the address of the chunk of the node (edge) to which the Iterator points. The data access method receives the ID again from the Data Accessor and simply adds the address of the chunk to it.

We plan to implement a generic Memory Manager class, which is parameterized by this function. After this is done, no further type definitions are necessary to define such a Memory Manager for a particular graph representation.

Generic pointer: Each node (edge) record contains, as an additional slot, a generic pointer (i.e., a void pointer in C++) to "hang on" all algorithm specific data. In this case, the notification method of the Memory Dispatcher first collects the requirements of all Data Accessors and returns the same ID as in the previous subscenario. The initialization method of the Memory Dispatcher allocates one chunk for each node (edge), whose size equals the sum of all requirements. The generic pointer points to this chunk.

In principle, the access method does the same as in the former subscenario, except that each data access needs an additional pointer evaluation.

We plan to implement a generic Memory Dispatcher class for this case as well.

No additional space: After all, nodes and edges need not provide additional space for algorithm specific data. In this case, it might be the best strategy to copy the whole graph into a representation where additional space *is* provided as an additional chunk like in the first subscenario.

Complications. There are several variations on this key scenario where the situation is much more difficult.

Additional graph structure: If a subroutine needs a more sophisticated graph structure, we realize this by additional node and edge data and,

possibly, additional global data. For instance, it may be reasonable to apply algorithms that work on planar graphs and their duals even in contexts where the input graph may or may not be planar. Possible concrete examples are:

- An algorithm works on a planar subgraph of the input graph and makes use of the dual graph.
- A package may first test the input graph for planarity (and, possibly, other categories), and for reasons of performance, it chooses a subroutine tailored to this category instead of a general subroutine.

Such an additional graph structure may be realized by two additional edge data items, each of which refers to one face. Each Structural Iterator that works on the extended data structure contains a Data Accessor, which refers to these data items.

Consistency conditions: The former case is an example for a general problem: Keeping the graph representation consistent. In fact, whenever the primal graph is changed, the dual graph must be changed, too. We propose to solve this problem using the "Observer" design pattern (cf. glossary). So far. we did not develop more concrete ideas for this problem.

Multiple data items: If we need a bounded number of disjoint copies of some data item, we may reserve as many slots in the chunk. However, if we need an unbounded, unforeseeable number of copies, the data model introduced so far is too simple. Important examples are:

- The algorithm (or a subroutine) is recursive, and we cannot give in advance a reasonably tight bound for the depth of the recursion.
- A subroutine is called several times in a pseudo–parallel fashion, and we do not want to restrict the number of pseudo–parallel invocations in advance. For concrete examples, remember the last example for design pattern Loop Kernel in Sect. 3.3.

In the former case, the node and edge data required by the algorithm and its subroutines should be organized by an unbounded stack; in the latter case, they should be organized by an unbounded queue; other situations may require further types of containers.

We do not store multiple data items directly in the chunk. Instead, the chunk contains only the head of the container. Each item of this container maintains a chunk in turn, too, and all these chunks have the same size, namely the total size required by this algorithm and all its subroutines.

All these chunks are organized recursively in exactly the same way. That is, if one of these subroutines is recursive or pseudo–parallel (or whatsoever) itself, we again reserve only the space for the head of the required container and organize copies of all data items in this container.

Equivalent Data Acessors: We call two Data Accessors for different subalgorithms *equivalent* if they work on the same logical data item. For example, two Data Accessor which both control access to the dual graph of the same primal graph are equivalent. Therefore, they must not require different slots from the Memory Dispatcher. This problem cannot be solved automatically, but must be solved by the programmer, simply by cloning equivalent Data Accessors. Clearly, the clone method of the Data Accessor must not notify the associated Memory Manager of additional space requirements.

Compatible Data Accessors: Two Data Accessors are said to be *compatible*, if they may refer to non–disjoint pieces of the chunk without changing the semantics of the algorithms. For example, consider two subalgorithms which are called at different stages of the algorithm such that neither calls the other one directly or indirectly. Data Accessors for these two algorithms whose data need not be maintained after termination of the respective subalgorithm are compatible. In order to reduce the size of the chunks, it might be reasonable not to reserve disjoint space for all Data Accessors, but to allow compatible Data Accessors to refer to overlapping space.

This problem cannot be solved without the aid of the programmer either. We propose to extend the Memory Dispatcher model as follows. Every Memory Dispatcher may be notified of pairs

of compatible/incompatible Data Accessors. The method of the Memory Dispatcher which makes the Memory Dispatcher ready for work calls an optimizer. This optimizer assigns each Data Accessor an offset, where equivalent Data Accessors have exactly the same offset and incompatible Data Accessors are assigned offsets such that their corresponding slots in the chunk are disjoint.

In particular, this optimizer yields the sizes of the chunks. Minimizing the sizes of the chunks is $\mathcal{NP}$–hard, since it is a generalization of the hypergraph coloring problem. (If all slots have same size, this problem is exactly a hypergraph coloring problem.) However, good solutions might be computable with negligible run–time overhead, for the following reasons.

- The instances are very small.
- The instances might usually have a rich interval graph structure, which may be used for good heuristics.

5 Case Study

To evaluate our concepts empirically, we are just implementing a case study. In that, we focus on the maximum flow problem and its applications. Here we do not go into the details. This will be done in the documentation that will accompany the case study. The case study will consist of the following parts.

Max–flow algorithms: The augmenting path algorithm by Ford and Fulkerson, the strictly polynomial variant developed by Edmonds and Karp, several variants on the push–relabel algorithm, several variants on Dinits' algorithm. See [AMO93]. All algorithms will be implemented according to the concepts in Sect. 3.

Efficient implementation: A revision of Goldberg and Cherkassky's implementation of the push–relabel algorithm[GT88], which implements the same performance tunes, but conforms to the concepts in Sect. 3. This part of the case study ought to show whether or not our concepts are suitable for high–performance implementations.

Applications: Here we concentrate on applications where reuse of an implementation that is *not* tailored to this specific context is a non–trivial problem.

Scenarios: We plan to implement concrete examples for several scenarios introduced in Sect. 4. In particular, we plan to implement a graph class that is tailored to flow algorithms and performance–tuned to some extent.

Glossary

Items and containers. A *container* is a data structure that manages a set of *items*, all of which are of the same type, the *item type*. This includes, for instance, arrays, stacks, queues, lists, files, search trees of all kinds, hash tables. A search tree is an example for a *sorted* container, whereas arrays, stacks, queues, and files are *sequenced* containers.

In many libraries, parameterized graphs are implemented as containers with two item types: the node data type and the edge data type.

Encapsulation. An abstract data structure is defined by a logical state, which may change over life time, and a couple of access methods. For example, the logical state of a sorted container is given by the (multi-)set of all items, whereas the logical state of a sequenced container is given by the ordered tuple of all items. The essential access methods of, say, a stack are "push," "pop," and "top."

Encapsulation wraps an interface (called a *class*) round a concrete implementation of a data structure. Such an interface consists solely of access methods. The implementation can be accessed only through these methods and is, hence, hidden from all algorithms and data structures that use this particular data structure. Variables of a class type are usually called *objects*.

Encapsulation is the most basic object–oriented feature.

Templates. Many algorithms are applicable to a wide range of data structures (*genericity*). For instance, a sorting routine applies to any sequenced container, provided a total ordering is defined on the item type. We call this condition a *template requirement*. Likewise, many containers are

generic in this sense, because here the item type may be arbitrary, too.

A template is an implementation of an algorithm or a data structure where one or more item types are not hard–wired, but replaced by symbolic names (*template parameters*). Such a template can be applied to any data structure that fulfills the requirements, without any additional programming effort. Typically, the compiler draws one concrete implementation in machine code from the template for every combination of item types to which the data structure or algorithm is applied.

Static polymorphism. This is a more advanced application of templates. For example, consider the *external mergesort algorithm*. In principle, this algorithm works on every data structure that provides at least the functionality of an unbounded queue. This is the case for all common non–static sequenced containers. To make an implementation of this algorithm applicable to all of them, the algorithm gets two template parameters: the item type and the concrete data structure for organizing the items. The compiler passes an invocation of the mergesort algorithm, if and only if the data structure provides the required functionality syntactically.[1] Mismatches of method names may be resolved using design pattern "Adapter" (see below).

The STL library makes extensive use of static polymorphism.

Dynamic polymorphism. Essentially, there are two disadvantages to static polymorphism.

- When instantiating a statically polymorphic class, the polymorphic parameters must be hard–wired in the source file, so that the compiler may insert the corresponding code. This makes it impossible to let the choice depend on input data or user interaction at run time. Dependency on input data is sometimes necessary for good performance, because different data structures with the same functionality may be preferable for different inputs.
- Consider an algorithm with several template parameters. If this algorithm is called with many different combinations of these template parameters, the object code might become tremendous.

The object–oriented feature *inheritance* provides a way to overcome these problems. For example, consider a polymorphic container class, say a dictionary. We define a *pure base class*, which consists solely of an interface to dictionaries. Then we *derive* the class interfaces of all concrete representations of dictionaries from this base class. That is, the interfaces of all these classes are either equal to the interface of the base class or extensions thereof. Now consider an algorithm that receives a dictionary as an input parameter, and assume that the concrete representation of the data structure does not matter for this algorithm. Then the type of the formal parameter is the base class.

When invoking the algorithm, the actual parameter is an object of any concrete derived class; we say that the *static type* of this object is the base class, and that the *dynamic type* is this derived class.

However, the same method must be implemented in different ways for different concrete representations. For example, the code for inspecting an AVL–tree to find a specific value differs from the code for inspecting a B–tree. The correct method is chosen and invoked at run time, depending on the dynamic type of the parameter. (Typically, each object of a derived, concrete class additionally stores a pointer to a table of all methods of its dynamic type, and the correct method is chosen by evaluating this pointer.)

In summary, dynamic polymorphism is more flexible than static polymorphism. However, dynamic polymorphism puts an additional run–time overhead on every call to a polymorphic function. In contrast, in the static case even the normal overhead caused by function calls can be essentially saved using the inlining feature of C++.

Inheritance hierarchies. The original intent of inheritance is to model general subtype–supertype relationships. For example, cycles, triangles, and quadrilaterals are all subtypes of a general "shape," rectangles and kites are sub-

[1] Clearly, the compiler cannot check whether the *semantics* of the functionality meets the requirements, because this is in general a non–computable task.

types of quadrilaterals, and squares are a common subtype of rectangles and kites.

The inheritance mechanism may be used to make the interfaces of these types compatible, that is, the subtype has the same interface as the supertype (or an extended interface), and an object of a subtype may be used whereever an object of the supertype may be used. In particular, dynamic polymorphism is a special case of subtype–supertype modeling, where all subtypes are immediately inherited from a common supertype, and the supertype is a pure base class.

Run–time type information. In some object–oriented languages (e.g., C++), the dynamic type of a dynamically polymorphic object may be retrieved at run time from the additional type information that are internally added to every object of a dynamically polymorphic class. This information is called *RTTI (run–time type information).*

Design patterns. Some design problems appear time and again in many different contexts. Typically, only a few solutions are adequate. A *design pattern* is an explicit formulation for one solution to a specific design problem.

To understand the very nature of design patterns, it is crucial to realize that design patterns are by no means a formal programming methodology. Design patterns are informal descriptions of useful rules of thumb. They are strictly classified by the problem classes they have been developed for. For example, there are pairs of design patterns such that both patterns are intended to solve different design problems, but one pattern may be seen as a special case of the other one (see below, Decorator/Adapter). Unlike in formal treatment, the former design pattern is regarded a first–class pattern in its own right and treated independently of the latter pattern.

Examples from [GHJV94]:

Iterator: A class to iterate over all items of a container in a linear order. At any stage of its life time an Iterator object points to one item. Several Iterators may iterate over the same container simultaneously. The item concept in LEDA [MN95] is a primitive variant on Iterators. Obviously, the Iterator pattern is more powerful than the straightforward "first/next" strategy, where the container class itself provides a method first and a method next to pass all items. It is also more powerful than a "forall" loop construct.

Command: Very often, classes work together in a client/server relation. That is, the client class invokes the methods of the server class to delegate some basic operations, but not vice versa. To keep the coupling of both classes loose, one can define a class whose only intent is to moderate the client/server relation. A class serving this purpose is called a Command.

Builder: A Builder class separates an algorithm from its output. More precisely, it serves as a "one–way" pipe. Instead of constructing the data structure itself, the algorithm delivers all computed information to the Builder in a preformatted manner. The Builder is tailored to one specific representation of the data structure to be constructed, and it constructs this data structure according to the information received from the algorithm.

Decorator/Adapter: An Adapter wraps a new interface round a given class to adapt it to a new context. A Decorator is an Adapter where the new interface equals the old one or extends it, and the functionality of the class is extended or its semantics is changed (or both). These two patterns are treated independently of each other, because they usually apply to completely different situations.

Observer: In the simplest case, this is an interplay of an observed class and an observing class. An object of the former class provides a notification method, which is called with an object of the latter class as a parameter. This method inserts the parameter in an internal list of the observed object. The latter class provides another notification method: Whenever the observed object executes some of its own methods, it calls this notification method for each object of the observing class in its internal list.

A simple example are Smart Iterators defined on a sequenced container. A container object maintains an internal list of all Iterators that refer to elements of this container. Whenever an element is inserted in the container or removed

from it, each Iterator in this internal list is notified in advance. If the Iterator refers to the element being removed, this Iterator forwards itself before the removal operation takes place. Otherwise, the Iterator must only update its position number (if the Iterator class provides one).

References

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows*. Paramount Publishing International, 1993.

[Boo91] Grady Booch. *Object–oriented design with applications*. Benjamin/Cummings Publishing Comp., Inc., 1991.

[Fla95] Bryan Flamig. *Practical algorithms in C++*. Coriolis Group Book, 1995.

[Fre87] Greg N. Frederickson. Fast algorithms for shortest paths in planar graphs with applications. *SIAM J. Comput.*, 16:1004–1022, 1987.

[FZZ94a] Arne Frick, Walter Zimmer, and Wolf Zimmermann. Karla: an extensible library of data structures and algorithms. Technical report, Universität Karlsruhe, Fakultä für Informatik, 1994.

[FZZ94b] Arne Frick, Walter Zimmer, and Wolf Zimmermann. On the design of reliable libraries. Technical report, Universität Karlsruhe, Fakultä für Informatik, 1994.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns, elements of reusable object–oriented software*. Addison–Wesley Publishing Company, 1994.

[GS93] G. Gallo and Mario G. Scutella. Toward a programming environment for combinatorial optimization: a case study oriented to max–flow computations. *ORSA J. Computing*, 5:120–133, 1993.

[GT88] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum flow problem. *J. of the Association for Computing Machinery*, 35:921–940, 1988.

[KLM+93] Dietmar Kühl, Arfst Ludwig, Rolf H. Möhring, Rudolf Müller, Jörn Schulze, and Karsten Weihe. ADLIPS user manual, 1993. http://www.informatik.uni-konstanz.de/~weihe/manuscript s.html#paper16.

[Mey94] Bertrand Meyer. *Object–oriented software construction*. Prentice–Hall, 1994.

[MN95] Kurt Mehlhorn and Stefan Näher. LEDA: a library of efficient data structures and algorithms. *Communications of the ACM*, 38:96–102, 1995.

[Näh95] Stefan Näher. LEDA manual version 3.1. Technical Report MPI-I-95-1-002, Max–Planck–Institut für Informatik, 1995.

[Omo91] S.M. Omohundro. The Sather language. Technical Report 3, Int. Comp. Sci. Inst. Berkeley, 1991.

[Sou94] Jiri Soukop. *Taming C++*. Addison–Wesley, 1994.