Designing Software with Modula–3

Peter Klein
Lehrstuhl für Informatik III
RWTH Aachen

Peter Klein
Lehrstuhl für Informatik III
Ahornstr. 55
RWTH Aachen
52074 Aachen, Germany

e–mail: pk@i3.informatik.rwth–aachen.de
Tel.: +49/241/80–21311
Fax.: +49/241/8888–218

1. Introduction

In the middle of the sixties, the world of software production saw its first great crisis. Software development so far had mainly to do with the implementation of mathematical algorithms which were more or less simple to write down but very time–consuming to compute. As the size and complexity of software systems grew dramatically, programmers were faced with a whole new dimension of problems:

- No decomposition schemes for complex tasks existed. There was no experience how to structure the problem into manageable units and how to describe this structure in a formal notation.
- In large software systems, many people were working on a project without sufficient knowledge about how to organize the course of the project, i.e. how to plan, run, and supervise the single activities comprising the project.
- In the early days of programming, people often had a notion of efficiency which proved to be inadequate for the development of large systems. Now, it is not so important to save a few bytes here and some processor cycles there, but adaptability, portability, maintainability and the like became the crucial features of software.
- The programming languages existing at that time (and unfortunately heavily used until today) were not suited for the new class of problems to be solved.
- Complex products were developed without a systematic knowledge about how to ensure certain quality requirements.

Some of these problems are still unsolved and important research topics.

In 1968/69 (cf. [NR68], [BR69]), a new approach to the software development process was advised. Strong similarities between the production of software and classical engineering tasks lead to the conclusion that the techniques devised in engineering sciences should be applied to software production also, and the term *software engineering* was coined. This was a clear rejection of the notion of programming considered as an "art" (cf. [Knuth68]).

One of the early results of software engineering was the development of so–called *life–cycle models* which were aimed to describe the activities occurring during the software development process, their results, and their dependencies. Many of them have been published so far, mostly differing only in their granularity (cf. e.g. [Boehm82]). The rise of new programming paradigms though has introduced a new kind of life–cycle models in the past years (cf. e.g. [Agresti86]). They can be called *continuous* models in contrast to the *discrete* models, because they do not consider the activities during software development as separated units of work with a clearly defined result. Instead, they view program development as an evolutionary process where the product – starting with a rapid prototype more or less automatically derived from an abstract specification – is transformed into a "better" system in a sequence of refinement steps. Although some of the ideas behind this scheme are quite attractive, this paradigm has not yet passed the test of being appropriate for large systems in classical application areas.

In this paper, we follow another idea: Activities are not assigned to phases as in most discrete life–cycle models. This leads to the impression that they are executed in a chronological order, which is unrealistic in most cases. Instead, we group the activities according to the logical level on which they operate. Fig. 1 shows our so–called *working area model* (cf. [Nagl90]).

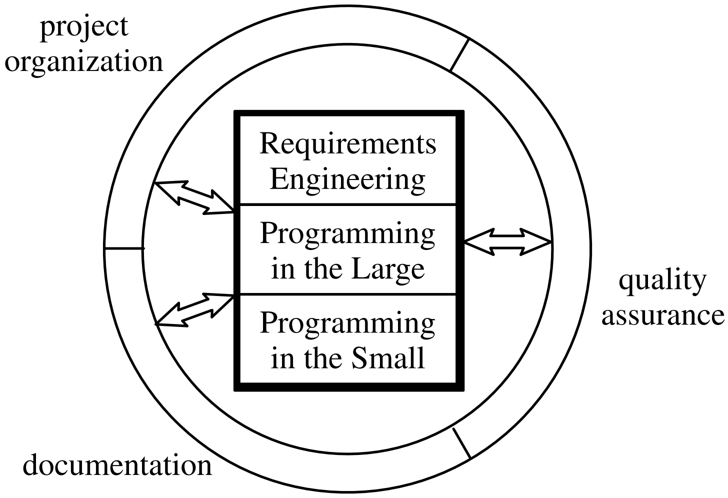


Fig. 1: The working area model

The box shows the three basic working areas of software development:

- The working area *Requirements Engineering (RE)* consists of all activities which serve the specification of the outside behavior of the system to implement. Often, this specification is made in collaboration with the (potential) purchaser of the system. Activities in this working area include:
  - Analyzing and modelling the given situation,
  - modelling the desired future situation,
  - checking the requirements specification of the system for feasibility, and
  - modifying the requirements specification in accordance to new user demands.

  The central document of the RE working area is the *Requirements Specification*.
- The *Programming in the Large (PiL)* area covers all activities dealing with the software system as a whole. Some example activities are:
  - Analyzing the requirements specification,
  - identification of basic design units,
  - specification of the interactions between the design units,
  - checking the architecture for feasibility,
  - integration of implementation units,
  - checking the system against the requirements specification, and
  - modifying the architecture in accordance to new requirements.

  The central document of the PiL area is the *Design Specification* or *System Architecture*, in the following mostly called architecture or design.
- In the *Programming in the Small (PiS)* area, the actual implementation of the system is performed. The PiS area contains:
  - Analyzing and understanding the part of the architecture to be implemented,
  - implementation of the design units in a programming language,
  - testing of the implementation units,

- checking of the implementation units against the design specification, and
- modifying the implementation e.g. in the case of errors or new design specifications.

The central document in this area is the *source code* of the system.

Beside these basic working areas, every activity during the project has to be organized, planned, and documented. Furthermore, it has become common practice for large projects to perform an additional checking of all results by a central authority for quality assurance.

In the past years, the main interest slowly changed from PiS to the other working areas. Although much good work is done in all areas, the research area of program design has experienced the most drastic boost. Many people (including the author) consider the architecture as the most important part of software development. The reason for this is simple: the main expenditure during the development process concerning manpower, time, and money is still the PiS. A good design makes PiS easy in that the programmer can concentrate on a problem with a comprehensible complexity. Errors made in the implementation can be found and eliminated more easily because realization details are encapsulated in modules. For the same reason, software systems are more adaptable and portable than before. On the other hand, errors made in the design may lead to an enormous waste of implementation efforts. One may argue that the same dependency holds for requirements specification and architecture, but a good and adaptable architecture always represents a set of similar requirements, so changes in the requirements are readily integrated into a good design.

Especially the meteoric rise of the object–oriented paradigm has led to a host of design methods (cf. e.g. [WWW90], [Booch91], [CY91], [RBPEL91], [CABDGHJ94]). Unfortunately, the enthusiasm about the ”invention” of the object–orientation shows the features of an ideology. The object–oriented paradigm introduces a new terminology and ”reinvents” things which were already known for a long time. Sometimes it seems as though experiences in software design are neglected deliberately because they do not fit into the scheme, or they might even be lost completely because not so few people think that there was no software design before object–orientation showed us the way. The author is far from condemning the ideas of object–orientation, on the contrary it is their great merit that the research on the design topic has made great advantages recently.

Compared to the publications cited above, the approach presented in this paper does not claim to be a method, i.e. it does not claim to know how the design of an arbitrary software system can be devised. In our opinion, this is not possible at all. If we remember the similarities between software design and classical engineering tasks, this would mean that a uniform method could solve problems in constructing a car as well as when building a house. In order to solve specific problems, we have to study certain application and system classes in detail; e.g. the problem class of compiler construction shows how the careful investigation of an application class leads to very strong results which are transferable to other problem classes afterwards.

Nevertheless, the comparison between software design and classical engineering techniques is not completely suitable. Just as a programming language is a universal tool to solve arbitrary implementation tasks, we can think of a design language which can describe arbitrary software systems. This paper proposes such a language. Although it is not attached to a method which also tries to commit the designer on how to model a problem, it helps very well with the design because it is not settled on the programming language level. As we will see, it restricts the usage of a programming language in some points, on the other hand it is able to express details which cannot be mapped (directly) onto one of the programming languages available today. So, our approach is more a methodology which can be extended to a method if sufficient knowledge about the application area exists.

The architecture language presented here has a long tradition and has been revised several times (cf. e.g. [Altmann79], [Gall82], [Nagl82], [LN85], [Lewerentz88], [Nagl90], [Börstler93]). In some respects, it has the same history as Modula–3 when compared to its predecessors (cf. e.g. [Wirth82], [Lampson83], [RLW85], [Rovner86], [CDGJKN88], [CDGJKN89], [Nelson91], [Harbison92]): starting with a basic notion of modularity, it has seen quite some extensions and smoothing of "rough edges". The most striking similarity is the careful integration of inheritance into the given conceptual framework. We therefore consider Modula–3 an ideal PiS "partner" for our architecture language. Of course, a design in our language may be mapped onto (almost) any other programming language with more or less effort, too.

In the next chapter, the basic concepts of our architecture language will be presented. Then, we discuss some general design problems and the correct usage of the architecture language. These parts are essentially adapted from [Nagl90]. Finally, the integration of Modula–3 and other programming languages with the architecture language will be dealt with.

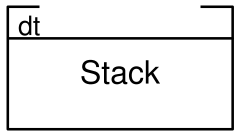
2. A module concept

As mentioned in the introduction, one of the crucial results of the PiL working area is the *decomposition* of the system under development into *manageable units*. In this paper, we will call such units *modules*, although this term is generally used in various different ways. Even in the context of Modula–3, the word module denotes something different than what is meant here. There is no precise definition of what a module is, but we will try to give the reader a notion of the term in that we enumerate some characterizations.

- A module is a *logical unit* of the system with a clearly defined purpose in a given context. Typical for such a unit is that its purpose can be described in one sentence.
- A module represents a *design decision*. The sum of all design decisions is apparent from the architecture.
- A module is an *unity* consisting of *data and operations*.
- A module is a unit with a certain *complexity*.
- A module offers some resources to the rest of the system. We call all of the resources offered by one module the *interface* of that module.
- The internals of a module are encapsulated, i.e. the module can only be used through its interface. We call these internals the *body* or *implementation* of the module.
- A module can be viewed as an *abstraction*: the interface provides access to "abstract" resources, the module abstracts from the realization of these resources.
- A module is *replaceable* by another module with the same interface without changing the semantics of the system.
- The *correctness* of a module can be shown independent of the rest of the system by checking the realization against the interface.
- A module can be *developed independently* from other modules. This includes the implementation, testing, and documentation of the module. A module is therefore the basic *unit of work* during the implementation of the system.
- A module is the basic *unit of reuse* in a system. Many modules can be used in a different context or system than in the one in which they were developed.

In the following we will see that a module in the above sense is not an arbitrary collection of objects featured by the programming language, but there is a small set of *module types* which describe a certain kind of abstraction. Before we discuss these types, we introduce the notation we want to use in this paper.

Our notation is a mixed graphical/textual language. The system's overall architecture is described by a graphical overview which shows the modules constituting the system. For every module, a separate textual specification of the module's interface exists. The graphical representation of a module named Stack looks like this:



The notation expresses our idea that a module is a unit consisting of an interface and a body, the latter not accessible from the outside. The dt entry identifies Stack to be a data type module; this will be explained later. Additionally, the module will be accompanied by an interface specification. This specification describes the resources the module offers to its clients. For a stack, typical resources would be push, pop, or read.

2.1. Module Types

One of the characteristics for modules given above is that a module provides an abstraction of how the interface resources are implemented in a programming language. Basically, two types of abstraction can be distinguished:

- *Functional abstraction* is at hand if the module has some kind of *transformation character.* This means that an interface resource transforms some kind of input data into corresponding output data. Functional abstraction facilitates the hiding of algorithmic details of this transformation.
- *Data abstraction* is present if the module encapsulates the access to some kind of *memory.* The module hides the realization of the data representation. The module's interface only shows how the data can be used, not how it is mapped onto the computer's storage.

Because functional abstraction is quite close to the human way of thinking, we start our discussion of module types with *functional modules*. Exactly because functional abstraction is comparably easy to understand, it was and is heavily used by people without design experience: it is quite natural to decompose a complex task into several smaller tasks. A typical example is that for every function at the user interface, one module exists to implement this function. If the function is too complex for one module, it delegates some subfunctions to other modules. When the module dependencies are drawn as a graph, the result looks more or less like a tree.

One big problem with this kind of functional decomposition is that similarities between modules are easily overlooked and the same functionality is implemented several times. And, what is even worse, details about the representation of data structures are not encapsulated, since several functional units (spread over several modules) need to know this representation. This resulted in the badly maintainable systems often found today. Therefore, many object–oriented methods do not support functional abstraction at all or only with little attention. But this leads to equally incomprehensible architectures, since functional modules do have a justified place in the design. Actually, they can be found often in transformation problems: e.g. the phases of a compiler like scanner, parser, analyzer, and code generator would be functional modules in our sense. Note that the module names already suggest some sort of ”active” character of functional modules, whereas data abstraction modules mostly have ”passive” behavior.

Let us now look at an example for a functional module's interface description.

```
FUNCTIONAL MODULE INTERFACE Scanner;
   (* Provides functions to convert a text stream into a token stream. *)

IMPORT TextStream, TokenStream;

EXCEPTION ScanError;                    (* no valid token could be scanned *)

PROCEDURE IsTokenAvailable(text: TextStream.T): BOOLEAN
            RAISES {TextStream.ReadError};
   (* Returns TRUE if another token is available on the text stream. *)

PROCEDURE GetNextToken(text: TextStream.T; tokens: TokenStream.T)
            RAISES {TextStream.ReadError, TokenStream.WriteError, ScanError};
   (* Reads one token from the text stream and writes it to the token stream. *)

END Scanner.
```

This example allows some further remarks on functional modules. An important property of functional modules is that they *may not contain memory* unless some code inside the module's body is executed. In other words, as soon as the execution of an interface resource of the module is finished, the module has no knowledge about previous calls. The reason for this restriction is that a functional module with a state commonly contains two implementation decisions in one module: one for the actual functionality of the module and one for the state of the module, which should be modelled as a separate data abstraction module instead. Note that any module with an internal memory can be made stateless if the state is stored elsewhere and either read by the functional module or passed to every function by the client. Note also that the restriction does not mean that the body of a functional module may not have any global variables: e.g. it is absolutely legal if a module computing trigonometric functions uses an internal table of key values, or if our scanner module uses a buffer of cached characters. This is no violation of the principle that the mapping of input to output values is independent of the run–time "history" of the module (though it may depend on the history of data abstraction modules used by the functional module).

Another point worth mentioning is that our language does not support means to define (or even describe) the semantics of interface resources. We assume that a description of a resource's functionality is given as a plain text comment. If a more formal description of the interface is required, we propose that an appropriate specification language like Larch (cf. [GHM90], [Jones91]) is used.

Before we take a closer look at data abstraction modules, let us summarize some frequent application areas for functional modules:

- Modules performing control and coordination tasks (e.g. the program's main module),
- transformation problems like the scanner example,
- evaluation tasks (which is actually a special class of transformation problems), and
- supporting services on data structures. This will be explained later.

In general, functional modules solve some subtask of a more complex task.

Unlike functional decomposition, the *data abstraction principle* has been neglected for a long time. The central idea is that the *representation* of a data structure cannot be accessed directly, instead the client has to read or modify the structure only through a given set of *access operations*. The data structure and its implementation are blended into a unity.

In our language, data abstraction is supported by two module types, namely *data object* and *data type modules*. We first present data object modules through an example.

```
DATA OBJECT MODULE INTERFACE SymbolTable;
   (* Provides storage for symbols during context sensitive analysis. *)

IMPORT Symbol;

EXCEPTION
   SymbolDeclaredTwice;                    (* the symbol is already in the table *)
   SymbolNotDeclared;                      (* the symbol is not in the table *)

PROCEDURE Init();
   (* Wipes out the table. *)

PROCEDURE Store(symbol: Symbol.T) RAISES {SymbolDeclaredTwice};
   (* Stores a symbol. *)
PROCEDURE Remove(symbol: Symbol.T) RAISES {SymbolNotDeclared};
   (* Removes a symbol from the table. *)

PROCEDURE Get(ident: TEXT): Symbol.T RAISES {SymbolNotDeclared};
   (* Returns the symbol with the given name. *)

PROCEDURE IsDeclared(ident: TEXT): BOOLEAN;
   (* Returns TRUE if the symbol with the given name is in the table. *)

END SymbolTable.
```

As is obvious from the interface, the module encapsulates some kind of memory, i.e. the module ”remembers” data between interface procedure calls. In our design language, data object modules are the *only* ones which have a *static state*, i.e. a state which is visible on the design level. We will come back to this later. Note also that data object modules may not export types.

The last module type in our language is the *data type module*. Again, we start with an example.

```
DATA TYPE MODULE INTERFACE TokenStream;
   (* Abstract data type for a stream of tokens. *)

IMPORT Token, StreamMode;

EXCEPTION
   IOError;                          (* general input/output error *)
   WriteError;                       (* could not write to the stream *)
   ReadError;                        (* could not read from the stream *)
   ModeError;                        (* operation not allowed in this mode *)

TYPE
   T <: Public;
   Public = OBJECT METHODS
              open(name: TEXT; mode: StreamMode.T): T RAISES {IOError};
                 (* Opens and returns a stream with the given name.
                 The stream will be in read, write, or append mode depending
                 on the mode parameter. *)
```

```
                close() RAISES {IOError};
                  (* Closes the stream. *)

                read(): Token.T RAISES {ReadError, ModeError};
                  (* Reads one token from the stream. *)

                write(token: Token.T) RAISES {WriteError, ModeError};
                  (* Writes the token to the stream. *)

                rewind() RAISES {IOError};
                  (* Resets the stream so that the next read/write returns/writes
                  the first token in the stream. *)

                isTokenAvailable(): BOOLEAN RAISES {IOError, ModeError};
                  (* Returns TRUE if another token is available from the stream. *)
              END;
END TokenStream.
```

A data type module exports *exactly one type*. To be more precise, it exports one type identifier (see below) and the access operations for instances of this type. In this way, data type modules provide the client with means to create an arbitrary number of objects of that type. Although data type modules are templates for the creation of memory, they may not contain a global state, i.e. the execution of an interface resource on an identical instance of the type always modifies this instance in the same way. Of course, as with functional modules, this does not mean that the body of the module may not contain global information, but operations on one instance of the type may not have visible side–effects on other instances. In this sense, all state information visible for clients of the module is contained in the instances of the type and therefore dynamic: no actual memory exists until some clients instantiates the type.

The exported type has always the name T. So, if DataType is a data type module, DataType.T always denotes the type identifier exported by DataType. The Public identifier is used to denote the publicly accessible part of the type and should not be used by the client. Note that the textual interface description is redundant here, because the DATA TYPE MODULE clause already determines that a data type is exported by the module. But since we want to derive a PiS interface from this description later, we distinguish between the PiL and the PiS parts of the description, even if some PiS code could be derived from the PiL part. We will encounter a similar situation in the context of specialization.

In the distinction between data object and data type modules, our approach differs from most other design methods. Commonly, these methods distinguish between class diagrams (which would only use data type modules) and object diagrams. Class diagrams are considered static, whereas object diagrams are used to describe the dynamic behavior of the system. In our language, we decided to facilitate *only a static description* of the system. We think that our architecture language should only describe facts which do not touch the Programming in the Small area. (The same argument will occur again in the section on module dependencies.) And, in general, it is not possible to decide which objects exist in some dynamic state of the system without looking at the implementation. Of course, the designer will have some kind of dynamic behavior of his system in mind. But since we want a strict distinction between design and implementation, the designer cannot be sure whether the actual objects existing in the system will exist and behave that way. On the other hand, if the designer wants to annotate some sort of memory in the system, data object modules provide means to do that in a way the implementation cannot ignore.

A common misunderstanding is that data object modules are an annotation for the fact that exactly one instance of a certain type exists. Therefore, there should also be annotations for arbitrary cardinalities, e.g. to determine that exactly four instances of some type exist. But in our language, data object modules are not instances of some type, they are just an encapsulation of global system state. Of course, data object modules can be easily converted into data type modules and vice versa, but this changes the semantics of the system. And the definition of arbitrary cardinalities would again interfere with our separation of PiL and PiS. Let us consider the example from above: the designer wants to express that some data object or some type's instances contain four instances of another type, like a car which contains four wheels. In that case, the interface might contain resources like getLeftForewheel etc. Note that this is all the designer can lay down without looking at the implementation: if the design allows the car module to instantiate wheels, he cannot assure that it does this exactly four times. But, for the design, this does not matter at all as long as the object behaves as a car. A more difficult case would be that the designer wants to lay down that *exactly* four instances of a type exist. Although this case is rare, it might occur when a situation has to be modeled very close to reality. Consider e.g. we want to model a computer system with four disk controller cards all managed by the operating system. In the real world, the operating system simply cannot create a new disk controller at run–time because it contains hardware elements. The operating system can only work with the controllers which are physically present. If the designer does not want to abstract from this situation, he will have to introduce four data object modules with identical interfaces. Actually, we do not care whether the chips and resistors on all these cards are the same, or whether some of them might contain completely different hardware elements. Analogously, it is of no concern if the implementations of the four data object modules are different or not, as long as they implement the same interface. On PiS level though we might want not only to ensure that the implementations are identical, but also to avoid four (identical) copies of the source text. Although this is not really a design topic, we will come back to this problem in the context of generics.

2.2. Module relationships

After presenting the different kinds of basic design units, we now introduce the means we provide to describe interactions between these units. First of all, some modules want to make use of resources offered by other modules. We distinguish three different logical levels on which such dependencies can be discussed:

1. The first prerequisite for the interaction between a client and a server module is that the design allows the client to access the resources offered by the server. We call this the *usability level*.
2. If the architecture allows some module to access another module (on the usability level), the client may make use of this by actually using some of the server's resources (e.g. if the client's implementation contains a call of a procedure offered by the server). This use is static, i.e. it can be determined by looking at the implementation of the client. Therefore, it is on the *static uses level*.
3. If a static use of some client resource is executed during the run–time of the program, we say that this is on the *dynamic uses level*.

The previous section already mentioned that our architecture language does not consider PiS aspects. Therefore, the design cannot determine whether the implementation makes use of a resource or not. It can only *allow* some module to use another module's resources. So, if we talk about some module using or importing another module, this is always on usability level.

Besides the *usability relationships* from above, we also have *structural relationships* between modules. These are used to express structural design concepts and allow or forbid certain usability relationships.

We start our discussion with a relationship called *local containment*. This is a structural relationship describing that some module is contained in another module. From this containment, some rules derived from the block–structuring idea present in many programming languages follow. This relationship forms a *local containment tree* in the module dependency graph. Placing a module in such a tree means that the module can only be accessed from certain parts inside this tree. Consider fig. 2 for an example.

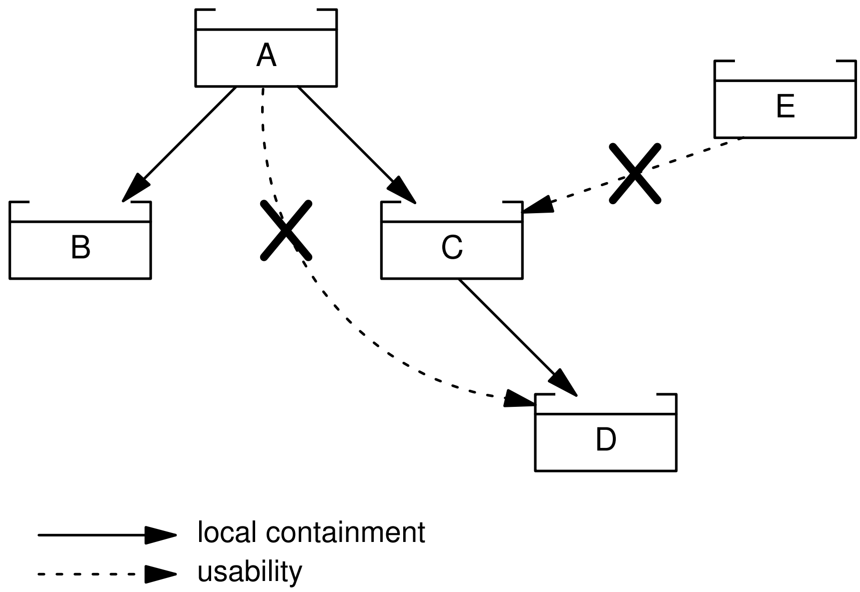


Fig. 2: A simple local containment tree

Module E may not use C because it is contained in A; analogously A cannot use D because it is contained in C. The purpose of local containment is to *hide internals* of the tree from the outside. In this sense, it introduces *information hiding* on the architecture level. Let us summarize what usability relationships are possible in a local containment tree: A module can use itself, its successors, its predecessors, and direct successors of all predecessors (especially its brothers). This is completely analogous to the rules of locality and visibility in block–structured programming languages like Algol, Pascal, and Modula–3. We say that *potential local usability* exists between the module and its above mentioned relatives in the containment tree.

One problem with the relationship of potential local usability is that many relationships are made possible between modules which are not necessary. For a quite simple containment tree, fig. 3 shows the bulk of resulting potential local usability relationships.

Of course, the design should give a more detailed description about which of these relationships are really useful. We therefore do not use the potential local usability. Instead, we introduce the *local usability* relationship. Local usability is a relationship which is explicitly drawn into the architecture, although such a relationship may only be defined between modules for which a potential local usability exists. In other words, the local usability relationships are a *subset* of the potential local usabilities *defined by the designer*. Most local usability edges are parallel to local containment edges. But especially in recursive problems, this is not always the case. Fig. 4 shows two examples for typical situations. On the left side we have a containment tree with local modules locally used by their parents. Such structures can be found e.g. when a complex functional module delegates parts of the functionality to other functional modules. The right side shows a part of a recursive descendent parser: the module for compiling statements uses the module for compiling expressions because statements may contain expressions. But e.g. the module compiling factors makes recursive calls to the Expr module because expressions may occur in factors.

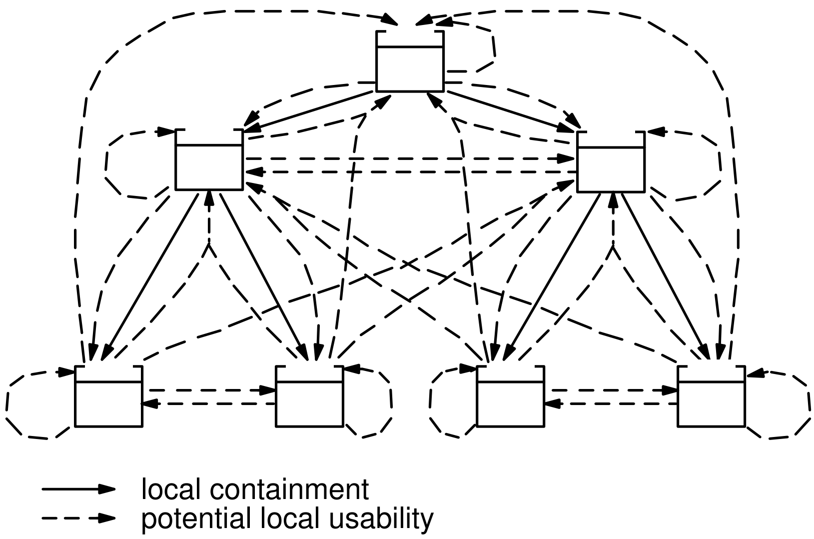


Fig. 3: Potential local usability

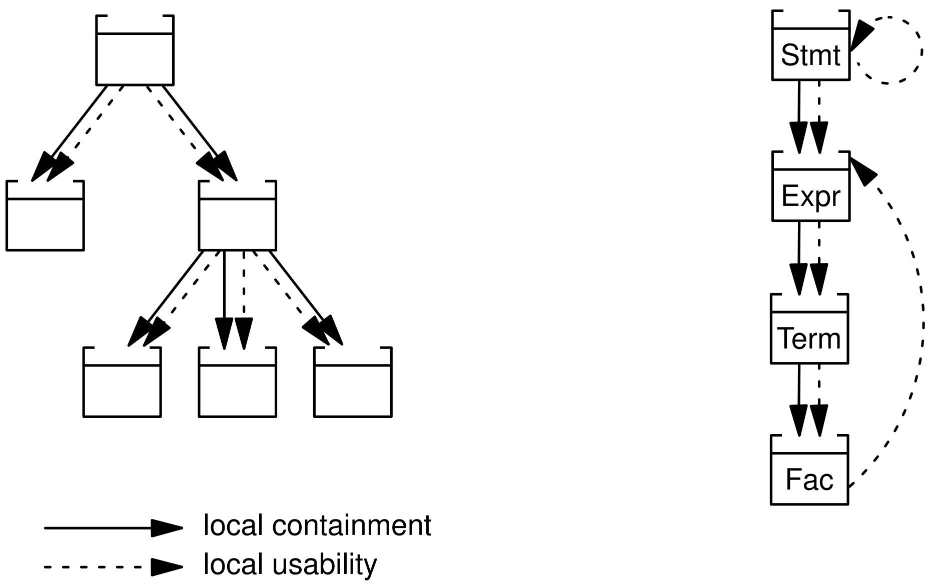


Fig. 4: Examples for local usability

Note that for every local containment edge, there has to be a parallel local usability edge. Otherwise, the local containment edge would be senseless. Nevertheless, we keep these edges in the diagram for clarity. In our textual interface description, we include a special clause for expressing the local containment relationship. The local usability relationship is denoted by an IMPORT statement. For the example from fig. 4, we would write

```
FUNCTIONAL MODULE INTERFACE Expr CONTAINED IN Stmt;
IMPORT Term;
...
END Expr.
```

The local usability relationship introduced so far is not suited for all situations where one module wants to use resources from another module. This is particularly the case if some module should be usable by arbitrary clients, possibly from different containment structures. Consider the situation where two modules A and B want to use resources from a general library module H. If we would restrict ourselves to the local usability relationship, we could insert H neither under A nor under B, because the local containment structure would forbid the other one to access H. The only solution in which both modules can access H would be to insert it under some common parent of A and B in the containment tree, as is shown in the left side of fig. 5. This does not only violate our rule that a parallel local usability edge for the containment edge exists, it also distorts the semantics of the architecture: H is a low–level service and should therefore be below A and B. Additionally, if the need for H's resources should occur in another containment tree, the designer would have to duplicate H or he has to introduce a new common root for the modules using H. In the latter case, the independence of the two containment trees would be lost.

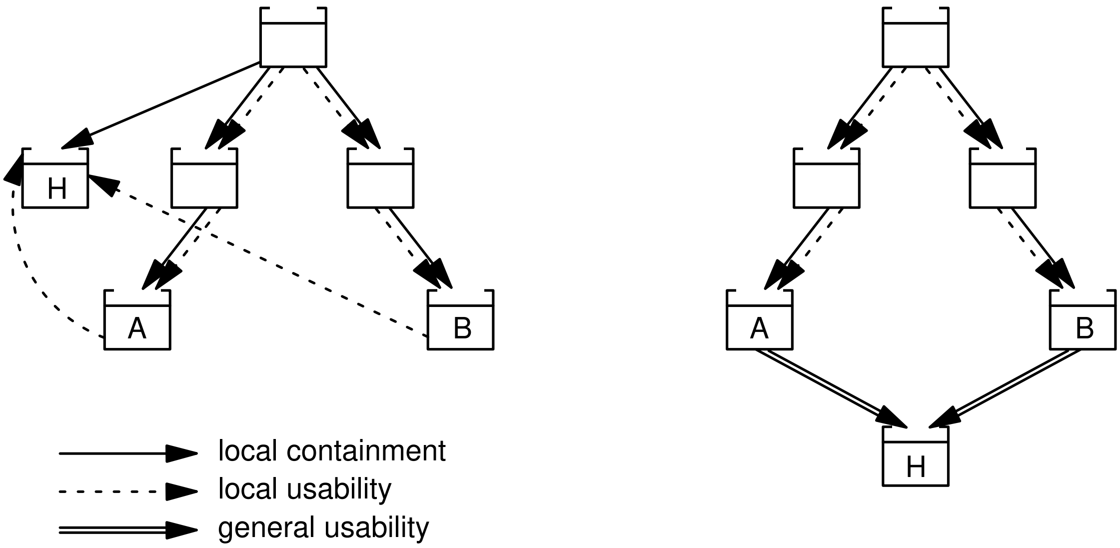


Fig. 5: Local vs. general usability

The obvious solution for this situation is given on the right side of fig. 5. This diagram introduces a new kind of usability relationship: the *general usability*. With this relationship, the designer has means to express that a module exports general resources which can be accessed from all other modules for which a corresponding general usability relationship exist. Note that there is no structural relationship directly connected to general usability: general usability edges from any part of the system can end in one module. On the other hand, it is not unusual that only one client uses some server module through general usability. Whether or not a module is generally or locally usable is a question of the kind of server module. If a module offers general services to clients, it should be inserted using the general usability. If it offers services which are only useful in a certain context, it should be inserted using the local containment and local usability relationships. Of course, a module can only be generally usable if it is not contained in another module.

In the textual interface specification, we use the IMPORT clause again to denote a general import. We do not have to distinguish between local and general imports here because the existence or absence of a CONTAINED IN clause in the server module already determines which kind of usability is possible. We use different line styles in the diagram notation only for clarity.

The other structural relationship in our language besides local containment is the *specialization relationship*. Although the concept of specialization is influenced by ideas from object–oriented programming languages, we want to make a clear distinction between PiS and PiL aspects here. When we talk about specialization, we mean a certain relationship between modules *on the* *design level*. Whether the actual implementation language's type system supports specialization in some way should not influence our architecture.

The specialization relationship can only exist between data type modules. To some extend, specialization is quite similar to general usability: both express that one module uses another module to implement its interface. But the semantics of specialization describes a very restricted case of general usability. If some data type is a specialization of another data type, this implies that every instance of the special type has *at least all the properties* of an instance of the general type. We call the special type a *subtype* of the general type, which is vice versa referred to as the *supertype* of the special type. This terminology extends to arbitrary ancestors and predecessors of a data type module in the specialization hierarchy. Therefore, the specialization relationship is also called *subtype relationship*. We want to avoid this term because it might be confused with the subtyping feature of some programming language.

An important characteristic of the specialization relationship is that the set of operations offered by the subtype is a superset of the set of operations on the supertype. We therefore do not have to repeat these operations in the subtype's interface. For an example of specialization, reconsider the TokenStream interface from the previous section. We now want to specify a specialization of this data type which is able to perform additional navigation operations on a stream. Again, the specification is redundant in so far as the specialization relationship is denoted on PiL level with the IS A clause as well as on PiS level with the T <: Public; Public = TokenStream.T construction (which declares T as a subtype of TokenStream.T in Modula–3).

```
DATA TYPE MODULE INTERFACE SearchableTokenStream IS A TokenStream;
   (* Provides token streams with navigation operations. *)

IMPORT TokenStream, Token;

EXCEPTION NotFound;                          (* No such token in the stream. *)

TYPE
   T <: Public;
   Public = TokenStream.T OBJECT METHODS
                  searchNextOccurence(token: Token.T)
                        RAISES {NotFound, TokenStream.IOError};
                     (* Skips all tokens up to and including the given token. *)

                  searchPreviousOccurence(token: Token.T)
                        RAISES {NotFound, TokenStream.IOError};
                     (* Rewinds to the last previous occurence of the given token. *)
               END;
END SearchableTokenStream.
```

Note that all operations defined for the type TokenStream.T are available for instances of SearchableTokenStream.T, too. We say that the supertype's operations are *inherited* by the subtype. This does not necessarily mean that the implementation of an inherited operation in the subtype is really the same as the supertype's implementation, this is a question of PiS.

The notion that an instance of a subtype has all the properties of its supertype allows us to introduce the *substitution principle* which says that a subtype's instance may appear wherever a supertype's instance is required. In our example from above, this means that a procedure which takes a TokenStream.T instance as input or produces a TokenStream.T instance as output may actually take or return a SearchableTokenStream.T instance instead. Note that the specialization relationship as defined above guarantees that it is safe to do so: if some operation requires a TokenStream.T instance as input, it does not have to care whether the actual parameter might be a SearchableTokenStream.T, because all operations it might perform on the object are available on SearchableTokenStream.T instances as well. For a client expecting some function to return a TokenStream.T instance, the situation is the same: it can safely ignore that the return value might have more properties than it expects, because the properties of the object returned are definitely a superset of what is ensured by the interface description of the supertype.

A common case in the context of specialization is that several subtypes of one data type exist. Note the semantical difference between a data type module accessed by other data type modules via general usability and two subtypes of a common supertype (cf. fig. 6): whereas on the left side A provides some *arbitrary resources* which are used by independent modules B and C, the right side shows a data type module A which represents the *common properties* of B and C.

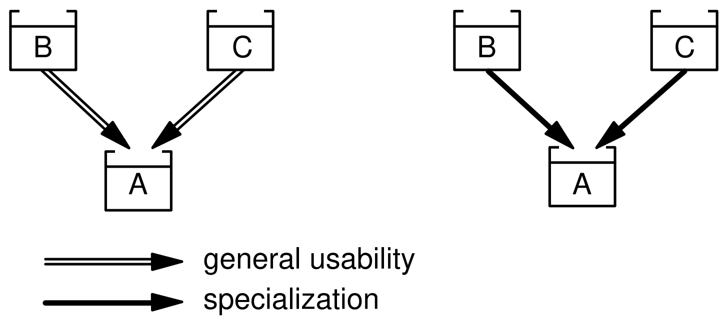


Fig. 6: General usability vs. specialization

In the situation given by the right side of fig. 6, we can distinguish several sorts of operations defined for the data type offered by A: sometimes, we want to express that all subtypes of A export a certain operation, but A cannot implement this operation since it depends on other properties of the subtype. For an example, consider the data type Symbol.T from the previous section. Now we want to define subtypes of Symbol.T for the different types of symbols, e.g. variable symbols, constant symbols, procedure symbols etc. Some properties are common for all specializations of Symbol.T, e.g. every symbol has an identifier. This should therefore be a property of Symbol.T. Other properties may be unique for one subtype, e.g. a type symbol (in Modula–3) might contain a pointer to the structure class it belongs to. Other properties might be shared by several subtypes, like a flag whether the symbol denotes a writable designator or not. In this case, an appropriate specialization of Symbol.T should be created for the encapsulation of these shared properties. Finally, there are properties which are common to all specializations of Symbol.T, but they are implemented differently for every subtype. The standard example would be to ask some symbol about its kind. Although it should be possible to perform this operation on arbitrary symbols, it cannot be computed without knowledge about the specific subtype of the symbol in question. In this case, we talk about *virtual operations* and of *virtual data types* if some type contains at least one virtual operation. An important consequence is that virtual data types must not be instantiated, i.e. a client may not create an object of a virtual data type – they are just helpful for structuring the specialization hierarchy. In our textual interface description, we denote a virtual property by using a NIL assignment. Note that we do not define whether or not a subtype redefines some supertype's operation, this is a PiS question. But using virtual operations as shown above, we can lay down that every subtype *has to* (re)define this operation if the subtype should have instances. If it does not define the operation, the subtype is again virtual.

```
DATA TYPE MODULE INTERFACE Symbol;
...
TYPE
   T <: Public;
   Public = OBJECT METHODS
                  ...
                  getKind(symbol: T): SymbolKind.T := NIL;
                  ...
               END;
END Symbol.
```

As with local containment and local usability, the structural specialization relationship is accompanied by a usability relationship, namely the *specialization usability.* A subtype module needs to import at least the supertype's type identifier. In general, the subtype's implementation will also want to call operations of the supertype. To make such an access possible, we introduce specialization usability edges in our graphical architecture notation. Again, we do not distinguish between specialization usability and other kinds of usability in the textual notation because this can be determined from the graphical notation or from the context of the module. Note that the structural relationship of generalization implies no usability relationships: a subtype module may not use some supertype's operation just because it is a subtype, all usability relationships have to be introduced explicitly by the designer. On the other hand, the structural relationship again determines the set of possible usability relationships. Every subtype module must have a specialization usability edge to its immediate supertype module and may have additional specialization usability edges to arbitrary predecessors in the specialization hierarchy. The introduction of the specialization usability may sound unusually restrictive for readers with experiences in object–oriented programming, but the reasons for the differentiation of a structural and a usability aspect of the specialization relationship are the same as those discussed for the local containment and local usability relationships: the structural relationship itself allows far too many potential usabilities, and only a few of them actually describe a dependency between modules. So, we consistently distinguish between relationships which show the structure of the system and relationships which denote usabilities and therefore dependencies between modules.

2.3. Subsystems

To conclude the presentation of our architecture language, we introduce the concept of *subsystems*. Obviously, the module level introduced so far is too fine–grained for the description of large software systems. We therefore need design units which allow a *hierarchical specification* of the architecture. Subsystems allow the designer to express such units which are ”greater” than modules: they may contain an arbitrary number of modules and other subsystems. Most of the characterizations given for modules at the beginning of this chapter can be applied to subsystems as well: first of all, subsystems are *units of abstraction*. They have an interface which describes the resources which can be accessed from the outside. The interface of a subsystem is a composition of explicitly selected modules and/or subsystems inside the subsystem. Furthermore, subsystems are *units of work*, *units of testing*, and *units of reusability.* For this reason, subsystems should – just like modules – obey the rules of *low coupling* and *high cohesion*: the modules and subsystems should not interact more than necessary with other units on the same design level. On the other hand, a module or subsystem should only contain logically related resources. We shall use the term *component* in the following if we mean a module or a subsystem.

As stated above, the designer decides explicitly which interfaces in the subsystem contribute to the subsystem's interface. An immediate consequence is that we cannot assign each subsystem a unique type as we could with modules. We therefore do not distinguish different subsystem types. Nevertheless, we sometimes talk about functional, data object, or data type subsystems if the subsystem's interface consists of one or more modules of the corresponding type.

Of course, being part of a subsystem is a structural relationship between the corresponding components. Accordingly, we already introduced two natural candidates for subsystems in the previous section, namely local containment trees and specialization hierarchies. We will treat these structures as subsystems without further notice. Of course, the designer can insert additional subsystems on intermediate hierarchy levels consisting of subtrees or subhierarchies. Note that the interface of a local containment subsystem is implicitly given by the root of the tree.

The textual description for subsystems in our language can be taken from the following example:

```
SUBSYSTEM INTERFACE SymbolManagement;

EXPORT Symbol, SymbolTable;

END SymbolManagement.
```

Since the "implementation" of subsystems is also on design level, there is an additional notation for the body of the subsystem.

```
SUBSYSTEM BODY SymbolManagement;

CONTAINS Symbol, SymbolTable;

END SymbolManagement.
```

In our graphical notation, we distinguish between a *black–box* and a *white–box* notation for subsystems (cf. fig. 7). The bottom–up arrows indicate that a module contributes to the subsystem's interface.

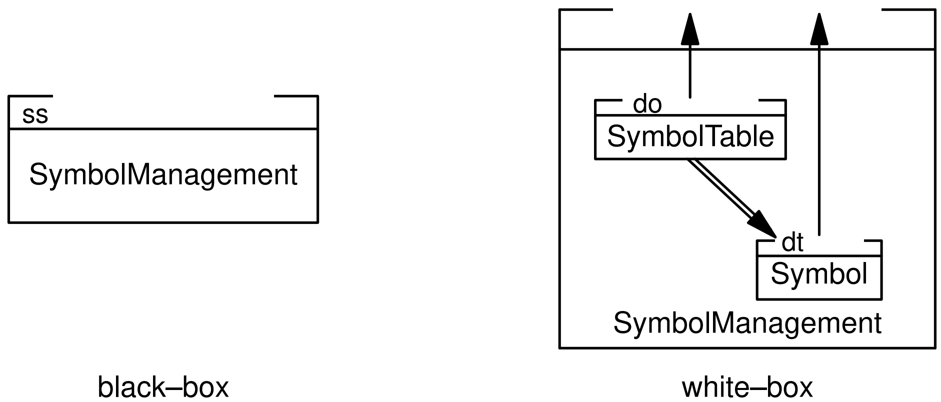


Fig. 7: Black–box and white–box notation for subsystems

As was already mentioned above, we treat containment trees and specialization hierarchies as subsystems, i.e. for every containment tree we implicitly add

SUBSYSTEM INTERFACE *RootModuleName*System;
EXPORT *RootModuleName*;
END *RootModuleName*System;

SUBSYSTEM BODY *RootModuleName*System;
CONTAINS *<All modules in the containment tree>*;
END *RootModuleName*System;

and analogous subsystems for specialization hierarchies.

2.4. Architecture description constraints

This section summarizes the syntactical and semantical rules an architecture specification should obey.

On the *graphical architecture description level*, we have the following constraints:

- All component names in the architecture are unique.
- Every component has at least one incoming usability edge.
- Data abstraction modules may not use their own interface, i.e. edges leaving a data abstraction module may not enter the same module.
- A component is contained in at most one other component, i.e. at most one local containment edge can end in a component.
- Every local usability edge has to be consistent with the local containment structure, i.e. there has to be a potential local usability relationship between the components (cf. fig. 3).
- For every local containment edge, there has to be a parallel local usability edge.
- No general usability edge can end in a locally contained component.
- Specialization edges can connect only data type modules.
- Every specialization usability edge has to be consistent with the specialization hierarchy, i.e. specialization usability edges may connect only modules which are directly or indirectly related by specialization edges.
- For every specialization edge, there has to be a parallel specialization usability edge.
- No specialization edge can end in a locally contained module.
- A component contained in a subsystem can be accessed from the outside only if it contributes to the interface of the subsystem.

On the *textual architecture description level*, there are the following constraints:

- The resources exported by some module must match its type. A functional or data object module interface may contain procedures, constants, and exceptions. A data object module should contain some operation Init to (re–)initialize the internal state. A data type module interface exports exactly one type and may contain constants and exceptions. The type is named T and is described as an opaque object type as shown in the section on module types. It should have at least one init operation to initialize an instance of that type.
- The resources in some interface should be as orthogonal and coherent as possible, i.e. some operation should not be replaceable by a combination of other operations. Furthermore, all operations should be semantically closely related.
- The names of all the resources offered by some interface must be unique.
- The resources imported from some other module must be exported by this module.
- Every imported resource should be used.
- The parameter types used in the interface of some module have to be importable by all possible clients of that module.

Finally, there are dependencies between the textual and the graphical level:

- For every component in the graphical description, there has to be a textual description and vice versa.
- All imports in the textual description must be consistent with corresponding edges in the graphical description.

Note that most of these rules can be statically checked by an appropriate tool for the specification of architectures according to the proposed language. Such a tool is currently under development; some details follow in section 4.1.

3. Some methodological remarks

We already stated in the introduction that we do not consider our approach as a method which shows *how to model* arbitrary software systems, it only facilitates the *architecture specification* of software systems. On the other hand, we do think that using our architecture language helps to avoid common design errors in that it is strictly restricted to the PiL level. We neither lay down how the architecture for a given problem can be obtained, nor do we explicitly support or even enforce the use of certain programming language features. But this does not mean that we have no idea about how the architecture language might be used, or how a design can be mapped onto some existing programming language. This chapter discusses some points about the correct usage of our language, while the next chapter takes a closer look at the integration of a given design with programming languages in general and Modula–3 in particular.

3.1. On module types, relationships, and architecture levels

The module types and relationships presented for our architecture language all have certain conceptual relatives in the world of programming languages. Whereas e.g. the local usability relationship has its roots in the ideas of structured programming languages and their nested scopes, the general usability relationship stems from the import feature of languages like Ada, Modula–2, and Modula–3. Data type modules and specialization are supported to a high degree by object–oriented languages, and functional modules are the classical decomposition units in most of the other imperative and functional languages. In contrast to many existing design approaches, we do not value some module types or relationships higher than others, each has its rightful place in almost any software system. In the following, we try to give a rough notion of where this place might be.

First of all, let us consider what mistakes are frequently made when common decomposition strategies are applied.

- The *top–down strategy* tends to prefer the concept of locality: systems developed top–down often have a more or less tree–like architecture. These trees reflect the functional decomposition of complex tasks into smaller subtasks, so functional modules are predominant. This approach is prone to code duplication and to spreading data realization details over the system, because common functionality is easily overlooked and data abstraction is disregarded.
- The *bottom–up strategy* on the other hand tries to develop generally (re)usable units. This often clutters the architecture with unneeded functionality. Data abstraction is not really supported by this approach, but more easy to realize than with a top–down decomposition.
- Many modern design methods facilitate some sort of *object–centered strategy*. Data abstraction and classification are massively supported, but coordination and integration aspects are neglected.

We can learn here that it is generally not a good idea to model the vertical architecture levels separately. Instead, the design should always consider *all* levels of a (sub)architecture *at the same* *time*. If this results in too many components in one diagram, the design should resolve the problem by introducing new *hierarchy levels* and treat the resulting subsystems as black boxes. Of course, these black boxes have to be specified then in the same manner. This should lead to a decomposition in which every subsystem and the complete system itself can be designed and understood as a comprehensible (sub)architecture with a clearly defined interface. Note how this scheme stresses our notion of subsystems as units of work and abstraction on the PiL level.

Although this is not generally true, many systems show a typical pattern in the way certain module types and relationships occur on certain architecture levels. Whereas the higher levels consist mainly of functional components embedded in containment structures, lower levels are dominated by generally usable data abstraction modules. This is plausible in so far as that functionality concerning controlling, integrating, and dialog handling is encapsulated in the higher levels of the system. Basic layers on the other hand often serve collection or bookkeeping tasks which need data structures to store their information. Fig. 8 shows a rough sketch of the global layout of such a system.

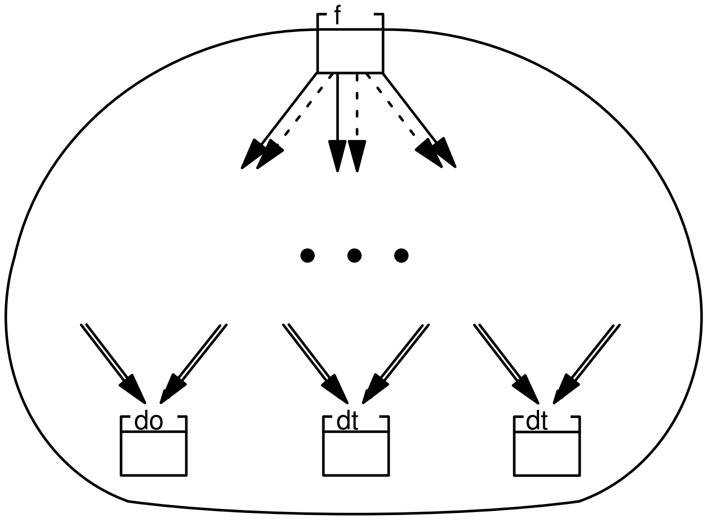


Fig. 8: Module types and relationships on different architecture levels

3.2. On data abstraction and functional abstraction

One of the most serious mistakes made by unexperienced designers is the wrong application of data and functional abstraction. Until now, we only introduced the module types responsible for the representation of the appropriate abstractions. If these module types are used properly (i.e. respecting the restrictions we imposed), some common mistakes can be avoided. First of all, every module should be *created* with a *given* type. Otherwise, it is often hard to assign a type to a module afterwards. This is almost always a sign of a badly constructed module which encapsulates more than one implementation decision. In this section, we try to give some hints on how to choose the correct module type. Basically, the following two rules should be obeyed:

- The designer should always consider the *interface* of a module, *not the implementation*. For example, in some data object module for a stack, the interface suggests that all exported resources like push or pop form a unit operating on a data structure. In the implementation, one might think that these operations are functional, because they ”compute” a transformation on the global data structure by manipulating pointers, computing indices or the like.
- The designer should always consider the *module itself*, *not the surrounding system* in which it is embedded. Every single module represents a design decision, and the type of the module depends on this decision. For example, a module controlling dialogs in a database application is functional, although the system as a whole serves the preservation of states.

Let us now take a closer look at the symbol table example from the previous chapter. Suppose we want to store the symbols in a balanced binary tree. Since some tree operations like rebalancing are quite complex, one might have the idea to extract this functionality into a functional module (cf. fig. 9a). This would result in components like

```
FUNCTIONAL MODULE INTERFACE TreeTraversal CONTAINED IN SymbolTable;
...
PROCEDURE SearchNode ...;
...
END TreeTraversal.

FUNCTIONAL MODULE INTERFACE TreeBalancing CONTAINED IN SymbolTable;
...
PROCEDURE Balance ...;
...
END TreeBalancing.
```

The problem with this design is that delegating complex tree operations to an intermediate functional level demands that the SymbolTable module reveals the tree's internal data structure to the functional modules, so the data abstraction principle is violated: many modules would have to be adapted if the internal representation for the tree is changed.

Another idea would be to insert a module for search trees below the symbol table (cf. fig. 9c):

```
DATA OBJECT MODULE INTERFACE SearchTree;
...
PROCEDURE InsertEntry ...;
PROCEDURE FindEntry ...;
...
END SearchTree.
```

The SearchTree solution has the disadvantage that it reduces the operations inside SymbolTable to trivial tasks: most operations have not more functionality than to pass their parameters to the corresponding SearchTree operation. On the other hand, the implementation task for SearchTree is now almost as complex as for the SymbolTable module before.

As the most advantageous solution, we could implement the table on top of a module for balanced binary trees (cf. fig. 9b):

```
DATA OBJECT MODULE INTERFACE BalancedTree;
...
PROCEDURE InsertLeftChild ...;
PROCEDURE GetRightChild ...;
PROCEDURE DeleteNode ...;
...
END BalancedTree.
```

In general, stacking data abstraction layers like above is preferable to introducing intermediate functional levels. On the other hand, every data abstraction level should represent a noticeable, but not too great change in the view on the data structure as it appears in the interface. If such a data structure cannot be found, it might be appropriate to insert functional components between data abstraction modules.

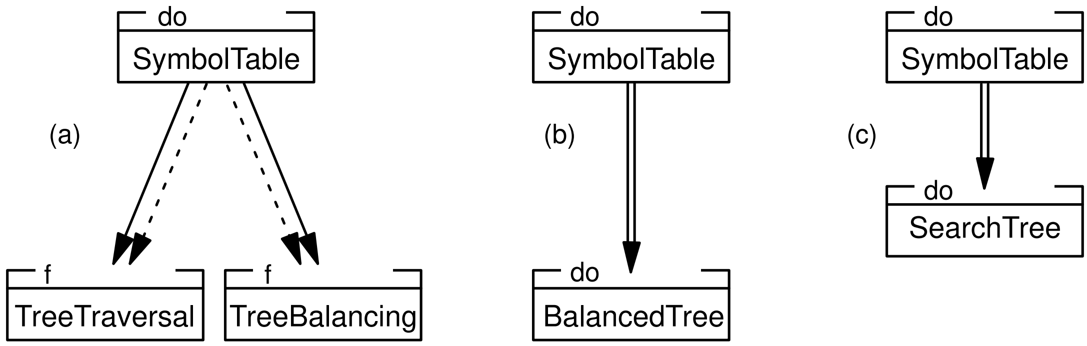


Fig. 9: Designing below data abstraction modules

A more common case of an interaction between functional and data abstraction modules can be found in transformation situations. Here, some sort of computation maps complex data onto other complex data. For an example, let us consider the scanner example from the previous chapter again. In the graphical notation, this module would be embedded as shown in fig. 10.

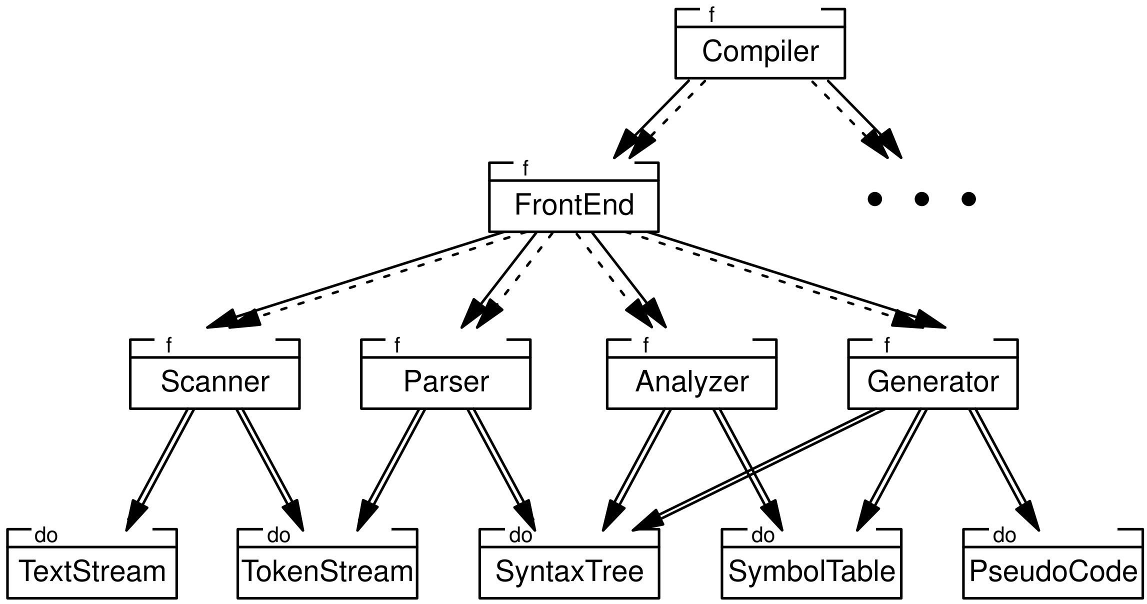


Fig. 10: A compiler as an example for transformation systems

Although it is quite obvious how to model such situations within our architecture notation framework (and especially considering the compiler example), two serious design errors are frequently made in comparable systems:

- In older systems, we often find that data abstraction decisions are ignored, i.e. the realization of the containers between the functional components is not properly encapsulated. This results in complex data streams flowing between the functional components. Any change of the internal representation of the data results in appropriate changes in the functional part of the system.
- Object–oriented design methods generally follow the motto that an object should know (and encapsulate) all operations some client wants to perform on the object. Such ideas tend to blur the distinction between functional and data abstraction, because decisions about how to *store* information are not separated from decisions about how to *handle* this information. This often tempts the designer to attach all functionality of the system to certain (often arbitrary) data abstraction components.

In other words, while the first approach tends to move data representation information into functional components, the second makes the same mistake the other way around. Apart from the fact that the design looses comprehensibility because major implementation units simply vanish from the architecture, in the end both violate abstraction principles in the same way and with the same consequences.

3.3. On entries and collections

This section introduces another natural candidate for subsystems besides the already mentioned local containment trees and specialization hierarchies. To begin with, let us consider some design units which are commonly modelled as data abstraction modules:

- First, we have complex structured *records*. On the abstract (interface) level, such objects look like compounds containing a set of (named) fields which may be accessed individually by read/write operations. On the implementation level, a record may be realized as a corresponding record object in the given implementation language, as a set of tables connecting the composed object to its fields, as a compressed bit array, as a linked list of fields, etc.
- Second, there are *collections* of other data abstraction units. On the interface level, these are aggregations which can be manipulated through operations like store, remove, find, and change a certain element. The interface determines the algebraic properties of the aggregation (set, bag, map, stack, queue, relation, sequence, graph, etc.) by supplying one or more access paths to the contained elements (indexed, by key, first–in–first–out, last–in–first–out, etc.) and ensuring certain constraints (e.g. that an element may be stored only once in the collection). On the realization level, there is a wide variety of implementation options for collections, like heap implementations (B–Trees, hash tables, linked lists, etc.), persistent implementations (in a database, file, etc.), remote implementations (on some node in a network, as a separate process, etc.), and many more.

Often, records do not exist outside of some context, i.e. they are usually stored in some sort of collection. (If they are implemented as heap objects, they are *always* stored in at least one collection, namely the runtime heap of the program system.) We already encountered such a situation in the previous chapter where symbols were stored in a symbol table. Because it is a common pattern in any architecture to store objects in a collection, we take a closer look at these *entry–collection situations*.

First, we note that it is important to model the two data abstraction decisions for the entries and collections as separate modules. It is a common mistake to mix up both data types into one module, this leads to a bad adaptability and reusability of both data type implementations. Furthermore, as was already mentioned in the previous chapter, it is generally appropriate to model an entry–collection situation as a subsystem. Accordingly, we call these *entry–collection subsystems*. We now want to discuss some common patterns for entry–collection subsystems. To begin with, a quite simple pattern was shown in fig. 7. It is sketched again in fig. 11.

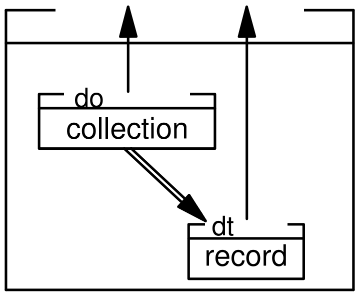


Fig. 11: A simple entry–collection subsystem

In this case, the collection is a data object module. Furthermore, the entries for the collection are records. In other situations, it might be necessary to have more than one collection in the system. Then, we have to design the collection as data type module. Furthermore, we might want to store these collections in another collection: this situation is sketched in fig. 12.

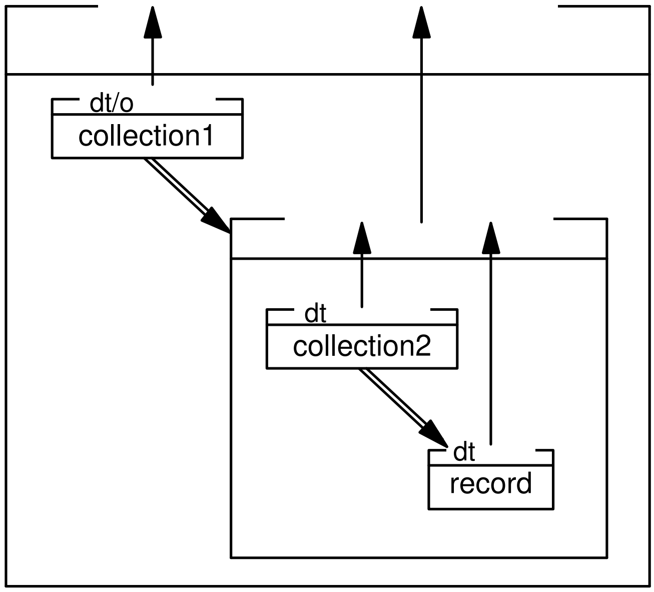


Fig. 12: Collections as entries

The main collection here could be a data object module or a data type module (which, in the latter case, can be stored in some collection again).

A more thorough investigation of the symbol table example shows two extensions of the pattern shown in fig. 11. First, the collection (SymbolTable.T) is implemented using a binary tree as shown in fig. 9. This binary tree is a collection of symbols as well, so we actually have a situation as in fig. 13.

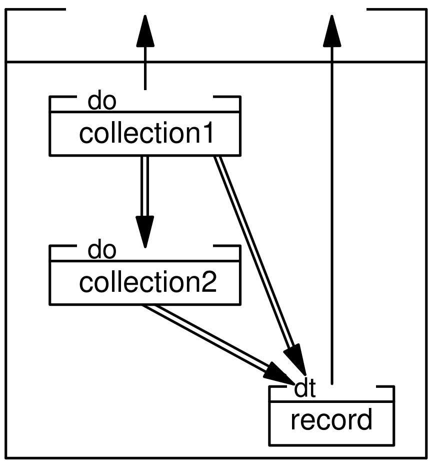


Fig. 13: Multiple layers of collections

The other extension which can be found in the symbol table example is that there is not only one fixed entry type, but actually all instances of subtypes of the symbol type can be stored in the collection. This, again, is a typical situation. The collection is called *heterogeneous* (in contrast to *homogeneous*) in this context, because instances of different types can be stored in it (cf. fig. 14). The types of possible entries are not arbitrary though, they are all subtypes of some base type. The collection itself deals only with this base type, and the substitution principle allows us to use subtype instances instead.

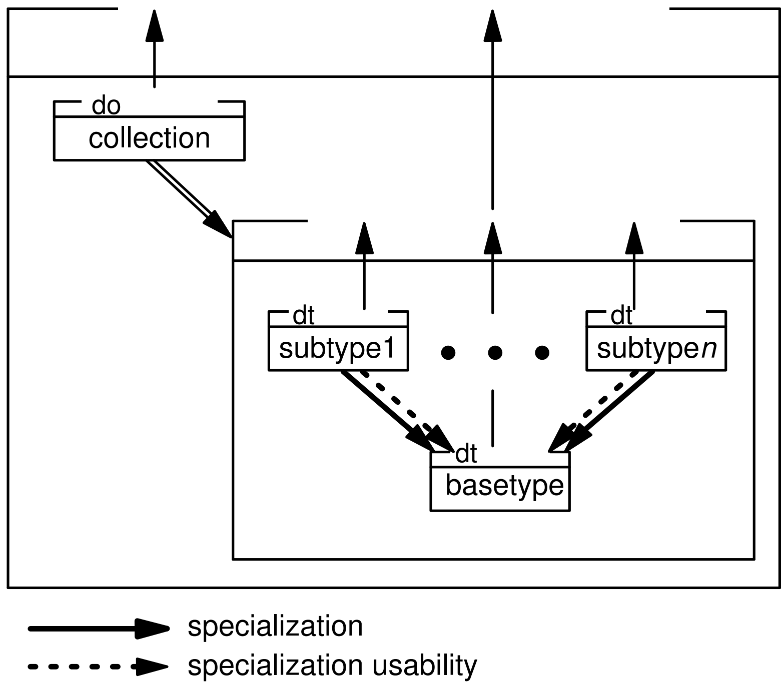


Fig. 14: Heterogeneous collections

Although the entry types in a heterogeneous container as discussed above can be different, they are related on architecture level by specialization relationships. In other situations, we find heterogeneous collections where the entry types are only grouped on a semantical level. When we consider e.g. a data type for graphs, we find that a graph is a collection of nodes and edges. The resulting architecture situation is shown in fig. 15.

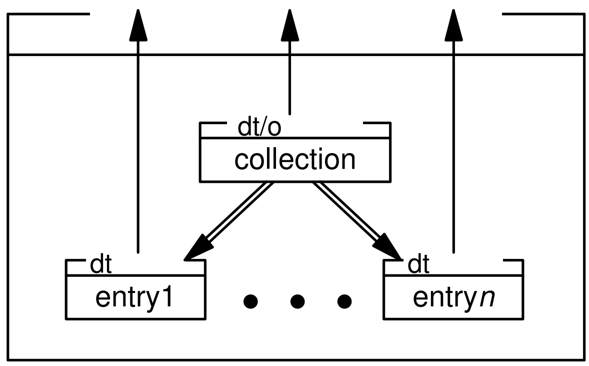


Fig. 15: Collections with different entries

To conclude this chapter, we want to summarize some essential properties of entry–collection subsystems.

- Entries are usually records but can be arbitrary data type modules or data type subsystems in general. Common applications for subsystems are specialization hierarchies and entry–collection subsystems. There can be more than one entry type, in this case the possible entry types are usually the subtypes of some given base type.

- Collections can be data type modules, data object modules, and data abstraction subsystems. A common case is that the collection is implemented on top of another collection.

3.4. On reusability

In the introduction we already mentioned some reasons why we consider the PiL working area to be important. One of these reasons is that a good design leads to software components which are *reusable* in some way. Since reusability is generally accepted to be an essential part of software design considerations, we want to discuss some aspects of reusability in the following.

We can distinguish several strategies to support the development of reusable and maintainable software systems. The first strategy is to *identify general system components*. The problem of finding general components can be discussed best on the graphical architecture level. From our personal experience, thinking too closely in terms of interface definitions or in terms of real–world objects leads to a design where similarities are interpreted wrongly or overlooked. Also, note that there are quite different degrees of generality. A module or subsystem can be general in the sense that it is used by more than one other component in an architecture. We denoted this in our graphical architecture specification by means of general usability edges. Another kind of reusability would be if some component is implemented as a process which can be accessed by different other application processes at run–time. The most important kind of reusability though is the property of components to be usable in more than one software architecture, i.e. it could be used in other (possibly quite different) systems than the one for which it was created originally. Part of the sensation caused by the rise of the object–oriented paradigm is due to the fact that object–orientation is said to support this reusability to a considerable degree. The idea behind this concept is that, in the course of time, a vast library of general components will be developed and made available by programmers and software companies all over the world. These components can be integrated into a system under development by defining appropriate subtypes, i.e. an increasing amount of code does not have to be programmed anew but can be ”inherited” into the new system. But, of course, identifying general components actually has got nothing to do with abstract data types or specialization. Nevertheless, libraries with reusable data types are indeed quickly evolving. Currently, the greater problem seems to be to *find* a module in these libraries which provides some sort of service. See [Börstler93] for a discussion of reusability and classification of software components.

The second strategy to be presented here is to design software according to the concept of a *unit construction system*. The units of our construction kit are modules and subsystems. Following this idea, not only one fixed system, but a set of related variants of a software system can be designed by assembling some or all of the given construction units. Our compiler example would be a typical candidate for such a system: e.g. global and local optimization phases would be construction units which may or may not be configured into a concrete system. A front–end for some source language and a back–end for some machine architecture would be units which can be ”plugged together” with other units for intermediate code generation etc. The main difference between this strategy and the previous one is that *all* system components are considered to be construction units. On the other hand, most of the components are reusable only in the different variants of the system.

As a third strategy we consider the *transformation of code into data*. If some functionality of the system should be easily changeable, it is generally better not to hard–wire it into functional components, but to extract the variable parts into data abstraction components. The functional part is thereby reduced to the evaluation of the data part. For example, in compiler construction, it is a widespread technique not to write specific scanners and parsers for every source language, but to reuse existing functional driver components with exchangeable scanner and parser tables.

Another example would be to put error messages or other user interface related parameters into tables rather than directly into the program code.

Another strategy is to *generate* a software system or parts of it instead of developing it by hand. There are several aspects of this strategy: one possibility is to derive an implementation or an implementation framework from a specification in a more or less methodological manner. The derivation itself is performed manually, but sufficiently supported by the methodology to reduce the development task considerably. An example for this strategy is the derivation of a recursive descendent parser from a given grammar (cf. [Wirth86]). A similar approach can be found in the JSP/JSD method (cf. [Jackson83]). Another possibility is to generate parts of the system automatically from a description using appropriate tools. These parts might be executable code or tables which can be evaluated by drivers. Common examples for this approach are compiler compilers which generate scanners and parsers (or scanner and parser tables to be used with standard drivers) from a language description (cf. [ASU86]) or graphical user interface builders. A quite unspecific example for this strategy is to instantiate generic templates, which will be discussed again later. Finally, it is sometimes possible to directly execute an abstract problem specification, i.e. some tool can interpret the specification and simulate the operational behavior of the specified system. An example for the latter can be found in [Zündorf94]: complex data structures are modelled as typed graphs, and operations on them are described using graphical replacement rules. An integrated tool supports direct interpretation of such specifications as well as generating code for a rapid prototype. A common property of all generation methods is that it is not the generated code which is reusable, but the method or the tool which generates it. In the case of direct specification execution, reusability even raises to the specification level in the sense that parts of one specification might be reusable in others. In other words, program generation differs from the other strategies mentioned before in that it supports reusability on the level of the development process, not on the level of the resulting system. The generated (part of the) system may or may not be ”designed” according to the previously mentioned strategies.

Eventually, we want to discuss the strategy of *finding similarities*. Although this sounds banal, all other strategies are special cases of this ”meta–strategy”, and it can be very difficult to apply it. There can be similarities between system components, architecture patterns, architecture frameworks, designing techniques, application classes, and many more. The direct representation of this strategy in the PiL area is to define a reasonable specialization structure in an architecture, i.e. to analyze similarities between data abstraction components and to put common properties into appropriate supertypes. A more abstract and also more important aim is to develop a *standard architecture* for a given application class. Such an architecture contains general components common to all systems designed according to the standard architecture. Other components have to be adapted for a peculiar system: in the best case, specific components can be generated by tools, or by supplying appropriate tables. In the worst case, they have to be hand–coded as usual. Although it is desirable to have standard architectures for all important application classes, only very few classes actually have a generally accepted coarse structure, most noticeably compilers.

At the end of this section, we want to summarize the presented strategies in a diagram.

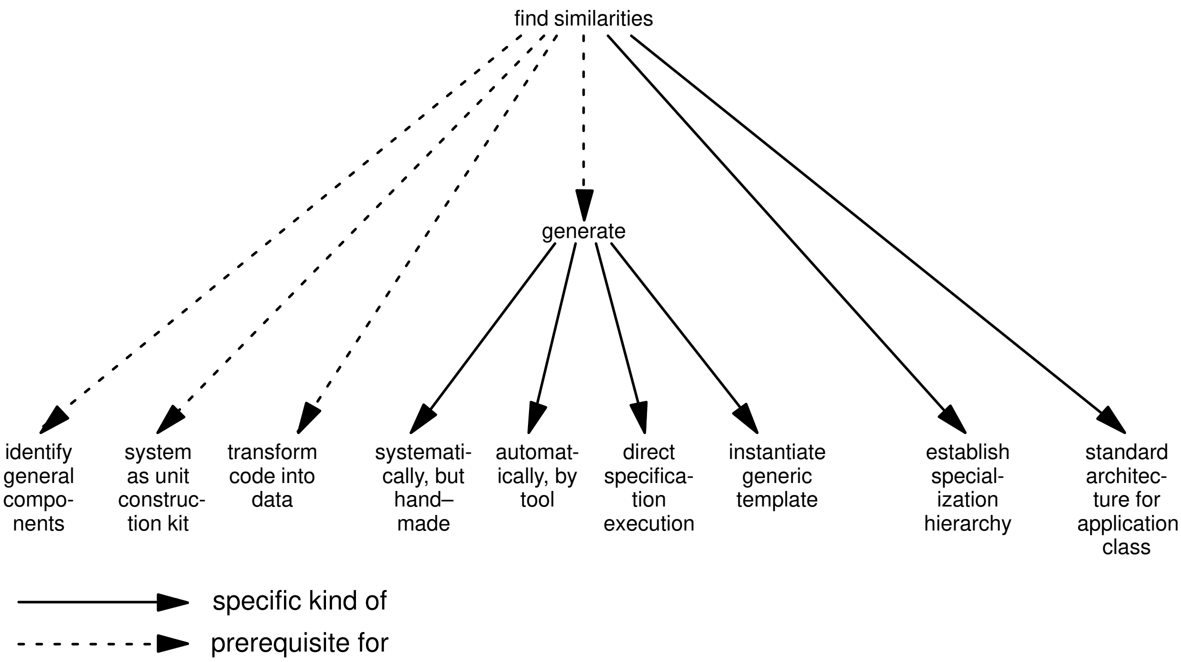


Fig. 16: Strategies for reusability and adaptability

4. Implementing the architecture

This chapter contains a short discussion on how an architecture in the sense of our specification language can be mapped onto an existing programming language. After some general points, we will concentrate on generic situations and their implementation.

4.1. Mapping the design onto Modula–3

As is obvious from the previous chapters, our textual interface description follows closely the syntax and conventions of the Modula–3 programming language – actually, it is syntactically a superset of a Modula–3 interface. Since the design itself is completely on the PiL level, it should be no problem though to apply the ideas of this paper to other implementation languages as well, [Nagl90] discusses some of the transformation issues in detail for Fortran, C, Pascal, and Ada.

Since a module description in our architecture language contains a Modula–3 interface, we can create such an interface directly by simply commenting out the keywords which are not part of the Modula–3 language. It is important though to leave them in the source text in order to remind the programmer of the syntactical and semantical constraints the module types and relationships impose on the implementation. Some of these are not supported by the programming language, so they have to be realized by dint of the discipline of the programmer (possibly with the help of an appropriate tool).

- Modula–3 supports modules, but no module types. The general interface layout for the different module types (e.g. that a data object module may not export types) can be laid down statically, but the semantical constraints as discussed in the section on module types demand disciplined programming. Note that we use the OBJECT notation for all abstract data types. This is for two reasons: First, this notation is easier to read; the unity of data and operations is shown more clearly. Second, it is not the responsibility of a data type module to decide whether some client uses this type through specialization or by another kind of import. In this sense, exporting an object type instead of a simple opaque type with procedures makes the interface more general.

- The only module relationship supported by Modula–3 is the general usability. All other usability relationships can be simulated with general usability, but the additional constraints have to be ensured by discipline. Specialization is supported on type level by inclusion polymorphism, but not on module level.
- Subsystems are not supported by Modula–3.

We now want to summarize the rules which should be obeyed on the PiS level (in addition to the already mentioned rules for the architecture specification itself). Some of them have to be enforced by disciplined programming, others are ensured by the compiler.

- The Modula–3 interface for some module matches the textual interface description with the additional keywords included as comments.
- The resources imported from some other module must be exported by this module.
- Every imported resource should be used.
- All imports in the module body must be consistent with corresponding edges in the graphical description.
- The resources exported by the module must be implemented by its body.
- The statical use of other module's resources is consistent with the usability, i.e. they are imported by the module body.
- The implementation of some module must be consistent with the module type. Note especially that functional and data type modules may not contain any (visible) state.
- Implementations may not use their own interface resources unless explicitly stated in the architecture. This might be useful for recursive problems like the Stmt example in fig. 4, but it is forbidden for data abstraction modules by the constraints from section 2.4.

As was already mentioned, we are currently working on a tool which supports our architecture language and the development of software systems using this language. At the moment, only Modula–3 is supported as PiS language, but the tool is designed to handle other programming languages and even mixed–language systems as well. The tool is able to parse arbitrary source text directory trees and to generate the graphical architecture description for the given sources with automatic layout. The textual interface description is supported as Modula–3 source text with the additional keywords placed in special comments. The user can then create, delete, edit, and compile components from the graphical user interface. During the development process, several rules concerning the correct usage of the architecture language as mentioned here and in section 2.4. are checked. Beside the basic functionality directly connected to the architecture language, the tool is also devised as a platform for the integration of arbitrary additional development aids like revision control systems, problem tracking systems, cross reference tools, debuggers, etc. If you are interested, feel free to contact the author for details.

4.2. Transformation problems

Unfortunately, implementing a good architecture may need more thought than creating an interface and implementation for every module as described above. Two problems occur frequently during the transition from PiL to PiS:

- In many cases, the system under development has to fulfil certain requirements which make the direct realization of the design difficult or even impossible. Important examples for such requirements are:
  - The clean separation of abstraction units with orthogonal interfaces could result in an intolerable loss of time or space performance.

  - Building a distributed system demands a separation of the architecture in client and server portions, probably introducing new modules for handling the distribution itself.
  - Using certain external software components like a user interface library or a database system might force the developer to integrate components which do not fit smoothly into the original design. The same problem can occur with components generated by code generating tools like compiler compilers.
- The restrictions imposed on the usage of the underlying programming language might seem to strict in special situations. For example, if a record data type as introduced in the previous chapter is implemented using a record constructor of the programming language with simple read/write operations for every field, and it is obvious that this will not change during the lifetime of the system, it might seem sensible to export this data type directly instead of encapsulating it using an opaque object type.

The latter point is generally a matter of arrangement. In the end, the implementation of an interface description is a result of applications of transformation rules. These will be definitely different if different target languages are to be supported, but even for one target language the rules may or may not grant a certain freedom in how to implement an interface. Especially for Modula–3, it is strongly advised to map the textual module interface description from the architecture as closely as possible onto the implementation source text. Nevertheless, the development team might decide to drop the restriction that the Modula–3 interface is always identical to the interface description except for the additional keywords. But such a decision should always be thoroughly devised and documented in order to avoid confusion and violation of abstraction principles.

The first problem mentioned above is more serious. Quite often, a good design is discarded because certain requirements could not be met. Or, what is even worse, finding a good design was not even attempted because of the prejudice that a clean architecture and obeying certain (especially performance) restrictions are conflicting goals. Indeed, it is not unusual that problems like those listed above demand modifications to an architecture. But this does not necessarily mean that the original design was futile. On the contrary, the modified architecture might not be understandable at all if there is no documentation about the original design ideas. We therefore suggest that the original architecture is still considered to be the central document on the PiL level, even though it might not be implementable in a straightforward way. In this case, we talk about an *ideal* or *virtual architecture*. All design considerations should be discussed on the basis of this virtual architecture. If necessary, the virtual architecture has to be transformed into a *concrete* or *real architecture*. Again, this transformation should be described and documented as formally as possible. In this way, the virtual architecture with the original design decisions is preserved. Modifications to the system architecture can be distinguished in those who concern the functionality and structure of the system itself and those concerning secondary requirements. If other secondary requirements have to be met, only the corresponding transformations must be changed without affecting the virtual architecture. Analogously, a given set of transformations can be applied again on a modified virtual architecture to yield an appropriate mapping onto a new concrete architecture.

4.3. Generics

This last section deals with a mechanism which is technically not on PiL level, but important enough to deserve a special discussion. The main idea of genericity is to write (generic) *templates* for system components. In the template, an arbitrary number of details is not wired into the code, instead the template code refers to these details using formal parameter names. The programmer can then create a concrete component by supplying the missing details in the template. We call these details (generic) *parameters*, and the process of creating a component using a generic template and generic parameters (generic) *instantiation*. Accordingly, the resulting component is called a (generic) *instance*.

A common example for generic templates are collections as discussed previously. The generic parameter for the template is the entry type for the collection. The reason for this approach is that many collections are more or less independent of the type of objects they can store, i.e. the interface and implementation of the collection do not refer to special properties of the entry type. A stack module for example looks about the same whether it is a stack of integers or a stack of strings, they only differ in that the identifier INTEGER is consistently replaced by the identifier TEXT when used as the entry type. Therefore, it would be obvious to write a generic template for stacks which refers to a formal identifier Entry instead of a concrete type identifier. A stack of integers can then be created by instantiating the stack template, supplying INTEGER as generic parameter.

In the following, we want to make some restrictions to the general definition of genericity:

- We consider the instantiation of a generic template as a static process, executed before the actual compilation of the program system. This conforms to the generics facility offered by some programming languages like Modula–3, C++, Eiffel, and Ada. This approach is also quite easy to simulate in any other imperative programming language.
- For reasons of clarity, we use only modules as generic parameters, i.e. the template expects its formal parameters to be replaced by module names in the instantiation process.

The first point explains why generics are actually not a PiL subject: the generic templates do not exist on the design level, the architecture only shows generic instances. Using generic instantiation to create a component is merely a technique to avoid several source code copies which differ from each other only in a very restricted way. Nevertheless, our architecture language allows generic templates to be inserted in the specification. The reason for this is that the same argument which is used for the introduction of generics on the PiS level holds for our textual interface specification as well: we avoid multiple interface specifications which can be actually derived from a generic specification through instantiation. In this sense, the following description of how to insert generic templates into the design can be regarded as an optional extension of the architecture language as defined in chapter 2. The graphical notation looks like in fig. 17. Note that the second restriction mentioned above makes it easy to understand the dependencies between template, parameters, and the instance from the specification.

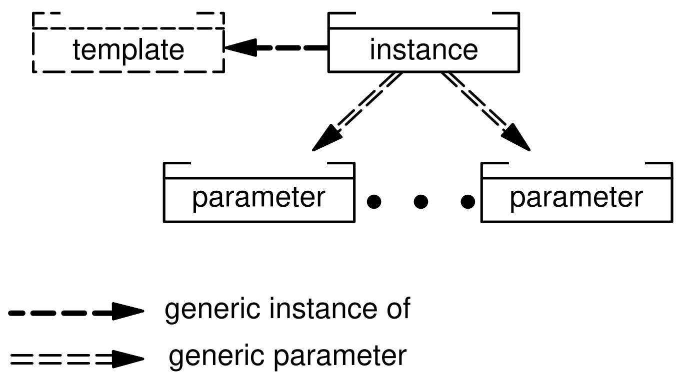


Fig. 17: Generics

The graphical description for the integer stack example is given in fig. 18.

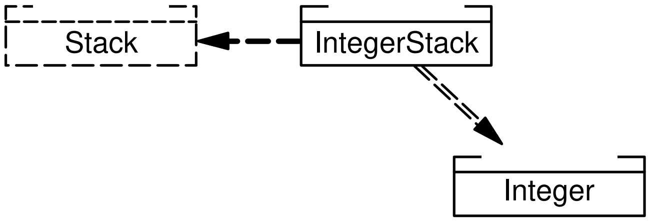


Fig. 18: A stack of integers

The textual interface description for the template is shown below.

```
GENERIC DATA TYPE MODULE INTERFACE Stack(Entry);
...
TYPE
    T <: Public;
    Public = OBJECT METHODS
                    ...
                    push(entry: Entry.T);

                    pop(): Entry.T RAISES {EmptyStack};
                    ...
                END;
END Stack.
```

Finally, here is the description of the generic instance. This instance provides a type IntegerStack.T which has the same properties as shown in the template with Entry replaced by Integer.

```
DATA TYPE MODULE INTERFACE IntegerStack= Stack(Integer)
END IntegerStack.
```

Again, it is quite obvious how to map this design onto Modula–3. Problematic though is that we introduced generics on the level of components, not only on the level of modules. Conceptually, generic subsystems are just as important as generic modules. Consider the situation where some collection is build on top of one or more other collections, e.g. the stack data type module implemented using a data type module for lists. In this case, it would be natural to instantiate a generic subsystem consisting of a generic stack and a generic list module. In our textual description, we allow this by specifying

```
GENERIC SUBSYSTEM INTERFACE StackSystem(Entry);

EXPORT Stack;

END StackSystem.


GENERIC SUBSYSTEM BODY StackSystem(Entry);

CONTAINS Stack, List;

END StackSystem.
```

Note that a generic subsystem may contain non–generic components. From the viewpoint of the instantiation process, they can be thought of as generic components without generic parameters. Note also that the generic parameters of each component contained in a subsystem must be a subset of the generic parameters of the subsystem itself.

We want to conclude this section with a short summary. We have presented generics as a concept which is technically part of the mapping of a design onto a programming language. Generics provide a powerful way of parameterizing system components, thereby creating adaptable and reusable templates. The instantiation of a template is related to the specialization of a data type, but is considered to be a static process. We also discussed how genericity can be used on design level to specify components as instances of generic specifications. Both aspects of generics are quite independent: generic facilities of programming languages can be used for the implementation of arbitrary modules, and components specified as generic instances can be implemented with or without using PiS genericity. For Modula–3, generics can and should be implemented just like modules, i.e. by placing additional keywords in comments. Since Modula–3 does not support generic subsystems, the instantiation of a subsystem has to be mapped onto one instantiation for every component of the subsystem.

5. Summary

In this paper, we proposed an adapted and slightly revised version of the design specification language from [Nagl90]. The basic building blocks of the language are modules. Every module has an interface which describes the services it offers to the rest of the system, and a body which determines how these services are implemented. These bodies are subject to the Programming–in–the–Small working area and are not described by the architecture language. Furthermore, we distinguish different module types in accordance to the kind of abstraction which is encapsulated by the module, namely functional, data object, and data type modules. The language features larger specification units than modules in the form of subsystems. Subsystems may consist of an arbitrary number of modules and other subsystems. In contrast to the approach of stepwise refinement, subsystems provide an abstraction concept: on every hierarchy level of the system, the subsystems represent abstraction units with a body and an interface. The design lays down which components constitute a subsystem, and which components contribute to the interface of the subsystem. Finally, the language supports different relationships to describe dependencies between system components. They can be divided in relationships which define the structure of the system (local containment and specialization) and those which allow one component to use the resources of another component for their realization (local, general, and specialization usability). The specification itself consists of two interrelated parts: a graphical notation which describes the global dependencies between the system components, and a textual specification which gives a detailed description of all component interfaces. An important property of the language is that it is strictly restricted to the (static) Programming–in–the–Large level: it does not consider dynamic aspects of the running system.

Besides presenting the syntax and semantics of the architecture language, we also showed how it can be used to discuss methodological aspects of software design in a more or less abstract way. Although we do not claim to know how arbitrary systems can be designed, we think that our language provides a conceptual framework for the consideration of architecture scenarios. We gave examples of how to model certain standard situations and we demonstrated how our language can be used to discuss architecture patterns.

Finally, we touched some points concerning the transference of a given design onto an implementation, especially when Modula–3 is chosen as implementation language. Some of these points are quite trivial, others suffer from the fact that some features of our language can be mapped directly neither on Modula–3 nor on any other programming language existing at the moment. We also discussed generics as an aid for the mapping of a design onto an implementation as well as an optional extension to the design language, and we described the coarse conceptual outline of a tool currently under development which supports analysis, design, and development of software using our architecture language.

References

[Agresti86] W. Agresti (Ed.): *New Paradigms for Software Development*, IEEE Computer Society Press, 1986.

[Altmann79] W. Altmann: *A New Module Concept for the Design of Reliable Software*, in P. Raulefs (Ed.): *Workshop on Reliable Software*, Hanser–Verlag, 1979.

[ASU86] A.V. Aho, R. Sethi, J.D. Ullman: *Compilers – Principles, Techniques, and Tools*, Addison–Wesley, 1986.

[Boehm82] B.W. Boehm: *Software Engineering Economics*, Prentice Hall, 1982.

[Booch91] G. Booch: *Object–Oriented Design with Applications*, Benjamin Cummings, 1991.

[Börstler93] J. Börstler: *Programmieren–im–Großen: Sprachen, Werkzeuge, Wiederverwendung*, Ph.D. Thesis, RWTH Aachen, 1993.

[BR69] J.N. Buxton, B. Randell (Eds.): *Software Engineering Techniques*, *Report on a conference, Rome, 1969*, NATO Scientific Affairs Division, 1969.

[CABDGHJ94] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, P. Jeremes: *Object–Oriented Development – The Fusion Method*, Prentice Hall, 1994.

[CDGJKN88] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, G. Nelson: *Modula–3 Report*, Research Report 31, DEC SRC, 1988.

[CDGJKN89] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, G. Nelson: *Modula–3 Report (revised)*, Research Report 52, DEC SRC, 1989.

[CY91] P. Coad, E. Yourdon: *Object–Oriented Design*, Yourdon Press, 1991.

[Gall82] R. Gall: *Structured Development of Modular Software Systems – The Module Graph as Central Data Structure*, Proc. WG’81, Workshop on Graphtheoretic Concepts in Computer Science, Hanser–Verlag, 1982.

[GHM90] J.V. Guttag, J.J. Horning, A. Modet: *Report on the Larch Shared Language, Version 2.3*, Technical Report 58, DEC SRC, 1990.

[Harbison92] S. Harbison: *Modula–3*, Prentice Hall, 1992.

[Jackson83] M. Jackson: *System Development*, Prentice Hall, 1983.

[Jones91] K.D. Jones: *LM3: a Larch Interface Language for Modula–3 – A Definition and Introduction*, Technical Report 72, DEC SRC, 1991.

[Knuth68] D.E. Knuth: *The Art of Computer Programming*, Volumes 1–3, Addison–Wesley, 1968/69/73.

[Lampson83] B.W. Lampson: *A Description of the Cedar Language*, Technical Report CSL–83–12, Xerox Palo Alto Research Center, 1983.

[Lewerentz88] C. Lewerentz: *Interaktives Entwerfen großer Programmsysteme – Konzepte und Werkzeuge*, Ph.D. Thesis, RWTH Aachen, Informatik–Fachberichte, Springer–Verlag, 1988.

[LN85] C. Lewerentz, M. Nagl: *Incremental Programming in the Large: Syntax–Aided Specification Editing, Integration and Maintenance*, Proc. 18th Hawaii International Conference on System Sciences, 1985.

[Nagl82] M. Nagl: *Einführung in die Programmiersprache Ada*, Vieweg–Verlag, 1982.

[Nagl90] M. Nagl: *Softwaretechnik: Methodisches Programmieren im Großen*, Springer–Verlag, 1990.

[Nelson91] G. Nelson (Ed.): *Systems Programming with Modula–3*, Prentice Hall, 1991.

[NR68] T. Naur, B. Randell (Eds.): *Software Engineering, Report on a conference, Garmisch, 1968*, NATO Scientific Affairs Division, 1968.

[RBPEL91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen: *Object–Oriented Modeling and Design*, Prentice Hall, 1991.

[RLW85] P. Rovner, R. Levin, J. Wick: *On Extending Modula–2 for Building Large, Integrated Systems*, Research Report 3, DEC SRC, 1985.

[Rovner86] P. Rovner: *Extending Modula–2 to Build Large, Integrated Systems*, IEEE Software 3(6), 1986.

[Wirth82] N. Wirth: *Programming in Modula–2*, Springer–Verlag, 1982.

[Wirth86] N. Wirth: *Compilerbau*, B.G. Teubner Stuttgart, 1986.

[WWW90] R. Wirfs–Brock, B. Wilkerson, L. Wiener: *Designing Object–Oriented Software*, Prentice Hall, 1990

[Zündorf94] A. Zündorf: *Programmierte Graphersetzungssysteme – Implementierung und Anwendung*, to appear as Ph.D. Thesis, RWTH Aachen, 1994.