Verified Software Components

Murali Sitaraman, William F. Ogden, and Bruce W. Weide

Technical Report RSRG-12-01
School of Computing
100 McAdams
Clemson University
Clemson, SC 29634-0974 USA

January 2012

Verified Software Components

Murali Sitaraman
School of Computing
Clemson University
Clemson, SC 29634
E-mail for Correspondence: murali@clemson.edu

William F. Ogden
Bruce W. Weide
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210

Web page: www.cs.clemson.edu/group/resolve

Acknowledgments

We thank the past and present members of the RESOLVE/Reusable Software Research Groups (RSRG) at Clemson and Ohio State. The research behind this work has been funded in part by grants from the US National Science Foundation.

Fundamentals

Fundamentals

1 An Introduction to Software Components

“Everything should be as simple as possible, but no simpler.”

–Albert Einstein

1.1 A Characterization of Components and Component-Based Systems

Software development and maintenance are complex and challenging activities. To manage the complexity, typical systems are composed from a number of components. Some components of a system may be pre-existing. Some others will have to be specially built, possibly in parallel, by members of a team. In this scenario, it is typical for one person to develop a component that is used by someone else. Team software development requires much communication among developers, while simultaneously introducing tremendous scope for miscommunication. Intense and incorrect communication among developers clearly decreases their productivity and the quality of the developed software. This central problem in large-scale software development and maintenance is an essential focus of component-based software design.

Component-based software design, through suitable modularization, minimizes the communication that is needed among modules of a system. It demands that the communication that does occur is both necessary and sufficient, and that it is precisely documented. These properties of good design are essential to facilitate independent production and quality assurance of modules early, and to avoid expensive surprises at module integration time later in the software development cycle.

1.1.1 Fundamental Role of Specifications

Good design works on the hypothesis that it is possible to modularize a system so that users of a component can work with a lot less information about the component than its developers. The fundamental issues here are how to distinguish and isolate the information that is needed to use a module from details of its implementation and how to present this information to users. Some essential and desirable properties of user-centric or client-centric information about a module, commonly termed the specification of the module, are clear. The specification should be such that it is all the clients need to know in order to use the module, because otherwise clients will need to communicate with developers to get information that is not given to them. The necessary and sufficient information contained in the specification should be presented in terms that are understandable to the clients who are without the knowledge of implementation details of the

component. The specification must be precisely described in a formal language to avoid miscommunication. For clients of a component to work independently and in parallel with the developers of a module, the specification must be defined even before the component has been implemented. This forces the specification to be such that it can be implemented at least in one way, but yet general with respect to how exactly it might be realized so that developers can consider alternative techniques for its implementation and corresponding efficiency trade-offs. Ultimately, the specification must be simple. To the extent a specification fails to meet one or more of these considerations, its utility is considerably decreased.

Developing a good specification for a component is a difficult and challenging activity. It adds considerable additional expense to software development and significantly affects software production schedule. Few computing curricula educate students in techniques for good module specification and fewer practitioners know how to develop specifications. In addition, specification of modules that need to exhibit concurrent behavior, for example, continues to be a research problem. In spite of these difficulties, there is no substitute for good specification in software design, because only specifications can enable effective and error-free communication among module developers in large-scale systems. Only specifications can facilitate correct and independent module development. For components with behaviors that we do know how to specify cleanly, specifications can improve both productivity and quality significantly.

The cost for developing and using specification is easier to justify for components that are used more frequently than others. In typical top-down design that leads to one-of-a kind modules routinely, there is less justification for the idea. In bottom-up and object-based design where there is more emphasis on (repeated) reuse of existing components, the economic justification for specifying reused modules is more compelling. Specification of reusable modules has practical value, however, only if reusability is achieved without compromising efficient performance. The domain of data structures and algorithms where performance is a central theme poses a particular challenge, because it is essential to show both that these techniques can be captured in efficient reusable components and that the components can be specified properly. Using specifications and implementations of suitable components for this domain, this volume illustrates the principles of how to design and develop good software components. In the process, we hope to motivate more general application

of the principles to other domains including language translation, databases, and operating systems, and to motivate construction of large systems reusing such components.

The rest of this chapter establishes context and terminology for engineers, researchers, and practitioners familiar with other approaches to component-based software engineering.

1.1.2 Specification Principles: Information Hiding and Abstraction

There is no universally accepted definition of a software component. But specification assumes a central role in any useful definition. To understand further why this is the case, we begin with a simple observation about any engineering component (not necessarily a software component):

- Some details are internal to the component and are important to an insider, but not an outsider.
- Some details are external to the component and are important to an outsider, but not to an insider.

Information Hiding

We need a way of separating what's inside a component from what's outside it. This separation is achieved through a component specification the objective of which is to capture what is important to both insiders and outsiders. The primary reason for considering a component is to enhance the understandability of a situation to the people who play the dual roles of clients and implementers. There is no reason to do this unless there is to be some *information hiding* about what is inside the component. To illustrate this principle, we use the familiar stack behavior as an example[1].

[1] In [Weide 00], Bruce Weide explains that principles discussed in this section can be understood using any engineering component or system, not necessarily a software component. We reproduce examples from [Weide 00] in footnotes, such as here: It is helpful to think of a TV as a system so makers of other audio/video systems as well as end users in their living rooms can ignore details like picture tubes, tuners, and other internal technical aspects, and treat the TV as a single component with a few external buttons, switches, and connectors that control a picture that appears on a screen and some sound that goes with it. The internal information is hidden.

Stack behavior is simple, yet to capture it properly a number of fundamental design questions, similar to those in developing components to capture more complex behaviors, need to be addressed. Since our focus is on developing principles that apply to more complex situations, it is especially important not to be lured into simple-minded design solutions that are adequate for stacks or other simpler structures, but that do not scale up.

A stack is a linear structure that permits entries to be removed in a last in first out (LIFO) order, and it is useful in constructing language processors such as compilers and editors. The LIFO access structure of a stack distinguishes it from random access array structures and other linear structures that do not impose any restrictions on the access order. Clearly, instances of these other structures can be used in situations that demand stack behavior, if all users follow suitable conventions. But use of less restricting structures, in cases where stack behavior is demanded, introduces the possibility that operations that are not consistent with stack behavior might be performed, either intentionally or accidentally. This problem is acute in large software development efforts, because different people are involved in developing and maintaining a module during its lifetime.

To capture stack behavior, we need a data structure for storing the contents of a stack, and procedure bodies for at least two operations to manipulate the structure: The Push operation permits addition of an entry to a stack and the Pop operation removes the most recently inserted entry from the stack. (These operations get their names from the commonly used push down metaphor of a stack.) The data structure and the code for procedures are intricately coupled. Changes to the data structure will lead to changes in the code. In addition, the code for the procedures should be mutually compatible. For, example, the code for Push and Pop need to adhere to a common convention of where the most recent entry in the stack will be placed in the data structure. Decoupling of interdependent data structures and procedures into stand-alone syntactic units is poor software engineering.

What we need is a *specification* of a stack component that *hides information* about the implementation details from clients, and provides a simpler, conceptual view that can be used only for LIFO access. Design and development of such components that allow only a prescribed behavior can preclude inadvertent user violation of the behavior and eliminate a common source of errors in team software development.

Unlike earlier programming languages where procedures serve as the sole modularization construct, object-based languages include mechanisms to group data structures and procedures together and hide irrelevant information through interfaces. The languages support the principle of information hiding by syntactically separating *interfaces* (syntactic aspects of specification) from implementations of components. An interface typically includes a listing of (types of) objects, operations and their parameters, and a client is syntactically restricted to using only items from this listing. An implementation of the interface provides code. It contains details of data structures and procedure bodies for the operations. This general idea, also termed encapsulation, is supported through syntactic structures such as classes in C++ and Java, modules in Modula-2, and packages in Ada.

In spite of the importance of the idea of encapsulation for software engineering and availability of programming language support, the idea has found only a limited application in computing. Components that group data structures and procedures to capture simple stack and queue behaviors are common. But the dichotomy of data structures and algorithms is pervasive, and often the structures and algorithms are separated into different syntactic units though the separation is rarely justified. For a particular algorithm to work efficiently, in general, it needs to be closely coupled to particular data structures. Details of the coupling should be hidden. A software engineering treatment of data structures & algorithms needs to employ the information hiding principle for proper modularization of behaviors of both data structures and algorithms.

Abstraction and Modeling

To effectively use a component and for independent development of components of a large system, we need a specification that communicates to potential users necessary and sufficient information about the component in a way that is understandable to clients. Abstraction is the technique that permits going beyond merely hiding internal details of a system that are considered inessential from the outside, to "reconceptualizing" the

component by explaining its externally visible behavior in terms that are fundamentally different from those that would explain its internal details[2].

For the stack component, it is clear that a user should know at least the parameters to the component (if any) and parameters to Push and Pop operations. But this information is not sufficient. To call an operation, in addition to its parameter list, clients ought to know the effect of calling of the operation. Clearly, however well conceived, choice of suitable names for operations and their formal parameters alone is insufficient to communicate its effect to users. Comments are typically used to describe the behavior of a component in its interface, because current programming languages do not include formal, syntactic slots for describing all necessary information.

The fundamental difficulty is in presenting the effects of using a component in its interface precisely, yet in terms understandable to clients. This task requires establishing suitable models of the principals, both for those that are within the scope of the component and those in the environment with which the component interacts. To explain the effect of an operation on a stack,

---

[2] Returning to the TV example, if none of the complex internal details of the TV are superfluous (a reasonable assumption, since extraneous items probably would add to the TV's cost without any associated benefit) then hiding information about any of them is certain to limit the ability to describe the *behavior* of the TV. Hiding anything about the inside of the TV requires making up some kind of a behavioral cover story to explain the TV's externally visible behavior. This is the motivation abstraction, the other side of the information-hiding coin.

Consider the instruction manual for a TV. It is effectively the TV's interface description for clients, and includes both structural and behavioral aspects. The manual describes what externally accessible buttons, connectors, dials, pictures, and other items are available to connect the TV with other audio/video systems, and their effects in terms of how they operate the TV. Here is a sample statement from an actual TV instruction booklet:

This TV automatically memorizes the channel being received.

Most people know better! But the implicit anthropomorphic model of the TV helps a client understand how the TV behaves. It does not help anyone understand how that behavior is implemented, but after all, hiding that information is its purpose. Few people could understand the TV's behavior if this part of the manual discussed the circuit that does the "memorizing." And the manufacturer surely wants the flexibility to change that circuit for the next model year without having to go back and rewrite this part of the instructions. So, this is actually an excellent behavioral cover story.

for example, it is essential to have a model for stack. In general, such modeling is fraught with potential problems. While the models should be adequate for the purpose, pragmatic constraints may suggest settling for approximately correct models instead of searching for ones that are most suitable. Neither the defects of the chosen models nor their inadequacy may be obvious, until after software is developed and deployed. Given the inherent difficulty of modeling in software development, and in general, in science and engineering, our objective is reduced to ensuring that our framework has a specification language that is rich enough to express chosen models.

Formal vs. Informal Specifications

Use of informal models and analogies (e.g., "k-ary tree is like a family tree"), physical metaphors, or pictures are perhaps useful to introduce an idea, but they fall way short of the demands of good module description. In addition to being naturally imprecise, such an ad hoc approach clearly does not scale up to descriptions of complex modules in realistic software development. To present interface descriptions precisely, formal notations are needed. A temptation is to promote the data structures from module implementation to the interface as implementation-oriented models, and then to describe behavioral effects of a module in terms of the data structures. (Some programming languages through syntactic structures facilitate inclusion of internal module information in a private section of the module interface, inadvertently encouraging modules to be described to users in terms of those internal details.) This tried and failed approach thwarts almost all central benefits of module specification and foils developmental independence, because module users are burdened with understanding and depending upon internal module details that are complicated and are likely to change.

A component specification should be simultaneously formal and in terms understandable to users. The language of Mathematics is best suited for modeling and descriptions of modules, because of its precision, expressiveness, and common knowledge of the notations and terms. While use of mathematical models and notation will not alleviate the difficult problem of whether the chosen models capture real world situations adequately and correctly for a software module, they provide a precise common basis for communication among software developers.

Ultimately a specification is useful in practice only if there are implementations to provide the specified behavior. When a specification is expressed formally, it becomes possible to test rigorously that an implementation indeed produces the specified behavior. More importantly, mathematical specifications can facilitate mechanical verification of implementation correctness. A formal framework that allows modules to be specified must also include a sound and relatively complete reasoning system to guarantee implementation correctness with respect to its specification. The ability to reason with ease, formally or informally, is one of the key dimensions in specification and implementation language design.

1.1.3 Motivation for Parameterized Components

The need for a given behavior arises in widely different contexts. For example, in different applications of "stack" behavior, the information that needs to be stored in the stack will be different. A text editor that allows a sequence of "undo last command" actions might use a stack to keep information, such as which particular command was invoked and on what part of the text the command was applied. A top-down design approach might suggest developing a specialized stack module for each application such as the text editor. Such a solution is expensive because it leads to numerous similar, but not identical modules. Even where it is possible to borrow a stack module developed for one purpose and edit it slightly for another, there is a significant quality assurance cost. This is because even minor changes to a software module can drastically affect its behavior and re-certification is essential to assure that the changes do not produce any unintended side-effects.

Ideally, it should be possible have a single stack module that allows module users to supply the kinds of entries to be stored in the stack depending on the application of the module. To facilitate this generality, a stack module should have a parameter that is the type of entries to be pushed onto and popped from a stack. To use such a parameterized stack module, it must be first instantiated with a suitable parameter for the type of entries. Using a parameterized module in different contexts requires different instantiations, but no direct change is needed to the module.

Some modern languages includes features to build parameterized data encapsulation modules. Class templates in C++ and generic packages in Ada are among such features useful for the purpose. In addition to savings

in production time, development and use of parameterized modules can lead to savings in compilation costs because it is possible to compile generic modules and produce reusable object code. Similar cost savings are possible in verification because it is possible to verify parameterized modules independently of the context in which they are used.

1.1.4 Implementation Substitutability and Modular Reasoning

Some components and their specifications in a software system are likely to be tailored to the special needs of the application, and are unlikely to be of general use. They may be biased by the specific contexts and particular implementation choices. Some other components, such as those that capture data structures and algorithms have more general applicability. In these cases, to produce a specified behavior, it is usually possible to use different combinations of data structures and algorithms. Stated in other words, different component implementations may provide the same visible functional behavior, though the implementations may differ in their utilization of resources and performance. This observation leads to the idea of specifying a reusable concept, as opposed to specifying the behavior of a particular module.

A concept can be realized by alternative implementations using different data structures and algorithms. It is reusable across applications, because it makes few assumptions on the context in which it might be employed. For a general-purpose concept, there is usually no one best implementation module. Different implementations offer resource trade-offs such as between time and space, and other trade-offs such as between efficiency and predictability. Alternative applications of a concept may choose one implementation module over another based on their performance considerations. Since a single concept is useful to capture a variety of implementations, few concepts are sufficient to capture a vast domain such as data structures and algorithms. Different algorithmic techniques then simply serve to illustrate alternative implementations of the concepts and corresponding trade-offs.

When a concept specification is realized in a variety of ways, a client may *substitute* one implementation for another for performance or other reasons,

while leaving the functionality of the system unaffected[3]. The idea of implementation substitutability leads us to *modular reasoning* or specification-based reasoning: It is to reason about a component's behavior without knowing anything besides its specification, and the specifications of components reused in implementing it. This means that a client should be able to reason about a component without any knowledge of its implementation details. An implementer of a component cannot assume any knowledge of the client environment either, other than those documented explicitly in the component specification.

Achieving the modular reasoning property is a central issue component-based design in engineering. It makes it possible to localize understanding, reasoning, and maintenance of systems. It allows different vendors to deliver the same functionality with different auxiliary characteristics (e.g., price). However, we can count on these economic benefits *only* when it is legitimate to replace one implementation with another, without worrying that the overall system might break as a result of the substitution. An excellent discussion of economic issues in the context of modular computer system design, as well as an insightful commentary on the theoretical importance of "modularity" can be found in (Baldwin and Clark, 2000).

Design-Time Relationships

The idea of implementation substitutability also raises the need to distinguish the current version of a system from other functionally-identical variations that might be obtained from it by compatible substitutions of implementations[4]. Since implementation substitutability is the hallmark of a

[3] This *substitutability* property is what offers, for example, the option of buying any of a large number of TV models from any of a number of TV vendors, plugging your TV into the wall and connecting it to the same remote speakers and other audio/video systems, and expecting the whole combination to work as advertised. Systems such as modern audio/video systems that satisfy the substitutability property are sometimes called *component-based systems* because various vendors can supply subsystems to meet *standard* interface specifications. These systems generally do differ in internal details and, of course, in other ways (e.g., price) that are inessential in meeting the standard interface specifications.

[4] The impact of this observation becomes apparent when considering diagrams that often are used to convey information about the nesting structure of physical systems. In a diagram of a specific physical system, such as a particular TV, the original notion of a system makes sense: a piece of the physical world with an exterior, an interface, and an interior. But a system diagram of this TV is misleading as a claim about the *design* of the

component-based system design, diagrammatic techniques for representation of component-based systems must be such that they are suitable to represent a whole class of systems, instead of a single current system. In particular, nested-box diagrams are inadequate for depicting system designs that are explicitly component-based. If these diagrams are used, a combinatorially explosive number of diagrams would be needed to depict a single component-based design.

Software engineering notations such as UML –the current *de facto* industry standard Unified Modeling Language – (Fowler and Scott, 1999) are useful to overcome the problem of nested diagrams. UML class diagrams can be used to capture important relationships that might hold between the various artifacts describing a component-based system — not between the systems they describe, but between the artifacts themselves. An example of such a diagram is shown in Figure 1. This diagram shows both the relationships among component specifications and implementations and the potentially explosive number of specific component-based systems that could be built using these components. There are other interesting relationships that could be shown here, but our goal is merely to explain the idea of design-time relationships using an example. In this diagram, rectangles stand for specifications and implementations. Arrows connecting rectangles denote *coupling* or *dependence* relationships. The diagrams shows that both *R1* and *R2* implement *S* and that the implementation *R1* uses components with specifications *S1* and *S2* to realize the behavior specified in *S*. The diagram documents substitutability, e.g., that the components whose internal implementations details are explained in *R1* and *R2* are substitutable for each other in any client that needs the behavior specified in *S*.

TV, because it ignores the substitutability principle. Thinking of the interior of a particular system as a fixed thing, and not simply as one of the many possible interiors that might be compatible with the abstract description, simply overlooks the key reason for component-based system design.

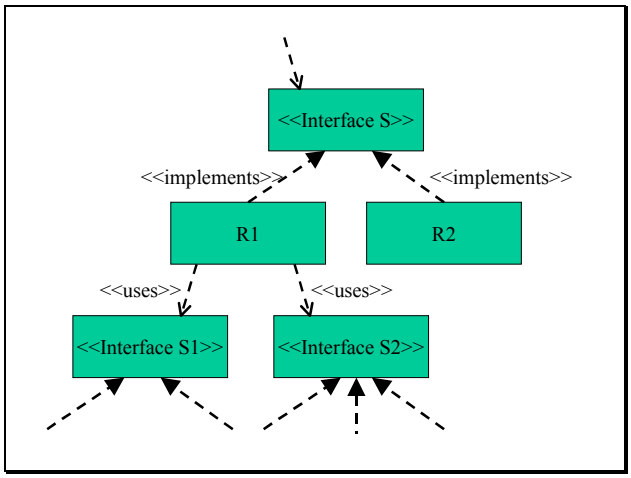

Figure 1 — A UML class diagram showing important design-time relationships

Specifications, Implementations, and Coupling

In general, when a design diagram shows that one artifact is coupled with one or more other artifacts, then it means that to understand or modify the first artifact, an understanding of the other artifacts is necessary. While some dependencies are unavoidable in a component-based system, we want to minimize dependencies. It is clear that *long dependency chains* are *undesirable*, because it implies understanding of a lot of artifacts to understand one – if the dependencies are transitive.

We observe that that there are no transitive dependencies in Figure 1. Even better, we see that no dependency chain is longer than 1. This observation remains in tact even if we extend the figure to include several other specifications and their implementations, as long as we include only similar implements-uses relationships. This is because specifications serve as firewalls between implementations in the figure. To understand or modify an implementation, only the specifications to which it refers to directly need to be understood. No implementations will ever need to be stood.

It is possible that a specification will use other specifications in order to specify new concepts based on ones documented earlier. This is an unavoidable transitive dependency. A careful choice needs to made between the extremes of attempting to make a specification entirely self contained and attempting to separate every factorable aspect of it.

While specification-based dependencies are unavoidable, implementation to implementation dependencies are undesirable and often avoidable. If an implementation uses another implementation directly, which in turn uses yet another directly, and so on, then to understand the first implementation the entire chain of implementations need to be understood. This breaks the modular reasoning property, and leads to software that is difficult to understand or maintain. For these reasons, we will avoid implementation-to-implementation dependencies entirely. There will be no long implementation-dependency chains in our designs.

In summary, a framework for component-based software design must allow concept design, development of multiple implementations, and implementation substitutability as well as modular behavioral and performance reasoning. These issues affect both the design of languages and software designed using those languages.

2 Principles of Design and Specification

2.1 Basics of Interface Design

Software components intended for reuse should be designed and developed deliberately with great attention to a host of details. The first section of this chapter provides a characterization of good interface designs for components. In motivating good designs, we appeal to universal engineering principles such as generality, simplicity, efficiency, and utility. Good designs involve consideration of both programmatic and mathematical aspects. The second and third sections of this chapter address those issues.

2.1.1 Definition and Use of Parameterized Components

The principle of generality suggests that reusable components for capturing behaviors of container structures, such as stacks and queues, should be parameterized by the type of entries to be inserted. All modern languages including Ada, C++, and the 2003 version of Java allow definition and use of generic components and allow them to be instantiated with an arbitrary type of entries. A mostly incomplete skeleton of the interface of a parameterized stack concept, named Stack_Template, is shown in Figure 2 in RESOLVE notation.

```
Concept Stack_Template (type Entry; ...);

    Family Stack …

    Operation Push (E: Entry; S: Stack);
        ...
    Operation Pop (R: Entry; S: Stack);
        ...
    Operation Depth (S: Stack): Integer;
        ...
    Operation Clear (S: Stack);
        ...
end Stack_Template;
```

Figure 2: A Skeleton of Parameterized Stack Concept

To emphasize that the component needs to be implemented and instantiated before objects of type Stack can be declared and used, we have used the keyword **Family** in

declaring Stack. Before we can use objects of type Stack in a client program, we need to construct a facility. A **facility** is a template instantiation statement, and it supplies the appropriate parameters to the concept as well as picks a particular implementation for that concept. Figure 3 depicts the immediate relationships. In this figure, Stack_Template is shown to have two implementations (or realizations); one that is time efficient and another that is space efficient.

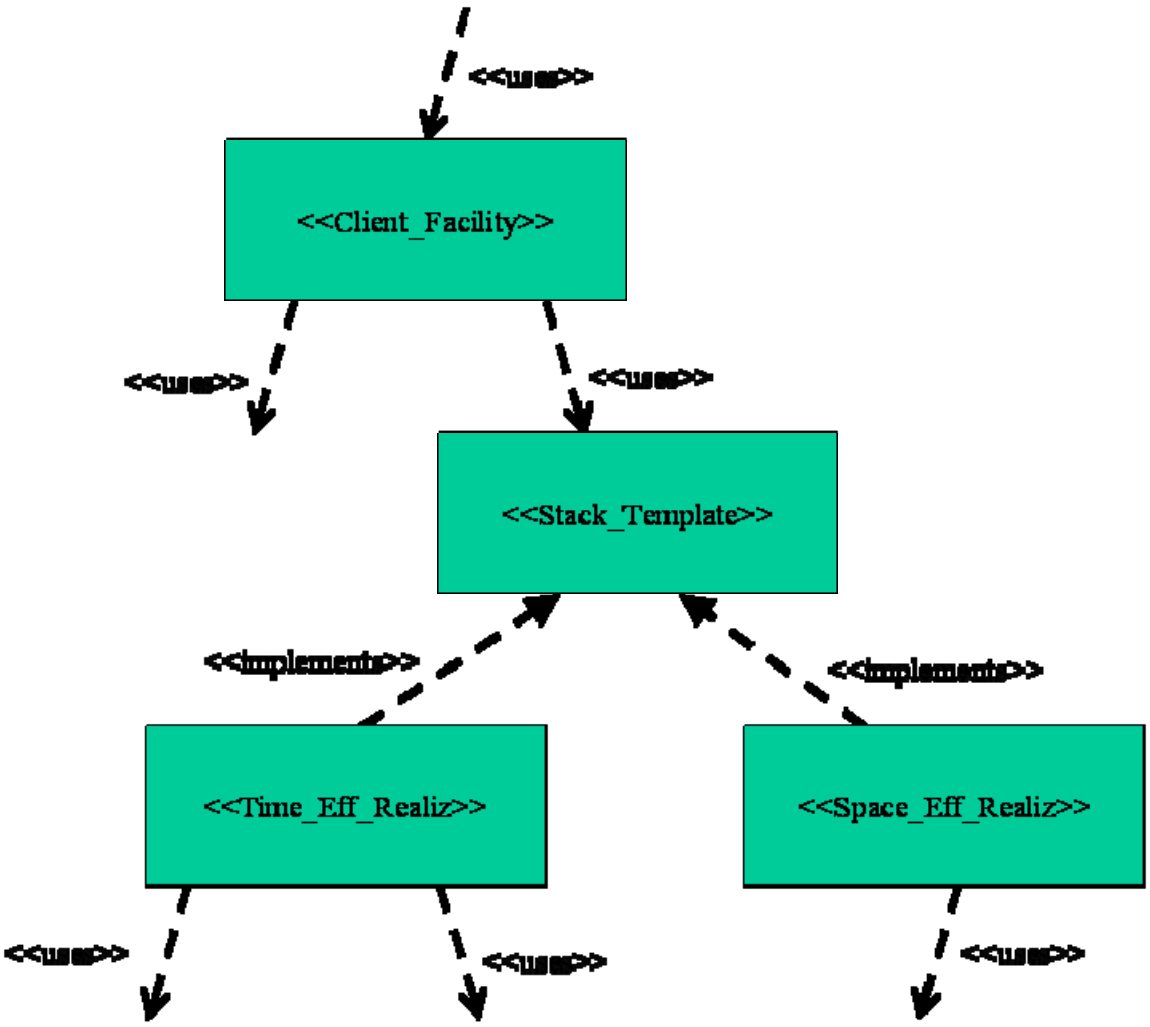

Figure 3: A Pictorial Depiction of Relationships

Using *Time_Eff_Realiz*, one implementation of Stack_Template given in Chapter 3, Figure 3 shows an instantiation statement. In general, the implementation choice is dictated by performance considerations.

```
Facility Client_Facility;
    uses Stack_Template;

        Facility Integer_Stack_Fac is Stack_Template (Integer, …)
            realized by Time_Eff_Realiz;

        Var S, T: Stack;
        Var I: Integer;
        …
        Push(I, T);
        …
end Client_Facility;
```

Figure 4: Declaration and Use of a Stack Facility

Once Stack_Template is instantiated, we can declare variables *S* and *T* to be stacks. Then we can apply any of the operations such as Push provided by the facility or the universal operation swap to *S* and *T*. These stacks then become first class objects in our programs just the same as integer or character type variables are.

Note that when declaring objects or variables *S* and *T* it is not necessary to indicate that they are to be of the Stack type provided by the facility *Integer_Stack_Fac*, because there is only one Stack facility in the current scope. If there are several stack facilities in the current scope, then it is necessary to qualify the Stack type declaration with the name of the appropriate facility and write *Integer_Stack_Fac.Stack*. However, it is not necessary to write *Integer_Stack_Fac.Push(I, T)*, since a compiler can easily tell from the fact that *T* is associated with *Integer_Stack_Fac* that the appropriate Push operation comes from there too.

The example also shows that we can construct custom one-of-a-kind facilities, such as *Client_Facility*, that is not parameterized and that provides one specific implementation. However, in well- engineered software, most facilities should be introduced by instantiating previously developed concepts and realizations.

2.1.2 Types and Objects

Several details of the skeleton interface in Figure 2 demand explanations. The first of these is the observation that the interface of the component (Stack_Template) and the type it provides (Stack) have been distinguished. This separation gives the flexibility of defining multiple related types in a single interface, without introducing more complicated machinery such as nested structures or factory patterns. This flexibility is essential occasionally to specify closely related types (e.g., points and lines) and operations that involve arguments of both types. Providing generality along this dimension – the ability to define multiple types within a single interface, on the other hand, appears to have little impact on the ability to compile or verify implementations of concepts efficiently. While the debate whether classes are the same as types will continue in object-oriented languages, it is clear that components are not and should not be types.

We have made a conscious decision not to specify a single stack object, though in general, a concept may define a single object that provides a specified behavior or define a type that can be used to declare variables of that type. In typical object-oriented programming, a "self" object is defined which acts as an implicit parameter to all operations. Defined in this way, Push operation, for example, will have only one parameter. An example of such an interface definition is given in Figure 5.

```
Concept Object_Oriented_Stack_Template (type Entry; ...);

    Var Self: Stack …

    Operation Push (E: Entry);
        ...
end Object_Oriented_Stack_Template;
```

Figure 5: An Object-Oriented Stack Concept

In special situations where exactly one stack variable is needed, this alternative concept avoids the need for declaring stack variables or passing them as parameters to Push and Pop operations as illustrated in Figure 6[5].

[5] It is possible to declare and use multiple facilities of Object_Oriented_Stack_Template. Where multiple such stack facilities are needed, however, it is necessary to qualify each call with the name of the object, leaving no

```
Facility OO_Client_Facility;
    uses Object_Oriented_Stack_Template;

        Facility My_Stack is Object_Oriented_Stack_Template (Integer, …)
            realized by Time_Eff_Realiz;

        Var I: Integer;
        …
        My_Stack.Push(I);
        …
end OO_Client_Facility;
```

Figure 6: Declaration and Use of a Stack Object

The difficulty with mechanisms that allow only definition of objects, but not types, arises where it is essential to specify operations involving more than one object of the provided type. Handling of data movement operations involving multiple objects of a type (e.g., replication of an object) and operations such as Integer addition becomes awkward, unless two separate kinds of objects are considered.

While we do not want to preclude concepts that provide special-purpose objects, we should not make it unnatural to develop concepts that provide operations involving multiple objects of a type. The approach of defining user-defined types instead of objects is also consistent with traditional programming language practice of defining built-in types, such as Integers. For these reasons, in general, our concepts define types, unless there is a compelling reason otherwise. We will discuss examples of concepts (e.g., pointers) later in this book that turn out to be more natural when a combination of objects and types, are defined and used.

2.1.3 Operations

Simplicity is an essential virtue of a good concept. For a concept to be easy to understand and to facilitate development of multiple implementations of the concept, the principle of simplicity suggests that it should have a minimal, yet sufficiently complete set of *primary* operations. This goal of minimality demands that a system

---

advantages for using object-oriented stacks. The only difference is that the stack object appears precedes the operation instead of following it as an argument.

for concept design must allow concepts to be enhanced with additional operations. We defer discussion of language features for concept enhancements to a later section. The specification of a concept for Stack_Template limits legal manipulations of stack variables by providing a particular set of primary operations. It contains operations (and parameters) that are named to be consistent with the metaphor being used for the programming concept under description. The concept provides Push[6] and Pop[7] operations to allow users to insert and remove entries, respectively, from a stack. It also includes an operation to get the depth of a stack. The most common use of the Depth operation is to check if a stack is empty[8]. The ability to detect an empty stack is needed routinely whenever a stack user needs to perform some computation on every entry of the stack using a loop with terminating condition that the stack is empty. The operation is also useful to help avoid the abnormal case of popping a stack when it is empty. The operations Push, Pop, and Depth together form a primitive basis for stacks. They are orthogonal and no one of them can be realized using a combination of others.

---

[6] We use action names such as Push and Pop to operations that affect objects, so that they read like commands. To operations that return some information about an object, we use nouns such as Depth as names. When an operation returns a boolean value, we give it a name that reads like a yes/no question. Is_Empty, for example, is a suitable name for an operation that tests whether a stack is empty. We list parameters to operations in an order that makes it read well with the name of the operation. For example, "Push E *on* S" reads better than "Push *on* S, E". While keeping an eye toward readability, we also follow the convention of making the object of the type defined by the concept as the last parameter for consistency.

[7] In languages such as Java, operations such as Pop need to be defined to return the popped entry. This is because in Java parameter passing is by reference copying and the only way a new object can be returned is through a function return or an extra level of indirection. Defining Pop as a side-effecting function is a common, but undesirable programming practice. The next chapter explains how new objects can be returned through parameters efficiently.

[8] A minor issue of uniformity arises in choosing between Depth and Is_Empty operations for stacks and other container objects. Depth operation clearly provides more information and it is possible to realize Is_Empty readily using Depth operation. But whereas the utility of Is_Empty is obvious in typical loops that need to examine all stack entries, the need for getting the current depth of a stack is less obvious. Depth operation makes it slightly awkward to accomplish the common user task of checking for an empty stack. As minor as this consideration may be, it is clear that a collection of concepts within a library must be consistent with respect to how this issue is resolved to avoid potential confusion and incompatibility.

In Stack_Template interface, we have also included an operation to clear a stack. It is motivated by observing that in the object-oriented paradigm for software engineering, it is routine practice to clear and reuse a given object. It is indeed possible to clear a stack S, for example, by popping all its elements. But this approach is expensive. Other indirect approaches for clearing a stack, discussed later, also turn out to be expensive, in general.

The potential and frequent use of clearing demands a more efficient implementation of the operation, where possible. When Clear operation is included in the interface, an implementation of the data abstraction can produce the effect of clearing much more efficiently by merely resetting an internal field in the representation. A more complete justification for inclusion of Clear operation is somewhat involved and may not be fully apparent until aspects of realizations and concept enhancements are discussed in later chapters. It suffices to say, however, that interface design involves pragmatic and performance considerations and certainly is not limited to providing just adequate functional behavior.

Before a Stack variable can be manipulated, an implementation of Stack_Template needs to create a suitable representation to hold the contents of a stack and initialize it appropriately. For the purpose, the stack concept needs to include an initialization operation in the specification. This operation needs a special status because for correct working of most implementations, the initialization operation must be invoked on a stack variable before any of the other operations can be invoked. In RESOLVE language, initialization is a required implicit operation on every type provided by a concept. This operation is to be invoked automatically by a compiler when a variable of the type is declared. Similarly, a finalization operation is implicitly defined for every type and it is to be called invoked as the last operation on a variable before exiting the scope of the variable.

Externally visible effects of initialization (and finalization) must be stated explicitly in concepts because users and implementations must have a common and clear understanding of their effects. In some implementations of some concepts, initialization and finalization may be trivial, and the corresponding procedures may need no code. Some others may have no initialization code, because the initialization work is handled in an amortized fashion when other operations are invoked. However, the uniformity in the strategy of always defining initialization/finalization operations and automatically invoking them eliminates a routine source of software errors, without complicating compilation. These reasons

have also motivated modern programming languages such as C++ and Java to facilitate automatic initialization and finalization.

2.1.4 Storage Space Management

Storage space management is another efficiency issue that affects the design of data structure concepts. Specification of a (locally) bounded version of a stack concept, for example, may limit the maximum growth of a stack to be within a client-supplied bound. Bounded versions encapsulate implementations that allocate local pools of memory a priori through static or dynamic storage allocation. Alternatively, it is possible to specify a version of a stack concept that is locally unbounded, where growth of a stack is limited only by global memory capacity bounds. Implementations of such a concept will depend on the global memory pool from which they allocate/deallocate storage from that pool dynamically. It is possible to conceptualize a generalized Stack_Template (to be discussed in a future section) that is more flexible with respect to how storage is managed.

We choose to study a more restricted and straightforward version of the stack concept as a first example, because it is simple, yet serves to introduce and highlight other important issues. A skeleton of the concept is shown in Figure 7. The concept, in addition to the type of entries on the stack, also has a parameter to allow users to choose suitable maximum limits for the depths of stacks. We start with this (locally) bounded version of a stack concept, also because it allows us to simplify the implementation discussion in the next chapter.

```
Concept Stack_Template (type Entry; Max_Depth: Integer);

    Type Family Stack …

    Operation Push (E: Entry; S: Stack);
        ...
    Operation Pop (R: Entry; S: Stack);
        ...
    Operation Depth (S: Stack): Integer;
        ...
    Operation Rem_Capacity (S: Stack ): Integer;
        ...
    Operation Clear (S: Stack );
        ...
end Stack_Template;
```

Figure 7: Interface of a Locally Bounded Stack Concept

The interface in Figure 7 provides an operation, named Rem_Capacity[9], to allow users the ability to detect how much capacity is remaining on the stack. The operation is useful, for example, to avoid pushing an additional entry on a full stack. In general, a well-designed concept must provide users the ability to detect situations, such as empty and full stacks, which fail to satisfy the requirements of one or more of the operations. Only when users can detect those situations, the concept can meaningfully relegate the responsibility of avoiding such situations to users.

[9] Given the ability to check the depth of a stack, the purpose of Rem_Capacity operation is less apparent, because it is easy enough to compare the current stack depth with its maximum depth. But in modular software construction, the maximum depth of a stack may be set by one module, whereas a different module that does not have direct access to the maximum depth may have to check the full stack condition. Rem_Capacity also provides more information than the more typical Is_Full operation.

2.2 Efficient Computing Using Invisible References

"their introduction into high-level languages has been a step backward from which we may never recover."

Tony Hoare,

commenting on references in 1973

It is standard programming practice now to use an interface design such as Stack_Template to suppress non-trivial representation details of objects such as Stacks. Languages include mechanisms (e.g., private data) so that users of stack objects cannot access details of the representation. This is a direct result of following the principle of information hiding. However, one other information hiding question remains to be answered: Should or can the interface also hide whether a stack is a reference or a value variable? This is a basic question that arises in defining and using every object. In motivating an answer to this question, we first focus our attention on how programming languages define "value" objects, such as Integers. Figure 8 contains an example piece of code involving two Integer objects I and J, based on typical language-supplied instructions such as addition and multiplication to manipulate Integers:

```
I := I + J;
J := I – J;
I := I - J;
```

Figure 8: An Example Piece of Code Using Integer Objects.

One way to view Integer operations such as + and – is that they are macros, i.e., shorthand notations for describing relatively lengthy binary representations and process. In other words, they help reuse assembly code, and (hence, save time for the typist). Most of us know this is not true — notations in programming languages serve purposes other than clever macro-expansions. By introducing Integer objects and operators such as + and -, programming languages are forcing us to *conceptualize* them in *mathematical terms* instead of as bit manipulations. Implicitly, we treat integer program objects as though they have values from the set of mathematical integers, and associate operators such as + and - with the + and - operators on integer values in mathematics. Unlike the computer, our explanation of the statement I := I + J makes no reference to how integers are represented on a

specific computer or a specific micro program: addition of integers results in a value consistent with adding the integer values of these two objects in mathematics[10]. The mathematical explanations of Integer objects and operations have a profound influence on our understanding and reasoning about programs that use integers. This impact becomes clear in an attempt to understand the values of Integers I and J in the code in 0. This understanding is difficult, if not impossible, without an abstract view of Integer objects and operators. When the explanations of the symbols are based on mathematical terms, in addition to being simple and understandable to an audience unfamiliar with bit sequences and micro programming, it becomes possible to understand and reason about the code. The above code exchanges the values of I and J, under the assumption that no overflows/under flows result from additions and subtractions. There are two key observations concerning this analysis:

Integers are value variables. Every Integer variable has an independent value. Because Integer representations are inherently small, no references and hence, no reference-value distinction is needed. Integer assignment and parameter passing do not cause aliasing. Change to an Integer variable I affects only I. It does not change the value of some other Integer variable J as a side effect, even if I was assigned to J previously.

Integers and Integer operations have "well understood" mathematical counterparts. "I + J" means the same thing in programming as in mathematics, except that the former is subject to computational bounds.

To understand the importance of the first observation, suppose that Integers I and J are represented indirectly using references. In this case, there is potential for I and J to be aliases. Accounting for this *potential* aliasing (even if no aliasing is present in a particular use), introduces a significant source of complexity in the reasoning. The rest of this chapter discusses this issue further and explores the impact of the first observation in computing with non-trivial objects. The next major section is devoted to a detailed examination of the second observation and its impact.

---

[10] This abstract explanation is definitely not precise or accurate, and it is rarely written down. A more accurate description, in fact, should highlight the subtle distinctions between programming objects and operators and their mathematical counterparts. For one thing, there is never an overflow when two integers are added in mathematics. However, integers that can be represented in every practical computer is bounded. Addition of two integers may lead to overflows and hence, answers that are not consistent with mathematical integer addition may result.

If only value variables are allowed in a language, as in pure functional programming, then values of all variables remain decoupled and reasoning becomes quite simple. It is indeed possible to represent and view all objects as values, not just integers. This will simplify reasoning, but potentially large representations of objects would have to be copied when assigning one object to another and in parameter passing. Unfortunately, efficient and pure functional programming remains an open research problem, though several research groups are working on addressing the problem. The approach in this book is concerned with imperative programs practiced widely in languages such as C++.

A solution approach to the problem of efficient large object movement is to represent large objects with one level of indirection using a reference. In some languages, this indirection is introduced implicitly, as with references in Java, C#, Lisp, and Smalltalk. In some other languages, such as Pascal, Ada, C, and C++, indirection needs to be programmed explicitly with pointers. We refer to both kinds of explicit and implicit pointers as *references*. Both have the same operational effect on execution performance: it takes constant time to copy a reference, whereas copying an object's entire representation is generally more expensive.[11]

2.2.1 Simplicity vs. Soundness of Reasoning

Ideally, we would like to use references, yet ignore the references in reasoning. In fact, this is typically what practicing programmers and students in object-oriented computing appear to do routinely: They use a simplified *clean* view of objects. In the *clean view*, one just ignores indirection and pretends that reference variables directly denote the values (i.e., representations) of the objects to which they refer.
Figure 9 illustrates three different views of a variable *U* whose type is a bounded stack of trees.

[11] Pointer arithmetic as in C or C++ adds additional complications to reasoning. To help focus this paper, we will ignore pointer arithmetic.

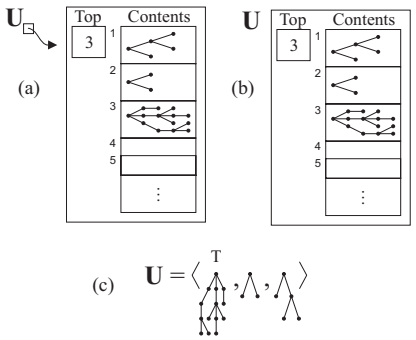

Figure 9: Three Views of a Bounded Stack of Trees

Figure 9(a) shows the actual representation of *u* as a reference to an elaborate structure. The view in Figure 9(b) is clean because it suppresses the fact that *U* is a reference to this representation data structure. Although the clean view of objects as values (with or without abstraction) is desirable for programming and reasoning about programs, it is *unsound* in the presence of aliasing. To see why, consider an example case of aliasing introduced by assignment of references. Suppose that a reference variable *V* is assigned another *U*. This assignment, written *V* = *U* in language such as Java, leads to aliasing between *U* and *V*; both variables refer to the same representation (location). In the clean view, *U* and *V* have the same value—which accurately reflects reality immediately after the assignment. Later, however, if the representation associated with *U* is modified, then the clean view and reality will disagree. In the clean view only the value of *U* changes, while the value of *V* remains the same. However, in reality there is just one shared representation that changes, leading to an inconsistency between the actual view shown in Figure 10(a) and the clean view shown in Figure 10(b). This inconsistency between the actual situation and a simplified view leads to unsound reasoning.

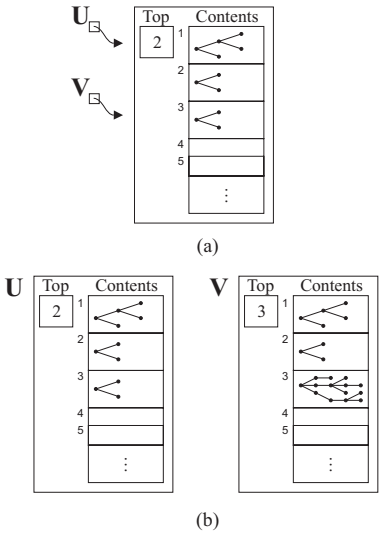

Figure 10: Two Views of Aliased Stacks – (a) Actual View and (b) Clean View

2.2.2 Understanding Why We Copy

Use of a reference in representing non-trivial objects, motivated by efficiency considerations, raises important software engineering questions that need to be adequately addressed. The most fundamental one is if the assignment statement (denoted by "=" or ":=") should be built into a language for non-trivial objects. If assignment is defined to mean "deep copying", then it becomes frequently expensive, and in general, infeasible for arbitrarily large structures. Alternatively, if assignment is defined to mean shallow reference copying, then aliasing and related understanding and reasoning complexities are introduced routinely in large object movement. Neither of these two choices nor any other that custom-made solution that is a combination of shallow and deep copying satisfies our dual concerns of efficiency and good software engineering.

Given the basic problems with deep and shallow copying, we begin with an examination of the purpose and usage of copying in computing. We consider a simple example. Suppose we need to reverse the contents of a stack S. This can be accomplished by the piece of code given in Figure 11. The code in this figure and subsequent figures are given in RESOLVE, a notation designed for ease of understanding, in the spirit of Pascal. Readers can readily imagine corresponding code in their favorite programming languages.

```
Clear(Temp);
While (Depth(S) > 0)
do
    Pop(Next_Entry, S);
    Push(Next_Entry, Temp);
end;
Copy_to(S, Temp);
```

Figure 11: An Example Case of Incidental Copying

In this situation, it is clear that the intent of copying from Temp to S is to transfer the value of Temp to S. In particular, computing that follows the copying statement usually involves S, but not Temp. In other words, following the copying, the value of Temp is rarely of consequence, though Temp must still have a meaningful value and it should behave like a stack variable.

This example leads us to explore another situation where copying is routinely used. The code for *Push*, *Pop*, or *Top* operations on Stacks typically involve copying. In *Push*, the parametric object is copied in to the container structure that represents a stack. In *Pop* (or *Top*), an object from the container structure is copied and returned to the caller. Is the copying that happens in *Push* and *Pop* incidental or essential? In the case of *Push*, the value of *Next_Entry* after the call is irrelevant because it is replaced by the next call to *Pop*. In other words, it is only *incidental* that Push makes a copy of *Next_Entry*. Similarly, once the top Stack element from the container is returned in *Next_Entry* in the code for *Pop*, it is unnecessary for the container to retain a copy of that top entry. Copying that takes place in *Pop* is also incidental. It is unlikely that a simple example like the one given here alone will convince a reader that copying has limited utility. That case cannot be made without a vast collection of examples. For now we have the following working hypothesis: When non-trivial objects are assigned to each other, the intent is usually information transfer, not copying: Similarly, information transfer is what is intended when objects are copied into and out of container structures such as arrays, stacks, trees. We conclude this subsection, noting that some modern programming languages have attempted to minimize the impact of the large object assignment problem, by including features that allow software developers to custom define their solutions. In these languages, for each user-defined object, programmers may choose to use the default assignment implementation for "=", which typically copies references for large objects. Alternatively programmers may redefine "=" to be a call to a specially-defined procedure that does expensive deep copying or something in between deep and shallow reference copying. Programmers may also preclude

assignments all together. While this approach appears to provide software developers the flexibility, the negative software engineering consequences of developing and integrating software objects each of which may define assignment statement differently are obvious. The meaning of object transfer just cannot be left to programmer discretion.

2.2.3 Swapping As an Alternative to Copying

Assuming that much of the copying is incidental, how exactly can we transfer the value of one object to another, efficiently? The alternative we use is swapping (though it is not the only alternative). Swapping replaces copying as our basic technique for object movement, though we retain value copying for objects such as Integers, which are easy to replicate without introducing references. The Swap statement, denoted by "X :=: Y", interchanges the values of X and Y, and it leads to transfers in both directions. Swapping is assumed to be defined implicitly on all objects and will serve as our basic object transfer mechanism. When large objects are uniformly (and automatically by a compiler) represented using a level of indirection, swapping works in constant time using a constant space, without introducing aliasing. Swapping has the additional benefit of leaving the transferred object with a meaningful value, which is important because there is no way to preclude its use. Where the value of transferred object is not of interest, it is easy to ignore its value. In other cases, computations can take advantage of the two-way transfer offered by the Swap statement.

The code in Figure 11 becomes as shown in Figure 12 where swapping is used instead of copying. The (minimal) impact of using swapping instead of copying in the code for Push and Pop operations is discussed in the next chapter.

```
Clear(Temp);
While (Depth(S) > 0)
do
    Pop(Next_Entry, S);
    Push(Next_Entry, Temp);
end;
S :=: Temp;
```

Figure 12: An Example Use of Swapping

Parameter Passing by Swapping

Swapping is a suitable technique to pass parameters to operations efficiently and uniformly, because of its advantages. To pass large objects efficiently, some programming languages use compiler-oriented parameter passing modes (e.g., passing by reference) to allow software developers to instruct compilers how certain parameters should be passed. This choice becomes automatic if the supporting system is designed suitably. In particular, the system can choose to reference all objects using one level of indirection, and optimize out the indirection for trivial representations such as for Integers. Large object parameters can then be transferred into and out of procedures efficiently using swapping. Therefore, RESOLVE notation does not include parameter passing modes, such as *var* or *val*. When any stack operation is invoked, for example, the representation of an arbitrary stack object as well as that of an object of an arbitrary type Entry are likely to be expensive to replicate, and they will all be passed by swapping using references by compilers. This uniformity in parameter passing thus eliminates the need for traditional compiler instruction modes on how parameters should be passed, allowing for a more general and important usage for modes as discussed in the next chapter.

For smaller objects, such as Characters and Integers, we allow both copying, denoted by ":=", and swap statements. Assignments are useful and inexpensive for smaller objects, and there is no reason to discourage their use. Swap statements are needed here for uniformity. This uniformity is important in parameter passing by swapping, and in developing implementations of generic objects such as stacks, where ":=:" can be used as a uniform operator for moving objects of generic type Entry.
While copying in the form of assignments is the major source of aliasing, there is yet another common source of aliasing, termed the *repeated argument problem*. Repeated arguments occur when two (or more) references to the same object are passed in a single call to a procedure, leading to aliasing among the formal parameters of the called procedure. Repeated arguments may occur implicitly, when two array elements are passed to a procedure, as in the call P(A[i], A[j]). In such situations, there may be no way to statically detect, in general, the aliasing that may result if i = j. A subtler kind of implicit repeated argument is also possible. This is aliasing that occurs between actual arguments and global variables that are accessible within the called procedure. For example, in the call P(x), if the procedure P also has direct access to the global variable X, then the procedure's environment introduces an alias between the global X and P's formal parameter.

Can parameter passing by swapping work in the presence of repeated arguments? Fortunately, one other advantage of swapping turns out to be that it offers a relatively straightforward and natural solution to this problem, as explained in a later section on overlapping parameters.

Another question concerning the use of swapping arises in the context of sub classing. The design principles espoused in this book generally avoid the need for sub classing. In language designs where sub classing is central, swapping between a super class object and a subclass object can be interpreted to have the semantics of destructive transfer: the value of the sub class object is transferred to the upper class object as intended, but the sub class object is left with an initial value specified for its type.

2.2.4 Unavoidable Uses of Deep and Shallow Copying

Though we have argued against replication of large objects and aliasing in routine data movement, occasionally replication and aliasing are essential. Where needed, replication can be handled easily by enhancing basic concepts with replication capability (a topic discussed later in this book).

While the objective is to minimize the complexity of reasoning by avoiding aliasing, an accurate view of objects with aliasing is necessary in low-level modules that deal with manipulation of references for efficient processing of list or tree-like structures. In these modules, aliasing is used intentionally to share storage, avoid copying, and allow multiple access paths. Thus in such modules aliasing and the corresponding reasoning complexity are unavoidable. On the other hand, most components in a large program are based on reusing objects, and in these higher-level components the value-based view offers significant software engineering benefits.

To separate the components where reasoning with the clean view is sound from those where it is not, data abstraction is crucial. In particular, one must abstract away the details of sharing and references found in the implementation of low-level modules by using abstract data types. For example, one can view a doubly linked list abstractly as a pair of strings or sequences, and in this abstract view there is no need to think about the sharing implicit in the representation. Internally, however, references and aliasing are employed. Since the number of such low-level modules would presumably be small, the additional annotations required by these type systems would not be such a drawback as they would be if one tried to use them in all modules, including the higher-level client modules. In this book we first focus on the more common higher-level client modules, where a clean view is both desirable

and possible. Later, we discuss a family of lower-level concepts to capture pointer structures. Our conclusion is only that replication or aliasing, introduced by assignment statements, is unsuitable for routine object transfer.

2.3 Behavioral Specification

The type and operation names listed in a component interface must be augmented with user-oriented explanations, because even the most meaningful names only hint at potential effects. Typically, informal explanations are given as illustrated in Figure 13.

2.3.1 Informal Specifications

```
Concept Stack_Template (type Entry; Max_Depth: Integer);
        (* requires that Max_Depth is greater than 0 *)

    Family Stack;
        (* initialization ensures that every stack S is empty *)

    Operation Push (E: Entry; S: Stack);
        (* requires that S is not full *)
        (* ensures that E becomes the top entry of S *)

    Operation Pop (R: Entry; S: Stack);
        (* requires that S is not empty *)
        (* ensures that the top entry of S is removed and that it is placed in R *)

    Operation Depth (S: Stack): Integer;
        (* ensures that depth of S is returned *)

    Operation Rem_Capacity (S: Stack ): Integer;
        (* ensures that the maximum number of entries that can be pushed on S is
returned *)

    Operation Clear (S: Stack );
        (* ensures that S is empty *)
end Stack_Template;
```

Figure 13: An Informal Interface Specification

It is arguably better to provide informal explanations than providing none at all. The explanations in do not make any mention of hidden details. However, there are several problems with such explanations. Figure 13 assumes that a user has some a

priori knowledge of a stack and understands the terms "empty stack," "top", and "depth". These assumptions may be indeed valid and reasonable for objects such as stacks and queues, but clearly for arbitrarily new objects the assumptions would not hold. The difficulties often arise in explaining concepts with which users are entirely unfamiliar. At one time or the other, to one audience or another, every concept is brand new. Even if this "assumed prior knowledge" is transferred to the audience, it has to be done for starting from scratch every new object, because every new one would use its own terminology. Arriving at a common terminology (e.g., using "size" for all objects instead of depth for stacks and length for queues) is often at odds with the idea of introducing metaphors that are well-known to simplify understanding. Unfortunately, there is really no good solution here.

A second problem with informal explanations in natural languages is the ambiguity inherent in those languages. What is perfectly clear and obvious to one reader seems unintuitive to another. Meyer documents a host of problems even with a carefully-conceived and carefully-refined natural language document for software specification in (Meyer, 1987).

Another key problem is one of completeness, though this problem is independent of whether the explanations are given in a formal or a natural language. The specification in Figure 13 leaves several questions unresolved. For example, consider the operation Push, where the specification only ensures that "E becomes the top entry of S." What about the rest of the entries in S? Do they remain the same? Do they remain in the same order? What can a user assume about E after the call? It is possible to refine the specification of Push and elaborate as shown in Figure 14.

Operation Push (E: Entry; S: Stack);
(* requires that the depth of S is not equal to Max_Depth *)

(* ensures that E becomes the top entry of S; that the entries that were in S before the call to Push remain the same and stay in the same order; and that the value of E becomes unspecified after the call. *)

Figure 14: A More Elaborate Specification of Push

The elaborate explanations are better, assuming that other more basic questions such as if E means its reference or value and what exactly a stack is, are resolved somewhere else. A careful reader might notice that even this elaborate explanation leaves plenty of room for ambiguity.

A key purpose of component specification is to facilitate reasoning about programs built using that component. Given specifications such as the one in Figure 14 for Push operation, there is really no systematic way in which a user can employ it in reasoning. The difficulty becomes apparent by comparing the reasoning process for the code based on Integers given in Figure 8. To reason mechanically or manually, but soundly, about a program such as the one in Figure 3, it is essential to have a mathematical conceptualization of Stack (and Integer) variables and operations. This is the topic of the next subsection.

2.3.2 A Formal Specification of Stack Behavior

A suitable mathematical model for a linear ordered structure such as a (parameterized) stack is a string or a sequence of entries. In general, identification of appropriate models is a difficult software engineering problem, because the chosen models should have just sufficient structure to capture essential behavior. While a model that is too simple may be inadequate, a model that is overly complicated makes it too difficult to comprehend. Among models that are equally appropriate, naturally we prefer ones that are more commonly used and understood in standard mathematics than defining and using new ones. Regardless, a concept specification language must make it possible to define and use mathematical models and corresponding vocabulary.

Given the view of a stack as a string, it is quite easy to explain the behavior of stack operations. For example, the string <3, 6, 1> denotes the value of an example integer stack. The behavior of Push operation can be explained as adding the new entry, say 7, to one front of the string and the Pop operation can be explained as removing the entry from the front of the string. Depth operation is specified to return the length of the string. The mathematical modeling makes it possible to explain behavior precisely and concisely. This behavior is captured formally in Figure 15 which shows a behavioral specification for Stack_Template.

Stack_Template **uses** String_Theory. String_Theory is a **math unit** that defines string theoretic terms precisely. This and other mathematical units are discussed in a later section. In String_Theory, Λ stands for an empty string, α o β denotes concatenation of two strings α and β in the specified order, and $|\alpha|$ denotes the length of a string α. (Note: The string concatenation symbol is often left implicit in discussions on formal languages, where $\alpha\beta$, is used for example, to denote concatenation of strings α and β.)

```
Concept Stack_Template(type Entry;  evaluates Max_Depth: Integer);
        uses Std_Integer_Fac, String_Theory;
            requires Max_Depth > 0;

    Type Family Stack  ⊆ Str(Entry);
        exemplar S;
        constraints |S| ≤ Max_Depth;
        initialization
            ensures S = Λ;

    Operation Push(alters E: Entry; updates S: Stack);
        requires |S| < Max_Depth;
        ensures S = <#E> ∘ #S;

    Operation Pop(replaces R: Entry; updates S: Stack);
        requires |S| > 0 ;
        ensures #S = <R> ∘ S;

    Operation Depth( restores S: Stack ): Integer;
        ensures Depth = ( |S| );

    Operation Rem_Capacity( restores S: Stack ): Integer;
        ensures Rem_Capacity = (Max_Depth - |S| );

    Operation Clear(clears S: Stack);
end Stack_Template;
```

Figure 15: A Concept for Locally Bounded Stack_Template

Given the inevitable role of mathematics in specifying concepts, it is essential that we use symbols and shorthand notations that are used routinely in mathematics. We freely use infix mathematical expressions/definitions, subscripts, and superscripts in this book, for example. Non-trivial concepts are likely to require usage of non-trivial mathematical formulations, and deviating from standard usage only introduces an additional level of complexity to what is already complex. Ultimately, software development systems must be able to handle the nuances of mathematics. In the interim, however, support systems may limit descriptions to use of only common keyboard characters. For example, **empty_string** and *, respectively, stand as the ASCII equivalents of terms Λ and ∘. An Appendix contains a listing of these and other equivalent terms.

In Stack_Template, values of Stack variables are conceptualized as strings of entries, denoted by **Str**(Entry). Using an **exemplar** stack variable S, the specification tells a client what is true of every stack variable. The **constraints** clause informs a client that a bounded stack just cannot become too long. From the **initialization ensures** clause, a client has the guarantee that whenever a stack variable is declared, it has Λ for its initial value (because the corresponding initialization procedure in the implementation module is automatically invoked by the support system). The practice of providing well-defined initial values, in situations where it is natural is often helpful to avoid a class of routine software errors. Occasionally, the initialization clause may specify a set of possible initial values, instead of a single value. In yet other cases, initialization clauses and the finalization clause may be omitted. However, it is important to note that an unspecified initialization clause still allows a client to assume that a variable has a value from the specified mathematical space upon declaration, but only that a particular value is not known. This principle that a variable will always have a value from the mathematical domain of its type makes both formal and informal reasoning easier, because it eliminates special handling of variables having "undefined" values.

In general, initialization (and other operations) requires that there be sufficient memory capacity to execute the operation successfully. The specification also needs to include global storage capacity-related requirements, at least implicitly. We defer discussion of this important topic to a later chapter.

The rest of the concept provides specifications of other Stack operations. User-oriented specifications of operations are expressed naturally in terms of the requirements of users to call an operation and the externally visible effects of calling the operation. When users violate the requirements, guarantees become void. Operation Push, for example, obligates the user through its **requires** clause that there be space on the parametric stack object S for an additional entry. The **ensures** clause guarantees that after Push is called, stack S will be updated to be the incoming value of entry #E concatenated with the incoming contents #S of the stack. Push does not specify how the entry E might be altered by this call. In general, we allow ensures clauses of operations to be loose to allow maximum flexibility for implementers.

It is instructive to compare the short, yet precise specification of Push in Figure 15 with the verbose, yet imprecise informal specification of the same operation in Figure 14. Arguably, there is some effort involved in developing mathematical specifications because suitable models need to be identified. This effort is easily amortized, when the several potential reads of the reusable specification are

considered. With the informal specifications, every reader would have to spend considerable time in interpretation. Even worse, there may be different interpretations for different readers, and all these interpretations may be different from the intent of the specification writer. Our experience has shown that, with little training, undergraduate Computer Science students have little difficulty in reading formal specifications.

The rest of the specification describes the requirement and guarantee clauses on the Pop, Depth, Rem_Capacity and Clear operations precisely in string-theoretic terms. The specification of Clear gives its parametric stack S the initial stack value Λ, and it gets this meaning from its parameter mode as explained in the next subsection. The important point here is that by conceiving of stacks as strings, we can give a complete and coherent explanation of all of the operations on stacks. Absolutely no reference to details of any one implementation such as arrays, pointers, or linked lists is needed. Absence of such details simplifies understanding of the concept for users yet provides developmental freedom for implementers of the concept, so long as users and implementers adhere to their obligations and guarantees stated in the specification.

The only operator that can be used on objects other than those explicitly listed in the concept is the swap operator. For example, if S and T are declared to be of Stacks from the *same* facility, then their values can be transferred to each other using the statement S :=: T. Neither classical assignment operator nor equality checking operator is defined implicitly on non-trivial objects, such as Stacks. Syntactic errors will result if they are used on Stack objects.

We conclude this section noting that nothing precludes defining concepts that are inconsistent. For example, the constraints on the provided type and specifications of operations may not be consistent with each other. This inconsistency in the specification will be eventually detected when attempting to show correctness of a realization for the specification. Though it will be nice to have a system that finds inconsistencies at the time of concept definition even before realizations are developed, it is not possible to automate this process in general. However, a sophisticated supporting system may give warnings in certain straightforward situations.

Specification Parameter Modes

The role of parameter modes in a language that permits specification of operations is less apparent because the effects of operations can be expressed adequately using

requires and ensures clauses. In a system where all parameters are passed uniformly and efficiently by swapping, the role of modes is even less obvious. Our use of modes is motivated by a combination of human and software engineering considerations ranging from comprehensibility to efficiency. Modes inform users, and in some cases compilers and verifiers, intended roles of parameters. They make specifications more readily comprehensible, permit detection of a class of routine specification errors, and allow efficiency considerations and compiler optimizations. This role of parameter modes is more general than their usual role in programming languages. At least seven different modes are useful as explained here.

In Stack_Template, a stack object S is passed to Push operation using the *updates* mode. The ensures clause gives a final value to S at the end of the operation based on #S, its initial value before the call. This is normally the mode used to pass large objects that encapsulate non-trivial data structures to their constructor operations that are intended to produce (small) purposeful changes. The entry object E to Push is passed using *alters* mode to inform clients that the implementer of the operation may leave an arbitrary value (of type Entry) in E at the end of the operation. Clients might eventually deduce this by noticing that the ensures clause makes no mention of the final value of E, but alters mode makes it manifest that E will be trashed.

An alternative specification of the Push operation may require that the entry will be re-initialized to the initial value of type Entry by passing E using *clears* mode. If this mode is used, the ensures clause needs to make no mention of the final value of E either, but that's because this mode adds an implicit Entry.**Is_Initial**(E) conjunction to the ensures clause. The language-defined predicate Is_Initial is true if and only if its parameter has an initial value for its type. Use of clears mode instead of alters mode demands that the implementer make sure that E is in fact reinitialized at the end of the operation. In a more demanding, but commonly used specification of Push, entry E may be passed using the *restores* mode. If this mode is used, the ensures clause needs to make no mention of final value of E either, because the *restores* mode adds an implicit E = #E term to the ensures clause. Here the implementer may temporarily change to value of E, but he is responsible to bring it back to its starting value.

A variety of considerations motivate the use of alters mode over restores or clears mode for passing objects of an arbitrary Entry type to Push operation. A software engineering problem with use of restores (or clears) mode is over-specification. Use of alters mode provides most flexibility to the implementers and can be realized most efficiently, without requiring neither reinitialization nor expensive replication and restoration of an arbitrary object of an arbitrary type. Alters mode is also most

suitable from a client perspective, because depending on the needs of his application, the client can optionally replicate the entry object before calling Push or clear the entry object on return. This rationale for using alters mode for the generic type Entry in Push operation suggests that in general, alters mode is normally the most appropriate choice for inserting arbitrary entries into generic data structures.

Entry object R is passed in the *replaces* mode to Pop operation to inform a user that the initial value in R is irrelevant and that it will be replaced by the value specified in the ensures clause. The specification of functional operation Is_Empty (and Is_Full), through the use of the restores mode for stack S, guarantees that the final value of parametric stack object after the operation will be the same as its initial value. The *preserves* mode, not used in Stack_Template specification is similar to the restores mode in that it adds the term S = #S to the ensures clause, but different because it also adds a stronger constraint that the implementer must never change the value of S. This mode is intended to allow sharing of preserved objects among several processes in a concurrent system.

We want functional operations such as Depth and Rem_Capacity to retain as many of the familiar properties of mathematical functions as possible. In particular, side effects, such as changing values of parameters, are undesirable. Therefore we allow only *restores*, *preserves*, and *evaluates* modes for parameters. The meanings of restores and preserves mode parameters are already established. The *evaluates* mode demands that any argument passed for that parameter must be a functional expression that will be evaluated when the operation is called. The mode is typically used in the context of objects such as Integers that are returned as results of a variety of functional operations. The evaluates mode allows such function calls to be used in parameter passing[12].

While we want to allow loose ensures clauses that are relational for general procedures in order to avoid over-specification, for functions we need the ensures clause to provide a single-value for the result. For this reason, we insist upon an explicit formula in the ensures clause. Ideally we would also like to have our functions be totally defined, but sometimes requires clauses may be needed. The returned value of a function may be assigned to a variable of the function return type to replace the value of that variable.

---

[12] We have used the more general evaluates mode also for the Max_Depth parameter to the concept. This is to allow the generality of passing an
Integer expression as an argument at concept instantiation time.

The type Integer used in the specification of function operations Depth and Rem_Capacity come from Std_Integer_Fac, which is why this facility is listed in the **use** clause. Std_Integer_Fac is an instantiation of Integer_Template realized in a standard way (possibly in hardware). Where Boolean variables are necessary, such as in a concept that provides Is_Empty and Is_Full operations, Std_Boolean_Fac can be imported through the uses clause. Std_Boolean_Fac is a facility of Boolean_Template realized in a standard way. Concepts for Boolean_Template, Integer_Template, Array_Template, Record_Template, and Pointer_Template and others that are normally considered to be built into languages are given in a separate section. Integer_Template, for example, defines typically built-in type Integer and common Integer operators. The uniformity handling of all types makes it easy for both users and support systems, because, for example, the same principles of behavioral reasoning apply regardless of whether a module uses integers, arrays, records or generic stacks.

Specification of an Object-Oriented Stack Concept

We conclude this chapter with an explanation of an object-oriented version of stack specification. As noted in section 2.1.2, this design is less desirable. Our main objective here is to illustrate essential notations for writing specifications based on abstract module level (or facility level) variables. Concepts of some data structures (e.g., pointers) turn out to be more natural when module level variables, in addition to types, are defined and used.

The specification in Figure 16 corresponds to the interface in Figure 5, and it provides a single stack object instead of a stack type. The object-oriented stack concept provides a stack object that is conceptualized as a string of entries. The initialization ensures clause gives an initial value to this object at the time of facility instantiation. For all facilities, the RESOLVE support system uniformly invokes facility-level initialization and finalization procedures from the implementation when a facility enters and exits scope, respectively. This is how the Self object's initialization procedure is invoked automatically.

In the specification, operations OO_Push and OO_Pop update facility-level variable Self, whereas OO_Depth and OO_Rem_Capacity operations restore it. The restores clauses of OO_Depth and OO_Rem_Capacity operations may be omitted, because facility-level variables are assumed to be restored, by default, unless specified otherwise.

Other common stack specifications are explored in the exercises.

```
Concept Object_Oriented_Stack_Template(type Entry;
                                       evaluates Max_Depth: Integer);
        uses Std_Integer_Fac, String_Theory;
            requires Max_Depth > 0;

    Var Self: Str(Entry);
        constraints |Self| ≤ Max_Depth;
        initialization
            ensures Self = Λ;

    Operation OO_Push(alters E: Entry);
        updates Self;
        requires |Self| < Max_Depth;
        ensures Self = <#E> ∘ #Self;

    Operation OO_Pop(replaces R: Entry);
        updates Self;
        requires |Self| > 0 ;
        ensures #Self = <R> ∘ Self;

    Operation OO_Depth(): Integer;
        restores Self;
        ensures OO_Depth = ( |Self| );

    Operation OO_Rem_Capacity (): Integer;
        restores Self;
        ensures OO_Rem_Capacity = ( Max_Depth - |Self| );

    Operation OO_Clear();
        clears Self;
end Object_Oriented_Stack_Template;
```

Figure 16: Specification of an Object-Oriented Stack_Template

Exercises

1. Summarize the disadvantages of assignment for large object data transfer. Explain why swapping is the most suitable realization of transfer. In particular, consider the following alternatives to (swapping) for object transfer: (a) the transferred object is set to a special reference value, say Null, after transfer. (b) the initialization operation is automatically invoked on the transferred object.
2. Give examples of situations where a replica of a large object is needed. How would you create a stack replica?
3. Give examples of situations where aliasing is useful.
4. Show that it is possible to implement Rem_Capacity and Clear operations using the other three stack operations. What is the time complexity of your implementation?
5. It is possible to implement an Is_Empty operation whose specification and implementation are shown below. Write the specification and implementation of an Is_Full operation.

```
    Operation Is_Empty( restores S: Stack ): Boolean;
        ensures Is_Empty = ( |S| = 0);
    Procedure
        If (Depth(S) = 0) then
1.                  Is_Empty := true
2.          else
3.                  Is_Empty := false;
4.          end;
5.      end Is_Empty;
6.
```

6. Consider substituting Is_Full operation that returns whether a stack is full instead of Rem_Capacity operation. Now it is possible to check if a stack is full in a module other than the one in which the concept is instantiated. Contrast the current interface design that includes Depth and Rem_Capacity operations with the following others: Depth/Is_Full operations and Is_Empty/Is_Full operations.
7. Defensive_Stack_Template is another commonly used specification of stack behavior in which stack implementations are responsible for notifying callers when operations are called improperly.
   A. Complete this specification that provides Stack type and operations Defensive_Push, Defensive_Pop, and others.

B. Explain why Stack_Template is more desirable than Defensive_Stack_Template.
C. Are Depth and Rem_Capacity operations necessary for Defensive_Stack_Template?
D. Rewrite Defensive_Push and Defensive_Pop operations as side-effecting functions so that they return boolean error values. What are the negative consequences of using side-effecting functions for software engineering?

```
Concept Defensive_Stack_Template (type Entry;
                    evaluates Max_Depth: Integer);
      uses Std_Boolean_Fac, Std_Integer_Fac, String_Theory;
           requires Max_Depth > 0;

   Type Family Stack ⊆ Str(Entry);
      exemplar S;
      constraints |S| ≤ Max_Depth;
      initialization ensures S = Λ;

   Operation Defensive_Push (alters E: Entry; updates S: Stack;
                                          replaces full_flag: Boolean);
      ensures if |#S| < Max_Depth then S = <#E> ∘ #S and not full_flag
                        and if |#S| = Max_Depth then full_flag;

   Operation Defensive_Pop(replaces R: Entry; updates S: Stack;
                                        replaces empty_flag: Boolean);
      ensures ???;

   Operation Depth( restores S: Stack ): Integer;
      ensures Depth = ( |S| );

   Operation Rem_Capacity( restores S: Stack ): Integer;
      ensures Rem_Capacity = (Max_Depth - |S| );

   Operation Clear(clears S: Stack);
end Defensive_Stack_Template;
```

Figure 17: A Concept for Defensive Stack_Template

8. Copy_Based_Stack_Template is a commonly used specification of stack behavior that is based on the assumption that it is inexpensive to copy entries.

A. Complete such a specification that provides Stack type and operations Copying_Push, Remove_Top, Copy_Top, Depth, and Rem_Capacity. Copying_Push restores the value of its parametric entry on return. Remove_Top that removes the top entry from its parametric stack, but does not return it to the caller. The specification of Copy_Top is given below.

Operation Copy_Top(**replaces** Top_Entry: Entry; **restores** S: Stack);
requires $|S| > 0$;
ensures $\exists$ Rst: **Str**(Entry) $\ni$ S = <Top_Entry> $\circ$ Rst;

B. Explain why Stack_Template is more desirable than Copy_Based_Stack_Template.

9. The principles of defining single instance, defensive, and copy-based versions of stacks are orthogonal and can be combined. Specify a commonly used Copy_Based_Def_OO_Stack_Template and explain its limitations. (An exercise in the next chapter illustrates how this concept can be realized using Stack_Template.)
10. Complete the specification of Invented_Here_Stack_Template that defines Stack type and Push_Arb_Entry, Remove_Top, Swap_Top, Depth, and Rem_Capacity operations. Your concept should provide essentially the same overall functionality as Stack_Template. Why is this specification less desirable than Stack_Template?
11. Complete the specification of Also_Invented_Here_Stack_Template.

```
Concept Also_Invented_Here_Stack_Template     (type Entry;
                                                evaluates Max_Depth: Integer);
        uses Std_Integer_Fac, String_Theory;
              requires Max_Depth > 0;

    Type Family Stack ⊆ (Top: Entry; Rest: Str(Entry));
        exemplar S;
        constraints |S.Rest| < Max_Depth;
        initialization
              ensures S.Rest = Λ;

    Operation Swap_Top(alters E: Entry; updates S: Stack);
        requires true;
        ensures E = #S.Top and S. Top = #E and S.Rest = #S.Rest;

    Operation Move_Down(updates S: Stack);
        requires |<S.Top> ∘ S.Rest| < Max_Depth ;
        ensures S.Rest = <#S.Top> ∘ #S.Rest;

    Operation Move_Up(updates S: Stack);
        requires |S.Rest| > 0 ;
        ensures <S.Top> ∘ S.Rest = #S.Rest;

    Operation Depth( restores S: Stack ): Integer;
        ensures Depth = ( |S.Rest| + 1);

    Operation Rem_Capacity( restores S: Stack ): Integer;
        ensures Rem_Capacity = (Max_Depth - |S.Rest| – 1);

    Operation Clear(clears S: Stack);
end Also_Invented_Here_Stack_Template;
```

12. It is possible to specify a variety of other stack operations. Specify operations to check empty and full stacks, to replicate a stack, and to test equality of two stacks. The specification of a stack reverse operation is given below as an example. What are the ramifications of including these and other potentially useful operations such as Is_Empty, and Is_Full operations in an All_In_One_Stack_Template?

Operation Flip (**updates** s: Stack);
ensures S = #S^{Rev};

13. In RESOLVE, the swap operator can be used only to swap two variables of the same type from the same facility. The swap operator is neither available nor useful on single instance stacks, because each facility provides only one stack. Examine the ramifications of allowing assignment or swapping of stacks from different facilities on language design, compilation, static analysis, and runtime execution.

3 Principles of Implementation and Enhancement

3.1 Basics of Component Implementation

This chapter discusses aspects of component implementation design and development. The focus of this first section is on developing implementations (or realizations) of data abstractions. The second section explains how data abstractions can be enhanced with additional operations and how these new operations can be implemented.

The example implementations given in this chapter illustrate a variety of issues. They illustrate how parameterized or generic components can be implemented efficiently and elegantly using the swapping paradigm. They explain the performance rationale for developing and using multiple implementations of the same specification. A system that allows development of specifications and implementations must make it possible to show that an implementation is correct with respect to its specification, and to analyze the performance behaviors of the implementation. To accomplish this goal, implementations must be annotated with suitable mathematical assertions. The annotations, while necessary for automated reasoning, are shown to be critical for documenting key implementation design decisions. The documentation in turn makes it easier for human understanding and maintenance. The chapter includes an informal performance comparison of implementations.

Implementations are written in a programming language-like notation so that a compiler can easily translate it to efficient executable code. This notation is necessarily quite different from the mathematical notation used in specifications and assertions where the purpose is formal description of behavior to users (and verifiers). Use of a single notation for both purposes, invariably compromises either specification or programming demands.

3.1.1 A First Implementation with Assertions to Capture Design Rationale

Identification of suitable structures for implementation of a concept, in general, is far from automatic. Typically it is a non-trivial design exercise involving consideration of a variety of performance trade-offs. Often, there is no one best performing realization of a concept. It becomes necessary to develop and use alternative implementations for a (reusable) concept to satisfy the needs of different applications of the concept. In this subsection, the focus is on two implementations of Stack_Template. Figure 1 provides a pictorial context for the implementations.

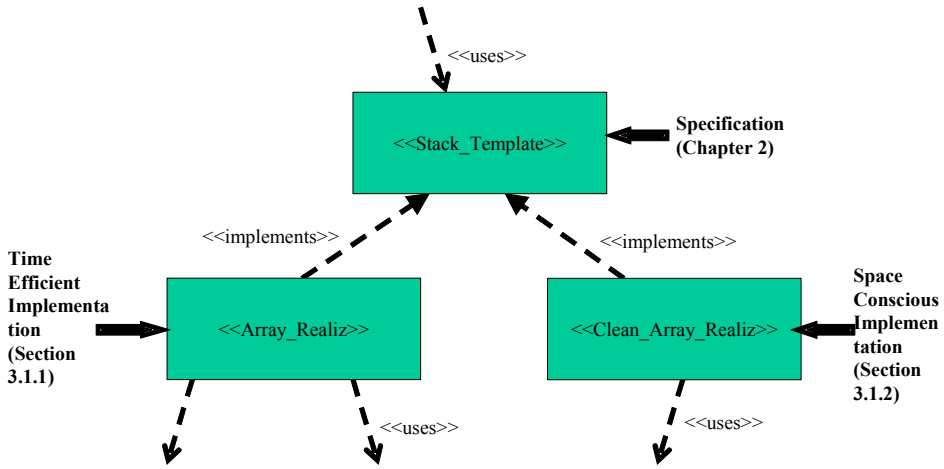

Figure 1: A Diagram to Provide Context for Implementations

A First Implementation

Figure 1 contains a skeleton of one implementation of Stack_Template that uses standard realizations of arrays and records. The figure is missing mathematical assertions that capture key design decisions and that are essential in reasoning that the implementation is correct. The assertions are discussed in the next subsection. The generic implementation makes use of template parameters Entry and Max_Depth to create a suitable array to hold the contents of a stack. The realization in Figure 1 is straightforward, but it does differ from a more common implementation of Stack_Template in a typical programming language in subtle, but important ways as explained in this section.

```
Realization Array_Realiz for Stack_Template;
        uses Record_Template, Static_Array_Template;

    Type Stack = Record
                        Contents: Array 1..Max_Depth of Entry;
                        Top: Integer;
                end;

        initialization (* Contents Array and Top Integer are initialized.
                        No additional code is necessary. *)
        end;

        finalization (* Rep. is finalized. No additional code is necessary. *)
        end;

    Procedure Push(alters E: Entry; updates S: Stack);
        S.Top := S.Top + 1;
        E :=: S.Contents(S.Top);
    end Push;

    Procedure Pop(replaces R: Entry; updates S: Stack);
        R :=: S.Contents(S.Top);
        S.Top := S.Top - 1;
    end Pop;

    Procedure Depth(preserves S: Stack): Integer;
        Depth := (S.Top);
    end Depth;

    Procedure Rem_Capacity (preserves S: Stack): Integer;
        Rem_Capacity := (Max_Depth – S.Top);
    end Rem_Capacity;

    Procedure Clear(clears S: Stack);
        S.Top := 0;
    end Clear;
end Array_Realiz;
```

Figure 2: Programmatic Aspects of an Array-Based Realization of Stack_Template

RESOLVE implementation language, like most programming languages, includes special syntax to make it easy to use standard realizations of commonly-used types such as arrays, records and integers, and their operations. However, like all other types, these types are defined through concepts listed in the realization **uses** clause. The concepts provide suitable mathematical models for the types, initial values for variables of the types, and operations to manipulate the variables of the types as detailed in a later chapter. Before arrays or records can be used, the corresponding parameterized concepts Static_Array_Template and Record_Template need to be instantiated and realized in facility declarations much in the same way as any other concept. The syntax serves merely as a shorthand for these more elaborate declarations. It is possible, for example, that a library will contain alternative array and record concepts, as well as alternative realizations of the concepts, but to use non-standard realizations, no special syntax is available; explicit facility declarations need to be created[13].

Integers, arrays, and records are initialized as specified in the initialization ensures clauses in their respective concepts. When a client declares a stack variable, a record is created with its *Top* Integer field initialized to 0 and each entry of its *Contents* array initialized to a proper value for type Entry. To supplement (but not in lieu of) automatic representation initialization, a Stack realization optionally may provide an initialization procedure to change initial values. For example, a different realization of Stack_Template may need the Top field to be initially 1 instead of 0, and it can do so by providing code for initialization. Similarly, a finalization procedure may be provided optionally. In the current realization, empty bodies of initialization and finalization are shown only to illustrate how explicit code for those procedures could be provided, when needed. These procedure bodies default to empty bodies and can be omitted. The initialization/finalization procedures are invoked automatically on when a stack variable enters/exits scope, respectively. Exercises and later chapters explore more time efficient array and stack realizations that amortize the cost of array initialization/finalization.

---

[13] The realization does not list Std_Integer_Facility in the uses clause, though integers are used, because it is already included in the uses clause of concept Stack_Template. Complete facilities have been previously defined for booleans and integers, because Boolean_Template and Integer_Template have no user-supplied parameters, unlike arrays and records. Before arrays and records can be used, in principle, the corresponding templates must be instantiated. However, we have included syntax to avoid the need for explicit instantiation when standard realizations of these concepts are used.

The realization contains a **procedure** for each Stack operation, based on the record representation of a stack. The procedures for Push and Pop swap entries into and out of array locations, instead of copying entries using assignments. The :=: operator used here is actually a syntactic shorthand for a call to an array operation that swaps the entry at a specified array index with a given entry.

Assertions to Capture Design Rationale and Ease Reasoning

Implementations need to be annotated with assertions that capture key design decisions. Two key assertions (among others) are essential to capture the rationale in data abstraction implementations. By specifying these assertions, a designer in charge of an implementation may dictate or outline the implementation process for one or more developers involved in writing that implementation.

The first of these assertions concerns abstraction in specification. While stacks are conceptualized as mathematical strings of entries in the presentation to users in Stack_Template, when Array_Realiz is used to create a facility of Stack_Template, Stack type variables are actually represented as records containing an array and an Integer. Unless there is a proper correspondence between the abstract view of a stack given in the specification and the representation used in the implementation of a stack, it is not possible to reason that the implementation actually captures the specification. More generally, to facilitate mechanical or manual reasoning that a realization is correct with respect to its specification, a realization needs to be annotated with a variety of assertions. Figure 3 completes the array realization by providing type correspondence and conventions clauses. In the figure we have omitted the empty initialization and finalization procedures.

The **correspondence** clause explains how to read the value in the *Contents* array as the string described in conceptualization. Here it says that the conceptual level string **conc.**S corresponds to the string formed by concatenating together the *S.Contents* array entries from 1 to *S.Top* and then reversing this string. The mathematical definitions for string concatenation and reversal, denoted by Π and *Rev*, respectively, are formally defined in String_Theory (section #). The correspondence clause is necessary to check that the code for procedures such as Push and Pop based on the array representation of a stack satisfy their respective specifications based on a string model of a stack in Stack_Template concept[14].

[14] To write realization assertions, mathematical models of the structures used in the realizations such as arrays and records are essential. In Static_Array_Template, an

The **conventions** clause contains constraints on the realization records for stacks which make it possible to interpret them as strings as well as information which is needed to keep the implementations of all the various operations consistent. Here the convention is that *S.Top* always have a value which makes sense in the correspondence. Each procedure may assume that the Stack representation satisfies the convention when the procedure is called. At the end, each procedure needs to leave the representation in a state that satisfies the convention.

The conventions clause also called *representation invariant*, is usually written ahead of the correspondence clause because the correspondence needs to be interpreted only for representations that satisfy the conventions. The correspondence clause is also called an *abstraction function* or more generally, an *abstraction relation*, because it gives a conceptual value for a given representation value. It is essential to check that the clause is "well defined", i.e., for every representation state that satisfies the conventions, the correspondence clause must define at least one conceptual value. In general, multiple representation values may correspond to the same conceptual value, and multiple conceptual values may correspond to a single representation value. In the array realization of stacks, for example, all records with the same Top integer and with the same entries in the array between 1 and S.Top map to the same conceptual stack, regardless of what entries are stored between S.Top + 1 and Max_Depth in the array.

---

array of entries is modeled as a mathematical function from integers to entries, and hence, *S.Contents* is treated as a function in the correspondence assertion. A record with 2 elements is modeled as an ordered pair in Record_Template.

```
Realization Array_Realiz for Stack_Template;
        uses Record_Template, Static_Array_Template;

    Type Stack = Record
                    Contents: Array 1..Max_Depth of Entry;
                    Top: Integer;
                end;
            conventions 0 ≤ S.Top ≤ Max_Depth;
            correspondence Conc.S = (∏ i=1 to S.Top <S.Contents(i)>)^Rev ;

    Procedure Push(alters E: Entry; updates S: Stack);
        S.Top := S.Top + 1;
        E :=: S.Contents(S.Top);
    end Push;

    Procedure Pop(replaces R: Entry; updates S: Stack);
        R :=: S.Contents(S.Top);
        S.Top := S.Top - 1;
    end Pop;

    Procedure Depth(preserves S: Stack): Integer;
        Depth := (S.Top);
    end Depth;

    Procedure Rem_Capacity (preserves S: Stack): Integer;
        Rem_Capacity := (Max_Depth – S.Top);
    end Rem_Capacity;

    Procedure Clear(clears S: Stack);
        S.Top := 0;
    end Clear;
end Array_Realiz;
```

Figure 3: Array-Based Realization with Assertions

In addition to correspondence and conventions assertions, realizations also need to include other assertions such as invariants for loops, as discussed in the next chapter. These assertions are needed to check realization correctness, regardless of whether

correctness is to be established mechanically and manually. They should be supplied by realization writers who hold the intuition for why together the chosen structures and procedures constitute a valid implementation of a concept. When it is not possible to show that a realization satisfies its concept, either its code or its assertions need to be re-written.

The mathematics of why a realization does or does not work correctly is at least as much a part of software development as "programming". Software developers ought to be equipped with techniques to prepare correct and efficient realizations, because it is infeasible to generate (efficient) realizations or realization assertions automatically. However, current verification technology does make it possible check and use programmer-supplied assertions in verification of correctness with limited human intervention in the process.

3.1.2 Performance Trade-Offs

A Space Conscious Implementation

Figure 4 contains an alternative realization of Stack_Template. This implementation uses the same representation as the other one for a stack. However, the designer has identified minimizing space usage as the key issue. To accomplish this goal, the designer has specified that objects that are not part of the current conceptual stack should be released as soon as possible. In other words, the part of the internal array that does not contribute to the conceptual stack must not contain arbitrary objects (except initialized objects).

To see how this is done, first note that the correspondence assertion here is identical to the assertion in the previous implementation. The difference is that there is an additional representation convention of keeping all unused array elements "clean". By this convention, any array entry that is not a part of the stack has an initial value for the Entry type. In expressing the convention, we have defined and used a local mathematical predicate *Array_Is_Clean* on the stack representation. This definition is in turn based on the language-defined predicate **Is_Initial** for type Entry.

```
Realization Clean_Array_Realiz for Stack_Template;
    uses Record_Template, Static_Array_Template;

  Type Stack = Record
                 Contents: Array 1..Max_Depth of Entry;
                 Top: Integer;
               end;
    Definition Array_Is_Clean( SR: Stack ): B = (∀i: Z, if SR.Top < i
        and i ≤ Max_Depth then Entry.Is_Initial(SR.Contents(i));
    conventions 0 ≤ S.Top ≤ Max_Depth and Array_Is_Clean( S );
    correspondence Conc.S = ( ∏_{i=1}^{S.Top} <S.Contents(i)> )^{Rev} ;

  Procedure Push(clears E: Entry; updates S: Stack);
    S.Top := S.Top + 1;
    E :=: S.Contents(S.Top);
  end Push;

  Procedure Pop(replaces R: Entry; updates S: Stack);
    Var Fresh_Val: Entry;
    R :=: S.Contents(S.Top);
    S.Contents(S.Top) :=: Fresh_Val;
    S.Top := S.Top - 1;
  end Pop;

  Procedure Depth(preserves S: Stack): Integer;
    Depth := S.Top;
  end Depth;

  Procedure Rem_Capacity (preserves S: Stack): Integer;
    Rem_Capacity := (Max_Depth – S.Top);
  end Rem_Capacity;

  Procedure Clear(clears S: Stack);
    Var Fresh_Stk: Stack;
    S :=: Fresh_Stk;
  end Clear;
end Clean_Array_Realiz;
```

Figure 4: A Space Conscious Implementation of Stack_Template

Use of the clean array convention naturally leads to Stack procedures that are different from those in the first realization in Figure 2. In Clean_Array_Realization, notice that Pop procedure uses a local variable to finalize the (arbitrary) entry supplied as a parameter to Pop to leave an initial Entry value in the array location from which the top-most entry is returned. This is essential to maintain the convention. The Clear procedure is also slightly more complicated because of the convention, because it is essential to have a clean Contents Array. For this purpose, the procedure creates a new stack and exchanges it with the parametric stack. The alternative strategy of immediately finalizing potentially large, arbitrary entries and keeping only clean or initial entries in the array leads to efficient storage use at the cost of time for cleaning entries. Time-space trade-offs such as these routinely arise in software design. A software development method must make it possible to document and compare performance estimates of alternative realizations of a concept. The topic of performance estimation and analysis are covered in detail in later chapters.

In Figure 4, notice that the Entry is specified to be of *clears* mode in procedure Push, whereas it is loosely specified to be in *alters* mode in the specification of Stack_Template. Though the specification does not demand that the pushed entry have any particular value on return, by giving it clears mode, the implementer is recording more information than necessary. This information is not available to a client who must rely only on the specification in Stack_Template. However, it is potentially useful, for example, in readily deciding that for a Stack_Template with a clearing Push specification, Clean_Array_Realiz provides a valid implementation. Similarly, the realization writer is recording that Is_Empty and Is_Full procedures *preserve* their parametric stacks, though the specification requires that they only restore them. Such information potentially is useful in concurrent computing. Naturally, realizations can only strengthen the modes given in the concept specifications. In particular, only the following mode changes are legal. A realization may use of clears, preserves, restores, and alters modes for a parameter in the alters mode in the specification. It may use restores or preserves mode for a parameter in the restores mode in the specification.

An Informal Performance Comparison

While correctness is a fundamental concern in any implementation, good performance is an essential property a useful implementation. Ideally, the performance estimate for a realization will be expressed in conceptual terms that are understandable to users without a complete knowledge of the realization. Estimates of both execution time durations as well as memory capacity requirement are needed to ensure that a realization is suitable for an application. Though historically the focus has been on the former, the importance of the latter for good software engineering cannot be overstated. We would like to able to predict not only how long it will take our software to execute, but also if and when our systems will run out of memory. The time and space estimates allow users to pick from among alternative realizations of a concept, ones that best fit the performance needs of their applications.

Stack_Template concept, chosen as the first example for its familiarity and simplicity, is not a prime candidate for motivating alternative realizations and related performance trade-offs. Later chapters contain more compelling examples, as well as discuss other important topics including formal definitions of notations for expressing performance characteristics, exact time analysis, and specification of precise memory capacity requirements. The present discussion is to provide only an informal performance comparison of the two stack realizations and to outline the kinds of time/space trade-offs that arise in devising efficient realizations for generic data abstractions. In the Clean_Array_Realization, to maintain the clean array convention, Pop and Clear procedures initialize and finalize an entry and a stack, respectively. These procedures are slower than their counterparts in the first realization. Clear procedure is particularly inefficient. The first realization, in general, uses more storage to achieve improved execution efficiency. To see why this is the case, notice that the Pop procedure in the first realization (a.k.a. regular array realization) merely swaps the top stack entry with the replaced parametric entry, leaving an arbitrary entry in the array. Therefore, array locations that do not correspond to the abstract stack may contain arbitrary entries. Since an arbitrary entry usually takes up more storage than an initial entry for complex entry types, in general, the dirty array representation hoards more storage than the clean array representation for the same abstract stack.

Before concluding this subsection we note that the clean array implementation of Stack_Template can be written using the first implementation of Stack_Template, without starting from scratch. The idea is to use the more efficient regular array realization, and always provide it with clean entries as parameters to Pop. The

principles of writing such an implementation are discussed in the next section and explored in the exercises.

3.1.3 Implementation of One Component Reusing Another

Once concepts such as Stack_Template are specified and realized, they can be used to build realizations for other concepts much in the same fashion as how array and record concepts are used to build realizations. This section contains a few examples that illustrate a variety of ideas in "layering" one reusable component using one or more previously developed components. The examples additionally support the generality in the design of Stack_Template and show that it is more fundamental, because it can be used to implement other variations. Exercises explore additional examples.

Specification and Implementation of a Traditional Stack_Template

For a first example, we consider a more classical Stack_Template component in which Stack operations copy entries (instead of swapping them). What we copy here are "values" of entries without introducing a reference-value distinction. Figure 5 contains the specification of Copy_Based_Stack_Template. Values of Stack type here are modeled as strings of Entries as in Stack_Template, but the specification of operations Copying_Push, Copy_Top, and Removing_Pop are different. Unlike Push, Copying_Push restores its parametric entry. Unlike Pop, Removing_Pop does not return the top entry. Instead the concept provides is another operation Copy_Top to retrieve the top-most entry.

```
Concept Copy_Based_Stack_Template (type Entry;
                                   evaluates Max_Depth: Integer);
        uses Std_Boolean_Fac, Std_Integer_Fac, String_Theory;
            requires Max_Depth > 0;

    Type Family Stack ⊆ Str(Entry);
        exemplar S;
        constraints |S| ≤ Max_Depth;
        initialization
            ensures S = Λ;

    Operation Copying_Push (restores E: Entry; updates S: Stack);
        requires |S| < Max_Depth;
        ensures S = <E> ∘ #S;

    Operation Copy_Top(replaces Top_Entry: Entry; restores S: Stack);
        requires |S| > 0 ;
        ensures ∃ Rst: Str(Entry) ϶ S = <Top_Entry> ∘ Rst;

    Operation Removing_Pop(updates S: Stack);
        requires |S| > 0 ;
        ensures ∃ Old_Top_Entry: Entry ϶ #S = <Old_Top_Entry> ∘ S;

    Operation Depth( restores S: Stack ): Integer;
        ensures Depth = (|S|);

    Operation Rem_Capacity( restores S: Stack ): Integer;
        ensures Rem_Capacity = (Max_Depth - |S|);

    Operation Clear(clears S: Stack);
end Copy_Based_Stack_Template ;
```

Figure 5: Specification of Copy-Based Stack_Template

Figure 6 contains the context and Figure 7 contains the code for a realization of Copy_Based_Stack_Template using Stack_Template.

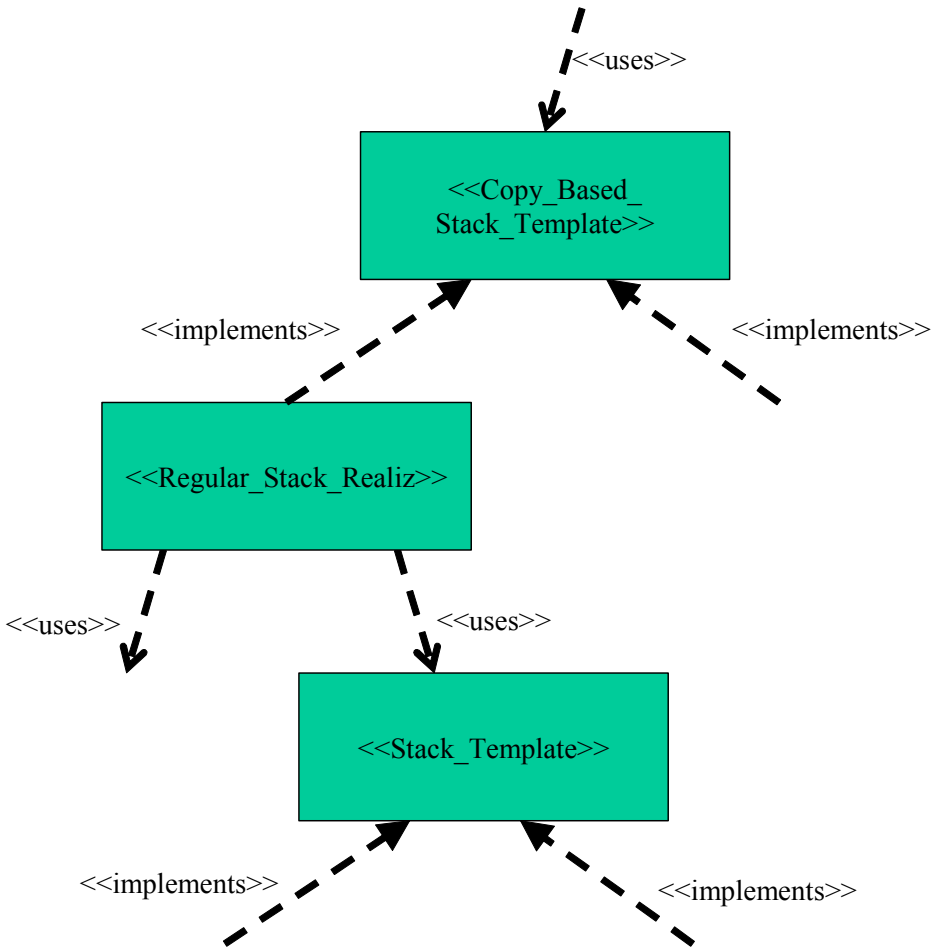

Figure 6: Context for Implementing One Component Using Another

Unlike Stack_Template, to develop a generic realization of Copy_Based_Stack_Template we need an operation to copy entries. Since the type Entry is supplied as a parameter from the outside at the instantiation time, the Copy_Entry operation must also be supplied as a parameter, because there is no automatic procedure to copy objects of arbitrary type. This realization parameter, given in parentheses following the realization name, specifies both the expected signature and behavior of Copy_Entry operation that should be supplied. Notice that the operation is to be supplied as a parameter to the realization not to the concept, because it is only necessary to write realization code. The specification of the concept uses only the equality predicate on Entry type (implicitly for the restores

mode parameters, for example) and requires no code for implementing the predicate. In general, different realizations of a generic concept may demand different Entry procedures to be supplied for instantiation.

In the realization in Figure 7, the code for the Stack procedures Copying_Push and Copy_Top make use of the Copy_Entry operation to restore their parametric entry and stack, respectively. When entries are non-trivial, Copy_Entry operation, and the procedures based on the operation are expensive. The rest of the procedures (including initialization and finalization procedures) call through to the corresponding regular stack procedures. In procedures Depth, Rem_Capacity, and Clear, we have qualified the calls to regular stack procedures with its facility declaration, to disambiguate that the calls are not recursive. It is important to compare this realization with the ones for Stack_Template in sections 1.1.1and 1.1.2 that make use of swapping for entry movement.

```
Realization Regular_Stack_Realiz (
        operation Copy_Entry (replaces Copy: Entry; restores Orig: Entry);
                ensures Copy = Orig;) for Copy_Based_Stack_Template;
        uses Stack_Template;
    Facility Stk_Fac is Stack_Template(Entry, Max_Depth)
        realized by Array_Realiz;

    Type Stack = Stk_Fac.Stack;
        correspondence Conc.S = S;

    Procedure Copying_Push (restores E: Entry; updates S: Stack);
        Var E_Copy: Entry;
        Copy_Entry(E_Copy, E);
        Push(E_Copy, S);
    end Copying_Push;

    Procedure Copy_Top(replaces Top_Entry: Entry; restores S: Stack);
        Var T_Copy: Entry;
        Pop(Top_Entry, S);
        Copy_Entry(T_Copy, Top_Entry);
        Push(T_Copy, S);
    end Copy_Top;

    Procedure Removing_Pop(updates S: Stack);
        Var Top_Entry: Entry;
        Pop(Top_Entry, S);
    end Removing_Pop;

    Procedure Depth( restores S: Stack ): Integer;
        Depth := Stk_Fac.Depth (S);
    end Depth;
    Procedure Rem_Capacity( restores S: Stack ): Integer;
        Rem_Capacity := Stk_Fac.Rem_Capacity (S);
    end Rem_Capacity;
    Procedure Clear(clears S: Stack);
        Stk_Fac.Clear (S);
    end Clear;
end Regular_Stack_Realiz;
```

Figure 7: A Realization of Copy-Based Stack_Template

A Second Example

The specification of Object_Oriented_Stack_Template was given in the previous chapter. Figure 4 contains a realization. The realization declares a facility of Stack_Template and uses a Stack from the facility to represent a single instance Stack variable. This is an atypical realization because a stack from one facility is used to represent another stack, but it does provide one more example of using one component in realizing another.

```
Realization Regular_Stack_Realiz for Object_Oriented_Stack_Template;
        uses Stack_Template;

    Facility Stk_Fac is Stack_Template(Entry, Max_Depth)
        realized by Array_Realiz;

    Var Loc_Stack: Stk_Fac.Stack;
    Facility Correspondence Self = Loc_Stack;

    Procedure OO_Push(alters E: Entry);
        Push(E, Loc_Stack);
    end;

    Procedure OO_Pop(replaces R: Entry);
        Pop(R, Loc_Stack);
    end;

    Procedure OO_Depth(): Integer;
        OO_Depth := Depth(Loc_Stack);
    end SI_Depth;

    Procedure OO_Rem_Capacity (preserves S: Stack): Integer;
        OO_Rem_Capacity := Rem_Capacity(Loc_Stack);
    end SI_Rem_Capacity;
```

```
    Procedure OO_Clear();
        Clear(Loc_Stack);
    end;
end Regular_Stack_Realiz;
```

Figure 8: A Realization of Object_Oriented_Stack_Template Using Stack_Template

The realization in Figure 8 provides no type, but contains a facility-level variable. When a facility of Object_Oriented_Stack_Template is instantiated, facility variable Loc_Stack is created. The value of Loc_Stack directly corresponds to the conceptual variable *Self* as stated in the correspondence clause. The unspecified facility convention defaults to true. In general, facility correspondence and convention clauses may relate multiple realization and conceptual variables. Facility initialization and finalization involve no code, because Loc_Stack is automatically initialized and finalized. The rest of the single instance stack procedures call through to the corresponding regular stack procedures, using Loc_Stack as an additional parameter.

The Regular Stack realization in Figure 8 makes use of the array realization for Stack_Template facility, though any other realization of Stack_Template would provide the same functionality. The performance of the Regular_Stack_Realiz however depends on the performance of the realization chosen for Stack_Template.

Exercises

1. Why should the mode of the Entry parameter to Push remain in the alters mode in the above realization? Would this mode change, if the Stack_Template facility is realized using Clean_Array_Realization Why or why not?
2. Rewrite the array realization in Figure 3 using the slightly different array representation given below. Your solution should include suitable correspondence and conventions assertions.

```
        Type Stack = Record
                        Contents: Array 0..(Max_Depth – 1) of Entry;
                        Size: Integer
    end;
```

3. A Clearing_Stack_Template is similar to Stack_Template, except that the parametric entry to Push is specified to be in **clears** mode. Realize

Clearing_Stack_Template directly using an array, with and without clean array convention.

4. Realize Clearing_Stack_Template using Stack_Template.
5. Realize Defensive_Stack_Template (given as exercise ? of Chapter 2) using Stack_Template.
6. Realize Stack_Template using Copy_Based_Stack_Template. Why is this not a good realization of Stack_Template?
7. Develop another realization of Stack_Template that uses the same convention as the Clean_Array_Realization, except that it uses a parametric Clear_Entry operation to clear entries in Pop and Clear procedures. Since an entry can be cleared more efficiently, in general, than initializing a new entry and swapping with it, this is a more efficient realization of Stack_Template than Clean_Array_Realization.

3.2 Enhancements to Components

A basic issue in designing a data abstraction component is identification of a suitable set of operators. Availability of a large number of convenient operators can make a data abstraction easy to use in a variety of applications. However, including every conceivably useful operation makes the tasks of specification and realization of data abstractions unreasonable. Inclusion of too many operations in a concept can make it unwieldy and incomprehensible, and discourage development of alternative implementations of the concept. A sensible compromise is to add a language mechanism that allows easy extension of the set of operations provided by a concept. This topic is the focus of this chapter.

Even with a mechanism to enhance concepts, the design question of whether a particular operator should be included in a data abstraction specification or relegated to be an extension remains. Answering this question demands a careful and pragmatic analysis of simplicity, utility, and performance, and it is rarely straightforward. While large concepts that fail to relegate operators that are clearly secondary to be extensions are poorly designed, minimization of concepts to the extent that all meaningful uses of the concept need extensions is not a good design either. Our working hypothesis is that for every reusable data abstraction, a fixed interface with a basis set of useful operations — not necessarily the perfect set — can be identified and realized. If and where this hypothesis fails to hold, custom-made solutions will continue to dominate computing.

3.2.1 Specification, Implementation, and Use of Enhancements

An enhancement is a reusable extension to a concept. To illustrate the issues in defining and realizing enhancements, we return to the stack example. Some potentially useful extensions to a stack concept are operations to copy a stack, copy the top of a stack, check whether a stack is empty, reverse a stack, or test equality of two stacks . These convenience operations add little to the basic stack functionality and are strictly *secondary*, because the effect of each can be realized using a combination of the primary stack operations. The enhancement mechanism makes it possible to specify and implement stack secondary operations as “add ons” to the stack concept. It frees stack implementers from being burdened with the extra operations, at the same time providing stack users the convenience of enhancing a stack facility with one or more additional operations. Figure 9 shows the specification of an enhancement to flip or reverse a stack. Figure 10 illustrates the

relationship between an enhancement specification, its implementation (shown in Figure 11) with the parent concept.

```
Enhancement Flipping_Capability for Stack_Template;
    Operation Flip(updates S: Stack );
        ensures S = (#S)^Rev;
end Flipping_Capability;
```

Figure 9: Specification of a Stack Enhancement

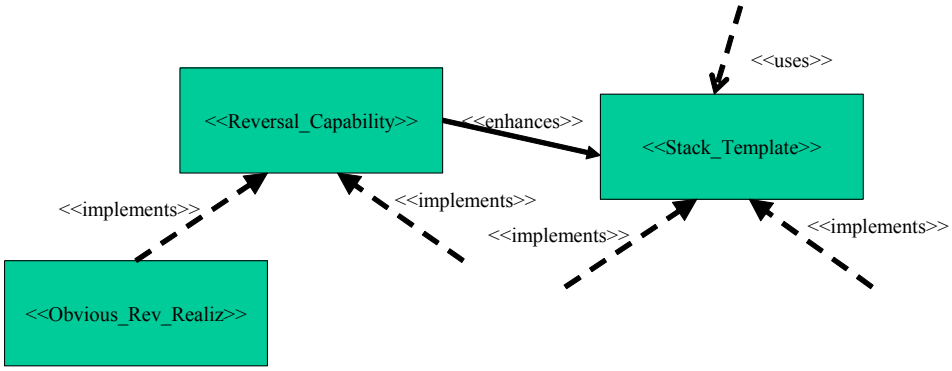

Figure 10: A Design-Time Relationship Diagram for an Enhancement

```
Realization Obvious_Flipping_Realiz for Flipping_Capability of Stack_Template;
    Procedure Flip(updates S: Stack );
        Var S_Flipped: Stack;
        Var Next_Entry: Entry;

        While (Depth(S) /= 0)
        do
            Pop(Next_Entry, S);
            Push(Next_Entry, S_Flipped);
        end;
        S :=: S_Flipped;
    end Flip;
end Obvious_Flipping_Realiz;
```

Figure 11: An Implementation of Stack Flipping Enhancement

The heading **Enhancement**…**for** signifies that Flipping_Capability is written in the context of Stack_Template, making it possible to use any of the information in the Stack concept in the specification or realization of the enhancement. Both the specification and realization of the enhancement are generic, i.e., they are independent of the type of entries in the stack. In the specification, S is of Stack type family from Stack_Template, modeled by a string of entries of the arbitrary parametric type Entry. In the ensures clause, we have used *Rev*, a mathematical definition of string reversal given in String_Theory.

In the realization heading, in general, it may be necessary to identify the concept for which the enhancement is written, in addition to the name of the enhancement. For the current example, the more elaborate heading would explicitly qualify the enhancement: *Stack_Template.Flipping_Capability*.
This elaboration may be needed, because an enhancement by the same name might be defined on multiple concepts. Where there is no potential for confusion, as in the present case, we omit the name of the concept for brevity.

The realization contains a procedure for Flip operation based on Stack primary operations. The straightforward realization code illustrates the use of the swap operator to transfer the contents of the local stack to the parametric stack at the end.

Reuse of Components with Enhancements

It is important to note that the enhancement specification and implementation depend only on Stack_Template concept. They can be used with any implementation of the concept. Figure 12 illustrates how the enhancement can be added to a stack facility declaration easily by a user. The facility declaration shows how Stack_Template realized by Clean_Array_Realiz can be enhanced with Flipping_Capability. The facility Reversable_Int_Stack_Fac provides Stack Flip operation in addition to the Stack type and primary operations and it can be used just like any other Stack primary operation. The same facility declaration may be used to extend a concept by one or more enhancements, as illustrated through exercises.

```
Facility Reversable_Int_Stack_Fac is Stack_Template(Integer, 90)
        realized by Clean_Array_Realiz
      enhanced by Flipping_Capability
        realized by Obvious_Flipping_Realiz;

Var S: Reversable_Int_Stack_Fac.Stack;

   Flip(S);
```

Figure 12: A Facility Declaration Using an Enhancement

Loop Design Assertions: Invariants and Progress Metrics

As in the case of data abstraction implementations, it is possible to capture the design rationale for a loop using two new kinds of assertions. Together these assertions state the intended behavior of a loop formally. They must be supplied by procedure writers. While the assertions make it easier to understand and modify loops, they are essential to ease manual reasoning and make mechanical reasoning feasible. The next chapter is devoted to explaining the use of the assertions in systematic reasoning. Figure 13

```
Realization Obvious_Flipping_Realiz for
                    Flipping_Capability of Stack_Template;
   Procedure Flip(updates S: Stack );
      Var S_Flipped: Stack;
      Var Next_Entry: Entry;

      While (Depth(S) /= 0)
         updating S, S_Flipped, Next_Entry;
         maintaining #S = (S_Flipped)^Rev ∘ S;
         decreasing |S|;
      do
         Pop(Next_Entry, S);
         Push(Next_Entry, S_Flipped);
      end;
      S :=: S_Flipped;
   end Flip;
end Obvious_Flipping_Realiz;
```

Figure 13: Stack Flipping Implementation with Assertions

The **maintaining** clause for a loop is an *invariant* among the loop variables and it describes essentially what makes the loop work. It must be true at the beginning and at the end of each iteration of the loop, including the first and last iterations. It is also necessary for the procedure writer to record the rationale for why a loop will terminate. This is done by specifying in the **decreasing** clause for a loop a progress metric that is a natural number which reduces on each pass through the loop. Since there are no infinite strictly decreasing sequences of natural numbers, this means that the loop must terminate. When suitable[15] maintaining and decreasing assertions are supplied for loops, it becomes possible to verify formally that a procedure is correct as detailed in Chapter #. The process of verification also involves checking that the programmer-supplied loop maintaining assertion is indeed an invariant and that the progress metric is actually decreasing.

In the present example, the maintaining clause for the loop in the Flip procedure records the facts that the initial contents of *S_Orig* is kept distributed between the *S_Orig* and *S_Flipped* Stacks and that the original stack is of length is always within Max_Depth. At the end of the loop, using this invariant assertion and the fact that

[15] For terminating loops, it may not always be possible to find a natural number as a progress metric. But it is always possible to find a decreasing ordinal number.

Stack S is empty, it is easy to conclude that $\#S = (S\_Flipped)^{Rev}$. Therefore, after the swap statement we have $\#S = (S)^{Rev}$ which is the ensures clause of the Flip procedure. The proof of correctness of a procedure involves showing not only that it satisfies its ensures clause at the end but also that no internal obligations are violated. In particular, it is essential to show that the requires clause of Push operation that is called in the loop is always satisfied. Proof that this is the case follows from the invariant and the Stack_Template constraint on the (maximum) depth of any every stack. In general, the loop invariants must be strong enough to show that the ultimate procedure obligation that it satisfies it ensures clause and internal procedure obligations, such as those arising from requires clauses of operations that are called.

3.2.2 Additional Principles of Enhancements

This section illustrates additional issues in developing and using enhancements using more examples.

A Second Example

Figure 14 contains the specification of an enhancement to copy a stack and Figure 15 contains an implementation.

```
Enhancement Copying_Capability for Stack_Template;
    Operation Copy_Stack(replaces S_Copy:Stack; restores S_Orig:Stack);
        ensures S_Copy = S_Orig;
end Copying_Capability;
```

Figure 14: Specification of Stack Copying_Capability

The interesting aspect of the Copy_Stack procedure is that it uses a Copy_Entry operation, in addition to Stack primary operations. This operation is needed to copy entries of a generic stack. Since the type Entry is supplied as a parameter from the outside at the stack facility instantiation time, the Copy_Entry operation must also be supplied as a parameter to the realization, because there is no automatic procedure to copy objects of arbitrary type. This realization parameter, given in parentheses following the realization name, specifies both the expected signature and behavior of Copy_Entry operation that should be supplied. Notice that the operation is to be supplied as a parameter to the realization not to the enhancement specification, because it is only necessary to write realization code. The specification of the enhancement uses only the equality predicate on Entry type, implicitly in the ensures clause when the stacks *S_Orig* and *S_Copy*, conceptualized as two strings of entries, are equated. This example illustrates the general need for the language feature to be able to pass operations on generic Entry type as parameters to realizations of concepts or enhancements to concepts. In general, different realizations of a generic concept or enhancement may demand different Entry procedures to be supplied by users, depending on how they need to manipulate the generic entries.

```
Realization Obvious_C_C_Realiz(
    operation Copy_Entry(replaces Copy: Entry; restores Orig: Entry);
            ensures Copy = Orig; )
                for Copying_Capability of Stack_Template;
    Procedure Copy_Stack(replaces S_Copy:Stack; restores S_Orig:Stack);
        Var Next_Entry, Entry_Copy: Entry;
        Var S_Flipped: Stack;

        While (Depth(S_Orig ) /= 0)
            updating S_Orig, Next_Entry, S_Flipped;
            maintaining #S_Orig = (S_Flipped)^Rev ∘ S_Orig;
            decreasing |S_Orig|;
        do
                Pop(Next_Entry, S_Orig);
                Push(Next_Entry, S_Flipped );
        end;
        Clear( S_Copy );
        While (Depth(S_Flipped ) /= 0)
            updating S_Copy, S_Orig, Next_Entry, Entry_Copy, S_Reversed;
            maintaining S_Copy = S_Orig and
                                #S_Orig = (S_Flipped)^Rev ∘ S_Orig;
            decreasing |S_Flipped|;
        do
                Pop(Next_Entry, S_Flipped);
                Copy_Entry(Entry_Copy, Next_Entry);
                Push(Next_Entry, S_Orig);
                Push(Entry_Copy, S_Copy);
        end;
    end Copy_Stack;
end Obvious_C_C_Realiz;
```

Figure 15: A Realization of Stack Copying_Capability

Figure 16 illustrates how to instantiate an Integer Stack facility enhanced with Copying_Capability. The instantiation uses a previously declared Copy_Integer operation as a parameter to the Stack Copying_Capability realization. The code for Copy_Integer procedure merely calls through to the Integer Replica operation from Std_Integer_Fac. Integer Replica, which is a function operation, cannot be passed directly as a parameter to the Copying_Capability realization, because its syntactic

form does not match the procedural requirement. The example instantiation in Figure 6 is illustrative of the general case where suitable local procedures needs to be declared and passed as parameters to realizations.

```
Facility Example_Fac;
    uses Std_Integer_Fac, Stack_Template;

    Operation Copy_Integer(replaces C: Integer; preserves Orig: Integer );
        ensures C = Orig;

    Procedure Copy_Integer(replaces C: Integer; preserves Orig: Integer );
        C := Replica(Orig);
    end Copy_Integer;

    Facility Copyable_Int_Stack_Fac is Stack_Template( Integer, 90 )
                realized by Clean_Array_Realiz
            enhanced by Copying_Capability
                realized by Obvious_C_C_Realiz(Copy_Integer);

    (* rest of the example facility omitted *)
end Example_Fac;
```

Figure 16: Instantiation of A Stack Facility with Copying_Capability

We conclude this subsection with a discussion of an alternative realization for Stack Copying_Capability, named Shorter_C_C_Realiz in Figure 17. In this realization, the code for Copy_Stack procedure is shortened using a call to the Flip_Stack operation. There is no performance advantage to be gained by this shortened code, however. We discuss the alternative only to illustrate how a new extension to a concept can be implemented using previously-developed enhancements. In the realization heading, we pick other Stack_Template enhancements as well as their realizations for use within the current realization. But this change in the heading has no impact on how this realization is used. To use the Shorter_C_C_Realiz for Copying_Capability in the Stack facility declaration in Figure 16, we merely need to replace Obvious_C_C_Realiz with the new realization. No mention of Reversing_Capability is needed, unless the client needs the Stack Flip operation.

```
Realization Shorter_C_C_Realiz(
    operation Copy_Entry(replaces Copy: Entry; restores Orig: Entry);
        ensures Copy = Orig)
                    for Copying_Capability of Stack_Template
                            enhanced by Flipping_Capability
                            realized by Obvious_Flipping_Realiz;

    Procedure Copy_Stack(replaces S_Copy:Stack; restores S_Orig:Stack);
        Var Next_Entry, Entry_Copy: Entry;
        Var S_Flipped: Stack;

        Flip (S_Orig);
        S_Flipped :=: S_Orig;
        Clear( S_Copy );
        While (Depth(S_Flipped) /= 0)
            updating S_Copy, S_Orig, Next_Entry, Entry_Copy, S_Flipped;
            maintaining S_Copy = S_Orig and
                #S_Orig = (S_Flipped)^Rev ∘ S_Orig;
            decreasing |S_Flipped|;
        do
                Pop(Next_Entry, S_Flipped);
                Copy_Entry(Entry_Copy, Next_Entry);
                Push(Next_Entry, S_Orig);
                Push(Entry_Copy, S_Copy);
        end;
    end Copy_Stack;
end Shorter_C_C_Realiz;
```

Figure 17: An Alternative Realization of Stack Copying_Capability

Custom Realization of Enhanced Concepts

Occasionally, it is possible to improve the performance of an enhancement operation significantly, i.e., by an order of magnitude, if it is implemented together with the primary operations on the concept in a single realization module. For example, it is possible to realize the stack concept along with Flip operation so that Push, Pop, and Flip all take only a constant time. Arguably there is little to be gained with an efficient Stack reversal operation that is unlikely to find much use. The main objective here is to illustrate how the syntax makes it possible to prepare and use such custom-made realizations. Custom realizations are expensive, and should be

developed only in extreme situations, where performance benefits and the enhanced functionality justify the costs.

A slightly more useful application of the idea is discussed in a later chapter where we show how to realize a communal version of the stack concept along with a Copy_Stack efficiently so that Push, Pop, and Copy_Stack are all constant time operations. Such a realization amortizes the cost of copying a stack over other stack operations. It involves use of a concept that captures pointers. The objective here is to illustrate how our syntax makes it possible to prepare and use such custom-made realizations.

```
Realization Custom_Stack_ Realiz
        for Stack_Template enhanced by Reversal_Capability;
    (* exercise: details of the realization body *)
end Custom_Stack_ Realiz;

Facility Reversable_Int_Stack_Fac is Stack_Template( Integer, 90 )
    enhanced by Reversal_Capability
    realized by Custom_Stack_Realiz;
```

Figure 18: Declaration and Use of a Custom-Made Realization for an Enhanced Concept

Implementation of Custom_Stack_Realiz that provides Flip operation in addition to Stack type and primary stack operations is left as an exercise.

Role of Enhancements

Though we have devoted a chapter to the topic of reusable enhancements, it is important to emphasize they are likely to be defined and used relatively infrequently in practice. This is because if the functionality provided by an enhancement to a concept is reused widely, then such functionality must be considered inclusion in the core concept even though it might be secondary. Enhancements and their realizations do serve as useful means in this book to illustrate and reinforce the general utility of a concept, rationalize the design of a concept, and to communicate other software engineering principles such as formal reasoning.

With the syntax for the enhancement mechanism given in this chapter, it is not possible to inherit from multiple concepts and define a new reusable extension. For

example, it is not possible to create an enhancement to provide an operation that copies a stack to a queue, because it involves referring to both Stack_Template and Queue_Template concepts. While it is debatable whether a software engineering language should include a mechanism to support this feature, it is clear that operations that couple independent concepts are unlikely to be used frequently. If operations involving multiple concepts are used routinely, then the cohesiveness of the concepts is questionable; a higher-level abstraction is possibly missing. If they are needed only rarely or in a particular application, then it is best to custom design the solution for the situation at hand.

Exercises

1. In reasoning about the loop in Figure 2.2, you need to make use of the following fact: $|\#S| \leq$ Max_Depth. Explain how you can deduce this to be a fact and why it is needed in the proof of correctness of the Flip procedure.
2. What will go wrong if the call to Copy_Entry operation is removed from the second loop in Copy_Stack procedure?
3. Explain why each conjunct in the maintaining clause for the second loop is necessary in arguing that Copy_Stack works.
4. Rewrite the facility declaration Example_Fac in Figure 6 using (regular) Array realization for Stack_Template and Shorter_C_C_Realiz for Stack Copying_Capability. How many changes are needed to make this realization switch? Why might you want to switch realizations, in general?
5. Write an enhancement for the Stack_Template that provides two operations: one to check if a stack is empty and another to check if it is full. Write a realization for your enhancement.
6. Write an Equality_Checking_Capability enhancement for the Stack_Template and a realization. The realization takes Are_Equal_Entries as a parameter and provides Are_Equal_Stacks operation.
7. Write a Clearing_Push_Capability enhancement for the Stack_Template and a realization. The enhancement provides an operation Clearing_Push that clears the entry pushed on the stack.

8. Write a Copying_Push_Capability enhancement for the Stack_Template and a realization. The enhancement provides operation Copying_Push that restores the entry to be pushed on a stack.
9. Declare a stack of stacks of integers facility enhanced with Copying_Capability.
10. Declare an Integer_Stacking_Facility enhanced with Reversal_Capability, Equality_Checking_Capability, and Copying_Capability.

11. Write a Reversal_Prefix_Capability enhancement for the Stack_Template and a realization. The Prefix_Reversal operation has the specification:
    Operation Prefix_Reversal(**clears** Pre: Stack; **updates** S: Stack);
        requires |Pre| + |S| ≤ Max_Depth;
        ensures S = #Pre^{Rev} ∘ #S;
12. Write a Deleting_Capability enhancement for the Stack_Template and a realization. The Delete_Entries operation has the specification:
    Operation Delete_Entries(**restores** k: Integer; **updates** S: Stack);
        requires 0 ≤ k ≤ |S|;
        ensures Is_Suffix(S, #S) **and** |S| + k = |#S|;
13. Write a Presence_Checking_Capability enhancement for the Stack_Template and a realization. The Is_ Present operation has the specification:
    Oper Is_ Present(**restores** E: Entry; **restores** S: Stack): Boolean;
        ensures Is_ Present = (∃ α, β : Str(Entry) ∍ S = α ∘ <E> ∘ β);
14. Write a custom realization for Stack_Template enhanced by Flipping_Capability so that all operations including Flip (but excluding initialization and finalization) take only a constant time.

15. Write a realization for Random_Access_Capability.

```
Enhancement Random_Access_Capability for Stack_Template;
    Operation Remove (replaces E: Entry; restores k: Integer;
                                        updates S: Stack);
        requires 1 ≤ k ≤ |S|;
        ensures  ∃ α, β : Str(Entry) ϶  #S = α ∘ <E> ∘ β  and
                        |α| + 1 = k and S = α ∘ β;
    Operation Insert (alters E: Entry; restores k: Integer;
                                        updates S: Stack);
        requires 1 ≤ k ≤ |S| + 1;
        ensures  ∃ α, β : Str(Entry) ϶  #S = α ∘ β  and
                        |α| + 1 = k and S =  α ∘ <#E> ∘ β;
end Random_Access_Capability;
```

16. Refer to the Mystery_Capability enhancement and its realization (given on the next page). If the type Entry is Integer and #S = <4, 13, 5, 9, 8>, #T = <2, 6>, and n = 3, then what would the values of S and T be if and when Mystery_Op terminates? Using your answer as a guide, complete the assertions in the specification and realization.

```
Enhancement Mystery_Capability for Stack_Template;
    Oper Mystery_Op (updates S, T: Stack; restores n: Integer );
        requires
        ensures
end Mystery_Capability;
Realization Undocumented_Realiz for Mystery_Capability;
    Procedure Mystery_Op(?? S, T: Stack; ?? n: Integer );
        Var E: Entry; Var i: Integer;

        Clear(T); i  := 0;
        While i  ≠ n
            maintaining ??
            decreasing ??
        do
                Pop(E, S); Push(E, T); i := i +1;
        end;
    end Mystery_Op;
end Undocumented_Realiz;
```

3.3 A Second Component Example

The objective of this section is to illustrate the generality of the principles motivated in the first part of the book using a second component specification, design, implementations, and enhancements. Ideally, a reader would treat the contents of this section as useful exercises.

3.3.1 Interface Design and Specification

Figure 19 contains a generic queue concept that permits insertion and removal of entries in a first in first out order. Mathematical strings are also suitable for conceptualizing queues and providing specifications of Queue operations as illustrated in the figure. In general, we will reuse standard mathematical ideas such as strings in the description of multiple concepts to decrease the overhead of formal specification and to increase comprehensibility.

An interesting aspect of the design here is the inclusion of an operation Swap_First_Entry to access the front entry of a queue. Efficiency considerations preclude the inclusion of a more common operation, such as Get_Front that leads to copying the front entry. The first entry can be looked at and returned, using two calls to the Swap_First_Entry operation. This feature is useful in such situations where a single look ahead is needed. While Swap_First_Entry can be implemented as an enhancement using other Queue primary operations, it works in constant time when implemented directly as a primary operation (see exercise).

```
Concept Queue_Template(type Entry; evaluates Max_Length: Integer);
        uses Std_Integer_Fac, String_Theory;
                requires Max_Length > 0;
    Type Family Queue ⊆ Str(Entry);
        exemplar Q;
        constraint |Q| ≤ Max_Length;
        initialization ensures Q = Λ;

    Operation Enqueue(alters E: Entry; updates Q: Queue);
        requires |Q| < Max_Length;
        ensures Q = #Q ∘ <#E>;

    Operation Dequeue(replaces R: Entry; updates Q: Queue);
        requires |Q| > 0;
        ensures #Q = <R> ∘ Q;

    Operation Swap_First_Entry(updates E: Entry; updates Q: Queue);
        requires |Q| > 0 ;
        ensures  ∃ Rem: Str(Entry) ϶ #Q = <E> ∘ Rem and Q = <#E> ∘ Rem;

    Operation Length(restores Q: Queue): Integer;
        ensures Length = (|Q|);

    Operation Rem_Capacity(restores Q: Queue): Integer;
        ensures Rem_Capacity = (Max_Length - |Q|);

    Operation Clear(clears Q: Queue );
end Queue_Template;
```

Figure 19: A Basic Queue_Template Concept

3.3.2 An Efficient Implementation

Figure 20 contains an implementation of Queue_Template in which all procedures, except initialization and finalization, take only a constant time.

```
Realization Circular_Array_Realiz for Queue_Template;
        uses Record_Template, Static_Array_Template;
    Type Queue = Record
                        Contents: Array 0..Max_Length - 1 of Entry;
                        Front, Length: Integer;
                 end;
        conventions 0  <= Q.Front < Max_Length and
                              0  <= Q.Length <= Max_Length;
        correspondence
            Conc.Q = ∏ (i=Q.Front .. Q.Front + Q.Length - 1) < Q.Contents(i mod Max_Length) > ;
    Procedure Enqueue(alters E: Entry; updates Q: Queue);
        Q.Contents(Q.Front + Q.Length mod Max_Length) :=: E;
        Q.Length := Q.Length + 1;
    end Enqueue;

    Procedure Dequeue(replaces R: Entry; updates Q: Queue);
        Q.Contents(Q.Front) :=: R;
        Q.Front := (Q.Front + 1) mod Max_Length;
        Q.Length := Q.Length - 1;
    end Dequeue;

    Procedure Swap_First_Entry(updates E: Entry; updates Q: Queue);
        Q.Contents(Q.Front) :=: E;
    end Swap_First_Entry;

    Procedure Length(restores Q: Queue): Integer;
        Length := Q.Length;
    end Length;
    Procedure Rem_Capacity(restores Q: Queue): Integer;
        Rem_Capacity := Max_Length – Q.Length;
    end Rem_Capacity;
    Procedure Clear(clears Q: Queue );
        Q.Front := 0; Q.Length := 0;
    end Clear;
end Circular_Array_Realiz;
```

Figure 20: An Implementation of Queue_Template

3.3.3 Design Variations

Literature contains several variations of the Queue concept that contain other primary operations in addition to those in Queue_Template. Almost all these variations violate the Queue metaphor to some extent.

Another variation of Queue concept allows entries to be added to the front or the back of the queue, but permits removals only from the front. This concept, termed, provides the following two primary operations in addition to those in Queue_Template. The complete specification for the concept is given in Figure 7.

Operation Inject(**alters** E: Entry; **updates** Q: Preemptable_Queue);
requires |Q| < Max_Length ;
ensures Q = <#E> ∘ Q;

Operation Swap_Last_Entry(**updates** E: Entry;
updates Q: Preemptable_Queue);
requires |Q| > 0 ;
ensures ∃ Pre: **Str**(Entry) ϶ #Q = Pre ∘ <E> **and** Q = Pre ∘ <#E>;

A most general, and possibly least useful (why?) version of Queue concept allows entries to be added and removed both to the front and the back of the queue. This concept, termed, Deque_Template, provides the following two primary operations in addition to those in Queue_Template:

Operation Inject(**alters** E: Entry; **updates** Q: Deque);
requires |Q| < Max_Length ;
ensures Q = <#E> ∘ Q;

Operation Eject(**replaces** R: Entry; **updates** Q: Deque);
requires |Q| > 0 ;
ensures #Q = Q ∘ <R>;

Exercises

1. The principle of maximum regularity tells us that a **requires** clause should not be needlessly restrictive. How does this apply to using
   A. "|Q| > 0" as the requires clause for Clear?
   B. "|Q| > 0" for Dequeue?
   C. "|Q| < Max_Length" for Enqueue?

2. Write an Appending_Capability enhancement for Queue_Template and a realization:
   Operation Append_Q(**clears** P: Queue; **updates** Q: Queue);
   requires |P| + |Q| ≤ Max_Length;
   ensures P = #P ∘ #Q;

3. Write a Reversal_Capability enhancement for Queue_Template and two realizations: one using a local stack and iteration and another using recursion. For the recursive procedure, you need to provide a decreasing metric.
4. Write an enhancement for the Queue_Template that provides two operations: one to check if a queue is empty and another to check if it is full. Write a realization for your enhancement.
5. Write an Equality_Checking_Capability enhancement for the Queue_Template and a realization. The realization takes Are_Equal_Entries as a parameter and provides Are_Equal_Queues operation.
6. Declare a Stack of Queues of Reals facility of max depth 20 enhanced with Reversal and Equality_Checking capabilities.
7. Write a Replication_Capability enhancement for Queue_Template and its realization to provide the following operation:
   Operation Replica(**restores** Q: Queue): Queue;
   ensures Replica = (Q);

8. Using the Replication_Capability enhancement, programmers could revert to traditional assignment-statement-based programming with queues. What problems could this engender?
9. Write an alternative realization of Merge_Capability in which you do not swap the two queues Q and R, instead using a boolean flag to determine which of the two queues should be dequeued next. Your answer should include proper internal assertions.
10. Develop a Presence_Checking_Capability enhancement (and a realization) for Queue_Template. Does your specification of the operation restore the queue?

11. Write a realization for the Deleting_Capability enhancement, which provides the operation:

    Oper Delete_Entries(**restores** k: Integer; **updates** Q : Queue);
        requires $0 \le k \le |Q|$;
        ensures Is_Suffix(Q, #Q) **and** $|Q| + k = |\#Q|$;

12. Realize the enhancement:

    Enhancement Rotating_Capability **for** Queue_Template;
        Operation Demote_First_Entry(**updates** Q: Queue);
            requires $|Q| > 0$;
            ensures $\exists E$: Entry, $\exists \alpha$: Str(Entry) ϶ #Q = <E> ∘ α **and** Q = α ∘ <E>;
        Operation Rotate(**restores** k: Integer; **updates** Q: Queue);
            requires $|Q| > 0$ **and** $k >= 0$;
            ensures $\exists \alpha, \beta$: Str(Entry) ϶ #Q = α ∘ β **and**
                $\exists n$: N ϶ n*|#Q| + $|\alpha|$ = k **and** Q = β ∘ α;
    end Rotating_Capability;

13. Front_Preemptable_Queue_Template differs from Queue_Template in that it includes Swap_First_Entry as a primary operation. Show why it is reasonable to make Swap_First Entry a primary operation on Queues but not on stacks as a corollary of the following three sub-questions.
    A. Specify and realize a Swap_First_Entry_Capability enhancement for Stack_Template. Does your procedure take only a constant-time?
    B. Realize Front_Preemptable_Queue_Template so that all operations, except initialization and finalization, take a constant time.
    C. Assuming that Swap_First_Entry were not included as a primary operation on queues, show how it too could be realized as a secondary operation. Does your procedure take only a constant-time?

14. Write constant-time procedures for all Preemptable_Queue_Template operations, except initialization and finalization. Why is Swap_First_Entry not an operation in Preemptable_Queue_Template?
15. Explain why you might want to use Stack_Template or Queue_Template, though Preemptable_Queue_Template provides the functionality of both.
    A. Realize Stack_Template using a Queue_Template, by completing the following realization.
    B. Compare the performance of the Queue_Based realization with that of the Array realizations of Stack_Template.

```
Realization Queue_Based_Realiz for Stack_Template;
    uses Queue_Template;

    Facility Q_Fac is Queue_Template( Entry, Max_Depth )
        realized_by Circular_Array_Realiz;
    enhanced by Appending_Capability
        realized by Obvious_A_C_Realiz;

    Type Stack = Q_Fac.Queue;
        correspondence  conc.S = S;
    (* Complete code for procedures *)
end Queue_Based_Realiz;
```

16. Realize the Queue_Template using Stack_Template.
17. Why is it not necessary to include Swap_First_Entry or Swap_Last_Entry operation in Deque_Template?
18. Since Deque_Template provides the functionality of Stack_Template, all variations of Queue_Template equally efficiently, explain why is it not sufficient for a component library to contain only Deque_Template.
19. The Mystery_Capabilities enhancement provides the operations whose realizations are given below. Complete the requirement and guarantee clauses in the enhancement and internal assertions in the realization.

```
Enhancement Mystery_Capabilities for Queue_Template;
    Operation Mystery_Op1(?? Q: Queue; ?? E: Entry );
        requires
        ensures
    Operation Mystery_Op2(?? Q, QB: Queue );
        requires
        ensures
end Mystery_Capabilities;
```

```
Realization Underdocumented_Realiz for Mystery_Capability;
    Procedure Mystery_Op1(?? Q: Queue; ?? E: Entry );
            Var T: Queue;
            Dequeue(E, Q);
            While Length(Q) > 0
                    maintaining
                    decreasing
            do
                    Enqueue(E, T);
                    Dequeue(E, Q);
            end;
                    Q :=:  T;
    end Mystery_Op1;

    Procedure Mystery_Op2(?? Q, QB: Queue );
            Var QF: Queue;
            Var E: Entry;
            Var Length_Diff: Integer;

            Clear( QB );
            Length_Diff  := 0;
            While Length(Q) > 0
                    maintaining
                    decreasing
            do
                    Dequeue(E, Q);
                    Enqueue(E, QB);
                            Length_Diff  := Length_Diff + 1;
                            If Length_Diff = 2 then
                                    Dequeue( E, QB);
                                    Enqueue( E, QF);
                                    Length_Diff  := 0;
                            end;
                    end;
                    Q :=: QF
            end Mystery_Op2;
    end Underdocumented_Realiz;
```

4 Analysis of Implementation Correctness

4.1 Testing and Formal Inspection

One of the key advantages of writing specification is that it becomes possible to check whether a given component implementation meets its specification. This is the topic of this chapter. The first section concentrates on testing and inspection.

Testing is one approach for detecting "bugs" in an implementation. The idea here is to execute an implementation on selected, valid inputs and check if the outputs correspond to those given in its specification. Since the input domain of most non-trivial operations is large, when using testing, checking is possible on only a small subset of values from the input domain. If the implementation fails to produce expected outputs, then the implementation does not meet its specification, i.e., it has bugs. However, if no errors are found on sample inputs, we cannot come to any conclusion about the correctness of the implementation. We can only conclude that the implementation "works" on the sample inputs. Choosing a suitable sample is key to effective testing. If the sample is representative and covers different aspects of the specification and the implementation behaves as expected on all those inputs, then there is more likelihood that the implementation will also work as specified after it is deployed.

Successful testing can only reveal the presence of errors, but not their absence. Formal verification is an alternative to testing, and it has the dual objective of determining whether an implementation is correct with respect to its specification. Unlike testing, it is a static approach and it involves no execution of the implementation. The goal here is to prove mathematically implementation correctness on all valid inputs. Correctness proofs are conclusive. No probability theory is involved. In general, verification is more expensive than testing. To minimize and amortize the cost of verification, ideally, an implementation should need to be verified only once in its lifetime.

There are other approaches that combine the principles of testing and verification. Tracing (or formal inspection) is one of them and it is discussed in this section. In tracing, an implementation is analyzed statically on a sample set of valid inputs. As in the case of testing, implementation errors may or may not be revealed on those inputs. If no errors are revealed during tracing and the sample set is representative, then there is a higher

likelihood that the implementation meets its specification. However, tracing involves no execution of the implementation. In addition to input/output comparison, tracing also helps check the logical correctness of the implementation on selected inputs.

4.1.1 Specification-Based Testing

We begin with an example to explain the basics of specification-based testing. Figure 38 contains the specification of a Swap_Top_Capability for Stack_Template. Figure 39 contains an implementation. The objective is to check whether or not the implementation meets the specification.

```
Enhancement Swap_Top_Capability for Stack_Template;
        Operation Swap_Top (updates S: Stack; updates E: Entry);
                requires |S| > 0;
                ensures ∃ rst: Str(Entry) ϶ #S = <E> ∘ rst ∧ S = <#E> ∘ rst;
end Swap_Top_Capability;
```

Figure 38: An Example Specification

```
Realization Obvious_Realiz for Swap_Top_Capability;
        Procedure Swap_Top (updates S: Stack; updates E: Entry);
                Var Top: Entry;

                Pop(Top, S);
                Push(E, S);
                E :=: Top;
        end Swap_Top;
end Obvious_Realiz;
```

Figure 39: An Implementation of Swap_Top Specification

Test Plans

In testing an implementation, first we need to identify a suitable set of valid inputs. We can identify these inputs either by looking only at the specification or by also considering the implementation. The first approach is specification-based and it is commonly called “black box” testing. Equivalence partitioning of input and output domains to cover all interesting behaviors, and boundary value testing are common black box testing

techniques. In black box testing, the test inputs can be identified even before an implementation has been developed.

When implementation details are used to identify test inputs, it becomes "white box testing." Here, test inputs are chosen to cover different control flows and data flows in the implementation. A good testing strategy should employ both approaches to identify test inputs, if the implementation source code is available at the time of testing. Exercises explore the merits of black box and white box testing, as well as classical techniques for identifying test inputs.

Once an adequate test set has been chosen, inputs should be checked to be valid, i.e., that they meet the pre-conditions of the module or operation under testing. A test plan contains expected output(s) for each test input based on the post-conditions. Before a generic module such as Swap_Top_Capability can be tested, it needs to be instantiated with particular values for module parameters. A generic module should be tested for multiple instantiations. Assuming, Swap_Top_Capability is used to enhance a facility of Stack_Template with Integer as Entry and 10 as Max_Depth, a sample test plan for testing Swap_Top operation is shown in Table 1.

	Valid Inputs	**Expected Outputs**
1	S = <3, 7, 5>, E = 6	S = <6, 7, 5>, E = 3
2	S = <8>, E = 6	S = <6>, E = 8
3	S = <4, 3, 7, 4, 5, 9, 8, 6, 3, 3>, E = 8	S = <8, 3, 7, 4, 5, 9, 8, 6, 3, 3>, E = 4

Table 1: A Test Plan for Swap_Top Enhancement

Notice that there is no need to test Swap_Top on an empty stack, because it does not satisfy the precondition and thus is not a valid input. Once a test plan has been formulated, the implementation is executed using a suitable driver program on each input. On each input point, the actual outputs are compared with expected outputs. If there is a mismatch, then it indicates an implementation error, provided the expected outputs are properly derived.

4.1.2 Formal Inspection Using Tracing

The objective of tracing Swap_Top procedure on the first input point is shown in Table 1. At the beginning of the procedure, we *assume* that the parameters have valid input values specified in the test plan. The objective of tracing is to *confirm* that expected output values result in the parameters at the end of the procedure. Here, our main objective is to check that S and E have values <6, 7, 5> and 3, respectively at the end of the procedure when the procedure is initiated with S = <3, 7, 5> and E = 6. While the end result is clearly important, during tracing we would also like to be sure that nothing goes wrong in the middle. In particular, when Swap_Top calls other operations, such as Pop and Push, we want to confirm that their preconditions are not violated. This additional checking is done in the states before calls to operations Pop and Push in the table.

State	**Statement**	**Assume**			**Confirm**
		S	E	Top	
0		<3, 7, 5>	6		
	Var Top: Entry;				
1					$\|S\| > 0$
	Pop(Top, S);				
2					$\|S\| < 10$
	Push(E, S);				
3					
	E :=: Top;				
4					$S = <6, 7, 5> \wedge E = 3$

Table 2: Example Objectives of Tracing

Given the objectives of tracing for a particular input point, then we trace the values of participating variables at each state of the procedure. To do this, we *assume* that each operation that is called by the procedure works as specified. We get the values of variables after a call to an operation from the post conditions of that operation. In Table 3, Top gets values 0 in state 1 from Entry (Integer, in this case) initialization. The values in states 2 and 3 are derived from the post conditions of Pop and Push, respectively. In state 3, because Push alters the Entry E to an unspecified value, E's value is indicated by "?".

Notice the assumption that values of variables that are not involved in an operation call are unchanged (e.g., the value of E after the call to Pop (Top, S)). This assumption is true for all operation calls in RESOLVE, but it does not always hold in other languages, making tracing of code in those languages significantly more complex than in RESOLVE.

State	**Statement**	**Assume**			**Confirm**
		S	E	Top	
0		<3, 7, 5>	6		
	Var Top: Entry;				
1		<3, 7, 5>	6	0	$\|S\| > 0$
	Pop(Top, S);				
2		<7, 5>	6	3	$\|S\| < 10$
	Push(E, S);				
3		<6, 7, 5>	?	3	
	E :=: Top;				
4		<6, 7, 5>	3	?	$S = <6, 7, 5> \wedge E = 3$

Table 3: A Tracing Table for Swap_Top Procedure

Proof in state 1

Confirm $|S| > 0$;
Confirm $|<3, 7, 5>| > 0$; (* from assumption in state 1 *)
True. (* from string theory *)

Proof in state 2

Confirm $|S| < 10$;
Confirm $|<7, 5>| < 10$; (* from assumption in state 2 *)
True. (* from string theory *)

Proof in state 4:

Confirm $S = <6, 7, 5> \wedge E = 3$;
True. (* from assumption in state 4 *)

Figure 40: Proof of Assertions to Be Confirmed in Tracing Table 3

Once the tracing table is complete, we attempt to confirm the assertions in each of states 1, 2, and 4, using the assumptions at that state as shown in Figure 40. If any one assertion in the last column cannot be confirmed, then

it indicates an implementation error, assuming there are no errors in the tabulation itself.

In tracing a procedure with conditional statements on an input point, only those paths that satisfy the conditions need to be traced. In tracing a procedure with a loop (or recursion), the loop (or recursion) needs to be unrolled as many times as necessary. As an alternative to unrolling, we can use the specification of the recursive procedure that is called or the invariant of the loop (given in the maintaining clause) to discover errors. Exercises consider tracing of non-trivial code.

4.2 Modular Verification Using a Tabular Approach

Unlike testing or tracing, formal verification provides conclusive proof that an implementation is correct with respect to its specification on *all* valid inputs.

4.2.1 Introduction to Modular Verification

The goals of formal verification for procedure Swap_Top are shown in Table 4. In the table, we denote the value of a variable at a state with the variable's name subscripted with the number of the state. For example, S0 denotes the value of Stack variable S at state 0 and S4 denotes its value at state 4.

In state 0, the precondition of Swap_Top operation becomes an assumption. The second conjunction in the assumption is a result of assuming the constraints given in Stack_Template specification. The constraints are true for every stack variable, including any one that is passed to Swap_Top. In state 4, the post condition of Swap_Top operation needs to be confirmed. In this assertion #S is replaced with S0, the value of S in the initial state of the procedure. In addition to confirming the post condition of Swap_Top, we also need to confirm that all calls made by Swap_Top are valid. In particular, in state 1, the state immediately before the call to Pop, we need to confirm the precondition of Pop. Similarly, the precondition of Push is to be confirmed in state 2.

State	Statement	Assume	Confirm
0		$\|S0\| > 0 \wedge$ $\|S0\| \leq$ Max_Depth	
	Var Top: Entry;		
1			$\|S1\| > 0$
	Pop(Top, S);		
2			$\|S2\| <$ Max_Depth
	Push(E, S);		
3			
	E :=: Top;		
4			$\exists$ rst: Str(Entry) ϶ S0 = <E4> ∘ rst ∧ S4 = <E0> ∘ rst

Table 4: Objectives of Swap_Top Procedure Verification

Table 5 additionally contains assumptions based on calls to external operation calls. The post condition of every operation that is called becomes an assumption with appropriate symbol substitutions in the state after the call. Also, variables that are unchanged by a call are assumed to retain their values. Once the table is done (mechanically or manually, but using only symbolic substitutions of pre- and post-conditions of operations), each assertion in the confirm column needs be verified using the assumptions in states preceding the assertion. The proofs of all confirm assertions are shown immediately following Table 5. We can conclude a procedure is *correct* with respect to its specification if all assertions in the confirm clause are formally discharged.

State	Statement	Assume	Confirm
0		$\|S0\| > 0 \wedge$ $\|S0\| \leq$ Max_Depth	
	Var Top: Entry;		
1		Entry.**Is_Initial**(Top1) $\wedge$ S1 = S0 $\wedge$ E1 = E0	$\|S1\| > 0$
	Pop(Top, S);		
2		S1 = <Top2> $\circ$ S2 $\wedge$ E2 = E1	$\|S2\| <$ Max_Depth
	Push(E, S);		
3		S3 = <E2> $\circ$ S2 $\wedge$ Top3 = Top2	
	E :=: Top;		
4		S4 = S3 $\wedge$ E4 = Top3 $\wedge$ Top4 = E3	$\exists$ rst: Str(Entry) $\ni$ S0 = <E4> $\circ$ rst $\wedge$ S4 = <E0> $\circ$ rst

Table 5: Verification Table for Swap_Top Procedure

Proof in state 1

Confirm $|S1| > 0$;
Confirm $|S0| > 0$; (* from assumption in state 1 *)
True. (* from assumption in state 0 *)

Proof in state 2

Confirm $|S2| <$ Max_Depth;
Confirm $|S2| + 1 \leq$ Max_Depth; (* from number theory *)
Confirm $|S1| \leq$ Max_Depth; (* from string theory and assumption in state 2 *)
Confirm $|S0| \leq$ Max_Depth; (* from assumption in state 1 *)
True. (* from assumption in state 0 *)

Proof in state 4:

```
Confirm ∃ rst: Str(Entry) ϶ S0 = <E4> ∘ rst ∧ S4 = <E0> ∘ rst;
Confirm ∃ rst: Str(Entry) ϶ S0 = <Top3> ∘ rst ∧
            S3 = <E0> ∘ rst;          (* from assumption in state 4 *)
Confirm ∃ rst: Str(Entry) ϶ S0 = <Top2> ∘ rst ∧
            <E2> ∘ S2 = <E0> ∘ rst;   (* from assumption in state 3 *)
Assume rst = S2;
Confirm S0 = <Top2> ∘ S2 ∧
      <E2> ∘ S2 = <E0> ∘ S2;          (* from assumption rst = S2 *)
Confirm S0 = S1 ∧
      <E1> ∘ S2 = <E0> ∘ S2;          (* from assumption in state 2 *)
Confirm S0 = S0 ∧
      <E0> ∘ S2 = <E0> ∘ S2;          (* from assumption in state 1 *)
True.                                 (* trivially true *)
```

Figure 41: Proof of Correctness for Swap_Top Procedure

While proofs of procedures, such as Swap_Top, do not need any additional assertions other than pre- and post-conditions of operations, recursive procedures and procedures with loops need additional assertions that must be supplied by programmers.

4.2.2 Verification of Recursive Procedures

To illustrate verification of a recursive implementation of a procedure, we consider an operation to append two queues. The specification of Queue_Template is given in the next chapter. Figure 42 contains a specification for the operation.

```
Enhancement Append_Capability for Queue_Template;
      Operation Append (updates P: Queue; clears  Q: Queue);
            requires    |P| + |Q| <= Max_Length;
            ensures     P = #Q ∘ #Q;
end Append_Capability;
```

Figure 42: Specification of a Queue Enhancement

Figure 43 contains a recursive implementation. Along with the implementation, a programmer needs to supply a *decreasing* metric for establishing termination, |Q| in the present case. The use of this metric (typically a natural number and an ordinal number, in the general case) will become apparent in the proof of correctness.

```
Realization Recursive_Realiz for Append_Capability of Queue_Template;
        Recursive procedure Append (updates P: Queue; clears  Q: Queue);
                decreasing |Q|;

                Var E: Entry;
                If Length (Q) > 0 then
                        Dequeue(E, Q);
                        Enqueue(E, P);
                        Append(P, Q);
                end;
        end Append;
end Recursive_Realiz;
```

Figure 43: A Recursive Implementation

Inductive Reasoning

Reasoning of implementations that involve recursion (or iteration using a loop) requires an understanding and application of the principle of *induction*. To show Append works, we do induction on the length of Q, i.e., |Q| the metric given in the decreasing assertion. The base case is when |Q| = 0. The first part of induction is showing that Append works on the base case. The inductive case covers the situation when |Q| > 0. To show Append works for the inductive case, we will hypothesize (or assume) that the recursive call to Append works correctly, as long as the length of parameter Q passed to the recursive call is strictly smaller than the length of Q that came in. Additionally, we need to confirm that this is indeed the case before the (recursive) call to Append.

To understand this principle, let us consider an example, where Queue_Template is assumed to be instantiated with Integers. In the base case, when Append is called with P = <3, 7> and Q = Λ, we need to confirm that the code works as specified. Now consider the case when Append is called with P = <3, 7> and Q = <4, 2, 6>. For this inductive case, we

assume that Append works correctly when called with P = <3, 7> and Q = <4, 2>. Using this assumption, we then confirm that Append will also work for the case P = <3, 7> and Q = <4, 2, 6>. Proceeding in this way, finally, we have: if Append works correctly when called with P = <3, 7> and Q = Λ, then it will also work when called with P = <3, 7> and Q = <4>. But we have separately established that Append works for the base case, i.e., when called with P = <3, 7> and Q = Λ. Therefore, Append will work correctly when called with P = <3, 7> and Q = <4>; therefore it will work for P = <3, 7> and Q = <4, 2>; and therefore, for P = <3, 7> and Q = <4, 2, 6>. Of course, we also have to establish that Append terminates, i.e., |Q| decreases in each recursive call.

We are now ready to formalize the argument. Table 6 lists the assumptions and confirmation assertions (or obligations) at each state. In verification involving conditional statements such as if-then-else statements or loops, the assumptions and obligations arise only under certain conditions and hence, they should be viewed as implications. For example, in the table the assumptions and obligations in states 2 through 5 are meaningful only when |Q1| > 0. Instead of writing down each of these assertions explicitly as implications following |Q1| > 0, we have listed the conditions separately under the column, titled, *path conditions*. For example, in state 2, the assumption is $|Q1| > 0 \Rightarrow (P2 = P1 \wedge Q2 = Q1 \wedge E2 = E1)$ and the obligation is $|Q1| > 0 \Rightarrow (|Q2| > 0)$. Assumptions and obligations without a path condition are true at all times. In the ensuing proofs in Figure 44, we make use of appropriate assumptions and obligations, based on the implications.

Proof of Termination

In state 4, we have a termination obligation based on the decreasing assertion and it is |Q4| < |Q0|. This obligation is in addition to the precondition of the (recursive) call to Append that is called immediately following this state. In state 5, we make the inductive assumption that the recursive call to Append works as specified. The assumptions in state 6 depend on the condition of the "if" statement. If |Q1| > 0 then the values of P, Q, and E are the same as their values in state 5; otherwise, their values are the same as they were in state 1. The assumptions in state 6 are listed separately for each of these two cases. The obligations in state 6 need to be proved for both cases.

State	Stmt	Path Condition	Assume	Confirm
0			$\lvert P0 \rvert + \lvert Q0 \rvert \leq$ Max_Length $\wedge$ $\lvert P0 \rvert \leq$ Max_Length $\wedge$ $\lvert Q0 \rvert \leq$ Max_Length	
	Var E: Entry;			
1			P1 = P0 $\wedge$ Q1 = Q0$\wedge$ Entry.**Is_Initial**(E1)	
	If Length (Q) > 0 **then**			
2		$\lvert Q1 \rvert > 0$	P2 = P1 $\wedge$ Q2 = Q1 $\wedge$ E2 = E1	$\lvert Q2 \rvert > 0$
	Dequeue(E, Q);			
3		$\lvert Q1 \rvert > 0$	P3 = P2 $\wedge$ Q2 = <E3> $\circ$ Q3	$\lvert P3 \rvert <$ Max_Length
	Enqueue(E, P);			
4		$\lvert Q1 \rvert > 0$	P4 = P3 $\circ$ <E3> $\wedge$ Q4 = Q3	$\lvert P4 \rvert + \lvert Q4 \rvert \leq$ Max_Length $\wedge$ $\lvert Q4 \rvert < \lvert Q0 \rvert$
	Append(P, Q);			
5		$\lvert Q1 \rvert > 0$	P5 = P4 $\circ$ Q4 $\wedge$ E5 = E4	
	end;			
6.1		$\lvert Q1 \rvert > 0$	P6 = P5 $\wedge$ Q6 = Q5 $\wedge$ E6 = E4	P6 = P0 $\circ$ Q0
6.2		$\neg$ ($\lvert Q1 \rvert > 0$)	P6 = P1 $\wedge$ Q6 = Q1 $\wedge$ E6 = E1	P6 = P0 $\circ$ Q0

Table 6: Verification Table for a Recursive Procedure

Proof in state 2:

Assume $\lvert Q1 \rvert > 0$; (* implication assumption *)
Confirm $\lvert Q2 \rvert > 0$;
True. (* from assumption in state 2 *)

Proof in state 3:

Assume $|Q1| > 0$; (* implication assumption *)
Confirm $|P3| < Max\_Length$;
Confirm $|P2| < Max\_Length$; (* from assumption in state 3 *)
Confirm $|P1| < Max\_Length$; (* from assumption in state 2 *)
Confirm $|P1| + |Q1| \leq Max\_Length$;
(* from number theory and implication assumption *)
Confirm $|P0| + |Q0| \leq Max\_Length$;
(* from assumption in state 1 *)
True. (* from assumption in state 0 *)

Proof in state 4 for conjunction $|P4| + |Q4| \leq Max\_Length$:

Assume $|Q1| > 0$; (* implication assumption *)
Confirm $|P4| + |Q4| \leq Max\_Length$;
Confirm $(|P3| + 1) + |Q3| \leq Max\_Length$;
(* from string theory and assumption in state 4 *)
Confirm $(|P2| + 1) + (|Q2| - 1) \leq Max\_Length$;
(* from string theory and assumption in state 3 *)
Confirm $|P2| + |Q2| \leq Max\_Length$; (* from number theory *)
Confirm $|P1| + |Q1| \leq Max\_Length$;
(* from assumption in state 2 *)
Confirm $|P0| + |Q0| \leq Max\_Length$;
(* from assumption in state 1 *)
True. (* from assumption in state 0 *)

Proof in state 4 for termination conjunction $|Q4| < |Q0|$:

Assume $|Q1| > 0$; (* implication assumption *)
Confirm $|Q4| < |Q0|$;
Confirm $|Q3| < |Q0|$; (* from assumption in state 4 *)
Confirm $|Q3| + 1 \leq |Q0|$; (* from number theory *)
Confirm $|Q2| \leq |Q0|$; (* from assumption in state 3 *)
Confirm $|Q1| \leq |Q0|$; (* from assumption in state 2 *)
Confirm $|Q0| \leq |Q0|$; (* from assumption in state 1 *)
True. (* from number theory *)

Proof in state 6 when $|Q1| > 0$

Assume $|Q1| > 0$; (* implication assumption *)
Confirm P6 = P0 ∘ Q0;
Confirm P5 = P0 ∘ Q0; (* from assumption in state 6 *)
Confirm P4 ∘ Q4 = P0 ∘ Q0; (* from assumption in state 5 *)
Confirm (P3 ∘ <E3>) ∘ Q3 = P0 ∘ Q0;(* from assumption in state 4 *)
Confirm P3 ∘ (<E3> ∘ Q3) = P0 ∘ Q0;
(* from string theory; associativity of ∘ *)
Confirm P2 ∘ Q2 = P0 ∘ Q0; (* from assumption in state 3 *)
Confirm P1 ∘ Q1 = P0 ∘ Q0; (* from assumption in state 2 *)
Confirm P0 ∘ Q0 = P0 ∘ Q0; (* from assumption in state 1 *)
True. (* from string theory *)

Proof in state 6 when $\neg (|Q1| > 0)$:

Assume $\neg (|Q1| > 0)$; (* implication assumption *)
Confirm P6 = P0 ∘ Q0;
Confirm P1 ∘ Q1 = P0 ∘ Q0; (* from assumption in state 6 *)
Confirm P0 ∘ Q0 = P0 ∘ Q0; (* from assumption in state 1 *)
True. (* from string theory *)

Figure 44: Proof of Total Correctness for a Recursive Procedure

The literature makes a distinction between *partial* and *total correctness* proofs. Total correctness requires a proof of termination in addition to partial correctness. The decreasing clause is essential for proving total correctness. Exercises illustrate the ideas further.

A Second Example

We conclude this discussion with a second example. Figure 45 contains the specification of a Queue Reverse operation. Figure 46 contains a recursive implementation, the proof of which is left as an exercise.

```
Enhancement Reversal_Capability for Queue_Template;
        Operation Reverse (updates Q: Queue);
                ensures     Q = #Q^Rev;
end Reversal_Capability;
```

Figure 45: Specification of a Queue Reverse Enhancement

```
Realization Recursive_Realiz for Reversal_Capability of Queue_Template;
        Recursive procedure Reverse (updates Q: Queue);
                decreasing |Q|;

                Var E: Entry;
                If Length (Q) > 0 then
                        Dequeue(E, Q);
                        Reverse(Q);
                        Enqueue(E, Q);
                end;
        end Reverse;
end Recursive_Realiz;
```

Figure 46: A Recursive Implementation of Queue Reverse Procedure

In completing the proof, it becomes necessary to appeal to a common theorem involving reversal from string theory. **Theorem** Str_Rev_Thm:

$\forall \alpha$: Str(Entry), E: Entry, $(\langle E \rangle \circ \alpha)^{Rev} = \alpha^{Rev} \circ \langle E \rangle$;

4.2.3 Verification of Iterative Procedures

Figure 47 contains an iterative version of procedure Append.

```
Realization Iterative_Realiz for Append_Capability of Queue_Template;
        Procedure Append (updates P: Queue; clears  Q: Queue);
                Var E: Entry;
                While Length (Q) > 0
                        updating P, Q, E;
                        maintaining (P ∘ Q = #P ∘ #Q);
                        decreasing |Q|;
                do
                        Dequeue(E, Q);
                        Enqueue(E, P);
                end;
        end Append;
end Iterative_Realiz;
```

Figure 47: An Iterative Implementation

Verification of a procedure containing a loop also employs the principle of induction, though the approach is different because of the difference in structure. To enable verification, programmers need to identify an iteration-independent *invariant* about a loop. This is a statement of what relationship a loop maintains among the loop variables. The *maintaining clause* is true at the beginning and at the end of every iteration (including the beginning of the first and end of the last iterations). Values of variables that are in scope, but that are not listed in the updating clause are assumed to be invariant, automatically. Some examples of invariants for the loop in Append procedure are given below:

$|\#P| \leq |P|$;
$|Q| \leq |\#Q|$;
$|P| + |Q| \leq$ Max_Length;
$|P| + |Q| = |\#P| + |\#Q|$;
$P \circ Q = \#P \circ \#Q$

Some loop invariants are stronger than others. The invariant $P \circ Q = \#P \circ \#Q$, for example, implies $|P| + |Q| = |\#P| + |\#Q|$, another invariant of

this same loop. To show that a loop is correct, the person writing a loop is responsible for identifying and writing down an invariant. Identification of "suitable" loop invariants comes from practice. To show that a loop terminates, as in the case of recursive implementations, a **decreasing** assertion is also needed.

Proof Process for Loop Verification

Once updating, maintaining, and decreasing clauses are identified, there are three steps (in no particular order) in the verification of a loop.

Step 1: To prove that the invariant is actually an invariant

One step is to prove that the maintaining clause supplied by the programmer is actually an invariant, i.e., it is true at the beginning and at the end of every iteration. This is necessary because we do not want to prove a wrong procedure to be correct based on a faulty invariant. To show that the invariant is true at the end of every iteration, we appeal to the principle of induction as explained below.

There is a basic and inductive step in the proof. First we show that the loop invariant is true before the first iteration. For the inductive step, we assume that the invariant and the loop condition are true at the beginning of an (arbitrary) iteration and show that the invariant is true at the end. This works because, if the invariant is true at the beginning of the first iteration, then it must also be true at the end of the first iteration because we have proved so in the inductive step. The assertion that is true at the end of the first iteration is also true at the beginning of the next iteration; since the invariant is true at the beginning of the second iteration, it must be true at the end of that iteration and so on. Proceeding this way, we can conclude that the invariant must be true at the end of the last iteration of the loop.

Step 2: Using the invariant to prove obligations following the loop

Once the invariant has been shown to be true, it can be assumed to be true in the state immediately after the loop. This assumption is valid, because the invariant has been shown to be true at the end of each iteration including the last one inductively in the previous step. In the state after the loop, the negation of loop condition also becomes an assumption. These facts are used in showing that the obligations following the loop are true.

Step 3: Proving termination

For total correctness, it is essential to show that loops terminate. To show termination we need to show that there is "progress". Using the decreasing assertion, we show that the metric specified in the assertion is strictly smaller at the end of an iteration compared to its value at the beginning. Since the metric has a natural number (or an ordinal number, in general) as its value, if it decreases in every iteration it will eventually become 0 and the loop will terminate. These steps are illustrated for Append in Table 7.

State	**Stmt**	**Path Condition**	**Assume**	**Confirm**
0			$\|P0\| + \|Q0\| \leq$ Max_Length $\wedge$ $\|P0\| \leq$ Max_Length $\wedge$ $\|Q0\| \leq$ Max_Length	
	Var E: Entry;			
1			P1 = P0 $\wedge$ Q1 = Q0 $\wedge$ Entry.**Is_Initial**(E1)	**(**P1 $\circ$ Q1 = P0 $\circ$ Q0)
	While Length (Q) > 0 **updating** P, Q, E; **maintaining (**P $\circ$ Q = #P $\circ$ #Q); **decreasing** |Q|; **do**			
2		$\|Q2\| > 0$	$\|Q2\| > 0 \wedge$ **(**P2 $\circ$ Q2 = P0 $\circ$ Q0)	$\|Q2\| > 0$
	Dequeue(E, Q);			
3		$\|Q2\| > 0$	P3 = P2 $\wedge$ Q2 = <E3> $\circ$ Q3	$\|P3\| <$ Max_Length
	Enqueue(E, P);			
4		$\|Q2\| > 0$	P4 = P3 $\circ$ <E3> $\wedge$ Q4 = Q3	**(**P4 $\circ$ Q4 = P0 $\circ$ Q0) $\wedge$ $\|Q4\| < \|Q2\|$
	end;			
5			**(**P5 $\circ$ Q5 = P0 $\circ$ Q0) $\wedge$ $\neg$ ($\|Q5\| > 0$)	P5 = P0 $\circ$ Q0

Table 7: Verification Table for an Iterative Procedure

In the table, there is an obligation the obligation in state 1, the state immediately before the loop, is the induction base step and it is to prove that the invariant is true at the beginning of the first iteration. In writing down the obligation, we substitute P0 and Q0 for #P and #Q, respectively; for P and Q, we have also substituted P1 and Q1, because these are their "current" values in state 1.

In state 1, the first state within the loop, we assume the invariant for induction. We also assume that the loop condition is true, since we have entered the loop. The obligations in state 2 stem from the precondition of Dequeue, and those in state 3 come from the precondition of Enqueue. The assumptions in states 3 and 4 are determined from the ensures clauses of Dequeue and Enqueue. In state 4, the last state within the loop, we need to prove that the loop invariant is true.

For the loop in Append, the decreasing clause is shown to be valid, by showing that at the end of an iteration $|Q|$ is strictly less than $|Q4|$ at the beginning of that iteration, i.e., $|Q4| < |Q2|$.

In state 5, immediately after the loop, we assume that the loop invariant is true. Using this assumption and the negation of the loop condition, we then attempt to prove the obligations in that state. If the invariant is not sufficiently strong, it would not be possible to complete this step of the proof. Figure 48 contains a proof.

Proof in state 1:

```
    Confirm (P1 ∘ Q1 = P0 ∘ Q0);
    Confirm (P0 ∘ Q0 = P0 ∘ Q0);
                                    (* from assumption in state 1 *)
    True.                           (* from assumption in state 0 *)
```

Proof in state 2:

```
    Assume |Q2| > 0;                (* implication assumption *)
    Confirm |Q2| > 0;
    True.                           (* from assumption *)
```

Proof in state 3:

Assume $|Q2| > 0$; (* implication assumption *)
Confirm $|P3| < Max\_Length$;
Confirm $|P2| < Max\_Length$; (* from assumption in state 3 *)
Confirm $|P2| + |Q2| \leq Max\_Length$;
(* from number theory and implication assumption *)
True. (* from assumption in state 2 *)

Proof in state 4 for invariant conjunction **(**$P4 \circ Q4 = P0 \circ Q0$):

Assume $|Q2| > 0$; (* implication assumption *)
Confirm ($P4 \circ Q4 = P0 \circ Q0$);
Confirm $(P3 \circ \langle E3 \rangle) \circ Q3 = P0 \circ Q0$;(* from assumption in state 4 *)
Confirm $P3 \circ (\langle E3 \rangle \circ Q3) = P0 \circ Q0$;
(* from string theory; associativity of $\circ$ *)
Confirm $P2 \circ Q2 = P0 \circ Q0$; (* from assumption in state 3 *)
True. (* from assumption in state 2 *)

Proof in state 4 for termination conjunction $|Q4| < |Q2|$:

Assume $|Q2| > 0$; (* implication assumption *)
Confirm $|Q4| < |Q2|$;
Confirm $|Q3| < |Q2|$; (* from assumption in state 4 *)
Confirm $|Q3| + 1 \leq |Q2|$; (* from number theory *)
Confirm $|Q2| \leq |Q2|$; (* from assumption in state 3 *)
True. (* from number theory *)

Proof in state 5:

Confirm $P5 = P0 \circ Q0$;
Confirm $P5 \circ \Lambda = P0 \circ Q0$; (* from string theory *)
Confirm $P5 \circ Q5 = P0 \circ Q0$;
(* from string theory and assumption $\neg (|Q5| > 0)$ in state 5 *)
True. (* from assumption in state 5 *)

Figure 48: Proof of Total Correctness for an Iterative Procedure

We conclude this discussion with other examples. Figure 49 contains the specification of a Stack Reverse operation. Figure 50 contains an iterative implementation, the proof of which is left as an exercise.

```
Enhancement Reversal_Capability for Stack_Template;
    Operation Reverse(updates S: Stack );
        ensures S = (#S)^Rev;
end Reversal_Capability;
```

Figure 49: Specification of A Stack Reversal Enhancement

```
Realization Obvious_Realiz for Reversal_Capability of Stack_Template;
    Procedure Reverse(updates S: Stack );
        Var S_Reversed: Stack;
        Var Next_Entry: Entry;

        While (Depth(S) > 0)
            updating S, S_Reversed, Next_Entry;
            maintaining (#S = (S_Reversed)^Rev ∘ S) ∧ |#S| ≤ Max_Depth;
            decreasing |S|;
        do
            Pop(Next_Entry, S);
            Push(Next_Entry, S_Reversed);
        end;
        S :=: S_Reversed;
    end Reverse;
end Obvious_ Realiz;
```

Figure 50: An Iterative Implementation of Stack Reversal

Finally to see how verification applies for a procedure with multiple loops, consider an iterative implementation of Queue reverse operation using a local Stack. The implementation and its proof are left as exercises.

4.2.4 Proof of Data Abstraction Realization

We conclude this chapter with a proof of correctness of a data abstraction realization where correspondence and conventions assertions are involved.

Figure 51 reproduce the array realization of Stack_Template for convenient reference.

```
Realization Array_Realiz for Stack_Template;
        uses Record_Template, Static_Array_Template;

    Type Stack = Record
                    Contents: Array 1..Max_Depth of Entry;
                    Top: Integer;
                end;
            conventions 0 ≤ S.Top ≤ Max_Depth;
            correspondence Conc.S = (∏ from i=1 to S.Top <S.Contents(i)>)^Rev;

    Procedure Push(alters E: Entry; updates S: Stack);
        S.Top := S.Top + 1;
        E :=: S.Contents(S.Top);
    end Push;

    Procedure Pop(replaces R: Entry; updates S: Stack);
        R :=: S.Contents(S.Top);
        S.Top := S.Top - 1;
    end Pop;

    Procedure Depth(preserves S: Stack): Integer;
        Depth := (S.Top);
    end Depth;

    Procedure Rem_Capacity (preserves S: Stack): Integer;
        Rem_Capacity := (Max_Depth – S.Top);
    end Rem_Capacity;

    Procedure Clear(clears S: Stack);
        S.Top := 0;
    end Clear;
end Array_Realiz;
```

Figure 51: Array Realization of Stack_Template Including Assertions

There are two aspects in proving correctness. The first concerns showing that the correspondence is well defined. The second issue is to prove correctness of each procedure with respect to its abstract specifications.

Proof of Well-Defined Correspondence

Confirm ∀S: Stack, **if** 0 ≤ S.Top ≤ Max_Depth **then**
∃**Conc**.S: String(Entry) ∋
$$\mathbf{Conc}.S = \left(\prod_{i=1}^{S.Top} <S.Contents(i)>\right)^{Rev} \textbf{ and } |\mathbf{Conc}.S| \leq Max\_Depth;$$

Proof by contradiction

Assume that the assertion is not true.

(0 ≤ S.Top ≤ Max_Depth) **and**
not (∃**Conc**.S: String(Entry) ∋
$$\mathbf{Conc}.S = \left(\prod_{i=1}^{S.Top} <S.Contents(i)>\right)^{Rev} \textbf{ and } |\mathbf{Conc}.S| \leq Max\_Depth);$$

This becomes

(0 ≤ S.Top ≤ Max_Depth) **and**
(∀**Conc**.S: String(Entry),
$$\textbf{not } (\mathbf{Conc}.S = \left(\prod_{i=1}^{S.Top} <S.Contents(i)>\right)^{Rev} \textbf{ and } |\mathbf{Conc}.S| \leq Max\_Depth);$$

This becomes

(0 ≤ S.Top ≤ Max_Depth) **and**
(∀**Conc**.S: String(Entry),
$$\textbf{not } (\mathbf{Conc}.S = \left(\prod_{i=1}^{S.Top} <S.Contents(i)>\right)^{Rev}) \textbf{ or}$$
$$\textbf{not } (|\mathbf{Conc}.S| \leq Max\_Depth);$$

This becomes

(0 ≤ S.Top ≤ Max_Depth) **and**
(∀**Conc**.S: String(Entry),

not (**Conc**.S = $\left(\prod_{i=1}^{S.Top} <S.Contents(i)>\right)^{Rev}$) **or**
not ($|$**Conc**.S$|$ ≤ Max_Depth);

This becomes

(0 ≤ S.Top ≤ Max_Depth) **and**
(∀**Conc**.S: String(Entry),
Conc.S ≠ $\left(\prod_{i=1}^{S.Top} <S.Contents(i)>\right)^{Rev}$) **or**
$|$**Conc**.S$|$ > Max_Depth;

This becomes

(0 ≤ S.Top ≤ Max_Depth) **and**
∀**Conc**.S: String(Entry),
Conc.S ≠ Λ **or**
Conc.S ≠ $\left(\prod_{i=1}^{S.Top} <S.Contents(i)>\right)^{Rev}$) **or**
$|$**Conc**.S$|$ > Max_Depth;

This is **false** when **Conc**.S = Λ.

Proof of Correctness of Procedures

II.1. Proof of Procedure initialization;
It has no code, but Contents and Top are initialized.
Assume Stack.**Is_Initial**(S);
Confirm 0 ≤ S.Top ≤ Max_Depth **and**
if Conc.S = $\left(\prod_{i=1}^{S.Top} <S.Contents(i)>\right)^{Rev}$ **then Conc**.S = Λ;
Proof: Left as exercise.

II.2. Proof of Procedure Push;

The proof does not show absence of array bound violations. This subproof is left as an exercise.

Prove:

Assume $0 \le S.Top \le Max\_Depth$ **and**
$\quad$ **Conc**.S = $\left(\prod_{i=1}^{S.Top} \langle S.Contents(i) \rangle\right)^{Rev}$ **and** $|$**Conc**.S$|$ < Max_Depth;
S.Top := S.Top + 1;
E :=: S.Contents(S.Top);
Confirm $0 \le S.Top \le Max\_Depth$ **and**
$\quad$ **if Conc**.S = $\left(\prod_{i=1}^{S.Top} \langle S.Contents(i) \rangle\right)^{Rev}$ **then**
$\quad\quad$ **Conc**.S = <#E> o #**Conc**.S;

Prove (after state-based symbol substitution):

Assume $0 \le S0.Top \le Max\_Depth$ **and**
$\quad$ **Conc**.S0 = $\left(\prod_{i=1}^{S0.Top} \langle S0.Contents(i) \rangle\right)^{Rev}$ **and**
$\quad\quad$ $|$**Conc**.S0$|$ < Max_Depth;

S.Top := S.Top + 1;
E :=: S.Contents(S.Top);

Confirm $0 \le S2.Top \le Max\_Depth$ **and**
$\quad$ **if Conc**.S2 = $\left(\prod_{i=1}^{S2.Top} \langle S2.Contents(i) \rangle\right)^{Rev}$ **then**
$\quad\quad$ **Conc**.S2 = <E0> o **Conc**.S0;

Proof table:

Assume $0 \le S0.Top \le Max\_Depth$ **and**
$\quad$ **Conc**.S0 = $\left(\prod_{i=1}^{S0.Top} \langle S0.Contents(i) \rangle\right)^{Rev}$ **and**
$\quad\quad$ $|$**Conc**.S0$|$ < Max_Depth;

S.Top := S.Top + 1;

Assume S1.Top = S0.Top + 1 **and** S1.Contents = S0.Contents **and**
E1 = E0;

E :=: S.Contents(S.Top);

Assume S2.Top = S1.Top **and**
E2 = S1.Contents(S1.Top) **and** S2.Contents(S1.Top) = E1 **and**
∀index: Z, **if** index ≠ S1.Top **then**
S2.Contents(index) = S1.Contents(index);

Confirm 0 ≤ S2.Top ≤ Max_Depth **and**
if Conc.S2 = $\left(\prod_{i=1}^{S2.Top} \langle S2.Contents(i) \rangle\right)^{Rev}$ **then**
Conc.S2 = <E0> o **Conc**.S0;

Simplification using assumptions in state 2:

Confirm if (S2.Contents(S1.Top) = E1 **and**
∀index: Z, **if** index ≠ S1.Top **then**
S2.Contents(index) = S1.Contents(index)) **then**
0 ≤ S1.Top ≤ Max_Depth **and**
if Conc.S2 = $\left(\prod_{i=1}^{S1.Top} \langle S2.Contents(i) \rangle\right)^{Rev}$ **then**
Conc.S2 = <E0> o **Conc**.S0;

Simplification using assumptions in state 1:

Confirm if (S2.Contents(S0.Top + 1) = E0 **and**
∀index: Z, **if** index ≠ S0.Top + 1 **then**
S2.Contents(index) = S0.Contents(index)) **then**
0 ≤ S0.Top + 1 ≤ Max_Depth **and**
if Conc.S2 = $\left(\prod_{i=1}^{S0.Top+1} \text{<S2.Contents(i)>}\right)^{Rev}$ **then**
Conc.S2 = <E0> o **Conc**.S0;

Now use assumptions in state 0:

Assume 0 ≤ S0.Top ≤ Max_Depth **and**
Conc.S0 = $\left(\prod_{i=1}^{S0.Top} \text{<S0.Contents(i)>}\right)^{Rev}$ **and**
$|$**Conc**.S0$|$ < Max_Depth;

Confirm if (S2.Contents(S0.Top + 1) = E0 **and**
∀index: Z, **if** index ≠ S0.Top + 1 **then**
S2.Contents(index) = S0.Contents(index)) **then**
0 ≤ S0.Top + 1 ≤ Max_Depth **and**
if Conc.S2 = $\left(\prod_{i=1}^{S0.Top+1} \text{<S2.Contents(i)>}\right)^{Rev}$ **then**
Conc.S2 = <E0> o **Conc**.S0;

Proof:

First confirm $0 \le S0.Top + 1 \le Max\_Depth$; exercise.

Rewrite remaining assertion as below:

Assume Conc.S0 = $\left(\prod_{i=1}^{S0.Top} <S0.Contents(i)>\right)^{Rev}$;

Confirm if (S2.Contents(S0.Top + 1) = E0 **and**
∀index: Z, **if** index ≠ S0.Top + 1 **then**
S2.Contents(index) = S0.Contents(index)) **then**
if Conc.S2 =
<S2.Contents(S0.Top + 1)> o $\left(\prod_{i=1}^{S0.Top} <S2.Contents(i)>\right)^{Rev}$ **then**
Conc.S2 = <E0> o **Conc**.S0;

On simplification, this becomes:

Assume Conc.S0 = $\left(\prod_{i=1}^{S0.Top} <S0.Contents(i)>\right)^{Rev}$;

Confirm if (∀index: Z, **if** index ≠ S0.Top + 1 **then**
S2.Contents(index) = S0.Contents(index)) **then**
if Conc.S2 = <E0> o $\left(\prod_{i=1}^{S0.Top} <S2.Contents(i)>\right)^{Rev}$ **then**
Conc.S2 = <E0> o **Conc**.S0;

On simplification, this becomes:

Assume Conc.S0 = $\left(\prod_{i=1}^{S0.Top} <S0.Contents(i)>\right)^{Rev}$;
Confirm if Conc.S2 = <E0> o $\left(\prod_{i=1}^{S0.Top} <S0.Contents(i)>\right)^{Rev}$ **then**
Conc.S2 = <E0> o **Conc**.S0;

On simplification, this becomes:

Confirm if Conc.S2 = <E0> o **Conc**.S0 **then**
Conc.S2 = <E0> o **Conc**.S0;

True.