April 28, 1997

SRC Research Report 145

Modularity in the Presence of Subclassing

Raymie Stata

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Systems Research Center

The charter of SRC is to advance both the state of knowledge and the state of the art in computer systems. From our establishment in 1984, we have performed basic and applied research to support Digital's business objectives. Our current work includes exploring distributed personal computing on multiple platforms, networking, programming technology, system modelling and management techniques, and selected applications.

Our strategy is to test the technical and practical value of our ideas by building hardware and software prototypes and using them as daily tools. Interesting systems are too complex to be evaluated solely in the abstract; extended use allows us to investigate their properties in depth. This experience is useful in the short term in refining our designs, and invaluable in the long term in advancing our knowledge. Most of the major advances in information systems have come through this strategy, including personal computing, distributed systems, and the Internet.

We also perform complementary work of a more mathematical flavor. Some of it is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. Other work explores new ground motivated by problems that arise in our systems research.

We have a strong commitment to communicating our results; exposing and testing our ideas in the research and development communities leads to improved understanding. Our research report series supplements publication in professional journals and conferences. We seek users for our prototype systems among those with whom we have common interests, and we encourage collaboration with university researchers.

Modularity in the Presence of Subclassing

Raymie Stata

April 28, 1997

Publication History

This report is a revision of a thesis of the same title submitted to the Department of Electrical Engineering and Computer Science at the Massachusetts Instutite of Technology. This revision is also published by MIT as MIT–LCS–TR–711.

Abstract

Classes are harder to subclass than they need be. This report addresses this problem, showing how to design classes that are more modular and easier to subclass without sacrificing the extensibility that makes subclassing useful to begin with.

We argue that a class should have two interfaces, an *instance interface* used by programmers manipulating instances of the class, and a *specialization interface* used by programmers building subclasses of the class. Instance interfaces are relatively well understood, but design principles for specialization interfaces are not.

In the context of single inheritance, we argue that specialization interfaces should be partitioned into *class components*. A class component groups part of a class's state together with methods to maintain that state. Class components establish abstraction boundaries *within* classes, allowing modular replacement of components by subclasses. Achieving this replaceability requires reasoning about each component as an independent unit that depends only on the specifications of other components and not on their implementations.

We introduce the concept of *abstract representation* to denote the view of a class's state given in its specialization interface. This view is more detailed than the view used to describe instances of the class, revealing details that describe the interfaces between class components. It is less detailed than the actual implementation, hiding implementation details that should not be exposed even to specializers.

We also consider multiple inheritance, specifically, Snyder's model of encapsulated multiple inheritance. We advocate separating class components into individual classes called *mixins*. Instantiable classes are built by combining multiple mixins. With the mixin style of design, class hierarchies have more classes than in equivalent single-inheritance designs. These classes have smaller, simpler interfaces and can be reused more flexibly.

To explore the impact our ideas might have on program design, we consider classes from existing libraries in light of the proposed single- and multiple-inheritance methodologies. To explore the impact our ideas might have on language design, we present two different extensions to Java, one that provides a level of static checking for single-inheritance designs, and another that adds the encapsulated model of multiple inheritance.

Acknowledgements

This report revises a PhD dissertation of the same title. My advisor, John Guttag, contributed to the thesis in innumerable ways. His own research on software engineering, and the courses he's developed and taught, were important, technical influences. By turning my attention to the right issues, offering key, technical insights, keeping me focused at the right level of abstraction, and providing expert feedback on an endless stream of notes, papers, and drafts, he guided me in turning vague intuitions into useful ideas and eventually into a dissertation. By mixing unlimited freedom to explore ideas with invaluable wisdom and advice, he created a unique environment in which I could develop as a researcher. By offering encouragement, support, patience, and understanding, he made graduate school survivable. He made my graduate school experience irreplaceable.

Barbara Liskov also contributed to the thesis in essential ways. Her own work on programming methodology and language design, and the courses she's developed and taught, influenced and inspired the thesis. As a member of my committee, she worked hard to help me extract, refine, organize and explain the central ideas of the thesis. Her effort went well beyond the call of duty, and the thesis is much better for it.

Luca Cardelli was also an invaluable member of my committee. Once again, his own work on the foundations of object-oriented programming and type systems influenced the thesis. In his insightful comments on papers and chapters, he brought a valuable perspective to the thesis. He asked incisive questions—and suggested important cuts—that improved many aspects of the thesis.

My work benefited greatly from input from many people, including Steve Garland, Alan Heydon, Daniel Jackson, Depak Kapur, John Lamping, Gary Leavens, Rustan Leino, Andrew Meyer, Greg Nelson, Nate Osgood, Anna Pogosyants, Yang-Meng Tan, Jim O'Toole, Franklyn Turbeck, John Turkovitch, Mark Reinhold, Mark Vandevoorde, Michael Vanhilst, and Jeannette Wing. David Evans in particular spent many hours discussing ideas and commenting on writing and presentations. My work benefited from experience gained at Draper Labs, and from a productive stay at Digital Equipment's System Research Center.

This work owes much to the seminal writing of Dijstra, Hoare, and Parnas. It also owes much to the people, past and present, who made MIT the great institution it is today, and in particular to those who made 6.001, 6.170, 6.035, and 6.821 such great courses.

I would like to thank my family and friends for their support and encouragement and for giving me the confidence needed to start the thesis—and to finish. I would like to thank in particular my parents for their love and support and for encouraging me in my interests and education. I'd like to offer special thanks to Kimberly Sweidy for her support, patience, inspiration, understanding, encouragement, wisdom, exuberance, and love.

Contents

Chapter 1

Introduction

Reuse and modularity are two important principles for improving programmer productivity. Object-oriented class libraries support software reuse in two ways. First, they define "black boxes" that can be used as-is in a variety of contexts. Second, they define "extensible boxes" that can be customized via subclassing to fit the needs of a particular context. Achieving modularity for black-box reuse is well-understood, but we do not yet understand how to achieve modularity for extensible-box reuse. The goal of this report is to reconcile modularity with the extensibility afforded by subclassing.

Modular designs are composed of loosely coupled components. An important technique for decreasing coupling is to use specifications to draw abstraction barriers around components. (Specifications are documentation that hides implementation details; we use the term "specification" and "documentation" interchangeably.) A component is said to be *independent* of other components if it depends only on their specifications and is said to *break abstraction barriers* if it depends on their implementations.

In existing programming methodologies, classes are treated as the unit of modularity. As a result, documentation for class libraries draws abstraction barriers around entire classes (see, *e.g.*, [Borland94] and [Microsoft94]). This is fine for instantiators, clients whose code manipulates instances of a class. However, specializers, clients who create subclasses, use method override to replace *parts* of classes, and thus interact with classes at a finer granularity. For specializers, classes are too coarse a unit of modularity. As a result, when given documentation in terms of entire classes, specializers are forced to break abstraction barriers.

For example, consider a class `IntSet` of mutable integer-set objects. An informal specification for `IntSet` is given in Fig. 1.1. This documentation draws an abstraction barrier around the entire `IntSet` class. Such documentation is fine for instantiators, but it does not answer the questions of specializers. For example, assume a programmer wants to build a multiset abstraction by subclassing `IntSet`. Can it be done? If so, which methods should be overridden? What constraints must be met by overridden methods? These questions are not answered by documentation that takes the entire class as the unit of modularity.

Because of method override, specializers need a unit of modularity smaller than the entire class. At first, one might think that individual methods should be the unit of modularity for specializers; after all, methods are what get replaced by method override. However, if methods were the unit of modularity, then the methods of a class would have to be independent of one another, *i.e.*, they could not share any hidden implementation information. As we shall see, this is too strict. In particular, to support extensibility for state, methods cannot always be made independent of each other.

Documentation for the IntSet class:
  IntSet objects are mutable, integer-set objects

  public void addElement(int el)
    Modifies **this**
    Effects Adds el to **this**.

  public void addElements(IntVector els)
    Modifies **this**
    Effects Adds elements of els to **this**.

  public boolean contains(int el)
    Effects Returns true iff el is a member of **this**.

  public IntEnumeration elements()
    Effects Returns an enumeration yielding the elements of **this**.

  public boolean isEmpty()
    Effects Returns true iff **this** is empty.

  public boolean removeElement(int el)
    Modifies **this**
    Effects If el in **this**, remove it and return true; otherwise, return false.

  public void removeElements(IntVector els)
    Modifies **this**
    Effects Removes elements of els from **this**.

  public int size()
    Effects Returns the number of elements in **this**.

  public String toString()
    Effects Returns a printable representation of **this**.

  ...other public methods elided...

  protected void uncache(int el)
    Modifies **this**
    Effects Invalidates the internal membership cache at el.

Figure 1.1: Description of `IntSet` typical of today's documentation.

To reconcile modularity and extensibility, this report starts with the assumption that classes should have two interfaces, one for instantiators and one for specializers. The report proposes entire classes as the unit of modularity for instance interfaces and *class components* as the unit of modularity for specialization interfaces. After describing class components, the report pursues a number of additional results that follow from using them as the unit of modularity. The report describes both formal and informal documentation for specialization interfaces. It describes a validation criteria for classes built out of class components. Finally, the report presents advice, based on class components, about designing specialization interfaces that are both modular and extensible.

The next section describes the language assumptions of the report. Sec. 1.2 gives an overview of class components and how they facilitate documentation, validation, and design. Sec. 1.3 both describes the organization of the report and summarizes its results.

1.1 Language model

This report assumes a model of object-oriented programming based on objects, object types, and classes. The model is a standard, single inheritance model, except that, like Theta ([Day95]), it separates object types from classes and it associates behavioral specifications with object types. Chapter 7 considers multiple inheritance.

An *object* is a set of instance variables and code for its methods.

An *object type* describes the behavior of objects. Looked at the other way around, we say that an object *implements* an object type when the object behaves as described by the type. In most work on object-oriented programming, object types include only *signature specifications* that describe the types of method arguments and results. In this report, object types are also associated with *object specifications* that describe the behavior of objects. Object specifications consist of an abstract description of the object's state together with descriptions of the behaviors of the object's methods. The method descriptions are given in terms of the abstract state, not in terms of the instance variables used to represent that state. Object specifications are described in more detail in the next chapter, and a formal approach is described in Sec. 4.1.

A *class* is a template consisting of method and instance variables definitions. *Instantiating* a class creates a new object that has the instance variables and method code defined by the class. Instantiating a class `C` creates a *direct instance* of `C`. An *instance* of `C` is a direct instance of either `C` or any subclass of `C`. Where object types describe the behavior of objects apart from any implementations, classes describe the implementations of objects. We say that a class *implements* an object type when all its direct instances implement the type.

A class is defined by a set of *local* method and instance variable definitions and an optional superclass from which additional definitions are *inherited*. A class can also define *class variables* that are shared by all instances of the class. A class *overrides* a superclass method by defining locally a method with the same name as one of its superclass's methods. As in many languages, a method can have one of three implementation categories: final, deferred, or overridable. Final methods cannot be overridden by subclasses, overridable methods can. Deferred methods are methods a class declares and calls in its other methods but for which it does not provide code. Subclasses provide code for deferred methods by overriding them. A class with deferred methods is called a *deferred class* and cannot be instantiated. (In the literature, deferred methods and classes are often called "abstract" methods and classes.)

Also in many languages, a method can be *public*, *protected*, or *private*, meaning, respectively, the method is visible to instantiators and specializers, is visible only to specializers, or is visible only inside the class itself. Except where explicitly noted, instance variables are private, *i.e.*, only visible inside the class itself.

The *instance interface* of a class is the interface used by instantiators; the *specialization interface* is the interface used by specializers and includes protected methods. The *instance specification* documents the instance interface of a class, while the *specialization specification* documents the specialization interface. Given our language model, one can see that specialization specifications need to document protected methods while instance specifications do not. We will show that there are other, more important differences between the two.

1.2 Class components

This report is centered around a simple idea: that classes should be built out of class components. A class component is a piece of state—called *substate*—and a set of methods responsible for maintaining that state.

Class components are illustrated in Fig. 1.2, which gives a partial implementation of `IntSet`. In this figure, `IntSet` has two class components, one for the actual elements of the set, and another for a cache used to cache membership tests. The substate of the elements component is represented by an `IntVector` object, an array of integers whose size changes dynamically. This part of the representation is maintained by the methods `addElement`, `removeElement`, and `elements`. The substate of the cache component is represented by an integer and a boolean and is maintained by the methods `contains` and `uncache`. As this example illustrates, class components are very much like data abstractions: encapsulated state manipulated by a set of operations. This similarity explains why class components are good units of modularity.

Class components have historical roots in programming conventions for extensible state. In the context of subclassing, *extensibility* means allowing a subclass to provide its own implementation of some aspect of its superclass. Programming languages directly support extensibility for methods by allowing subclasses to provide their own implementations for superclass methods. However, languages do not directly support extensibility for state: there is no mechanism that allows subclasses to provide their own representations of superclass state.

To address this asymmetry, programmers have developed class components as a convention that does allow subclasses to provide their own representations for superclass state. The key to this convention, as illustrated by `IntSet`, is that methods in the elements component call the methods of the cache component rather than accessing the instance variables representing the cache. Subclasses replace the representation of a superclass component by overriding the methods of a component with new code that uses a new representation. For example, `IntSet2` in Fig. 1.3 replaces the representation of the cache by overriding `contains` and `uncache` with code that represents the cache using a bit vector. This new representation can cache up to sixty-four hits, but it only caches hits on zero through sixty-three, so `IntSet2` is best used where membership tests on small, positive numbers dominate. (`IntSet2` inherits but does not use the instance variables `c_valid` and `c_val`, the old representation of the cache. Sec. 6.4.4 explains how these orphaned instance variables can be optimized away.)

In addition to facilitating extensibility, class components are good units of modularity for specialization interfaces. We saw earlier that the entire class is too big a unit. At the other extreme,

```
class IntSet {

 // ``elements'' component
   private IntVector els = new IntVector();

   public overridable void addElement(int el) {
     if (! els.contains(el)) els.addElement(el);
   };

   public overridable boolean removeElement(int el) {
     this.uncache(el); // Maintain cache validity
     return els.removeElement(el); // Call remove method of IntVector
   };

   public overridable IntEnumeration elements() {
     return els.elements();
   };

 // ``cache'' component
   private int c_val; // Value currently in cache
   private boolean c_valid = false; // True only if c_val is valid

   public overridable boolean contains(int el) {
     if (c_valid && c_val == el) return true;
     for(IntEnumeration e = this.elements(); e.hasMoreElements(); )
       if (el == e.nextElement()) {
         c_valid = true; c_val = el;
         return true;
       };
     return false;
   };

   protected overridable void uncache(int el) {
     if (c_val == el) c_valid = false;
   };

  ..other methods elided
};
```

Figure 1.2: Implementation of `IntSet`.

```
class IntSet2 extends IntSet {
 // Replace the ``cache'' component
   private long c_bits; // Used as a bitmap; caches 0 - 63 only
     // If c_bits[i] is true, then i is in the cache

   private final boolean c_test(int el)
     // If el is in range, return c_bits[el], otherwise return false
     { return 0 <= el && el < 64 && c_bits & (1 << el); }

   private final void c_set(int el, boolean val) {
     // If el is in range, set c_bits[el] = val
     if (0 <= el && el < 64)
       c_bits = (c_bits & ~(1 << el)) | (val ? (1 << el) : 0);
   }

   public boolean contains(int el) {
     if (c_test(el)) return true;
     for(IntEnumeration e = this.elements(); e.hasMoreElements(); ) {
       c_set(el, true);
       if (el == e.nextElement()) return true;
     };
     return false;
   };

   protected void uncache(int el)
     { c_set(el, false); };
};
```

Figure 1.3: Subclass of `IntSet` that replaces the cache.

individual methods are too small a unit of modularity. For example, consider the cache component of `IntSet`. The methods of this component, `contains` and `uncache`, share implementation information about the representation of the cache, *e.g.*, that `c_valid` and `c_val` are used to represent the cache and that `c_val` is a cached hit only when `c_valid` is true. This shared implementation information means that `contains` and `uncache` cannot be made independent of one another. However, the two methods taken together *are* independent: a specializer can replace them as a group with no knowledge of `c_valid` and `c_val`.

Thus, class components are ideal for reconciling extensibility and modularity. This realization leads to insights into design, documentation, and validation. For example, an important part of designing specialization specifications is deciding what state should be extensible. This extensible state should be subdivided into pieces that can be replaced independently, and each piece should be given its own class component. The documentation of specialization interfaces should identify the class components; this means documenting both the substate and the methods that make up the component. Class components should be independent of one another. This means that one component should not directly access the representations of other components but instead should access the substates of other components by calling their methods. For example, the `contains` method of `IntSet` does not directly access the representation of the elements of an `IntSet` but instead calls `elements`. Similarly, the `removeElements` method does not access the rep of the cache but instead calls `uncache`.

The need for independence feeds back into the design and documentation of specialization interfaces. If components are to be independent of one another, then each component must offer a *sufficient* interface to the others. For example, the existence of `uncache` is motivated by the desire to support independent access to the cache by `removeElement`. Also, documenting class components in an independent manner often requires exposing aspects of a class's state that is hidden from instantiators. For example, documenting the component containing `uncache` requires exposing the existence of the membership cache to specializers even though this cache need not be mentioned in the documentation given to instantiators.

1.3 Overview

This report introduces class components and explains why we think that they are the right unit of modularity for specialization interfaces. The report also explores the implications of designing specialization interfaces in terms of class components. The next chapter describes class components in more detail. It then describes how to document the specialization interface of classes built out of them. This documentation establishes abstraction boundaries around components, allowing subclasses to replace them without looking at the code of superclasses. The chapter also describes the differences between instance and specialization interfaces, justifying our decision to separate them.

Chapter 3 describes validation of specialization interfaces. This chapter first defines the validation criterion for specialization interfaces. Part of our criterion is the classical one for data abstractions: a class must implement the behavior described by its specification. Another part of our criterion is a new one introduced for specialization interfaces: the components of a class must be independent of one another. After defining this validation criterion, the rest of Chapter 3 focuses on the new aspect: reasoning about classes in a way that ensures the independence of components.

Chapter 4 describes formal specification and verification of specialization interfaces. This chapter formalizes results described informally by the previous two chapters. Chapter 5 extends these specifications in a way that improves the extensibility without sacrificing modularity.

Chapter 6 looks at design issues that arise in the context of specialization interfaces. First, it presents design guidelines for specialization interfaces, drawing on existing class libraries for examples. Next, it looks at the design of languages, presenting an extension to Java that supports class components.

Chapter 7 considers separating class components into separate classes, called *mixins*, that can be combined using multiple inheritance. The chapter assumes Snyder's encapsulated model of multiple inheritance, and it presents an extension to the Java language that embodies this model. The mixin style of design leads to class hierarchies with more classes than in equivalent single-inheritance designs, but in which classes have smaller, simpler interfaces and in which classes can be reused more flexibly.

Chapter 2

Basic methodology

Modularity is a product of both methodology and good design. Methodology defines the unit of modularity and the system of documentation. A good methodology allows modularity, but it does not necessitate it: good use of the methodology—*i.e.*, good design—is required as well. This chapter focuses on the methodology side of modularity; the design side is discussed a bit here and more in Chapter 6. This chapter looks in particular at documentation for classes. A good system of documentation is important because it describes modules as abstractions apart from any particular implementation. This is central to achieving independence, allowing clients and implementors of modules reason about their code independently. It also is central to achieving good design by making interface designs more tangible and thus easier to evaluate.

There are two units of modularity in our methodology. For instantiators, the unit of modularity is the entire class. Sec. 2.1 describes how to document instance interfaces. For specializers, the unit of modularity is class components. Sec. 2.2 describes the programming conventions for class components, and Sec. 2.3 looks at documentation for specialization interfaces given in terms of class components. These sections also introduce examples used throughout the report.

Although the mechanics of specifying instance and specialization interfaces are similar, the information contained by their specifications is different. Sec. 2.4 looks at this difference.

2.1 Instance interfaces

Chapter 1 explains that classes need two interfaces, one for instantiators and another for specializers. The one for instantiators takes the entire class as the unit of modularity. We draw abstraction boundaries around entire classes by documenting classes with object specifications. Object specifications describe the behavior of objects. The object specification documenting a class describes the behavior of direct instances of the class. (Documenting classes by describing their instances is nothing new and goes back at least to [Hoare72].)

`IntSetISpec`, an example object specification for `IntSet`, is given in Fig. 2.1. This is an informal specification, using the notation from [Liskov86] (formal specifications are described in Chapter 4). `IntSetISpec` illustrates the two basic parts found in all object specifications: the *abstract state*, which describes the state of objects, and the *method specifications*, which describe the behavior of the methods of objects.

The abstract state of `IntSetISpec` is given in the **state field** declaration (in general, there can

```
object specification IntSetISpec { // Documents instance interface

 state field elements; // A mathematical set of integers

 public void addElement(int el);
   // Modifies: this.elements
   // Effects: Adds el to this.elements.

 public boolean removeElement(int el);
   // Modifies: this.elements
   // Effects: Removes el from this.elements, returning true iff
   //   el is in to begin with.

 public IntEnumeration elements();
   // Effects: Returns an enumeration of integers in this.elements.

 public boolean contains(int el);
   // Effects: Returns true iff el is in this.elements.

 ..other methods elided
};
```

Figure 2.1: Instantiator's view of `IntSet`.

be multiple field declarations). These declarations declare *abstract-state fields*. Abstract-state fields are fields of objects much like instance variables, except that they do not exist at run-time. They are fictions created to abstract away from the details of instance variables. These fictional fields do not appear in any code, but they do appear in the specifications of methods.

Methods are specified in terms of *pre-conditions* that must be hold on entry to a method and *post-conditions* that are established on exit. The *requires clause* describes a method's pre-condition, constraining the arguments on which the method is defined. The code calling a method is responsible for establishing the method's pre-condition. None of the methods in `IntSetISpec` have pre-conditions, which means they can be called with any arguments. The *modifies* and *effects clauses* together describe a method's post-condition. The modifies clause constrains the behavior of the method by restricting what it is allowed to change: the method can only change what is listed in its modifies clause. The effect clause describes the behavior of the method, *i.e.*, it describes in what ways the method changes objects and what values the method returns. In Fig. 2.1, the post-condition of `addElement` says that `addElement` modifies the `elements` field of **this** by inserting `el` into it.

2.2 Conventions for extensible state

As discussed in Chapter 1, class components support extensibility for state. In the context of subclassing, extensibility means allowing subclasses to provide their own implementation of some aspect of their superclasses. For example, subclasses can provide their own code for deferred and overridable methods. With class components, the final, overridable, and deferred distinction can be applied to state. When applied to state, final, deferred and overridable are called *representation* *categories*.

Overridable and deferred state is extensible: subclasses can provide their own representations for it. The *overridable state* of a class is state for which the class provides a representation that can be replaced by subclasses. The *deferred state* is state a class assumes exist but for which it provides no representation, depending instead on subclasses to provide representations. *Final state* is not extensible: subclasses cannot provide their own representations for final state but rather must inherit the superclass's representation. Overridable, deferred, and final state are all supported by class components.

An overridable class component is a group of public and protected, overridable methods and a set of private instance variables maintained by them. Only methods in the component may access the instance variables assigned to the component. These methods are called the component's *accessors*. If a method outside the component needs to access the state represented by these instance variables, it must call the accessors of the component rather than directly access the variables. Thus, for example, `removeElement` in `IntSet` (Fig. 1.2) calls `uncache` rather than accessing `c_valid` and `c_val`, and `contains` calls `elements` rather than accessing `els`. As illustrated in by `IntSet2` (Fig. 1.3), which represents the cache using a bit-map, a subclass replaces the representation of an overridable component by overriding *all* accessors of the component with new code that accesses the new representation.

For convenience, an overridable class component may contain *helper methods*, private, final methods that are useful for implementing the component's accessors. For example, in `IntSet2`, the cache component contains the helper methods `c_test` and `c_set` that perform bit-level operations. Helper methods may only be called by methods in their own component and may not be called by methods in other components.

A deferred class component is a group of public and protected, deferred methods. Deferred components are also associated with *deferred state*. Deferred state is assumed to exist by the code of final and overridable methods, but the class provides no representation for it. The final and overridable methods of a class access the deferred state of a component by calling the component's deferred methods, which are also called *accessors*. A subclass provides a representation for deferred state by overriding these accessors with code that access a representation provided by the subclass.

An example of deferred state is given in Fig. 2.2. This figure presents a partial implementation of `Rd` ("reader"), a character input stream inspired by the Modula-3 library [Brown91]. Different subclasses of `Rd` read characters from different sources, *e.g.*, the source of `FileRd` is disk files, while the source of `SocketRd` is network connections. Buffering, using an internal array to facilitate the reading of characters off devices in blocks rather than individually, is important to the performance of readers. The state associated with buffering—and the code that manipulates that state—is included in `Rd` so it can be shared by all subclasses. The state associated with the source a reader is deferred because it is different in different subclasses. This deferred state is accessed by calling the deferred `nextChunk` (see, for example, the code of `getChar`). Subclasses provide a representation for this deferred state by providing code for `nextChunk` that accesses the subclass-provided representation.

Final state is not extensible, so it is a little outside of the topic of this section ("conventions for extensible state"), but we include a discussion of final components for completeness. A final class component is a group of public and protected, final methods and a set of associated instance variables. Although it is possible to use protected and public instance variables for final state, we assume that private instance variables are used (protected and public instance variables are discussed

```
class Rd { // Character input streams

  // Rd class implements source-independent buffering.

  // Deferred component: substate = char's not yet read from underlying source
    protected deferred char[] nextChunk();
      // Returns the next block of characters from the source.
      // Subclasses override this accessor with code that directly accesses
      // the representation of the underlying source of characters.

  // Final component: substate = buffer of characters
    private char[] buffered = new char[0];
    private int cur = 0;

    public final char getChar() throws EofException {
      if (cur == buffered.length) {
        buffered = this.nextChunk();
        cur = 0;
        if (buffered.length == 0) throw new EofException();
      };
      return buffered[cur++];
    };

  ..other methods elided
};
```

Figure 2.2: Partial implementation of `Rd`.

in Chapter 6). In Fig. 2.2, the state associated with the buffer of a reader is final. This state is represented by the instance variables `buffered` and `cur`.

Because the final state of a class cannot be replaced by subclasses, it is safe for all methods of the class to directly access its representation. For example, if the cache of `IntSet` (Fig. 1.2) were final rather than overridable, then the code of `removeElement` could manipulate `c_valid` and `c_val` directly instead of calling `uncache`. Thus, the purpose of grouping of methods into final components is not to enforce implementation restrictions but to help break up a class into smaller, more digestible pieces.

The conventions for extensible state can be summarized in two simple statements:

1. Partition methods and instance variables into final, overridable, and deferred components.

2. Implement each component independent of the the other overridable components in the class.

This second point is important to making an overridable component overridable: if subclasses of class C are going to replace component G and inherit other components, then C's implementation of those other components must be independent of the way C happens to implement G. We have already described one aspect of establishing this independence: the instance variables of a component may be accessed only by the methods in the component. However, independence goes beyond not looking at instance variables, *e.g.*, it also includes not depending on the code of methods. In general, one component is independent of another if it depends only on the *specification* of the other component, not on its implementation. The next section explains how to specify components. The next chapter looks the steps necessary to ensure that the implementation of a component depends only on these specifications and not on implementation details.

2.3 Specialization interfaces

An informal specialization specification for `IntSet` is given in Fig. 2.3. Specifications for specialization interfaces have two parts: an object specification that describes instances of the class, and a *division of labor* that partitions this object specification into class components. The form for the object-specification part is the same as for all object specifications although, as discussed below, the content of the object specifications for the instance and specialization interfaces differ. The division of labor assigns each abstract-state field and method of the specialization interfaces' object specification to one of the interfaces class components. As illustrated in Fig. 2.3, divisions of labor are given in the form of `component` clauses that group together **substate field** declarations, which indicate the abstract-state fields assigned to the component, and the method specifications of methods assigned to the component.

As suggested above, the content of the object specifications given for the instance and specialization interfaces differ. Specifications for the specialization interface are typically more detailed than those for the instance interface. Our theory does not require that the object specifications for these two interfaces differ, but in practice specializers typically need to know more about a class than instantiators do. For example, `IntSetSSpec` is more detailed than `IntSetISpec` in exposing the existence of the membership cache, allowing subclasses to replace it. As another example, `IntSetSSpec` has an invariant while `IntSetISpec` does not, an invariant important to the correct maintenance of the cache. These kinds of differences between the object specifications for the instance and specialization interfaces will be discussed in subsequent sections and chapters.

```
specialization specification IntSetSSpec {

 state field elements; // A mathematical set of integers
 state field cache; // Also a mathematical set of integers

 invariant cache ⊆ elements // All methods must preserve this

 overridable component {
   substate field elements;

   public void addElement(int el);
     // Modifies: this.elements
     // Effects: Adds el to this.elements.

   public boolean removeElement(int el);
     // Modifies: this.elements, this.cache
     // Effects: Removes el from this.elements, returning true iff
     //   el is in to begin with.

   public IntEnumeration elements();
     // Effects: Returns an enumeration of the integers in this.elements.
 };

 overridable component {
   substate field cache;

   public boolean contains(int el);
     // Modifies: this.cache
     // Effects: Returns true iff el is in this.elements.

   protected void uncache(int el);
     // Modifies: this.cache
     // Effects: Removes el from this.cache
 };

 ..other methods elided
};
```

Figure 2.3: Informal specialization specification of `IntSet`.

We did not mention invariants when we introduced object specifications, but any object specification can have an invariant. An invariant describes a constraint on the specification's abstract-state fields that must be established by constructors and preserved by all methods. The invariant of an object specification is an implicit part of all method specifications: it may be assumed on entrance and must be preserved on exit. Thus, for example, `contains` may assume the invariant even though it is not part of the explicit pre-condition of `contains`. Similarly, `removeElement` must preserve the invariant even though it is not part of the explicit post-condition of `removeElement`. Invariants are just one of many different kinds of information that might be put into object specification. Other examples include constraints for establishing history properties [Liskov94] or complexity information for bounding algorithms [Musser96]. We highlight invariants because, as discussed in the next section, they are particularly important in the context of specialization interfaces. For simplicity, we ignore other kinds of information that could be included in object specifications, but our results can be extended to handle additional information.

`IntSetSSpec` has two overridable class components, one associated with the `elements` field and the other associated with the `cache` field. `RdSSpec` (Fig. 2.4), a specialization specification for `Rd`, illustrates documentation for final and deferred components. Documentation for class components lifts the programming conventions for components to the level of abstract state. Even though deferred components do not have an implementation, they do have abstract state. Thus, at the abstract level, rather than at the level of instance variables, it becomes possible to describe the state assigned to deferred components.

2.4 Abstract representations

Instance and specialization interfaces are both documented using object specifications. However, as indicated earlier, the specializer's object specification is more detailed than the instantiator's. The specializer's object specification has a more detailed view of the class's state and includes specifications for protected methods. For example, `IntSetSSpec` reveals the existence of the membership cache to specializers and also includes the protected method `uncache`. Similarly, `RdSSpec` reveals that the source of a reader is split into the components `buffered` and `ondevice` and also includes the protected method `nextChunk` (*c.f.* the instance specification in Fig. 2.5). The extra details found in the specializer's object specification describe abstract interfaces that class components use to interact with each other.

Again, one way in which the specializer's object specification is more detailed than the instantiator's is by having a more detailed view of the class's abstract state. We call this more detailed view the *abstract representation*. This term emphasizes that it is at a level of abstraction between the fully-abstract state given in the instance specification and the instance variables manipulated by code. As a slogan, we say that the abstract representation should expose the implementation strategy without exposing implementation details. Implementation strategies includes internal mechanisms such as caching (*e.g.*, in `IntSet`), buffering (*e.g.*, in `Rd`), or the fact that some structures are sorted. Such strategies are "implementation details" as far as instantiators are concerned, but they are important in terms of the interactions among components.

The other way in which the specializer's object specification is more detailed than the instantiator's is by including protected methods. Protected methods arise where a class component maintains aspects of a class's state that is visible in the specializer's object specification of the class but not in the instantiator's. Often, such components include accessors to allow other components to

```
specialization specification RdSSpec {

  state field buffered; // Sequence of characters
  state field ondevice; // Sequence of characters

  final component {
    substate field buffered;

    public char getChar() throws EofException;
      // Modifies: this.ondevice, this.buffered
      // Effects: If this.buffered and this.ondevice are both empty,
      //   signals EOF.  Otherwise, first may (but may not) move a prefix
      //   of this.ondevice onto the end of this.buffered, then removes
      //   and returns the first character of this.buffered.
  }

  deferred component {
    substate field ondevice;

    protected char[] nextChunk();
      // Modifies: this.ondevice
      // Effects: Removes and returns a prefix of this.ondevice.
      //   Returns the empty sequence only if this.ondevice is empty.
  }

  ..other methods elided
};
```

Figure 2.4: Informal specialization specification of Rd.

```
object specification RdISpec {

  state field source; // Sequence of characters

  public char getChar() throws EofException;
    // Modifies: this.source
    // Effects: If this.source is empty, signals EOF.  Otherwise,
    //   removes and returns the first character of this.source.

  ..other methods elided
};
```

Figure 2.5: Informal instance specification of Rd.

manipulate this state in ways not available to instantiators. Protecting accessors hides them from instantiators but exposes the full interface to subclasses. For example, the `uncache` accessor of `IntSet` must be visible to subclasses because subclasses that replace the `elements` component need to call it and because subclasses that replace the `cache` component need to provide their own implementation for it. At the same time, this accessor should be hidden from instantiators because the cache is irrelevant to instantiators. A similar argument applies to the `nextChunk` accessor of `Rd`.

Another difference between the instantiator's and specializer's object specifications is the role played by invariants. In particular, in classes constructed from class components, invariants on the abstract representation serve in lieu of representation invariants. We call these invariants *abstract representation invariants* not only because they are invariants on abstract representations but also because of their role as surrogates for representation invariants.

Efficient implementations of methods must be able to make assumptions about the relationships among different parts of an object's state. For example, in `IntSet`, `contains` assumes that the cache of a set is a subset of the set's elements, and `removeElement` assumes that `els` contains no duplicate entried (*i.e.*, no number is stored in `els` more than once). In the context of classical data abstractions, both of these assumptions would be expressed as representation invariants. However, as discussed in the next few paragraphs, in the context of subclassing, not all such assumptions can be captured as representation invariants.

Even in the context of subclassing, representation invariants can still be used to capture assumptions that relate state within a component such as the "no duplicates" assumption from above. Such invariants can be established by data-type induction, *i.e.*, by making sure that the methods that have access to the instance variables preserve the invariant. In our example, the no duplicates property can be established by ensuring that `addElement`, `removeElement`, and `elements` all preserve it.

However, representation invariants cannot be used to capture assumption that relate state from multiple components such as the cache-validity property. This is because subclasses might replace the representations of some of the state involved. For example, consider a subclass of `IntSet` that replaces the `elements` component and inherits the `cache` component. The inherited code for `contains` still assumes that `cache` is a subset of `elements`, but the representation of `elements` has been replaced. Instead of expressing multi-component properties in terms of the concrete representation, they must be expressed in terms of the abstract representation. Thus, we see that the invariants on the abstract representation serve in lieu of invariants on representations to express properties that relate state from multiple components.

2.5 Summary

Classes have two interfaces, one for instantiators and once for clients. The unit of modularity for instance interfaces is the entire class. They are documented using object types.

The unit of modularity for specialization interfaces is class components. Class components are a programming convention that support the overridable, deferred and final representation categories for the state of classes. Under this convention, the methods and instance variables of classes are partitioned into final, overridable, and deferred components, and each component is implemented independently of the implementations of other overridable components.

The class components making up a class are documented by giving an object specification together with a division of labor that divides the state and methods of the object specification into class components. Specialization specifications need their own object specification and cannot use the instance specification. This is because the specialization specification reveals more detail, details pertaining to the interfaces between class components. When designing specialization interfaces, designers need to think in terms of an implementation strategy, *i.e.*, a level of abstraction above the implementation but below the instance specification.

Chapter 3

Validating classes

Validation is any activity intended to increase our confidence that a class behaves as intended. Validation typically consists of some combination of testing and reasoning. Testing involves placing the class in a particular context and seeing if it behaves as expected. Reasoning involves inspecting the implementation of a class and arguing that it will behave correctly in all possible contexts. Reasoning can be done formally or informally. Much formal reasoning can be done mechanically. Mechanical reasoning can range from simple type checks, to anomaly checking, to full, formal verification.

All forms of validation depend on some notion of what it means for a class to "behave as intended." We call this notion the *validation criterion*. Our starting point for defining such a criterion is the slogan "a class is correct if it meets its specification." Sec. 3.1 looks at what it means for a class to meet both its instance and specialization specifications. The Sections 3.2 and 3.3 apply this criterion to informal reasoning about the correctness of class relative to its specialization interface. Sec. 3.4 explains how the correctness of a class relative to its instance interface can be deduced from correctness relative to its specialization specification.

3.1 Validation criterion

Recall that classes have two specifications, one for instantiators and one for specializers. The instance specification of a class consists of an object specification. A class implements its instance specification if all instances of the class behave as described by the specification.

The specialization specification consists of an object specification together with a division of labor. The validation criterion for a class against its specialization specification is two-fold. First, instances of the class must behave as described by the object-specification part of the specialization specification. Second, each class component defined by the division of labor must be implemented *independently* of the implementations of the other overridable components in the class. This means that each class component can depend only on the specifications of other class components; it cannot depend on the implementations of methods nor on the representation of state assigned to other components. If components are independent, then subclasses can replace some of the components without breaking the others.

The following implementation of `removeElement` is behaves correctly but is not independent of the representation of another component:

```
public overridable boolean removeElement(int el) {
  boolean result = els.removeElement(el);
  if (result) c_valid := false; // Bug: accesses rep of cache!
  return result;
};
```

This code would be fine if the only validation criterion were that a class implement the object-specification part of its specialization specification. However, this code is not valid because it depends on the representation of the cache and thus is not independent of the implementation of the cache component.

The following code also behaves correctly but is also not valid, this time because it is not independent of the implementation of a method it calls:

```
protected overridable void uncache(int el) {
  c_valid = false; // Invalidate even if el not in cache
};

public overridable boolean removeElement(int el) {
  boolean result = els.removeElement(el);
  if (result) this.uncache(2); // Assume that uncache ignores the
  return result;               //    value of its argument.
};
```

This version of `uncache` always invalidates the cache no matter what the value of its argument. Although not the most efficient thing to do, it is still correct. This version of `removeElement` takes advantage of the fact that `uncache` ignores its argument. Like the version of `removeElement` that accessed `c_valid`, this version behaves correctly but is not valid because it depends on the implementation of `uncache`.

3.2 Reasoning about local components

The implementation of a class contains two kinds of components: components implemented locally and components inherited from a superclass. This section considers reasoning about local components. Local components include superclass components overridden by the class, deferred components of the superclass implemented by the class, and new methods defined by the class. We want to reason about the correctness of the local components of a class relative to the class's specialization specification.

The implementation of a local component consists of a set of instance variables that represent the state of the component and code for the component's methods. Programmers reason about this code—both formally and informally—in pretty much the same way they reason about code for classical data abstractions (see, *e.g.*, [Liskov86],[Dahl92]). This reasoning involves inspecting the code to make sure that it does what it is supposed to. We do not review this inspection process here. Instead, we explain the additional steps necessary to ensure that the code of a component is independent of other, overridable components in the class.

3.2.1 Calling methods

The first aspect of being independent of other components is to reason about calls to overridable methods in terms of specifications rather than implementations. Because calls to overridable components are reasoned about in terms of their specifications, these calls will still work as expected when subclasses replace the components with new code implementing the same specification.

3.2.2 Accessing state

Another aspect of being independent of other components is not depending on the representations of their state. This means that if the code in one components needs to access the state assigned to another, overridable component, it should do so by calling the methods of the other component rather than by accessing its instance variables. When done through methods, accesses of the state of an overridable component will still work as expected when subclasses replace the component.

Binary methods, generally a problem (see, *e.g.*, [Liskov93] and [Bruce96]), must be treated with care. A binary method of class `C` is a method that takes one or more arguments of type `C` in addition to **this**. In most languages, the code in binary `C`'s methods has privileged access to all arguments of type `C`, not just to **this**. In particular, this code can access the private instance variables of those arguments. This privileged access supports efficient implementations of some data types, but it must be used sparingly to to achieve independence.

Consider the following class:

```
class C {
  // Overridable ``count'' component
    private int m_count;

    public overridable getCount() { return m_count; }
    public overridable addCounts(C o)
      { return m_count + o.getCount(); }

  ...
}
```

The code for `addCounts` can safely access `m_count` of **this**. However, this code should not access `m_count` for `o`. This is because the class implementing `o` may be a subclass of `C` that has replaced the `count` component with code that does not use `m_count` to represent `count`. Instead of accessing `m_count` for `o`, `addCounts` should call `getCount` instead.

In short, instance variables representing overridable state should only be accessed for **this** and not for other arguments. For *all* of its arguments, a binary method *can* safely access the instance variables representing *final* state.

3.2.3 Assuming invariants

Invariants are important in reasoning about the correctness of method code [Liskov86]. For example, the code of `removeElement` (Fig. 1.2) assumes no duplicates in **this**.`els`, *i.e.*, that no element appears in **this**.`els` twice. The code of `contains` assumes that an element of the `cache` is also an element of `elements`.

In the context of class components, there are two kinds of implementation invariants. First, there are invariants that relate state within a single component, *e.g.*, the "no duplicates" invariant. Such an invariant can be established by showing that it is preserved by the code of each of the component's methods. For example, `addElement`, `removeElement`, and `elements` all preserve the no duplicates invariant. Because only the methods in a component access the instance variables of the component, only these methods need to be checked to establish an invariant on these instance variables.

The second kind of invariant relates state assigned to different components, *e.g.*, the `cache` validity invariant. As explained in the previous chapter, such invariants cannot be established directly on the instance variables. In place of such invariants, programmers must instead use abstract representation invariants, *i.e.*, invariants on the abstract state of the object specifications of specialization specifications.

3.2.4 Abstracting state

When choosing a representation for a class and coding the class's methods, the implementor has in mind an *abstraction function*, a relationship between the class's instance variables and the class's abstract state. For classical data abstractions, the abstraction function maps the entire representation to the entire abstract state. In the context of class components, each component needs its own *subabstraction function*. These subabstraction functions map the component's instance variables to the abstract state assigned to the component.

For example, `IntSet` (Fig. 1.2) needs two subabstraction functions, one for the `elements` component and the other for the `cache` component. The function for the `elements` component returns the set consisting of the elements of `els`. The function for the `cache` component returns the empty set when `c_valid` is false and the singleton set consisting of `c_val` when `c_valid` is true.

Subabstraction functions are central to formal verification of classes and will be discussed further in Chapter 4. However, even when classes are not formally verified, subabstraction functions are useful for informal reasoning, and it is a good idea for implementors to document the subabstraction function of each class component.

3.3 Reasoning about subclasses

Root classes are classes without superclasses, such as `IntSet` in Fig. 1.2. All non-deferred components of a root class are local, so root classes are easy to reason about: just reason about each local component as described in the previous section.

Subclasses are a little trickier. The validation criterion described in Sec. 3.1 applies equally to root classes and subclasses. Local components of a subclass—*i.e.*, superclass components overridden by the subclass, deferred components of the superclass implemented by the subclass, and new local components—are still reasoned about as described in the previous section. However, unlike root classes, subclasses contain inherited components. Also, unlike root classes, subclasses can use **super** to call superclass versions of methods.

This section discusses reasoning about inherited methods and **super**. It assumes that the subclass's set of abstract-state fields is the same as the superclass's. The next chapter handles the case when the sub- and superclass have different abstract-state fields.

3.3.1 Reusing superclass code

Subclasses can reuse code of their superclasses, both by inheriting components and by using **super** to call superclass versions of methods. Superclass code reused by the subclass may in turn call code provided by the subclass; for example, an inherited method may call an overridden one. Code provided by the subclass must meet assumptions made about it by inherited superclass code. (In addition to this issue, reasoning about **super** entails other issues discussed in Sec. 3.3.4.)

Superclass code makes two assumptions that subclasses need to respect. First, superclass code assumes that the superclass invariant is preserved. If this code is going to work when reused by the subclass, the subclass will have to preserve the superclass's invariant. For example, `contains` depends on the cache validity invariant; a subclass that inherits `contains` must preserve this invariant. To ensure that the subclass (and its subclasses) will preserve the superclass's invariant, we require that the subclass's invariant imply the superclass's.

The second assumption made by superclass code is that the methods of **this** implement the specifications given to them in the superclass. For example, the correctness of `contains` in Fig. 1.2 depends on the behavior of `elements`. If a subclass replaces `elements` and inherits `contains`, it should replace `elements` with code that does what `contains` expects it to. To ensure that the subclass's code for methods implements the behavior expected by superclass code, we require that the subclass specifications of overridable and deferred methods imply their specifications in the superclass. This rule applies to all methods for which the subclass *can* provide code, not just to methods for which the subclass actually does provide code. Applying this rule to all methods ensures that subclasses of the subclass, which can replace methods not replaced by the subclass, also respect assumptions made by the superclass.

3.3.2 Inherited methods

Local components are reasoned about by inspecting their code. However, recall that a goal of this report is to allow specializers to build subclasses without looking at the code of superclasses. We need a way to ensure that the code of inherited methods behaves as described by the subclass's specification without actually looking at this code.

We are assuming that the superclass implements its specification. This means that inherited methods behave as described by the *super*class's specification. If this superclass specification implies the subclass specification, then the subclass specification will also describe the inherited code. Thus, we ensure that inherited methods meet their subclass specifications by requiring that their superclass specifications imply their subclass specifications.

This implication is opposite the implication in the previous subsection. There, subclass specifications imply superclass specifications to ensure that subclass-provided code behaves as expected by the superclass. Here, superclass specifications imply subclass specifications to ensure that superclass-provided code behaves as described by the subclass. Note that the implications of this subsection and the previous one both apply to inherited, overridable methods. This means that the specifications of such methods in the sub- and superclass interfaces end up being logically equivalent.

A subclass that inherits a component must inherit the component as a whole—it may not separate the elements of the component. For example, a subclass of `IntSet` that inherited the `element` component may not put `addElement` and `removeElement` into separate class components. If a subclass did split them up, subclasses of this subclass could override `addElement`

Method kind	Implication(s) between specifications
Final	superclass $\Rightarrow$ subclass
Inherited overridable	superclass $\Leftrightarrow$ subclass
Overridden overridable	superclass $\Leftarrow$ subclass
Deferred	superclass $\Leftarrow$ subclass

Table 3.1: Behavioral subclassing.

without overriding `removeElement`, which would be an error. When a subclass replaces a component, it *may* break it up. In our example, a subclass that replaces the `elements` component could put `addElement` and `removeElement` into separate class components.

3.3.3 Behavioral subclassing

Collectively, we refer to the rules from the previous two subsections as *behavioral subclassing*. In short, these rules are:

1. The specifications of overridable and deferred methods in the subclass must imply their specifications in the superclass, and the invariant of the subclass must imply the invariant of the superclass.

2. The specifications of inherited methods in the subclass must be implied by their specifications in the superclass.

3. Components inherited from the superclass may not be broken up by the subclass.

These rules ensure that superclass code reused by the subclass behaves as expected. Table 3.1 summarizes the implications between the specifications of methods in the sub- and superclass specifications.

To understand how behavioral subclassing works, consider a specialization specification that is the same as `IntSetSSpec` (Fig. 2.3) in every way except for the specification of `elements`:

```
specialization specification IntSetSSpec2 {
  ...
  overridable component {
    ...
    public IntEnumeration elements();
      // Effects: Returns an enumeration of the integers in
      //   this.elements.  This enumeration yields these integers
      //   in ascending order.
  };
  ...
};
```

This specification for `elements` is stronger than the one in `IntSetSSpec`: it returns an enumeration that yields its elements in ascending order. If `IntSetSSpec` is the specialization specification of `IntSet`, could `IntSetSSpec2` be the specialization specification of a behavioral subclass of IntSet? The answer is "yes"—but only for subclasses that override the element component of IntSet. Because of rule (1), the subclass's specification of elements can be stronger than the superclass's only when the subclass overrides elements. This makes sense: Subclasses that inherit the element component cannot be specified by IntSetSSpec2 because the inherited code for elements does not (necessarily) meet this stronger specification. A subclass can strengthen the specification of superclass methods only when the subclass provides its own code for that method—code that meets the stronger specification.

Now consider another specialization specification, also the same as IntSetSSpec except for the specification of elements:

```
specialization specification IntSetSSpec3 {
  ...
  overridable component {
    ...
    public IntEnumeration elements();
      // Effects: Returns an enumeration of *some* of the integers
      //   in this.elements.
  };
  ...
};
```

This specification for elements is weaker than the one in IntSetSSpec: it returns an enumeration that yields a subset of the elements in **this**.elements. Could this be the specialization specification of a behavioral subclass of IntSet? The answer here is "no" because contains—and perhaps other methods of IntSet—depend on the stronger behavior of elements. A subclass cannot weaken the specification of superclass methods because inherited methods may depend on the stronger behavior. (By adding more information to specialization specifications, we could allow subclasses to weaken the specification of superclass methods. Such information might include, *e.g.*, the fact that contains is the only method that calls elements. However, we do not feel that the ability to weaken specifications is worth the extra complexity it would add to specifications and to reasoning.)

3.3.4 Shadowing components

Half way between inheriting a component on the one hand and completely replacing it on the other is to *shadow* a component. When a class shadows a superclass component, it replaces some of the component's methods but does *not* replace the representation of the component's substate. The class may also add new methods and new substate to a shadowed component.

CIntSet (Fig. 3.1) extends IntSet to count the number of times addElement is called. CIntSet shadows the elements component of IntSet: it does not replace the representation of elements but it extends the state of the component to include a counter. It also adds a new method. Notice that CIntSet uses **super** to invoke the superclass version of addElement. Shadowed components typically use **super** in this way.

The convention for shadowed components is as follows:

- The subclass does not provide a representation for the substate of shadowed components. Instead, the superclass's representation is used. In CIntSet, for example, the representation of elements is not replaced.

```
class CIntSet extends IntSet {

 // Shadowing ``elements'' component of IntSet:
  private int m_addCount;

  public void addElement(int el) {
     m_addCount += 1;
     super.addElement(el);
  };

  public int addCount() {
     return m_addCount;
  };
};
```

Figure 3.1: Subclass of `IntSet` that shadows a component.

- Because the superclass represents this substate using instance variables not visible to the subclass, the subclass cannot directly access the representation of this state. Instead, the subclass manipulates this state using **super** to call superclass accessors. In our example, `addElements` calls the superclass version of `addElements` to update the `elements` field.
- The subclass can add new substate fields to a shadowed components. These new fields *are* represented using subclass instance variables, and a local subabstraction function defines the new substate fields in terms of these instance variables. In our example, a new `addCount` field is added to the state of the `elements` component.
- The subclass need not override all methods of shadowed components. In our example, `removeElement` and `elements` are both inherited.
- A subclass cannot break up a shadowed component. That is, the component in the subclass must include at least the methods and substate fields of component in the superclass, although it may contain more. Our example follows this rule.

When **super** is used to call the superclass version of a method, the superclass's specification of the method is used to reason about the call.

Super can only be used in shadowed components. Further, a shadowed component cannot call superclass versions of methods in other components, only of methods in itself. Other uses of **super** are not modular. For example, imagine that a subclass of `IntSet` inherits the `elements` component but overrides the `cache` with a version of `contains` that called the `elements` method via **super**. This subclass would work fine. However, if a subclass of this subclass replaced the `elements` component and inherited the `cache` component, then the `cache` component would no longer work because its call to `elements` would invoke the wrong version.

More general mechanisms exist for invoking overridden code. For example, in C++, the invocation:

```
o->C::m(..arguments..)
```

invokes the code for method `m` provided by class `C`, where `C` is an arbitrary superclass of `o`. Whereas **super** allows the invocation of only the immediate superclass's version of a method, more general mechanisms such as the one in C++ allows the invocation of any superclass's version. The methodology defined by this report supports the invocation of only the immediate superclass's version of a method, not an arbitrary superclass's version. This restriction follows from the philosophy that subclassing is an implementation issue: the superclasses of a class is not considered to be part of a class's specification, so subclasses of the class should not be able to depend on those superclasses.

3.4 Reasoning about instance interfaces

So far, this chapter has looked at showing that a class implements its specialization specification. Reasoning about a class also includes showing that it implements it instance specification. The correctness of classes relative to their instance specifications cannot be reasoned about by inspecting code because the code of inherited methods is not available for inspection. Instead, we reason about the correctness of a class relative to its instance specification by relating the instance specification to the specialization specification.

The validation criterion for specialization interfaces says that a class must implement the object-specification part of its specialization specification. This fact can be used to reason about instance interfaces: a class will implement its instance specification if its instance specification is a behavioral supertype of the object-specification part of its specialization specification.

One object specification is a *behavioral supertype* another if it correctly describes all objects described by the other. The term super*type* is used because object specifications are used to document object types. A behavioral supertype is a "more general" specification, *i.e.*, it describes more objects. The behavioral subtype is "more specific." Behavioral subtyping is discussed in detail in [Liskov94]; formal rules for behavioral subtyping are given in Sec. 4.1.2. For the purposes of this section, a simple rule suffices: a subtype must have a more detailed view of an object than its behavioral supertypes do. This has three implications: the subtypes has more methods; the subtype has more abstract-state fields; the subtype has a stronger invariant and stronger method specifications.

For example, the object specification in Fig. 2.1 is a behavioral supertype of the object specification part of the specialization specification in Fig. 2.3. This is because the specialization specification reveals more methods (`uncache`) and more state (the cache) and has a stronger invariant. Thus, if we assume that `IntSet` (Fig. 1.2) correctly implements the specialization specification in Fig. 2.3, then it follows that it correctly implements the instance specification in Fig. 2.1.

In Java and most languages, the "class" construct defines both what this report calls a class *and* what this report calls an object type. That is, Java class C defines a type that is also named C. Inside the code of class C, **this** can be used wherever the type C is expected. In languages like this, good design practice requires that the instance specifications of all subclasses of C be behavioral subtypes of the instance specification of C. For example, code similar to the following appears in the ET++ `View` class (ET++ is a GUI framework [Weinand95]):

```
public ViewStretcher createStretcher() {
  DocView dv = this.getDocView();
  if (dv == null) return CreateViewStretcher(this);
  else return CreateViewStretcherWithRect(this, dv.contentRect());
};
```

The code of `createStretcher` passes **this** as an argument to other routines. These other routines expect their first arguments to meet the instance specification of `View`. The correctness of `createStretcher` assumes that the instance specifications of all subclasses of `View` are behavioral subtypes of the instance specification of `View`.

Behavioral subtyping of instance specifications does not follow automatically from behavioral subclassing. That is, if class D is a behavioral subclass of C, it does not follow that the instance specification of D is a behavioral subtype of the instance specification of C. In languages like Java in which D is treated like a subtype of C, programmers must explicitly check to ensure that the instance specification of D is a behavioral subtype of that of C.

3.4.1 Instance invariants

Note that the instance interface for `IntSet` is simpler than its specialization specification. The instance specification hides the membership cache. Unfortunately, behavioral subtyping by itself is not always enough to allow instance specifications to hide abstract-state fields of the specialization interface.

In `IntSet3` (Fig. 3.2), `fastRemove` removes elements from `elements` without invalidating the cache. This allows `removeElements` to remove multiple elements while invalidating the cache only once. Cache validity is not an invariant of `IntSet3` because it is not preserved by `fastRemove`. As a result, cache validity must be an explicit pre-condition of `contains`. We still would like to use the simple instance specification in Fig. 2.1 for `IntSet3`, but we cannot because behavioral subtyping will not allow us to get rid of the precondition of `contains`.

This is an example of a larger pattern not uncommon in object-oriented programs. Protected methods like `fastRemove` are allowed to “get inside” the implementation and do things that instantiators cannot do. In order to specify such methods, specialization interfaces must expose fields like `cache`. However, subclasses that call these protected methods usually “fix things” before returning, so fields like `cache` can be hidden in instance specifications. *Instance invariants* support this pattern.

Instance invariants are invariants that must be established by constructors and preserved by all *public* methods of an object but that may not be preserved by all protected methods of an object. For example, the cache validity invariant ($\mathbf{this}.c \subseteq \mathbf{this}.s$) is an instance invariant of `IntSet3`. Instance invariants always hold when an instantiator calls a public method, so they can be dropped from the preconditions of methods in instance specifications.

In general, let S be the object-specification part of a specialization specification, T be an instance specification, and I be an instance invariant of S. Let S' be the object specification that results from taking S and dropping all preconditions that are implied by I (*e.g.*, dropping the cache-validity precondition of `contains`). T is a valid instance specification for classes implementing S if S' is a behavioral supertype of T. This is just what is necessary to drop the `cache` field from the instance specification of `IntSet3`.

(In the formal notation of the next chapter, instance invariants are used to augment behavioral subtyping by augmenting the precondition rule as follows:

- Precondition: $I[\mathbf{this}^{pre}/\mathbf{this}^{\rho}] \wedge T.pre_m[A(\mathbf{this}^{pre})/\mathbf{this}^{pre}] \Rightarrow S.ospec.pre_m$

This augmented rule lets the instance specification drop preconditions that are implied by the instance invariant.)

```
class IntSet3 { // Alternative set implementation

 public void removeElements(int els[]) {
   // Modifies: this.elements, this.cache
   // Effects: Empties the cache and removes elements of els from this.elements.
   this.invalidate();
   for(IntEnumeration e = this.elements(); e.hasMoreElements(); )
      fastRemove(e.nextElement());
 };

 // ``elements'' component (overridable):
    private IntVector els = new IntVector();

    public overridable void addElement(int el) {
      // Modifies: this.elements
      // Effects: Adds el to this.elements
      ...same as Fig. 1.2
    };

    protected overridable boolean fastRemove(int el) {
      // Modifies: this.elements (does not change cache!)
      // Effects: Removes el, returning true iff el already in
      return els.removeElement(el);
    };

    public overridable IntEnumeration elements() {
      // Effects: Returns an enumeration of integers in this.elements
      ...same as Fig. 1.2
    };

 // ``cache'' component (overridable):
    private int c_val;
    private boolean c_valid = false;

    public overridable boolean contains(int el) {
      // Requires: this.cache ⊆ this.elements
      // Modifies: this.cache
      // Effects: Returns true iff el is in this.elements.  May update the cache,
      //   but will ensure that the new this.cache is a subset of this.elements.
      ...same as Fig. 1.2
    };

    protected overridable void invalidate();
      // Modifies: this.cache
      // Effects: Empties the cache
      c_valid = false;
    };
};
```

Figure 3.2: Alternative integer-set class.

In languages like Java that assume subclasses are subtypes, instance invariants cannot be established by data type induction. That is, they cannot be established by inspecting the constructors and public methods to see if the invariant is preserved. This problem can be fixed by making instance invariants part of specialization specifications, just like invariants must be made part of object specifications in the presense of subtyping [Liskov94]. That is, a specialization specification is now an object specification, a division of labor, *and* an instance invariant. The rules for behavioral subclassing are augmented to require the subclass's instance invariant implies the superclass's.

3.5 Summary

Validation is the broad process of increasing our confidence that a class behaves as intended. It includes such activities as testing, type checking, informal reasoning, and formal verification. All forms of validation depends on a validation criterion that defines what it means for a class to "behave as intended." The validation criterion for specialization interfaces has two aspects. First, functional correctness: instances of the class must implement the object-specification part of a class's specialization specification. This is the same as the classical validation criterion for data abstractions. Second, independence: the implementation of each class component must be independent of the implementations of other overridable components. This independence allows subclasses to replace overridable components without breaking inherited code.

Correctness of local components is reasoned about much in the same way correctness of classical data abstractions is reasoned about. To ensure independence of components, this classical reasoning has to be augmented by extra concerns:

- Calls to other methods in the class must be reasoned about in terms of their specification, not their implementations.
- One component must access the state of another overridable component by calling the methods of the other component, not by directly accessing instance variables.
- Abstraction functions must map on a per-component basis, *i.e.*, they must map the instance variables of each component to the abstract state assigned to the component. (The next chapter looks at abstraction functions in more detail.)
- Representation invariants can only relate instance variables within the same component. Invariants involving state from multiple components must be handled using abstract representation invariants.

Subclasses reuse superclass code. The rules of behavioral subclassing are needed to ensure (1) that subclass-provided code meets assumptions made about it by the reused code, and (2) that the subclass specification of inherited methods actually describes the inherited code. The two basic rules that ensure these are (1) the subclass invariant and specifications of overridable and deferred methods must imply their superclass equivalents, and (2) the superclass specifications of inherited methods must imply their subclass specifications.

With **super**, a subclass can shadow an overridable class component. In this case, the superclass retains responsibility for representing the substate assigned to the component. The subclass can add substate fields to a shadowed component. In this case, a local subabstraction function defines the new substate fields in terms of instance variables provided by the subclass.

The correctness criterion for a class's instance interface is the classical criterion that the class correctly implement the behavior described by the class's instance specification. Because of inherited methods, the code of a class cannot be verified directly against its specification. Instead, the correctness of a class relative to its instance specification is deduced from the correctness of the class relative to its specialization specification. In particular, the class implements its instance specification if the instance specification is a behavioral supertype of the object-type part of the class's specialization specification. The behavioral supertype relation can be extended with instance invariants to help hide fields of the abstract representation from instantiators.

Chapter 4

Formal specification and verification

This chapter formalizes the results of the previous two chapters. It presents formal specifications for specialization interfaces and describes how to verify classes against specialization specifications. We present these formal results for two reasons. First, they are useful in their own right. As one tries to increase the precision of informal documentation, it tends to get long-winded and, as a result, confusing. Formal specifications, on the other hand, are simultaneously precise, clear, and concise. In addition, formal specifications can be mechanically checked both for syntax and for certain semantic "goodness" properties [Guttag93], helping designers to catch mistakes. Formal verification, where it is possible, greatly improves confidence in the correctness of code. Even where verification against full specifications is not cost-effective, verification against partial specifications has proven useful [Detlefs96]. The second reason to pursue formal specifications and verification is that it helps us evaluate our informal results. By formalizing our informal techniques in a manner that is sound and elegant, we increase our confidence in the soundness of and, more generally, in the "goodness" of the informal techniques.

The first two sections of this chapter describe formal specifications for specialization interfaces. The first section presents a formal model for object specifications. Sec. 4.2 describes how object specifications are combined with division of labor specifications and also with constructor specifications to formally specify specialization interfaces. The following three sections describe the procedure for verifying classes against specialization specifications. This procedure has two steps. The first step is checking that the subclass is a behavioral subclass of its superclass. Sec. 4.3 formalizes the informal rules of behavioral subclassing described in Sec. 3.3.3. The second step is verifying the code of local class components, *i.e.*, components for which the class provides its own code. This involves both checking for functional correctness and for independence of component implementations. Sec. 4.4 shows how to verify normal class components; Sec. 4.5 describes verification for shadowed components. Sec. 4.6 ends this chapter by discussing specification and verification of constructors.

4.1 Formal object specifications

This report assumes the specification model for objects described in [Liskov94]. Like many models, this model assumes that program states include an environment that maps variables to object identifiers and a store that maps object identifiers to values. *State* is the the space of program states.

If ρ is a *State*, then:

- $\rho.env : Var \rightarrow Obj$
- $\rho.store : Obj \rightarrow Val$

where *Var* is the set of program variables, *Obj* is the set of object identifiers, and *Val* is the set of values that objects can take on. Given a variable x and a state ρ, x^ρ denotes $\rho.store(\rho.env(x))$, the value of x in state ρ. The store of a state is a partial function defined only on those objects that are allocated in that state; the set of allocated objects in state ρ, $alloc(\rho)$, is the domain of the $\rho.store$.

An object specification T has a number of components:

- $T.oids$, a subset of *Obj*, is the identifiers of objects subsumed by T.
- $T.sort$, a subset of *Val*, is a space of values, called a *sort*, defining the underlying values that objects subsumed by T can take on. T objects cannot always take on all of these values; the invariant of T limits the possibilities.
- $T.methods$ is a set of identifiers containing the names of T's methods. $T.pre_m$ is the precondition of method m and $T.post_m$ is its postcondition. Pre- and postconditions are expressed as boolean-valued terms with free variables **this**, *pre*, and, in postconditions, *post*. The variable **this** is the implicit "self" argument to the method, *pre* is the state before the method is called, and *post* is the state after the method is called. Preconditions are predicates on *pre*, and postconditions are relations on (*pre*, *post*)-pairs. Pre- and postconditions also contain free *Var* variables for the formal arguments of the method and the special free *Val* variable **result** for the result of the method.
- $T.inv$ is an invariant that all T objects must obey. $T.inv$ is a boolean-valued term containing the free variable **this** ranging over *Var* and ρ ranging over *State*. The invariant is interpreted as an axiom constraining allowable program states:

  $$\forall\, \mathbf{this} : T.oids,\ \rho : State \cdot \mathbf{this} \in alloc(\rho) \Rightarrow T.inv$$

  As in informal specifications, the invariant of T is considered an *implicit* part of the specification of T's methods. That is, a method may assume the invariant even if it is not implied by the method's precondition and must preserve the invariant even if it is not implied by the method's postcondition.

4.1.1 Example

An object specification for the instance interface of `IntSet` is given in Fig. 4.1. This specification employs a surface syntax that is close to the syntax used in the previous specification for informal specification.

Our object specifications use Larch Shared Language (LSL) *traits* to define sorts and function symbols used in specifications [Guttag93]. `IntSetISpec` in Fig. 4.1 uses the trait `Set`. The `Set` trait is given in Fig. 4.2. The `Set` trait first gives signatures for *insert*, *delete*, and other common functions on sets. Next, it asserts axioms that define these functions. The **generated by** axiom states that all sets can be generated from just the empty set and the *insert* function. When a specification

```
object specification IntSetISpec {  // IntSet instance specification

  uses Set(Int, IntSet);
  state is IntSet; // defines IntSet.sort

  public void addElement(int el) {
    Modifies: this
    Ensures: this^post = insert(el, this^pre)
  }

  public boolean contains(int el) {
    Ensures: result = el ∈ this^pre
  }

  ..other methods elided
}
```

Figure 4.1: Partial object specification for instance interface of `IntSet`.

uses a trait, it imports the definitions of that trait. A specification can rename symbols imported from a trait. For example, when the `IntSet` class uses the `Set` trait, it renames the sort `E` to `Int` and the sort `C` to `IntSet`, meaning `IntSet` (the sort) will be a set of `Int`. As this example indicates, we use separate name spaces for sorts and for classes.

4.1.2 Behavioral subtyping

This report assumes the behavioral subtype relationship defined in [Liskov94]. One object specification is a behavioral subtype of another if all objects described by the first specification are also described by the second.

A sufficient condition for object specification S to be a behavioral subtype of object specification T is the existence of an abstraction function A from $S.sort$ to $T.sort$ such that:

- Invariant rule: $S.inv \Rightarrow T.inv[A(\mathbf{this}^{\rho})/\mathbf{this}^{\rho}]$
- Method rule: For all $m \in T.methods$:
  - Pre-condition: $T.pre_m[A(\mathbf{this}^{pre})/\mathbf{this}^{pre}] \Rightarrow S.pre_m$
  - Post-condition:

$$S.post_m \Rightarrow T.post_m[A(\mathbf{this}^{pre})/\mathbf{this}^{pre},\, A(\mathbf{this}^{post})/\mathbf{this}^{post}]$$

The reversed implication for pre-conditions reflects their contravariant nature.

4.2 Formal specialization specifications

Specialization interfaces are specified with an object specification, a division of labor, and constructor specifications:

$$SpecializationSpec = ObjectSpec \times LaborDiv \times ConsSpecs$$

`Set(E, C):` **trait**
  includes `Integer %% Trait defining the sort Int`

  introduces
    $\{\}$ `:` $\rightarrow$ `C`
    insert`,` *delete*`:E,C` $\rightarrow$ `C`
    size`:C` $\rightarrow$ `Int`
    `_`$\in$`_:E,C` $\rightarrow$ `Bool`
    `_`$\cup$`_,` `_`$\cap$`_,` `_-_:C,C` $\rightarrow$ `C`
    `_`$\subseteq$`_:C,C` $\rightarrow$ `Bool`

  asserts
    `C` **generated by** $\{\}$, *insert*
    $\forall\, e, e_1, e_2$`:E`, s, s_1, s_2`:C`
      $insert(insert(s, e_1), e_2) \;=\; insert(insert(s, e_2), e_1)$
      $insert(s, e) \;=\; insert(insert(s, e), e)$
      $\neg(e \in \{\})$
      $e_1 \in insert(e_2, s) \;\Leftrightarrow\; e_1 = e_2 \vee e_1 \in s$
      $size(\{\}) \;=\; 0$
      $size(insert(e, s)) \;=\; (\textbf{if}\; e \in s \;\textbf{then}\; size(s) \;\textbf{else}\; size(s) + 1)$
      $e_1 \in delete(e_2, s) \;\Leftrightarrow\; (e_1 \neq e_2 \wedge e_1 \in s)$
      $e \in (s_1 \cup s_2) \;\Leftrightarrow\; e \in s_1 \vee e \in s_2$
      $e \in (s_1 \cap s_2) \;\Leftrightarrow\; e \in s_1 \wedge e \in s_2$
      $e \in (s_1 - s_2) \;\Leftrightarrow\; e \in s_1 \wedge \neg(e \in s_2)$
      $s_1 \subseteq s_2 \;\Leftrightarrow\; (s_1 - s_2) = \{\}$
  end `Set`

Figure 4.2: `Set` trait.

If S is a specialization specification, the object specification part of S is denoted by $S.ospec$. The division of labor part of S is denoted by two components, $S.methods_i$, the set of methods assigned to component i, and $S.substate_i$, the sort of the substate assigned to component i. Discussion of constructor specifications is deferred to Sec. 4.6.

To facilitate the partitioning of the abstract state into substates, $S.ospec.sort$ and each of the $S.substate_i$ must be Larch tuple sorts. Larch tuples are tuples with named fields, like immutable records in programming languages. Each field in the $S.ospec.sort$ tuple must appear in exactly one of the $S.substate_i$. This requirement can be written as:

$$S.ospec.sort = \prod_i S.substate_i$$

The tuple-sort product operator takes tuple sorts with disjoint field names and returns the tuple sort that combines all the field names and field sorts of all the individual sorts. Tuple-sort product is undefined on tuples that do not have disjoint field names.

A specialization specification for `IntSet` is given in Fig. 4.3. The abstract state for the entire specification is given by the "**state is**" followed by a tuple sort. The substates of components are given by the "**substate is**" declarations. Each field in the abstract state declaration must appear in exactly one of the substate declarations. The state of `IntSet` has two fields, s and c. The field s, which contains the elements of the set, is the substate of the component that contains the methods `addElement`, `removeElement`, and `elements`. The method `elements` returns an enumeration, which is specified with two fields, `seq` giving the values yielded by the enumeration, and `index` giving the current position of the enumeration. The field c, used by `contains` to cache membership tests, is the substate of the component that contains the methods `contains` and `uncache`.

4.3 Behavioral subclassing

Sec. 3.3 points out that, when reasoning about subclasses, it is not enough to reason about the correctness of local code. In addition, to ensure that superclass code reused by the subclass works as expected, the subclass must be shown to be a behavioral subclass of the superclass. The rules of behavioral subclassing constrain the specification of a subclass relative to the specification of its superclass. The rules of behavioral subclassing also take into account what methods the subclass has overridden. This section formalizes the rules for behavioral subclassing.

We break behavioral subclassing into two sets of rules. The first set of rules define inclusion for specialization specifications. One specialization specification *includes* another if all classes implementing the latter specification also implement the former. The second set of rules define simple behavioral subclassing. These are simple, very restrictive rules for behavioral subclassing.

By combining inclusion with simple behavioral subclassing, we get the definition for full behavioral subclassing given by the following equation:

$$\begin{aligned}
&Subclass(T, S, L) = \exists T', S' \cdot Includes(T, T') \wedge Simple(T', S', L) \wedge Includes(S', S) \\
&\text{where} \quad Full : SpclSpec \times SpclSpec \times OverrideList \rightarrow Bool \\
&\qquad\qquad Simple : SpclSpec \times SpclSpec \times OverrideList \rightarrow Bool \\
&\qquad\qquad Includes : SpclSpec \times SpclSpec \rightarrow Bool
\end{aligned}$$

```
specialization specification IntSetSSpec { // Spec of IntSet

   uses Integer, Set(Int, IntSet);
   state is [ s:IntSet, c:IntSet ];
   invariant this^ρ.c ⊆ this^ρ.s;

   overridable component {
      substate is [ s:IntSet ];

      public void addElement(int el) {
         Modifies this;
         Ensures this^post.s = insert(this^pre.s, el);
      };

      public boolean removeElement(int el) {
         Modifies this;
         Ensures this^post.s = delete(this^pre.s, el) ∧ result = el ∈ this^pre.s;
      };

      public IntEnumeration elements() {
         Ensures result.index = 0 ∧ len(result.seq) = size(this^pre.s)
            ∧ (∀i · i ∈ this^pre.s ⇔ i ∈ result.seq);
      };
   }

   overridable component {
      substate is [ c:IntSet ];

      public boolean contains(int el) {
         Modifies this.c;
         Ensures result = el ∈ this^pre.s;
      };

      protected void uncache(int el) {
         Modifies this.c;
         Ensures el ∉ this^post.c ∧ this^post.c ⊂ this^post.c;
      };
   };
};
```

Figure 4.3: Formal specialization specification for `IntSet`

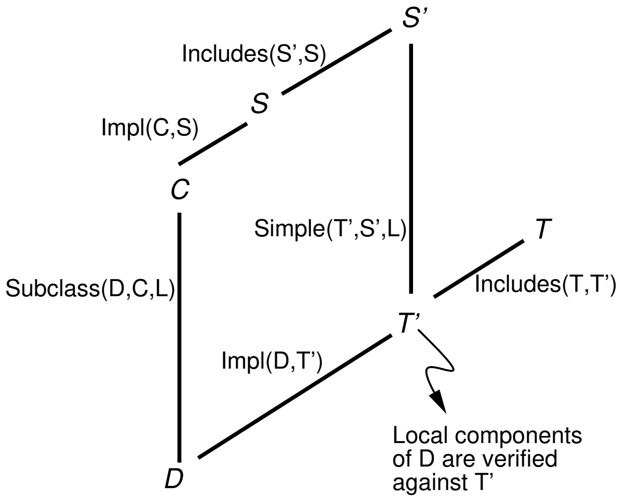


Figure 4.4: Full behavioral subclassing.

Both full and simple behavioral subclassing are relations on two specialization specifications plus a list of methods that have been overridden by the subclass. Inclusion is a relation on two specialization specifications.

Fig. 4.4 graphically denotes the above equation. In this figure, class D is a subclass of class C. L is the set of methods overridden by D. We are given that C implements specialization specification S. We want to know: Is D specified by specialization specification T is a behavioral subclass of C? Our rule for full behavioral subclassing answers "yes, if there exist two intermediate specifications T' and S' such that S includes S', T includes T', and T' is a simple behavioral subclass of S'." A full verification of D includes verifying the local components of D. As indicated by the figure, these components are verified against specialization specification T'. Verification of local components is discussed in Sec. 4.4.

Full and simple behavioral subclassing are both behavioral subclassing relationships. That is, they both ensure that superclass code reused by the subclass works as expected. As mentioned earlier, simple behavioral subclassing is very restrictive. Full behavioral subclassing generalizes this restrictive relationship by combining it with inclusion. Splitting behavioral subclassing into inclusion and simple behavioral subclassing simplifies the overall description. In particular, behavioral subclassing defined here allows for the sub- and superclasses to have different abstract states, a case ignored in Sec. 3.3. By handling differences of state in the rules for inclusion, we avoid spelling out complicated rules to handle corner cases that arise when subclasses override components.

Sec. 4.3.1 gives sufficient conditions for inclusion. Sec. 4.3.2 defines simple behavioral subclassing. Sec. 4.3.3 gives an example of full behavioral subclassing.

4.3.1 Inclusion of specialization specifications

Specialization specification τ includes specialization specification σ if all classes that implement σ also implement τ. This section gives two sets of sufficient conditions for inclusion.

One way a specialization specification can include another when it merges components of the other. When specialization specification τ includes σ by merging components, τ and σ are the

specialization specification `IntSetSSpec2` { `// Alternative spec for IntSet`

  uses `Integer, FiniteMap(IntBoolMapSort, Int, Bool);`
  state is `[` m`:IntBoolMapSort,` c`:IntSet ];`
  invariant $\forall e \cdot e \in \mathbf{this}^{\rho}.c \Rightarrow (\mathit{defined}(\mathbf{this}^{\rho}.m, e) \wedge \mathit{apply}(\mathbf{this}^{\rho}.m, e))$;

  overridable component {
    substate is `[` m`:IntMapSort ];`

    public `void addElement(int el)` {
      Modifies this;
      Ensures $\mathbf{this}^{post}.m = \mathit{update}(\mathbf{this}^{pre}.m, \texttt{el}, \mathit{true})$;
    };

    public `boolean removeElement(int el)` {
      Modifies this;
      Ensures
        $\mathbf{this}^{post}.m = \mathit{update}(\mathbf{this}^{pre}.m, \texttt{el}, \mathit{false})$
        $\wedge$ **result** $= \mathit{apply}(\mathbf{this}^{pre}.m, \texttt{el})$;
    };

    ...
  }

  ...
};

Figure 4.5: Map-based specification for `IntSet`

same (textually) in every way except that two or more components of σ may be *merged* in τ. Two components are merged by combining both their methods and substate fields. For example, the specification of `IntSet` in Fig. 4.3 is included by a specification that is the same except that it only has a single component to which all state and all methods are assigned.

Another way a specialization specification can include another is when it can be simulated by the other. For example, the map-based specification of `IntSet` in Fig. 4.5 can be shown to include the set-based specification in Fig. 4.3 in this way.

When τ includes σ by simulation, then τ and σ must have the same method suite and the same grouping of methods into components. In addition, there must exist an abstraction function from $\sigma.\mathit{ospec}.\mathit{sort}$ to $\tau.\mathit{ospec}.\mathit{sort}$ that respects component boundaries. When composed with this abstraction function, τ must both respect the assumptions of σ and must correctly describe code provided by classes implementing σ.

In more detail, the abstraction function A from σ to τ must fulfill the following criteria:

1. *A respects component boundaries*. τ and σ have the same groupings of methods into components. A must map the substate of each component independently, *i.e.*, it must be the product of multiple subabstraction functions A_i:

$$A(x) = \prod_i A_i(x \downarrow \sigma.\mathit{substate}_i)$$

for each component i, where the range of A_i is $\tau.substate_i$. ($x \downarrow S$ is x projected onto S, *i.e.*, x with only the S fields.)

2. *Under A, τ respects assumptions made by classes implementing σ.* Assume a class `C` implements σ. This code makes assumptions about **this** that must be met by subclasses that override methods of `C`. These assumptions are that subclasses will preserve the invariant of σ, and also that subclasses will implement the specifications of deferred and overridable methods given in σ. τ must respect these assumptions.

   This can be formalized as:

$$\tau.ospec.inv[A(\mathbf{this}^{\rho})/\mathbf{this}^{\rho}] \Rightarrow \sigma.ospec.inv$$

   If there is an instance invariant, then a similar implication must hold. Also, for each deferred and overridable method m:

$$\sigma.ospec.pre_m \Rightarrow \tau.ospec.pre_m[A(\mathbf{this}^{pre})/\mathbf{this}^{pre}]$$
$$\tau.ospec.post_m[A(\mathbf{this}^{pre})/\mathbf{this}^{pre}, A(\mathbf{this}^{post})/\mathbf{this}^{post}] \Rightarrow \sigma.ospec.post_m$$

3. *Under A, τ correctly describes code provided by classes implementing σ.* Again, assume a class `C` implements σ. This means the code of `C` implements the method specifications of σ. τ must correctly specify this code. This is assured when the specifications of these methods in τ is no stronger than their specifications in σ. That is, for all final and overridable methods m:

$$\tau.ospec.pre_m[A(\mathbf{this}^{pre})/\mathbf{this}^{pre}] \Rightarrow \sigma.ospec.pre_m$$
$$\sigma.ospec.post_m \Rightarrow \tau.ospec.post_m[A(\mathbf{this}^{pre})/\mathbf{this}^{pre}, A(\mathbf{this}^{post})/\mathbf{this}^{post}]$$

4.3.2 Simple behavioral subclassing

Simple behavioral subclassing is defined by three rules:

- The abstract states and invariants of the sub- and superclass specifications are the same (textually).

- Methods common to the sub- and superclass must have the same specifications in the sub- and superclass *except* for superclass methods overridden by the subclass, which may have stronger specifications in the subclass. There are no constraints on the specifications of new methods introduced by the subclass.

- The subclass must inherit method components as a whole and cannot break them up. That is, the subclass's division of labor must group inherited methods and associated state the same as the superclass's does. However, the subclass may regroup the state and methods of components it replaces.

As mentioned before, these rules for behavioral subclassing are very restrictive, much more so than the rules given in Sec. 3.3. These restrictions are overcome by combining this rule with inclusion.

Regarding method signatures, the rules of the programming language determine what changes are allowed in method signatures. For example, Java allows a subclass to make the return-type of a method more specific (*i.e.*, a subtype), but does not otherwise allow changes in method signatures.

4.3.3 Example

We illustrate full behavioral subclassing with `PrioritySet` (Fig. 4.6 and Fig. 4.7), a subclass of `IntSet`. `PrioritySet`s are like sets except they support a method `pop` which extracts the least element from the set. The implementation of `PrioritySet` inherits the methods `contains` and `uncache`, overrides the methods `addElement`, `removeElement`, and `elements`, and defines the new method `pop`.

In showing that `PrioritySet` is a behavioral subclass of `IntSet`, the first step is to find S', a new specification for `IntSet` included by the old `IntSet` specification. S' must have the same object specification as the specification in Fig. 4.6. One such specification is the specification in Fig. 4.6 minus the `pop` method. This inclusion relation can be proven using the abstraction function:

$$toPSet(x) = \begin{cases} \{\} & \text{for } x = \{\}_p \\ insert(toPSet(x'), i) & \text{for } x = insert_p(x', i) \end{cases}$$

Fig. 4.6 is T', a simple behavioral subtype of S' against which `IntPrioritySet` is verified. In this case, T and T' can be the same.

4.4 Verifying local components

As discussed in Sec. 3.1, each class component must implement its specification and must be independent of the implementations of other overridable components. We verify that a component is correct using standard data refinement procedures ([Hoare72], [Liskov86], [Dahl92]) adapted to ensure independence. This section describes this adaptation. It starts with an overview of our approach, discusses the mechanics, and ends with an example. This section assumes that **super** is not used in to implement local components; the next section discusses **super**.

4.4.1 Achieving independence

To achieve independence, the standard refinment procedures must be adapted in two ways. First, the code of one method must be verified in terms of the specifications of other methods it calls. This is simple enough. Second, just as the code of one component may not access the instance variables of another, the *formal reasoning* of one component cannot make reference to the instance variables of another. Achieving this second adaptation is more subtle.

The domain of the abstraction function used to verify a method defines the "concrete" view of **this** used to reason about the method. In the standard data refinement techniques, every method is verified using the same abstraction function. This function maps the *entire* concrete state to its entire abstract state. This means the concrete view of **this** used to reason about every method includes all the instance variables of **this**. Unfortunately, this approach (illustrated in Fig. 4.8a) violates the restriction about the formal reasoning of one component not referring to the instance variables of others. (For simplicity, this chapter assumes abstraction functions rather than abstraction relations, but the results can be extended to abstraction relations.)

To adapt the standard refinement procedure to class components, each component needs its own view of the concrete state of **this**. For each component, the state of **this** has two parts, the state assigned to the component, called the *internal state*, and the state assigned to other components, called the *external state*. The concrete view of **this** seen by a component includes only to the

specialization specification `PrioritySetSSpec` {

uses `Integer, Set(Int, IntSet), PrioritySet(Int, PrioritySet);`

state `[` p`:PrioritySet,` c`:IntSet ];`
invariant $\forall e \cdot e \in \mathbf{this}^{\rho}.c \Rightarrow e \in_p \mathbf{this}^{\rho}.p$;

overidable component {
substate `[` p`:PrioritySet ];`

public `void addElement(int el)` {
Modifies **this**;
Ensures $\mathbf{this}^{post}.p = insert_p(\mathbf{this}^{pre}.p, \mathtt{el})$;
};

public `boolean removeElement(int el)` {
Modifies **this**;
Ensures $\mathbf{this}^{post}.p = delete_p(\mathbf{this}^{pre}.p, \mathtt{el}) \ \wedge\ \mathbf{result} = \mathtt{el} \in_p \mathbf{this}^{pre}.p$;
};

public `int pop()` **throws**`(Bounds)` {
Modifies **this**;
Ensures $\mathbf{this}^{post}.p = rest(\mathbf{this}^{pre}.p) \wedge (\mathbf{this}^{pre}.p = \{\}_p \Rightarrow \mathbf{throws}(\mathtt{Bounds}))$
$\wedge (\mathbf{this}^{pre}.p \neq \{\}_p \Rightarrow \mathbf{result} = head(\mathbf{this}^{pre}.p))$;
};

public `IntEnumeration elements()` {
Ensures $\mathbf{result}.index = 0 \ \wedge\ len(\mathbf{result}.seq) = size(\mathbf{this}^{pre}.s)$
$\wedge\ (\forall i \cdot i \in \mathbf{this}^{pre}.s \Leftrightarrow i \in \mathbf{result}.seq)$;
};
};

overridable component {
substate `[` c`:IntSet ];`

public `boolean contains(int el)` {
Modifies **this**.c;
Ensures $\mathbf{result} = \mathtt{el} \in_p \mathbf{this}^{pre}.p$;
};

protected `void uncache()` {
Modifies **this**.c;
Ensures $\mathtt{el} \notin \mathbf{this}^{post}.c$;
};
};
};

Figure 4.6: Specialization specification of `PrioritySet`.

```
PrioritySet(E, C): trait
  assumes TotalOrder(E)
  includes Integer

  introduces
    {}_p : → C
    __ ∈_p __: E, C → Bool
    insert_p, delete_p: E, C → C
    biggest: C → E
    rest: C → C
    size_p: C → Int

  asserts
    C generated by {}_p, include_p
    ∀ e, e1, e2: E,  p: C
      ¬(e ∈ {}_p)
      insert_p(insert_p(p, e1), e2) = insert_p(insert_p(p, e2), e1)
      insert_p(p, e) = insert_p(insert_p(p, e), e)
      e1 ∈ insert_p(e2, p) ⇔ e1 = e2 ∨ e1 ∈_p p
      size_p({}_p) = 0
      size_p(insert_p(e, p)) = (if e ∈_p p then size_p(p) else size_p(p) + 1)
      e1 ∈_p delete_p(e2, p) ⇔ (e1 ≠ e2 ∧ e1 ∈_p p)
      e = head(p) ⇔ (e ∈ p ∧ (e1 < e ∨ e1 ∉_p p))
      rest({}_p) = {}_p
      p ≠ {}_p ⇒ (e ∈ rest(p) ⇔ e ≠ head(p))
  end PrioritySet
```

Figure 4.7: Trait definint priority sets.

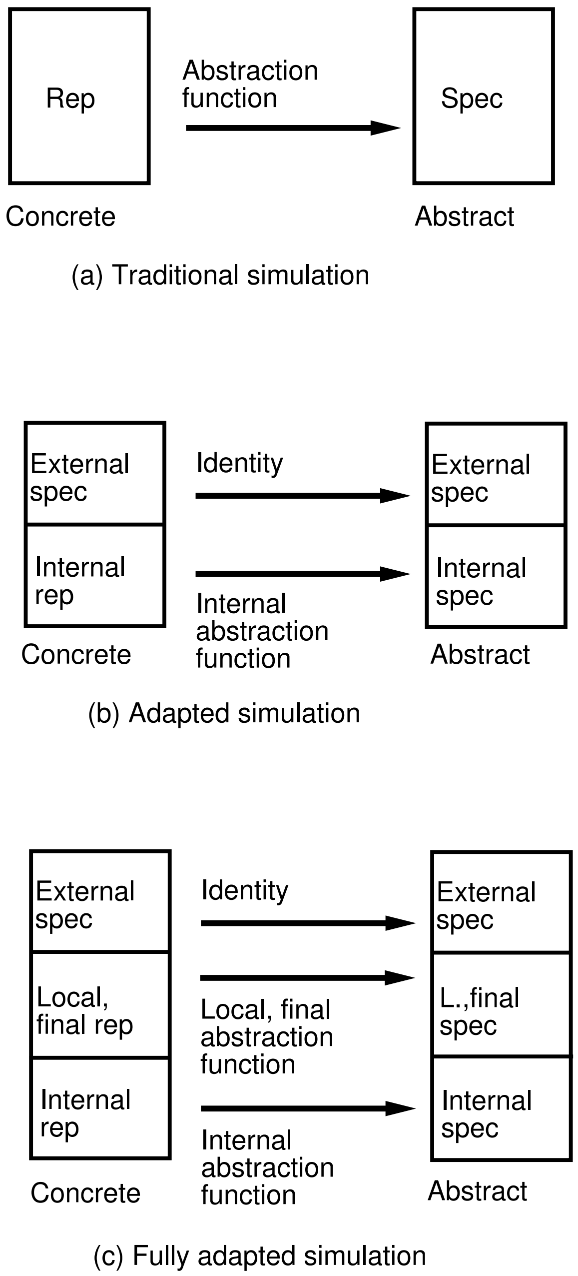


Figure 4.8: Classical and adapted refinements.

instance variables representing the component's internal state. This view hides the representation of the external state by treating that state *abstractly*.

This per-component view of **this** is illustrated in Fig. 4.8b. The concrete view of **this** seen by each class component is a product of the internal representation and the external, *abstract* state. In each component, the concrete view of **this** is related to the fully abstract view by a compound abstraction function that is the identity on the external state and is defined by an internal abstraction function on the internal state. With this compound abstraction function, one can verify method code as in standard refinements. The identity part of the compound abstraction function hides the internal abstraction functions of other class components, achieving independence: if the internal abstraction function of one component changes, the verification of other components is still valid.

This basic adaptation of the standard refinement techniques must be extended to deal with final state. Recall from Sec. 2.2 that every component in a class can access the representation of the class's *local* final state (local final state is final state implemented by the class itself, as against final state inherited from the superclass). This is safe because subclasses cannot replace the representation of final state. Thus, the view of **this** seen by each class component includes both its own representation *and* the representation of the class's final state. This extended view of **this** is illustrated in Fig. 4.8c. Each class component uses the same abstraction function to map the representation of the final state.

4.4.2 Mechanics

This subsection looks at the mechanics of verifying local components of class C against specialization specification T. Table 4.1 summarizes symbols defined in this subsection and gives particular values used to verify the `addElement` method of `IntSet`.

For each local component j, the verifier defines a subabstraction function A_j:

$$A_j \, : C.rep_j \rightarrow T.substate_j$$

where $C.rep_j$ denotes the instance variables assigned to the local component j ($C.rep_j$ is a tuple with named fields, where each field is an instance variable). This function defines how the component's instance variables are used to represent the component's substate.

Each local component is verified separately. Let i be the component we are interested in verifying. Let `lf` be the local component of final state. For the purpose of verification, we can assume without loss of generality that there is only one component of local final state. We assume for now that $i \neq \texttt{lf}$; we discuss $i = \texttt{lf}$ below.

We now define the concrete view of **this** used to verify the code of component i. As illustrated by the left-hand side of Fig. 4.8c, this view includes the abstract, external state, the concrete, local, final state, and the concrete, internal state. Thus, the code of component i is verified as if **this** had the sort:

$$\textbf{this} : C.rep_i \times C.rep_{\texttt{lf}} \times E_i$$

where E_i is shorthand for:

$$E_i = \prod_{j \notin \{i, \texttt{lf}\}} T.substate_j$$

that is, the state of **this** whose representation the code of component i may not directly access. The state in E_i is state assigned to deferred components and to overridable components other than i.

$$
\begin{array}{rcl}
C.rep_i & = & \text{Instance variables assigned to component } i \\
 & = & [\texttt{els} : \texttt{IntVector}] \\
C.rep_{\texttt{lf}} & = & \text{Instance variables representing local final state} \\
 & = & [\,](\texttt{IntSet} \text{ has no final state}) \\
T.substate_i & = & \text{Sort of substate of component } i \\
 & = & [s : \texttt{IntSet}] \\
T.substate_{\texttt{lf}} & = & \text{Sort of local final state} \\
 & = & [\,] \\
E_i & = & \text{State whose representation component } i \text{ may not access} \\
 & = & \prod_{j \notin \{i,f\}} T.substate_j \\
 & = & [c : \texttt{IntSet}] \\
A_i & = & \text{Subabstraction function for component } i \\
 & : & C.rep_i \rightarrow T.substate_i \\
 & = & \lambda\, r \cdot [s := toSet(r.\texttt{els})] \\
A_{\texttt{lf}} & = & \text{Subabstraction function for local final state} \\
 & : & C.rep_{\texttt{lf}} \rightarrow T.substate_{\texttt{lf}} \\
 & = & \lambda\, r \cdot [\,] \\
V_i & = & \text{Function for verifying component } i \\
 & : & (C.rep_i \times C.rep_j \times E_i) \rightarrow T.ospec.sort \\
 & = & \lambda\, r \cdot A_i(r{\downarrow}C.rep_i) \times A_i(r{\downarrow}C.rep_{\texttt{lf}}) \times (r{\downarrow}E_i) \\
 & = & \lambda\, r \cdot [s := toSet(r.\texttt{els}),\ c := r.c]
\end{array}
$$

Table 4.1: Symbols for verifying local methods in component i of class C against specialization specification T. Also given are particular values for $i = \texttt{add}$, the class component of `IntSet` containing `addElement`, `removeElement`, and `elements`.

Next, we define the abstraction function V_i with which the code of component i is verified. V_i is the combination of the three subabstraction functions in Fig. 4.8c. V_i that has the signature:

$$V_i : (C.rep_i \times C.rep_{\texttt{lf}} \times E_i) \rightarrow T.ospec.sort$$

V_i is defined by the equation:

$$V_i(r) = A_i(r{\downarrow}C.rep_i)A_i(r{\downarrow}C.rep_{\texttt{lf}}) \times (r{\downarrow}E_i)$$

where *tup*$\downarrow$*Sort* is result of projecting tuple *tup* onto tuple sort *Sort*, *i.e.*, only the *Sort* fields of *tup*.

The methods of component i are verified using V_i according to standard refinement procedures (see, *e.g.*, [Liskov86]). Where the code of methods make calls to other methods, the specifications of the called methods are used to reason about the calls.

The special case of $i = \texttt{lf}$, *i.e.*, verifying the component of local, final state, is a bit simpler than the general case. In this case, the state of **this** is divided into two parts rather than three: the local, final state and the rest of the state. This component is verified using the abstraction compound function:

$$V_{\texttt{lf}}(r) = A_i(r{\downarrow}C.rep_{\texttt{lf}}) \times (r{\downarrow}E_{\texttt{lf}})$$

where

$$E_{\texttt{lf}} = \prod_{j \neq \texttt{lf}} T.substate_j$$

object specification `IntVectorISpec { // IntVector instance spec`

  uses `Integer, Sequence(Int, IntSequenceSort);`
  state `IntSequenceSort;`

  public `void addElement(int el) {`
    Modifies this;
    Ensures $\mathbf{this}^{post} = \mathbf{this}^{pre} \vdash \mathtt{el}$;
  `};`

  public `boolean removeElement(int el) {`
    Modifies this;
    Ensures $\exists s_1, s_2 \cdot$
      $\mathtt{el} \notin s_1 \;\wedge\; (\mathtt{el} = head(s_2) \;\vee\; isEmpty(s_2))$
      $\wedge\; \mathbf{this}^{pre} = s_1 \,||\, s_2$
      $\wedge\; \mathbf{this}^{post} = s_1 \,||\, tail(s_2)$
      $\wedge\; \mathbf{result} = (\mathtt{el} \in \mathbf{this}^{pre})$;
  `};`

  public `boolean contains(int el) {`
    Ensures $\mathbf{result} = (\mathtt{el} \in \mathbf{this}^{pre})$;
  `};`

  public `int size() {`
    Ensures $\mathbf{result} = size(\mathbf{this}^{pre})$;
  `};`

  `// .. spec's of other methods elided ..`
`}`

Figure 4.9: Instance specification of `IntVector`.

The methods of the local final component are verified using the abstraction function $V_{\mathtt{lf}}$ in the usual manner.

4.4.3 Example

We illustrate the verification procedure on the `addElement` method of `IntSet`. To verify `addElement`, one needs the instance specification for `IntVector` given in Fig. 4.9, which in turn uses the LSL `Sequence` sort from [Guttag93]. A full verification is not presented here, just enough to illustrate the use of V_i to verify method components.

To verify `addElement`, one must define $A_{\mathtt{add}}$, the subabstraction function for its components. Informally, $A_{\mathtt{add}}$ constructs an `IntSet` by inserting each element of **this**.`els` into an initially empty set. More formally, $A_{\mathtt{add}}$ maps the instance variable assigned to the component to the substate maintained by the component as follows:

$$A_{\mathtt{add}}(r) = [s := toSet(r.\mathtt{els})]$$

where the function *toSet* is defined as:

$$toSet(x) = \begin{cases} \{\} & \text{for } x = empty \\ insert(toSet(x'), i) & \text{for } x = x' \vdash i \end{cases}$$

(The functions *empty* and $\vdash$ generate `IntSequenceSort` values; the sequence $x \vdash i$ contains the elements of sequence x followed by integer i.)

From $A_{\texttt{add}}$ follows the abstraction function $V_{\texttt{add}}$:

$$\begin{aligned} &V_{\texttt{add}}(r) \\ &\quad = [s := A_{\texttt{add}}(r).s,\ c := r.c] \\ &\quad = [s := \mathit{toSet}(r.\texttt{els}),\ c := r.c] \end{aligned}$$

(`IntSet` has no final state, so we can ignore $A_{\texttt{lf}}$ in this case.) $V_{\texttt{add}}$ is used to verify the code in the `addElement` component.

To verify a method in the `addElement` component, $V_{\texttt{add}}$ is composed with the specification of the method to transform the specification into the concrete domain assumed by the implementation. Next, the proof rules of the language are used to show that the code of the method meets its transformed specification.

For example, composing $V_{\texttt{add}}$ with the **ensures** clause of `addElement` yields:

$$V_{\texttt{add}}(\mathbf{this}^{post}).s = \mathit{insert}(V_{\texttt{add}}(\mathbf{this}^{pre}).s, \texttt{el})$$

which expands to:

$$\mathit{toSet}(\mathbf{this}^{post}.\texttt{els}) = \mathit{insert}(\mathit{toSet}(\mathbf{this}^{pre}.\texttt{els}), \texttt{el})$$

The code of `addElement` can be verified against this in two cases, first when `el` is not already in $\mathbf{this}^{pre}.\texttt{els}$ and second when it is:

1. When `el` is in $\mathbf{this}^{pre}.\texttt{els}$, one can conclude that:

   $$\texttt{el} \in \mathit{toSet}(\mathbf{this}^{pre}.\texttt{els})$$

   Also, because `addElement` is not called in this case, one can conclude:

   $$\mathit{toSet}(\mathbf{this}^{post}.\texttt{els}) = \mathit{toSet}(\mathbf{this}^{pre}.\texttt{els})$$

   Combining these with the lemma

   $$i \in s \ \Rightarrow\ s = \mathit{insert}(s, i)$$

   which is implied by the idempotence of *insert*, it follows that:

   $$\mathit{toSet}(\mathbf{this}^{post}.\texttt{els}) = \mathit{insert}(\mathit{toSet}(\mathbf{this}^{pre}.\texttt{els}), \texttt{el})$$

   So the **ensures** clause is met in this case.

2. When `el` is not in $\mathbf{this}^{pre}.\texttt{els}$, the effects of the invocation of `IntVector`'s `addElement` method must be taken into account. When this invocation returns, it establishes as its postcondition:

   $$\mathbf{this}^{post}.\texttt{els} = \mathbf{this}^{pre}.\texttt{els} \vdash \texttt{el}$$

   Applying *toSet* to each side, we get:

   $$\mathit{toSet}(\mathbf{this}^{post}.\texttt{els}) = \mathit{toSet}(\mathbf{this}^{pre}.\texttt{els} \vdash \texttt{el})$$

It follows from the definition of *toSet* that:

$$\mathit{toSet}(s \vdash e) = \mathit{insert}(\mathit{toSet}(s), e)$$

Putting these two together, one gets:

$$\begin{aligned}&\mathit{toSet}(\mathbf{this}^{post}.\texttt{els})\\ &\quad = \mathit{toSet}(\mathbf{this}^{pre}.\texttt{els} \vdash \texttt{el})\\ &\quad = \mathit{insert}(\mathit{toSet}(\mathbf{this}^{pre}.\texttt{els}), \texttt{el})\end{aligned}$$

So the **ensures** clause is met in this case too.

A complete verification of `addElement` must also verify the **modifies** clause and must verify that `addElement` preserves the class's invariant. The **modifies** clause requires that only **this** is modified. This is true, but we must assume no "representation exposure," *i.e.*, that changes to `els` affect only **this** and no other objects (*c.f.*, [Schaffert81, Leino95]).

Showing that `addElement` preserves its invariant means assuming

$$\mathbf{this}^{pre}.c \subseteq \mathit{toSet}(\mathbf{this}^{pre}.\texttt{els})$$

and proving:

$$\mathbf{this}^{post}.c \subseteq \mathit{toSet}(\mathbf{this}^{post}.\texttt{els})$$

Informally, the invariant is preserved because `addElement` does not change c and at most adds an element to `els`. In more detail, we have just shown that:

$$\mathit{toSet}(\mathbf{this}^{post}.\texttt{els}) = \mathit{insert}(\mathit{toSet}(\mathbf{this}^{pre}.\texttt{els}), \texttt{el})$$

The body of `addElement` does nothing to change the c field, so:

$$\mathbf{this}^{pre}.c = \mathbf{this}^{post}.c$$

Substituting these equalities into the assertion to be proved yields:

$$\mathbf{this}^{pre}.c \subseteq \mathit{insert}(\mathit{toSet}(\mathbf{this}^{pre}.\texttt{els}), \texttt{el})$$

This assertion is implied by:

$$\mathbf{this}^{pre}.c \subseteq \mathit{toSet}(\mathbf{this}^{pre}.\texttt{els})$$

which we can assume.

4.5 Shadowed components

To verify shadowed components (Sec. 3.3.4), both the rules of simple behavioral subclassing and the verification procedure for local methods must be modified.

The rules for simple behavioral subclassing are augmented to indicate what happens to shadowed components. A subclass cannot regroup methods of shadowed components. When a subclass shadows a component, then the shadowing component (in the subclass) must include at least the

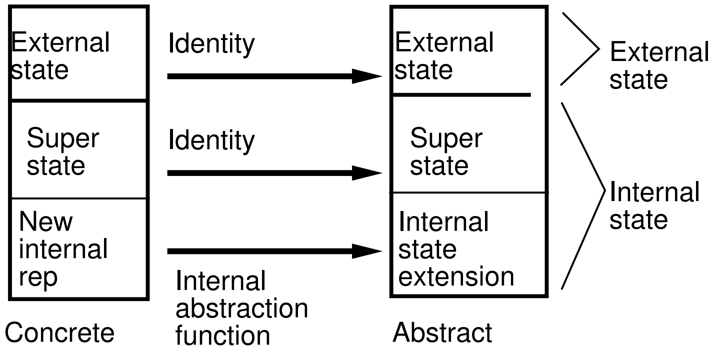


Figure 4.10: Refinement when **super** is used.

$$
\begin{array}{rcl}
C.rep_i & = & \text{Instance variables assigned to component } i \\
S.substate_i & = & \text{Sort in superclass of substate of component } i \\
E_i & = & \text{Additional substate fields added by } C \text{ to component } i \\
T.substate_i & = & \text{Sort of substate of component } i \\
 & = & S.substate_i \times E_i \\
A_i & = & \text{Subabstraction function for component } i \\
 & : & C.rep_i \rightarrow E_i \\
V_i & = & \text{Function for verifying component } i \\
 & : & (C.rep_i \times S.substate_i \times \prod_{j \neq i} C.substate_j) \rightarrow C.ospec.sort \\
 & = & \lambda s \cdot (s \uparrow C.rep_i) \times A_i(s \downarrow C.rep_i)
\end{array}
$$

Table 4.2: Verifying component i, a class component of class C that shadows a superclass component. T is the specialization specification of C, and S is the specialization specification of C's superclass. E_i are fields of S that are being made part of the substate of i in T.

methods and all substate fields of the shadowed component (in the superclass). The shadowing component may contain additional methods and substate fields. The shadowing component can strengthen the specification of methods that it overrides, even methods whose superclass version is invoked via **super**.

The verification procedure is changed (a) to adapt the abstraction function for shadowed components and (b) to clarify which specification should be used to reason about calls via **super**. Change (b) is simple: the superclass's specification is used. The subclass version of m is verified against the subclass specification for m, and subclass invocations of m through **this** are also reasoned about using the subclass specification for m.

Change (a) requires adding more structure to abstraction functions. This new structure is illustrated in Fig. 4.10 and summarized in Table 4.2 (for simplicity, we ignore local final state here). Assume that G is a superclass component shadowed by G' in the subclass. When verifying G', the substate fields of G, *i.e.*, the substate fields represented by the superclass, are treated like external state. That is, they are taken as part of the concrete view of **this**, and an identity function is used to abstract them. G' may contain more fields than G does. These new subclass fields are defined by an internal abstraction function mapping from the new instance variables of the subclass.

4.6 Specifying and verifying constructors

So far, this report has largely ignored constructors because, while the verification of regular methods is largely language independent, the verification of constructors is not. Mechanisms for object construction differ from language to language, and the details of constructor verification depend on these differences. This section presents the specification and verification of constructors in the context of the Java language ([Javabeta95]).

In Java, object construction is a multi-step process involving *initializers*, initialization expressions associated with instance variables, and *constructors*, special methods defined in a class for object initialization. Assume that class C_n is being instantiated, where C_{n-1} is the direct superclass of C_n, C_{n-2} is the direct superclass of C_{n-1}, and so on back to C_0, which is `Object`, the universal root class.

- First, space for a C_n object is allocated, and its instance variables are "pre-initialized" to default values according to their type, *e.g.*, numerical variables are initialized to zero, object references are initialized to nil. (The motivation for pre-initialization is described below.)
- Constructors are called in a top-down manner, *i.e.*, the constructor for C_0, the root class, is called first, then the constructor for C_1, and so on. (In Java, classes can have more than one constructor, but, for simplicity, we assume classes have only one.)
- Between the calls to the constructors of C_i and C_{i+1}, the instance variables of C_{i+1} are initialized by evaluating their initializers. Initializers are evaluated according to the textual order of instance-variable declarations in the class definition.

Object construction in Java (and in most languages) is intricate because both constructors and initializers can call methods of the object being constructed. Because of method override, these calls might dispatch to method code *lower* in the class hierarchy than the constructor making the call. As a result, these methods might see instance variables that have not yet been initialized by a constructor. Java's "pre-initialization" of instance variables ensures that these methods will not access completely uninitialized variables, but this pre-initialization will not in general establish representation invariants assumed by these methods.

To support straight-forward verification of constructors, a programming convention is assumed. In this convention, initializers are assumed have no side-effects. Further, initializers and constructors can only call *initialization methods* of **this**: final, private methods that are *only* called by initializers and constructors and not by other methods. Initialization methods cannot assume any representation invariants.

4.6.1 Specification of constructors

Like other routines, constructors are specified using pre- and postconditions. However, to simplify verification, the postconditions of constructors are structured as conjunctions of per-component postconditions. If T is a specialization interface, then:

- $T.pre_{cons}$ is the precondition of T's constructor.
- $T.post^{i}_{cons}$ is a postcondition assertion for T's constructor that only mentions substate fields assigned to component i. $T.post^{i}_{cons}$ is defined for final and overridable components only.

$$
\begin{array}{rcl}
R & = & \text{All local instance variables of } C \\
 & = & \prod_{i \in L} C.rep_i \\
O & = & \text{Inherited and deferred state} \\
 & = & \prod_{i \notin L} T.substate_i \\
V_{cons} & = & \text{Constructor abstraction function} \\
 & : & (R \times O) \rightarrow T.ospec.sort \\
 & = & \lambda\, s \cdot (s{\downarrow}O) \times \prod_{i \in L} A_i\,(s{\downarrow}C.rep_i)
\end{array}
$$

Table 4.3: Derivation of constructor abstraction function for verifying initialization code of class C against specialization specification T. L is the set of local class components, *i.e.*, components for which C provides its own code. The subabstraction functions A_i must be the same as those in Table 4.1.

- The full postcondition of T's constructor is the conjunction of its per-component postconditions:

  $$T.post_{cons} = \bigwedge_i T.post^i_{cons}$$

  Again, i ranges over final and overridable components only.

The motivation for structuring constructor postconditions in this way is that the constructor of a class should be able to *selectively* assume the postcondition of its immediate superclass's constructor, *i.e.*, it should be able to assume postconditions relating to components it inherits but not those relating to components it replaces.

4.6.2 Verification of constructors

Constructors, initializers, and initialization methods are all verified using the *constructor abstraction function*. Constructors must initialize all instance variables of a class, not just the instance variables assigned to a single class component. Thus, the concrete view of **this** assumed by this abstraction function treats the substates of all local components in terms of their representations. The constructor abstraction function is the product of the subabstraction functions used to verify local components; a definition of the constructor abstraction function is given in Table 4.3.

On entry, the implementation of the constructor can assume $S.post^i_{cons}$ for components i that have *not* been overridden, where S is the specialization specification of the superclass. Further, the implementation can assume that the instance variable initializers have been executed. Under these assumptions, the body of the constructor can be verified in the usual way using the constructor abstraction function.

Chapter 5

Improving extensibility

There is a trade-off between modularity and extensibility. At one extreme, allowing specializers to look at the source code of superclasses permits the widest range of customized subclasses, but is not at all modular. At the other extreme, requiring that specializers override *all* methods of superclasses is very modular—it permits the widest range of changes to superclasses that do not affect subclasses—but it results in no code reuse. The methodology based on class components outlined in the previous chapters strikes a balance between extensibility and modularity. However, we can improve the extensibility of this approach without sacrificing modularity. This chapter explains how.

Sec. 5.1 looks at splitting the specialization specifications of overridable methods into one used when the method is inherited and another used when the method is overridden. (Including the instance specification, this brings the number of specifications for overridable methods to three. Chapter 7 looks at a way to simplify this situation.) Sec. 5.2 looks at improving the extensibility of formal specifications. By parameterizing sorts on a per-object basis, the behavior of methods can be specified in a manner that is both precise and extensible.

5.1 Particular and assumed specifications

Consider `IntSet4`, in Fig. 5.1. `IntSet4` is just like `IntSet` except that `elements` yields the elements of the set in ascending order (`IntSet4` achieves this by keeping `els` sorted). We would like to specify `IntSet4` such that (a) the instance specification of `IntSet4` says that `elements` returns a sorted listing, (b) a subclass that inherits the `elements` component can assume that `elements` returns a sorted listing, and (c) a subclass that overrides the `elements` components does *not* have to implement an `elements` method that yields its elements in ascending order.

This flexibility can be achieved if the specialization specification of a overridable method can have two specifications, an *particular specification* that describes what the class's own version of the method actually does and an *assumed specification* that constraints what subclass versions of the method can do. The particular specification is stronger than (*i.e.*, implies) the assumed one. In the case of `IntSet4`, the particular specification of `elements` would say that `elements` yields the elements of the set in ascending order, while the assumed specification would say that it yields the elements of the set in any order.

With two specifications, the rules for reasoning about classes do not change, but they have to be

```
class IntSet4 {

 // ``elements'' component
   private IntVector els = new IntVector();
     // Rep invariant: no duplicates and kept in sorted order;

   public overridable void addElement(int el) {
     int i = findIndex(el);
     if (i > 0 && els.elementAt(i-1) == el) return;
     els.insertElementAt(el, i);
   };

   public overridable boolean removeElement(int el) {
     int i = findIndex(el);
     if (i == 0 || els.elementAt(i-1) != el) return false;
     this.uncache(el);
     els.removeElementAt(el, i-1);
     return true;
   };

   public overridable IntEnumeration elements() {
     return els.elements();
   };

   private final int findIndex(int el) {
     % Effects: Helper function for elements component that returns
     %   the highest index at which el can be inserted into els
     %   while keeping els in sorted order
     if (el < els.firstElement()) return 0;
     if (els.lastElement() <= el) return els.size();

     int lo = 0;
     int hi = els.size() - 1;
     while(lo+1 < hi) { /* Invariant: els[lo] <= el < els[hi]  */
       int mid = hi - (hi-lo)/2;
       if (el < els.elementAt(mid)) hi = mid;
       else lo = mid;
     }
     return hi;
   }

  ..other methods elided
};
```

Figure 5.1: Integer sets with ordered enumeration.

clarified to indicate which specification is used when:

- The particular specification is used when reasoning about the implementation of an overridable method, *i.e.*, the code of a method must do what the particular specification of the method says it does. Because the particular specification implies the assumed specification, this reasoning will ensure that the method meets both specifications.

- The assumed specification is used when reasoning about a call from one method to another method. For example, in `IntSet4`, the call to `elements` in `contains` must be reasoned about using the assumed specification of `elements`, which means `contains` cannot assume that elements are yielded in ascending order.

- The particular specifications of methods are used when when reasoning about the instance specification of a class by comparing it to the specialization specification.

- The superclass's particular specification for the method is used when reasoning about a call via **super** to a superclass version of a method.

- In the informal rules for behavioral subclassing (Sec. 3.3), the rules for overridable methods must be clarified:

    1. The particular specifications of inherited, overridable methods in the subclass must be implied by their particular specifications in the superclass.

    2. The assumed specifications of all overridable in the subclass must imply their assumed specifications in the superclass.

- In the formal rules for inclusion (Sec. 4.3.1), the implication on method specifications in the "respects clause" (rule (2), page 41) applies to the assumed specifications of overridable methods, while the implication on method specifications in the "describes clause" (rule (3), page 41) applies to the particular specifications of overridable methods.

    In the formal rules for simple behavioral subclassing (Sec. 4.3.2), where it says that overridden methods may have stronger specifications in the subclass, this means that the assumed specification in the subclass must imply the assumed specification in the superclass.

Note that there need not be any relation between the particular specifications of a sub- and superclass.

5.2 Extensible specialization specifications

Fig. 5.2 gives `IntCollection`, a partial class for building integer collections. This class is inspired by the `Collection` class in the Smalltalk library [Goldberg89]. This class provides code for about two dozen methods common to collection classes, such as membership-testing, adding and removing multiple members from an array, and so-on. Subclasses provide their own implementation for the `elements` component and can inherit these two dozen methods.

An informal specification for `IntCollection` is given in Fig. 5.3. The aim of this specification is to be as general as possible. This will allow the reuse of `IntCollection` in all kinds

```
class IntCollection {

 // Deferred ``elements'' component
   public deferred void addElement(int el);
   public deferred boolean removeElement(int el);
   public deferred IntEnumeration elements();

 // ``cache'' component (same as IntSet)
   private int c_val; // Value currently in cache
   private boolean c_valid = false; // True only if c_val is valid

   public overridable boolean contains(int el) {
     if (c_valid && c_val == el) return true;
     for(IntEnumeration e = this.elements(); e.hasMoreElements(); )
       if (el == e.nextElement()) {
         c_valid = true; c_val = el;
         return true;
       };
     return false;
   };

   protected overridable void uncache(int el) {
     if (c_val == el) c_valid = false;
   };


 ..About two dozen other methods implemented in terms of addElement,
   removeElement, elements, and contains.  For example:

 public void removeElements(int els[]) {
   for(IntEnumeration e = this.elements(); e.hasMoreElements(); )
      this.removeElement(e.nextElement());
 };

 ...
};
```

Figure 5.2: Base class for integer-collection classes.

of collection classes, *e.g.*, both ordered and unordered collections, and both set-like and multiset-like collections. To achieve this generality, notice that the description of the `elements` field—"a collection of integers"—is vague. The specifications of `removeElement` of `elements` are also vague—they make vague hints about values being in `elements` "multiple times."

A formal specification can clear up this vagueness, but classes like `IntCollection` are challenging to specify. What mathematical space should be used to model the `elements` field? This space needs to include sets, multisets, sequences, and a wide-variety of other collection values. How should the behavior of methods be specified, given that the behavior in different subclasses can vary markedly? For example, in a set-like subclass, `addElement` is an idempotent operation, *i.e.*, after an element is inserted the first time, all subsequent insertions of that element have no effect. In multiset-like subclasses, `addElement` is *not* idempotent: each insertion of an element into such an object changes the state of the object.

The challenge of designing mathematical spaces that are crafted so as to be reusable is not unique to specifying specialization interfaces. The issue also arises in the context of instance specifications. For example, most collection libraries have a collection type that is a supertype of all other collection types, and specifying this type raises the same issue. The issue also arises in the context of designing LSL handbooks, collections of LSL traits defining related sorts [Guttag93]. Below we propose a solution that seems promising for specifying both the instance and specialization specifications of classes. However, the important point here is the identification of the general problem as it pertains to specifying classes. If formal methods are to be applied to object-oriented programs, some solution to this problem—whether the one here or some other—will be necessary.

Our solution is to carefully craft the sorts used to model the abstract state of classes like `IntCollection`. An *extensible sort* is a larger space of values containing subspaces of values for which we can assert properties independently. In classes such as `IntCollection`, where the abstract state of different subclasses need different, contradictory properties, each subclass will get its own subspace of values and will assert the needed properties. A *class assertion* is an axiom associated with a class that asserts properties for the subspace of an extensible sort associated with all subclasses of the class. A *component assertion* is an axiom associated with a class component that asserts properties for just those subclasses that inherit the component.

5.2.1 Extensible sorts

When it comes to designing the abstract spaces for extensible classes, the challenge is this. First, we need to define a space of values that all share some general properties. For example, all `IntCollections` share a general notion of "membership" and of "inserting" and "removing" members of a collection. At the same time, we want to be able to divide that space of values into subspaces that have additional properties, where the additional properties of different subspaces might be contradictory. In our example, we want to carve up the very general space of "collections" into subspaces for sets, multisets, and other kinds of collections. For some of these subspaces (set-like collections), insertion is idempotent, while for others (multiset-like collections), it is not. Similarly, for some subspaces (unordered collections), insertion is commutative, while for others (ordered collections), it is not. We need to be able to assert properties on each subspace independently.

For each *object* subsumed by a specification like `IntCollection`, our approach is to create a subspace for the object. That is, each individual `IntCollection` object has its own subspace of

```
specialization specification IntCollectionSSpec {

 state field elements; // A collection of integers
 state field cache; // A set of integers

 invariant if integer i ∈ cache then i is a member of elements

 overridable component {
   substate field elements;

   public void addElement(int el);
     // Modifies: this.cache
     // Effects: Adds el to this.elements.

   public boolean removeElement(int el);
     // Modifies: this.elements, this.cache
     // Effects: Removes el from this.elements, returning true iff
     //   el is in to begin with.  In some subclasses, eg,
     //   IntMultiSet, an element may have to be removed multiple
     //   times before it is completely gone from this.elements.

   public IntEnumeration elements();
     // Effects: Returns an enumeration of the integers in this.elements.
     //   This enumeration yields each distinct integer only once, even
     //   integers that are in this.elements multiple times.

 };

 overridable component {
   substate field cache;

   public boolean contains(int el);
     // Modifies: this.cache
     // Effects: Returns true iff el is in this.elements.

   protected void uncache(int el);
     // Modifies: this.cache
     // Effects: Removes el from this.cache
 };

 ..other methods elided

};
```

Figure 5.3: Informal specification for `IntCollection`.

Holder: **trait**

includes Int, Sequence(Int, IntSequenceSort), StateTrait

introduces
new: Obj $\rightarrow$ IntHolder
ins: IntHolder, Int $\rightarrow$ IntHolder
tag: IntHolder $\rightarrow$ Obj

del: IntHolder, Int $\rightarrow$ IntHolder
mem: IntHolder, Int $\rightarrow$ Bool
$goodSeq$: IntHolder, IntSequenceSort $\rightarrow$ Bool
$measure$: IntHolder $\rightarrow$ Int

asserts
IntHolder **generated by** new, ins
$\forall\, o, o'$: Obj, h: IntHolder, s: IntSequenceSort, i, i': Int
$tag(new(o)) = o$
$tag(ins(h, i)) = tag(h)$

$\neg mem(new(o), i)$
$mem(ins(h, i), i') = (i = i' \vee mem(h, i'))$

$measure(new(o)) = 0$
$measure(ins(h, i)) = (mem(h, i)\, ?\, measure(h) : 1 + measure(h))$

$goodSeq(h, s) \Rightarrow (mem(h, i) \Leftrightarrow i \in s)$
end Holder

Figure 5.4: Holder trait.

abstract values it can take on. We define these subspaces by tagging values with Objs (object identifiers). The subspace of abstract values that an object o can take on is the set of values tagged with o. Fig. 5.4 gives a trait for IntHolder, a value space for the abstract state of IntCollection. The *new* operation takes an *Obj* argument o and returns an empty IntHolder value tagged by o. The axioms for *tag* (which returns the object tagging a value) indicate that any IntHolder value generated by inserting integers into *new(o)* is also tagged by o. Thus, we see that associated with each Obj is a whole space of IntHolder values.

Tagging values gives us the freedom to define properties for both IntHolder as a whole and also for subspaces of IntHolder. We can define a property on all IntHolder values by quantifying over all tags. For example, the trait in Fig. 5.4 asserts general properties of *mem* and *goodSeq* that apply to all IntHolder values. We can define a property on subspaces of IntHolder by quantifying over the objects for which the property should apply. For example, the assertion:

$$\forall\, h : \texttt{IntHolder}, i : \texttt{Int} \cdot InstanceOf(tag(h), \texttt{IntSet}) \Rightarrow ins(h, i) = ins(ins(h, i), i)$$

says that insertion is idempotent for IntHolder values that are tagged by an instance of IntSet. ($InstanceOf(o, C)$ is a boolean-valued function returning true when Obj o is a direct or indirect instance of class C.)

A formal specification for IntCollection is given in Fig. 5.5. This specification uses

specialization specification `IntCollection` {

  uses `Integer, Set(Int, IntSet), Holder;`
  state `[` h`:IntHolder,` c`:IntSet ];`
  invariant $\mathbf{this} = \mathit{tag}(\mathbf{this}^{\rho}.h) \;\wedge\; e \in \mathbf{this}^{\rho}.c \;\Rightarrow\; \mathit{mem}(\mathbf{this}^{\rho}.h, e)$;

  overridable component {
    substate `[` h`:IntHolder ];`

    public `void addElement(int el)` {
      Modifies **this**;
      Ensures $\mathbf{this}^{post}.h \;=\; \mathit{ins}(\mathbf{this}^{pre}.h, \texttt{el})$;
    };

    public `boolean removeElement(int el)` {
      Modifies **this**;
      Ensures $\mathbf{this}^{post}.h \;=\; \mathit{del}(\mathbf{this}^{pre}.h, \texttt{el}) \;\wedge\; \mathbf{result} \;=\; \mathit{mem}(\mathbf{this}^{pre}.h, \texttt{el})$;
    };

    public `IntEnumeration elements()` {
      Ensures $\mathbf{result}.index \;=\; 0 \;\wedge\; \mathit{goodSeq}(\mathbf{this}^{pre}.h, \mathbf{result}.seq)$;
    };
  };

  overridable component {
    substate `[` c`:IntSet ];`

    public `boolean contains(int el)` {
      Modifies $\mathbf{this}.c$;
      Ensures $\mathbf{result} \;=\; \mathit{mem}(\mathbf{this}^{pre}.h, \texttt{el})$;
    };

    protected `void uncache(int el)` {
      Modifies $\mathbf{this}.c$;
      Ensures $\texttt{el} \notin \mathbf{this}^{post}.c$;
    };
  };
};

Figure 5.5: Formal specialization specification for `IntCollection`

specialization specification `IntSetSSpec2` {

  uses `Integer, Set(Int, IntSet), Holder;`
  state [h:`IntHolder`, c:`IntSet`];
  invariant $\mathbf{this} = tag(\mathbf{this}^{\rho}.h) \;\;\wedge\;\; e \in \mathbf{this}^{\rho}.c \;\Rightarrow\; mem(\mathbf{this}^{\rho}.h, e)$;

  asserts
    $\forall h : \texttt{IntHolder}[\mathbf{this}],\; s : \texttt{IntSequenceSort},\; i, i' : \texttt{Int}$
      $ins(ins(h, i), i') = ins(ins(h, i'), i)$
      $\wedge\; ins(h, i) \;=\; ins(ins(h, i), i)$
      $\wedge\; mem(del(h, i), i') \;=\; (i \neq i' \;\wedge\; mem(h, i'))$
      $\wedge\; goodSeq(h, s) \;\Rightarrow\; len(s) = measure(h)$;

  `.. rest of specification is same as for IntCollection`
};

Figure 5.6: Using class assertions to specialize `IntHolder`

`IntHolder` as its abstract state. The invariant **this** $=$ $tag(\mathbf{this}^{\rho}.h)$ ensures that the subspace of `IntHolder` values associated with object x are used to describe the abstract state of x.

5.2.2 Specializing extensible sorts

To *specialize* an extensible sort is to assert properties about interesting subspaces of the sort. "Interesting" subspaces include spaces associated with all the instances of some class and instances of all classes that inherit a given class component.

Class assertions are used to specialize an extensible sort for all instances of a class. A class assertion appears in the specialization specification of a class. These assertions can be quantified over sort expressions of the form "*Sort*[**this**]," where *Sort* is an extensible sort. Inside the class C, the class assertion "**asserts** $\forall x$: *Sort*[**this**] $\cdot$ P" is interpreted as:

$$\forall x : Sort \cdot InstanceOf(tag(x), C) \Rightarrow P$$

This desugaring generalizes in the expected manner for multiple occurrences of quantification over sorts of the form *Name*[**this**].

Fig. 5.6 shows a class assertion that asserts set-like properties for those `IntHolder` values associated with instances of `IntSet`. The assertion here is interpreted as:

$$\begin{array}{l}
\forall h : \texttt{IntHolder},\; s : \texttt{IntSequenceSort},\; i, i' : \texttt{Int} \\
\;\; InstanceOf(tag(h), \texttt{IntSet}) \Rightarrow \\
\;\;\;\; (ins(ins(h, i), i') = ins(ins(h, i'), i) \\
\;\;\;\;\; \wedge ins(h, i) = ins(ins(h, i), i) \\
\;\;\;\;\; \wedge mem(del(h, i), i') = (i \neq i' \wedge mem(h, i')) \\
\;\;\;\;\; \wedge goodSeq(h, s) \Rightarrow len(s) = measure(h))
\end{array}$$

Note that this asserts properties of the sort `IntSet` (Fig. 4.2), idempotence and commutativity, that are missing from `IntHolder`.

Class assertions constrain both direct instances of a class and also instances of all subclasses of the class. Sometimes, constraining all subclasses limits reuse. For example, the code for `IntSet` in Fig. 1.2 implements the specification in Fig. 5.6. However, using this specification for `IntSet` limits potential reuse. The specification in Fig. 5.6 restricts all subclasses to implement sets when in fact `IntSet` could be used to implement other collection classes (*e.g.*, multisets).

This is really a variation on the story of Sec. 5.1: different specifications are needed for describing a component and for constraining subclasses that replace a component. For instances of `IntSet` and for subclasses that inherit the `elements` component, we want a specific specification that explains that the `elements` component is implementing *sets* in particular. For subclasses of `IntSet` that override the `elements` component, we want a general specification like the one for `IntCollection` in Fig. 5.5 that says that `elements` must implement some kind of `IntHolder`—but any kind of `IntHolder`.

We can handle this dual requirement by associating assertions with *components* rather than with classes. For example, another specification of `IntSet` is given in Fig. 5.7. This specification is the same as the one in Fig. 5.6 except that the specializing assertion for `IntHolder` has been moved into the `elements` component. Inside a class component G, the component assertion "**asserts** $\forall x : Sort[\textbf{this}] \cdot P$" is interpreted as:

$$\forall x : Sort \cdot (tag(x) \text{ inherits } G) \Rightarrow P$$

In other words, the abstract-value space of all subclasses that inherit the code of G are subject to the assertion of G. The code for methods in G may assume P. For example, when `IntSet` is specified by Fig. 5.7, the implementation of `elements` can assume that insertion is idempotent, so the code for `addElement` (Fig. 1.2) need not add an element if it is already in `elements`. Code for methods outside of G may *not* assume P. For example, consider the following code for `contains`:

```
public overridable boolean contains(int el) {
  if (c_valid && c_val == el) return true;
  if (this.removeElement(el)) {
    c_valid = true; c_val = el;
    this.addElement(el) // Bug! Assumes insertion is commutative
    return true;
  } else return false;
};
```

This code assumes that element insertion is commutative. This assumption would be correct if `IntSet` were specified by Fig. 5.6. However, if `IntSet` is specified by Fig. 5.7, this assumption is incorrect outside of the `elements` component.

specialization specification `IntSet` {

  uses `Integer, Set(Int, IntSet), Holder;`
  state `[` h`:IntHolder,` c`:IntSet ];`
  invariant $\textbf{this} = \mathit{tag}(\textbf{this}^{\rho}.h) \;\wedge\; e \in \textbf{this}^{\rho}.c \Rightarrow \mathit{mem}(\textbf{this}^{\rho}.h, e)$;

  class component { `// ``elements component''`
    substate is `[` h`:IntHolder ]`

    asserts
      $\forall s$`:IntSequenceSort,` i, i'`:Int`

$$\begin{aligned} & \mathit{ins}(\mathit{ins}(\textbf{this}.h, i), i') = \mathit{ins}(\mathit{ins}(\textbf{this}.h, i'), i) \\ & \wedge\ \mathit{ins}(\textbf{this}.h, i) \;=\; \mathit{ins}(\mathit{ins}(\textbf{this}.h, i), i) \\ & \wedge\ \mathit{mem}(\mathit{del}(\textbf{this}.h, i), i') \;=\; (i \neq i' \;\wedge\; \mathit{mem}(\textbf{this}.h, i')) \\ & \wedge\ \mathit{goodSeq}(\textbf{this}.h, s) \;\Rightarrow\; \mathit{len}(s) = \mathit{measure}(\textbf{this}.h); \end{aligned}$$

    `.. rest is same as for IntCollection`
  };

  `.. rest same as for IntCollection`
};

Figure 5.7: Using class assertions to specialize `IntHolder`

Chapter 6

Design implications

This chapter looks at the design implications of the previous chapters. It focuses on the design implications for class libraries, but it also looks at design implications for programming languages.

The goal of this report is to show how to improve the specialization interfaces of class libraries. The previous chapters showed a methodology—a unit of modularity with associated documentation and reasoning techniques—that achieves modularity without sacrificing extensibility. However, as suggested in an earlier chapter, a good methodology allows good designs but does not necessitate them. Good use of the methodology—*i.e.*, good design—is required as well. This chapter explores some principles of good design implied by the methodology.

There are two levels in the design of class libraries. One level is the decomposition of the overall library into type and class hierarchies. The other level is the design of the interfaces of individual classes. This chapter considers the latter, the design of specialization interfaces of individual classes. It presents a series of tips for designing these interfaces and also describes how languages can support these tips.

Sec. 6.1 describes the examples used to illustrate the points of this chapter. Sec. 6.2 describes control abstractions, a building block like class components for designing specialization interfaces. This chapter does not discuss control abstractions in detail, but it is important to understand how control abstractions and class components fit together. Sec. 6.3 discusses the design of specialization interfaces using class components. Our approach is to concentrate on identifying a good abstract representation and on picking representation categories for the fields of the abstract representation. Decomposing this into class components follows naturally once the abstract representation is designed.

The last section of this chapter looks at the implications of class components on another area of design: the design of programming languages. It suggests a construct for expressing overridable class components in one's code. It shows how such a construct can be used to enforce many of the rules discussed in Chapter 3.

6.1 Design examples

This chapter illustrates its points using three examples. Two of the examples have already been seen: `IntSet` and `Rd`.

The third example is a new one: "views," a class central to most GUI libraries. A view is a

Functionality	Openstep	ET++
Root class:	`NSObject`	`Object`
Responder:	`NSResponder`	`EvtHandler`
View:	`NSView`	`VObject`

Table 6.1: Class hierarchies of two GUI libraries.

rectangular region on the screen that programs can draw into and that responds to user events such as mouse clicks and key strokes. Views are arranged in a view hierarchy, with a superview containing a number of subviews containing subsubviews and so on. Mouse-click events are directed to the lowest view in the view hierarchy containing the click; other input events, such as keystrokes and commands generated by menus, are directed to the view that currently has the input focus.

We compare and contrast the view classes in two GUI libraries, Openstep [Next94] and ET++ [Weinand95]. We consider both to be well-designed libraries, an opinion that has been seconded in the literature (*e.g.*, [Holzle93],[Lewis95]). Openstep and ET++ are similar in their class hierarchies and in the general functionality of various classes. However, the details of their specialization specifications are different.

The view-related part of the class hierarchies of Openstep and ET++ are shown in Table 6.1. In both libraries, the immediate superclass of the view class is a "responder" class that factors out functionality for processing mouse, keyboard, and other events. The immediate superclass of the responder class is the root class of the library, which provides generic functionality such as object copying, equality testing, and dynamic type identification.

Table 6.2 lists some of the abstract-state fields of the view class and their representation categories in the different libraries. The `geometry` field contains the location and size of the view. The `nextResponder` points to the next responder in the program's responder chain (an event such as a keyboard stroke is passed along the responder chain until a responder chooses to process it). The `superview` and `subviews` fields contain the view hierarchy. The `observers` field is unique to the ET++ library. The `observers` field contains a list of objects registered to receive notification when a `VObject` changes; it is motivated by the change-propagation mechanism in the SmallTalk Model-View-Controller paradigm [LaLonde91].

A specialization specification for the class `View` is given in Fig. 6.1. This specification is referred to in our design discussions later in this chapter. `View` is close to the ET++ view class,

Field	Openstep	ET++
`geometry`	final, encapsulated	final, exposed
`nextResponder`	final, encapsulated	deferred
`superview`	final, encapsulated	final, encapsulated
`subviews`	final, encapsulated	overridable
`observers`	—	overridable

Table 6.2: Abstract-state fields of two view classes.

```
specialization specification ViewSSpec {

 state field subviews; // Set of Views
 state field superview; // Single View, or nil
 ..other fields elided

 final component {
   substate field superview;

   public View getSuperview();
     // Effects: Returns superview.

   public void setSuperview(View newSuper);
     // Modifies: this
     // Effects: Sets superview to newSuper.
 };

 overridable component {
   substate field subviews;

   public void addSubview(View newSub) throws BoundsException;
     // Modifies: this, newSub
     // Override effects: If subviews is full, throws BoundsException,
     //   otherwise, inserts newSub into subviews and sets
     //   newSub.superview to this.
     // Inherited effects: Always throws BoundsException because the
     //   default representation of subviews has zero capacity.  Subclasses
     //   that expect to have subviews must override this component.

   public void removeSubview(View oldSub)
     // Modifies: this, oldSub
     // Effects: If oldSub in subviews, removes it and
          sets oldSub.superview to null.

   public ViewEnumeration subIter();
     // Effects: Returns an enumeration that yields the
     //   subviews of this in turn.
 };

 // Not accessors for any state, so outside of any component
 public final void drawAll();
   // Effects: Draws this by calling drawSelf, then
   //   draws each subview by calling drawAll.

 public deferred void drawSelf();
   // Effects: draws this.

 ..other methods elided
};
```

Figure 6.1: Partial specialization interface of `View`.

```
class View {

  public void addSubview(View newSub) throws BoundsException
    { throw new BoundsException(); };

  public void removeSubview(View oldSub)
    { };

  public ViewEnumeration subIter()
    { return new EmptyViewEnumeration(); };

  ...
};
```

Figure 6.2: Representation of `subviews` in `View`.

but avoids details that are not relevant to our discussion. We consider here only two abstract-representation fields of `View`: `superview`, containing the superview of a view, and `subviews` containing the set of subviews of a view.

The specification in Fig. 6.1 shows two class components of the `View` class, a final component that maintains the `superview` state and an overridable component that maintains the `subviews` state. In addition, Fig. 6.1 includes two methods, `drawAll` and `drawSelf`, that are not accessors of any component and thus are outside of any component; these are discussed further in the next section.

The default representation of `subviews`, given in Fig. 6.2, holds no subviews: any attempt to add a subview results in a `BoundsException` being thrown. This makes `addSubview` a good candidate for having both an assumed and an particular specification (Sec. 5.1). The assumed specification of `addSubview` describes the behavior required of all subclasses: `addSubview` adds a new subview to a `View`, subject to an unspecified bound on the number of allowable subviews. The particular specification refines this general specification, indicating that the default representation of `subviews` holds *no* subviews so `addSubview` will always throw `BoundsException`.

6.2 Control abstractions

Class components are one building block that can be used in the design of specialization interfaces. Control abstractions are another. Control abstractions support extensibility for behavior. Although control abstractions are beyond the scope of this report, no realistic discussion of the design of specialization interfaces can completely ignore them. This section briefly describes control abstractions and discusses how they interact with class components. (A detailed discussion of control abstractions can be found in [Kiczales91].)

A *control abstraction* is a method whose behavior can be customized by overriding methods that it calls. Control abstractions are final or overridable methods containing algorithms in which key details are embodied by calls to deferred and overridable methods called *specializing methods*. By overriding these specializing methods, subclasses can customize the behavior of control abstractions.

```
public final void drawAll() {
  this.drawSelf();
  for(ViewEnumeration e = this.subIter(); e.hasMoreElements(); )
    e.nextElement().drawAll();
};
```

Figure 6.3: Implementation of `drawAll`.

The method `drawAll` (in `View`) is an archetypical control abstraction. This method is responsible for drawing the entire view, including graphics handled by the view itself and any graphics handled by subviews. As shown in Fig. 6.3, `drawAll` does this by first calling `drawSelf` to draw its own graphical content, and then calling the `drawAll` method of each of its subviews. Subclasses specialize `drawAll` by overriding its specialization methods `drawSelf` and `subIter` (overriding `subIter` implies overriding the entire `subviews` component). For example, a dialog box is typically implemented as a superview containing a number of "controls" (*e.g.*, buttons) as subviews. The `drawAll` behavior of `DialogView` is defined by overriding `drawSelf` to draw a background for the dialog and overriding `subIter` to yield the controls inside the dialog.

Class components and control abstractions can overlap. An accessor of a class component can be a specializing method of a control abstraction. For example, in `View`, `subIter` is a specializing methods of `drawAll` and also an accessor of the `subviews` component. Also, a method can be both an accessor and a control abstraction. For example, in `Rd`, `getChar` is both an accessor of the `buffered` component and also a control abstraction whose specializing method is `nextChunk`.

Sometimes a method is purely a control abstraction (`drawAll`) or purely a specializing method (`drawSelf`) and does not directly access the representation of any state. In specifications, it is convenient to place such methods outside of any **component** clause. When reasoning about these methods, we treat them as each belonging to a particularly simple component, a component having no substate and only one method. Because we consider these methods as belonging to single-method components, we sometimes call them *singletons*.

6.3 Designing specialization interfaces

Choosing a good abstract representation is important to designing the specialization interfaces of classes. We recommend the following, iterative process for designing abstract representations:

1. Select a preliminary abstract representation and abstract representation invariant.
2. Pick representation categories for fields of the abstract representation, *i.e.*, categorize the fields as final, deferred, and overridable.
3. Group overridable fields into components and pick accessors for each component.
4. Iterate, taking into account the implementations and subclasses enabled and ruled-out by the current iteration.

The following subsections discuss the first three steps in turn.

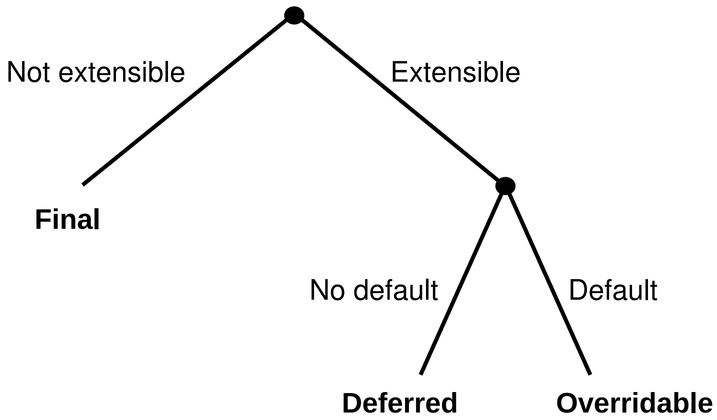


Figure 6.4: Decision tree for category selection.

6.3.1 Selecting a preliminary abstract representation

The first step ties the process of designing specialization interfaces into the overall process for designing the class library. By the time the designer gets around to designing the specialization interface, the abstract state of the instance interfaces has already been decided. The abstract state of the specialization interface is picked as an abstract representation of that instance specification. Many times, as in the case of `View`, the abstract states of the instance and specialization specifications can be the same. Other times, as in `Rd` and `IntSet`, extra state is revealed in the specialization interface.

6.3.2 Picking representation categories

Once a preliminary abstract representation has been selected, representation categories for each field must be picked. In all likelihood, the designer already has a good idea of what the representation categories will be, but it is worth considering some of the design considerations that go into this choice.

Fig. 6.4 gives a decision tree for choosing a representation category for a field. The designer must first decide whether or not the field should be extensible, *i.e.*, whether or not subclasses should be able to provide their own representations for a field. If the field is extensible, the designer must decide if it should be overridable or deferred.

Each of these decisions is discussed below in turn. These discussions illustrate their points using both `Rd` and `View`. The details of `Rd` and `View` given here differ from those in the class libraries that inspired them, but the discussion in this section applies to the actual libraries. The representation categories for `Rd` and `View` are summarized in Table 6.3.

6.3.2.2 Extensible vs. non-extensible

The first step in choosing a representation category for a field is to decide whether or not it should be extensible. The benefit of extensibility is that it fosters reuse both by allowing customized functionality and by allowing specialized implementation trade-offs at different points in the class hierarchy. The deferred `unbuffered` field of `Rd` allows customized functionality. The deferred field allows

Category	`Rd` class	`View` class
Deferred	`unbuffered`	—
Final	`buffered`	`superview`
Overridable	—	`subviews`

Table 6.3: Examples representation categories.

different subclass of `Rd` to use different devices (*e.g.*, disks, network connections) to represent the unbuffered part of a reader.

The overridable `subviews` field allows customized space trade-offs. The usage of `subviews` shows a pattern that can be exploited: the view class hierarchy has a large number of leaf classes, *i.e.*, views like buttons whose instances have no subviews, and also a large number of "filter" view classes, views like "boardered view" whose instances have only one subview. By making `subviews` extensible, it can be represented in a space-efficient manner: leaf classes can represent `subviews` without using any space, and filter classes can represent `subviews` with only an instance variable rather than with a heap-allocated collection object used by more general `View` classes.

Extensible state can lead to more reuse. ET++ makes extensive use of extensible state to reducing the default size of a view object, making views lightweight. This encourages their reuse at a fine level of granularity, *e.g.*, to represent individual cells in a spreadsheet program. Openstep, on the other hand, does not use extensible state (it gets extensibility from control abstractions). In Openstep, views take a lot of space, so programmers avoid them at a fine-level of granularity, instead developing their own mechanisms for fine-grain drawing and event handling.

Despite the reuse that can come from extensible state, we make the following, conservative recommendation: An abstract representation field should be final unless there is a specific case for making it extensible. For example, the structure of `superview` is very uniform: every `View` has exactly one parent (except the root), so there is no reason to make it extensible. There are two reasons for this recommendation. First, with extensible fields there are more opportunities for errors in both designs and implementations. Being conservative about extensibility leads to more reliable software.

Second and more important, being conservative allows designers to put-off deciding exactly how a class should be extensible until such time as there are real-world needs to drive design. It is relatively easy to change a field from final to overridable without breaking existing classes. However, it is hard to change a representation strategy that has been exposed in the specialization interface, especially in third-party libraries, because such changes can break subclasses. For example, for `View`, a specializer that finds a compelling reason for making `superview` overridable can negotiate to have `View` changed, and this change will not break existing `View` subclasses. By being conservative about exposing representation strategies, builders of class libraries can wait until their "customers" come to them with real-world examples of where extensibility is needed. This will lead to better design decisions.

6.3.2.2 Overridable vs. deferred

If a designer chooses to make an abstract representation field extensible, then the designer must next choose between overridable or deferred. This choice depends on whether or not there exists a suitable default representation for the field. When a field is made extensible for behavior reasons (*i.e.*, to allow for customized functionality), there is often no sensible default representation. This case applies to `unbuffered`.

When a field is made extensible for performance reasons (*i.e.*, to allow for customized trade-offs), there is often a default representation that can be given. This case applies to `subviews`. The `subviews` field follows a general pattern for overridable fields: usually a low-space representation is chosen as the default, under the assumption that subclasses will replace it as needed with a higher-space representation that is either faster or provides more functionality.

6.3.2.2 Final fields

If a designer chooses to make an abstract representation field non-extensible, then the designer must decide whether or not to expose it. We have been assuming that final state is encapsulated. We feel this is the best choice. Encapsulated fields are represented with private instance variables. A class cannot access the representation of its superclass's encapsulated final fields, but instead must access them indirectly by calling superclass methods.

For performance reasons, final abstract representation fields are sometimes exposed by using protected instance variables to represent them, allowing subclasses to access them directly rather than having to call superclass methods. To support direct access, the representation of an exposed field must be well documented. This means documenting the instance variables in the field's representation, the invariants these variables obey, and the abstraction function relating these variables to abstract-values of the field they represent ([Edwards96]).

In the Modula-3 `Rd` class (but not in Fig. 2.4), `buffered` is an exposed field. Reading characters from input streams is often part of the critical path of the kinds of systems programs Modula-3 is meant to support. The designers of `Rd` exposed the field to minimize the performance overhead on this critical path.

Final fields should not be exposed unless there is a really good reason. The trade-off between exposing and encapsulating a final field is the old one between performance on the one hand and safety and maintainability on the other. By allowing direct manipulation, exposed fields allow subclasses to avoid the method-call overhead inherent in indirect manipulation. However, exposing fields requires subclasses to maintain invariants and reason about superclass abstraction functions, potential sources of error. Also, a change to the representation of an exposed field can invalidate code in subclasses, introducing maintenance difficulties. In the long-term, exposing fields can actually harm performance. Once a field is exposed, it is basically impossible to change its representation, especially in class libraries. This makes it impossible to change to a more efficient representation, and it makes it impossible to make the field overridable, which would let subclasses provide optimized representations.

6.3.3 Picking accessors for components

Sufficiency of interfaces is an old issue in the design of data abstractions [Kapur88]. A sufficient interface is one that provides enough functionality to allow effective access to objects; an insufficient interface is one whose methods make it hard or impossible to manipulate objects. Sufficiency is also an issue in the specialization interface, but at a finer level of granularity.

Class components are really data abstractions embedded within the specialization interface. The issue of sufficiency applies to each of these embedded data abstractions. For example, the `subviews` component of `View` would not be of much use without the `subIter` method to allow code outside the component to access the set of subviews. In general, the methods of a component should form a sufficient interface to the component's substate.

Making sure that each component has a sufficient interface is an important part of picking methods for a class, but it is not the only issue involved. Other considerations include a sufficient interface for instantiators, and also the control abstraction in the specialization interface. A full methodology for designing method suites is beyond the scope of this chapter.

6.4 Language design

Class components have implications for the design of programming languages as well as for the design of classes. The central implication is to add overridable class components as a checked construct of classes. With such a construct, compilers can check that the instance variables of overridable components are only accessed by their own code, that subclasses override components as units, and that subclasses shadow components correctly. This construct also facilitates a space-saving optimization.

Deferred and final components do not suggest any language features. Deferred and final components affect the way programmers think about the design and implementation of classes, but they do not impose restrictions on code the way overridable components do. For example, the convention for deferred components says that deferred abstract state should be accessed by calling deferred methods. Deferred abstract state is a specification concept, so there is no compiler-checkable restriction here. The convention for final components says that any method in a class may access the instance variables representing the class's final abstract state, so again there is no compiler-checkable restriction here. (As mentioned in Sec. 2.2, the purpose of grouping of methods into final components is not to enforce implementation restrictions but to help break up a class into smaller, more digestible pieces.)

The conventions for overridable components, in contrast, define a number of restrictions on the implementations of classes, restrictions that can be enforced by compilers. Sec. 6.4.1 below describes what a construct for overridable components might look like. Sec. 6.4.2 describes how such a construct can be used to check the local code of a class. Sec. 6.4.3 describes how this construct can be used to ensure that subclasses correctly override and shadow superclass components. Finally, Sec. 6.4.4 explains how such a construct can be used for a space-saving optimization.

In the discussion below, we assume that overridable components are supported in the language's syntax and that checking is done by the compiler. An alternative is to use comment-embedded annotations to support overridable class components and to do checking with a tool that works beside the compiler. This alternative approach is taken by LCLint to support data abstraction in C [Evans96]. Except for the discussion on optimization, the discussion below applies equally to both approaches.

6.4.1 A construct for overridable components

Support for overridable class component requires just a small amount of syntax. What is needed is a construct in class definitions for grouping declarations of methods and instance variables. These groups constitute the overridable components of the class.

Fig. 6.5 gives an implementation of `IntSet` using a class component construct (*c.f.* Fig. 1.2). The syntax designating class components is trivial, yet it conveys a lot of information. This information is useful to programmers because it documents the internal modules making up the class. And, as shown in the following subsections, this information is useful for mechanical checking and for optimization.

An overridable method that appears outside of a component construct is a singleton. From the perspective our methodology, such a method belongs to a class component consisting of only the method itself and no other methods or any abstract-state fields. As suggested in Sec. 6.2, singletons are typically pure control abstractions or pure specializing methods for control abstractions.

6.4.2 Overridable components as scoping units

By treating the component construct as a scoping unit, and by treating the declarations of instance variables and final methods that appear inside a component as private to the component, useful checks can be made on the local code of a class.

One part of the programming conventions for overridable components described in Chapter 2 is that only the methods of a component may access the representation of the state assigned to the component. With a component construct, this convention can be checked. Instance variables declared inside a component construct should be visible only to code inside that component. Attempts to access instance variables from outside the variables' component should be an error. (Inside a component construct, there is not need to clutter the declarations of instance variables with **private**, *e.g.*, compare Fig. 6.5 to Fig. 1.2.)

Another part of the conventions for overridable components is helper methods, which are private, final methods inside an overridable component. The helper methods of a component may access the component's instance variables, but they can only be called by other methods in the component. The component construct allows enforcement of this part of the convention as well. Any final methods declared inside a component should be treated as helper methods. Such a method should have access to the instance variables of the component, but it should only be visible to other methods in the component.

Yet another part of the conventions of Chapter 2 is the restriction on binary methods. This restriction says that the code for methods in overridable component G can only access the instance variables of G for **this** and not for other arguments. The component construct allows compilers to check this restriction.

6.4.3 Overridable components as units of override

Another aspect of the programming conventions for overridable components is restrictions on how they can be overridden. When a subclass replaces a component, it must replace it as a unit, replacing all methods of the component and not just some of them. With the component construct, compilers can enforce this requirement.

```
class IntSet {
  overridable component { // elements component
    IntVector els = new IntVector(); // Elements of IntSet

    public void addElement(int el) {
      if (! els.contains(el)) els.addElement(el);
    };

    public boolean removeElement(int el) {
      this.uncache(el); // Maintain cache validity
      return els.removeElement(el); // Call remove method of IntVector
    };

    public IntEnumeration elements() {
      return els.elements();
    };
  };

  overridable component { // cache component
    int c_val; // Value currently in cache
    boolean c_valid = false; // True only if c_val is valid

    public boolean contains(int el) {
      if (c_valid && c_val == el) return true;
      for(IntEnumeration e = this.elements(); e.hasMoreElements(); )
        if (el == e.nextElement()) {
          c_valid = true; c_val = el;
          return true;
        };
      return false;
    };

    protected void uncache(int el) {
      if (c_val == el) c_valid = false;
    };
  }

  .. more methods elided
};
```

Figure 6.5: Implementation of `IntSet` using component construct.

Instead of replacing a superclass component, a subclass might shadow it, *i.e.*, it might replace some of its methods without replacing its representation. We recommend that the language syntax includes an explicit indication of shadowing; for example, components that shadow superclass components might have the keyword **shadow** in front. An explicit indication of shadowing allows specialized checking of shadowing. Invocations through **super** can be restricted to subclass components that shadow superclass components. Further, a subclass component shadowing superclass component G can be restricted to use **super** only when invoking superclass versions of methods in G. The compiler can also check that the subclass does not regroup a shadowed component. That is, if methods m and n are overridden by the subclass and both belong to the same, shadowed component of the superclass, then m and n must belong to the same component in the subclass.

6.4.4 Eliminating unused instance variables

When a component is replaced by a subclass, the subclass inherits the old representation of the component's substate but does not use it. For example, `IntSet2` (Fig. 1.3), which replaces the `cache` component of `IntSet`, inherits the instance variables `c_valid` and `c_val` but does not use them. The class-component construct allows compilers to optimize away this wasted space. In particular, when a subclass replaces (rather than shadows) a component of its superclass, the subclass need not inherit the instance variables declared inside the method group. That is, at runtime, space need not be allocated in the object for these variables. Of course, the instance variables of shadowed components should *not* be optimized away because they are used by the subclass.

This optimization will require some advances in compiler technology. For example, compilers typically compile accesses to instance variables as fixed offsets from a base pointer. If subclasses can remove some instance variables, these offsets will no longer be fixed. Constructors also pose a challenge. Constructors need to be able to access all the instance variables of a class, even those defined inside overridable components. When a constructor is inherited into a class that optimizes-away some of the instance variables accessed by the constructor, the constructor code will have to be changed to read and write a temporary variable rather than actual instance variables. We have not pursued solutions to these problems, so at this point the possibility of optimizing-away instance variables is still speculative.

Chapter 7

Multiple inheritance

In discussing single inheritance, we started from current practice. We assumed standard language mechanisms and were inspired by existing, successful designs. The single-inheritance results do not imply new approaches to structuring class hierarchies, but rather apply at the level of specialization interfaces of individual classes. Multiple inheritance is different. There are few large, successful libraries that make aggressive use of multiple code inheritance. Thus, while for single inheritance we were inspired by today's best designs, there are no such starting points for multiple inheritance. Also, as discussed in [Snyder87], the multiple inheritance models of most languages are inherently non-modular. Thus, we cannot even start with a standard language model.

Rather than start with conventional language and design assumptions, this chapter starts with the ideas of the previous chapters, ideas on the design, documentation, and validation of specialization interfaces. Based on these ideas, it suggests a mechanism for multiple inheritance that solves some problems with single-inheritance designs.

One problem with single inheritance is a lack of coherence. In most object-oriented libraries, classes perform many functions rather than a single, well-focused purpose as demanded by the principle of *coherence* ([Pressman92, Booch94]). In single-inheritance designs, classes serve two purposes: they are templates from which useful instances can be instantiated, and they are superclasses from which subclasses can inherit. In addition, the specialization specification of overridable components serves two purposes: it both describes the class's own version of components and constrains subclass versions. The result is up to three different specifications for overridable methods, clearly a somewhat complicated situation.

Another problem with single inheritance is tight the coupling of component implementations. Single inheritance forces designers to commit to particular combinations of implementations of different components. For example, previous chapters have shown two implementations of the `elements` component and two implementations of the `cache` component packaged into three classes. By requiring designers to commit to particular combinations of component implementations, single inheritance rules out some combinations, limiting the potential for reuse.

These problems can be solved by turning class components into separate classes. A group of these smaller classes, each implementing different facets of an object's functionality, can be combined to form instantiable classes. We call these smaller classes *mixins*, after the process of "mixing" them together to form larger classes. ("Mixin" was first coined in the Flavors community [Weinreb81], but our use follows [Booch94].)

The first section below describes how mixins can be supported in Java with only minimal lan-

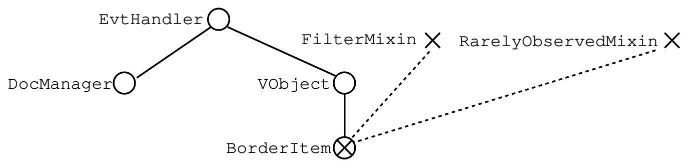


Figure 7.1: Example type and class hierarchy.

guage changes. The next section explains how a simplified subset of the specification and verification techniques of the previous chapters applies directly to mixins. Sec. 7.3 goes into more detail on the advantages of mixin-based designs over designs based on single inheritance.

7.1 Mixins

A multiple inheritance mechanism suitable for our purposes can be added to Java by adding a new kind of class called a "mixin." Like Java's existing classes, mixins define a set of variables and methods that can be inherited into subclasses. Unlike Java's existing classes, mixins cannot be instantiated, nor do they define types (*i.e.*, the name of a mixin cannot be used in type expressions). Classes defined by Java's existing **class** module are called *instantiable classes* to distinguish them from mixins.

In our modified language, programmers can build instantiable classes by subclassing from one or more mixins, but they cannot subclass from other instantiable classes. In this language, the class hierarchy is completely separated from the type hierarchy. The type hierarchy consists of Java interfaces at the inner nodes and instantiable classes at the leaves. The class hierarchy consists of mixins at the inner nodes and again instantiable classes at the leaves. An example hierarchy is given in Fig. 7.1. In this figure, O's stand for types, X's stand for classes. Instantiable classes, which define both object types and classes, join the two hierarchies at their leaves, so they are represented by an O and X superimposed. Solid lines stand for subtyping, and dashed lines for subclassing

Mixins can be added to the syntax of Java with only a slight modification of the grammar in [Javabeta95]:

$$\begin{array}{rcl}
\textit{class_declaration} & \rightarrow & \textit{class_modifiers}\ \textbf{class}\ \textit{identifier}\left[\textit{super}\right]\left[\textit{interfaces}\right]\textit{class_body} \\
\textit{mixin_declaration} & \rightarrow & \textbf{mixin}\ \textit{identifier}\left[\textit{super}\right]\left[\textit{interfaces}\right]\textit{class_body} \\
\textit{super} & \rightarrow & \textbf{mixins}\ \textit{mixin}^{*} \\
\textit{mixin} & \rightarrow & \textit{identifier} \mid \textit{identifier}\ \textbf{hide}\ \{\ \textit{identifier_list}\ \}
\end{array}$$

Java's existing *class_declaration* non-terminal does not change; it declares instantiable classes. The new non-terminal *mixin_declaration* declares mixins. Java's existing non-terminal *super* is changed to allow multiple supertypes and also to allow hiding of supertype methods (see below).

The methods in an instantiable class's body must all be final. Mixins contain final and deferred methods. We assume here that mixins do not have overridable methods. A real design would allow overridable methods in mixins. Overridable methods would be used only as specializing methods of control abstractions (Sec. 6.2), not as accessors for overridable state. We do not have anything to say about control abstractions in the context of mixins, so we ignore overridable methods to simplify the discussion.

```
mixin IntSetCore {
  deferred boolean contains(int el);
  deferred void uncache(int el);

  private IntVector els = new IntVector();

  final void addElement(int el)
  {  if (! els.contains(el)) els.addElement(el); };

  final boolean removeElement(int el) {
    this.uncache(el); // Maintain cache validity
    return els.removeElement(el); // Call remove method of IntVector
  };
  final IntEnumeration elements() { return els.elements(); };
};

mixin IntSetSmallCache {
  deferred IntEnumeration elements();

  private int c_val; // Value currently in cache
  private boolean c_valid = false; // True only if c_val is valid

  final boolean contains(int el) {
    if (c_valid && c_val == el) return true;
    for(IntEnumeration e = this.elements(); e.hasMoreElements(); )
      if (el == e.nextElement()) { c_valid = true; c_val = el; return true; };
    return false;
  };
  final void uncache(int el) { if (c_val == el) c_valid = false; };
};

mixin IntSetBVCache {
  deferred IntEnumeration elements();

  private long c_bits; // Used as a bitmap; caches 0 - 63 only

  private boolean c_test(int el)
  { return 0 <= el && el < 64 && c_bits & (1 << el); }
  private void c_set(int el, boolean val) {
    if (0 <= el && el < 64)
      c_bits = (c_bits & ~(1 << el)) | (val ? (1 << el) : 0);
  }

  final boolean contains(int el) {
    if (c_test(el)) return true;
    for(IntEnumeration e = this.elements(); e.hasMoreElements(); ) {
      c_set(el, true);
      if (el == e.nextElement()) return true;
    };
    return false;
  };
  final void uncache(int el) { c_set(el, false); };
};
```

Figure 7.2: Mixins for building `IntSet`.

Fig. 7.2 presents some mixins for building the integer set classes used as examples throughout the thesis. With mixins, `IntSet` from Fig. 1.2 could be expressed as simply:

```
class IntSet mixin IntSetCore, IntSetSmallCache, IntSetExtras
{ }
```

where `IntSetExtras` is another mixin that includes the operations elided in Fig. 1.2. Notice the duality between `IntSetCore` and `IntSetSmallCache`: what is deferred in one is final in the other, allowing them to be mixed together.

Interface types for mixins

Mixins need to have an *interfaces* clauses because the code for a mixin sometimes needs to pass **this** as an argument. For example, consider `FilterMixin`, given in Fig. 7.3, a mixin that might be used to create `View` classes (see Fig. 7.1). In the code for `addSubview`, **this** is passed to the `setSuperview` of another `View` object. The `setSuperview` method expects a `View` as an argument, so for `FilterMixin` to be correct, all classes inheriting from it `FilterMixin` must implement `View`.

For each Java interface I in the mixin's *interfaces* clause, the variable **this** can be used where a Java interface I is expected. The *interfaces* clause of any subclasses of the mixin must include a subtype of I.

Access control

Access control is simpler for mixins than for single-inheritance classes. Because classes now serve only one purpose, they need only two access levels: private and non-private. There is no need for a protected level of protection. Private members can only be accessed inside the class defining them. If an interface I is in a class's *interfaces* clause, then none of I's methods can be private in that class. Only final methods can be private, not deferred methods.

When non-private members are inherited from mixins, by default they remain non-private in the subclass. However, this default can be changed in two ways. First, the **hides** clause of the *mixin* production allows subclasses to hide members of superclasses. Hidden members are treated as if they were private members of superclasses. Only final methods can be hidden. Second, the access control of inherited methods can be changed is with *access declarations*. An access declaration declares are subclass declarations that declare new (stronger) levels of access control for inherited methods (*c.f.* C++ access declarations, [Stroustrup91]).

Semantics of multiple inheritance

With multiple code inheritance, two questions arise that do not arise in the single inheritance case. First, what happens when a class inherits the same field from two or more mixins? Second, what happens when a class inherits the same mixin through multiple paths in the mixin hierarchy?

A class can inherit multiple instance variables with the same name, but at most one of them can be non-private and the rest must be private. A class can also inherit multiple methods with the same name, but at most one of them can be non-private and final; the rest must be either private or deferred. (Java supports overloading, so the "names" of methods include their argument types.)

```
mixin FilterMixin implements View {
  private View m_subview;

  final void addSubview(View v) throws BoundsException {
    if (m_subview != null) throw new BoundsException();
    m_subview = v;
    m_subview.setSuperview(this);
  };

  final void removeSubview(View v) {
    if (m_subview == v) {
      m_subview.setSuperview(null);
      m_subview = null;
    };
  };

  final ViewEnumeration subIter() {
    return new SingletonViewEnumeration(m_subview);
  };
};
```

Figure 7.3: Implementation of `FilterMixin`.

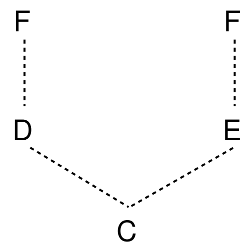


Figure 7.4: Example mixin hierarchy.

When a class inherits private methods and instance variables from multiple superclasses, they are kept distinct in a subclass, even when they have the same name. For example, if class `C` inherits from mixins `D` and `E` (Fig. 7.4), and `D` and `E` both have a private instance variable `i`, then `C` has two instance variables named `i`, one visible only to the code in `D` and the other visible only in `E`. One can think of this in terms of each class having a distinct name space for private fields.

A class cannot inherit non-private instance variables with the same name from multiple superclasses. When a class inherits non-private methods with the same name from multiple superclasses, they are merged, becoming a single method in the inheriting class. For example, if `D` and `E` both have a deferred, non-private method named `m`, then `C` also has only one method named `m`, and this `m` is also `D` and `E`'s `m`. One can think of this in terms of a class and all of its superclasses as sharing a single name space for non-private fields. In the context of this single name space, declarations of deferred methods "declare" fields, while declarations of instance variables and final methods "define" fields. Our rule is this shared name space can contain at most one definition of a field, although it may contain multiple declarations of deferred methods.

C++ does *not* use this merging semantics. Instead, all methods are kept distinct the way we keep private fields distinct. In terms of name spaces, name spaces of sub- and superclasses are distinct for all members, private and non-private. This makes the multiple-inheritance semantics of C++ unsuitable to mixin-style programming (although a clever use of templates ([VanHilst96]) can yield the desired semantics).

When a class inherits the same mixin through multiple paths, the class incorporates two copies of the mixin. This rule embodies the "tree inheritance" approach of [Snyder87]. In our example (Fig. 7.4), `D` and `E` both inherit from mixin `F`. As a result, `C` inherits the methods and instance-variables of `F` *twice*. This implies that `C`, `D`, and/or `E` must hide one copy of each of `F`'s non-private members (except deferred methods) so that `C` does not inherit them twice.

7.2 Specification and verification

The documentation and reasoning techniques of the previous chapters apply directly to the mixin mechanism described above. Mixins are specified with specialization specifications. However, unlike the general specifications of Chapter 4, specifications of mixins have only two components, one for deferred methods and another for final methods. Instantiable classes only have instance interfaces, so they can be specified with object specifications. Java interfaces are also specified with object specifications. Specifications for the mixins in Fig. 7.2 are given in Fig. 7.5. `IntSetElements` can be used to specify `IntSetCore` and other mixins that provide a representation for the elements of an integer set. `IntSetCache` can be used to specify mixins that implement a cache for membership testing, such as `IntSetSmallCache` and `IntSetBVCache`.

Verification of mixins proceeds almost exactly as described in Chapter 4. The specialization specification of the subclass must be a behavioral subclass of *all* the superclass specifications. When a mixin has an *interfaces* clause, each interface specification must be related to the mixin's specialization specification according to the rules for instance interfaces given in Sec. 3.4. Instance invariants can be used for mixins. However, because subclasses can add methods that break such invariants, data-type induction cannot be used to establish instance invariants for mixins. Instead, explicit instance invariants are needed in their specialization interface as described in Sec. 3.4.1.

Instantiable classes are specified with object specifications. However, as explained in previous chapters, instantiable classes cannot be verified directly against instance specifications because the code of inherited methods is not available for inspection. Instead, instantiable classes are verified against specialization specifications that are invented by the verifier strictly for the purposes of verification. The instance specification is then verified by comparing it to this intermediate specialization specification using rules for instance interfaces in Sec. 3.4.

7.3 Program design

Now that the language, specification, and verification mechanisms are defined, we can turn our attention to designing programs using these mechanisms. The impact of mixins on design is examined by considering our integer collection and GUI library examples from earlier chapters. No libraries have been designed using the mixin style proposed by this report, so the comments in this section are speculative.

Multiple inheritance eliminates the need for overridable class components. What would be overridable components in single-inheritance designs become separate mixins with multiple inheritance. Eliminating overridable components reduces the chance of errors by simplifying the design, implementation, and use of specialization interfaces.

In single-inheritance designs, class components are packaged together into classes. In mixin designs, they are separated into their own classes as illustrated by the integer-set related mixins given in Fig. 7.2. Looking at this case in more detail, Fig. 7.6 shows how the integer set classes from previous chapters package two different implementation of each of the two components `elements` and `cache` into three classes. Table 7.1 indicates how a mixin design would separate these four component implementations into their own class. This illustrates how mixins encourage designers to break libraries into a larger number of smaller, more focused classes than one finds in single-inheritance hierarchies. Besides being more more coherent, these more focused mixins can be combined in a flexible manner than class components can be. For example, in the mixin design but not in the single-inheritance design, one can reuse the existing classes to construct an integer set class with a sorted `elements` and a bit-vector `cache`.

In traditional object-oriented languages, the type and class hierarchies are combined. In the mixin style, the two hierarchies are separate. One benefit of separating the type hierarchy from the class hierarchy is that the class hierarchy can be designed strictly in terms of behavioral subtyping and no longer need be influenced by code-sharing relationships. Fig. 7.7 shows both the existing ET++ `VObject` hierarchy and an alternative hierarchy uninfluenced by code sharing concerns. In the existing hierarchy, the abstract class `CompositeVObject` exists for code sharing only. In the alternative hierarchy it is not needed, making the hierarchy simpler. In the existing hierarchy,

```
specialization specification IntSetElements {
 state field elements, cache;
 invariant cache ⊆ elements

 final component {
   substate field elements;

   void addElement(int el);
     // Modifies: this.elements
     // Effects: Adds el to this.elements.

   boolean removeElement(int el);
     // Modifies: this.elements, this.cache
     // Effects: Remove el from this.elements; return true iff el in at start.

   IntEnumeration elements();
     // Effects: Returns an enumeration of the integers in this.elements.
 };

 deferred component {
   substate field cache;

   boolean contains(int el);
     // Modifies: this.cache
     // Effects: Returns true iff el is in this.elements.

   void uncache(int el);
     // Modifies: this.cache
     // Effects: Removes el from this.cache
 };
};

specialization specification IntSetCache {
 state field elements, cache;
 invariant cache ⊆ elements

 final component {
   substate field cache;

   boolean contains(int el);
     // Modifies: this.cache
     // Effects: Returns true iff el is in this.elements.

   void uncache(int el);
     // Modifies: this.cache
     // Effects: Removes el from this.cache
 };

 deferred component {
   substate field elements;
   IntEnumeration elements();
     // Effects: Returns an enumeration of the integers in this.elements.
 };
};
```

Figure 7.5: Mixin specifications.

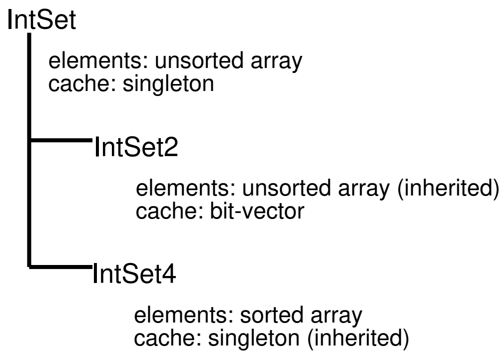


Figure 7.6: Classes implementing integer sets.

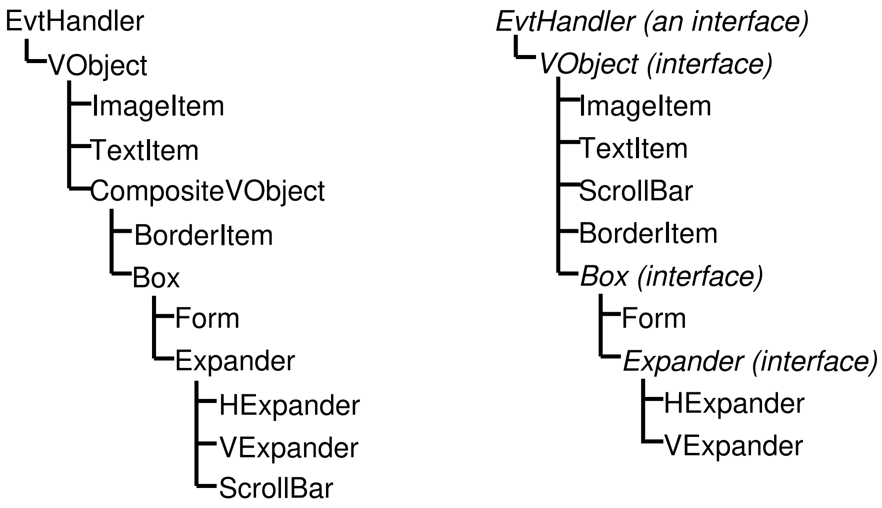
Mixins for the `elements` component	
`IntSetCore`	Stores elements in an unordered array
`IntSetSortedCore`	Stores elements in an ordered array
Mixins for the `cache` component	
`IntSetSmallCache`	Caches one membership hit at a time
`IntSetBVCache`	Caches numbers 0-63 using a bit-vector

Table 7.1: Mixins for building integer set.

`ScrollBar` appears under `Expander`. `ScollBars` are not meant to be used by instantiators as if they were `Expanders`, but they are a subtype of `Expander` because `ScrollBar` is conveniently implemented by reusing `Expander` code. In the alternative hierarchy, `ScrollBar` appears directly under `VObject` where it more logically belongs.

Another benefit of separating the type and class hierarchies is that Java's interfaces will more likely be used. In Java, to get good code reuse, designers put classes high in the type hierarchy. However, once a Java class is introduced into the type hierarchy, all subtypes below that type must be defined using Java classes. This leads to libraries that do not use interfaces very aggressively. For example, if the `VObject` hierarchy was done in Java, then, for code sharing purposes, `EvtHandler` would be a Java class, meaning the entire `VObject` hierarchy would consist of Java classes. Java's own GUI class library [Gosling96] uses classes high in the type hierarchy, and as a result it makes very little use of interfaces.

With mixins, interfaces can be pushed deeper in the type hierarchy. For example, in Fig. 7.7b, `EvtHandler` and other types with *(interface)* next to their names would be defined using interfaces. This allows the use of multiple subtyping deeper in the type hierarchy. For example, a class `OLEVObject` could be a subtype of both `VObject` and `OLEObject` (where `OLEObject` would be an interface for a Microsoft OLE-compliant object [Microsoft94]). As this example indicates, pushing interfaces deeper into type hierarchies may also have some advantages when it comes to object-oriented, client-server programming standards. In Microsoft's OLE and OMG's CORBA, programs interact with distributed objects using "interfaces." Like Java interfaces, these distributed-object interfaces describe the method suite of an object apart from the object's implementation. In a distributed version of Java, OLE or CORBA interfaces could be a special kind of Java interface,



(a) Old type hierarchy (b) Alternative hierarchy

Figure 7.7: Restructuring the ET++ type hierarchy.

allowing seamless interaction of Java code with distributed CORBA objects.

7.4 Summary

A well-designed mechanism for multiple code inheritance allows designers to use separate "mixin" classes in place of overridable class components. In languages like Java and Theta [ThetaWeb95] that have modules for defining object types apart from classes, such a multiple inheritance mechanism is easy to add.

In the mixin style of design, classes become smaller and more focused. Because they have only two class components—one final and one deferred—they are easier to design, code, verify, and use. Because mixins are focused in their functionality, they can be reused in a highly flexible manner. In the mixin style of design, types and classes are separated, leading to type hierarchies that are not influenced by code sharing considerations. In Java, the mixin style of design would encourage the use of Java interfaces deeper in type hierarchies, allowing more aggressive use of multiple supertypes and perhaps allowing better integration with schemes for distributed objects.

Chapter 8

Conclusion

8.1 Summary

Classes need two interfaces: instance interfaces that say how to manipulate instances, and specialization interfaces that say how to build subclasses. The unit of modularity for instance interfaces is the entire class. To support extensibility, the unit of modularity for specialization interfaces is smaller: the class component.

Class components are a programming convention that support the representation categories of final, deferred, and overridable for the state of classes. This convention consists of partitioning the class into components consisting of both state and methods. The implementation of a component must not only be functionally correct but must also be independent of the implementations of other components. Achieving this independence requires adding a division of labor to the documentation of specialization interfaces to indicate component boundaries. It also requires reasoning about each component as an independent unit that depends only on the specifications of other components and not on their implementations.

The abstract representation is the view of a class's state given in its specialization interface. This view is more detail than the view given in the instance interface, revealing details that describe the interfaces between class components. The specialization interface also includes the abstract representation invariant, an invariant on the abstract representation that serves in lieu of inter-component representation invariants. Selecting a good abstract representation and associated invariant is central to designing a good specialization interface.

Mixins take the idea of class components a step further by breaking class components into separate modules. Mixin-style programming separates the type and class hierarchies and uses Snyder's encapsulated, multiple inheritance for the class hierarchy. Mixin-style programming leads to smaller, more focused classes that can be reused in more flexible combinations.

8.2 Contributions

This report has focused on improving specialization interfaces. It makes a number of contributions that can have an immediate impact on the design, documentation and implementation of class libraries. It also makes contributions that, in the longer term, may change the way class libraries are structured.

A central contribution of our work is to provide guidance on designing class libraries that are more extensible and easier to reuse. The ideas of abstract representations, abstract representation invariants, representation categories, and class components give designers a tool box with which to tackle the designs of specialization interfaces. Until now, representation categories have been implicit in only a few of the very best libraries. This report makes them available to all designers, and it helps avoid errors found in even those very best designs. Also, in our view, even these very best designs do not use overridable fields aggressively enough. Space-time conflicts make it difficult to design classes that are applicable in a wide variety of contexts. Overridable fields help resolve these conflicts.

The most immediately useful contribution of this report is a framework for documenting the specialization interfaces of class libraries. Current documentation is frustratingly bad. This leads to long learning times for most class libraries and also to incorrect use of classes. Further, as pointed out in Chapter 1, current documentation does not tell specializers what they need to know to build subclasses. Recognizing this reality, most vendors ship the source code of their class libraries [Atkinson92]. When customers look at source code, they become dependent on implementation details that vendors may want to change, so vendors become hamstrung when improving implementations, and customers have to worry about new versions invalidating their code.

The ideas in this report can improve documentation in a number of ways. Our exhortation to document the abstract-state of classes is an important but neglected saw. Separate specifications for instance and specialization interfaces will lead to better documentation for both. Class components, abstract representations, and abstract representation invariants provide the modularity needed to allow specializers to build subclasses without depending on the implementations of superclasses.

Another immediately applicable contribution is to explain the difference between the validation criterion for classical data abstractions and for classes in the context of subclassing. This can help implementors of classes in two ways: it can help them correctly use subclassing to build new classes, and it can help them reason about their classes so they will not break when subclassed. This last is particularly important. There are many programming errors that effect only subclasses of a class and not instances, *e.g.*, assuming an invariant that might not hold in all subclasses. It is hard to build test suites of subclasses to uncover such errors. Thus, in the context of subclassing, testing is even less effective than it is for classical data abstractions, and being rigorous about correctness during coding is even more important.

This report suggests interesting directions for language design. Our construct for overridable class components involves only a minor extension to languages, like Java, that have some kind of "interface" or "object type" construct. Such a construct has important benefits. It increases the extent to which the design of a class's specialization interface is manifest in its implementation, helping implementors avoid errors. The component construct further helps implementors avoid errors by supporting a number of compile-time checks, checks that catch the kinds of errors that affect subclasses but not instances. Given the difficulty of testing specialization interfaces, we hope the class-component construct soon finds its way into languages or at least into checking tools.

Mixins are another language construct suggested by this report. The impact of mixins will be farther out than that of class components because they entail a more radical departure from current design practice. However, mixins may have an important place in the future of programming. We imagine a future of programming in which programs are (possibly distributed) networks of collaborating objects. Ninety-five percent or more of the objects in these networks will be off-the-shelf, and the last five percent, which will embody the application-specific aspects of programs, will be put together out of object-building components. These networks of objects will be constructed with the close cooperation of end-users, perhaps even by the end-users.

The objects in these networks will be glued together using widely-accepted standards for object "interfaces" like Microsoft's COM. Actually, these "interfaces" are really object types as defined in Sec. 1.1. The objects themselves will be implemented using a wide-variety of systems, such as Visual Basic, C++, and Java. The mixin style of programming seems a promising way of implementing objects in this future where object types and implementations will be completely separated. Although the mixins in Chapter 7 are for building classes, a similar approach can be taken for building individual objects ([Steyaert95]). This is important because, in our view, object-based as well as class-based approaches to object construction will be commonplace. The mixin style has the potential of being simpler than existing class- and object-based approaches. Instead of giving programmers large, complicated objects or classes with lots of knobs for customization, the mixin style gives them small, coherent chunks of functionality that can be combined in an easy manner. With mixins, there is no notion of "replacing" parts of modules, there is only the simpler notion of fitting parts together.

8.3 Related work

The idea of documenting data abstractions in terms of abstract- rather than concrete state has a long history going back at least to [Hoare72]. This existing work forms the foundation of the specification techniques presented in Chapters 2 and 4, but it does not address the issues that arise in the context of subclassing.

There are a number of object-oriented specification languages [Lano93]. These languages are "object-oriented" in the sense of adding object-oriented features like subtyping and specification inheritance to specification languages like Z [Spivey92]. However, they do not address the problems raised in trying to specify specialization interfaces. Related to this work is work on specifications and behavioral subtype relations for type hierarchies [Leavens89, America91, Liskov94]. We borrowed heavily from this work, especially the work of Liskov and Wing [Liskov94]. But again, none of this work addresses specialization interfaces, including Leavens' more recent work on Larch/C++ [Cheon94].

An active area of work is on kernel languages. This work aims at finding the lambda-calculus of object-oriented programming, *i.e.*, a tiny language kernel that embodies the essence of object-oriented programming. Currently, there are four major approaches (a survey of three can be found in [Wadler94], and the fourth is presented in [Castagna95]). The focus of this work has been on type systems that eliminate "message not understood" errors without giving up too much of the flexibility of untyped object systems. This pursuit has lead to the recognition that "[code] inheritance is not subtyping" [Cook90], and that flexible subclassing requires special consideration (see, *e.g.*, [Abadi95] and [Fisher95]).

An important connection between our work and work on kernel languages is the separation of subtyping and subclassing. This separation, central to our own work, was first fully developed in the context of kernel languages. Our work provides additional reason to separate subtyping and subclassing: even when a subclass is a subtype in the *non*-behavioral sense of subtyping, its instance specification may not be a *behavioral* subtype of its superclass's instance specification (Sec. 3.4).

Little work has been done on verification of specialization interfaces. Our own work is based on the traditional approach to verifying data abstractions, which goes back again to [Hoare72].

Leaven's work on verification [Leavens89] considers verification of programs in the presense of behavioral subtyping, but it does not consider verification of specialization interfaces.

Rustan Leino's report [Leino95], which presents modular verification techniques for object-oriented programs, does consider verification in the presence of subclassing. [Leino95] is more formal than our work and includes a soundness proof for a restricted verification problem. This extra formality has proven useful for generating verification conditions in the Extended Static Checker [Detlefs96].

Our work is more general than [Leino95]. The generality of our work is compatible with Leino's work and could inform extensions to that work. [Leino95] considers a language model in which subtyping and subclassing are combined, where we consider a more general language model. [Leino95] supports only final and deferred abstract-state fields, while we consider overridable fields as well. [Leino95] allows only one specification per method, *i.e.*, subclasses inheriting a method also inherit a specification for that method. This restriction makes it harder for subclasses to refine the behavior of methods (although [Leino95] does allow indirect refinement by allowing refinement of the abstraction functions that define abstract-state fields). This restriction makes it hard to hide specialization details from instantiators, and it rules-out having separate assumed and particular specifications for overridable methods. [Leino95] is more general in that it supports arbitrary numbers of interfaces to classes while we support only two (instance and specialization interfaces). However, [Leino95] does not recognize as we do that specializers need different *kinds* of information (*e.g.*, representation categories for abstract-state fields, both assumed and actual specifications for overridable methods).

Many text books cover object-oriented design (*e.g.*, [Rumbaugh91], [Booch94]). However, none consider the design of specialization interfaces. [Kiczales91] and [Kiczales92] present principles and advice for designing specialization interfaces, but they concentrate on layered control abstractions and do not discuss class components, abstract representations or representation categories. Control abstractions go back to Simula, where they are part of the programming lore (see, *e.g.*, [Pooley87]). [Lamping93] presents static checking for control abstractions. [Lamping93] anticipates the idea of class components, but it fails to distinguish them from control abstractions. Lamping's later paper on checking specialization interfaces [Lamping94] deals purely with control abstractions. The Standard Template Library of C++ [Musser96] makes extensive use of control abstractions; interestingly, in the STL, extensibility is provided by parametric polymorphism rather than by subtype polymorphic.

The term "mixin" was first coined in the Flavors community [Weinreb81]. It has been used in a number of contexts; see, *e.g.*, [Keene88], [Bracha90], [Booch94], [Taligent94], [VanHilst96]. In the context of multiple-inheritance languages like C++, a mixin is a partial class that implements a small part of the functionality of a larger class (see [Booch94]). A mixin is distinguished from other partial classes by the intent of its designers. A typical, partial class is like a mostly completed puzzle, with deferred methods representing a few missing pieces. A mixin is like a single puzzle piece, with final methods representing the tabs on a puzzle piece and deferred methods representing the indentations into which tabs of other pieces fit.

8.4 Future work

The ideas of this report suggest a number of software-development tools. At one extreme is commenting conventions from which printed documentation can be extracted; at the other extreme is full, formal verification of classes. In between, this report suggest annotations to support static checks. A valuable property of the ideas in this report is that they suggest tools along this entire spectrum of formality, allowing the level of formality to be dictated by the needs of a project rather than by the tools.

The verification procedure presented by this report is based on the traditional simulation techniques for data abstractions. There are still open issues regarding these traditional techniques, even for classical data abstractions without subtyping or subclassing. One of the thorniest issues is modular reasoning about changes to abstract values of objects in the presence of shared, mutable objects. Typical of the kinds of hard questions raised in this context is the following: If an object of type T is in the representation of objects of type S and an object of type T is modified, which (if any) objects of type S are also modified? Although some progress has been made in this area ([Schaffert81, Leino95]), practical solutions have not yet been found.

An area that needs further work is specifications for control abstractions. As suggested in [Kiczales92], formal specifications of control abstractions may require a more operational flavor than the specifications used in this report, *e.g.*, specifications that explain the behavior of a method in terms of invocations of other methods. It remains to be seen whether true, operational specifications are needed, or whether declarative specifications with an operational flavor will work (see, *e.g.*, the use of "actions" in [Birrell91]).

As mentioned above, a number of books describe full methodologies for object-oriented programming, but none of these methodologies pay as close attention designing specialization interfaces as we have. Thus, an obvious next step is to use the ideas in this report to improve the treatment of specialization interfaces.

Perhaps the most important area of future work will be to get more experience in applying the ideas of this report to real-world projects. The design study in Chapter 6 as well as examples in other chapters suggest that we are headed in the right direction. But this experience-base is inadequate, and future experience will surely lead to improvements. An important question we would like to explore is whether or not our approach is too restrictive: does it rule out good designs we don't want to rule out? Also, we would like to have a better assessment of the practical value of extensible specifications described in Sec. 5.2.

Practical experience is even more important for the more speculative work on multiple inheritance in Chapter 7. For single inheritance, our ideas are inspired by successful class library, but this is not the case for multiple inheritance. Thus, for multiple inheritance, the goal of more experience is not further refinement but rather to test basic utility. One possible experiment would be to use the ideas in Chapter 6 to design a single-inheritance, GUI application framework, and then use the ideas in Chapter 7 to convert it into a mixin design. Two versions of a number of sample applications could be built using the two frameworks, providing a basis for comparison and evaluation.

8.5 Conclusion

We have shown that modularity in the presence of subclassing is indeed possible. We feel the aspect of our work most responsible for its success is our focus on abstract state. In concluding, we would like to draw attention to this aspect of our approach and explain why it is so important.

Objects are state plus behavior. In object-oriented languages (and for data abstraction in general), abstraction for behavior is supported more directly than it is for state. The act of invoking methods encourages a decoupling of clients from the implementations of methods, providing a natural abstraction boundary. Although clients *can* depend on aspects of a method's implementation they should not depend on, such dependence is not a necessary part of method invocation and if anything is discouraged by it. The act of accessing instance variables is fundamentally different: it strongly couples clients to the implementation of state.

We do not believe there is a linguistic solution to this asymmetry. For example, we do not believe there is any way of adding "abstract-state fields" as a language construct. Instead, the asymmetry needs to be addressed via programming methodologies. Mechanically, this means encapsulating access to instance variables so that clients no longer interact directly with the implementation of the state of objects but instead call methods. More importantly, this means that programmers need to learn to design using abstract-state fields, to base documentation on them, and to *think* in terms of them.

When new mechanisms for data abstraction are introduced, research in software engineering must find the right way of thinking about abstract state in the context of those new mechanisms. In essence, this report does just that: it figures out how to think about abstract state in the context of subclassing. As other approaches to data abstraction are introduced, *e.g.*, object-based [Ungar91], role-based [VanHilst96], and subject-oriented [Harrison93] programming, the issue of abstract state will have to be revisited again.

Bibliography

[Abadi95] M. Abadi and L. Cardelli. On subtyping and matching. *ECOOP '95 Proceedings* (Aarhus, Denmark, Aug. 1995). Published as *LNCS 952*, pages 145–67. Springer Verlag, Berlin, Aug., 1995.

[America91] P. America. Designing an object-oriented programming language with behavioural subtyping. *Foundations of Obj.-Orien. Lang.* (Noordwijkerhout, The Netherlands, May/June 1990). Published as *LNCS 489*, pages 60–90. Springer-Verlag, 1991.

[Atkinson92] B. Atkinson. Panel: reuse—truth or fiction. *OOPSLA '92 Conf. Proceedings* (Vancouver, Oct. 1992). Published as *SIGPLAN Notices*, **27**(10):41–2. ACM, Oct. 1992.

[Birrell91] A. D. Birrell, J. V. Guttag, J. J. Horning, and R. Levin. Thread synchronization: a formal specification. In *Systems Programming with Modula-3*, pages 119–29. Prentice Hall, 1991.

[Booch94] G. Booch. *Object-Oriented Analysis and Design, with Applications*, 2nd ed. Addison-Wesley, Reading, MA and London, UK, 1994.

[Borland94] *Borland ObjectWindows Programmer's Guide*, version 2.5. Borland, Inc., Scotts Valley, CA, 1994.

[Bracha90] G. Bracha and W. Cook. Mixin-based inheritance. *ECOOP/OOPSLA '90 Conf. Proceedings* (Ottawa, Canada, Oct. 1990). Published as *SIGPLAN Notices*, **25**(10):303–11. ACM, Oct. 1990.

[Brown91] M. R. Brown and G. Nelson. I/O streams: abstract types, real programs. In *Systems Programming with Modula-3*, pages 130–69. Prentice Hall, 1991.

[Bruce96] K. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. Leavens, and B. Pierce. On binary methods. *Theory and Practice of Obj. Sys.*, 1996. *(To appear.)*.

[Castagna95] G. Castagna and G. T. Leavens. Foundations of object-oriented languages: 2nd workshop report. *ACM SIGPLAN Notices*, **30**(2):5–11. ACM Press, Feb. 1995.

[Cheon94] Y. Cheon and G. T. Leavens. A quick overview of Larch/C++. *JOOP.*, **7**(6):39–49. SIGS, Oct. 1994.

[Cook90] W. R. Cook, W. L. Hill, and P. S. Canning. Inheritance is not subtyping. *Proc. 17th POPL* (San Francisco, CA, Jan. 1990), pages 125–35. ACM, Jan. 1990.

[Dahl92] O.-J. Dahl. *Verifiable Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1992.

[Day95] M. Day, R. Gruber, B. Liskov, and A. Myers. Subtypes vs. where clauses: constraining parametric polymorphism. *OOPSLA '95 Conf. Proceedings* (Austin, TX). Published as *ACM SIGPLAN Notices*, **30**(10):156–68. ACM, Oct. 1995.

[Detlefs96] D. L. Detlefs. An overview of the extended static checking system. *Proc. the Workshop on Formal Methods in Softw. Practice* (San Diego, CA). Published as *SIGSOFT*. ACM, Jan. 1996. *To appear.*

[Edwards96] S. H. Edwards. Representation inheritance: a safe form of "white box" code inheritance. *Proc. the Fourth Intl Conf. on Softw. Reuse* (Washingtond, DC., Apr., 1996), pages 195–204. IEEE Comp. Soc. Press, Apr. 1996. (*To appear.*).

[Evans96] D. Evans. Static detection of dynamic memory errors. *ACM SIGPLAN 1996 PLDI* (Philadelphia, PA., May 1996), page ACM., May 1996.

[Fisher95] K. Fisher and J. C. Mitchell. A delegation-based object calculus with subtyping. *Fundamentals of Computation Theory: Proc. 10th Intl Conf., FCT '95* (Dresden, Germany, Aug. 1995). Published as *LNCS 965*, pages 42–61. Springer Verlag, Berlin, Aug. 1995.

[Goldberg89] A. Goldberg and D. Robinson. *Smalltalk-80: The Language*. Addison-Wesley, Reading, MA and London, UK, 1989.

[Gosling96] J. Gosling and F. Yellin. *The Java Application Programming Interface Volume 2: Window Toolkit and Applets*. Addison-Wesley, Reading, MA and London, UK, May, 1996. (To appear.).

[Guttag93] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[Harrison93] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). *OOPSLA '93 Conf. Proceedings* (Washington, DC, Oct. 1993). Published as *ACM SIGPLAN Notices*, **28**(10):411–27. ACM, Oct. 1993.

[Hoare72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, **1**(4):273–81. Springer-Verlag, 1972.

[Holzle93] U. Hölzle. Integrating independently-developed components in object-oriented languages. *ECOOP '93 Proceedings* (Kaiserslautern, Germany, July 1993). Published as *LNCS 707*, pages 36–56. Springer-Verlag, 1993.

[Javabeta95] Sun Microsystems, Inc. *The Java Language Specification (1.0 Beta)*, 30 Oct. 1995.

[Kapur88] D. Kapur and M. Srivas. Computability and implementability issues in abstract data types. *Sci. of Comp. Prog.*, **10**(1):33–63. North-Holland, Amsterdam, Feb. 1988.

[Keene88] S. E. Keene. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Addison-Wesley, Reading, MA and London, UK, 1988.

[Kiczales91] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, Cambridge, MA, 1991.

[Kiczales92] G. Kiczales and J. Lamping. Issues in the design and specification of class libraries. *OOPSLA '92 Conf. Proceedings* (Vancouver, Oct. 1992). Published as *SIGPLAN Notices*, **27**(10):435–51. ACM, Oct. 1992.

[LaLonde91] W. R. LaLonde and J. R. Pugh. *Inside Smalltalk, Volume II*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[Lamping93] J. Lamping. Typing the specialization interface. *OOPSLA '93 Conf. Proceedings* (Washington, DC. Oct. 1993). Published as *SIGPLAN Notices*, **28**(10):201–14. ACM, Oct. 1993.

[Lamping94] J. Lamping and M. Abadi. Methods as assertions. *ECOOP '94 Proceedings* (Bologna, Italy, July 1994). Published as *LNCS 821*, pages 60–80. Springer-Verlag, 1994.

[Lano93] K. Lano and H. Haughton, editors. *Object-Oriented Specification Case Studies*. Prentice Hall, New York, 1993.

[Leavens89] G. T. Leavens. *Verifying Object-Oriented Programs that use Subtypes*. PhD thesis, published as Technical report MIT–LCS–TR–439. Lab. for Comp. Science, MIT, Feb. 1989.

[Leino95] K. R. M. Leino. *Toward Reliable Modular Programs*. PhD thesis, published as Technical report CS–TR–95–03. California Inst. of Techn., Pasadena, CA, Jan. 1995.

[Lewis95] T. Lewis. *Object-Oriented Application Frameworks*. Manning Publications, Co., Greenwich, CT, 1995.

[Liskov86] B. Liskov and J. Guttag. *Abstraction and specification in program development*. MIT Press/McGraw-Hill Book Co., 1977.

[Liskov93] B. Liskov. A history of Clu. *2nd History of Prog. Lang. Conf. (preprints)* (Cambridge, MA. Apr. 1993). Published as *SIGPLAN Notices*, **28**(3):133–47. ACM, Mar. 1993.

[Liskov94] B. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. on Prog. Lang. and Sys.*, **16**(6):1811–41. ACM, Nov. 1994.

[Microsoft94] *Microsoft Visual C++ Volume Two: Programming with MFC and Win32*, version 2.0. Microsoft Press, Redmond, WA, 1994.

[Musser96] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, Reading, MA and London, UK, 1996.

[Next94] Next Computer, Inc. *Openstep Specification*, Oct., 1994. Available via anonymous FTP at `ftp.next.com` in `pub/OpenStepSpec`.

[Pooley87] R. J. Pooley. *An Introduction to Programming in SIMULA*. Blackwell Scientific Publications, Oxford, 1987.

[Pressman92] R. S. Pressman. *Software Engineering: a Practitioner's Approach*, 3rd edition. McGraw Hill, Inc., 1992.

[Rumbaugh91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[Schaffert81] J. C. Schaffert. *Specification and Verification of Programs using Data Abstraction and Sharing*. PhD thesis. MIT, 1981.

[Snyder87] A. Snyder. Inheritance and the development of encapsulated software components. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 163–88. MIT Press, 1987.

[Spivey92] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, Englewood Cliffs, NJ, 1992.

[Steyaert95] P. Steyaert and W. De Meuter. A marriage of class- and object-based inheritance without unwanted children. *ECOOP '95 Proceedings* (July 1995), pages 127–43. Springer Verlag, Berlin, 1995.

[Stroustrup91] B. Stroustrup. *The C++ Programming Language*, 2nd edition. Addison-Wesley, 1991.

[Taligent94] Taligent. *Taligent's Guide to Designing Programs: Well-Mannered Object-Oriented Design in C++*, The Taligent Reference Library. Addison-Wesley, Reading, MA and London, UK, 1994.

[ThetaWeb95] B. Liskov, D. Curtis, M. Day, S. Ghemawat, R. Gruber, P. Johnson, and A. Myers. Theta reference manual. MIT LCS, PMG memo 88, Feb. 1995. `http://www.pmg.lcs.mit.edu/Theta.html`.

[Ungar91] D. Ungar and R. B. Smith. Self: the power of simplicity. *Lisp and Symbolic Computation*, **4**(3):187–205. Kluwer Academic Publishers, July 1991.

[VanHilst96] M. VanHilst and D. Notkin. Using C++ templates to implement role-based designs. *Intl Symp. on Object Technologies for Advanced Softw. '96* (Kanazawa, Japan, Mar. 1996), Mar. 1996. *(To appear.)*.

[Wadler94] P. Wadler (ed.). Type systems for object-oriented programming: special issue of *Journal of Functional Programming*. *J. of Func. Prog.*, **4**(2):125–283. Cambridge Univ. Press, Apr. 1994.

[Weinand95] A. Weinand and E. Gamma. ET++: a portable, homogeneous class library and application framework. In *Object-Oriented Application Frameworks*, pages 154–94. Manning Publications, Co., Greenwhich, CT., 1995.

[Weinreb81] D. Weinreb and D. Moon. *Lisp Machine Manual*, 4th Ed. Symbolics, Inc./MIT AI. Lab., 1981.