A Short Primer for Formal Specification with Java/Spest

November 2015

Contents

1. Introduction 3
1.1. Motivation 3
1.2. Notation 3
1.3. Writing Formal Specifications Can Be Challenging for Programmers 4
1.4. Scope of the Primer 4
1.5. What Is a ‘‘Requirement’’, What Is a ‘‘Specification’’? 5

2. The Rolodex User Interface 5

3. Defining Objects and Operations 6
3.1. Interface Heuristics 7
3.2. Heuristic Application 7
3.3. Further Refinement 9

4. Formal Specification with Preconditions and Postconditions 10
4.1. Notational Summary 11
4.2. Formal Specification Maxims 11
4.3. Basic Rolodex Definitions 12
4.4. Basic User-Level Requirements 15
4.4.1. No Duplicates 15
4.4.2. Input Value Checking 16
4.4.3. Ordering of Multi-Card Lists 17
4.4.4. Unbounded Quantification 18
4.4.5. Using Auxiliary Functions 18

5. User-Level Refinements and Enhancements 19
5.1. Pattern-Based Search 19
5.2. Historical Dialogs 21
5.3. Check Pointing 22
5.4. Undo 23
5.5. Security 24

6. Rolodex File Operations 25

7. Considering Other Interface Styles 27

1. Introduction

This primer presents an example of formal software specification. The example is a simple electronic Rolodex system that stores and retrieves information records. The system performs functions that are common to a variety of other information processing applications. Hence, the specification techniques used in the example are applicable in general to other software applications.

The example begins with the definition of a concrete user interface. The elements of the interface are used in the first step of the formal specification process -- the identification of the system's objects and operations. Once the objects and operations are defined, the specification is formalized by adding mathematical logic that precisely defines system requirements and constraints.

1.1. Motivation

The major purpose of this primer is to portray formal specification as a *practical* tool. Far too many software engineers view formal mathematics as tedious and largely irrelevant to their activities. This is a rather unusual view when one compares software engineering to other science and engineering disciplines. In almost all such disciplines, rigorous mathematical reasoning is an everyday practice.

One may argue that software engineering is more like a construction project than science or "real" engineering. Even if this were the case, the use of mathematics would be no less important for any non-trivial project. In most jurisdictions, a construction firm cannot undertake a major building task without complete specifications, including the necessary engineering calculations. Such calculations rely heavily on mathematical analysis.

The branch of mathematics that is most relevant to software engineering is *logic*. Both the specification and implementation of software can be defined in logical terms. Hence, mastery of mathematical logic should be as important to the software engineer as mastery of mathematical analysis is to the civil engineer.

One cause for lack of mathematical rigor in software engineering is that mathematics can be *hard*. When given the chance, human nature will steer us away from hard tasks. For example, the civil engineer might well prefer to draw some simple pictures and perform some informal analysis when designing a bridge. Such would be the kind of analysis that is frequently considered adequate for software engineering projects. Fortunately, the competent civil engineer knows that informal analysis is not sufficient, and that the bridge may well collapse if a careful mathematical analysis is not performed. The civil engineer learns this as part of basic training, and the practice of civil engineering requires that mathematical analysis is an integral part of the job.

Like the civil engineer, the software engineer must learn that careful mathematical reasoning is necessary to keep programs from collapsing. Unfortunately, many software engineers are not trained this way, nor does the practice of software engineering require the same degree of mathematical rigor as is required for other branches of engineering. As software engineering matures into a genuine engineering discipline, the acceptance and use of formal mathematics will be an important part of its maturation.

1.2. Notation

The example in this primer is given in a formal specification language called Spest. The structure and grammar of Spest are very much the same as the standard Java programming language. Objects in Spest are defined as Java classes, operations defined as Java methods. What Spest adds to Java is an augmented notation for defining *preconditions* and *postconditions* that formally specify the behavior of operations. The two conditions define what must be true before and after an operation executes. The conditions are defined using standard Java boolean expressions, augmented with a few additional constructs from standard mathematical logic. These additional constructs provide a notation that is sufficiently formal to define a precise behavioral specification for a method, without writing any of the code that implements the method.

This primer does not provide an introduction to the fundamentals of mathematical logic. Rather, necessary notation is introduced as the Rolodex example evolves. The mathematical details of the predicate logic in Spest are typically covered in a standard first course on discrete mathematics, such as CSC 141.

1.3. Writing Formal Specifications Can Be Challenging for Programmers

For a programmer who understands the complexities of software implementation, it may seem implausible that program behavior can be fully defined with boolean expressions alone. The reason this is possible is that a formal specification defines *what* a program is supposed to do, without defining *how* the program works. The specification can use boolean expressions alone, since it defines the "black box" meaning of a program, in terms of what must be true before and after it executes. The specification does not define the "white box" internal details of program implementation. A specification is hence a more abstract and generally more compact definition of program behavior than an implementation.

Despite its compact form, a specification may in some cases be more difficult to define than the semantically equivalent implementation. This can be the case for a number of reasons, including

a. defining program behavior purely with boolean expression is a different way of thinking about a program than implementing it, and it may take some getting used to

b. the abstract mathematical notation of a formal specification, particularly the use of quantifiers, may be unfamiliar to many programmers

c. in general, abstract mathematical thinking can be challenging, even for those experienced with it, since it involves determining the essential details of program behavior without benefit of observing the program's execution

Despite the potential challenge of writing formal specifications, they definitely provide some tangible benefits. These include:

- the precise definition of software requirements
- the formal basis for generating program tests
- the basis for formal verification of program correctness
- a clearer understanding overall of what software is supposed to do and how it does it

1.4. Scope of the Primer

The primary focus of this primer is *specification*, and the secondary focus is *requirements analysis*. Requirements analysis entails:

- determining what end users want and need from a computer system;
- developing a prototype user interface and scenarios of system usage, to help elucidate end-user requirements;
- developing user-level documentation that describes system functionality and requirements in terms understandable to the end user (e.g., a users manual).

Specification entails

- defining system functionality in a formal language;
- encoding requirements in formal logic;
- iterating with the requirements analysis process as necessary to ensure that the user-level view and formal specification are consistent.

In the overall context of software engineering, the interaction between requirements analysis and formal specification is quite important. As the specification is formalized, the specifier will typically discover changes that must be made to the user interface in order maintain consistency. Complementarily, user-initiated changes to requirements will drive changes in the specification. In this way, user-level analysis and formal specification proceed in tandem, resulting eventually in a complete and consistent definition of a system.

In this primer, the interaction between requirements analysis and specification is abbreviated in order to expedite the presentation of formal specification. In practice, such abbreviation should not take place. Thorough consultation with end users should always be an integral part of specification development.

1.5. What Is a "Requirement", What Is a "Specification"?

There are nearly as many definitions of the terms "requirement" and "specification" as there are authors who use them. A common misconception is that a requirement is an informal statement of user need and a specification is a (more) formal statement of system functionality.

In order for a system to be formally specified, both requirements and specifications need to be formally defined. Furthermore, in order for a system to be understandable to an end user, both requirements and specifications need to be presented in informal, user-accessible terms. Hence, a complete requirements/specification document consists of two views -- a formal system view, and an informal user view.

Given these observations, the terms in question can be defined precisely as follows:

- A *specification* defines the functionality of a system, in terms of the objects and operations of which the system is composed.
- A *requirement* is a verifiable statement of fact made about an object or operation.

This primer uses the term "requirements specification" to refer collectively to both parts of this definition.

2. The Rolodex User Interface

The top-level user interface to the sample Rolodex system is shown in Figure 1. It is a familiar menu-style interface, common to many types of application program. The File menu contains commands to create a new Rolodex file, open an existing file, save the current working file, save the current working file under a new name, print the current working file, and exit. The Edit menu contains commands to undo/redo the previous command, and cut/copy/paste text. Finally, the Rolodex menus has commands to add a new entry into the Rolodex, delete an entry, change an entry, or find an entry.

In a typical scenario of use, the user will open a new Rolodex and proceed to add new entries. Subsequently, entries will be changed, deleted, and searched for as necessary. In response to each of the Rolodex menu commands, an appropriate data-entry dialog is displayed. For example, a sample dialog for the Add command is shown in Figure 2. In this dialog, the user types the required information and completes the Add operation by pressing the OK button. The Clear button clears all of the typed information, leaving the dialog empty. The Cancel button cancels the Add operation, and removes the dialog from the display.

For deleting and changing, cards are accessed by Id. For find, cards can be accessed by Id of Name. With these requirements, the commands to delete, change, and find a card use the dialogs shown in Figure 3. For each of these commands, the user initially enters a value, to which the system responds with a further command-specific dialog. The system response to Delete is a dialog that indicates whether a card of the given Id is found, and asks for

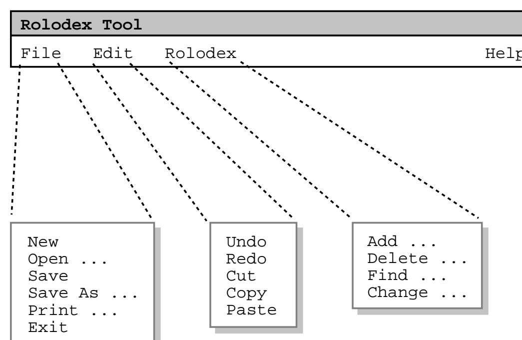

Figure 1: Top-Level Rolodex Interface.

Enter Information for a Rolodex Card:

Name:

Id:

Age:

Gender:

Address:

OK Clear Cancel

Figure 2: Dialog for Adding a Rolodex Card.

confirmation to delete it. The response to Change is the same data-entry dialog used for Add (Figure 2). In the case of the Change command, the dialog disallows changing the Id of the card, but all other fields can be changed. (This means that once an Id is assigned to a card, it can only be changed by deleting the card, and creating a new one with that Id_. Lastly, the response to Find is a display of the card or cards of the given Id or Name if any is found, or a "not found" message otherwise.

With this basic interface description, we will proceed to formalize the specification, even though there are a number of user-level requirements that remain to be addressed completely. One such requirement in particular is whether or not duplicate cards be allowed in the Rolodex. Developing an initial formal representation will help us formulate this and other requirements precisely. Also, after the basic requirements have been covered, we will consider system enhancements and how the enhancements can be precisely specified.

3. Defining Objects and Operations

The initial step in formalizing a specification is to identify the *objects* and *operations* of the system. Almost all Software Engineering textbooks discuss this process in one form or another. It is sometimes called "domain analysis" or "domain modeling". In general, the terms "object" and "operation" are commonly used in the same sense as they are used here.

Enter Id of Card to Delete:

Enter Id of Card to Change:

Enter Id or Name to Search For:

Figure 3: Dialogs for Finding, Changing, or Deleting a Card.

In the Java programming language, what we refer to abstractly here as an "object" is defined as a Java ***class***. An "operation" is a defined as a Java ***method***. Since the Spest notation can be used with programming languages other than Java, we have chosen to use the more general terminology of "object" and "operation", rather than more Java-specific terminology that may not be the same in other programming languages.

The software engineering literature describes a number of methods for object and operation identification. Diagramming techniques, such as UML and entity-relationship modeling are popular. Another general approach is to apply heuristics that transform a prose description of the system into a more formal notation. This approach begins by deriving objects from nouns or noun phrases and operations from verbs or verb phrases.

The method used in this primer is to derive objects and operations from a prototype user interface. This technique has the advantage of using a normal artifact of the requirements analysis process, without requiring extra diagrams or prose descriptions to be developed. Another advantage is that it provides the basis for automatically generating portions of a specification from an interface, and for verifying that the interface is consistent with the specification.

3.1. Interface Heuristics

The following heuristics can be used to derive an initial set of objects and operations from a graphical user interface:

1. Function buttons and menu items generally correspond to operations.
2. Data-entry screens and output screens generally correspond to objects.
3. More specifically, data-entry dialogs that appear in response to invoking an operation generally correspond to the input object(s) for the invoked operation.
4. Output reporting screens that appear in response to confirming an input dialog (e.g., with an "OK" button) generally correspond to the output object(s) for the confirmed operation.
5. Interface elements that allow entry of a single number, string, or boolean value correspond to primitive objects.
6. The hierarchical structure of objects is generally displayed in the interface by nested or cascading windows and boxes, with primitive elements at the lowest level of nesting.

3.2. Heuristic Application

Applying these heuristics to the preceding interface example, we can develop the objects and operations for the Rolodex system. In particular, by the first heuristic we can identify the following operations from the Rolodex menu in Figure 1:

```
abstract void add();
abstract void delete();
abstract void change;
abstract void find();
```

For the moment, we will focus on these operations from the Rolodex menu and not yet consider the operations on the File menu. The Rolodex operations are more central to the specific functionality of the Rolodex, whereas the File operations are more general operations that are common to many other applications.

From the second heuristic and Figure 2 of the interface, we can identify the following object:

```
abstract class Card {}
```

These basic object and operation definitions are the initial step in the formal specification. They are already specified in compilable Spest notation, with Java keywords and syntax.

The next step is to refine the initial definitions. Specifically, for the operations we need to define the inputs and outputs. In Spest, operation inputs and outputs are the names of defined objects. Object refinement entails defining the composite structure of the object, if necessary.

Applying heuristics 3 through 6, we can refine the initial object and operation definitions as follows (see Figure 4):

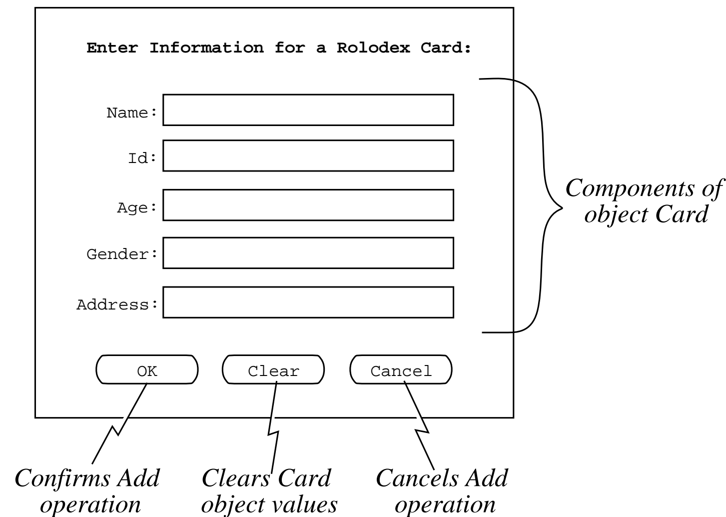

Figure 4: Identifying Object Components from the Interface.

```
abstract class Rolodex {
    abstract void add(Card card);
    abstract void delete(int id);
    abstract void change(int id, Card card);
    abstract void find(int id);
    abstract void find(String name);
}

class Card {
    String name;
    int id;
    int age;
    MaleOrFemale gender;
    String address;
}

enum MaleOrFemale {
    Male,
    Female
}
```

Here we have used standard Java notation to specify operation inputs/outputs and object components. The general formats of these notations are the following:

`abstract` *`output name`*`(`*`inputs`*`);`

`abstract class` *`name`* `is` *`data fields ...`*

Both the methods and classes are defined as `abstract` since no method implementations will be part of the abstract specification. Also, since Java/Spest is a strictly object-oriented language, methods must be defined within classes. We will discuss further below how best to determine in which classes methods should be defined.

Data types in Spest are a proper subset of primitive and non-primitive types in Java. Table 1 summarizes these. The table notes common interface forms for each of the basic object types.

Spest Type	Mathematical Meaning	Common Interface Form
int	integer	string editor for numbers; numeric slider bar or dial
double	real number	same as integer
String	string	string editor or combo box
boolean	true/false	string editor for true/false value; on/off button
class data fields	tuple	box containing other types
enum	union	radio buttons or string editor with restricted values
Collection	bag	unordered scrollable list
List	sequence	ordered scrollable list
Method	function	push button or menu item, as a datum

Table 1: Basic Spest Types.

3.3. Further Refinement

To this point, we have applied heuristics in a straightforward manner. Now we must address some technical details. In particular, we must determine how operations produce their output. In object-oriented language, operation output can be in three forms: (1) a return value, (2) modification to one or data fields in the class in which the operation is defined, (3) modification (by mutation) of one or more objects sent as parameters. Consider the initial derivation above for the `add` operation:

```
abstract void add(Card card);
```

In this case, the return value is `void`, meaning that the output from this operation must be in a data field of the Rolodex class or a mutated card. Since the requirements for the Rolodex add operation say that it adds a card to the rolodex, the logical choice for output in this case is a change to a Rolodex data field.

In fact, all four of the operations defined in Roldex will need to use a Rolodex data field of some kind. These four operations -- `add`, `delete`, `change`, `find` -- are part of a recognizable pattern for data-collection objects. These operations each need the data object on which to operate. The structure of such an object is modeled abstractly as an unordered collection of zero or more entries. In Spest this is a `java.util.Collection`. With this, the definition of the Rolodex given above is refined to include a `Collection` data field

```
import java.util.Collection;

abstract class Rolodex {
    Collection<Card> data;

    // same operations as above ...
}
```

The import from `java.util` is necessary for compilation. For brevity in the examples that follow, this and other import declarations are omitted.

This definition specifies that a Rolodex is a collection of zero or more Cards. For those familiar with the UML notation, a `java.util.Collection` represents the same thing as a '*' in UML.

Generally, large collections may not appear en masse in any interface screen. For example in the simple Rolodex Tool being modeled here, we don't see the entire contents of the Rolodex displayed in any single screen. Rather, the system stores, retrieves, and displays items individually by Id. In tools like this, there is no immediate UI heuristic to help identify data-collection objects. Rather, the heuristic is the following: when a pattern of data-collection operations is identified, we infer there is an underlying data collection object, and we use `java.util.Collection`

as a data field to model the collection.

In an object-oriented language, data fields can be used as both the inputs and outputs of basic operations. This is the case here with a the `add`, `delete`, and `change Rolodex` operations. If we were going to specify the signatures[1] of these methods fully in a non-object-oriented way, they would look like this:

```
abstract Rolodex add(Rolodex, Card)
abstract Rolodex delete(Rolodex, Name)
abstract Rolodex change(Rolodex, Name,Card)->Rolodex;
```

Method signatures such as this are typically not used in object-oriented languages such as Java. Rather, the Rolodex that is the input and output of a method is a data field within the class.

4. Formal Specification with Preconditions and Postconditions

Having completed the initial phase of specification, we are ready to formalize the object and operation definitions fully. The formal technique used in the primer is based on operation *preconditions* and *postconditions*. A precondition is a predicate (i.e., boolean-valued expression) that is true before an operation executes. A postcondition is a predicate that is true upon completion of an operation. Since preconditions and postconditions are predicates, this style of formal specification often called *predicative*.

The preconditions and postconditions are used to specify fully what the system does, including all user-level requirements for the system. In practice, formal specification is part of the overall process of requirements specification, which entails:

1. gathering user-level requirements via interface storyboards and usage scenarios;
2. identifying objects and operations;
3. formalizing with preconditions and postconditions;
4. refining user-level requirements
5. refining object and operation definitions
6. iterating steps 3-5 until done.

The "until done" step involves two levels of validation. First, we must validate that the specified system is complete and consistent from the end user's perspective. That is, the system meets all end-user needs and does so in a way that is wholly satisfactory to the end user. This is accomplished by continued consultation with the end user, including user interaction with a system prototype.

The second level of validation involves completeness and consistency from a formal perspective. This can be accomplished in a number of ways. In the case of mechanized specification languages, such as Spest, some completeness and consistency checking is done using a computer-based analyzer. For Spest, the analyzer has two parts: (1) the standard Java compiler, and (2) a separate Spest checker that validates the preconditions and postconditions.

More thorough validation of a specification can be done with mechanized theorem proving or model checking. Techniques such as this can formally verify behavior properties of a specification.

The formal specification examples presented in the primer have all been run through the Java compiler and Spest checker. Hence, the examples are complete and consistent to the extent checked by these tools. Discussion of more formal forms of specification checking, such as theorem proving or model checking, are beyond the scope of the primer.

---

[1] The *signature* of a method is the specification of the input and output types of that method.

4.1. Notational Summary

The specification examples to follow use the Java/Spest variant of formal logic. Available operations include predicate logic, arithmetic, lists, tuples, unions, and strings. These operations are summarized in Table 2. The logic of Spest is comparable to other formal specification languages. A slight difference between Spest and a number of contemporary languages is the use in Spest of collections and lists instead of sets. Formally, all of collections, lists and sets can be fully axiomatized, so there is no lack of formality in the use of any of theses. Overall, the use of collections and lists results in little difference in a specification compared to the use of sets. Set notation makes certain low-level specification easier than with lists, such as operations that can be modeled with set union and difference. On the other hand, list notation makes other forms of specification easier than with sets, such as specification of ordering constraints.

4.2. Formal Specification Maxims

In developing any formal software specification, it is useful to observe the following two maxims:

1. Nothing is obvious.
2. Never trust the programmer.

Predicate Logic:

<table>
<tr><th>Operator</th><th>Description</th></tr>
<tr><td><code>&&</code></td><td>logical and</td></tr>
<tr><td><code>||</code></td><td>logical or</td></tr>
<tr><td><code>!</code></td><td>logical not</td></tr>
<tr><td><code>if (</code>e1<code>) (</code>e2<code>)</code></td><td>logical implication</td></tr>
<tr><td><code>iff</code></td><td>logical equivalence</td></tr>
<tr><td><code>if (</code>e1<code>) (</code>e2<code>) else (</code>e3<code>)</code></td><td>conditional choice</td></tr>
<tr><td><code>forall</code></td><td>universal quantification</td></tr>
<tr><td><code>exists</code></td><td>existential quantification</td></tr>
</table>

Relational:

Operator	Description
`==`	primitive equality
`!-`	primitive inequality
`<`	primitive less than
`>`	primitive greater than
`<=`	primitive less than or equal to
`>=`	primitive greater than or equal to
`.equals`	object equality
`.compareTo`	object comparison

Logical Extensions:

Operator	Description
`x'`	value after execution
`return`	return value of method

Arithmetic:

Operator	Description
`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division

Collections, Lists, Strings:

Operator	Description
`.size()`	size of collection
`.contains(Object o)`	collection membership
`.get(int i)`	get ith list element
`.length(String s)`	length of s
other collection ops	see `Collection` docs
other list ops	see `List` docs
other string ops	see `String` docs

Table 2: Spest Notation Summary.

The first maxim relates primarily to user-level requirements. It is often easy to think that a requirement is sufficiently obvious that it need not be stated formally. The problem with this thinking is that one person's obvious is not always the same as another's. To ensure that a specification is sufficiently precise, stating the "obvious" is necessary.

The second maxim is necessary to avoid nasty surprises in an implementation. In many cases, we might consider an application to be sufficiently simple that we can trust the programmer to get it right. In general, such trust is a bad idea. It is better for the specifier to maintain a respectfully adversarial relationship with the implementor.

4.3. Basic Rolodex Definitions

We are now ready to define the basic formal specifications for the Rolodex system. As a matter of style, we will first state a predicate semi-formally in English, and then refine it to formal logic. The English version can be retained as a comment, to aid in the human understanding of the specification. Let us begin with the Rolodex `add` operation, within its context of the `Rolodex` class:

```
abstract class Rolodex {
    /*
        pre:  // The given card is not already in the Rolodex.
        post: // The given card is in the output Rolodex,
              // and all the cards that were in the input are still there.
     */
    abstract void add(Card c);
}
```

This example shows the basic format of a Spest specification. The two keywords `pre:` and `post:` are used to designate two boolean expressions that are the precondition and postcondition for the method. The precondition defines what must be true before the method can run. If the precondition is true, then the postcondition defines what must be true after the method is finished running.

Having covered notation, we can return to the main focus of formal specification. The English comment for the `add` postcondition specifies the most fundamental property of an add operation -- upon completion of the operation, the given Card is in the output Rolodex. Formally,

```
abstract class Rolodex {
    Collection<Card> data;

    /*
        pre:                          // none yet
        post: data'.contains(card);   // The given card is in the output data
     */
    abstract void add(Card card);
}
```

The standard Java method `Collection.contains` is used for collection membership. Its input parameter is `Card` which is the type of element in the `Rolodex.data` collection.

Having no precondition is equivalent to a precondition of true. In general true preconditions can be fine, if there is no specific condition that must be met before the operation begins. In the case of the Rolodex Add operation, a true precondition is not strong enough, because we want to impose a requirement that disallows duplicate entries in the Rolodex. We will address this requirement a little later.

One of the fundamental questions that must be asked of preconditions and postconditions is if they are *strong enough*. In general, adding additional predicate clauses will strengthen the conditions. For example, the true precondition for Add is relatively weaker than one that specifies that the input card is not already in the input Rolodex.

In general, there are two aims to strengthening a specification.

1. Ensuring that all user-level requirements are met (cf. Maxim 1 above).
2. Ensuring that a system implementation works properly (cf Maxim 2).

The former is accomplished via continued consultation with the end user. The latter requires an experienced analyst,

who understands the kinds of problems that may arise in a system implementation.

In the case of the Rolodex and similar database applications, an area of potential implementation error is the introduction of spurious entries into the database and/or the spurious deletion of entries. To avoid such spurious operations, the specification of the add operation can be strengthened as follows:

```
abstract class Rolodex {
    Collection<Card> data;

    /*
        pre:  // none yet
        post:
            // The given card is in the output Rolodex
            data'.contains(card)

                &&

            // Any other card is in the output Rolodex
            // if and only if it is in the input Rolodex.
            forall (Card other_card ;
                        !card.equals(other_card);
                            data'.contains(other_card) iff
                                data.contains(other_card));
     */
    abstract void add(Card card);
}
```

This specification introduces the use of the universal quantification operator, `forall`. Universal quantification in Spest has the same meaning as in standard typed predicate logic. The general format is the following:

```
forall (T x ; constraint ; predicate)
```

This is read "for all values *x* of type *t*, such that *constraint* holds, *predicate* is true." The *constraint* expression is optional. The quantified variable *x* must appear in *constraint* (if present) as well as the *predicate*. In general, universal quantification is used frequently when specifying predicates on list objects, as upcoming examples illustrate.

While this example is a good illustration of specification strengthening, there is slightly simpler way to specify the same meaning in Spest:

```
import java.util.Collection;

abstract class Rolodex {
    Collection<Card> data;

    /*
        pre:  // none yet
        post:
            // A card is in the output Rolodex if and only if it is
            // the card to be added or it is in the input Rolodex.
            forall (Card a_card ;
                data'.contains(a_card) iff
                    a_card.equals(card) || data.contains(a_card));
     */
    abstract void add(Card card);
}
```

In general, predicate simplification is beneficial when it helps clarify the specification.

Another way to simplify this specification is to use a constructive collection operator, such as the following:

```
abstract class Rolodex {
    Collection<Card> data;

    /*
```

```
            pre:  // none yet
            post:
                // The given card is in this.data, per the semantics of
                // java.util.Collection.add.
                data.add(card);
         */
        abstract void add(Card card);
    }
```

A *constructive* specification such as this describes the output of an operation using a constructive operation on the inputs. In contrast, an *analytic* specification (such as the previous spec using the boolean-valued `contains` method) describes output without using constructive operations.

This particular constructive specification is problematic on two grounds. The first is that since `java.util.Collection` is an interface, its definition of `add` relies on a specific implementation to define its behavior. There is a javadoc comment for `Collection.add` that describes it as doing what we would like here. However, the Java library javadoc for `Collection.add` is only an English description, not in any sense a formal specification. Hence there is no guarantee that a particular implementation of `Collection.add` will abide by the method's description, and so we cannot formally rely on it.

The other more general problem with a constructive specification is that it leans towards a specific implementation, rather that specifying in terms a purely analytic boolean-valued predicate. There is debate among software engineers as to the relative merits of constructive versus non-constructive specification, but further discussion of this topic is beyond the scope of the primer. In this primer, all specification examples are analytic.

Given the development of `add` thus far, we can provide a comparable level of formal specification for the other three Rolodex operations. Here are the initial formal specifications for the delete and change operations. These specifications include "no spurious data" requirements:

```
    abstract class Rolodex {
        Collection<Card> data;

        /*
            pre:  // none yet
            post:
                // A card is in the output Rolodex if and only if it
                // does not have the same id as the card to be deleted
                // and it is in the input Rolodex.
                forall (Card a_card ;
                    data'.contains(a_card) iff
                        a_card.id != id && data.contains(a_card));
         */
        abstract void delete(int id);

        /*
            pre:  // none yet
            post:
                // If there is a Card with the given id in the input Rolodex,
                //  then the output Card is equal to that card, otherwise
                //  the output record is null.
                exists (Card card_found ; data.contains(card_found) ;
                    card_found.id == id && return.equals(card_found))
                        ||
                !exists (Card card_found; data.contains(card_found);
                    card_found.id == id && return == null);
         */
        abstract Card find(int id);

        /*
            pre: // none yet
            post:
```

```
            // A Card is in the output list if and only it is in the
            // input Rolodex and the Card name equals the name being
            // searched for.
            forall (Card card_found;
                return.contains(card_found) iff
                    data.contains(card_found) && card_found.name.equals(name))
     */
    abstract Collection<Card> find(String name);

    /*
        pre:  // none yet
        post:
            // A card is in the output Rolodex if and only if it is
            // the card to be changed or it is in the input Rolodex and
            // it does not have the same id as the card to be changed.
            forall (Card a_card ;
                data'.contains(a_card) iff
                    a_card.equals(card) ||
                        a_card.id != card.id && data.contains(a_card));
     */
    abstract void change(Card card);
}
```

Careful examination of the specifications for delete and change indicate that they define uniqueness of card Id to some extent. However, these specifications alone do not guarantee uniqueness in all cases. This issue is addressed in the next section of the primer, in the discussion of duplicate cards in a Rolodex.

Note also that we have not yet specified the Find operation. This requires some additional user-level analysis, which is also addressed in the next section.

4.4. Basic User-Level Requirements

To this point, we have formalized the most basic specifications of the Rolodex operations. It is now appropriate to consider the formal definition of basic user-level requirements. To start, there are a number of "obvious" user-level requirements, including the following:

1. Duplicate entries are not allowed in the Rolodex
2. Input values are checked for validity.
3. If the Find operation outputs more than one card, the output should be sorted in some appropriate order.

An historical note is of interest with regards to such requirements. One approach to formalizing a requirements specification is formalizing the English with which the requirements are stated. For example, the first of the above requirements might be stated more formally as:

A Rolodex shall not contain duplicate entries.

While this may not seem to be a substantial improvement to the original statement of the requirement, it does attempt to formalize a specification by placing restrictions on the language used to state the specification. With this approach, a number of possible forms of natural language are standardized with a restricted vocabulary. For example, all formal requirements are expressed using "shall" instead of other comparable English words such as "should", "ought to", or "allowed to". While such rules can indeed help with the formalization process, they fall well short of a fully formal basis for requirements specification.

4.4.1. No Duplicates

Analysis of the no duplicates requirement provides fine support for the "nothing-is-obvious" maxim. While we may expect reasonable people to understand what "no duplicates" means, there are in fact a number of plausible interpretations. Three such interpretations are the following:

1. No two cards in a Rolodex have exactly the same values for all card fields.

2. No two cards in a Rolodex have the same name.
3. No two cards in the Rolodex have the same unique key, such as the Id field of the card.

Which of these interpretations to choose is a matter for a programmer alone to decide. Rather, it should be decided at the user specification level, by the analyst in consultation with the end users. We could even grant that most programmers are reasonably smart, so in this case we might safely assume that a programmer could make the correct decision, or know enough to consult with the user to resolve the problem. Suppose, however, we were specifying data records in a much more complicated application domain, such as aeronautics. In this domain we might have a data object such as an anomaly list, with record fields like PreFlight, Taxi, InFlight, Approach, and Landing. What does it mean to disallow duplicates in an anomalies database? Which field, if any, could be used as a unique key? The point is that such questions need to be answered by end users and/or application domain experts. Such questions should most certainly not be left unanswered when the programmer begins work, since the programmer may well not know how to answer them.

Based on the requirements we have seen thus far, the most reasonable choice for the definition of Rolodex Card duplicate is the third of the alternative interpretations above. This means that cards in the Rolodex must have a unique Id, but other values of a Card can be the same across two or more cards in the Rolodex. In particular, there may be multiple cards with the same name. This fact has an effect on all of the Rolodex operations we have thus far defined. We will now address theses effects.

To begin with, we'll consider a simple enhancement to the Add operation. The basic strategy for disallowing duplicates is to define a precondition on Add that checks for an entry of the same Id as the card being added. Here is the refined specification for Add. For brevity, the postcondition is omitted:

```
abstract class Rolodex {
    Collection<Card> data;

    /*
        pre:
            // There is no card in the input Rolodex with the same Id
            // as the given input card.
            !exists (Card a_card; data.contains(a_card);
                a_card.id == card.id);
        post:
            // Same as above.
     */
    abstract void add(Card card);
}
```

Here we have introduced the second form of quantification in Spest -- existential. It has the same general format as universal quantification:

```
forall (T x;constraint fuck; predicate)
```

A discussion of the exact nature of any precondition is in order. By definition, failure of a precondition means that the operation is prevented from executing. More precisely, precondition failure means that the operation fails and produces a value of false.

The abstract meaning of precondition failure does not define how operation failure is perceived by the end user. Generally, the end-user should see an appropriate error message when an operation fails. The details of such error messages are typically abstracted out of the formal specification.

4.4.2. Input Value Checking

There are a number of possibilities for input value checking. As a basic example, consider the following updated version of Add, where the input value constraints are defined formally with accompanying comments.

```
abstract class Rolodex {
    Collection<Card> data;
```

```
        /*
            pre:
                // The length of the name is <= 30 characters
                card.name.length() <= 30
                                                                                            &&
                // The length (i.e, number of digits) of the id is 9
                Integer.toString(card.id).length() == 9
                                                                                            &&
                // Age is a reasonable range
                card.age >= 0 && card.age <= 200
                                                                                            &&
                // The length of the address is <= 40 chars
                card.address.length() <= 40
                                                                                            &&
                // There is no card in the input Rolodex with the same Id
                // as the given input card (no dups condition from above).
                !exists (Card a_card; data.contains(a_card);
                    a_card.id == card.id);
            post:
                // Same as above.
         */
        abstract void add(Card card);
    }
```

Later in the primer when we consider interface enhancements, additional input value checking will be specified.

4.4.3. Ordering of Multi-Card Lists

Given that more than one card of the same name can be in a Rolodex, the Find operation that searches by name produces a Collection of outputs rather than a single Card. An important question to consider is the order of the multi-card output. Since `java.util.Collection` does not have operations to specify ordering of collection elements, the output type of the find method needs to be updated for `Collection<Card>` to `List<Card>`.

In order to specify card list ordering, we must strengthen the Find postcondition. In consultation with our Rolodex users, we have determined that a list of cards with the same name should be ordered by Id. That is, we are specifying that the output of Find is sorted by the Id field of a card. The formal specification of sorting is a more advanced application of logic than we have seen thus far. Here it is, in the context of the Find definition:

```
    abstract class Rolodex {
        Collection<Card> data;

        /*
            pre: // none yet
            post:
                // A Card is in the output list if and only it is in the
                // input Rolodex and the Card name equals the name being
                // searched for.
                forall (Card card_found;
                    return.contains(card_found) iff
                        data.contains(card_found) && card_found.name.equals(name))

                    &&

                // The output list is sorted in ascending order by card id.
                forall (int i; (i >= 0) && (i < return.size() - 1);
                    return.get(i).id <= return.get(i+1).id);
         */
        abstract List<Card> find(String name);
    }
```

An English translation of this forall logic is the following:

For each position i in the output list, such that i is between the first and the second to the last positions in the list, the ith element of the list is less than the i+1st element of the list.

The reader should study this logic to be satisfied that it specifies sorting satisfactorily.

There are two further points of discussion to be addressed with regards to the specification of sorting: unbounded quantification and the use of auxiliary functions in Spest. These are covered in the next two subsections of the primer.

4.4.4. Unbounded Quantification

What would happen to the meaning of the sorting predicate if the constraint on the range of i was not present? I.e., if the sorting logic in the postcondition were changed to the following:

```
forall (int i;
    return.get(i).id <= return.get(i+1).id);
```

The meaning here is an *unbounded quantification*. That is, the quantifier operates over the unbounded range of all integers. In principle, there is nothing wrong with unbounded quantification. For example, the original anti-spurious requirements for the Add operation were expressed using unbounded quantification. One might argue for range restrictions on the grounds of efficiency, but as noted earlier, efficiency of this nature is not of concern in an abstract specification.

The potential problem with unbounded quantification is that the body of the universal quantifier may not have the correct value in an unbounded range, and hence the value of the entire quantifier expression may be false when we expect it to be true, or may throw an exception, which we do not want.

This is in fact the case in the unbounded quantification used in the sorting predicate for `find`. Specifically, the evaluation of `return.get(i)` throws an exception if `i` is outside the bounds of `return`.

The exact outcome of the unbounded quantification depends on the semantics, i.e., formal definition, of a particular specification language. In general, however, unbounded quantification is potentially problematic under any logical semantics. The point is that one needs to be careful when using unbounded quantification to ensure that the body of the quantifier has a well understood value over the entire unbounded range of quantification. This is particularly the case when quantifying over the elements of a list.

4.4.5. Using Auxiliary Functions

The postcondition in the most recent definition of `find(String name)` is a little lengthy. In practice, predicates significantly longer than this can appear in the specification of a complex operation. When pre- or postconditions become unduly long, it is useful to use auxiliary functions to organize the logic.

In Spest, an auxiliary function is defined as a boolean-valued method in the class where the function is used in a predicate. The logic of the auxiliary function is defined with a postcondition of the form "`return == ...`", where "`...`" is a boolean expression that appears in one or more predicates.

The purpose of an auxiliary function is modularize a piece of logic, give it a mnemonic name, and allow that logic to be invoked in one or more places. This can help make predicates more readable and understandable.

As an example, here is the preceding definition of `find(String name)` using two auxiliary functions:

```
abstract class Rolodex {
    Collection<Card> data;

    /*
        pre: // none yet
        post:
            cardsFound(name, return)
              &&
            sortedById(return);
```

```
            // The output list is sorted in ascending order by card id.
     */
    abstract List<Card> find(String name);

    /*
        post:
            // Return true when a card is in the given cards list
            // if and only it is in the input Rolodex and the
            // Card name equals the name being searched for.
            return ==
                forall (Card card_found;
                    cards.contains(card_found) iff
                        data.contains(card_found) &&
                            card_found.name.equals(name))
     */
        abstract boolean cardsFound(String name, Collection<Card> cards);

    /*
        post:
            // Return true if the given card list is sorted
            // in ascending order by card id.
            return ==
                forall (int i; (i >= 0) && (i < cards.size() - 1);
                    cards.get(i).id <= cards.get(i+1).id);
     */
     abstract boolean sortedById(List<Card> cards);
}
```

5. User-Level Refinements and Enhancements

In this section we consider a number of refinements and enhancements to the user interface of the Rolodex system and how these can be specified formally.

5.1. Pattern-Based Search

Suppose we would like to locate Rolodex cards by keys other than just the name. To be fully general, we could allow search by patterns for any one of the keys. At the interface level, the Find operation would present the same dialog used for Add. In the case of Find, entries in the dialog box would be patterns rather than just strings or numbers. For example, Figure 5 shows a search dialog that will find all cards with age less than 40 and gender male. An entry for any one of the five card fields can contain a single instance of one of the following patterns:

Operator	Meaning
x	matches the value x (a string or number)
< x	matches all values less than x
> x	matches all values greater than x
x - y	matches all values between x and y

For simplicity, we assume that exact match is necessary for strings, but this requirement is probably too strict for a practical user interface. E.g., a user should be able to enter "Smith" in the name field to find all cards with "Smith" somewhere in the name. The reader is invited to enhance the specification that follows with a feature for such partial matching.

Given below are selected excerpts of the formal specification for the enhanced Find operation. The specification focuses on searching with patterns in the Age field of a card. Formal specification of searching by the other card fields (name, id, gender, and address) is very similar. The gist of the pattern-search specification is the definition of

Enter Search Information:

Name:

Id:

Age: < 50

Gender: M

Address:

OK Clear Cancel

Figure 5: A Pattern Search Dialog.

the object SearchInfo and the additions to the postcondition of Find. The comments in the code the major objects and operations further

```
import java.util.Collection;
import java.util.List;

abstract class PatternBasedSearch {

    /**
     * Find zero or more cards that match the constraints specified in the
     * given SearchInfo.
        post:
            // All cards in the output list must be found according
            // to the given search info, and the output list must be
            // sorted by Card id.
            cardsFound(r,si,return)
            //    &&
            //sortedById(return);
     */
    abstract List<Card> find(Rolodex r, SearchInfo si);

    /*
        post:
            // Cards in the given card list consist of those, and only
            // those in the given Rolodex that match the given search info.
            forall (Card c;
                return.contains(c) iff r.contains(c) && match(c,si))
     */
    abstract boolean cardsFound(Rolodex r, SearchInfo si, List<Card>cl);

    SearchInfo si2;
    /*
        post:
            return ==
                matchName(c.name, si.np) &&
                matchId(c.id, si.idp) &&
                matchAge(c.age, si.ap) &&
                matchGender(c.gender, si.sp) &&
```

```
                matchAddress(c.address, si.adp);
     */
    abstract boolean match(Card c, SearchInfo si);

    abstract boolean matchName(String name, NamePattern np);
    abstract boolean matchId(int id, IdPattern idp);

    /*
        post:
          return ==
            if (ap.op == PatternOp.AgeEqual) (age == ap.age1) ||
            if (ap.op == PatternOp.AgeLessThan) (age < ap.age1) ||
            if (ap.op == PatternOp.AgeEqual) (age == ap.age1) ||
            if (ap.op == PatternOp.AgeLessThan) (age < ap.age1) ||
            if (ap.op == PatternOp.AgeGreaterThan) (age > ap.age1) ||
            if (ap.op == PatternOp.AgeRange)
                (age >= ap.age1 && age <= ap.age2)
     */
    abstract boolean matchAge(int age, AgePattern ap);

    abstract boolean matchGender(MaleOrFemale gender, GenderPattern gp);
    abstract boolean matchAddress(String address, AddressPattern ap);
}

/**
 * Each component of SearchInfo is a search pattern that corresponds to
 * one of the fields of a card.
 */
class SearchInfo {
    NamePattern np;
    IdPattern idp;
    AgePattern ap;
    GenderPattern sp;
    AddressPattern adp;
}

class NamePattern { /* ... */}
class IdPattern { /* ... */}

/**
 * An AgePattern allows the user to search for cards with an age value
 * less than a given age, greater than a given age, between a range of
 * given ages, equal to a specific age, or with a specific age.
 */
class AgePattern {
    PatternOp op;
    int age1;
    int age2;
}

enum PatternOp {AgeEqual, AgeLessThan, AgeGreaterThan, AgeRange}

class GenderPattern { /*... */}
class AddressPattern { /*... */}
```

5.2. Historical Dialogs

It is typical in a graphical user interface that a dialog retains values from the last time it was displayed. For example, when the Add dialog is displayed for the second time and beyond, it could contain the last values entered. Some

users may find such historical dialogs undesirable, so the system could contain an option that turns the feature on or off. With historical dialogs on, each dialog box displays the previously entered value(s). With historical dialogs off, each dialog box is empty when displayed.

Given below are selected excerpts of the formal specification for optional historical dialogs. The gist of the specification is the definition of the SystemState object and the decomposition of the main Rolodex operations into two operations. For example, Add is decomposed into InitiateAdd and ConfirmAdd. From the user interface perspective, InitiateAdd is invoked by selection of Add in the Rolodex menu; ConfirmAdd is invoked by selection of the OK button in the Add dialog.

```
import java.util.List;

abstract class HistoricalDialogs {
    /*
      post:
        // If the historical dialog option is on, then output the previously
        // entered card data, else output empty card data.
        if (r.state.options.showprevdata)
             (cd.c == r.state.lastAddInput)
        else (cd.c == null)
     */
    abstract CardData initiateAdd(Rolodex r, CardData cd);

    /*
      pre: // Same precondition as original add,  but replace all occurrences of
           // variable c with cd.c
      post: // Same postcondition as original Add, with same replacements as in
            // precondition
     */
    abstract Rolodex confirmAdd(Rolodex r, CardData cd);
}

/* See discussion below */
class CardData {Card c; boolean flag;}

class Rolodex {SystemState state; List<Card> data;}
class SystemState {Card lastAddInput; int lastDeleteInput;
    LastChangeInput lastChangeInput; SearchInfo lastSearchInput;
    Options options;}
class LastChangeInput {String name; Card c;}
class Options {boolean showprevdata; /* ... */}
```

The CardData object has no other purpose than to constrain a Confirm operation to take input from the companion Initiate operation. The flag component of CardData has no other purpose than to ensure that Card and CardData are distinct types. This specification borders on too operational, since its purpose to to specify an order of operations. In general, the specification of such ordering should be done judiciously, since it constrains an implementation to a particular style of interaction.

5.3. Check Pointing

As a file-based system, the Rolodex specification has a `File->Save` operation that the user explicitly invokes. Software like may have a feature that saves user data at regular intervals on behalf of the user.

Given below are selected excerpts of the formal specification for such checkpointing functionality. Note the addition of a `FileSpace` object, which is a model for an external operating system file storage. This is used further below, in the specification of the Rolodex File operations.

```
abstract class Rolodex {
    Collection<Card> data;
    SystemState state;
    FileSpace fs;

    /*
     post:
        // previous postcondition logic, with the following &&'d on:

        if (state.options.checkpointOn)
          (if (state.checkpointCount == 0)   // it's time to do checkpoint save
           (state.checkpointCount' == state.options.checkpointInterval &&
            exists (File f; fs.contains(f); f.data.equals(this)))
        else                                 // not time yet to checkpoint
         (state.checkpointCount' == state.checkpointCount - 1))
     */
    void add(Card card) { boolean b = state.options.checkpointInterval == 10;}

}

class FileSpace {Collection<File> files;}
class File {String name; FileData data;}
abstract class FileData extends Rolodex {
    /* Abstractly, FileData is just a Rolodex */};
class SystemState {Options options; int checkpointCount;}
class Options {/* previous components and */
    boolean checkpointOn; int checkpointInterval;}
```

There is also the addition of options to turn checkpointing on and off, and to specify the checkpointing interval. The interval is defined as the number of Add operations that have occurred since the last checkpoint. The interval could be extended to apply to any other Rolodex operation by adding the same logic to the postcondition of the other operations.

Also of note is the logical idiom used to specify the incrementing of a value, in this case the checkpoint counter. The logic to do this is

```
state.checkpointCount' == state.checkpointCount - 1
```

A likely implementation of this postcondition would use an assignment statement like this

```
state.checkpointCount = state.checkpointCount - 1
```

which performs the increment. In thinking about this implementation, one might initially specify the postcondition like this

```
state.checkpointCount == state.checkpointCount - 1
```

However, this does not make good sense, since as a boolean expression it's always false. This is a good example of why the prime notation is necessary to distinguish between the pre-value and post-value of an expression.

The '==' equality operator is not of course the same as an assignment operator, even though it takes on the characteristics of assignment in cases like incrementing. In general, an equality expression can "feel like" assignment when we are specifying a behavior that is directly implemented using assignment.

5.4. Undo

It is common for systems to have an Undo function that reverses the effects of the most recent operation. Given below are selected excerpts of the formal specification for a simple one-level undo facility. The specifications for Add and Undo operations are given. A complete specification would include updates to Delete and Change, comparable to those for Add. The specification for the more typical form of undo that allows multiple operations to be undone would use a list of undone values rather than a single value.

```
import java.util.Collection;
```

```
/* Same as for historical dialogs, plus one added field */
class SystemState {Card lastAddInput; int lastDeleteInput;
    LastChangeInput lastChangeInput; SearchInfo lastSearchInput;
    Options options;
    Collection<Card> prev_data;}  // Added field for previous value of data

/* Same as earlier Rolodex, plus added SystemState field. */
abstract class Rolodex {
    Collection<Card> data;
    SystemState state;

    /*
        pre:  // as above
        post:
            // A card is in the output Rolodex if and only if it is
            // the card to be added or it is in the input Rolodex.
            forall (Card a_card ;
                data'.contains(a_card) iff                          //'
                    a_card.equals(card) || data.contains(a_card))
                &&
            // The previous state data are the input data
            state.prev_data'.equals(data);
     */
    abstract void add(Card card);

    /*
        post:
            // If the input Rolodex has a previous card data,
            // then the cards of the output Rolodex are that data
            // and the previous of the output is null (so only one
            // level of undo is possible).  Otherwise, the output
            // Rolodex is the same as the input Rolodex.
            if (state.prev_data != null)
                ((data'.equals(state.prev_data)) && (state'.prev_data == null))
            else
                (data'.equals(data));
     */
    abstract void undo();
}
```

5.5. Security

Security can be specified in a number of forms. A basic form is to have different levels of users, with different access privileges.

In the case of our simple Rolodex, a plausible security requirement would be to allow only privileged users to perform the Add operation. Given below are selected excerpts of the formal specification for such a security scheme. The gist of the specification is to define a UserTable containing UserInfo records. The records are tuples of a UserId and privilege Level. Each user is assigned a system Id (potentially different from a card id) and a privilege level. When a user logs in to the system, the user id and password are supplied, whereupon the privilege level is extracted from the user table.

Based on the user level, a SystemState value is returned with a flag indicating whether or not the user can add. This system state is an initialized value, including an initial empty collection of `prev_data`. It is used to initialize the system state value in a rolodex. The specification below does not define the operations necessary to maintain a user table, i.e., adding and deleting user records. It also does not address any issues of password encryption, since the password is represented as a plain string.

```
import java.util.Collection;
```

```
/* Same as for undo dialogs, with one field added. */
class SystemState {Card lastAddInput; int lastDeleteInput;
    LastChangeInput lastChangeInput; SearchInfo lastSearchInput;
    Options options; Collection<Card> prev_data;
    boolean addOK;}  // Added field for indicating if add is OK

abstract class UserTable {
    Collection<UserInfo> userInfo;
    /* Spec for this find user operartion are comparable to Rolodex.find */
    abstract UserInfo find(String uid);
}
class UserInfo { String uid; String password; Level level; }
enum Level { Privileged, Nonprivileged }

abstract class AccessControl {
    UserTable users;
    /*
        pre:
           exists (UserInfo user ; users.userInfo.contains(user) ;
              user.id.equals(id) && user.password.equals(password));
       post:
           // Adds are OK if the given user is privileged,
           // otherwise adds are not OK.
           if (users.find(id).level == Privileged)
               (return.state.addOK == true)
           else
               (return.state.addOK == false)

              &&

           return.prev_data == null

           // and all other fields of return are null
     */
    abstract SystemState login(String id, String password);
}


/* Same as earlier Rolodex, with updates add method precondition. */
abstract class Rolodex {
    Collection<Card> data;
    SystemState state;    // Initial value set to return value from login

    /*
        pre:
            // Same as above, plus this additional &&'d clause:
            state.addOK;
        post:
            // Same as above
     */
    abstract void add(Card card);
}
```

6. Rolodex File Operations

There are number of approaches for defining the file operations of the Rolodex system, or other comparable system that uses external file storage. The approaches vary in how abstractly versus how concretely the external file storage medium is modeled.

The file space model presented here is very abstract. It defines a file as having a name and rolodex file content. The model does not specify file permissions or other file attributes of which the rolodex user might be aware, and which a complete rolodex system would deal with. For example, if the user tried to open an unreadable rolodex file, the rolodex tool should provide the user with a suitable error message. Doing this would require refinement of the File object to contain permission information.

One aspect of the abstract File model that does not necessarily require refinment is the representation of file data directly as a Rolodex object. This means that it is entirely up to the implementation to convert a Roldodex value into a form suitable for saving on a file. This is considered to be a detail that is reasonable to leave to the implementation, as long as rolodex content on a file are assumed to be unreadable in saved form.

If we wanted a user to be able to view a rolodex file in some human readable form, such as plain text for example, then the specification of rolodex file data would have to be defined in a string-based form. This in turn would require the specifications of the open and save operations to be extended to parse text and generate text, respectively.

Here now is the abstract specification of file operations:

```
import java.util.Collection;

class FileSpace {Collection<File> files;}
class File {String name; FileData data;}
abstract class FileData extends Rolodex {
    /* Abstractly, FileData is just a Rolodex */};

public abstract class AbstractFileSpace {

    Collection<Rolodex> files;

    /*
        post: return.data == null;
     */
    abstract Rolodex fileNew();

    /*
     * Open is essentially a find operation, with the same form of
     * postcondition as a basic find.

        post:
            exists (File f ; files.contains(f) ;
                f.name.equals(name) && return.equals(f.data))
                    ||
            !exists (File f ; files.contains(f) ;
                f.name.equals(name) && return == null);
     */
    abstract Rolodex open(String name);

    /*
        pre:
            // The given rolodex has already been saved
            exists (File f; f.data.equals(rolodex));
        post:
            // This postcond is uses the same logic as Rolodex.add
            forall (File f;
                files'.contains(f) iff
                    f.data.equals(rolodex) || files.contains(f));
     */
    abstract void save(Rolodex rolodex);

    /*
        post:
```

```
            forall (File f;
                files'.contains(f) iff f.name.equals(name) &&
                    f.data.equals(rolodex) || files.contains(f));
     */
    abstract void saveAs(Rolodex rolodex, String name);

    /*
     * The specs for print are beyond the scope of this primer.  They rely
     * on the data specification of the printed output, and once that is
     * specified, the postcond of print defines a mapping from a Rolodex
     * to the specified printed form.
     */
    abstract PrintedRolodex print(Rolodex rolodex);
}

abstract class PrintedRolodex { /* Some appropriate printed form */ }
```

7. Considering Other Interface Styles

An important property of an abstract specification is to be as free of concrete interface details as possible. To illustrate this point, we can consider two different forms of concrete user interface, both of which map to the same abstract specification that has been developed in the primer.

Figure 6 shows a pushbutton-style interface to the Rolodex. It is much the same as the original interface, except the pulldown menus have been replaced with pushbuttons. In addition, the data entry area does not change. Rather, the user simply types in the card data fields, and selects a desired operation when done. Additional dialogs will pop up as necessary to request further inputs or display results.

The only significant difference for the specifier with this interface is that the both the Card and SearchInfo objects are displayed in the same physical screen area, rather than in separate dialogs. Initially, it might have been slightly more difficult to recognize Card and SearchInfo as separate objects.

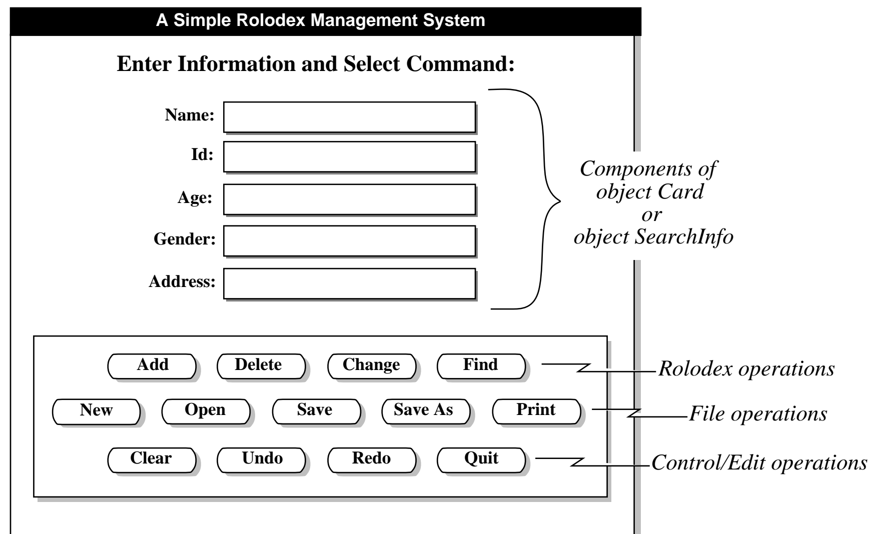

Figure 6: A Pushbutton-Style UI.

It is worth noting that to the extent the interface is confusing to the specifier, it may well be the end user as well. The notion of "form follows function" is an important one in software specification. That is, a system that is coherent and easy to specify for the analyst will be equally coherent and easy to use for the end user, and vice versa.

In either of the graphical interfaces for the Rolodex system, keyboard shortcuts could be available for the menu and/or pushbutton operations. Such shortcuts should have absolutely no effect on the formal specification.

Figure 7 shows a plain text-based Rolodex interface. This interface style is typical of that used with UNIX shell. The same abstract Rolodex specification applies equally well to the textual interface as it does to the other two graphical interfaces.

Command	**Arguments**
a[dd]	name, id, age, gender, address
d[el]	name
c[hange]	name, id, age, gender, address
f[ind]	name
n[ew]	
o[pen]	file
s[ave]	[file]
[p]rint	[1]
[u]ndo	
[r]edo	
[q]uit	

Figure 7: A UNIX Shell Style Text UI.