On The Development Of Library Units

Jaime Niño
Computer Science department
University of New Orleans
New Orleans, LA 70148
jaime@cs.uno.edu

ABSTRACT

The teaching of the development of library units seems to lack its own module for undergraduate education, although references to such topic can be found mainly for the professional programer. CS2 provides a great opportunity to teach an initial set of best practices and patterns to design library units which provide students with a strong foundation for later use in the development of APIs. We present a carefully selected set of patterns and best practices culled through several years teaching data structures. These patterns and practices are aimed to guide the student in the specification, implementation and testing of library units.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education**]: Design, implementation, testing.

General Terms

Containers, library, patterns, best practices.

Keywords

Containers, library, design, implementation, testing.

1. INTRODUCTION

Teaching data structures provide us with a great opportunity to instill in the students the need for good *specification*, *design*, *implementation*, *testing* and *documentation*. This opportunity is unlike any other in the teaching of software development due to the fact that clients of library units are programmers who build other library units or applications with them and for whom the quality of the stated properties is essential for their usability and longevity. This is contrasted with the development of applications where the goodness of these properties becomes relevant mainly during maintenance, a software facet to which students are usually not exposed during their tenure.

In this paper we present a set of fundamental best practices and patterns [2] that provide students with a guide to follow in the development of library units. These practices are intended to be illustrated by the instructor and put into practice by the students in CS2. The guide covers library development cycle from unit specification to testing. We will use Java in the exposition, although all programming features used can be readily mapped to features in other similar languages. Throughout we will use examples of a Java interface `List` [see appendix] as the setting in which to illustrate issues.

/SIGCSE'10/, March 10-13. 2010, Milwaukee, WI USA.

There is literature, [3] among others, exposing the use of patterns in CS2; this paper goes beyond that by presenting a guide of the complete development process of a library unit based non only on identifiable patterns but also best practices.

Due to length limitations no discussion is given on the representation of data as values or as mutable data. Since the choice of language for the examples is Java, data representation can be seen as mutable. Nor we have space to illustrate how an initial implementation in terms of concrete types might get evolved into a generic one. Neither we discuss testing methodologies.

A shorter preliminary version of this paper was given in [7]. This is a completely revised and expanded version augmenting the ideas found therein as well as providing a larger report on classroom experience.

In the paper, by containers we mean classes whose instances are objects that contains instances of other homogeneous objects.

2. DEVELOPMENT METHODOLOGY

Library unit development process should proceed via the most fundamental iterative code development methodology:

- *Specify a little - Design a little - Implement a little - Test a little*

The nature of specifying and implementing a library unit gives the student the opportunity both to be the author of and a client of the unit. It gives the opportunity for exploration of the proper interface of methods as well as their usability. The simultaneous development of test methods gives the programmer immediate feedback to the feel, usability and usefulness of the library. And regardless of the process methodology used by the eventual software shop a student joins, a developer will be well-served by building on this basic software development methodology to grow and evolve programs.

3. ABSTRACTION

Commonly a library unit provides different implementations of its specification. Hence, the best-practice to be used for its specification and the management of different implementations is

- *Abstract what varies*

To abstract in this context consists of using a construct, such as a Java interface or abstract class, that allows for the separation of the specification of the unit's functionality from the tree of different implementations. And that is what varies, i.e. the possible different implementation flavors based on data structures choices or method implementations, while the unit's semantics remains the same throughout all these implementations.

4. LIBRARY REQUIREMENTS

An initial analysis of the data structures presented in CS2, data containers, impose on them a starting set of requirements:

Requirement 1: Containers should be independent of the type of data maintained therein.

Requirement 2: Each container can have, and it usually does, several different implementations.

Requirement 3: A programmer should easily provide a new implementation for a container. In turn, a container's client should not be concerned about implementation choices and, when demanded, should be given a transparent access to each regardless of possible future changes in implementations.

The first requirement places a mandate on the generality of the specification of the data to be placed in the container. The second and third requirements dictate a judicious design of a library unit as its choice of implementation will impact both the clients as well as their implementers. Specifically, clients should be isolated from the different implementations when using the abstraction, while at the same time the library should provide clients a way, as transparent as possible, to select a desired implementation. On the other hand, implementers should be provided with an interface that *minimizes* the work needed to provide new implementations.

5. LIBRARY SPECIFICATION

Based on this initial analysis, we can start the guide on the quality of the specification:

- *Make the specification of the type of data in the container as generic as possible*

Generic data types allow clients to reuse the container by simply instantiating the container with the desired entry type(s).

- *Specify data containers via Java interfaces*

Java interfaces provide a mechanism to separate specification hierarchies from implementation hierarchies. This in turn reduces the number of methods that must be known about a library unit by its clients, while the definition of the type is opaque to them.

5.1 Unit Cohesion

Turning to the quality of the abstraction that a library interface should provide, the containers at hand (lists, stacks, queues, trees, etc.) yield a criteria for free:

- ***Single Responsibility principle****: The data container interface must provide one cohesive abstraction, and thus one reason to change during its evolution*

The Java interface must abstract only one type of container. An interface should not attempt to model both stacks and queues, which, although having some similarities, are fundamentally different. This in turn minimizes the number of members the library unit must have and the effort spent on its life-cycle. Also, it allows for easier usability by clients, as the methods are related for the purpose at hand. Last but not least, it reduces its memory footprint.

5.2 Member Specification

For the actual specification of the members of the library unit, there are a few fundamental criteria to keep in mind [1, 5]:

- *Be consistent*

The `List` interface provides a method to add at a given location in the list as:

```
public void add(int location, Element data);
```

The set method to change an element at a given location should not be specified as:

```
public void set(Element data, int location);
```

The client with partial knowledge of the interface should be able to predict it to be:

```
public void set(int location, Element data);
```

- *Be minimal*

A poor way to achieve consistency can be by providing the methods `add` and `set` above in their two different forms. Reduce unnecessary duplication of methods.

- *Methods names matter*

Their purpose should be self documenting. Avoid cryptic abbreviations. Be consistent about the use of the name. Example: do not use the name `remove` sometimes, while other times you name the same kind of method `delete`.

- *Favor queries*

Queries are methods that return values with no change of state of the object. By their nature they are easier to develop.

- *Minimize the number of commands*

Commands are methods that may change the state of the object. Contrary to queries, these are harder to develop and debug.

- *A method is either a query or a command but not both*

Queries that change state are designed for efficiency. Split one such into a query and a command. It simplifies the semantics of methods to clients. This design choice in turn enhances its usability and eliminates undesirable and possibly nasty surprises. But their use for efficiency have its place in the implementation as private methods, and where clients do not have access to them.

- *Favor parameters' type specified via Java interface types*
- *Use appropriate parameters and return types*

Specifically, generalize parameter types using Java interfaces to take advantage of polymorphism, and specify the exact return type for the benefit of client documentation.

- *Avoid long parameter lists*

A long parameter list is a symptom that more abstraction is needed to encapsulate some of those parameters.

- *Do not overload a type for semantically different values.*

As an example, `int` is usually overloaded to semantically different values (age, year, etc.) providing the opportunity for the client to match semantically incorrect values. Use instead `enum` types in particular, or distinct types in general.

The above criteria will help clients use the container's interface appropriately while avoiding confusion and surprises.

6. ENTRY FUNCTIONALITY

There are operations on the container that depend on the functionality of its entries, such as their *order* for sorting, or an entry operation to be applied to all entries via an iterator. Although entry operations are outside the responsibilities for a container specifications, the latter should provide functionality parameterized by an entry operation to allows client to perform entry operations on all the container's data:

- *Use function objects to specify methods that depend on functionality of the container's entries.*

Use an interface to abstract a function that specifies functionality on the container entries (a so-called *function object*). The container in turn specifies functionality depending on the entries' functionality having the function object as a parameter type. At execution time, the client provides as argument an instance of such function object. As examples, function objects are used to specify the element's order for sorting and searching. The sort interface specifies as parameter a *compare* function (in Java use the `Comparator` interface, a function object) while the client invokes sorting (or searching) passing the specific implementation of *compare*. Similarly, we can abstract an operation to be applied to all container entries via an internal iterator on the container [see appendix `forEachDo`]. Here again, we abstract the entries' functionality using an interface as what varies are specific entry operations to be given by the client.

7. LIBRARY DOCUMENTATION

We start by quoting D. Parnas [8]

> *Reuse is something that is far easier to say than to do. Doing it requires both good design and very good documentation. Even when we see good design, which is still infrequently, we won't see the components reused without good documentation.*

Library programmers and clients need to completely understand the kind of abstraction provided by the interface, as well as the correct way to use it. Therefore good documentation is essential for its proper use, evolution and unit's lifetime on the hand of clients.

7.1 Documentation Of Specification

- *Provide a prolog paragraph in the interface stating its intent, and proper use*

It states what is being modeled with possible mention of limitations. It also includes examples of its use.

- *Identify and state abstraction invariants*

These guide the clients for what is always true, and the programmers to develop correct implementations.

- *Methods have contracts*

Each method should have a documentation header in the form of a contract [5] between the interface (the server) and its clients. This contract is stated via pre-conditions (*requires*) met by the client, and post-conditions (*ensures*) delivered by the server.

7.2 Documentation Of Implementation

- *Strive for self-documenting code*
- *Use invariants for loop documentation*
- *Aim for correct documentation. Prefer terse over verbose*

8. LIBRARY IMPLEMENTATION

In general, since there are several implementations for a given container, we must carefully design the overall structure of the implementations to reduce repeatability of code while reducing the effort to provide a new implementation. A useful pattern here is the *Narrow Interface* pattern. In this pattern the interface methods are classified in two kinds, *primitive* and *composite*, and makes use of abstract and concrete classes:

- *Identify primitive methods*

These are methods that can be written without the use of other methods of the unit. They may be implementation dependent.

- *Identify composite methods*

These are methods that can be written in terms of primitives.

- *Use abstract classes to factor out common code*

The abstract class will contain implementation of the composite methods and implementation independent primitives.

- *Subclass to provide concrete implementations*

Finally, concrete classes of the abstract class provide the implementations of the primitive methods.

As example, in the `List` interface we can identify several composite methods; among those, to add at the end, to remove an element, to search for an element. This process leaves 6 primitive methods: `size`, `copy, add, remove, set,` and `get`, where each of the last four specifies a position in the list as parameter [see appendix]. These methods may be implemented directly or by making use of an existing class via the *Wrapper* pattern. As example, the `List` functionality can be implemented directly using an array or by wrapping an ArrayList implementation.

9. CONCRETE CLASSES

Much has been written on the design and implementation of classes [2,3,4,5] and the use of patterns for this aim. Below are several basic best practices to keep in mind for classes:

- *Determine and state class invariants. Be sure that commands maintain them*
- *Data members ought to be private*
- *State of class should be small in number*
- *Concrete classes should be final*
- *Make copies of objects used in the initialization of instance variables*

When an instance variable references instances of a mutable class, initialize it using a copy of the reference to avoid unexpected or undesired effects by the reference.

10. SEPARATING SPECIFICATION FROM IMPLEMENTATION UNITS

Since containers have different implementations, we should provide the client with a library method that yields an instance of a client's specified implementation.

- *Use factory methods that generate specific container's implementations*

Factory methods allows for the hiding of class constructors. In this fashion the client does not need to know constructors' specifics. Hence, concrete classes can be made non-public, which will allow for the re-implementation of such for maintainability, performance and even renaming without breaking the client's code. Use the parameters of the factory methods to allow the client to specify the desired implementation. Using Java we cannot provide factory methods directly in the interface as static methods cannot be its members. They must be specified and implemented in a builder class that has access to the implementation choices. Such class will contain a static method `generate`, which takes as parameter the kind of implementation desired. Use static constants or an enum type to name existing implementations. As example below is a client's invocation of a factory method to generate an array-based list:

```
List<Integer> list =
ListBuilder.generate(ListBuilder.ArrayBased);
```

11. SUBTYPING ABSTRACTIONS

Subtyping interface types or class types must be done very judiciously. You must ascertain that the relationship between the subtype and the supertype is indeed an IS-A relationship. This relationship can be expressed via the Liskov Substitutability principle (LSP) as: *subtype instances can substitute supertype instances in arguments to method invocations and instantiation of objects*. In terms of method contracts, the LSP implies that supertypes' method pre-conditions must not be strengthen while supertype's method post-conditions must not be weaken in subtype contracts.

- *Subtypes implement the IS-A relationship*
- *Maintain the LSP. Subtype contracts do not strengthen pre-conditions or weaken post-conditions*
- *Subtypes maintain supertype's invariants*

For instance, `List` models a container where any number of homogeneous elements can be added, hence `BoundedList` it's not its subtype. The latter imposes a bound on the number of elements added; thus its instances cannot substitute `List`'s instances. Such subtyping would break the LSP. Examining the contract for `add` in `BoundedList`, it *requires* the list to have space to add an element; while for a `List` such requirement is void as it implicitly grows as needed. Therefore `add`'s pre-condition has been strengthen in `BoundedList`. Likewise `OrderedList` should not be a subtype of `List`, as its `add(element)` method places the element in order, while the corresponding `List add` appends it. Hence, in `OrderedList` the `add(element)` post-condition has been weakened as there is no guarantee of the position of the element added. In Java collections `Stack` is erroneously a subytpe of `Vector`.

- *Document use of composites for inheritance*

Note that following the *Narrow Interface Principle* in the implementation of a container's interface gives rise to self-calls: composite methods invoke primitive ones. From the point of view of performance, these self-calls must be documented for inheritance, as the re-implementation of primitives may impact the performance of the composites. Also when considering a new library implementation, primitive methods efficiency may impact the efficiency of composites. As example in `List, indexOf` is a composite written in terms of `get`, which is very efficient for array-based implementations. For linked-based implementations `get`'s performance degradation demands a direct implementation of `indexOf`.

12. VALIDATING CLIENTS' CONTRACT

Initially, we can start checking for object's state and correctness of input parameters via `assert` statements. This mechanisms aborts execution when the assertion is false, and does not provide helpful error message to the client nor a way to recuperate from the error.

Although the assert choice has its place in the early phases of development, the best choice to take is to deal with errors via exceptions. The question then is whether to use *unchecked* exceptions or *checked* exceptions.

- *Use unchecked exceptions wherever possible.*

Among reasons for this choice are:

1. Checked exceptions must be part of method specifications for the client making code evolution harder to attain when the exceptions need to be re-specified or new ones are added.

2. Checked exceptions affect client code as the client will need to explicitly provide try-catch blocks, or the client's method need to be specified to throw exceptions as well.

Use of unchecked exceptions, when warranted, eliminates these two issues and their error messages are sufficient.

13. EXCEPTIONS IN SERVER'S CODE

In the implementation of a method, the server may need to deal with exceptions. The main issue here is that of exceptions being raised in command methods, where the state of the object may be updated.

- *Keep object's state consistent in the face of exceptions.*

Make sure to leave the object in a *consistent* state in this situation. To this end, compute change of state using temporaries which will be used to update corresponding instance variables after all the temporaries have been successfully computed.

14. LIBRARY TESTING

The development of a library unit requires a very complete test harness. Develop a test bed using unit tests (JUnit is one choice) whereby each method development is supported with a set of test methods that are run as part of a suite of tests for the whole library unit.

- *Develop unit tests for regression and automation.*

Unit testing have these two characteristics. Regression testing provides a written programmed record of the expected behavior of the library unit, and it is used in testing during maintenance and evolution. On the other hand, automatic testing allows for the test results to be verified by the computer rather than by hand. Furthermore,

1. Testing provides a mechanisms to *write* to the unit's interface, i.e. the programmers are its clients. This in turn can be used to test the library completeness, effectiveness of purpose and usability.

2. The test harness becomes part of the library specification and documentation to be used during maintenance and evolution.

15. CLASSROOM EXPERIENCE

The author has been developing and using this guide for several years in CS2. Several of these principles and patterns appear in [2,5,7]. The guide is presented in a spiral approach starting with the specification and implementation of the `List` container. At the end of linear containers a formal presentation of the guide's principles and best-practices is given. Java Collections API is also used for further illustration.

Through homework students gain experience at container specification, initially by completing one started in lectures, later by supplying a complete one for containers such as `Bag, Set` or any of the several flavors of lists. Although students usually struggle with specifications, the main difficulty initially found is in method naming, proper documentation of the interface and the correctness of the methods' contracts.

When it comes to the implementations of an interface, students are expected to identify the composite methods that will appear implemented in the abstract class. Finally they provide concrete implementations as subclasses of it. Again proper documentation is a major issue, including the identification of class invariants and the writing of loop invariants. Students also struggle with method size.

Unit tests are also expected. Initially students tend to make unit tests for specific instances of the generic library unit at hand. When asked tests for different type instances, they initially fail to see or are not bothered by the repeated code generated. Since interface and implementations are generic, unit test development becomes a bit thorny as its testers methods should be generic as well [4,6].

I can happily report that towards the end of the semester students follow the guidelines and the initial difficulties identified earlier have largely vanished. The latter is due to the continuous repetition afforded by the numerous containers typically covered in CS2.

At semester's end and as closing to the overview of the library unit development principles presented, the main design characteristics are highlighted and are presented to serve as a guide to API design in general. A library unit developed using the guidelines presented above, and in lectures, exhibits the following characteristics:

- *Easy to use: Unit models only one thing and does it well.*
- *Easy to learn: Unit's interface fits its intended purpose.*
- *Hard to misuse: Unit provides no surprises to its users.*
- *Easy to implement: Implementer's interface is well-defined and small.*
- *Easy to read and maintain: Unit can be evolved due to its documentation and test harness.*

Since API development is a far more common activity than the development of applications from scratch, the singling out of these characteristics preceded by their illustration in CS2 is a major learning opportunity to be given to the students and it's fundamental to their growing software development maturity.

16. CONCLUSION

Teaching CS2 can be a very illuminating and richer experience for the students when the containers are presented under the light of principles of library unit design and implementation. The principles and patterns presented provide an initial, fundamental and by far complete set that will enrich the software development experiences we can provide to our students. The author strongly feels that these guidelines form a fundamental set to build on in later courses such as Software Engineering among others.

17. REFERENCES

[**1**] Hoffman D. On Criteria for Module Interfaces. *IEEE Trans.on Soft. Eng*. V16, 5, 1990.

[**2**] E.Gamma, R.Helm, R.Johnson, V.Vlissides. *Design Patterns.* Addison-Wesley. 1995

[**3**] N. Gelfand, M. T. Goodrich, R. Tamassia, Teaching Data Structure Design Patterns, ACM SIGCSE '98

[**4**] N. Naftalin, P. Wadler. Java Generics and Collections. O'Reilly, 2006.

[**5**] Niño,J. Hosch F. *An Introduction to Programming and Object Oriented Design Using Java*. Third Ed.Wiley. 2008.

[**6**] J.Niño. Java From The Trenches: Dealing with Object Orientation and Generics. Journal Of Computer Science in Colleges. Vol 22, No. 5. pgs. 12-22. 2007

[**7**] J.Niño. Introducing API Design in CS2. Journal Of Computer Science in Colleges. Vol 24, No. 4. pgs. 109-116. 2009

[**8**] Parnas, D. Software Aging. *Proc. of 16th Int. Conference Software Engineering*. 1994.

18. APPENDIX

Below you find the `List` interface, and `AbstractList` class, the basis of any complete implementation of `List` following the Narrow Interface Principle. [5] provides complete specification and implementation details.

```
/**
 * This interface models a finite list of Elements. List is not bounded.
 */
public interface List<Element> {

  /**
  * Number of elements in this List.
  * @ensure     this.size() >= 0
  */
   public int size ();

  /**
  * This List contains no elements. this.isEmpty() == (this.size() == 0)
  */
   public boolean isEmpty ();

  /**
  * The Element with the specified index.
  * @require    0 <= index && index < this.size()
  */
   public Element get (int index);

  /**
  * The index of the first occurrence of the specified element,
  * or -1 if this List does not contain the specified element.
  * @ensure     if this.indexOf(element) >= 0
  * this.get(this.indexOf(element)).equals(element)
  * && for all indexes j: j < this.indexOf(element) implies
  * !this.get(j).equals(element). If this.indexOf(element) == -1
  *            for all indexes j, !this.get(j).equals(element)
  */
   public int indexOf (Element element);

  /**
  * This List contains the specified Element.
  * this.contains(element) == (this.indexOf(element) = 0)
  */
   public boolean contains (Element element);

  /**
  * Append the specified Element to the end of this List.
  * @require    element != null
  * @ensure this.size()==old.size()+1,this.get(this.size()-1).equals(element)
  */
   public void add (Element element);

  /**
  * Insert the specified Element at the specified index.
  * @require: element != null, 0 <= index && index <= this.size()
  * @ensure: this.size() == old.size() + 1, this.get(index).equals(element)
  *           for all j: index >= j && j <  old.size() implies
  *           old.get(j).equals(this.get(j+1))
  */
   public void add (int index, Element element);

  /**
  * Replace the element at the specified position with the specified element.
  * @require    element != null, 0 <= index && index <= this.size()
  * @ensure     this.get(index).equals(element)
  */
   public void set (int index, Element element);
/**
  * Remove the element with the specified index.
  * @require    0 = index && index  this.size()
  * @ensure     this.size() == old.size() - 1
  *           for all j: index >= j && j < this.size() inmplies
  *           this.get(j).equals(old.get(j+1))
  */
   public void remove (int index);

  /**
  * Remove the first occurrence of element from this List.
  * Has no effect if the Element is not contained in this List.
  */
   public void remove (Element element);

  /**
  * A copy of this List.
  * @ensure     this.copy() != this, this.copy().size() == this.size()
  *                for all indexes j, this.get(j).equals(this.copy().get(j))
  */
   public List<Element> copy ();
}// end interface List<Element>

/**
 * Base class for implementing interface List. To complete implementation
* implement : size (), get (int), set (int, Element), add (int, Element),
*remove (int), copy ()
 */
public abstract class AbstractList<Element> imple-
ments List<Element> {
  public boolean isEmpty () {
    return this.size() == 0;
  }
/**
  * It iterates through the list invoking this.get(int) until
  * element is found or the list is exhausted. this.size() is also invoked.
  */
  public int indexOf (Element element) {
    // ... removed for lack of space}

  public boolean contains (Element element) {
    return this.indexOf(element) != -1;
  }

  public void add (Element element) {
    this.add(this.size(),element);
  }

  public void remove (Element element) {
    int index = indexOf(element);
    if (index != -1)
      this.remove(index);
  }
}//end abstract class AbstractList<Element>

/**
 * Abstracts an operation on an Element instance.
 * Model a function object.
 */
public interface Operator <Element> {
  public void execute (Element e) throws Exception;
}// end Operator

/**
 * Perform the specified Operation on each element of this List.
 * Add this method to List interface, and implement in AbstractList class.
 */
public void forEachDo (Operator<Element> op);
```