INTRODUCING API DESIGN PRINCIPLES IN CS2

Jaime Niño
Computer Science, University of New Orleans
New Orleans, LA 70148
504-280-7362
jaime@cs.uno.edu

ABSTRACT

CS2 provides a great opportunity to teach an initial set of best practices and patterns that equip students with a strong foundation for use later in the development of APIs. The best-practices presented guide the student in the specification, implementation and testing of units whose clients will be other programmers.

INTRODUCTION

Teaching data structures provides us with a great opportunity to instill in the students the need for good *specifications*, *documentation*, *design*, *implementation*, and *testing*. This is due to the fact that clients of library units are programmers who build other library units or applications with them. This is contrary to applications where these goodness characteristics become desirable and relevant only in 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. This guide covers library development cycle from specification to testing. We will assume the programming language to be Java, although all programming features used can be readily be mapped to features or constructs in other languages. Throughout our discussion we will use fragments of the `List` container to illustrate points where appropriate.

PROCESS DEVELOPMENT METHODOLOGY

The development process of an API fits the most fundamental iterative code development methodology:

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

Regardless of the overall process methodology (agile, waterfall, risk driven, etc.) in use by the eventual software shop a student joins, a skilled developer will be well-served by building on this basic software development iteration.

ABSTRACTION

The overall best-practice to be used for the specification and implementation of library units is

- *Abstract what varies*

To abstract in this context consists of using an abstraction mechanism (interface, or abstract class) that allows for the separation of the specification of desired functionality from implementation choices. Implementation is what varies, i.e. the possibly different flavors about a feature whose specifications remains the same throughout. In the guide we will specifically state the kind of abstraction mechanism to use.

LIBRARY SPECIFICATION

An initial analysis of the data structures in CS2, data containers, provides a starting set of requirement in the design of such APIs:

Requirement 1: Containers should be independent of the type of data contained therein.
Requirement 2: Each container has several different implementations.

The first requirement places a dictate in the generality of the specification of the data to be placed in the container. The second requirement mandates judicious design as its choice of implementation will impact both the clients as well as their servers, i.e. the implementors of the API. Specifically, clients should be protected from the different implementations when using the abstraction, while at the same time the API should provide client a way, as transparent as possible, to choose a desired implementation. On the other hand, implementors should be provided with an interface that minimizes the work needed to provide new implementations for the given API. Based on these requirement we can start the guide on the quality of the specification:

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

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

- *Specify data containers via interfaces*

Interfaces provide a mechanism to separate specifications hierarchies from implementation hierarchies. This in turn reduces the API features that must be known by a client about an API, while making the definition of the type opaque to its clients.

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

- *The data container interface must provide a cohesive abstraction*

The interface must abstract only one type of container. This in turn minimizes the number of features it must have and the effort spend evolving it during its life-cycle. Also, it allows for easier usability of the container by clients, as the interface features are related for the purpose at hand. Furthermore, it reduces its memory footprint.

When it comes to the actual specification of the API interface features, we have a few fundamental criteria to keep in mind [1]:

- *Be consistent*

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

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

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

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

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

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

- *Be minimal*

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

- *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 method that return values with no change of object's state.

- *Minimize the number of commands*

Commands are methods that may change the state of the object.

- *Favor parameters' type specified via interface types*

This use will help client use the API's interface regardless of the specific implementation choice the client may be using.

- *Use appropriate parameters and return types*

Avoid overloading types for different uses. As example, `int` used for enumeration.

- *Avoid long parameter lists*

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

ABSTRACTING DATA COMPONENT FUNCTIONALITY

There are operations on the collection that depend on specific functionality of its elements, such as the order of the elements, or an operation on the elements to be applied via an iterator.

- *Use function objects to specify behavior for the container's entries.*

A function object abstracts the specification of element's functionality using an interface. The API in turn specifies the feature depending on such functionality using the function object interface as a parameter type. The client provides as argument an instance of a concrete class providing the desired implementation. As concrete examples, function objects are used to specify the element's order for sorting and searching. The sort interface specifies an order function (in Java via the interface `Comparator`) while the client invokes sorting (or searching) passing the specific implementation of the order. Similarly, we can abstract a function that abstracts an operation to be applied to a collection of objects via an iterator over the collection. The client in turn provides an implementation of such interface for the specific operation desired. Here again, we abstract the functionality using an interface as what varies are the specific implementations that are provided by clients.

LIBRARY DOCUMENTATION

We start by quoting D. Parnas [5]

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.

API clients need to completely understand the kind of abstraction provided by the interface, as well as the correct way to use it. Therefore documentation is essential for the effectiveness and usability of the API.

- *Provide a prolog paragraph in the interface*

It states what is being modeled; it also includes explanation of sequences of uses of features when required.

- *Methods have contracts*

Each feature should have a documentation header in the form of a contract [3] between the interface (the server) and its client. This contract is stated in the form of preconditions to be met by client at call time, and post-conditions which are delivered by server at call's termination.

- *Use loop invariants for loop documentation*
- *Strive for correct documentation. Prefer terse over verbose.*

LIBRARY IMPLEMENTATION

Since there are several implementations for a given containers, carefully design the overall implementation structure to reduce repeatability of code while reducing the effort to provide a new implementation for the given interface. A useful pattern that help us here is the *Narrow Interface* pattern. In this pattern a container's interface features 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 API.

• *Identify composite methods*

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

• *Use abstract classes to factor out common code*

The abstract class contains implementation of composite methods in terms of primitive ones, and also primitive methods not depending on implementation specifics.

• *Subclass to provide complete implementations*

Subclass concrete classes to provide implementations of the primitive methods.

As example in `List` 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,` and the four methods `add, remove, set, get`, each taking a position in the list as parameter. These methods may be implemented directly or by making use of an existing class, via the *Wrapper* pattern. As example, the `Stack` functionality can be implemented directly or by wrapping a list implementation.

CONCRETE CLASSES

There are several best practices to keep in mind for concrete classes:

• *Determine and state class invariants. Be sure that commands maintain them.*

• *Data members ought to be private*

• *Class state 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.

SEPARATING SPECIFICATION UNITS FROM IMPLEMENTATION UNITS

Since containers have different implementations, we should provide the client with an API feature 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. Use the parameters of the factory methods to allow the client to specify the desired implementation. The use of factory methods permits the hiding of the actual implementations, which in turn will allow for the re-implementation of such, for maintainability or performance, without breaking clients code. Using Java, we cannot provide factory methods directly in the interface as static methods cannot be members of it. 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. An example of a client's invocation of a factory method:

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

SUBTYPING ABSTRACTIONS

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

• *Subtypes implement the IS-A relationship and do not break the LSP.*

For instance, since `List` models a container where any number of elements of the same type can be added, we should not subtype `BoundedList` from it. The latter imposes a bound on the number of elements that it can have; hence instances of it cannot substitute `List`'s instances. This use of subtyping would break the SLP. Examining the contract for `add` in `BoundedList`, it requires that the list indeed must have space to add the element; while for a List there is no such requirement, as it implicitly grows as needed. Hence the pre-condition has been strengthen in `BoundedList`. Thus subclassing it from `List` is not correct. Likewise `OrderedList` should not be a subtype of `List`, as its `add` method places the element in order, while the `add` method in `List`, appends the element to the end of the list. Hence in `OrderedList` the post-condition of the `add` method has been weakened as there is no guarantee of the position of the element added.

• *Document use of composites for inheritance*

Recall that interface implementations for containers follows the Narrow Interface Principle which gives rise to implementations making self-calls: composite methods make use of primitives. From the point of view of performance, these self-calls must be documented for inheritance, as implementation of primitives may impact the performance of the composites. As example in `List, indexOf` is a composite method written in terms of `get`, which is very efficient for array-based implementations. For linked-based, `get` linear performance affects `indexOf`, which should be reimplemented without making use of it.

VALIDATING CLIENTS' CONTRACT

We can initially 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 client a helpful error message 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 the negatives of using checked exceptions:

1. Checked exceptions must be part of the method specification for the client and therefore make code evolution harder to attain when the exceptions need to be changed.

2. Checked exceptions affects client code as client will need to explicitly provide try-catch blocks.

Use of unchecked exceptions when warranted eliminate these two issues and the error messages generated are sufficient.

EXCEPTIONS IN SERVER'S CODE

In the implementation of a feature, the server may need to deal with exceptions. The main issue to keep in mind in this situation is that of exceptions 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 temporary variables which will be used to update corresponding instance variables after all temporaries have been successfully computed.

TESTING THE LIBRARY

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

• *Develop unit tests for regression and automation.*

Unit testing have these two characteristics. With respect to regression, test suites provide a written programmed record of the expected behavior of the API, to be used during its maintenance and evolution. Unit test suites allow for the tests results to be verified automatically rather than manually. Hence, testing

1. Provides a mechanisms to *write* to the API. This in turn ca be used to test API completeness and usability.

2. The test harness is part of the API specification for use during maintenance and evolution.

GENERALIZING TO API DESIGN

The API's presented in CS2, designed and implemented using the best practices and patterns listed, share an overall set of characteristics that must be kept in mind in the development of API's. These characteristics include:

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

CLASSROOM EXPERIENCE

The author has been using this guide for several years while teaching CS2. Several of these principles and patterns are given in [2, 4]. The guide is presented in a spiral approach as the semester progresses. Towards the end of introduction to linear containers a review of most of the guide's principles and best-practices is given. Java's Collections API is also examined to provide more examples of the principles and patterns presented in the guide.

Students are given homework to try their hands on API design, using containers such as `Bag`, or any of several flavors of lists. Although a few students usually struggle with design, the main difficulty the author has always identified is in feature naming and in the documentation of the interface as well as the correctness of features' contracts. When it comes to implementations, again proper documentation is a major issue. In particular, the identification and writing of loop invariants.

CONCLUSION

Teaching CS2 can be a very illuminating and richer experience for the students when the containers are presented under the light of designing and implementing APIs.

The set of guidelines for API design presented provide an initial and fundamental set of principles and best-practices that will enrich the software development experiences we can provide to students. These guidelines will form a fundamental set to build on in later courses such as Software Engineering.

BIBLIOGRAPHY

[**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**] Meyer, B. *Object Oriented Software Construction*. 2nd. Ed. Prentice Hall. 1997.

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

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