STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume LVII, Number 4, 2012

ISSUES IN COLLECTIONS FRAMEWORK DESIGN

VIRGINIA NICULESCU, DANA LUPŞA, AND RADU LUPŞA

Abstract. A good framework/library can reduce the cost of developing an application. This study is an exploration of issues related to designing collections frameworks by analyzing the existing approaches and by emphasizing the fundamentals and the main desiderata of such developments. The corresponding theoretical concepts are analyzed, and for defining data structures an approach based on properties is discussed. Also, some design leading questions are specified in order to emphasize possible new development approaches.

1. Introduction

In the imperative programming setting data structures represent an old and a very important issue. So, different libraries and frameworks have been built in time, based on different programming paradigms.

Initially, the focus was on the structure of the data and on different strategies used for their representation into the memory. The behavior of such a structure was not strictly defined since anyway, the encapsulation of the data with the operations was not yet possible.

By introducing the concept of abstract data type [2], data structures were defined in a more accurate and formal way, by introducing well defined types. A step forward has been done on this subject with object oriented programming (OOP) - a higher order of abstraction being achieved.

Based on OOP we may not only define generic data structures by using type or parametric polymorphism, but also we can separate the definitions from the implementations by using interfaces [9]. Design patterns ([5], [7], [8]) moved things forward, and introduced more flexibility and reusability for data structures.

Genericity is another important issue related to the field of data structures. Any kind of data structure for a collection is formed by a number of elements which are usually of the same type. A specific collection has properties and behaviour which are not dependent on the type of its constitutive elements. So, generally, we may consider them as shape-dependent structures. When properties such as sorting are introduced they could became value-dependent structures.

Received by the editors: September 12, 2012.

2000 *Mathematics Subject Classification.* 68P05.

1998 *CR Categories and Descriptors.* E.1 [**Data**]: Data Structures; E.2 [**Data**]: Data Storage Representation .

Key words and phrases. data structures, collections frameworks, genericity, representation.

1.1. **Motivation and organization of this paper.** Our purpose is to investigate the possibility to create a good framework/library for working with collection data structures.

It is well known that a good framework can reduce the cost of developing an application by an order of magnitude because it lets you reuse both design and code. To consider the problem of software reuse in the moment of the design is not an easy task. A nice description of what this means is made in [10]: "Developing reusable frameworks cannot occur by simply setting down and thinking about the problem domain. No one has the insight to come up with the proper abstractions. Domain experts won't understand how to codify the abstractions that they have in their heads, and programmers won't understand the domain well enough to derive the abstractions. In fact, often there are abstractions that do not become apparent until a framework has been reused."

Collection class libraries have been criticized as being too unwieldy, too inflexible and generally difficult to use [16]. It has been pointed out, for example, that to provide for future flexibility the introduction of many incrementally different types is needed, but huge hierarchies are hard to understand and to use [11].

On short, a good framework is one that makes the programmers job easier and programs better [16]. Much of the engineering effort should go into the design of the class and the type system [11]. A clean, good design would keep things easy to implement and easy to use. In order to achieve this, our research investigates basic, fundamental properties of collection.

In order to make a fundamental analysis on the collections design we start with a short presentation of collections (Section 2). Section 3 collects some important research leading questions. We analyze existing solutions in Section 4 and Section 5. In Section 4, we present programming paradigms used by two existing solutions and emphasize their advantages and disadvantages. In Section 5, we examine some existing approaches for designing collections framework and compare them with our ideas.

2. Important issues regarding collections

2.1. **Collection and containers.** The term container in modern computer programming can actually refer to many things. A **container** is an object created to hold other objects. While the term container is used in C++ STL to denote collection containers, Java programmers use the term "collections" rather than "containers". We may consider that the meaning of "container" term is more related to the storage, and the term "collection" is more related to the represented concept.

In this paper, we are going to use the term collection. Exceptions are situations when we are presenting concepts or terminology used by C++ STL.

Java documentation [1] says that **a collection** is sometimes called a container and it is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data.

Collection	*Alternate names*	*Remarks*
Bag	multiset, collection	admit duplicate elements
Set		no duplicate elements
Sequence	list	elements are arranged in a strict linear order; order information has nothing in common with the elements themselves; it is provided as separate information within operations (when needed)
Stack		**specific:** insertions/extractions are made following a fixed (predefined) strategy: Last In First Out (LIFO);
Queue		**specific:** insertions/extractions are made following a fixed (predefined) strategy: First In First Out (FIFO);
DeQueue		**specific:** insertions/extractions can be made *to/from both ends*
Map	unique associative container, associative array, dictionary	elements are of type (key, value) ; keys are unique
Multimap	multiple associative container	elements are of type (key, value); keys are not unique

Table 1. The most used collections

We present in table 1 a list of most important collections that are known as widely used in programs, and that are provided by most collections frameworks. Collections alternate names and some important remarks are also presented in the table.

A study of collections framework design should pay special attention to its conceptual and logical foundations. A theoretical study of general collection properties is investigated in relation with STL [17] in the next section.

2.2. **Collection container properties.** A property is a feature that could be added to a data structure and then could be removed; it is something that fundamentally characterized the data structure.

Starting from general collection container concepts presented in [17], we make an inventory with the main properties that can make the difference between containers.

- unordered - sequence
- multiple - unique
- simple - pair
- non-associative - associative

The first property can be considered as default. These are unordered, multiple, simple, non-associative. A bag is a collection defined by using only default properties.

We should note that associative property means that we have a pair collection and the first element in the pair is the key - that have special properties associated with it. For example, associative containers are supposed to offer fast access to elements based on keys. That imposes restrictions over the possible choices to implement the collection.

Sorted is a property can be added to many collections but, in this way, they became value-dependent structures. In this study, we are not going to include it, since we restrain the analysis only to shape-dependent structures.

Collection	*sequence*	*unique*	*key-ed (associative)*
Bag	-	-	-
Set	-	yes	-
Sequence	yes	-	-
Stack	used for operation specification	-	-
Queue	used for operation specification	-	-
DeQueue	used for operation specification	-	-
Map	-	yes	yes
Multimap	-	-	yes

Table 2. Collections and their properties

2.3. **Building collections by their properties.** In this section we are going to present collections mainly based on the properties enumerated before.

We remarked that associative property is discussed only when we have a pair collection. Associative collections support efficient retrieval of elements (values) based on keys. That is why we are going to use only one property that we name it *key-ed* instead of having pair and associative properties.

Table 2 defines the basic collections based on the considered properties.

Still, we may observe that a *set* can be also defined as unordered, unique and key-ed collection. That is: the element is its key, no value is used, and we ask for efficient retrieval of elements based on keys.

Note that some of the basic collections imply specific restrictions regarding operations. Examples are adding and extracting elements for stacks, queues, and deQueues. Another example is the element access operation for key-ed collections. They are not expressed by using properties. Starting from this observation we may consider an extension of the properties set by including stack or queue element access types as properties. But this kind of approach would lead not only to an exponential growth of number of properties combination, but also will add, as properties, restrictions that are usually defined at operations level.

3. What to consider when designing collections frameworks

In order to make a fundamental analysis on a collections framework design we may start by putting some important research leading questions:

- Which are the fundamentals that should be considered when designing collections framework? (In this paper, section 2 presents shortly collections concepts and their properties).
- Is a hierarchical approach appropriate for developing collections framework? (see section 4.1) Are the types that correspond to the main collections in relations of subtyping kind only? If not, what other relations should be considered?
- What are the solutions to assure genericity? Which is best? (see section 4.2)
- How certain levels of abstractness influence collections framework properties? (See section 4.3.) Which has to be the leading focus: the needed behavior or the performance?

Within each of the above questions, we also have in mind the next question:

- What are the important lessons to be learn from others experience?

4. Collections framework design: a short overview of two existing solutions

In this section we are going to analyze STL versus Java Collection Framework (JCF) ([12], [15]). Both JCF and C++ STL provide collections frameworks. There are a number of differences, some of them stem from the language features and philosophy. Some other differences are simply design choices and we are not going to present them here.

4.1. **Interfaces versus Concepts.**

To define collections, Java [1] uses a clean separation between interfaces: `Collection`, `List`, `Set`, `SortedSet`, `Map`, `SortedMap` and implementations: `ArrayList`, `LinkedList`, `HashSet`, `TreeSet`, `HashMap`, `TreeMap`.

On the other hand, STL containers are all concrete classes, with no interface-implementation hierarchy, in order to make them more efficient.

4.1.1. *Problems defining a hierarchy.*

Java tried to classify collections based on their properties and behaviour. But things are not straight forward and Java classifications suffered small modifications over time. For example, `Vector` became `ArrayList`, while `Vector` stayed only for compatibility and is deprecated in the current version.

Note that, even if Java language is based on only one hierarchy of classes (derived from `Object`), in JCF there is not only one class hierarchy: `Map` is not a `Collection`. In the same time, STL `map` and `unordered_map` are truly containers of key-value pair (`std::pair<K,V>`).

In [16] it is specified that Java `Map` doesn't extend `Collection` by design. Collection could be made to extend Map, but forcing this, it leads to an unnatural interface. If a `Map` would be a `Collection`, its elements should be key-value pairs, but this provides a very limited `Map` abstraction. There are two important problems: accessing a

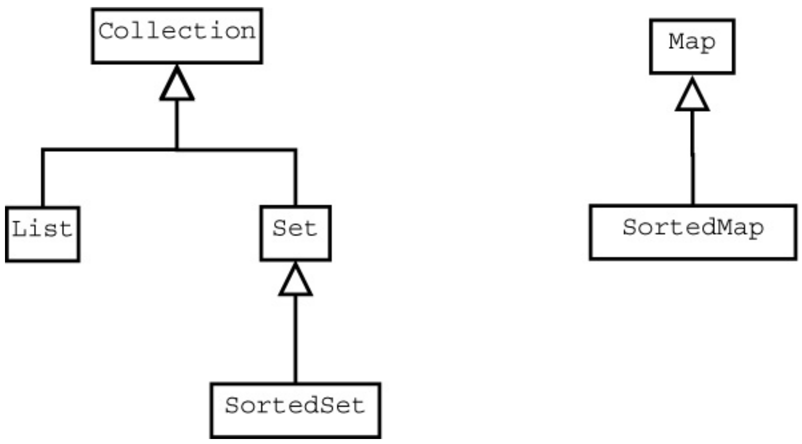

FIGURE 1. Java Collections Framework major interfaces.

value for a given key, and deletion of an entry for a given key, without knowing the value it maps to.

4.1.2. *Uniform naming scheme.* STL's containers have a uniform naming scheme, with identical names for functions with identical roles. Note that a set of member function names and arguments, together with their semantic, is called a *concept*, and a class that actually implements those functions is said to *implement the concept*. As an example, an STL container is required to have a pair of functions, `begin()` and `end()`, returning iterators to the first element and, respectively, just after the last element. This implicitly implies that there is an order between elements, so that we have a first and a last element. Although the concepts are not part of the language, but rather a convention between programmers, and there cannot exist a run-time polymorphism based on concepts, concepts allow a **compile-time polymorphism**: a function template can be parameterized on a type that implements the container concept; thus, that function can call `begin()` and `end()` on the argument of container type.

4.2. **Genericity.** Parametric genericity, initially represented in object oriented setting by source code reuse mechanism as C++ templates, became more and more popular and other object oriented languages as Java and C# enhanced their new versions with mechanisms that offer parametric types.

In C++ the template mechanism allows us not to create a single class, but to specify only once the pattern for creation of some classes that are different only by the type of some parameters. The template mechanism allows a very high degree of flexibility, but it is considered in some literature not a really parametric polymorphism mechanism since for each actual parameter a new class is created.

The mechanism which was included in Java since JDK 1.5 is considered more efficient since just one class is created for each parameterized class. Also, the mechanism

of parameterized Java classes allows bounded polymorphism – the specification of a certain behavior of parameters by interface implementation.
A similar mechanism is implemented in C# too.

A comparison between C++ templates and the extensions for generics of the C# and Java languages based on their suitability for scientific computing was done in [6]. These measurements suggest that both Java(TM) and C# generics introduce only little run time overhead when compared with non-parameterized implementations. With respect to scientific application, C# generics have the advantage of allowing value types (including builtin types) as parameters of generic classes and methods. Also, in [3] there is study about the performance of generics for scientific computing in various programming languages, based on a standard numeric benchmarks. The conclusion was that in current implementations of generics must be improved before they are used for efficiency-critical scientific applications.

The C++ templates mechanism is considered for implementing parametric polymorphism based on a "heterogeneous" approach. The "heterogeneous" approach constructs a special class for each different use of the type parameters. The compiled code is fast, but the object code could become bulky since we have many different versions of each class.

Java generics implement "homogenous" approach of the parametric polymorphism. Since is based on "type erasing" we have strong restrictions, and maybe the most important is represented by the impossibility of specifying static members for the generics.

4.3. Level of Abstractness.

From the previous comparison we may conclude that Java collections design is based on the corresponding abstract definitions (Abstract Data Types), when STL design has more utilitarian focus. So we may conclude that the abstraction is higher for first one. If the goal would be to establish to which extend these approaches are the most appropriate and useful for the common developers, then the task would be very difficult. The well trained developers may consider that the STL approach offers rapidity and better safety. On the other hand JCF is much easier to learn and deal with.

A low level of programming focuses on performance and usually doesn't use an intermediary tool as a framework. A framework design should be leaded by the behavior but in the same time it should not ignore the performance and safety. The choice should consider the programmer needs.

5. Some other collections frameworks approaches

There are others collections frameworks as well. The Guava project contains several of Google's core libraries that they rely on in their Java-based projects: collections, caching, primitives support, concurrency libraries, common annotations, string processing, I/O, and so forth. Each of these tools really do get used every day by Googlers, in production services [14].

There are also extensions of existing frameworks. For example, utilities available in java.util.Collections (from JCF) are extended by fastutil by providing type-specific maps, sets, lists and queues with a small memory footprint and fast access and insertion. Fastutil came up as a necessity during the development of a web crawler, as they needed to manage structures with dozens of millions of items very efficiently [13].

Another way to model existing collections, is to reconsider the way they are defined. A collections framework based on set theory is Yet Another Collections Library (YACL) [18]. The project YACL consider a model in which Function extends Relation extends Set. Bags and Sequences extend Function. Hence a Function (equivalent to Sun's Map class/interface) is a type of Set. They build a theoretically sound collections library on the top of JCF Set: Set implements java.util.Set. All classes can be constructed from java.util collections and maps (where applicable).

In [4], the aim is to define a collection library for Java which uses multiple inheritance to offer a flexible framework for defining collection types rather than providing a complex exhaustive set of particular collection classes. They identify a small number of software engineering concepts relevant to the design of libraries of collections. They distinguish three basic orthogonal semantic properties of collections: ordering of elements, definition and handling of duplicate elements, definition of keys for efficient search. They use the next properties : order (ordered, sorted, userOrdered), duplicates (duplFree, duplIgnore, duplError) and search (searchable) that are intended to extend JCF. Particular collection types should be built by using derivation and by specifying their properties in terms of these basic types. For example, the interface type 'Bag' can be defined as:

```
interface Bag[ELEMENT] extends Collection[ELEMENT] {}
```

and the type 'List' as:

```
interface List[ELEMENT]
extends UserOrdered[ELEMENT], Bag[ELEMENT] {}
```

Because we want to get an clean design, our idea is not to extend other collections, but rather to redefine collections themselves in terms of their properties. We identified some concepts relevant to the design of collections framework. Among them, there are some that are present in [4]: UserOrdered as a property that we named Sequence (in opposition with unordered), Duplicates (that is in opposition with unique, and we named it multiple). The definition of List in [4] corresponds to our definition, but we avoid using bag and collections in the same time. We also tried to avoid using any reference to bag when specifying a list. We also considered one new property in our classification (key-ed) although it was not our purpose to make an inventory of some new and not yet existing properties, as we based our approach on STL concepts [17].

We reconsider the way collections are defined by considering small number of software engineering concepts relevant to the design of libraries of collections.

6. Conclusions

Since in the literature there are different classifications and definitions for the types corresponding to different collections, the existing implementation solutions - frameworks - are also very different. The authors of this paper are conscious of, and try to overview different initiatives. In this paper, an inventory of theoretical concepts is made and existing collections frameworks are compared. We have specified some design leading questions and for each of them we have done an analysis in specific sections of this paper. Their role is to emphasize possible new development approaches.

The goal of this paper is to to investigate possible approaches to the design a good framework for working with collection data structures, by analyzing the existing solutions and by emphasizing the fundamentals and the main desiderata of such developments.

References

[1] J. Bloch, *The Java Tutorial. Trail: Collections* `http://docs.oracle.com/javase/tutorial/collections/`.

[2] L. Cardelli, P. Wegner, *On understanding types, data abstraction, and polymorphism* ACM COMPUTING SURVEYS, (1985).

[3] L. Dragan, S.M. Watt, *Performance Analysis of Generics in Scientific Computing*, Proceedings of Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'05), 2005, pp.93-100.

[4] M. Evered, G. Menger, J. L. Keedy, A. Schmolitzky, *A Useable Collection Framework for Java*, 16th IASTED Intl. Conf. on Applied Informatics, Garmisch Partenkirchen, 1998.

[5] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley, 1994.

[6] J. Gerlach, J. Kneis, *Generic programming for scientific computing in C++, Java, and C#.* Lecture Notes in Computer Science. Proceedings of International Workshop on Advanced Parallel Processing Technologies (APPT) Xiamen, China, 2003, pp.301-310.

[7] D. Nguyen, *Design Patterns for Data Structures*, SIGCSE Bulletin, 30, 1, March 1998, 336-340.

[8] V. Niculescu, G. Czibula, *Fundamental Data Structures and Algorithms. An Object-Oriented Perspective*, Casa Cartii de Stiinta, 2009 (in Romanian).

[9] V. Niculescu, *Storage Independence in Data Structures Implementation*, Studia Universitatis "Babes-Bolyai", Informatica, Special Issue, LVI(3), pp. 21-26, 2011.

[10] D. Roberts, R. Johnson, *Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks*, in Proceedings of the Third Conference on Pattern Languages and Programming, 1996, `http://st-www.cs.illinois.edu/users/droberts/evolve.html`

[11] C. Szypersky, S. Omohundro, S. Murer *Engineering a Programming Language: The Type and Class System of Sather*, in Programming Languages and System Architectures, ed. J. Gutknecht, Springer-Verlag, pp. 208-227, 1993.

[12] B. Stroustrup, M. Ellis, *The Annotated Reference C++ Manual*, Addison-Wesley, 1994.

[13] *fastutil: Fast & compact type-specific collections for Java*, `http://fastutil.dsi.unimi.it/`

[14] *Guava project*, `https://code.google.com/p/guava-libraries/`

[15] *Generic Java*, `http://download.oracle.com/javase/1.5.0/docs/guide/language/generics.html`

[16] *Java.The Collections Framework*, `http://download.oracle.com/javase/1.5.0/docs/guide/collections/`

[17] *STL Programmer's Guide*, `http://www.sgi.com/tech/stl/index.html`

[18] *YACL - Yet Another Collections Library*, `http://sourceforge.net/projects/zedlib`

Department of Computer Science, Babeş-Bolyai University, Kogalniceanu 1, 400084, Cluj-Napoca, Romania

E-mail address: {`vniculescu, dana, rlupsa`}`@cs.ubbcluj.ro`