
C++ Programming Exercises

Wolfgang Pree

C. Doppler Laboratory for Software Engineering
University of Linz

Motivation

What you will learn

Collection classes

Abstract root class Object

Abstract class Collection

Usage and extension of the library

Change propagation

Concept

Monitoring collections

Motivation

What you'll learn

First hands-on experience with a small class library as playground:

- use predefined classes without changes
- define a new class (→ programming by difference)
- experience the advantages of abstract classes
- do adaptations through object composition

Collection classes

Collection classes

Collections manage objects of static type **Object**:

ObjList: uses a double-linked list for managing the contained objects

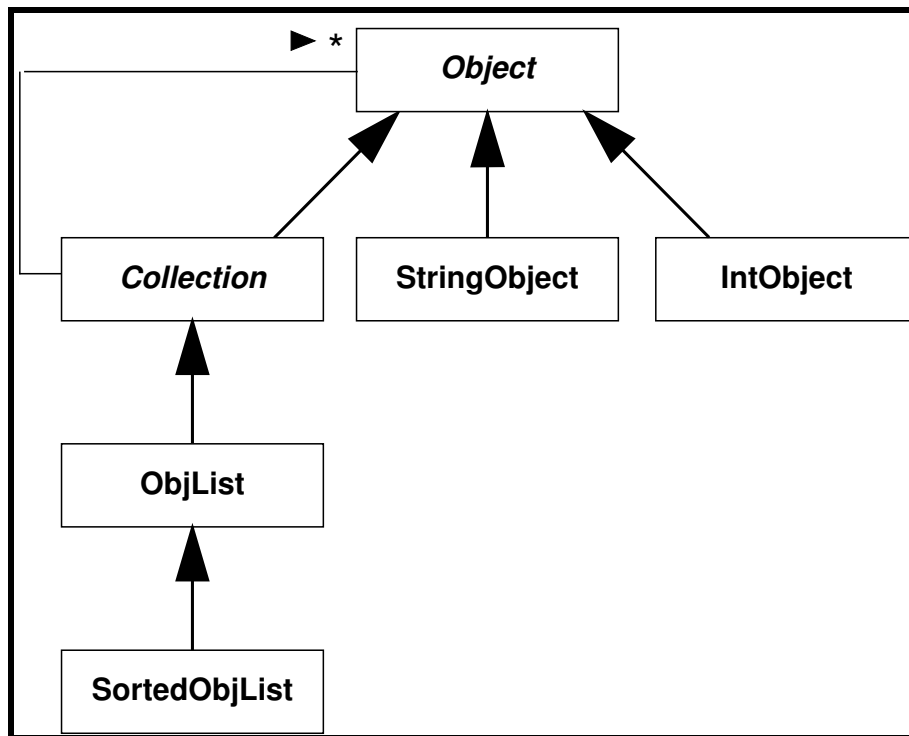
SortedObjList: sorts contained objects according to an object comparison

ObjArray: uses an array for managing the contained objects

Set: contains a particular object only once; an object that is equal to an already contained one is not inserted

Collection classes

The following classes form the basis of our exercises:



Class **Object**:

A tree-structured (->single inheritance) class hierarchy has a root class that is usually called **Object**.

Collection classes

It is pretty hard to find a protocol which is useful in all future subclasses. The Xerox PARC did pioneering work when defining the first Smalltalk library.

Examples of useful methods are

- methods for comparing an object with another one
- a method to check the dynamic type of an object

This could be implemented in C++ in the following way:

```
class Object {  
    // ...  
public:  
    // ...  
    virtual bool IsEqual(Object *anotherObject) =0;  
    virtual int Compare(Object *anotherObject) =0;  
  
    bool IsKindOf(char *className);  
  
    // ...  
};
```


Collection classes

Class **Object** is a typical example of an abstract class. For example, methods for comparing objects cannot be implemented at this level.

A subclass **StringObject** that represents character strings overrides the **IsEqual** method in the following way:

```
bool StringObject::IsEqual(Object *op)    {  
    return ( op->IsKindOf("StringObject")  &&  
            strcmp(cont,  
                    ((StringObject*)op)->cont) == 0  
            );  
}
```

Class interface of **StringObject**:

```
class StringObject: public Object {  
public:  
    char *cont;  
    // ...  
};
```

Collection classes

Remarks:

- the cast `(StringObject *)op` is necessary since the compiler knows only the static type of `op`, i.e., `Object *`

The compiler would report an error since class `Object` does not have the instance variable `cont`

- due to the dynamic typecheck we did by means of `IsKindOf` the cast is safe

Collection classes

Collection represents another good example of an abstract class. It factors out commonalities of object collections.

Object collections have at least the following properties in common: They can

- add an object
- remove an object
- enumerate contained objects (iteration)
- check whether an object is contained
- add all objects contained in another object collection
- remove all objects contained in another object collection

Collection classes

Corresponding C++ definition of class Collection:

```
class Collection: public Object {  
public:  
    Collection();  
    virtual void Add(Object *) =0;  
    virtual void Remove(Object *) =0;  
    virtual Sequence *Iterator() =0;  
    virtual bool Contains(Object *);  
    virtual int Size();  
    virtual void AddAll(Collection *);  
    virtual void RemoveAll(Collection *);  
    // ...  
};
```

Collection classes

Some methods of **Collection** cannot be implemented at this level (e.g., **Add** and **Remove**).

Nevertheless, some methods can be implemented based on the (abstract) protocol of **Collection** and **Object**:

The method **Contains** is implemented by enumerating all objects of the particular collection. Each object of the collection is then compared with the object that is passed as parameter to **Contains**.

```
bool Collection::Contains(Object *anObj) {  
    for each obj in Collection  
        if (obj->IsEqual(anObj))  
            return TRUE;  
    return FALSE;  
}
```

The methods **Size**, **AddAll** and **RemoveAll** are implemented in an analogous way.

Collection classes

Exercise 1

Use class `ObjList` and put some `StringObject` instances into an `ObjList` instance.

Write the code into the program's `main()` function.

Collection classes

Iterating over collections:

We have to discuss the method

Sequence ***Collection::Iterator()**

It returns a pointer to an object of class Sequence:

```
class Sequence {  
public:  
    Sequence() { }  
    virtual Object *Next()= 0;  
};
```

The idea is that each time the method **Next()** of a **Sequence** object is called, the next object of the collection is returned.

If the end of the collection is reached 0 is returned.

Collection classes

Sample code fragment for iterating over an `ObjList` instance
(The variable `oList` points to that `ObjList` instance):

```
{
    Sequence *iter;
    Object *op;

    iter= oList->Iterator();

    while ( (op= iter->Next()) != 0 )
        // do something with 'op'

    delete iter;
}
```

Exercise 2

Add code to the program's `main()` function of Exercise 1 that iterates over the `ObjList` instance and prints the corresponding string for each contained `StringObject` instance.

Collection classes

Remarks:

Subclasses of **Collection** override the **Iterator** method and return an appropriate instance of a subclass of **Sequence**.

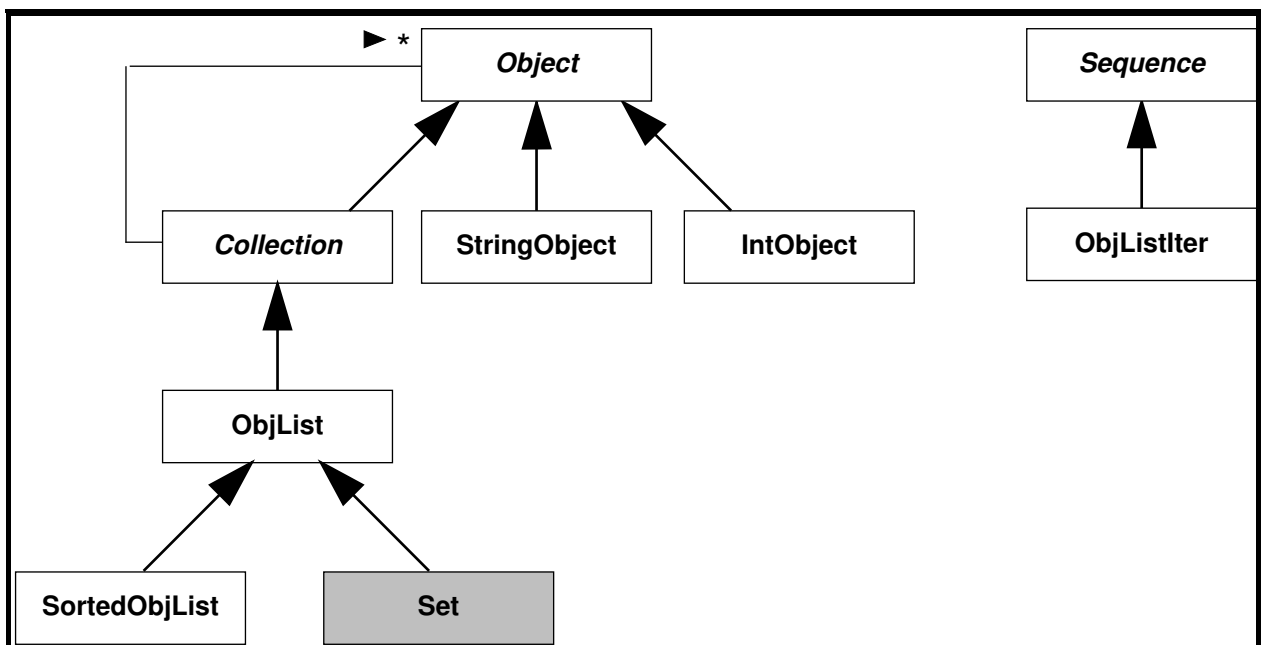
Why is a `Next()` method in class `Collection` not sufficient?

Collection classes

Exercise 3

Define a class **Set** as subclass of **ObjList**. Override method **Add** so that equal objects are only inserted once.

Test class **Set** in the program's **main()** function.



Change propagation

Change propagation

Change propagation is another mechanism that can already be implemented in class `Object`.

It helps to **coordinate activities of objects which depend on one another**:

An object A is registered by another object B to be dependent on B. If B changes, A has to be notified about this.

Change propagation

```
class Object {
    ObjList *dependents;
public:
    ...
    void AddDependent(Object *op) {
        dependents->Add(op);
    }
    void RemoveDependent(Object *op) {
        dependents->Remove(op);
    }
    void Changed() {
        for each obj in dependents
        obj->DoUpdate(this);
    }
    virtual void DoUpdate(Object *)= 0;
    ...
};
```

Change propagation

A few examples where change propagation can be applied:

OO diagram editor:

Lines connecting class/object symbols depend on these graphic representations (if a class/object is moved all lines connected to it have to be redrawn, too).

Process control:

The display of the current system state in a GUI has to be updated when external factors (e.g., temperature of a ladle furnace) change.

Change propagation is a generalization of the MVC (Model-View-Controller) framework, which will be discussed later.

Change propagation

Exercise 4

Define a class `CollMonitor` whose instances can be attached to any `Collection` object. A `CollMonitor` instance should report (e.g., by printing) the particular number of objects contained in a collection every time an item is added/removed from a collection.

Test class `CollMonitor` in the program's `main()` function.

