CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 23
April 21, 2016

Linear Container Design

Overview of linear container example

We've seen three closely related designs for linear containers:

- `19-Virtual`
- `22a-Multiple`
- `22b-Multiple-template`

Common to all three examples is the use of a linked list of `Cell` objects to implement various kinds of containers store data objects of type `Exam`.

Differences in functionality

`19-Virtual` implements a FIFO `class Stack` and a LIFO `class Queue` of unordered `Exam`s.

`22a-Multiple` and `22b-Multiple-template` add a `key()` method to `Exam` and implement two new data structures that make use of the key:

- `class List` is an unordered set of elements, where `push()` inserts an element at the front of the list and `pop()` prompts the user for the key of the element to be removed.
- `class PQueue` maintains its elements in a sorted list, where `push()` inserts an element into the middle of the list so as to maintain the elements in descending order, based on an underlying ordering on keys, and `pop()` removes the front (largest) element.

Class structure

The class structure of the three implementations are similar as are the main programs. All have have the following classes:

- `class Item` are the objects that live in the containers.
- `class Exam` are the application-specific user objects.
- `class Container` is the abstract interface for all containers.
- `class Linear` is the common code for the containers.

In `19-Virtual`, `Item` and `Exam` are synonymous.

In `22a-Multiple` and `22b-Multiple-template`, `Item` is derived from `Exam` and extends the comparison operators `<` and `==`.

Template structure

`22b-Multiple-template` adds a template parameter `class T` to represent the application-specific user object type.

In the sample application, `main()` instantiates the template as `List<Item>` and `PQueue<Item>`.

As before, `Item` is derived from `Exam`, so to create containers for a new user type `MyData` would require rewriting `Item` appropriately.

Further extensions

Can we make `Item` a template class, e.g.,

```
template <class T>
class Item : public T, public Ordered<KeyType> { ... };
```

Then the user with application-specific class `MyData` could just use `Item<MyData>` wherever `22b-Multiple-template` uses `Item`.

Two problems

Two problems must be overcome to make this approach work:

1. Type `KeyType` appropriate to `MyData` must be defined somewhere.
2. `Item` contains an `Exam`-specific constructor
   `Item(const char* init, int sc) :  Exam(init, sc){}`
   that would have to be eliminated in favor of something that would work in general.

Defining `KeyType`

Problem 1 can be solved by putting a `typedef` for `KeyType` into `class MyClass`. This type name can be used as follows:

```
template <class T>
class Item : public T, public Ordered<typename T::KeyType>
{ ... };
```

Constructing the data elements

Problem 2 is not so easily solved. Some possibilities:

- Have the user construct a `MyData` object, then have `Item` use `MyData`'s copy constructor, e.g.,

  ```
  Item(const T& data) : T(data) {}
  ```

  This adds a time penalty and a requirement that `T` be copyable.
- Instead of deriving `Item<T>` from `T`, compose a `T*` in `Item<T>`, and initialize it with a generic `Item` constructor, e.g.,

  ```
  T* base;
  Item(T* dt) : base(dt) {}
  ```

23a-Multiple-template

This second idea is implemented in example `23a-Multiple-template`.

A consequence of this design is that `main()` now mentions only the user type (`Exam`), not `Item`, effectively isolating the user interface from the underlying implementation, e.g.,

```
List<Exam> L;
L.put( new Exam("Ned", 29) );
L.put( new Exam("Leo", 37) );
L.put( new Exam("Max", 18) );
```

Storage management

A basic design question has to do with ownership of dynamic storage.

In STL, the user retains ownership of arguments, and a container such as `vector` manages copies of the elements.

In these linear container examples, ownership transfers to the container.

- The user creates a element using `new` and passes a pointer to the `put()` function.
- `pop()` returns ownership of the object to the user, who is then responsible for its eventual deletion.
- The container is responsible for deleting any objects it still contains when it goes away.

STL and Polymorphism

Derivation from STL containers

Common wisdom on the internet says not to inherit from STL containers.

For example,
http://en.wikipedia.org/wiki/Standard_Template_Library says,

> *"STL containers are not intended to be used as base classes (their destructors are deliberately non-virtual); deriving from a container is a common mistake."*

This reflects Rule 35 of Sutter and Alexandrescu,

> *"Avoid inheriting from classes that were not designed to be base classes."*

Replacing authority with understanding

C++ is a complicated and powerful language.

Some constructs such as classes are used for several different purposes.

What is appropriate in one context may not be in another.

Simple rules will not make you a good C++ programmer. Thought, understanding, and experience will.

Two kinds of derivation

C++ supports two distinct kinds of derivation:

- Simple derivation.
- Polymorphic derivation.

```
class A { ... };
class B : public A { ... };
```

We say `B` is **derived** from `A`, and `B` **inherits** members from `A`.

Each `B` object has an `A` object embedded within it.

The derivation is **simple** if no members of `A` are `virtual`; otherwise it is `polymorphic`.

How are they the same?

With both kinds of derivation, a function of the base class A can be **overridden** by a function in B.

In both cases, one can create and delete objects of class B.

Both A's and B's destructor are called when a B object is deleted.

```
#include <iostream>
class A { public:
  ~A() { std::cout << "A's destructor called" << std::endl; }
};
class B: public A { public:
  ~B() { std::cout << "B's destructor called" << std::endl; }
};
int main() { B bobj; }
```

Output:
```
B's destructor called
A's destructor called
```

What is simple derivation good for?

Some uses for simple derivation.

- Code sharing. A common base can be extended in different directions through derivation.
- Creating a new API to system resources (e.g., `14-StopWatch` demo).
- Increasing modularity through layering.

With simple derivation, the derived class is the public interface.

Often `protected` or `private` derivation is used to hide the base class from the users of the derived class.

What are the problems with simple derivation?

- Several objects derived from the same base type have little in common except for the embedded base type object in each.
- A base type pointer can only access the embedded base type object. The rest of the derived object is present but invisible. This is called **slicing**, where the derived part is conceptually “sliced off”.

What is polymorphic derivation good for?

- Polymorphic derivation allows for variability among objects with a common interface.
- The base class (possibly pure abstract) defines the interface.
- Each derived class defines a variant or implementation of the interface.

Some uses for polymorphic derivation.

- Heterogeneous containers. Example: An array of different kinds of employees.
- A mechanism whereby old code can call new code. By deriving from a predefined interface, existing functions that call virtual functions of the base class end up invoking new user-provided code.

What are the problems of polymorphic derivation?

Every polymorphic base class (containing even one virtual function) adds a runtime **type tag** to each instance.

This costs in both time and space.

- Time: Each call to a virtual function goes through a run-time dispatch table (the **vtable**).
- Space: Each instance of a polymorphic object contains a type tag, which takes extra space.
- Every polymorphic base class should have a `virtual` destructor.

Contrasts between simple and polymorphic derivation

Simple derivation:

- Low cost.
- Extends the base class.
- Derived class is the public interface; base class is a helper.
- Slicing is generally avoided as being not useful.

Polymorphic derivation:

- Higher cost.
- Implements the base class (in possibly multiple ways).
- Base class is the public interface; derived classes are helpers.
- Slicing is encouraged; virtual functions provide access to underlying derived class objects.

Containment as an alternative to simple derivation

Often the same class can be implemented using either containment or derivation.

Derivation:

```
class A { ... f() ... };
class B: public A { ... g() { f() ... } };
```

`A`'s public member functions are inherited by `B`.

Containment:

```
class A { ... f() ... };
class B { private: A a; ... g() { a.f() ... }
   public: f() { return a.f(); } };
```

Access to `A`'s public member functions requires a “pass-through” function for delegation.

Argument for containment

Containment is a more distant relationship than derivation.

Less coupling between classes is safer and less error-prone.

Using containment, derived class is explicit about what is exported.

For more info, see http://www.gotw.ca/publications/mill06.htm.

STL container as a base class

We apply these concepts to STL base classes.

Base classes are simple, not polymorphic (no virtual functions, no virtual destructor).

This means that they should only be used with simple derivation. They are not suitable as base classes for polymorphic derivation.

Often containment is preferable, but the large number of member functions they support makes it cumbersome to get the same degree of functionality in the derived class as comes “for free” with derivation.

Can I turn an STL container into a polymorphic base class?

Yes, sort of. Here's the idea.

```
#include <iostream>
#include <vector>
using namespace std;
class MyVectorInt : public vector<int> {
public:
  MyVectorInt() : vector<int>() {}
   virtual ~MyVectorInt() {
      cout << "Base class destructor is called" << endl; }
};
class Derived : public MyVectorInt {
public:
   ~Derived() {
      cout << "Derived destructor is called" << endl; }
};
```

A polymorphic base class

`MyVectorInt` is a polymorphic base class with virtual destructor and can be used as such.

```
int main() {
  MyVectorInt* p;  // a polymorphic pointer
  Derived* obj = new Derived();  // a derived object
  p = obj;         // ok to assign
  delete p;        // ok to delete; destructors called
}
```

Dynamic cast

It is always okay to cast a pointer to a derived class into a pointer to the base class, as in the previous example.

The reverse is only semantically meaningful if the allocated type of the object actually is the type to which it is being cast. In that case, one can use a `dynamic_cast` to effect the conversion.

```
MyVectorInt* p;
Derived* q;
...
q = dynamic_cast<Derived*>(p);
```

`dynamic_cast` returns `NULL` if `p` is pointing to something that does not have dynamic type `Derived*`.

Design Patterns

General OO principles

1. **Encapsulation** Data members should be private. Public accessing functions should be defined only when absolutely necessary. This minimizes the ways in which one class can depend on the representation of another.
2. **Narrow interface** Keep the interface (set of public functions) as simple as possible; include only those functions that are of direct interest to client classes. Utility functions that are used only to implement the interface should be kept private. This minimizes the chance for information to leak out of the class or for a function to be used inappropriately.
3. **Delegation** A class that is called upon to perform a task often delegates that task (or part of it) to one of its members who is an expert.

What is a design pattern?

A pattern has four essential elements.[1]

1. A *pattern name*.
2. The *problem*, which describes when to apply the pattern.
3. The *solution*, which describes the elements, relations, and responsibilities.
4. The *consequences*, which are the results and tradeoffs.

[1]Erich Gamma et al., *Design Patterns*, Addison-Wesley, 1995.)

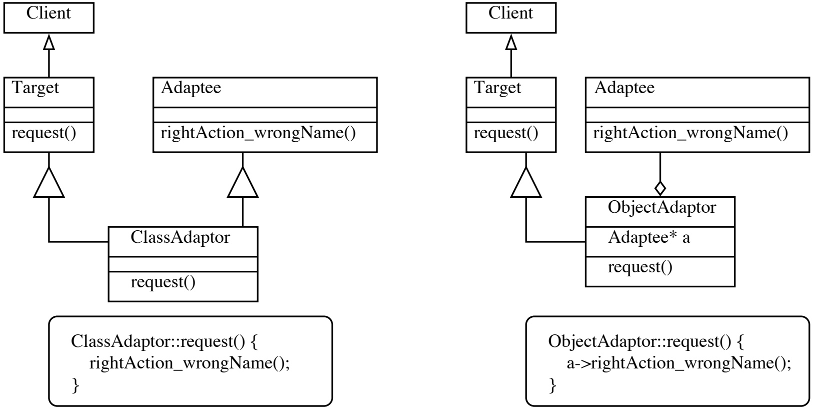
Adaptor pattern

Sometimes a toolkit class is not reusable because its interface does not match the domain-specific interface an application requires.

Solution: Define an adapter class that can add, subtract, or override functionality, where necessary.

Adaptor diagram

There are two ways to do this; on the left is a class adapter, on the right an object adapter.



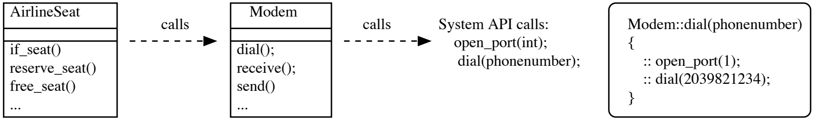
```
ClassAdaptor::request() {
   rightAction_wrongName();
}
```

```
ObjectAdaptor::request() {
   a->rightAction_wrongName();
}
```

Indirection

This pattern is used to decouple the application from the implementation where an implementation depends on the interface of some low-level device.

Goal is to make the application stable, even if the device changes.



Proxy pattern

This pattern is like Indirection, and is used when direct access to a component is not desired or possible.

Solution: Provide a placeholder that represents the inaccessible component to control access to it and interact with it. The placeholder is a local software class. Give it responsibility for communicating with the real component.

Special cases: Device proxy, remote proxy. In Remote Proxy, the system must communicate with an object in another address space.

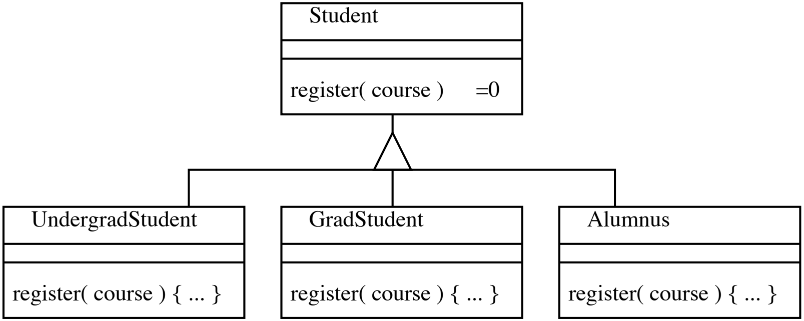
Polymorphism pattern

In an application where the abstraction has more than one implementation, define an abstract base class and one or more subclasses.

Let the subclasses implement the abstract operations.

This decouples the implementation from the abstraction and allows multiple implementations to be introduced, as needed.

Polymorphism diagram


Student
register(course) =0
UndergradStudent
register(course) { ... }
GradStudent
register(course) { ... }
Alumnus
register(course) { ... }

Controller

A controller class takes responsibility for handling a system event.

The controller should coordinate the work that needs to be done and keep track of the state of the interaction. It should delegate all other work to other classes.

Three kinds of controllers

A controller class represents one of the following choices:

- The overall application, business, or organization (facade controller).
- Something in the real world that is active that might be involved in the task (role controller).
  Example: A menu handler.
- An artificial handler of all system events involved in a given use case (use-case controller).
  Example: A retail system might have separate controllers for BuyItem and ReturnItem.

Choose among these according to the number of events to be handled, cohesion and coupling, and to decide how many controllers there should be.

Bridge pattern

Bridge generalizes the Indirection pattern.

It is used when both the application class and the implementation class are (or might be) polymorphic.

Bridge decouples the application from the polymorphic implementation, greatly reducing the amount of code that must be written, and making the application much easier to port to different implementation environments.

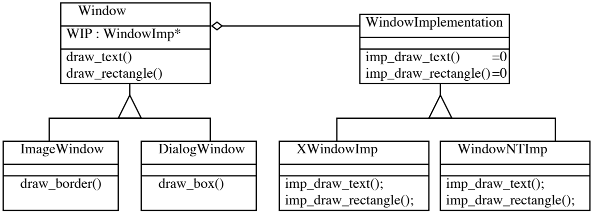
Bridge diagram

In the diagram below, we show that there might be several kinds of windows, and the application might be implemented on two operating systems. The bridge provides a uniform pattern for doing the job.

```
Window::draw_text() {
   WIP->draw_text();
}
```

```
ImageWindow::draw_border() {
   draw_rectangle();
}
```

```
DialogWindow::draw_box() {
   draw_rectangle();
   draw_text();
}
```


Window
WIP : WindowImp*
draw_text()
draw_rectangle()
WindowImplementation
imp_draw_text() =0
imp_draw_rectangle()=0
ImageWindow
draw_border()
DialogWindow
draw_box()
XWindowImp
imp_draw_text();
imp_draw_rectangle();
WindowNTImp
imp_draw_text();
imp_draw_rectangle();