CS2303 In Class Exercises

February 18, 2020

Abstract

Today's goal is to answer some questions from an earlier class, and that you can create (and read) sequence diagrams drawn in the context of object-oriented design, in particular in C++, but also more generally. Beyond this, appreciate the concept of Patterns as used in software design.

Name and WPI email: Kush Shah kshah2@wpi.edu

1 Questions

1.1 Virtual Destructors

Obtained from: `https://www.programmerinterview.com/c-cplusplus/virtual-destructors/`. in C++ a destructor is generally used to deallocate memory and do some other cleanup for a class object and its class members whenever an object is destroyed. Destructors are distinguished by the tilde, the $\sim$ that appears in front of the destructor name. In order to define a virtual destructor, all you have to do is simply add the keyword "virtual" before the tilde symbol.

First, an example that does not use virtual destructors:

```
#include iostream.h
class Base
{
public:
Base(){ cout<<"Constructing Base";}

// this is a destructor:

~Base(){ cout<<"Destroying Base";}
};

class Derive: public Base
{
public:
```

```
Derive(){ cout<<"Constructing Derive";}

~Derive(){ cout<<"Destroying Derive";}
};

void main()
{
Base *basePtr = new Derive();

delete basePtr;
}
```

The output will be:

```
Constructing Base
Constructing Derive
Destroying Base
```

The parent class's constructor gets called before the code in the child class's constructor. On delete, only the parent class's destructor gets called.

An example that does use virtual destructors:

```
class Base
{
public:
Base(){ cout<<"Constructing Base";}

// this is a virtual destructor:
virtual ~Base(){ cout<<"Destroying Base";}
};
```

The output will be:

```
Constructing Base
Constructing Derive
Destroying Derive
Destroying Base
```

Note that the derived class destructor will be called before the base class. So, now you've seen why we need virtual destructors and also how they work. One important design paradigm of class design is that if a class has one or more virtual functions, then that class should also have a virtual destructor.

The reasoning is, the existence of virtual functions is used to infer that there are child classes. If there is a child class, its destructor should be called.

1.2 Can we perform logic on the type in a template class?

Instead of using

```
template <typename T> int compare (const T&, const T&)
```

to compare any two types,
we would use

```
template<size_t N, size_t M>
int compare(const char (&) [N], const char (&)[M]);
```

to handle string literals.

The version that has two non-type template parameters will be called only when we pass a string literal or an array.

If we call compare with character pointers, the first version will be used.

```
const char* p1 = "hi";
const char* p2 = "mom";
compare(p1, p2);   <- calls the first template

compare("hi", "mom") <- calls the template with two nontype parameters
```

The above is not logic within a template, it is the provision of multiple templates. There is an activity called template specialization.

```
template<>
int compare(const char* const & p1, const char* const & p2)
{
return strcmp(p1,p2);
}
```

("const char* const &" means reference to a const pointer to const char.)

Above we are defining a specialization, and so the parameter types we specify here must match those in a previously declared template. The example above is a specialization of

```
template <typename T> int compare(const T&, const T&)
```

We are specializing T* to const char*. When a more specialized and a more generally applicable template are both available, the compiler will choose the more specialized.

Thus a specialization also is not logic within a template, rather, the logic is applied to choose which template.

```
/*
* Room.h
*
*  Created on: Feb 16, 2020
*      Author: Therese
*/

#ifndef ROOM_H_
```

```
#define ROOM_H_

#include <iostream>
using namespace std;
#include <stdbool.h>
#include <typeinfo>

template<typename T>
void printArray(T* array, int count)
{
cout << typeid(array[0]).name() << endl;
//const std::type_info& r2 = typeid(std::printf("%d\n", array[0]));
//cout << "rs is " << r2.name() << endl;
// int x[1] = {3};
// double y[1] = {2.4};
// char z[1] = {'c'};
int typeVal=-1;
if( typeid(array[0]) == typeid(int) )   //type_info& rhs) )
{
typeVal = 1;
}
if( typeid(array[0]) == typeid(double))
{
typeVal = 2;
}
else if (typeid(array[0]) == typeid(float))
{
typeVal = 3;
}
else if (typeid(array[0]) == typeid(char))
{
typeVal = 4;
}
switch(typeVal)
{
case 1:
puts("It's integers.");
break;
case 2:
puts("It's doubles.");
break;
case 3:
puts("It's float.");
break;
case 4:
puts("It's characters.");
```

```
break;
default:
puts("Unexpected value");
}

for(int i = 0; i < count; i++)
{
cout << array[i] << " ";
}
cout << endl;
}

typedef struct
{
int junk;
double stuff;
float moreStuff;
char forFun;

}SomeDataType;

class Room {
public:
Room();
virtual ~Room();
void setHeight(double h);
void setLength(double ln);
void setDepth(double d);
void setChar(char c);
char getChar();
void setSearched(bool s);

bool getSearched();

private:
double height;
double length;
double depth;
bool searched;
SomeDataType x;
};

#endif /* ROOM_H_ */
```

and the output was:

```
i
It's integers.
1 2 3 4 5
d
It's doubles.
1.1 2.2 3.3 4.4 5.5 6.6
c
It's characters.
a b c d e
```

Thus we see in the example above, we can run different code within a template, that depends upon the type of the data with which the template is working.

In 2017, yet another way was introduced, to do deal with logic on types, within a template. This involves "variant"s.

2 Instance of a Class

To understand the idea of an instance of a class, we first consider a class, and then we see what makes an instance of a class a slightly different idea.

We have seen an example of a class, such as Room. It has a .h file and a .cpp file.

```
/*
* Room.h
*
*  Created on: Feb 16, 2020
*      Author: Therese
*/

#ifndef ROOM_H_
#define ROOM_H_

#include <iostream>
using namespace std;

template<typename T>
void printArray(T* array, int count)
{
for(int i = 0; i < count; i++)
{
cout << array[i] << " ";
}
cout << endl;
}
```

```
typedef struct
{
int junk;
double stuff;
float moreStuff;
char forFun;

}SomeDataType;

class Room {
public:
Room();
virtual ~Room();
void setHeight(double h);
void setLength(double ln);
void setDepth(double d);
void setChar(char c);
char getChar();

private:
double height;
double length;
double depth;
SomeDataType x;
};

#endif /* ROOM_H_ */
```

and

```
/*
* Room.cpp
*
*  Created on: Feb 16, 2020
*      Author: Therese
*/

#include "Room.h"

Room::Room()
:height(0), length(0), depth(0),x({1, 2.1, 3.2,'w'})
// TODO Auto-generated constructor stub
{
}
```

```
Room::~Room() {
// TODO Auto-generated destructor stub
cout << "The destructor for Room was called." << endl;
}

void Room::setHeight(double h)
{
height=h;
}
void Room::setLength(double ln)
{
this->length=ln;
}
void Room::setDepth(double d)
{
(*this).depth=d;
}

void Room::setChar(char c)
{
x.forFun=c;
}
char Room::getChar()
{
return x.forFun;
}
```

The class is a form of input to a compiler. At execution time, we can choose to create an instance of the class.

```
Room* rP = new Room();
rP->setChar('v');
if(rP->getChar()!='v')
{
ans = false;
cout <<"testSetChar failed" << endl;

}
delete rP;
```

We create the instance with the keyword "new", and obtain a pointer, as we did in C with malloc. In C we can free the storage from malloc using the keyword "free", and in C++ we free the storage associated with an instance of a class with the keyword "delete", which invokes the destructor function.

We could certainly create multiple rooms, each with its own pointer. In homework 3 and 4, we had multiple rooms, and an array of their pointers.

The execution environment needs memory for the attributes of each instance of a class. The execution environment does not need multiple copies of the methods in the class. So the instance of the class has access to the methods, and its own values for its attributes.

Example:

```
Room* theRoomPs[10];//addresses for 10 rooms
...
for(int roomr = 0; roomr<*nrooms; roomr++)
{
...
//make a room and set its treasure
Room** aRoomP = theRoomPs;
aRoomP = aRoomP+roomr;
*aRoomP = (Room*) malloc(sizeof(Room));
//now set the treasures
(*aRoomP)->treasure = tempTreas;
(*aRoomP)->roomNumber = roomr;
...
}
```

In previous homework, we created multiple instances of the datatype Room.

Exercises: (As always, if you have questions, ask. After this occasion, this material becomes background knowledge.)

1. Create multiple instances of the class Room.

```
Room* theRoomPs[10];
for(int roomr = 0; roomr<*nrooms; roomr++){
Room** aRoomP = theRoomPs;
aRoomP += roomr;
*aRoom = (Room*) malloc(sizeof(Room));
}
```

3 Sequence Diagrams

In the C part of the course, we created sequence diagrams as we designed how our code would execute. The components (the parts that were boxed, and had vertical time lines) appeared in the code as memory usually obtained through malloc. The horizontal lines connecting one timeline to another were called messages at the level of abstraction of the sequence diagram, and were the origin of functions. These functions were invoked in .c files, from within other functions.

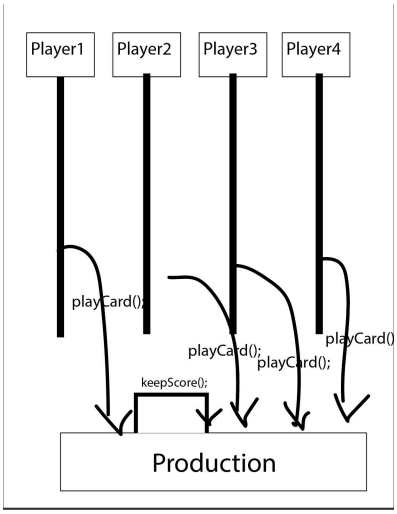
In C++, the components in the sequence diagram are instances of classes. The messages get converted to function invocations; these are now method calls on instances of functions.

We would check the searched status of a particular room, whose pointer is in an array of Room pointers, by bool sd = theRoomPs[col]-¿getSearched();

We can invoke the destructor, using the keyword delete. There is a corresponding notation on the timeline. It ceases to travel down the diagram, and ends with a large X.

Exercises: (As always, if you have questions, ask. After this occasion, this material becomes background knowledge.)

1. Create a sequence diagram for a round of bridge. Thus there are four cards that are played, and four players. One player wins the trick.



4 Patterns

In computer science, a software design pattern, in the sense of a template, is a general solution to a problem in programming. A design pattern provides a

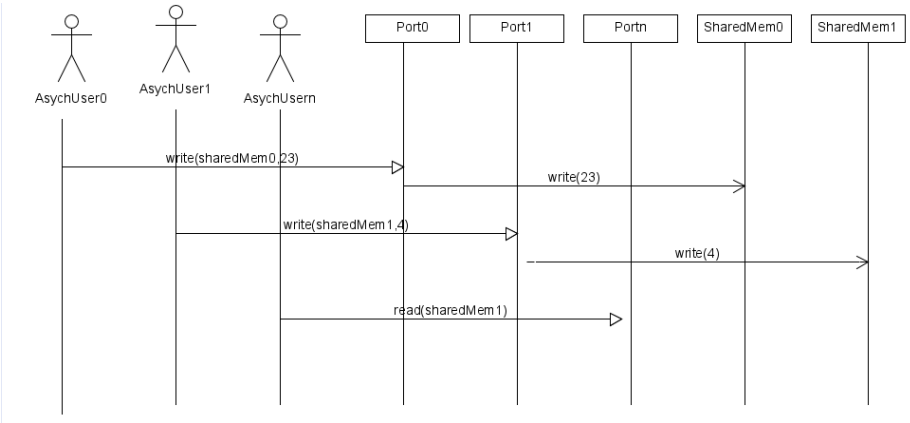


Figure 1: Consider that there may be no constraints on the order or timing of communications from the actors. Serialization must occur within the protocol being depicted.

reusable architectural outline that may speed the development of many computer programs.

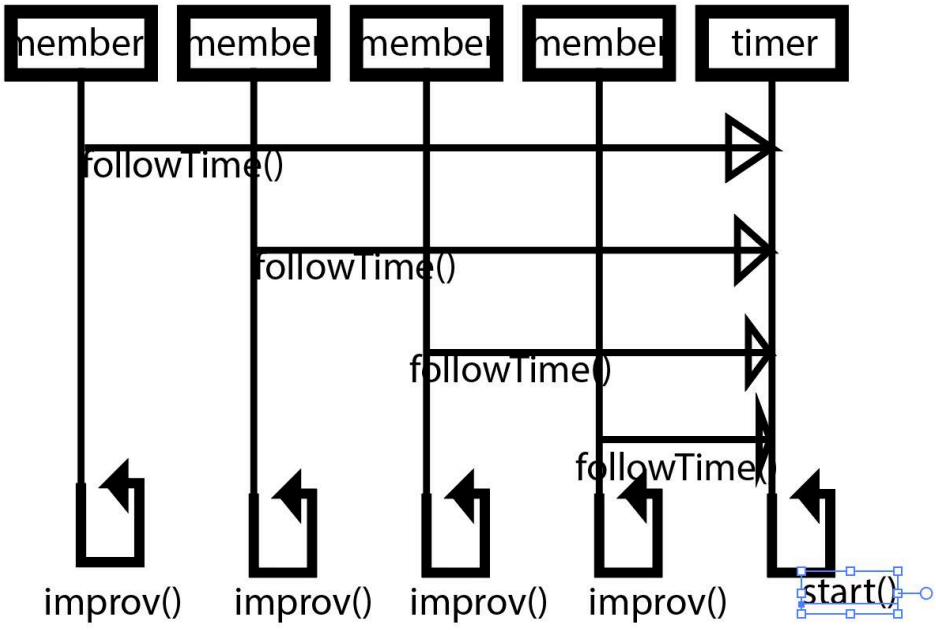
Patterns can be described in terms of sequence diagrams.

Example:

Figure 1 shows a set of asynchronous communicators, such as a swarm of robots, communicating through shared memory. Though shared memory is only one of several communication methods, there are techniques for converting between shared memory and the other techniques. Thus, shared memory is often used in proofs of correctness.

Exercise:

Diagram a marching band (alternatively, a chamber music group, or some members of a sports team) of size four. How do they start? How do they stay synchronized? How do they react to unexpected events?



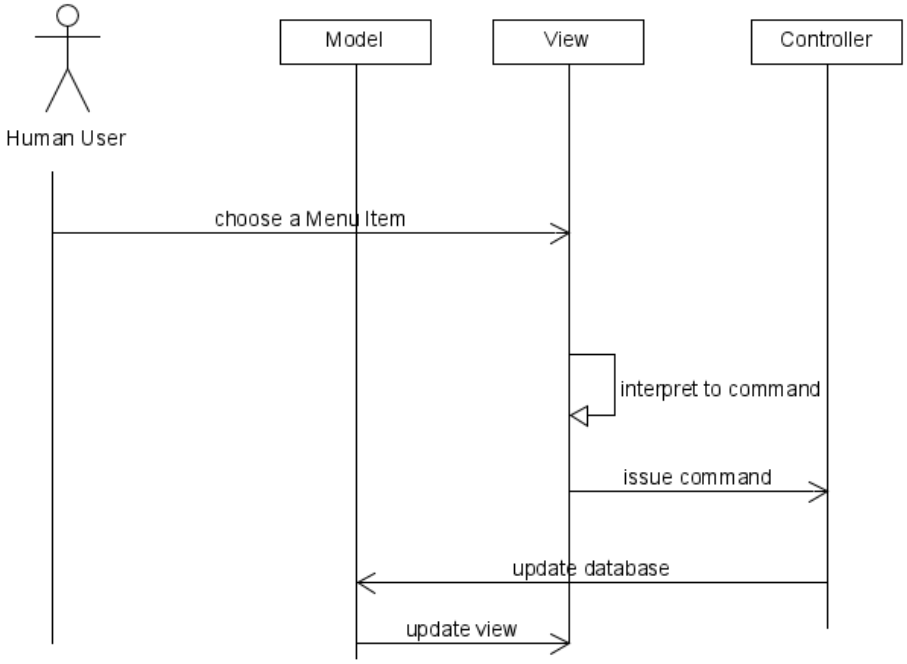


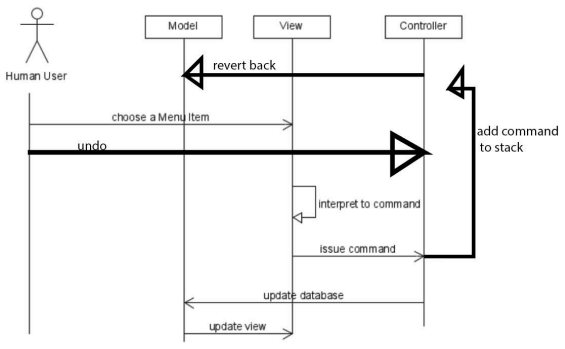
Figure 2: The model-view-controller paradigm separates the concerns of the HCI designer from the concerns of the database expert. The variety of input modalities that serves the user population is kept local to the view. The rest of the application uses a sufficient command set.

and keyboard accelerators, and context dependent mouse/trackball inputs. The view isolates the remainder of the software from this richness, providing to the rest of the software, a command set that is sufficient to convey the user's expressed wishes, without the variety of entry methods. The controller reacts to the user's expressed wishes, usually involving changes to the stored data.

We can represent this information in a sequence diagram. See Figure 2.

Exercises: (As always, if you have questions, ask. After this occasion, this material becomes background knowledge.)

1. Create a sequence diagram for an application using model-view-controller paradigm. Show how the undo command would be implemented.



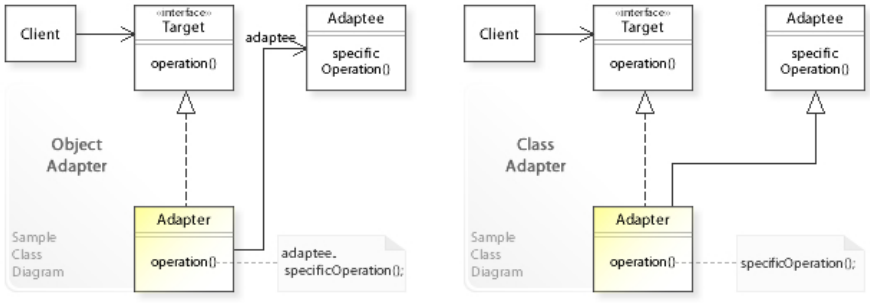


Figure 3: Wrapper pattern displayed in UMLobject diagram.

4.2 Wrapper

The wrapper pattern is useful in providing a desired interface to some existing logic.

The desired interface includes methods that can be invoked on the executable. The adapter accepts those methods, and converts them into the function invocations supported by the existing logic.

Exercises: (As always, if you have questions, ask. After this occasion, this material becomes background knowledge.)

1. Create a sequence diagram for an application using the wrapper pattern. There are three parties: the user software (called the client), the adaptor, which implements the interfaces desired in the target, and the existing (legacy) software (also called the adaptee). The user software issues an invocation. What happens next?

   the adapter will implement the interfaces required by the client by using the adaptee and changing it to make it compatible with the legacy software.

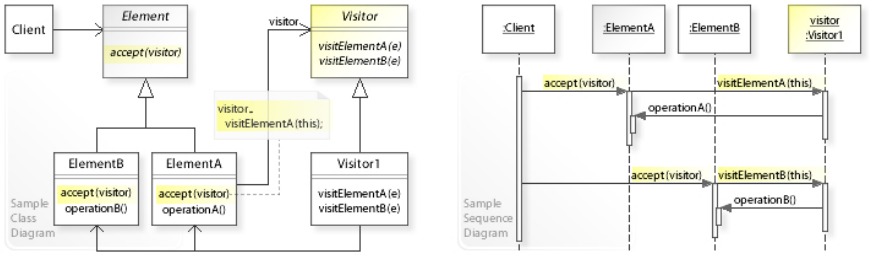


Figure 4: The visitor design pattern is presented as a combination of an object and a sequence diagram.

4.3 Visitor

The visitor pattern is useful for situations where a larger object is built out of components. An outside agent can invoke a function on the larger object, and the consequent invocations on the components can be managed by the Visitor. This pattern is presented in Figure 4.

Exercises: (As always, if you have questions, ask. After this occasion, this material becomes background knowledge.)

1. Create a sequence diagram for an application using the visitor pattern. The parties: the user software (called the client), the large object, which, unlike Figure 4 has three components. How would you modify the diagram so that three components are served?

   The client needs to be able to allow the visitor to interact with the adapter so that the adaptee is able to interpret the operations passed from the visitor.