Exercise Manual for Course 397

C++ Best Practices and Design Patterns

397/MA/S.3/905/S.2

by Ulf Bilting

Technical Editor:
Jasper Kent

© LEARNING TREE INTERNATIONAL, INC.
All rights reserved.

All trademarked product and company names are the property of their respective trademark holders.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, or translated into any language, without the prior written permission of the publisher.

Copying software used in this course is prohibited without the express permission of Learning Tree International, Inc.
Making unauthorized copies of such software violates federal copyright law, which includes both civil and criminal penalties.

Legend for Course Icons.....ii
Hands-On Exercises.....1
Hands-On Exercise 1.1: Using the Toolset.....3
Hands-On Exercise 2.1: Singleton Pattern.....7
Hands-On Exercise 2.2: Template Method Pattern.....9
Hands-On Exercise 3.1: Containers, Iterators, and Algorithms.....11
Hands-On Exercise 3.2: Function Objects.....15
Hands-On Exercise 3.3: Associative Containers.....19
Hands-On Exercise 4.1: Observer Pattern.....25
Hands-On Exercise 4.2: Observer Pattern Variations.....29
Hands-On Exercise 5.1: Using `std::thread`.....35
Hands-On Exercise 6.1: Memory Management.....39
Optional Hands-On Exercise 6.2: Serialization.....45
Hands-On Exercise 7.1: Composite and Visitor Patterns.....49
Alternative Hands-On Exercise 7.1A: Visitor Patterns.....53
Using Visual Studio.....59

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Legend for Course Icons

Standard icons are used in the hands-on exercises to illustrate various phases of each exercise.

	Major step		**Warning**
1. ❑	**Action**		**Hint**
	Checkpoint	STOP	**Stop**
	Question		**Congratulations**
	Information	BONUS	**Bonus**
	Solution/Answer		

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Organization of the exercises

This manual provides instructions for hands-on exercises that you will be doing throughout the course. Some of the terms used here assume some knowledge of Visual Studio. You can find the information you need to use this tool on the last page of this manual.

Solutions

You will find a solution suggestion to each exercise in the `SolutionEx` folders. The solution is a complete, runnable project that you can run and study. These solutions normally include the Bonus exercises.

Whether you spend the time coding or spend the time studying the solution, the choice is entirely up to you. Do what you feel will be of greatest benefit.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Objective

In this exercise, you will gain experience using Visual Studio. You will also encounter the pelican project for the first time, a program to which we will apply many modifications in future exercises.

Visual Studio

1. ❑ ~~Using the instructions on the last page of this manual, open the Visual Studio project Ex11. All projects are found in the folder `C:\Course\397`.~~

2. ❑ ~~Again using the instructions on the last page of this manual, open one or more of the project's source files.~~

3. ❑ ~~Build and run the program.~~

4. ❑ ~~Explore the pelican program. The workings of both a real pelican crossing and this simulation are described in Appendix A in the Course Notes.~~

   You will be making many modifications to the pelican program in future exercises.

Possible improvements—coming exercises

5. ❑ Look at `TimerSet.h`, the administrator class for all the `Timer` objects. We only want one administrator.

   Is there anything preventing the program from creating many `TimerSet` objects?
   ❑ *Yes* ❑ *No*

   In Chapter 2, we will present a solution: the Singleton design pattern.

6. ❑ Look at the `TimerSet` class again.

   How many `Timer` objects can it administer? Is this limit necessary?

   _____

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

In Chapter 3, we will introduce the Standard Template Library, which will give us a flexible container of `Timer` objects.

7. ❑ Look at `main.cpp`. In the loop, the `pollAllTimers` function of the `TimerSet` is called.

Is this really a job for the `main` program? Shouldn't the `TimerSet` manage its own polling?
❑ *Yes* ❑ *No*

In Chapter 5, we introduce threads, so `TimerSet` will be able to have its own separate activity going on in parallel.

There is more to improve. And many chapters left.

Can you find more improvements that ought to be implemented?
❑ *Yes* ❑ *No*

Finding and fixing a bug with the help of the C++11-scoped `enum`

8. ❑ Close the current project and open `Ex11bug`. Run it, and note that the `WAIT` light is on when it should be off, and vice versa.

This is due to a bug in `Pelican.cpp` where a `Pole` function is called (in four places) with the wrong argument: `myPole->setWaitLight(isPressed() ? Light::ON : Light::OFF );`. The argument is of type `LightState`, but the `setWaitLight` wants a `bool`. But, since the old `enum` type converts automatically to `int`, which in its turn, automatically is converted to `bool`, the compiler is happy with this code. It even looks nice to the human eye.

9. ❑ In `Light.h`, change the definition of `enum LightState` to `enum class LightState`.

10. ❑ By repeatedly recompiling, you can now find all the places where `Light::ON` needs to be changed to `Light::LightState::ON`, etc. There are quite a number of them.

There should remain four errors, which are the four calls to `setWaitLight`; they are now flagged with an argument type error.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

11. ❑ Change the four calls to `myPole->setWaitLight(isPressed());` so it gets the `bool` argument that it wanted in the first place.

Was it worth the effort?
❑*Yes* ❑*No*

Congratulations! You have familiarized yourself with the development environment and the theme project, and have warmed up your coding fingers.

STOP

This is the end of the exercise.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Objective

In this exercise, you will convert a class into a singleton.

The Pelican program in the Exercise 2.1 directory does not function correctly. This is because two `TimerSet` objects have been defined. `Timer` objects are added to one `TimerSet`, but `main()` defines and polls a different instance.

You will convert class `TimerSet` into a singleton, thus preventing the definition of more than one instance.

Part 1: Convert `TimerSet` into a singleton.

1. ❑ Compile and run the program. Notice that it does not work—the Pelican gets stuck.

2. ❑ Make `TimerSet` a singleton.

More detail...

a. ❑ In the `.h` file, add a public, static member function called `Instance`, taking no parameters and returning `TimerSet&`. Implement it in the `.cpp` file as a Meyers Singleton, as described in the Course Notes.

b. ❑ Make the default constructor `private`.

Part 2: Use `TimerSet` as a singleton.

3. ❑ Make any necessary modifications to the rest of the program.

More detail...

a. ❑ In `main`, remove the definition of `theTimerSet` and access the singleton as `TimerSet::Instance().pollAllTimers();`.

b. ❑ In `TimerSet.h`, remove the extern definition of `theTimerSet`.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

c. ❑ In `Timer::Timer()`, remove the definition of `theTimerSet` and add the timer to the `TimerSet` using `TimerSet::Instance().addATimer(this);`.

4. ❑ Build and run the program.

Congratulations! You have implemented and used a singleton.

If you have more time...

5. ❑ Identify, declare, and, if necessary, implement the missing constructors, destructors, and operators in the pelican program. For example, if objects of a class should not be copied, declare the copy constructor in a private section, but do not define it, and for a class that uses dynamic memory, declare and define a destructor. Where appropriate, use the `=delete` option to forbid usage of constructors.

Congratulations! You have completed the bonus.

STOP

This is the end of the exercise.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Objective

In this exercise, you will convert a class from using "traditional" polymorphism to the template method pattern. In this case, you will see a small but significant simplification in the code of the derived class.

Convert `Timer` to use template method

1. ❑ Change class `Timer` so that instead of a public, virtual function `tick()`, it has a public, non-virtual function `tick()` that calls a new, pure virtual function `doExpired()` when it detects that `timeToGo` has reached 0.

More detail...

   a. ❑ In file `timer.h`, make member function `tick()` non-virtual. This is the template method. Add a private pure virtual method `doExpired()`. This is the primitive step.

   b. ❑ In file `timer.cpp`, implement the function `void Timer::tick()`. This function should use `Timer::isExpired()` to determine if the timer should fire. It should then call `doExpired()`.

2. ❑ Change class `PelicanTimer` such that it overrides `Timer::doExpired()` rather than `Timer::tick()`. Because `doExpired()` is called only when the timer has expired, it is not necessary to call `Timer::isExpired()` to test for timer expiry.

More detail...

   a. ❑ In file `PelicanTimer`, change the function name `tick` to `doExpired`.

   b. ❑ Remove the check for timer expiry from this function, because it will be called only when the timer has expired. `doExpired()` should still call the target function.

3. ❑ Build and run the program. Verify that the program now works correctly.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Congratulations! You have converted a class to use the template method pattern.

If you have more time...

4. ❑ Currently, the lights do not actually flash when they are supposed to; they just have a gray background. Using class `Timer` (creating a new derivation), modify the `Light` class to make the lights really flash. You will not need to modify any other classes apart from `Light` and your new derived `Timer`.

Congratulations! You have completed the bonus.

STOP

This is the end of the exercise.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Objective

In this exercise, you will gain experience in using the Standard Template Library. In the early parts of this exercise, you will use loops to traverse a collection. Later in the exercise, you will see how these loops can be removed.

Note that each part of the exercises in this chapter builds on the previous part. It is not necessary to remove any code. Your output becomes easier to read if you add statements of the form:

```
cout << "\nExercise 3, Part X";
```

Preamble

1. ❑ Open the project `C:\Course\397\Ex31\Ex31.sln`.

2. ❑ In function `main()`, create an object of class `Pixel` and get it to print itself.

3. ❑ Familiarize yourself with class `Pixel`, then build and run the program.

Part 1: Containers

4. ❑ Declare a `vector` of `Pixel`, called `pixelVec`.

More detail...

a. ❑ To do this, first `#include <vector>`, then specify `using namespace std;`.

b. ❑ The declaration should look like this:

```
vector<Pixel> pixelVec;
```

5. ❑ Using the `push_back` member function of class `vector`, place five `Pixel` objects into the container, choosing attribute values that show the objects are unsorted.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

More detail...

a. ❑ One way to do this is to declare five objects of class `Pixel` individually, called, for example, `pixel1` through `pixel5`. Then, add a copy of each one to the vector, like this:

```
pixelVec.push_back(pixel1); // etc
```

6. ❑ Using `vector::operator[]()` in a loop, print each `Pixel`. Use `vector::size()` to ensure the loop executes the correct number of times.

`operator[]` allows you to access the vector as if it were an array.

7. ❑ Try the new C++11 initializer syntax to give the `vector` a list of `Pixels` when created.

8. ❑ Use the new C++11 `for`-loop syntax to print all `Pixels`.

Congratulations! You have used an SL container.

Part 2: Iterators

9. ❑ Modify your program to use an iterator to traverse the container. Print each element; the output should be identical to that of Part 1.

More detail...

a. ❑ To declare and initialize the iterator, use:

```
vector<Pixel>::iterator iter = pixelVec.begin();
```

10. ❑ Print the collection backward: Use a `reverse_iterator` and the functions `rbegin()` and `rend()` in place of `begin()` and `end()`. These functions return `reverse_iterators`. Incrementing a `reverse_iterator` with + + makes it go backward through the elements of the collection. `rbegin()` returns a reverse iterator initialized to the last element—the correct place for a `reverse_iterator` to start.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

More detail...

a. ❑ To declare and initialize the iterator, use:

```
vector<Pixel>::reverse_iterator r_iter =
pixelVec.rbegin();
```

Congratulations! You have used SL iterators.

Part 3: Algorithms

11. ❑ Write a global function, `pixelPrint`, taking a `const Pixel` by reference as parameter, and make this function call the `print` member function of its parameter.

12. ❑ Add a loop to print your collection using the `for_each()` function. Pass the address of `pixelPrint` as the third parameter to `for_each`. Don't forget to add `#include<algorithm>`.

13. ❑ Use the simple form of the `sort()` algorithm to sort your collection, then print it again. The objects should appear sorted.

Congratulations! You have used SL algorithms.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

If you have more time...

14. ❑ In your pelican project, replace the fixed-size array of `Timer*` in `TimerSet` with a `vector<Timer*>`, and make any necessary additional changes.

Congratulations! You have completed the bonus.

STOP

This is the end of the exercise.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Objective

In this exercise, you will use SL function objects and adapters. If you successfully completed Exercise 3.1, continue in that project; otherwise, open the project `C:\Course\397\Ex32\Ex32.sln`.

Function objects

1. ❑ Declare a vector of pointers to `Pixel`, `pixelPtrVec` and populate it with pointers to your five `Pixels`.

2. ❑ Try sorting this collection with the simple form of `sort()` and print the result.

   Does it work?
   ❑*Yes* ❑*No*

   If you marked "No," why not?

   _____

More detail...

   a. ❑ To print this vector, you will need to write a function `void pixelPtrPrint(const Pixel*)` for `for_each` to call.

3. ❑ Write a class `PixelPtrCompare` with the operator
   `bool PixelPtrCompare::operator()(const Pixel* l, const Pixel* r)const;`
   defined appropriately.

More detail...

   a. ❑ This operator should return `*l < *r`.

4. ❑ Declare an object of class `PixelPtrCompare` called `pixelPtrComp`, and pass this object as the third parameter to the `sort()` algorithm.

   Does it work?
   ❑*Yes* ❑*No*

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

`pixelPtrComp` is a function object. In this part, we will use the power of a function object more fully. Currently, `Pixels` are only sorted by their `x` value. We will provide an option on `y` instead.

5. ❑ Make class `PixelPtrCompare` a friend of `Pixel`.

More detail...

a. ❑ This must be done within the definition of the `Pixel` class.

6. ❑ Add to class `PixelPtrCompare` a `bool` data member called `sortOnY`. This data member will be used to determine whether `Pixel` objects are sorted according to their `x` or `y` values. Also, add a constructor that sets `sortOnY` to `false`, and a function to set the value of `sortOnY` to that of its `bool` parameter.

More detail...

a. ❑ These should be declared as:

```
PixelPtrCompare();
void setSortOnY(bool);
```

7. ❑ Modify `PixelPtrCompare::operator()` to return a value that depends on the appropriate data member of `Pixel`, depending on `PixelPtrCompare`'s current state (the value of `sortOnY` just added in the previous step).

Now that `PixelPtrCompare` is a friend of `Pixel`, it can access private data members of `Pixel` directly.

8. ❑ Modify `main()` to sort your collection on each member, and print the result in each case, without using a loop.

More detail...

a. ❑ Use `for_each` and the `pixelPtrPrint` you created.

9. ❑ Sort in reverse order and print a collection again, using a lambda expression to provide `sort` with the comparing functionality. Print with `for_each` and use a lambda here, as well.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

More detail...

a. ❑ The comparison lambda needs to be called with two `Pixel *` pointers and return a `bool` telling whether the parameter values are in the correct order.

```
[ ](const Pixel * l, const Pixel * r) {
return *r < *l;
}
```

b. ❑ The `for_each` lambda needs to be called with one `Pixel *` pointer and return nothing

```
[ ](const Pixel * l) {
l->print();
}
```

Congratulations! You have used function objects and lambda expressions.

STOP

This is the end of the exercise.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Objective

In this exercise, you will gain experience using the associative containers `set`, `multiset`, `map`, and `multimap`. If you successfully completed Exercise 3.2, continue in that project; otherwise, open the project `C:\Course\397\Ex33\Ex33.sln`.

Part 1: `set` and `multiset`

1. ❑ Declare a `set` of `Pixel`, called `pixelSet`.

More detail...

a. ❑ You will need to **`#include <set>`**.

b. ❑ The declaration should look like this:

```
set<Pixel> pixelSet;
```

2. ❑ Add three `Pixel` objects to `pixelSet`, ensuring that each has a different value for data member `x` and that the values of `x` are not in ascending order.

More detail...

a. ❑ To add elements to the set, type:

```
pixelSet.insert(Pixel(1,0));
pixelSet.insert(Pixel(3,0));
pixelSet.insert(Pixel(2,0));
```

3. ❑ `set` has a member function `size()` that returns the number of elements. Use `cout` to print this value for `pixelSet`.

4. ❑ Use the `for_each` function, calling `pixelPrint()`, to print the elements of `pixelSet`. Note the order in which the objects appear.

Hint...

You will find examples of this in Exercise 3.1.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

5. ❑ Add another element to `pixelSet`, having a value of `x` that is identical to one of the previous objects.

More detail...

a. ❑ To do this, type:

```
pixelSet.insert(Pixel(1,0));
```

6. ❑ Print the number of elements in the set.

Can you explain the result, remembering that a set cannot contain duplicates?

_____

7. ❑ Declare a `multiset` of `Pixel`, called `pixelMultiset`.

More detail...

a. ❑ `multiset` is declared in header `<set>`.

b. ❑ The declaration should look like this:

```
multiset<Pixel> pixelMultiset;
```

8. ❑ Repeat steps 2 to 6 for `pixelMultiset`.

How do the results differ?

_____

Congratulations! You have used the `set` and `multiset` templates from the Standard Library.

Part 2: `map` and `multimap`

9. ❑ Declare a `map` with an `int` key and a referent type of `Pixel`, called `pixelMap`. Use a `typedef` to make the code more readable.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

More detail...

a. ❑ You will need to `#include <map>`.

b. ❑ The declaration should look like this:

```
typedef map<int, Pixel> MapPixel;
MapPixel pixelMap;
```

10. ❑ Attempt to add three `Pixel` objects to `pixelMap`, two of them having the same key value.

More detail...

a. ❑ To add elements to the map, type the following:

```
pixelMap.insert(make_pair(1, Pixel(1,0)));
pixelMap.insert(make_pair(2, Pixel(2,0)));
pixelMap.insert(make_pair(2, Pixel(3,0)));
```

11. ❑ `map` has a member function `size()` that returns the number of elements. Use `cout` to print out this value for `pixelMap`.

Can you explain the value printed?
❑*Yes* ❑*No*

12. ❑ Use the `find` function to access the element with the key value `1`. Remember that in this case, the `find` function returns an iterator, not the actual object.

More detail...

a. ❑ To do this, type the following:

```
MapPixel::iterator found_it = pixelMap.find(1);
```

13. ❑ Check whether the object has been found, and if so, print it.

More detail...

a. ❑ If the object has not been found, `found_it` will be equal to `pixelMap.end()`.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

b. ❑ To print the object, type the following:

```
found_it->second.print();
```

14. ❑ Use the `find` function to attempt to access an element with a nonexistent key value (for example, 4). Check whether the object is found.

15. ❑ Use the subscript operator to return a reference to the object with key value 1, then print the object.

More detail...

a. ❑ To do this, type the following:

```
pixelMap[1].print();
```

16. ❑ Use the subscript operator to return a reference to an object with a nonexistent key value (for example, 4), then print the object.

What happens?

_____

17. ❑ Declare a `multimap` of `Pixel`, called `pixelMultimap`.

More detail...

a. ❑ `multimap` is declared in the header `<map>`.

b. ❑ The declaration should look like this:

```
typedef multimap <int, Pixel> MultimapPixel;
MultimapPixel pixelMultimap;
```

18. ❑ Repeat steps 10 and 11 for `pixelMultimap`.

How does the result differ?

_____

_____

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Congratulations! You have used the `map` and `multimap` templates from the Standard Library.

If you have more time...

19. ❑ A class template can be derived from class `Timer`.

For example:

```
template <class Target> class TimerT: public Timer
{
public:
  typedef void (Target::*TargetFunc)();

  TimerT(Target& target, TargetFunc targetFunc):
    myTarget(target),
    myTargetFunc(targetFunc)
  {
  }
private:
  void doExpired()  {
    (myTarget.*myTargetFunc)();
  }
  Target& myTarget;
  TargetFunc myTargetFunc;
};
```

Class `Pelican` and any other class that requires a `Timer` can instantiate a specialization of this class template. Note that with this approach, class `PelicanTimer` and other derivations of class `Timer` are no longer needed and can be deleted.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Congratulations! You have completed the bonus.

This is the end of the exercise.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Objective

In this exercise, you will convert the pelican program to use the observer pattern. This involves making the `Pelican` a concrete subject and the `Pole` a concrete observer..

Part 1: Implementing subject and observer

1. ❑ Open Exercise 4.1 in Visual Studio.

Hint...

```
C:\Course\397\Ex41\Ex41.sln
```

2. ❑ Implement classes `Subject` and `Observer` according to the notes. The files `Observer.h`, `Subject.h`, and `Subject.cpp` are already added to the project and contain `TODO` comments. Note that you may borrow almost all of the C++ code directly from the Course Notes slides that describe them.

3. ❑ Build the program to ensure that these classes compile.

Congratulations! You have implemented *subject* and *observer* classes.

Part 2: Implementing the observer pattern

4. ❑ Convert the class `Pelican` to a concrete subject, and change all calls to `setTrafficLights()` and/or `setWaitLight()` to `notify()`.

More detail...

a. ❑ Derive class `Pelican` publicly from class `Subject`.

b. ❑ In `pelican.cpp`, change all calls to `setTrafficLights()` and `setWaitLight()` to simply `notify()`.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

5. ❑ Remove every trace of `Pole` from `Pelican`: declaration, initialization, deletion, x/y parameters in the constructor, and x/y arguments when defining the `Pelican` in `main`.

6. ❑ Convert class `Pole` to a concrete observer, making `update()` get the state of the `Pelican`, and call `setTrafficLights()` and `setWaitLight()`. Add a reference to `Pelican` as a parameter to the constructor of class `Pole`. The constructor should attach the `Pole` to the `Pelican`.

More detail...

   a. ❑ Derive class `Pole` publicly from class `Observer`.

   b. ❑ In `Pole.h`, declare the function `update()`.

   c. ❑ In `Pole.h`, add a private member of type `Pelican&`. Add a parameter to `Pole`'s constructor to take a parameter of type `Pelican&`.

   d. ❑ In `Pole.ccp`, use this parameter to initialize the new member variable. In the constructor body, call `attach(*this)` on the `Pelican`.

   e. ❑ In `Pole.cpp`, implement function `update()` such that it gets the state of the `Pelican` and calls functions `setTrafficLights()` and `setWaitLight()`.

7. ❑ Build the program to ensure these classes compile.

Congratulations! You have created a *concrete subject* and a *concrete observer*.

Part 3: Creating multiple observers

8. ❑ Modify `main` to create two `Poles` at different locations on the screen (try `(20,0)` and `(40,0)`).

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

More detail...

a. ❑ This is done by adding to `main`:

```
Pole pole1(20, 0, thePelican);
Pole pole2(40, 0, thePelican);
```

9. ❑ Build and run the program, which should now display two `Poles` updating together. You will need to add a call to `update()` at the end of the `Pole` constructor, so that it sets all its initial lights when created. Try to leave it out and see what doesn't happen.

Congratulations! You have built a program based on the observer pattern.

If you have more time...

10. ❑ Create a new type of concrete observer, and attach an object of this class to the `Pelican`. For instance, create a beeper. `cout << "BEEP!"` that will print a beep "sound" when the Pelican goes green for pedestrians. (Our computers usually do not have audio switched on for classroom-environment reasons.)

How many classes are affected by this addition?

_____

Congratulations! You have completed the bonus.

STOP

This is the end of the exercise.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Objective

In this exercise, you will add a variation to the observer pattern known as a *hint*. It is a way of allowing a subject to tell its observers what has changed without the `Subject` and `Observer` classes needing any specific knowledge of classes derived from them.

Part 1: Sending a hint

1. ❑ Write a class called `Hint`, declaring it in the same file as your `Subject` class.

More detail...

   a. ❑ Use file `subject.h`.

2. ❑ Make `Hint` *polymorphic* by giving it a public virtual destructor. This class is going to be used as a base class for derived hints that will be identified using RTTI.

More detail...

   a. ❑ Define the destructor in the class declaration as follows:

```
virtual ~Hint(){}
```

3. ❑ Modify `Subject::notify` and `Observer::update` to accept a `Hint` by `const` reference as a parameter. Make `notify()` forward the `Hint` to each `Observer`'s `update()`.

More detail...

   a. ❑ The declarations should be:

```
void notify(const Hint&) const; // in class
Subject
virtual void update(const Hint&) = 0; // in class
Observer
```

4. ❑ Modify `Pole::update` and `Pelican`'s call to `Subject::notify` accordingly. At this stage, simply pass a `Hint` object and ignore it in `Pole::update`.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

More detail...

a. ❑ The signature for `update()` should be identical to that in `Observer`.

b. ❑ To call `notify`, create a local `Hint` object and pass it as a parameter. It will get destroyed automatically; e.g.:

```
Hint h;
notify(h);
```

5. ❑ Build and run the program, which should function exactly as it did before this exercise.

Congratulations! You have implemented a hint mechanism. It is significant that the program still works, because concrete subjects and concrete observers should be able to ignore hints and still function properly.

Part 2: Sending derived hints

6. ❑ Define the `LightHint` and `WaitHint` classes in the `pelican.h` file, both publicly derived from `Hint`.

7. ❑ `LightHint` should have an attribute denoting the state of the lights; `WaitHint` should have an attribute denoting the state of the WAIT light. It is helpful if each type of hint has a constructor taking the appropriate value.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

More detail...

a. ❑ The classes should look like this:

```
class LightHint: public Hint{
public:
  LightHint(Pelican::PelicanState s):
    state(s)
  {}
  Pelican::PelicanState state;
};
class WaitHint: public Hint{
public:
  WaitHint (bool p):
    pressed(p)
  {}
  bool pressed;
};
```

8. ❑ In `Pelican`, when the WAIT light state changes, create an object of class `WaitHint`, give its data member an appropriate value, and pass it as a parameter to `Subject::notify`.

More detail...

a. ❑ Change:

```
Hint h;
notify(h);
```

to:

```
notify(WaitHint(pressed));
```

for those calls to notify a WAIT light change.

9. ❑ Do the same when the light state changes.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

More detail...

a. ❑ Change:

```
Hint h;
notify(h);
```

to:

```
notify(LightHint(state));
```

for those calls to notify a light state change.

There are occasions when both the WAIT light and the other lights change state. You will need to call `notify()` twice—one for each.

At this stage, `Pole` will ignore the hint, but should still work. This is desirable because the `Subject` and `Observer` classes often change independently in the way yours just did.

Congratulations! You have made your concrete observer send a hint.

Part 3: Handling the hint using RTTI

10. ❑ In `Pole::update`, use RTTI to determine whether the `Hint` received is a `LightHint` or a `WaitHint`.

11. ❑ Cast the hint to the appropriate type so that you can access its data.

More detail...

a. ❑ Declare a pointer to the derived hint and cast to it:

```
const WaitHint* waitHintPtr;
waitHintPtr = dynamic_cast<const WaitHint*>(&h);
```

b. ❑ Look in the solution or the Course Notes to see other ways of performing this cast.

c. ❑ Perform a similar cast to see if a `LightHint` has been received, and if so, act on it.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

12. ❑ Act on the data received.

More detail...

a. ❑ Separate the functionality for setting the WAIT light from that for setting the other lights, so that only lights indicated by the hint need be modified according to the data received with the hint..

It is now necessary only to update either the WAIT light or the other lights, depending on the class of the hint received. It is no longer necessary to ask `Pelican` for state information.

What if the received `Hint` is of another `Hint` subclass?

_____

Congratulations! You have modified the code to *push* the changed data, whereas `Pole` previously had to *pull* it. You now have a way to vary the amount of information that is pushed on a case-by-case basis to optimize a system, without modifying `Subject`, `Observer`, or `Hint`.

STOP

This is the end of the exercise.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Objective

In this exercise, you will create a thread that repeatedly calls the `TimerSet::pollAllTimers()` function, relieving the `main` function of this duty. This is a much better solution since `TimerSet` really should take care of its own needed actions internally.

Declare and create a thread in `TimerSet` to avoid having `main` polling the `TimerSet`.

1. ❑ Open the project `C:\Course\397\Ex51\Ex51.sln`.

2. ❑ Add appropriate `#include <thread>` in `TimerSet.h`.

3. ❑ Add a thread member to `TimerSet`.

More detail...

a. ❑ In the `private` section, add `std::thread pollingThread;`

4. ❑ Add a `private` function called `TimerSet::poll10Hz` that calls `pollAllTimers()` every 100 ms, in an infinite loop.

Hints...

```
std::chrono::milliseconds pause(100);
       // to define the 100 ms time duration

std::this_thread::sleep_for(pause);
        // to sleep
```

5. ❑ Initialize the thread in the `TimerSet` constructor to run the function in the previous step. Use a lambda expression and note that `this` has to be in the capture list. Detach the newly created thread, since we do not want to `join` it later, but let it die on its own when the program terminates.

More detail...

a. ❑ In the initializer list of the `TimerSet` constructor:
`pollingThread( [this](){ poll10Hz(); } )`

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

b. ❑ In the body of the `TimerSet` constructor:
`pollingThread.detach();`

6. ❑ In `main`, remove the loop calling `pollAllTimers()`.

More detail...

a. ❑ Not needed anymore:

```
while (!kbd.isCharAvail())
{
  TimerSet::Instance()->pollAllTimers();
}
```

7. ❑ Rerun the program; it should work as before, but with better handling of the `TimerSet`.

Congratulations! You have created a thread and let it handle a separate task.

Using `std::thread`

8. ❑ Analyze the program with the added thread to see if there is anything shared between the `TimerSet` thread and the `main` thread.

Is there any risk of data race here; i.e., any risk for simultaneous updates?
❑*Yes* ❑*No*

If so, what could be done about it?

_____

9. ❑ Instead of polling `pollAllTimers` at regular intervals, we could rewrite the entire class to keep a list of future events and sleep for the exact amount of time until the next event, avoiding the ticking and also allowing a better resolution than the current one.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Have a thought on how this could be accomplished?

_____

(You will most probably not have time to implement it during class.)

Using the `std::function` template

The inheritance `Timer -> PelicanTimer` is necessary in old C++ to allow the specific type of member function pointer to be saved by the `PelicanTimer` class and used when calling the `timerExpired` method from `doExpired`. By using a lambda expression instead of a member function pointer and saving it as a `std::function` template object, we can get rid of the inheritance and implement everything in the `Timer` class.

10. ❑ Look at and then delete the `PelicanTimer` class files. The files that used it will include `Timer.h` instead.

11. ❑ In the `Timer` class, turn the `doExpired` method into a member object with the same name, having the type `std::function` template, with no parameter and no return value.

```
std::function<void(void)> doExpired;
```

Include the `<functional>` header file.

12. ❑ Let the `Timer` constructor take a parameter of the same type and let the constructor initialize `doExpired` with the value of the parameter.

13. ❑ In class `Pelican`, change the use of `PelicanTimer` to `Timer`. The `myTimer` member should be given a `Timer` object with a lambda expression that calls `timerExpired` as its body.

```
myTimer = new Timer([this](){ timerExpired(); });
```

The lambda expression captures `this`, since it is a `Pelican` member function that is called. This is exactly what we had to do by hand in the original version. For this, we needed a specially derived version of `Timer`.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

14. ❑ Compile and run.

Congratulations! You have completed the bonus exercise.

This is the end of the exercise.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Objective

In this exercise, you will identify and investigate the memory leaks to be found in a console application.

In Part 1 of the exercise, you will fix the leaks, using the `shared_ptr` template. In particular, you will see how this template facilitates effective memory management when handling exceptions.

In Part 2, you will implement a *parameterized factory method*, also known as a *virtual constructor*. This is a technique for allowing a client to request the creation of an object without specifying its exact class.

Part 1: The `shared_ptr` template

1. ❑ Open the project `C:\Course\397\Ex61\Ex61.sln`.

   The program asks for the name, age, and salary of a person. It then uses this information to create and initialize an object of class `Person`. Each time an object is created, its address is assigned to a pointer. When an exception is thrown, the `Person` pointer isn't put into the `vector`, so the program has a memory leak. The program throws exceptions if the age or salary entered is non-numeric or negative, making it more difficult to cure the leak by using `delete`. The number of `Person` objects is displayed before and after the pointer goes out of scope, and just before the program terminates.

   The `YoungPerson` class is used only in Part 2 of this exercise.

2. ❑ Build and run the program. Try entering non-numeric and negative ages, and note the object counts displayed. Now, try entering non-numeric and negative salaries and note the object counts displayed. Enter some valid values, as well. Finally, enter **q** in response to the name prompt to terminate the program.

   Looking at the program source, can you explain the counts you saw displayed? In particular, look at what happens when a negative age is entered, causing an exception to be thrown within a constructor.

   _____

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

More detail...

a. ❑ What you will see is that, if an exception is thrown within a constructor, the object is deemed not to have been constructed; no memory is consumed and the object's destructor should not be called. Conversely, when a negative salary is entered, the object has already been constructed and, if it is not deleted, a memory leak has occurred.

The next step is optional, but attempting it will give you a better idea of how useful the `shared_ptr` template can be.

3. ❑ Try fixing all of the memory leaks using `delete` on `personPtr`. Do this *without* removing any of the exceptions. Assume the program design requires these things to be defined as they are. A solution to this can be found in directory `C:\Course\397\Solutions\Ex61\InterimSolution`.

More detail...

a. ❑ To catch the leak that occurs when an invalid salary is entered, call `delete` on `personPtr` and assign `nullptr` to `personPtr` before throwing the exception. Assigning `nullptr` is necessary to stop the program from putting the deleted pointer into the vector.

b. ❑ To catch the leak caused by a negative salary, you will have to put the statement that calls `setSalary` into a dedicated `try` block. Follow this with a `catch` clause in which you call `delete` on `personPtr` and assign `nullptr` to `personPtr`.

Now you can solve the problem the easy way, using `shared_ptr`.

4. ❑ Revert to the original source files, and build and run the program. Check that the memory leaks have reappeared.

5. ❑ Change the type of `personPtr` from `Person*` to `shared_ptr<Person>`. You will also need to change its initialization from assignment syntax to constructor syntax, as in the notes. Don't forget the `#include`.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

More detail...

a. ❑ The definition should look like this:

```
std::shared_ptr<Person> personPtr(
                                 new Person(name, age));
```

6. ❑ Change the definition of `people` from
`std::vector<Person*> people;`
to
`std::vector<std::shared_ptr<Person>> people;`

7. ❑ Replace the loop that clears the collection with a call to `people.clear();`. This function will call the destructor for each `shared_ptr` in the `vector`, which, in its turn, will release the memory for each Person. Instead of calling `clear()` explicitly, we could enclose all the code declaring and using the `vector` in its own block. Then, the destructor of `vector` would clear all its contents at block exit. The solution project uses this approach.

8. ❑ Run the program.

a. Are there any leaks?
❑ *Yes* ❑ *No*

b. Which solution do you prefer?

_____

Congratulations! You have used the Standard Library `shared_ptr` template to cure memory leaks.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Part 2: Factory method

Sometimes, we would like to create an object without knowing its exact class. For instance, we have the `YoungPerson` class, which is intended to represent a person under 18, but this age limit is not known in `main.cpp`. Unfortunately, when we call `new Person`, a `Person` object will always be returned, never an object of a class derived from `Person`. If we want the possibility of a derived object, we can use what is known as a parameterized factory method. In this case, we write a function that returns a `Person*`, which can therefore return a pointer to either a `Person` or any class derived from `Person`. We use such a function in place of a constructor.

9. ❑ Add to class `Person` a public, static member function called `GetPerson`, taking name (a `string`) and age (an `int`) as parameters, and returning a `shared_ptr<Person>`.

More detail...

a. ❑ Here is the declaration:

```
static Person* GetPerson(std::string name, int
age);
```

10. ❑ Implement `GetPerson` such that it returns a new `YoungPerson` if age is less than 18; otherwise, it should return a new `Person`. Use `make_shared` to create the objects.

11. ❑ In `main()`, call this function instead of `new Person(name, age);`

More detail...

a. ❑ The call should look like this:

```
Person::GetPerson(name, age);
```

The parameter age is used by `GetPerson` to determine which kind of object to create. Hence, the term parameterized factory method.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Congratulations! You have implemented the factory method pattern.

If you have more time...

12. ❑ Rewrite your solution to use `unique_ptr` rather than `shared_ptr`. This is trickier since you cannot copy a `unique_ptr` into the collection; it needs to be *moved* there.

More detail...

Instead of using the copy constructor of `unique_ptr`, we want to `move` it to the container: `people.push_back(move(personPtr));`

Congratulations! You have completed the bonus exercises.

STOP

This is the end of the exercise.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Objective

In this exercise, you will take a program that creates and displays some objects and modify it to store and retrieve those objects using serialization.

1. ❑ Open the project `C:\Course\397\Ex62\Ex62.sln`.

2. ❑ Build, run, and examine the program.

3. ❑ Create some `Person` details by selecting **A**, then typing a first name, last name, and age in response to the prompts. Be careful to type accurately because this program may not be as robust as you would have written it to be.

4. ❑ Display the details by selecting **D**.

5. ❑ Exit the program and run it again. Notice that all of the details you entered on the previous run have disappeared.

6. ❑ Modify the program to use serialization. First, make the class `Person` serializable by adding to it a template function called `serialize`.

More detail...

   a. ❑ You should add a private member function template with the following definition:

```
template <class Archive>
void serialize(Archive& ar, const unsigned int
ver)
{...}
```

   b. ❑ The `serialize` function should call the archive object's `operator&` on each data member of class `Person`.

   c. ❑ Make the class `boost::serialization::access` a friend of class `Person`. In addition to adding a `friend` declaration, you will need to `#include <boost/serialization/access.hpp>`.

7. ❑ On program startup, look for a file called `file.dat`. If it is present, deserialize it into `people`, the SL collection of `Person*` defined in `main.cpp`.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

More detail...

a. ❑ Uncomment `#include` declarations for:

```
#include <fstream>
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <boost/serialization/list.hpp>
```

b. ❑ At the start of `main()`, enclosed within a `try` block, create an `ifstream` for `file.dat`, create a `text_iarchive` for the `ifstream`, and use `operator>>` to deserialize into `people`.

8. ❑ On program exit, serialize `people` into the file `file.dat`.

More detail...

a. ❑ At the end of `main()`, create an `ofstream` for `file.dat`, create a `text_oarchive` for the `ofstream`, and use `operator<<` to serialize `people`.

9. ❑ Build and run the program. You should find that data you enter in one run is still there in the next.

Congratulations! You have implemented serialization using Boost facilities.

If you have more time...

10. ❑ Extend the program to deal with derived classes. For instance, create and display `Employees` as well as `Persons`.

The statement needed in `Employee::serialize()` to serialize its base class is:

```
ar & boost::serialization::base_object <Person>(*this);
```


© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

You will also need to add this to `main.cpp`:

```
#include <boost/serialization/export.hpp>
BOOST_CLASS_EXPORT_GUID(Employee, "Employee");
```

This is to tell Boost about the class `Employee` since its objects are only serialized through pointers to `Person`.

Congratulations! You have completed the bonus exercise.

Save serialized data in XML format.

11. ❑ Change the types of Boost archives from `text_iarchive` to **`xml_iarchive`** and `text_oarchive` to **`xml_oarchive`**.

    And add three new includes:
    ```
    #include <boost/archive/xml_oarchive.hpp>
    #include <boost/archive/xml_iarchive.hpp>
    #include <boost/serialization/nvp.hp>
    ```

12. ❑ Change the file name (both for reading and writing) to **`file.xml`**

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

13. ❑ XML wants a tag name for each saved item. A Boost convenience macro will name the tag after the saved item. Use this with all three serialization operators in Boost: >>, << and &. For example:

```
ar & boost::serialization::base_object <Person>(*this); //
turns into

ar & BOOST_SERIALIZATION_BASE_OBJECT_NVP(Person);

ia >> people;
// turns into
ia >> BOOST_SERIALIZATION_NVP(people);
ar & firstName;
// turns into
ar & BOOST_SERIALIZATION_NVP(firstName);
```

14. ❑ Compile and run. Study the contents of the XML file.

Congratulations! You have serialized objects to XML files.

STOP

This is the end of the exercise.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Objective

In this exercise, you will explore the use of the composite and visitor patterns. The composite pattern is used to create, search, and print a phone number list that allows person entries, company entries, and groups of these entries.

The visitor pattern will allow easy addition of new operations on the Address Book structure.

There will be two steps:

- **Finishing the composite print function**
- **Adding a visitor that counts objects**

Part 1: Composite functions

1. ❑ Open the project `C:\Course\397\Ex71\Ex71.sln`.

2. ❑ Study how an Address Book is built with code in `main`, containing single entries, groups, and a nested group.

3. ❑ Draw the UML diagram for `Entry`, `PersonEntry`, `CompanyEntry`, and `GroupEntry`.

   Which class is the composite?

   _____

4. ❑ The `GroupEntry::print` function is currently empty. Implement it by printing the name of the `GroupEntry` (preceded by the `indent` space string), then calling `print` for all children entries. Each recursive call of the `print` function of children should make the `indent` spacing string a little longer, so the printed lines will reflect the groups with indentation.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Hint...

```
std::cout << indent << "Group: " << getName() <<
'\n';
       // print the group name with indentation
for(auto p : _entries)
{
      p->print(indent + "  ");
      // print all children with added indentation
}
```

5. ❑ Build and run the program; the printout should reflect the group structure. Ignore the visitor printout at the bottom for now.

Part 2: Using the visitor pattern to add functionality

6. ❑ Study the print functionality.

In how many classes is it spread out?

_____

7. ❑ Study the use of the `FindVisitor` at the end of `main`. Run the program to see the printed result.

In how many classes is the find functionality spread out?

_____

Compare with the previous step.

8. ❑ Add a `CountVisitor` class. It should visit the Address Book and count the number of `PersonEntries`, `CompanyEntries`, and `GroupEntries`, and add a way of printing or getting the result. Use the `FindVisitor` class as a model. Empty class files are already there for `CountVisitor`.

More details...

a. ❑ The `CountVisitor` class needs three private integers to count each type of `Entry` object. Initialize them to zero.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

b. ❑ The `CountVisitor` class needs to inherit the `EntryVisitor` interface.

c. ❑ The `CountVisitor` class needs to implement the pure virtual functions inherited from `EntryVisitor`. Look at `FindVisitor` as a model.

d. ❑ Implement the three functions from the previous step; e.g.:

```
void CountVisitor::visit(const PersonEntry& p)
{
        nPerson++;
}
```

e. ❑ Add a function `CountVisitor::printCounts()` that prints the integer values from Step 8a.

9. ❑ Add code in `main` to let the `CountVisitor` visit the Address Book, and afterward, present the result; i.e., the number of various objects.

Hint...

```
CountVisitor cv;
addressbook.accept(cv);
cv.printCounts();
```

How many other classes were affected when you added the `CountVisitor`?

_____

10. ❑ If you were to add anything new to the data components (the `Entry` hierarchy); e.g., an e-mail address or a new type of `Entry`, how would that affect the `Visitor` classes?

Congratulations! You have successfully used the composite and visitor patterns.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

If you have more time...

11. ❑ The printing functionality is currently in the composite hierarchy, which means that it is spread throughout all of the classes.

Would it be better to refactor the printing into a visitor, as well, to have all the printing code in one place?

_____

Any disadvantages?

_____

Try it, if you have time.

Congratulations! You have completed the bonus exercise.

STOP

This is the end of the exercise.

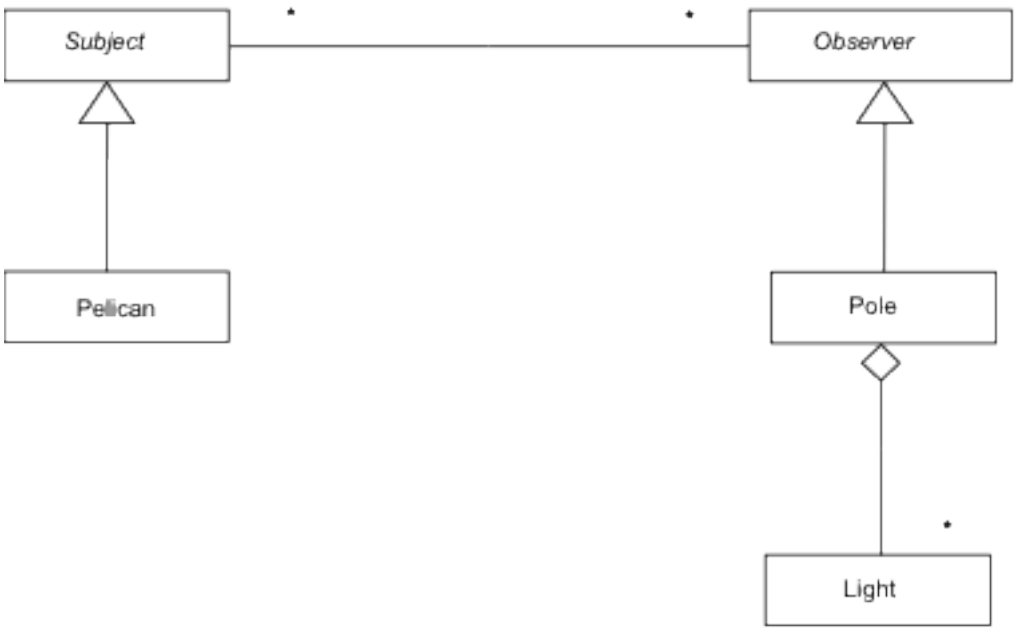
© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

Objective

You have already separated the state of the crossing (held in class `Pelican`) from the visual representation of that state (class `Pole`). In this exercise, you will further separate the state of the `Pole` from the means used to display it. We will do this using the visitor pattern. When the `Pole` class has determined which state each light should be in, you will visit each one and display it.

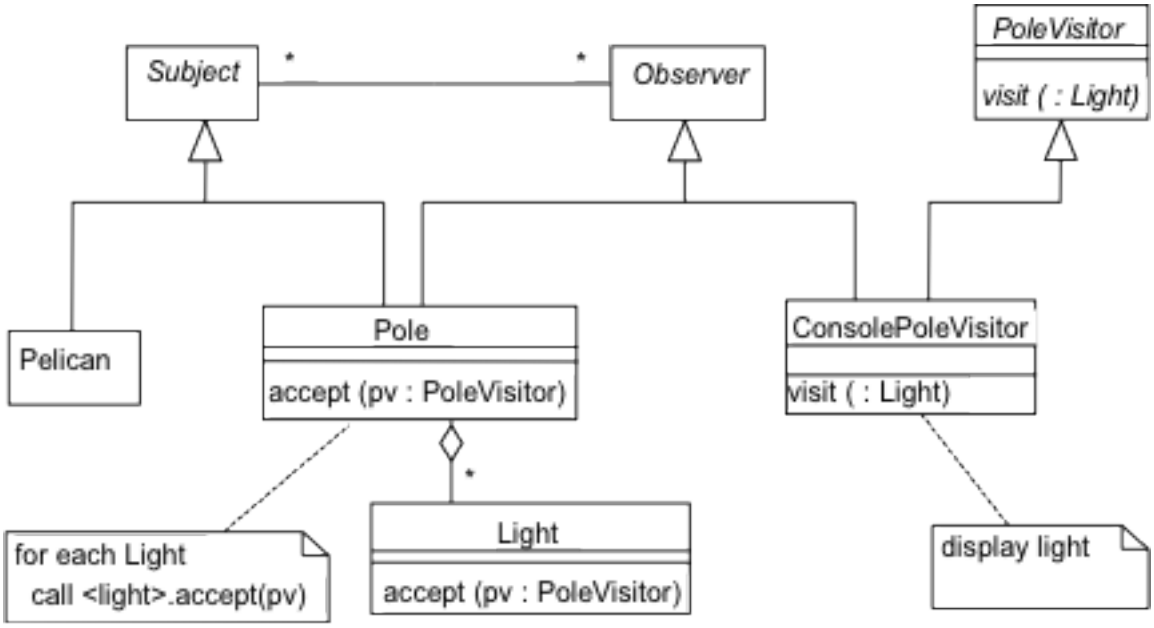
The class diagrams below show the pelican project before and after completing this exercise. Notice that `Pole` becomes both an observer and a subject. We will treat `Pole` as the root of the *object structure hierarchy*, even though, in this simple example, we are not using the full *composite pattern*.

Before this exercise:



© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

After this exercise:



Part 1: Implement an observer pattern between `Pole` (subject) and `ConsolePoleVisitor` (observer)

1. ❑ Modify `Pole` to become publicly derived from both `Observer` and `Subject`.

2. ❑ Amend `Pole::update` to call `notify` from its own `update` function, after it has set its lights and/or WAIT light.

3. ❑ Create a new file, `PoleVisitor.h`, and add it to your Visual Studio project. Define the base class `PoleVisitor` according to the diagram above. The parameter to `visit` should be a `Light`, passed by reference to avoid unwanted copying.

4. ❑ Create new files, `ConsolePoleVisitor.h` and `ConsolePoleVisitor.cpp`. Define the `ConsolePoleVisitor` class, according to the diagram above. The constructor for this class should take a `Pole` by reference as its parameter. It should store this as a reference member variable and then call `attach()` to attach itself to the `Pole`. For now, override member function `update` (declared virtual in class `Observer`) in class `ConsolePoleVisitor` to write a message to `cerr` indicating the function was called.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

More detail...

a. ❑ `ConsolePoleVisitor` should be publicly derived from both `PoleVisitor` and `Observer`. It will therefore need to override both `Observer::update` and `PoleVisitor::visit`.

5. ❑ Modify `main()` such that it creates a `ConsolePoleVisitor` for each `Pole`.

More detail...

a. ❑ A `ConsolePoleVisitor` is an `Observer` as well as a `PoleVisitor`, so attach it to a `Pole` in the same way you attached `Poles` to the `Pelican`.

6. ❑ Build and run your program.
In this test, check out your message flow: You should see that `ConsolePoleVisitor::update` is called each time the lights change.

Congratulations! Now you are ready to complete the implementation of the visitor pattern.

Part 2: Implement a visitor

7. ❑ Alter class `Light` so that it can return its color, position, and string through a set of access functions. Return `Screen:BLACK` if the `Light` is off and 128 + color if it is flashing.

More detail...

a. ❑ Declare these functions as:

```
int getColor() const;
std::string getName() const;
int getX() const;
int getY() const;
```

8. ❑ Add an `accept` member function to class `Light`, taking a `PoleVisitor` by reference as its parameter.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

More detail...

a. ❑ This function should call the `visit` function on the `PoleVisitor`, passing itself as parameter.

9. ❑ Remove the display functionality from `Light::set`. This will now be done by a visitor.

10. ❑ Add a member function `Pole::accept`, taking a `PoleVisitor` by reference as its parameter. This function should call `accept` on each of its `Lights`, passing the `PoleVisitor` as parameter.

11. ❑ Modify `ConsolePoleVisitor::update` such that it calls `accept` for the `Pole` to which it is attached, passing itself as parameter. Also, put a call to `accept()` in the constructor to force the initial refresh.

12. ❑ Implement `ConsolePole::visit` such that it displays the `Light` passed as its parameter in a console window, using `Screen::write`.

More detail...

a. ❑ You will need to call the state access functions of the `Light` to determine how and where to display it.

13. ❑ Build and run your program.

Congratulations! You have implemented the visitor pattern.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

If you have more time: To show the value of this pattern, create a second visitor class.

14. ❑ Create a visitor that adds up the total wattage being consumed by all the lights on a `Pole` at a particular time.
Assume a power usage of 60W for a light that is `ON`, 30W for `FLASH`, and 0W for `OFF`.
This visitor need not be launched automatically using the observer pattern.
Simply get it to respond to a key press in the main program.

You have implemented a second visitor with very little impact on existing code.

This is the end of the exercise.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.

This section describes all you need to know about Visual Studio for use in this course.

To open a project

Start Visual Studio by double-clicking the icon on the desktop. On the start page, click **Open Project**, or, from the File menu, select **Open | Project/Solution**. Navigate to the appropriate directory (e.g., `Ex11`), then double-click the `.sln` file.

To edit a file

Select the file by double-clicking in the file list on the left. Click in the file window and start typing.

To build a project

Select **Build | Build <exercise name>** or click the corresponding tool button. It is not necessary to save files before building; this happens automatically. Alternatively, press `<Ctrl><Shift><B>`.

To correct errors

If there are compilation errors in your project, they will be listed in the Build tab of the output window at the bottom of the screen. Selecting a line that reports an error or warning will take you directly to the offending line.

To run a program

Select **Debug | Start Without Debugging**. The program will automatically stop at the end with a "Press any key to continue" message. Alternatively, press `<Ctrl><F5>`.

You can also use Start Debugging or the green arrow tool button. To prevent the console from disappearing at the end of program execution, place a breakpoint at the end of `main()` by double-clicking in the left margin against the appropriate line. Alternatively, press `<Ctrl><F5>`.

© Learning Tree International, Inc. All rights reserved. Not to be reproduced without prior written consent.