|  |
| --- |
| **Assignment 2: Design Patterns** |

# Codes with Design Patterns

Use the existing codes with each of the required Design Patterns and add explanations of your codes.

## 1 Singleton Pattern (25%)

### Before : Given codes

|  |
| --- |
| class MyClass  {  private: // private member variables  int m\_value;  public: // constructor and destructor  MyClass( int value = 0 ) { m\_value = value; }  public: // getter and setter  int get\_value() { return m\_value; }  void set\_value( int value ) { m\_value = value; }  };  void func1( void )  {  MyClass \*MyObject = new MyClass;  MyObject->set\_value( 1 );  cout << "func1: Address of MyObject is " << MyObject << " and value is " <<  MyObject->get\_value() << '\n';  }  void func2( void )  {  MyClass \*MyObject = new MyClass;  MyObject->set\_value( 2 );  cout << "func2: Address of MyObject is " << MyObject << " and value is " <<  MyObject->get\_value() << '\n';  }  int main()  {  func1();  func2();  } |

### Output

The output should be looks like this;

|  |
| --- |
| func1: Address of MyObject is 00765ED0 and value is 1  func2: Address of MyObject is 007697A8 and value is 2 |

\* The address of your 'MyObject' can be different in your system.

### What to do

If you see the output, the address of 'MyObject' in func1() and func2() are different from each other. That means, two objects have created and it’s reasonable because each of the function has its own object with 'new' keyword.

You need to change this codes to make sure 'class MyClass' can have only one object for your application.

### After : Update the codes by yourself

Make the class responsible for its own global pointer and "initialization on first use" (by using a private static pointer and a public static accessor method). The client uses only the public accessor method.

|  |
| --- |
| class MyClass  {  private:  int m\_value;  public:  int get\_value() { return m\_value; }  void set\_value(int value) { m\_value = value; }  **// \*\* TODO \*\* Put your code here!**  };  **// \*\* TODO \*\* Put your code here! To allocate and initialize your object.**  void func1(void)  {  MyClass::instance()->set\_value(1);  cout << "func1: Address of MyObject is " << MyObject << " and value is " <<  MyObject>get\_value() << '\n';  }  void func2(void)  {  **// \*\* TODO \*\* Put your code here!**  cout << "func1: Address of MyObject is " << MyObject << " and value is " <<  MyObject>get\_value() << '\n';  }  int main()  {  func1();  func2();  } |

### Your Explanation

Explain how you designed, and how you implemented.

|  |
| --- |
| **1. Explain your design and your implementation**  I implemented a generic type Singleton Template that uses a .h only, allowing any class to inherit from this singleton.h where they can then define the type of object in their individual classes themselves, making this singleton template class dynamic enough to allow different types of object to inherit from the exact same class to turned into a singleton class.  How singleton works and how I’ve implemented it is by disabling a class’ default destructor and contructor. And only allowing that code to be executed via the singleton template itself. This allows me to control how many instances I should create. In the Singleton template class, I have an instance pointer that on first call of (Class)::GetInstance(), creates a new object. And on future GetInstance()s they will return the initial object created at the first call. This ensures only ONE object created hence being a singleton.  **2. Class diagram of your codes** |

**Unit Test**

Design a Unit test to test that your singleton pattern works

## 2 Factory Pattern (25%)

### Before : Given codes

|  |
| --- |
| class Map {  private:  // ...  public:  void AddField( Field \* ); // Add the field into the map.  void AddObstacle( Obstacle \* ); // Add the obstacle into the map.  };  class Field {  private:  int m\_field\_no;  public:  Field( int n ) : m\_field\_no( n ) {};  };  class Obstacle {  // ...  };  class MergeMap {  private:  // ...  public:  Map \* CreateMap();  };  Map \* MergeMap::CreateMap()  {  Map \*map = new Map;  Field \*first\_field = new Field( 1 );  Field \*second\_field = new Field( 2 );  Obstacle \*obstacle = new Obstacle;  map->AddField( first\_field );  map->AddField( second\_field );  Map->AddObstacle( obstacle );  return map;  } |

There are some issues with above codes;

* The flexibility of the code is poor.
* If you want to change or update the any of field or obstacle in the map, you need to update the codes inside the codes.

### After

Use the Factory Design Pattern to make better codes.

|  |
| --- |
| class Factory {  public:  **// \*\* TODO \*\***  **// Add your code here to make a virtual factories for Map, Field and Obstacle.**  };  Map \* MergeMap::CreateMap( Factory &factory )  {  Map \*map = factory.ProduceMap();  Field \*first\_field = factory.ProduceField( int 1 ); return new Field(1);  Field \*second\_field = factory.ProduceField( 2 );  Obstacle \*obstacle = factory.ProduceObstacle();  map->AddField( first\_field );  map->AddField( second\_field );  map->AddObstacle( obstacle );  return map;  } |

### Your Explanation

Explain how you designed, and how you implemented.

|  |
| --- |
| **1. Explain your design and your implementation**  Created a factory class that helps build objects including: Map, Field and Obstacle objects. Through this factory, there are 3 functions for each object’s building process respectively. Each function for e.g. BuildObstacle() requests for parameters that are then used to create the Obstacle object in the factory before returning it to the client. With this, a Factory pattern is created as it allows for objects to be build off arguments rather than code.  In Source.cpp, I create a map using MergeMap#CreateMap(Factory) and I print a toString version of the object in cout.  **2. Class diagram of your code** |

**Unit Test**

Design a Unit test to test that your factory pattern works

## 3 Decorator Pattern (25%)

Write the codes for the scenario below;

|  |
| --- |
| I have a requirement where Cake which can be of Chocolate, Strawberry, etc. So, consider a case that in future you want to add one more flavor to the cake then no need to extend BaseCake instead extend CakeDecorator with the new flavor. Decorator pattern makes it simple otherwise there may be a possibility to extend Cake class which will be difficult to maintain if more and more flavors are getting added. Moreover, it breaks the Open-Closed design principle because for extension of flavors the BaseCake is getting opened and extended.  With Decorator pattern decoupling is easy. |

### Output

The output should be looks like this;

|  |
| --- |
| Basic Cake :  Cake  Price of Cake : 30  Chocolate decorated Cake  Cake decorated with Chocolate  Price of Chocolate Cake : 70  Strawberry decorated Cake  Cake decorated with Strawberry  Price of Strawberry Cake : 110 |

### Your code

|  |
| --- |
| // Component  class Cake  {  public:  virtual string Serve() = 0;  virtual float price() = 0;  };      // Concrete Component  class BaseCake : public Cake  {  public:  **// \*\* TODO \*\* Put your code here!**  };    // Decorator  class CakeDecorator: public Cake  {  protected:  Cake \*m\_Cake;  public:    CakeDecorator( **/\* \*\* TODO \*\* Put your code here! \*/** ): m\_Cake(baseCake){}    string Serve()  {  return m\_Cake->Serve();  }    float price()  {  return m\_Cake->price();  }  };      // Concrete Decorator  class ChocolateCake: public CakeDecorator  {  public:  ChocolateCake(Cake \*baseCake): CakeDecorator(baseCake){}    **// \*\* TODO \*\* Put your code here!**  };      class StrawberryCake: public CakeDecorator  {  public:  StrawberryCake(Cake \*baseCake): CakeDecorator(baseCake){}    string Serve()  {  return m\_Cake->Serve() + " decorated with Strawberry ";  }  float price()  {  return m\_Cake->price() + 80;  }  };    int main()  {  Cake \*baseCake = new BaseCake();  cout << "Basic Cake \n";  cout << baseCake->Serve() << endl;  cout << baseCake->price() << endl;    Cake \*decoratedCake = new ChocolateCake(baseCake);  cout << "Chocolate decorated Cake \n";  cout << decoratedCake->Serve() << endl;  cout << decoratedCake->price() << endl;    delete decoratedCake;    decoratedCake = new StrawberryCake(baseCake);  cout << "Strawberry decorated Cake \n";  cout << decoratedCake -> Serve() << endl;  cout << decoratedCake -> price() << endl;    delete decoratedCake;  delete baseCake;  return 0;  } |

### Your Explanation

Explain how you designed, and how you implemented.

|  |
| --- |
| **1. Explain your design and your implementation**  I created a concrete class named “BaseCake” where it contains the main code for what a cake should have. Aka Serve() function and a Price() function. Where serve function returns “Cake is served” and Price returns the “Standard pricing” of a cake (Which I had set to $12.50). So then I create an abstract decorator for the cake, to allow for “Decoration” of the cake via the CakeDecorator class. And I create the ChocolateCake decorator class which then I would then override the Serve() and Price() functions in ChocolateCake to allow for a different pricing of cake, and a different Serve Message (Aka: for serve: “Cake is served with chocolate decoration”, for price its BaseCake price + some added value.  **2. Class diagram of your code** |

**Unit Test**

Design a Unit test to test that your decorator pattern works

## 4 Prototype Pattern (25%)

### Before

The architect has done an admirable job of decoupling the client from Action concrete derived classes, and, exercising polymorphism. But there remains coupling where instances are actually created.

|  |
| --- |
| class Action  {  public:  virtual void Iam() = 0;  };  class Batman: public Action  {  public:  void Iam()  {  cout << "Batman\n";  }  };  class Superman: public Action  {  public:  void Iam()  {  cout << "Superman\n";  }  };  class Ironman: public Action  {  public:  void Iam()  {  cout << "Ironman\n";  }  };  int main()  {  vector roles;  int choice;  while(true)  {  cout << "(1)Batman (2)Superman (3)Ironman (0)Go: ";  cin >> choice;  if(choice == 0)  break;  else if(choice == 1)  roles.push\_back(new Batman);  else if(choice == 2)  roles.push\_back(new Superman);  else  roles.push\_back(new Ironman);  }  for(int i = 0; i < roles.size(); i++)  roles[i]->Iam();  for(int i = 0; i < roles.size(); i++)  delete roles[i];  } |

### Output

The output should be looks like this;

|  |
| --- |
| Batman(1) Superman(2) Ironman(3) Go(0): 2  Batman(1) Superman(2) Ironman(3) Go(0): 1  Batman(1) Superman(2) Ironman(3) Go(0): 3  Batman(1) Superman(2) Ironman(3) Go(0): 0  Superman  Batman  Ironman |

### After

A clone() method has been added to the Action hierarchy. Each derived class implements that method by returning an instance of itself. A Factory class has been introduced that maintains a suite of "breeder" objects (aka proto- types), and knows how to delegate to the correct prototype.

|  |
| --- |
| class Action {  public:  virtual Action\* clone() = 0;  virtual void Iam() = 0;  };  class Factory {  public:  **// \*\* TODO \*\* Put your code here!**  private:  static Action\* s\_prototypes[4];  };  int main() {  vector roles;  int choice;  while(true)  {  cout << "(1)Batman (2)Superman (3)Ironman (0)Go: ";  cin >> choice;  if (choice == 0)  break;  roles.push\_back(  Factory::do\_action( choice ) );  }  for (int i=0; i < roles.size(); ++i)  roles[i]->Iam();  for (int i=0; i < roles.size(); ++i)  delete roles[i];  }  class Batman : public Action {  public:  **// \*\* TODO \*\* Put your code here!**  void Iam() {  cout << "Batman\n";  }  };  class Superman : public Action {  public:  **// \*\* TODO \*\* Put your code here!**  void Iam() {  cout << "Superman\n";  }  };  class Ironman : public Action {  public:  **// \*\* TODO \*\* Put your code here!**  void Iam() {  cout << "Ironman\n";  }  };  Action \*Factory::s\_prototypes[] = {  0, new Batman, new Superman, new Ironman  };  Action \*Factory::do\_action( int choice ) {  return **/\* \*\* TODO \*\* Put your code here! \*/**;  } |

### Your Explanation

Explain how you designed, and how you implemented.

|  |
| --- |
| **1. Explain your design and your implementation**  Basically I added a “Clonable” implementation for all my Action classes. Whereby calling clone() will return an object with identical data value and type as the one originally called from. Using this, I implemented a singleton factory that builds a specific Action\* via cloning it from a prototype list. In this way, for e.g. wherever the factory is called, (E.g. in Source.cpp), you do not need to include Action Types like “Batman, Superman, Ironman” you just need to include Action.h, and It will still generate the desired results (Aka the client side code does not need to know what Action Type object is created. Just has to make sure the arguments are parsed in correctly. The clone function since Action type does not have any data members is a function with no parameters and just generates an equivalent Action type object.  **2. Class diagram of your code** |

**Unit Test**

Design a Unit test to test that your prototype pattern works

**Rubrics**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Below Expectations | Within Expectations | Exceed Expectations |
| Source Code (10%) | Does not follow the correct implementation | Implemented correctly and accurately | Correct and accurate implementation with multiple examples with the design |
| Explanation (10%) | Poor and vague explanation | Accurate explanation | Able to explain accurately and in detail |
| Unit Test (5%) | Unit test does not work | Unit test works | Multiple unit tests are created for the single design pattern |