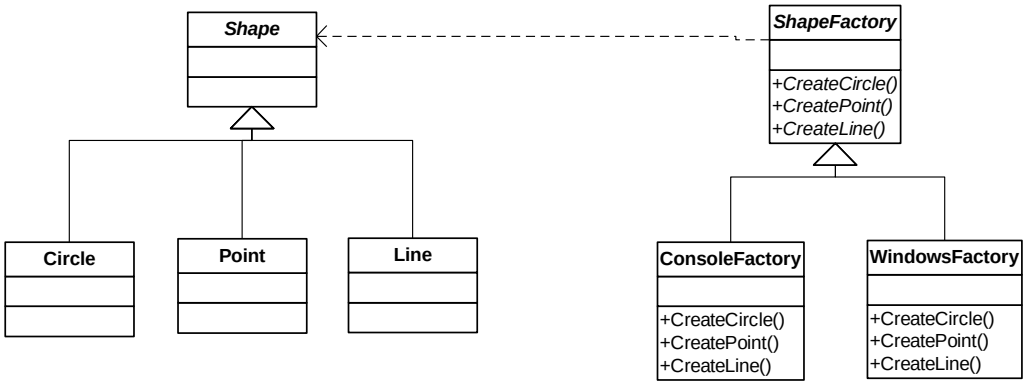
1. (Reengineering Factory Patterns I)

The starting point is the following GOF class hierarchy:



We wish to improve this design in a number of ways, for example reduction in the number of classes need and narrower interfaces.

Answer the following questions:

a) While keeping the same factory redesign the interface to create a single method whose return type is a tuple whole elements are the products that we wish to create.
b) Test the new code and check the validity of the tuple contents.
c) Compare the GOF solution with the new solution. For example, how much effort is needed with both design if we wish to a new product (for example, a class that models rectangles)?

2. (Reengineering Factory Patterns II)

The objective of this exercise is to reduce the explosion in the number of classes (*boilerplate code*) as new ways of creating products emerge from the requirements. In this case we create a single factory class, thus eliminating the need for a factory class hierarchy.

Answer the following questions:

a) Create a *single* class that is an explicit *composition* of factory methods; the latter methods should be implemented using `std::function` or `boost::factory` (the choice is yours). Pay attention to the input arguments and return type of the methods.
b) Instantiate the factory methods in part a) by lambda functions, function objects and *if there time* using `std::bind` to reuse the original code from the GOF factory code that you have already created.
c) Brainstorming question: consider the suitability of this design to other pattern such as *Strategy* and *Command*? How does the new design compare with the corresponding GOF design?
d) Brainstorm on how to integrate existing OOP code using lambda functions (using wrappers, for example) and `std::bind`. In particular, investigate how to modify objects using captured variables after they have been default initialised using GOF factories. A typical code snippet is:

```
Pointer<ProductFactoryModel_A> factory(new ProductFactoryModel_A());
short val = 3;
//auto facP1= [val](){ return Pointer<P1>(new P11(val)); };
auto facP1 = [&val, &factory]() {
auto p1 = factory->CreateP1();
p1->data = val; return p1; };
```

3. (Integrating *Template Method Pattern* and Factories)

A regular occurrence when creating class libraries it that we can discover code and functionality that can be used in other member functions. This is sometimes called *refactoring*.

Answer the following questions:

a) Going back to the original GOF solution, rewrite the code to create a line in terms of creating its endpoints. This *template function* is not abstract and it should be declares as `final`.
b) Create a class that models a polyline (collection of points). Write the factory method for this class. The implementation should use *template template parameter*.
c) Test the code.

4. (Template Method Pattern based on Universal Function Wrappers)

In this exercise we create a generic class that implements a template method with the following steps:

- Input: Generate input data.
- Processing: Call an algorithm that compute with the input data.
- Output: Dispatch the result of the computation to clients.

To scope the problem, we choose the function types corresponding to these steps as follows:

```
// Function categories
template <typename T>
                using FactoryFunction = std::function<T ()>;
template <typename T>
                using ComputeFunction = std::function<T (const T& t)>;
template <typename T>
           using DispatchFunction = std::function<void(T& t)>;
```

The class with the embedded template method is:

```
// Class with Input-Processing-Output
template <typename T>
           class TmpProcessor
{ // No inheritance used

private:
    FactoryFunction<T> _factory;
    ComputeFunction<T> _compute;
    DispatchFunction<T> _dispatch;

public:
           TmpProcessor(const FactoryFunction<T>& factory,
                        const ComputeFunction<T>& compute,
                        const DispatchFunction<T>& dispatch)
: _factory(factory), _compute(compute), _dispatch(dispatch) {}

           // The template method pattern
           virtual void algorithm() final
           {
                  T val = _factory();
                  T val2 = _compute(val);
                  _dispatch(val2);
           }
};
```

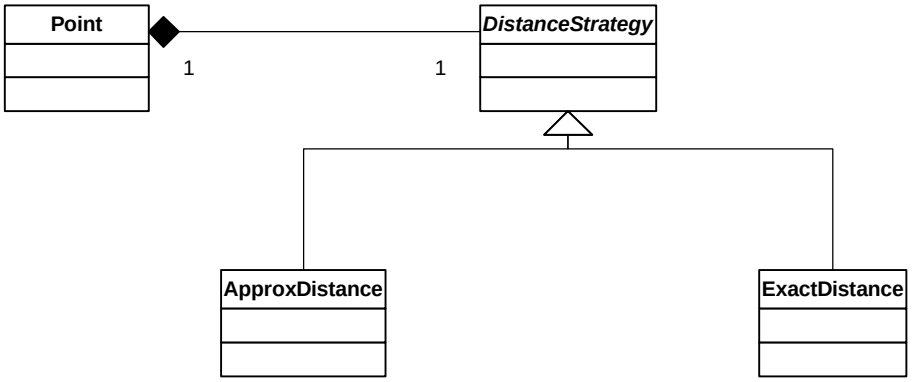
Answer the following questions:

a) Create the code for this class and test it with simple functions (for example, you can use lambda functions).
b) Conjure up a processing that consists of several compute-intensive sub-algorithms that can be run in parallel. Use C++ 11 *futures* to implement the solution. Take a specific example that you are familiar with and first create the *data dependency graph*.

Create a test program in which there are two instances of `TmpProcessor`, one using parallel programming and the other using sequential programming.

5. (Next Generation Strategy Pattern/*Mixins*/*Plugins)*

We return to the GOF Strategy pattern to compute the distance between two points. The related UML2 class diagram is:



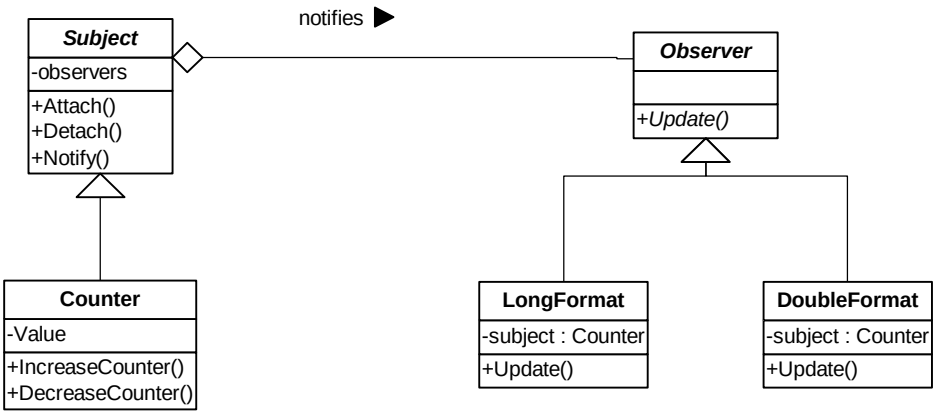
The objective is to use universal function wrappers to achieve the same end without the need to create class hierarchies.

Answer the following questions:

a) Create an interface representing an algorithm that computes the distance between two points (model as a function wrapper and pay attention to its signature).
b) We use the interface in class `Point`. Create both the *stateless* and *state-based* versions of the modified strategy. Test your code using both the Phytagoras and taximan algorithms.
c) Now consider the case when the algorithm needs to hold data (for example, a static counter that is incremented each time the algorithm is called) in-between function calls. How would you combine OOP classes and universal function wrappers to solve this problem?
   If there is time..implement the algorithms as a concurrent task using C++11 *futures*.

6. (Observer Pattern using Universal Function Wrappers)

We return to the GOF Observer pattern. The related UML2 class diagram is:



We now reengineer the design to avoid dependency on the Observer class hierarchy which is too restrictive.

Answer the following questions:

a) Implement the collection of `std::list<std::function<…>>` . Create observers to format counters in `long` and double format, as before.

b) Now implement the list of observers using *template template parameters*. Test the code in part a) using `std::deque`.
c) If time permits... implement this problem using Boost *signals2*.

7. (Concepts)

The goal of this exercise is to use a (deliberately) simplified and hard-coded Monte-Carlo option pricer and modify it with C++20 syntax to make the design more flexible and robust. The source code is given below. It consists of a struct OptionData to hold the data for an option, a class SDE that models *Geometric Brownian Motion (GBM)* and a main() method (call it the *client*) that contains the actual algorithm.

Ultimately, we wish to postpone implementation details and specific decisions for as long as possible with a view to creating highly generic and flexible software.

Answer the following questions:

1) Identify those features in SDE that reduce the flexibility of the code. It contains hard-code data and is essentially limited to GBM models. We note that SDE is a composition of a *diffusion* term and a *drift* term. We wish to support a larger class of SDEs and to this end we apply the *Bridge* pattern based on Concepts.
2) We wish to create an 'abstract' class that is composed of a diffusion term and a drift term and then instantiate it for specific cases (*Bridge* pattern):

```
// Interface contract specification
template<typename T, typename Data>
       concept IDiffusion = requires (T c, Data t, Data x)
       { c.diffusion(t,x); };

       template<typename T, typename Data>
       concept IDrift = requires (T c, Data t, Data x)
       { c.drift(t, x); };

       template<typename T, typename Data>
       concept IDriftDiffusion = IDiffusion<T, Data> && IDrift<T, Data>;

       template <typename T, typename Data> requires IDriftDiffusion<T, Data>
       class SDEAbstract
{ // System under discussion, using composition
  // Really a Bridge pattern

           private:
                  T _t;
           public:
           SDEAbstract(const T& t) : _t(t) {}
           Data diffusion(Data t, Data x)
           {
                  return _t.diffusion(t, x);
           }
           Data drift(Data t, Data x)
           {
                  return _t.drift(t, x);
           }

};
```

Use this model by integrating the code from below into the current structure. Discuss the advantages over that hard-coded solution. Give code examples.

3) Extend the model in part 2) to support GBM with Poisson jumps as well as to PDEs which have convection, diffusion, reaction and homogeneous terms. How would you model PDEs that have no convection terms?

4) Let's say we wish to model multifactor SDEs, for example multi-asset, Heston etc. Can you model these SDEs using parameter packs and variadic parameters?
5) Test the new code.

Code:

```
// TestMC.cpp
//
// C++ code to price an option, essential algorithms.
//
// We take Geometric Brownian Motion (GBM) model and the Euler method.
// We compute option price.
//
// 2012-2-26 Update using std::vector<double> as data storage structure.
// 2016-4-3 DD using C++11 syntax, new example.
// 2017-8-2 DD version 1 for book
//
// (C) Datasim Education BV 2008-2021
//
#include "OptionData.hpp" // in local directory
#include <random>
#include <memory>
#include <cmath>
#include <iostream>
#include "StopWatch.cpp"
class SDE
{ // Defines drift + diffusion + data
private:
std::shared_ptr<OptionData> data; // The data for the option
public:
SDE(const OptionData& optionData) : data(new OptionData(optionData)) {}
double drift(double t, double S)
{ // Drift term
return (data->r - data->D)*S; // r - D
}
double diffusion(double t, double S)
```

```
{ // Diffusion term
return data->sig * S;
}
};

int main()
{
std::cout << "1 factor MC with explicit Euler\n";
OptionData myOption{ 65.0, 0.25, 0.08, 0.3, 0.0, -1 };
/* myOption.K = 65.0;
myOption.T = 0.25;
myOption.r = 0.08;
myOption.sig = 0.3;
myOption.D = 0.0;
myOption.type = -1; // Put -1, Call +1*/
/*myOption.K = 100.0;
myOption.T = 1.0;
myOption.r = 0.0;
myOption.sig = 0.2;
myOption.D = 0.0;
myOption.type = -1; // Put -1, Call +1
*/
SDE sde(myOption);
// Initial value of SDE
double S_0 = 60;
// Variables for underlying stock
double x;
double VOld = S_0;
double VNew;
long NT = 100;
std::cout << "Number of time steps: ";
std::cin >> NT;
// V2 mediator stuff
long NSIM = 50000;
```

```
std::cout << "Number of simulations: ";

std::cin >> NSIM;

double M = static_cast<double>(NSIM);

double dt = myOption.T / static_cast<double> (NT);

double sqrdt = std::sqrt(dt);

// Normal random number

double dW;

double price = 0.0; // Option price

double payoffT;

double avgPayoffT = 0.0;

double squaredPayoff = 0.0;

double sumPriceT = 0.0;

// #include <random>

// Normal (0,1) rng

std::default_random_engine dre;

std::normal_distribution<double> nor(0.0, 1.0);

// Create a random number

dW = nor(dre);

StopWatch<> sw;

sw.Start();


for (long i = 1; i <= M; ++i)

{
// Calculate a path at each iteration

if ((i/100'000) * 100'000 == i)

{
// Give status after each 10000th iteration

std::cout << i << ", ";

}


VOld = S_0;

x = 0.0;

for (long index = 0; index <= NT; ++index)

{

 // Create a random number
```

```
 dW = nor(dre);

  // The FDM (in this case explicit Euler)

    VNew = VOld + (dt * sde.drift(x, VOld)) + (sqrdt * sde.diffusion(x, VOld) *
dW);

    VOld = VNew;

    x += dt;

}

// Assemble quantities (postprocessing)

    payoffT = myOption.myPayOffFunction(VNew);
    sumPriceT += payoffT;
    avgPayoffT += payoffT/M;
    avgPayoffT *= avgPayoffT;

    squaredPayoff += (payoffT*payoffT);
}

// Finally, discounting the average price

price = std::exp(-myOption.r * myOption.T) * sumPriceT/M;

std::cout << "Price, after discounting: "

<< price << ", " << std::endl;

double SD = std::sqrt((squaredPayoff / M) - sumPriceT*sumPriceT/(M*M));

std::cout << "Standard Deviation: " << SD << ", " << std::endl;

double SE = SD / std::sqrt(M);

std::cout << "Standard Error: " << SE << ", " << std::endl;

sw.Stop();

std::cout << "Elapsed time in seconds: " << sw.GetTime() << '\n';

return 0;

}
```

8. (Modules)

Discuss the following advantages of using *C++ modules*:

1) Modular design and *Single Responsibility Principle* (SRP).
2) C++ modules and header files living side by side.
3) Support for Information Hiding.
4) Less dependencies than using header files.

Create a mini-application containing three modules for Input, Processing and Output as well as a central Mediator. The code should show the application of the above features. You can take one of the sample cases from the videos.