Object-Oriented Software Design for Financial Libraries

Andrea Pallavicini

`a.pallavicini@imperial.ac.uk`

Financial Engineering, Banca IMI

Imperial College, Dept. of Mathematics. London

Corso Alta Formazione Finanza Quantitativa
Milano 2014-15

The opinions expressed here are solely those of the authors and do not represent in any way those of their employers.

Talk Outline

Imperative Programming Languages – I

- An imperative programming languages performs computations in terms of statements that directly change program states, or data-fields.
- The term is used in opposition to declarative programming, which expresses what the program should accomplish without prescribing how to do it in terms of sequences of actions to be taken.

  $\longrightarrow$ Logical programming (SQL) is an examples of declarative approach.

- The hardware implementation of almost all computers is imperative; nearly all computer hardware is designed to execute machine code, which is native to the computer, written in the imperative style.

Imperative Programming Languages – II

- Most of high-level languages are imperative: Matlab, Basic, Fortran, C, C++, Java, ...
- Some of them allows also for functional support by means of additional libraries: C++ with Boost.Lambda.
- Yet, different paradigms may be adopted:
  - $\longrightarrow$ an imperative approach consists in a linear sequence of commands accessing global variables;
  - $\longrightarrow$ a procedural approach allows for calls to subroutines;
  - $\longrightarrow$ an object-oriented approach focuses on data instead of functions.
  - $\longrightarrow$ a functional approach treats computation as the evaluation of mathematical functions avoiding states and mutable data.
  - $\longrightarrow$ an event-drive approach focuses on possible events whose occurrence determines the program flow.
  - $\longrightarrow$ . . .

References and Tutorials for C++

- Here, some references for C++ programming language.
    - $\longrightarrow$ C++ ISO standard. Available at `www.open-std.org`.
    - $\longrightarrow$ Stroustrup, A. (2000). The C++ Programming Language. Addison-Welsey Professional Computing Series.
    - $\longrightarrow$ Alexandrescu, A. (2001). Modern C++ Design: generic programming and design patterns applied. Addison-Welsey Professional Computing Series.
    - $\longrightarrow$ BOOST C++ Libraries. Available at `www.boost.org`.
- Here, some tutorials
    - $\longrightarrow$ Eckel, B. (2000,2003). Thinking in C++. Avalaible at author's homepage at `www.mindview.net`.
    - $\longrightarrow$ Information, C++ documentation, library reference, forum at `www.cplusplus.com`.
    - $\longrightarrow$ C and C++ programming at `www.cprogramming.com`.
    - $\longrightarrow$ The C/C++ Journal at `www.drdobbs.com`.
    - $\longrightarrow$ Information and forum at `www.codeguru.com`.

Squaring the Circle – I

- We start by considering a simple program written by using an imperative approach.
  - $\longrightarrow$ Which is the side of a square whose area is equal to the area of a circle of given radius ?
  $$\ell^2 \doteq r^2\pi \;\Rightarrow\; \ell = r\sqrt{\pi}$$

- We first try a Matlab code...

```
1 square_root_of_pi = 1.77245385;
2 radius = 1.0;
3
4 side = radius * square_root_of_pi;
5
6 sprintf('side is %f',side)
```

Squaring the Circle – II

- ...however a C implementation is quite similar, but we must take care of strong typing...

```c
#include <stdio.h>

int main(void)
{
  const double square_root_of_pi = 1.77245385;
  const double radius = 1.0;
  double side;

  side = radius * square_root_of_pi;

  printf("side is %f",side);
}
```

Squaring the Circle – III

- ...and also a C++ implementation has only few sugar on I/O facilities.

```cpp
#include<iostream>

int main(void)
{
  const double square_root_of_pi = 1.77245385;
  const double radius = 1.0;
  double side;

  side = radius * square_root_of_pi;

  std::cout << "side is " << side << std::endl;
}
```

Programming Paradigms

- Programming in C++ is not simply *translating* a Matlab or C code into a new language.
- Programming languages give us only a syntax to express models, but they do not make statements on how we describe the world we model (programming paradigm).
  - $\longrightarrow$ Some languages are best suited to focus on procedures (Matlab or C), other on data (C++).
  - $\longrightarrow$ We could force Matlab to handle objects (see for instance DataFeed module).
- Yet, learning C++ to write procedural codes is useless: you can stick on C and probably do a better job.
- From a historical perspective we see the development of programming paradigms from procedures to data (objects) or functions.

Some Comments

- Comments are required in a program.
  - $\longrightarrow$ Without them programs are next to impossible to debug, maintain, or enhance.
  - $\longrightarrow$ Write comments before or while you write the program, never after.
  - $\longrightarrow$ See bringing semantics into documentation.
- C++ is strong typed so that we must take care of types.
  - $\longrightarrow$ Compare 1.0/2.0 and 1L/2L.
- Why are real numbers called floating point numbers ?
  - $\longrightarrow$ See accuracy of floating point arithmetic.
  - $\longrightarrow$ See limits on floting point arithmetic.
- Why should we declare variables as constants ?
  - $\longrightarrow$ The primary purpose of constness is to provide documentation and prevent programming mistakes.
  - $\longrightarrow$ More later on when speaking of functions.

Starting from Procedures

- A procedural program is something like that (compare with previous code examples):
    1. set some variable for input and reserve some variable for output;
    2. manipulate such variables by means of algorithms (procedures);
    3. read from output variables the results.

- Examples of languages used for procedural programming are: Matlab, Basic, Fortran, C.

- Procedural programming paradigm is easy to use, but it is more suited for small projects, or for projects very sensitive to performances (since translating into imperative-style machine code can be better accomplished).
    $\longrightarrow$ Many languages (e.g. C++) allow object-oriented style along with procedural style.

Starting from Objects – I

- An object-oriented program starts from stating that a program is a model for a real-world problem:
  1. translate model's concepts and semantics into program's objects along with their attributes (data) and mutual relationships;
  2. associate algorithms (methods) to each object to access or modify its content;
  3. ask the objects for the results.

- Examples of languages used for object-oriented programming are: C++, C#, Java, Python.

Starting from Objects – II

- Object-oriented programming paradigm requires a good software design.
  - $\longrightarrow$ Which concepts are relevant to model our real-world problem ?
  - $\longrightarrow$ How to combine them ?

- Software design is a time consuming activity, but it is a fundamental one, but for very small projects.

- Often poor designed object-oriented software is difficult to use or to understand, and it could be much slower than a procedural program.

Starting from Functions – I

- A functional program is like a mathematical formula:
  1. list all operations (functions) the program shall perform avoiding references to external variables;
  2. glue such functions to obtain the program flow, and eventually a unique function;
  3. evaluate the function representing your program to get the the results.

- Examples of languages used for functional programming are: R, Excel, Erlang, XSLT.

Starting from Functions – II

- Functional programming paradigm allows for elegant solutions and it is well suited for very specific jobs.
- Medium-size or bigger projects may result complex to approach and manage if written in a pure functional style (think of them as a unique mathematical formula).
  - $\longrightarrow$ Some languages (e.g. C#, C++) allow a mix of object-oriented and functional style.

Procedural Programming Paradigm – I

- We start a tour of procedural programming style to illustrate its pros and cons by extending our squaring-the-circle example to use procedures.

- How to break the code into procedures is already a software design problem.

- We consider two different perspectives, and we show how procedural programming may natural lead to objects.

Procedural Programming Paradigm – II

- If we add subroutines, our squaring-the-circle example becomes

```
1  #include<iostream>
2
3  double squaring_the_circle(double radius);
4
5  int main(void)
6  {
7    const double radius = 1.0;
8    double side;
9
10   side = squaring_the_circle(radius);
11
12   std::cout << "side is " << side << std::endl;
13 }
```

Procedural Programming Paradigm – III

- where the subroutine performing the calculation is given by

```
1 double squaring_the_circle(double radius)
2 {
3    const double square_root_of_pi = 1.77245385;
4    double side;
5
6    side = radius * square_root_of_pi;
7
8    return side;
9 }
```

Procedural Programming Paradigm – IV

- The advantages of the procedural approach may be listed as following
  1. Modularity: inputs are usually specified syntactically in the form of arguments and the outputs delivered as return values.
  2. Scoping: preventing the procedure from accessing the variables of other procedures, including previous instances of itself, without explicit authorization.
  3. Re-use: simple self-contained interfaces allows procedures being a convenient vehicle for making pieces of code written by different people or different groups, including through programming libraries.
- The main side effects are
  1. Overhead: many procedure calls may downgrade performances.
  2. Design: data structures are not directly modelled, but their definitions should be inferred from subroutines' semantics.

Some Comments

- Variables within a function are created, initialized, and destroyed each time the function is called.
  - $\longrightarrow$ The keyword "static" can be used to promote the variable to be permanent. Avoid it !
  - $\longrightarrow$ See later on the use of singletons.
- C++ functions support default values for variables.
- C++ functions may be "inline" to boost up performances.

Emerging Objects – I

- In the previous example, when coding the squaring-the-circle procedure, we can upset our strategy and focus on structures instead of procedures.
- Let us try to *describe* the problem of squaring the circle and not to give an operative solution:
  1. consider a square and a circle (both are well defined mathematical entities);
  2. from elementary geometry we know how to calculate their elements;
     $\longrightarrow$ Define `side_of_square` and `area_of_circle` procedures.
  3. solve the problem of finding a square whose area is the same of a circle of given radius.

Emerging Objects – II

- Thus, we can write the following two procedures

```
1  double side_of_square(double area)
2  {
3    double side;
4    side = sqrt(area);
5    return side;
6  }
7
8  double area_of_circle(double radius)
9  {
10   const double pi = 3.14159265;
11   double area;
12   area = radius * radius * pi;
13   return area;
14 }
```

Emerging Objects – III

- Hence, our main program becomes

```cpp
#include<cmath>
#include<iostream>

double side_of_square(double area) {...}
double area_of_circle(double radius) {...}

int main(void)
{
  const double radius = 1.0;
  double side,area;
  area = area_of_circle(radius);
  side = side_of_square(area);

  std::cout << "side is " << side << std::endl;
}
```

Object-Oriented Programming Paradigm – I

- The concept of a square and of a circle are implicitly represented by their behaviour, namely by the area and side procedures.
- However, it is easier (and often more natural) introducing objects by explicit definitions.
  - $\longrightarrow$ Object-oriented programming paradigm allows for explicit object definitions.
- Objects may be considered as custom types of the language.
  - $\longrightarrow$ In our example we may define the types: `Square` and `Circle`.
- Some languages, such as C++ or Java, introduce a simple syntax to associate data and procedures to objects.
  - $\longrightarrow$ For instance, in C++ our previous procedures will be named: `Square::get_side` and `Circle::get_area`.

Object-Oriented Programming Paradigm – II

```
1  class Square
2  {
3  public:
4
5      Square(void) : s(0.0) {}
6
7      void set_side(double side) {s=side;}
8      void set_area(double area) {s=sqrt(area);}
9      double get_side(void) const {return s;}
10     double get_area(void) const {return s*s;}
11
12 private:
13
14     double s;
15 };
```

Object-Oriented Programming Paradigm – III

```
1  class Circle
2  {
3  public:
4
5     Circle(void) : r(0.0) {}
6
7     void set_radius(double radius) {r=radius;}
8     void set_area(double area) {r=sqrt(area/pi);}
9     double get_radius(void) const {return r;}
10    double get_area(void) const {return r*r*pi;}
11
12 private:
13
14    static const double pi;
15    double r;
16 };
17 const double Circle::pi = 3.14159265;
```

Object-Oriented Programming Paradigm – IV

```cpp
#include<cmath>
#include<iostream>

class Square {...}; class Circle {...};

int main(void)
{
   const double radius = 1.0;

   Circle circle; circle.set_radius(radius);
   Square square; square.set_area(circle.get_area());

   std::cout << "side is " << square.get_side()
             << std::endl;
}
```

Some Comments

- When a class is defined four methods are created:
    - $\longrightarrow$ default constructor (unless an explicit constructor is present);
    - $\longrightarrow$ copy constructor;
    - $\longrightarrow$ assignment operator;
    - $\longrightarrow$ destructor;
- When a class is designed we must decide its copy semantics. We have three cases
    - $\longrightarrow$ an instance of the class can be copied;
    - $\longrightarrow$ instances of the class can be moved, but not copied;
    - $\longrightarrow$ there is only one instance which cannot be moved nor copied.
- In C++11 a native support for move semantics is introduced.

Classes, Objects, and All That – I

- An object is a discrete bundle of functions and procedures, often relating to a particular real-world concept.
  - $\longrightarrow$ For instance, a circle, a yield curve, a floating rate note.
- Other pieces of software can access the object only by calling its functions (object interface).
- This paradigm focuses on data rather than processes, with programs composed by autonomous modules (classes), each instance of which (objects) contains the interface (methods) needed to manipulate its own data structure (members).
  - $\longrightarrow$ This is in contrast to procedural programming that focuses on the function of a module, rather than on its own data.
  - $\longrightarrow$ An object-oriented program may thus be viewed as a collection of interacting objects, as opposed to a list of tasks (subroutines) to perform.

Classes, Objects, and All That – II

- The goal of object-oriented paradigm is to manage dependencies within a program by dividing the program into chunks of manageable size, and hiding them behind object interfaces.

- The advantages of procedural programming are pushed further, while allowing for direct modelling of data structures.

- Code bloating and object-calling overhead may be an issue for performances if code is not well designed and implemented.

- Imperative languages allowing for a natural implementation of the object-oriented programming paradigm, such as Java or C++, usually share the following features: encapsulation, inheritance, dynamic dispatch.

Encapsulation – I

- Encapsulation means concealing the functional details of a class from objects that send messages to it.
  - $\longrightarrow$ In our example we use `Circle::set_radius` to set the circle's radius, but we do not access directly to the private data member `r`.
  - $\longrightarrow$ In such a way, if we change the implementation of the object (e.g. we use the area to define the circle instead of its radius), we are not forced to change the part of code interacting with the object (namely the `main` procedure).

Encapsulation – II

```cpp
1  class Circle
2  {
3  public:
4      Circle(void) : a(0.0) {}
5      void set_radius(double radius) {a=radius*radius*pi;}
6      void set_area(double area) {a=area;}
7      double get_radius(void) const {return sqrt(a/pi);}
8      double get_area(void) const {return a;}
9  private:
10     static const double pi;
11     double a;
12 };
13 const double Circle::pi = 3.14159265;
```

Some Comments

- Static member data are created once for the class instead of being created once per instance of the class.
- Also a method can be declared static.
  - $\longrightarrow$ Static methods can only access the static data members.
- Constant member data can be assigned only within a constructor.
  - $\longrightarrow$ Static constant data require a specific syntax.
- Also a method can be declared constant.
  - $\longrightarrow$ Constant methods can only access data members, but not modify them.
- More elaborated encapsulation policy can be achieved with "friend" methods and classes.

Inheritance – I

- Inheritance means extending a base class by defining a derived class as a superset of the states and behaviours of the originating (base) class.

  $\longrightarrow$ In our example we can derive our classes from a common base Shape which represents a generic (bi-dimensional) shape.

```
1 class Shape
2 {
3 public:
4    virtual void set_area(double area) =0;
5    virtual double get_area(void) const =0;
6 };
```

Inheritance – II

```cpp
1 class Square : public Shape
2 {
3 public:
4   virtual void set_area(double area) {s=sqrt(area);}
5   virtual double get_area(void) const {return s*s;}
6   ...
7 };
8
9 class Circle : public Shape
10 {
11 public:
12   virtual void set_area(double area) {r=sqrt(area/pi);}
13   virtual double get_area(void) const {return r*r*pi;}
14   ...
15 };
```

Some Comments

- A abstract class contains one or more pure virtual functions.
  - $\longrightarrow$ Abstract classes are used as interfaces.
  - $\longrightarrow$ Abstract classes may be derived to form abstract hierarchies of interfaces.
- Conversely a concrete class does not contain pure virtual functions.
  - $\longrightarrow$ Concrete classes are used as implementations.
  - $\longrightarrow$ Concrete classes may be derived to form concrete hierarchies of implementations.
- A concrete class (implementation) may derive from an abstract class (interface) to allow clients to use it.
  - $\longrightarrow$ More elaborated patterns can be written by means of "private" or "protected" derivations.
  - $\longrightarrow$ See later on the relationship of composition.

Dynamic Dispatch – I

- Dynamic dispatch occurs when the object itself determines at run time which code gets executed on invoking one of its methods.
  - $\longrightarrow$ In our example, when a method of Shape is invoked, the object checks if it is a Square or a Circle and acts accordingly.
- For instance, to set equal the area of two shapes we can implement the following method:

```cpp
1 class Shape
2 {
3 public:
4   void set_equal_area(const Shape& shape)
5   {
6     set_area(shape.get_area());
7   }
8   ...
9 };
```

Dynamic Dispatch – II

```cpp
1  #include <cmath>
2  #include <iostream>
3
4  class Shape {...};
5  class Square {...}; class Circle {...};
6
7  int main(void)
8  {
9     const double radius = 1.0;
10
11    Circle circle; circle.set_radius(radius);
12    Square square; square.set_equal_area(circle);
13
14    std::cout << "side is " << square.get_side()
15             << std::endl;
16 }
```

Some Comments

- When working with virtual functions in C++, the compiler build a virtual table which is used behind the scenes to help achieve polymorphism.
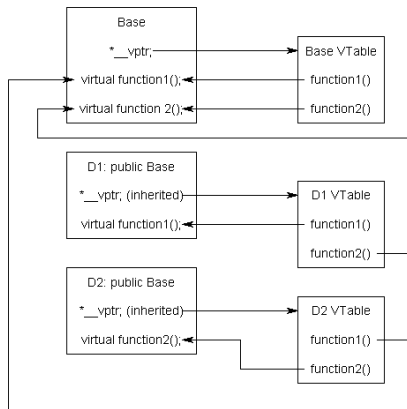
Expressing Objects – I

- Once a language allows for direct design of data structures and for a natural using of object-oriented paradigm, the focus of software engineering shifts towards design rather than implementation.
- A way of expressing objects and their interaction in term of a high level language is attractive.
- A first-choice solution is the Unified Modelling Language (UML), which can be used to specify, visualize, modify, construct and document the artifacts of an object-oriented software-intensive system under development.
  - $\longrightarrow$ UML is widely spread as a design tool in industrial applications.
  - $\longrightarrow$ UML models may be automatically transformed into code skeletons.
- All UML diagrams of this presentation are made with StarUML. Free download at `staruml.sourceforge.net`.

Expressing Objects – II
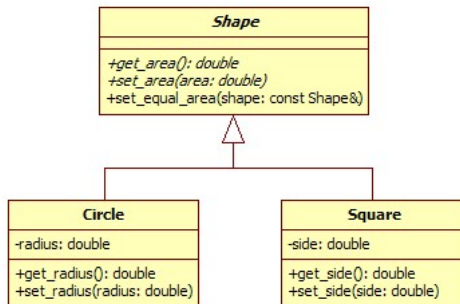
Unified Modelling Language

- UML is a standardized language whose specifics can be found at
  `www.uml.org`.
- UML has many types of diagrams useful for different contexts: data
  modelling, business modelling, object modelling, component
  modelling, etc. . .
  $\longrightarrow$ Here, we focus on class diagrams.
- Class diagrams represent the structure of a system by showing the
  system's classes, their attributes, and the relationships between the
  classes.
- Sequence diagrams represent how processes operate with one another
  and in which order.

UML Class Diagrams – I

- Basic elements of class diagrams are classes and relationships.
- In the class diagram these classes are represented with boxes which contain three parts:
  - $\longrightarrow$ the upper part holds the name of the class (type),
  - $\longrightarrow$ the middle part contains the attributes (data members) of the class,
  - $\longrightarrow$ the bottom part gives the operations (methods) the class can take or undertake

Concrete Type
-data
+public_method() -private_method()

Abstract Type
+public_virtual_method()

UML Class Diagrams – II

- In the class diagram relationships are represented with lines or arrows of different kinds. We focus on some of them.

◇—— Aggregation (A contains B): is a way to combine simple objects or data types into more complex ones, where contained objects may live independently of the container.

◆—— Composition (A is made of B): is a way to combine simple objects or data types into more complex ones, where contained objects cannot live independently of the container.

◁—— Generalization (A extends B): one of the two related classes (the sub-type) is considered to be a specialized form of the other (the super-type).

——→ Association (A uses B): one object instance can cause another one to perform an action on its behalf.

- - -→ Dependency (A modifies B): a change in one object may result in a change to the other one.

Talk Outline

Software Development Methods – I

- Several models exist to streamline the development process (each one has its pros and cons). On the opposite sides we list
  - $\longrightarrow$ Predictive methods (Waterfall, Spiral) focus on planning the future in detail. A predictive team can report exactly what features and tasks are planned for the entire length of the development process. Predictive teams have difficulty changing direction.
  - $\longrightarrow$ Adaptive methods (Agile, XP) focus on adapting quickly to changing realities. When the needs of a project change, an adaptive team changes as well. An adaptive team will have difficulty describing exactly what will happen in the future.
- When developing numerical libraries in a financial engineering team of bank, you should face a rapid varying context and a close contact with end users (traders).
  - $\longrightarrow$ Adaptive methods may be useful in such context.

Software Development Methods – II

	Predictive	Adaptive	Single User
Control	Formal	Internal	User
Time Frame	Long	Short	Short
Users	Many	Few	One
Programmers	Many	Few	One
Docs/Manual	Vital	Limited	None
Robustness	Vital	Vital	Weak

Being Predictive
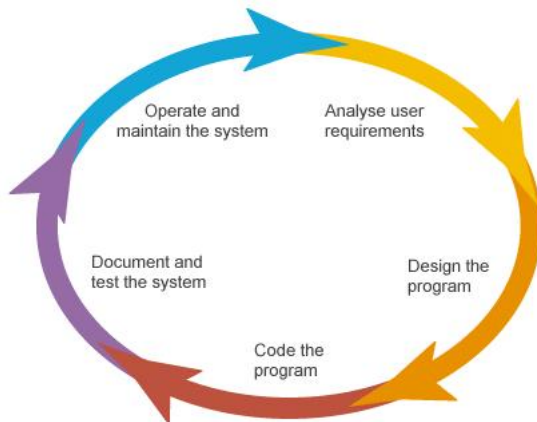
Being Adaptive

Being Quick and Dirty – I

- Let us go back to our squaring-the-circle example.
- Consider to extend the program to allow the user to specify one of the following two actions:
  1. the user gives the radius of a circle and he gets the side of the equivalent square,
  2. the user gives the side of a square and he gets the radius of the equivalent circle.
- We start from an hypothetical program session to get acquainted with the problem, and then we go on UML designing.

```
1 prompt> ./squaring_the_circle circle 6
2 prompt> square side is 10.634723
3
4 prompt> ./squaring_the_circle square 2
5 prompt> circle radius is 1.128379
```

Being Quick and Dirty – II

- A first (quick and dirty) tentative is simply the following

```cpp
#include <cmath>
#include <string>
#include <iostream>

class Shape {...};
class Square {...}; class Circle {...};

int main(int argc, char* argv[])
{
  std::string id;  // equal to "circle" or "square"
  double length;   // either radius or side value

  get_input(argc, argv, id, length);
  ...
```

Being Quick and Dirty – III

```cpp
1    ...
2    if (id=="square")
3    {
4      Square square; square.set_side(length);
5      Circle circle; circle.set_equal_area(square);
6      std::cout << "radius is " << circle.get_radius();
7    }
8    else
9    {
10     Circle circle; circle.set_radius(length);
11     Square square; square.set_equal_area(circle);
12     std::cout << "side is " << square.get_side();
13   }
14
15   std::cout << std::endl;
16 }
```

Being Quick and Dirty – IV

- The preceding example is useful because it shows a relevant design fault:
  - $\longrightarrow$ the `main` procedure behaves differently according to object identifiers without relying on object types.
- Consider to implement different procedures dealing with shapes. Had you follow the same strategy, the switch on object identifiers must be repeated in each procedure.
  - $\longrightarrow$ For instance, adding a new shape requires changing the switch statement of all the procedures.
- In general, clients operating with objects (e.g. the `main` procedure) must operate on such objects only via their interfaces (e.g. the `Shape::set_area` method) without requiring the knowledge of object type.
  - $\longrightarrow$ Object semantics must belong to objects and not be spread across the code.

Some Comments

- There three pre-defined I/O stream: "std::cin" – console input. "std::cout" – console output. "std::cerr" – console error log.
  - $\longrightarrow$ The standard template library headers "<iostream>" and "<iomanip>" describe how to use these streams.
- A call to "getchar()" may be used to stop the program.
- The standard template library header "<fstream>" describe how to manage file I/O.
- Also strings can be used as buffers supporting I/O operations, as described in the standard template library header "<sstream>".

Object Semantics – I

- The key point for a good software design is defining object semantics. In our case we should answer the following questions:
  - $\longrightarrow$ what is a shape ? (data members)
  - $\longrightarrow$ what is it used for ? (methods)
  - $\longrightarrow$ what is its rôle ? (relationships)
- In a squaring-the-circle problem
  - $\longrightarrow$ shapes may be circles or squares (or maybe other equilateral convex polygons);
  - $\longrightarrow$ shapes have two features: a characteristics length (radius, side or apothem) and an area;
  - $\longrightarrow$ the calculation algorithm returns the characteristics length of a shape, given another shape defined, in turn, by its characteristics length.
- We try to define the Shape interface, and we reformulate the remaining code accordingly.

Object Semantics – II

```
1  class Shape
2  {
3  public:
4    virtual double get_area(void) const =0;
5    virtual void set_area(double area) =0;
6    void set_equal_area(const Shape& shape)
7    {
8      set_area(shape.get_area);
9    }
10
11   virtual double get_length(void) const =0;
12   virtual void set_length(double length) =0;
13   void set_equal_length(const Shape& shape)
14   {
15     set_length(shape.get_length);
16   }
17 };
```

Object Semantics – III

```
1 class Circle : public Shape
2 {
3 public:
4
5   Circle(void) : r(0.0) {}
6
7   double get_area(void) const {return r*r*pi;}
8   void set_area(double area) {r=sqrt(area/pi);}
9   double get_length(void) const {return get_radius();}
10  void set_length(double length) {set_radius(length);}
11
12  double get_radius(void) const {return r;}
13  void set_radius(double radius) {r=radius;}
14  ...
15 };
```

Object Semantics – IV

```
1  class Square : public Shape
2  {
3  public:
4
5     Square(void) : s(0.0) {}
6
7     double get_area(void) const {return s*s;}
8     void set_area(double area) {s=sqrt(area);}
9     double get_length(void) const {return get_side();}
10    void set_length(double length) {set_side(length);}
11
12    double get_side(void) const {return s;}
13    void set_side(double side) {s=side;}
14    ...
15 };
```

Object Semantics – V

```
1  #include<cmath>
2  #include<string>
3  #include<iostream>
4
5  class Shape {...};
6  class Square {...}; class Circle {...};
7
8  int main(int argc,char* argv[])
9  {
10    std::string id_from,id_to;
11    double length_from;
12
13    get_input(argc,argv,id_from,length_from,id_to);
14    ...
```

Object Semantics – VI

```
1   ...
2   Shape* shape_from = create_shape(id_from);
3   shape_from->set_length(length_from);
4
5   Shape* shape_to = create_shape(id_to);
6   shape_to->set_equal_area(*shape_from);
7
8   std::cout << "length is " << shape_to->get_length()
9             << std::endl;
10 }
```

- Notice that the create_shape procedure hides the switch on object identifiers, so that the main procedure can deal only with generic shapes, and it does not require modifications if a new type of shape is introduced.

Some Comments

- Abstract classes cannot be handled by value, since they cannot be instantiated.
  - $\longrightarrow$ We must resort to pointers or references to them.
- When calling a method of an abstract class via a pointer or a reference to it, the run-time library of the compiler employs dynamical dispatching to re-direct the call to the method of the proper derived class.
- Some practice is needed to get acquainted with pointer and reference syntax.
  - $\longrightarrow$ More elaborated expressions are introduced in C++11 to cope with move semantics.

Design Patterns – I

- The need of a `main` procedure dealing only with generic shapes results in re-factoring the object creation code into a new `create_shape` procedure.
- Creating objects is a common tasks in many object-oriented applications.
  - When we realize the need for a common task we can resort to design patterns for pre-worked recipes.
- In software engineering, a design pattern is a general re-usable solution to a commonly occurring problem in software design.
  - $\longrightarrow$ A design pattern is not a ready-made solution that can be transformed directly into code. It is only a (working) recipe.

Design Patterns – II

- Some references for design patterns can be found on
  - $\longrightarrow$ Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994).
    Design Patterns: elements of reusable object-oriented software.
    Addison-Welsey Professional Computing Series.
    Hereafter cited as [GoF], namely the Gang of Four.
  - $\longrightarrow$ Buschmann, F., Meunier, R., Rohnert, H., Sommerland, P., Sral, M. (1996-2007).
    Pattern-Oriented Software Architecture. Wiley.
    Hereafter cited as [POSA].
    - Volume I: a system of patterns.
    - Volume II: patterns for concurrent and networked objects.
    - Volume III: patterns for concurrent and networked objects.
    - Volume IV: patterns for concurrent and networked objects.
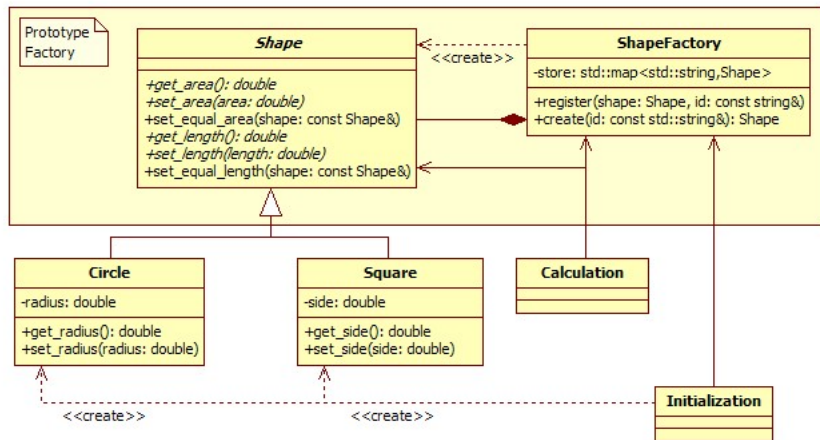
Design Patterns – III

- Design patterns can speed up the development process by providing tested, proven development paradigms.
  - $\longrightarrow$ Further, software documentation is greatly improved by using design patterns, so that learning code is simpler and quicker.
- Yet, be aware of using too complex tools for simple things: design patterns are useful if they bring real advantages.
- In GoF's book we find the Prototype design pattern as a well-suited choice for our need.

  "Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype" [GoF]

Design Patterns in Practice – I

- Since we need prototypes, we can split the `main` procedure into two procedures:
  - $\longrightarrow$ a first one for creating prototypes (once created we can store them into a manager), and
  - $\longrightarrow$ a second one for managing the calculation.
- We introduce as prototype manager a new object: the `ShapeFactory`.
  - $\longrightarrow$ The interface of the `ShapeFactory` should allow for object creation and registration.
- Notice that creating an object seems a simple operation for our example, but in realistic situations it may be a complex task, and starting from a prototype may be very useful.
- Now, we try to visualize all our objects and procedures in a comprehensive UML diagram.

Design Patterns in Practice – II

Design Patterns in Practice – III

```
1  class ShapeFactory
2  {
3  public:
4    Shape* create(const std::string& id) const
5    {
6      std::map<std::string,Shape*>::const_iterator p =
7        store.find(id);
8
9      if (p!=store.end())
10     {
11       Shape* shape = p->second;
12       return shape->clone();
13     }
14     else return 0;
15   }
16 };
```

Design Patterns in Practice – IV

- The shape factory creates new shapes starting from prototypes handled through a pointer to an abstract class.
  - $\longrightarrow$ We need a virtual method to produce a new instance of a given shape. Something we can name a "virtual constructor".

```cpp
class Shape
{
public:
  virtual Shape* clone(void) const =0;
  ...
}
```

Design Patterns in Practice – V

```
1 class Circle
2 {
3 public:
4   Circle* clone(void) const
5   {
6     Circle* circle = new Circle;
7     circle->set_radius(r);
8     return circle;
9   }
10  ...
11 }
```

Design Patterns in Practice – VI

```
1 class Square
2 {
3 public:
4   Square* clone(void) const
5   {
6     Square* square = new Square;
7     square->set_side(s);
8     return square;
9   }
10   ...
11 }
```

Design Patterns in Practice – VII

```cpp
1  #include<cmath>
2  #include<string>
3  #include<iostream>
4  #include<map>
5
6  class Shape {...};
7  class Square {...}; class Circle {...};
8  class ShapeFactory {...};
9
10 void initialize(ShapeFactory& factory);
11
12 double calculate(const ShapeFactory& factory,
13                  const std::string& id_from,
14                  double length_from,
15                  const std::string& id_to);
16 ...
```

Design Patterns in Practice – VIII

```
1  ...
2  int main(int argc,char* argv[])
3  {
4    std::string id_from,id_to;
5    double length_from;
6
7    get_input(argc,argv,id_from,length_from,id_to);
8
9    ShapeFactory factory;
10
11   double lenght_to =
12     calculate(factory,id_from,length_from,id_to);
13
14   std::cout << "length is " << lenght_to << std::endl;
15 }
```

Design Patterns in Practice – IX

```cpp
1  void initialize(ShapeFactory& factory)
2  {
3      factory.insert(new Square,"square");
4      factory.insert(new Circle,"circle");
5  }
```

```cpp
1  double calculate(const ShapeFactory& factory,
2    const std::string& id_from, double length_from,
3    const std::string& id_to)
4  {
5    Shape* shape_from = factory.create(id_from);
6    shape_from->set_length(length_from);
7    Shape* shape_to = factory->create(id_to);
8    shape_to->set_equal_area(*shape_from);
9    return shape_to->get_length();
10 }
```

Some Comments

- The standard template library defines a wide range of containers
    $\longrightarrow$ The proper container to use strongly depends your needs.
- Sequence containers: "<vector>", "<deque>", "<list>".
    $\longrightarrow$ C++11 additions: "<array>", "<forward_list>".
- Associative containers: "<set>", "<map>", "<multiset>", "<multimap>".
    $\longrightarrow$ C++11 additions: "<unordered_set>", "<unordered_map>", "<unordered_multiset>", "<unordered_multimap>".
- The standard template library containers can be accessed by means of iterators.

Exploiting the Object-Oriented Paradigm – I

- The most important lesson we learned is that you do not reap benefits from object-oriented programming automatically.
- The benefits result from a careful design along with implementation practices that exploit the object-oriented approach.
- Object-oriented programming allows for re-usability, reduces software maintenance, and improves productivity.
  - $\longrightarrow$ Yet, bad coding is a simpler activity, than exploiting object-oriented programming features.

Exploiting the Object-Oriented Paradigm – II

- Now, we take a further look at our square-the-circle example to understand the relationship between software design and development process.

- We recap the development process we followed for our square-the-circle project to analyze which parts of code are preserved, and which are modified.

Development Process of Squaring-the-Circle Project

- We receive some specifics: square the circle.
- We start a project and a release cycle.
- First development cycle.
  - $\longrightarrow$ Analyze user requirements: we start with prototyping to investigate our modelling framework.
  - $\longrightarrow$ Design the program: we isolate `Circle` and `Square` as relevant objects.
  - $\longrightarrow$ Code the program: we re-write our prototypes to deal with objects via `Shape` interface.
- Second development cycle.
  - $\longrightarrow$ Analyze user requirements: we discuss and extend specifics by introducing polygons with equivalent area.
  - $\longrightarrow$ Design the program: we group our objects to form the Prototype Factory pattern.
  - $\longrightarrow$ Code the program: we implement and release the first program (Hooray!).
- We close the first release cycle, and so on...

Interface and Implementation – I

- Once object semantics are defined according to the real world problem to be modelled (definition of object interfaces), we were able to face a new set of specifics (re-factoring).
- If our analysis of the real world problem was accurate, and new specifics refine it without changing the world, we can consider object interfaces stable during the life of our project.
  - $\longrightarrow$ If a new shape would be required, we can simply implement a new Shape derived class without affecting the program structure.

Interface and Implementation – II

- In the very first stages of development the object interfaces can change due to code re-factoring, which is good if required by a better comprehension of specifics, but it should not be a common activity.
- Class interfaces (and their relationships) should be defined in the first stages of the analysis procedure.
  - $\longrightarrow$ Listing names pertaining to the world to be modelled may be a good starting point.
  - $\longrightarrow$ Avoid names pointing to actions, and stick to concrete objects.

Interface and Implementation – III

- Object relationships should be stable and not ambiguous, namely they should avoid to form closed loops.
  - $\longrightarrow$ If a class has a complex behaviour, it could be useful either to split it or to manage it through different simpler interfaces (bridge design pattern from GoF).
  - $\longrightarrow$ If a class cannot be modified and we want to re-use in a different context, we can resort to a helper class adapting the new interface (adapter design pattern from GoF).

Interface and Implementation – IV

- As long as a project enters the production stage, its interfaces should freeze.
- In general, interfaces should evolve to be small, simple, and stable.
  - $\longrightarrow$ A re-usable class should be open for extension, but closed for modification
  - $\longrightarrow$ An instance of a derived class should be able to replace any instance of its superclass.
  - $\longrightarrow$ The dependency of one class to another one should depend on the smallest possible interface.

Input and Output Data-Flow – I

- When implementing a simple program, input data are usually hard-coded into the `main` procedure or are requested at command execution from the shell (in C by means of `scanf` function, in C++ by means of `std::cin` stream).

- Yet, programs may need very complex data structures, or they could not communicate with an input shell. Financial numerical libraries have both this problems.

- Financial data are complex and usually resides in databases (a simple file, or a view of relational database).

  $\longrightarrow$ A user cannot feed the application with a bond term-sheet each time he requests a price.

- Financial libraries are usually invoked within other applications (e.g. Excel, or book-keeping applications).

  $\longrightarrow$ An algorithm requiring data cannot call back the user for additional input.

# Input and Output Data-Flow – II
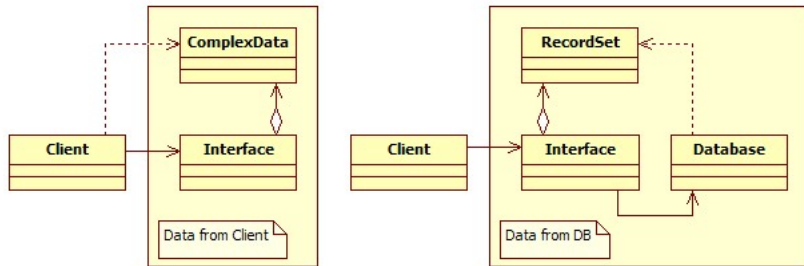
Static Data and Services – I

- Consider our example on squaring the circle. The shape factory is created at the beginning of the `main` procedure. Then, it is used both by the initialization and calculation procedures.
- When a program is transformed in a library, we strip the `main` procedure leaving in place all other functionalities.
  - $\longrightarrow$ Indeed, the `main` procedure is represented by the client calling the library (and we cannot control it).
- Thus, we have two possibilities:
  - $\longrightarrow$ requiring the client defines all the data structures needed by the library, and he passes them as arguments of library functions (as additional input data); or
  - $\longrightarrow$ allowing for a static variables which persists between library calls (usually implemented as singletons, see GoF).

Static Data and Services – II

- Singletons are unique (static) instances of classes which can be accessed by any library object. They represents services offered by the library to its objects.

- Singletons are created the first time a service access them, and they are destroyed when the library is ended.
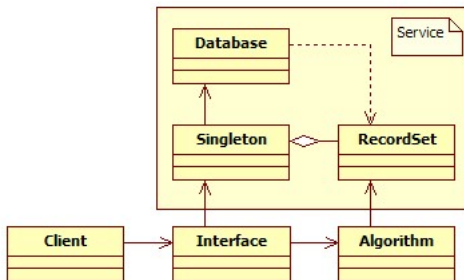
Static Data and Services – III

```
1  class ShapeFactory
2  {
3  public:
4    static ShapeFactory& instance(void)
5    {
6      if (!factory) factory = new ShapeFactory;
7      return *factory;
8    }
9
10   void insert(Shape shape,const std::string& id) {...}
11   Shape create(const std::string& id) const {...}
12
13 private:
14   static ShapeFactory* factory = 0;
15   ...
16 };
```

Static Data and Services – IV

```cpp
1 void initialize(void)
2 {
3   ShapeFactory::instance().insert(Square());
4   ShapeFactory::instance().insert(Circle());
5 }
6 double calculate(
7   const std::string& id_from, double length_from,
8   const std::string& id_to)
9 {
10  Shape shape_from =
11    ShapeFactory::instance().create(id_from);
12  shape_from.set_length(length_from)
13  Shape shape_to =
14    ShapeFactory::instance().create(id_to);
15  shape_to.set_area(shape_from)
16  return shape_to.get_length();
17 }
```

Static Data and Services – V

```
1  ...
2  int main(int argc, char* argv[])
3  {
4    std::string id_from, id_to;
5    double length_from;
6
7    get_input(argc, argv, id_from, length_from, id_to);
8
9    initialize();
10
11   double lenght_to =
12     calculate(id_from, length_from, id_to);
13
14   std::cout << "length is " << lenght_to << std::endl;
15 }
```

Some Comments

- The Loki library by A. Alexandrescu contains a good implementation of the Singleton pattern.
- More complex implementations should include the use of the Boost library which contain a wide range of well tested solutions and concepts along with a complete documentation.
  - $\longrightarrow$ Check this library before attempting to implement any generic code.
- Before coding, remember the following guidelines:
  - $\longrightarrow$ the program should work;
  - $\longrightarrow$ the program should be cheap to make;
  - $\longrightarrow$ the program should be as easy to use as possible.

Talk Outline

1. Programming Paradigms

2. Software Design

3. Financial Examples
   - The Monte Carlo Pricing Algorithm
   - Problem Analysis and Code Design
   - Design Patterns
   - Further Developments

Monte Carlo integration

- Consider a discounted payoff $\Pi(t, T; x)$ which depends on a random variable x, for instance the stock price at maturity. If we know the probability distribution of x we can calculate the price as follows

$$\mathbb{E}_t[\Pi(t, T; x)] \simeq \frac{1}{n} \sum_{i=1}^{n} \Pi(t, T; x_i)$$

where x_i is the i-th realization of the random variable and n is the number of realizations. The approximation holds better for larger values of n.

- We can estimate the error we have when approximating the true expectation by its Monte Carlo value ?

Monte Carlo Error Estimate – I

- Let us view $\Pi(t, T; x_i)$ as a sequence of independent identically distributed random variables, distributed as $\Pi(t, T; x)$.

- If their second momentum is finite, by the Central Limit Theorem we get

$$\lim_{n \to \infty} \frac{1}{\sqrt{n}} \sum_{i=1}^{n} \frac{\Pi(t, T; x_i) - \mathbb{E}_t[\Pi(t, T; x)]}{\mathrm{Std}_t[\Pi(t, T; x)]} \sim \mathcal{N}(0, 1)$$

- This holds since the addends, call them y_i, have zero mean, unit standard deviation and are iid, leading to

$$\mathbb{E}\left[e^{\frac{it}{\sqrt{n}} \sum_i y_i}\right] = \prod_{i=1}^{n} \mathbb{E}\left[e^{it \frac{y_i}{\sqrt{n}}}\right] = \left(1 - \frac{t^2}{2n} + o\left(\frac{t^2}{n}\right)\right)^n \longrightarrow e^{-\frac{1}{2}t^2} = \varphi_{\mathcal{N}}(t)$$

Monte Carlo Error Estimate – II

- We may write approximately and for large n

$$\mathbb{E}_t[\Pi(t, T; x)] \simeq \frac{1}{n} \sum_{i=1}^{n} \Pi(t, T; x_i) + \frac{\epsilon}{\sqrt{n}} \operatorname{Std}_t[\Pi(t, T; x)]$$

where ϵ is a standard normal random variable.

- The payoff standard deviation can be approximated in turn by estimating it from the sample standard deviation

$$\operatorname{Std}_t[\Pi(t, T; x)] \simeq \sqrt{\frac{1}{n-1} \left(\sum_{i=1}^{n} (\Pi(t, T; x_i))^2 - \frac{1}{n} \left(\sum_{i=1}^{n} \Pi(t, T; x_i) \right)^2 \right)}$$

Monte Carlo Error Estimate – III

- By knowing the error distribution, we can calculate the probability that our Monte Carlo approximation is, within a confidence level c, close to the true expectation.

$$\mathbb{Q}\left\{ \left| \mathbb{E}_t[\Pi(t, T; x)] - \frac{1}{n} \sum_{i=1}^{n} \Pi(t, T; x_i) \right| < \frac{c}{\sqrt{n}} \operatorname{Std}_t[\Pi(t, T; x)] \right\} = 2\Phi(c) - 1$$

- The following table is useful to size the probability associated to commonly used confidence levels

c	$\mathbb{Q}$
1	68.27%
2	95.45%
3	99.73%

Monte Carlo Simulations – I

- As an example consider to use the Monte Carlo algorithm to calculate a call option with the Black and Scholes model.

$$\mathbb{E}_0\big[\,(S_T - K)^+\,\big] =$$
$$\int_{\mathbb{R}} (e^x - K)^+\, p_{\mathcal{N}}(x)dx = \int_0^1 \left(e^{\Phi^{-1}(u)} - K\right)^+ du \simeq$$
$$\frac{1}{n}\sum_{i=1}^n \left(e^{\Phi^{-1}(u_i)} - K\right)^+ + \frac{\epsilon}{\sqrt{n}}\,\mathrm{Std}_0\bigg[\left(e^{\Phi^{-1}(u_i)} - K\right)^+\bigg]$$

  where u_i are drawn from a uniform distribution on the unit interval.
- What happens if we do not know the probability distribution of the underlying assets in closed form ?

Monte Carlo Simulations – II

- As an example consider again to use the Monte Carlo algorithm to calculate a call option with the Black and Scholes model.

$$d \log S_t = d \log F_0(t) - \frac{1}{2}\sigma^2 dt + \sigma dZ_t$$

where $F_0(t)$ is the stock forward price, σ is the stock price volatility and Z_t is a Brownian motion.

- We can simulate each stock price path up to option maturity date T by a discrete approximation, each step of size Δt given by (ϵ_t are iid standard normal variables)

$$\log S_{t+\Delta t} = \log S_t + \log \frac{F_0(t+\Delta t)}{F_0(t)} - \frac{1}{2}\sigma^2 \Delta t + \sigma\sqrt{\Delta t}\epsilon_t$$

Monte Carlo Simulations – III

- The Monte Carlo algorithm now becomes

$$\mathbb{E}_0\big[(S_T - K)^+\big] \simeq \frac{1}{n}\sum_{i=1}^{n}(s_i - K)^+ + \frac{\epsilon}{\sqrt{n}}\,\mathrm{Std}_0\big[(S_T - K)^+\big]$$

where s_i is the simulated value of stock price at maturity T along the i-th path, with $0 < \Delta t < 2\Delta t < \cdots < T$, obtained according to

$$\log S_{t+\Delta t} = \log S_t + \log \frac{F_0(t+\Delta t)}{F_0(t)} - \frac{1}{2}\sigma^2\Delta t + \sqrt{\Delta t}\epsilon_t$$

A Matlab Prototype

- We start with a prototype in Matlab to better understand the problem of Monte Carlo pricing.
  - $\longrightarrow$ We consider interest rates equal to zero.
  - $\longrightarrow$ We focus on Black ans Scholes model.
  - $\longrightarrow$ We consider for the prototype's payoff a call option.
- Here, there is the link to the code: `black.m`
- The code contains three functions to evaluate call price analytically, or with a Monte Carlo integration, or with a simulation.

```
1 function price = call(S,K,vol,T)
2 function [price,error] = mc_int_call(S,K,vol,T,n)
3 function [price,error] = mc_sim_call(S,K,vol,T,n,k)
```

A Vocabulary for Monte Carlo Pricing – I

- Before going on modelling it is important to list the candidates for being objects in our framework.

payoff	random variable	stock price
maturity	probability distribution	price
MC error	expectation	std deviation
normal variable	confidence level	call option
uniform variable	Black model	stock volatility
Brownian motion	MC path	MC engine

A Vocabulary for Monte Carlo Pricing – II

- We can sort the list to differentiate nouns coming from particular contexts. Further,
  - $\longrightarrow$ we strike out nouns or verbs which have not a relevant role in our application, so that they can be expressed in term of plain (not structured) data;
  - $\longrightarrow$ we put verbs in bold face (method candidates) to distinguish them from nouns (object candidates).

~~MC error~~	~~expectation~~	payoff	~~stock price~~
MC path	~~std deviation~~	call option	~~stock volatility~~
MC engine	random variable	**price**	
~~confidence level~~	~~uniform variable~~	~~maturity~~	
~~Brownian motion~~	normal variable		
Black model	~~probability distribution~~		

A Vocabulary for Monte Carlo Pricing – III

- Then, we can group surviving nouns in different semantic area, as a guess for object relationships, as long as method candidates.

	Monte Carlo	Random Variables	Payoffs
Nouns	MC path	random variable	payoff
	MC engine	normal variable	call option
	Black model		
Verbs			price

- Notice that when we write Brownian motion we are considering its approximation via a Euler scheme on the logarithm of stock price.

Designing Objects and Their Relationships – I

- We start from the previous vocabulary and connect each name (class) by drawing UML's relationships.
- Payoffs and random variables can be represented by mens of a generalization:
    - $\longrightarrow$ a call option is a generalization of a payoff.
    - $\longrightarrow$ Normal and uniform random variables are generalizations of random variable's concept.
- Thus, we can link such names by means of a particular UML relationship named *generalization*, and represented as $\Leftarrow\!\!\!\longrightarrow$.

Designing Objects and Their Relationships – II

- The names listed in the Monte Carlo's column are more difficult to model. They represent two levels of description:
  - ⟶ a Monte Carlo's engine is a generalization of a Black model, and
  - ⟶ a Monte Carlo's path is used by the Monte Carlo's engine.
- Which is the relationship between payoffs, random variables and Monte Carlo's classes ?
  - ⟶ answering such question leads to adopt a design pattern.
- Here, we list the possible UML relationships:

◇—	Aggregation	A contains B
◆—	Composition	A is made of B
◁—	Generalization	A extends B
⟶	Association	A uses B
- - ->	Dependency	A modifies B

Designing Objects and Their Relationships – III

- Here, a design proposal for Monte Carlo's code. The Black model's class seems playing a central role.
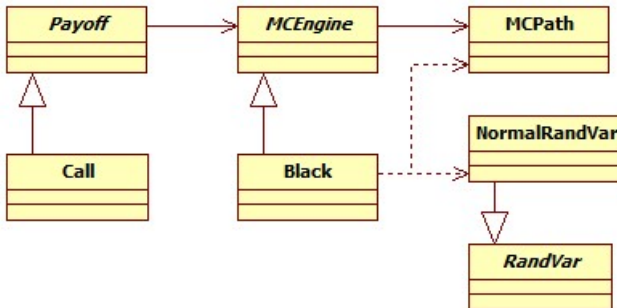
Refining the Design – I

- In our design the Monte Carlo's path is a concrete class created by the Black model and used by the Monte Carlo's engine just to calculate the payoff.
  - $\longrightarrow$ Thus, the Monte Carlo's path seems only an implementation detail, which we could drop from our design.
- On the other hand, Black's dynamics is written in term of stock price's process, namely the Monte Carlo's path,
  - $\longrightarrow$ Thus, it seems reasonable to consider it.
- Notice that the Black model is not simply its dynamics.
  - $\longrightarrow$ The Black model's class checks input market data, and it selects the discretization scheme and the sampling algorithm.
  - $\longrightarrow$ We can add a further class: the particular implementation of Black's dynamics (BlackPath).

Refining the Design – II

- In particular, the implementation of Black's dynamics class (BlackPath) could be introduced as a generalization of the Monte Carlo's path.
  - $\longrightarrow$ Thus, the Monte Carlo's path class is now considered as an abstract class.
- Then, to stress the importance of the Monte Carlo's path class in our design, we can consider it as a part of the Monte Carlo's engine.
  - $\longrightarrow$ Accordingly the pricing algorithm will be changed: first all the paths are generated, then the payoff is evaluated.
  - $\longrightarrow$ Further, with such design we could parallelize the pricing algorithm as we did for Matlab code.

The Bridge Design Pattern – I

The Bridge design pattern decouples an abstraction from its implementation so that the two can vary independently. [GoF]

- Such (structural) design pattern allow to avoid a permanent binding between an abstraction and its implementation. This might be the case, for example, when the implementation must be selected or switched at run-time.
- Both the abstractions and their implementations should be extensible by sub-classing. In this case, the Bridge pattern lets you combine the different abstractions and implementations and extend them independently.
  - $\longrightarrow$ Here, we are decoupling the model's hierarchy (Black, ...) from the hierarchy of their dynamics' implementation (Euler scheme, antithetic sampling, ...).

The Bridge Design Pattern – II

- Here, the proposed design refinement, implementing the Bridge design pattern from GoF book, which allows to decouple Monte Carlo's models from their dynamics (implementation).
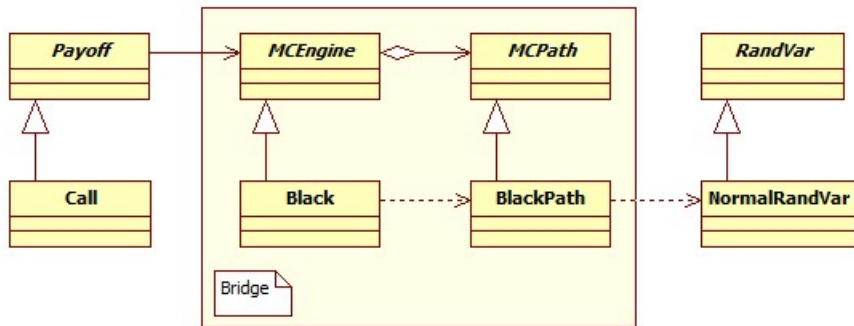
Changing the Payoff

- Our design can accommodate different payoffs (also path dependent) by extending via generalization the payoff class.
- Hints for new developments are
  1. Price an Asian option.
  2. Price a discrete-barrier option.
  3. Add payoffs' abstract factory to manage all the payoffs you created.
- In a more advanced description we could introduce a database to contain all the payoffs, and use a singleton to implement a library service to retrieve payoff definitions from such database.

Changing the Model

- Our design can accommodate different models implemented with different dynamics by extending via generalization the classes of the Monte Carlo's engine and path.
- Hints for new developments are
  1. Use antithetic sampling (change also Monte Carlo's error estimation).
  2. Introduce jumps to obtain a Merton model.
  3. Add models' abstract factory to manage all the models you created.
- In a more advanced description the hierarchy of model's dynamics can be implemented by meas of template policies.