Beyond Object-Orientation: C++ Application Design for Computational Finance – A Defined Process from Problem to Parallel Code

Daniel J. Duffy
Datasim Finance, e-mail: dduffy@datasim.nl

Abstract
In this article we introduce a defined and repeatable process to analyze, design, and implement software systems using a combination of techniques taken from *Structured Analysis* and the object-oriented programming style. We use top-down system decomposition and bottom-up programming techniques to create software systems. We also show how to leverage the new multi-paradigm features in C++ to promote maintainability and extendibility of the resulting code, and how to integrate C++ multi-threading and multi-tasking libraries with our model. As an example to reduce the scope, we introduce a model for Monte Carlo option pricing that we design and realize using several different language features in C++. The steps in the transformation from problem description to code are applicable to other kinds of system (for example, PDE and risk applications), as well as other programming languages (for example, C#). This article represents a continuation of the work in Duffy (2017), Duffy and Palley (2017), and Wilmott, Lewis, and Duffy (2014).

Keywords
computational finance, Monte Carlo simulation, system decomposition, C++11, concurrent and parallel software systems, "how to solve it" (Pólya), reasoning by analogy

Background and introduction
The object-oriented model is more than 40 years old, and it is probably the dominant programming paradigm. Its popularity was due to many factors, some of which were:

- Objects model real-world entities. They "directly model objects of the physical reality to which the software applies" (Meyer, 1997).
- Objects are easy to find ("they are there for the picking" (Meyer, 1997)).
- There was an unfounded belief in the late 1980s that object technology was necessary and sufficient for successful software construction. In particular, we could use the technology as the driver of the software process from the initial requirements elicitation phase to a working program.

In hindsight, these expectations were not fully realized as more organizations adopted this technology. Some general remarks on the current status based on the author's experience and observations are:

- The use of object technology did not necessarily improve the quality of communication between the different project stakeholders. For example, it has little support for techniques such as functional and top-down decomposition and data flow diagrams that are the bread-and-butter activities in *Structured Analysis* (DeMarco, 1978).
- Object technology is more useful in the detailed design phase of the software lifecycle. It is less useful as a tool to break a complex system into loosely coupled subsystems and components.
- Over-reliance on class hierarchies as the "driver" of software projects. Classes can be volatile and unstable, and deep class hierarchies become increasingly difficult to maintain as a software project evolves.
- Many C++-motivated object-oriented design techniques (for example, *software design patterns* (see Gamma, Helm, Johnson, and Vlissides, 1994)) are based on somewhat outdated object technology from the 1990s. The current version is C++11 and it has support for generic and functional programming models, which means that there are opportunities for the creation of next-generation design patterns based on a combination of object-oriented, generic, and functional programming styles.
- It is not clear to the author how traditional object-oriented technology can be used as the driver of system decomposition into concurrent *tasks*. Even in the case of a system that will run in single-threaded mode, we use system decomposition methods rather than initially looking for objects and classes. On a related note, we can imagine that porting single-threaded C++ applications to parallel code will be hugely time-consuming at best, and impossible at worst. It is a dilemma, and the possible choices are to do nothing to the application or to redesign it from scratch.

We now discuss the approach to software design that we take in this article. There are three main phases: first, we need to understand the problem, and then decompose it into subsystems (components), each of which has a major single responsibility; second, we try to pin down and standardize each component's *provides* and *requires* interfaces; third, we design the software internals of each component. There is also

an integration phase in which we test the application. This is part of project management (Boehm, 1986), but this important topic is beyond the scope of this article.

A special and possibly unique feature of the approach taken in this article is that we try to relate the current problem (in this case a Monte Carlo simulator) to a problem that we already know how to solve or that we have solved (this is an approach that mathematicians take all the time (Pólya, 1990)). In our case the Monte Carlo simulator is an example or instantiated system of a more general *resource allocation and tracking* (*RAT*) category, as introduced in Duffy (2004). There are five *domain categories* in all, and each one describes a set of applications sharing the same goals, structure, and behavior. In short, the approach is similar to *reasoning by analogy*.

System scoping and initial decomposition

In general, we try to understand as much of the problem as possible before moving to detailed design. To this end, our goal is to reduce project risk by locating the *boundaries* of the problem. In this way we must know what needs to be developed. Second, we decompose the initial amorphous and monolithic system into loosely coupled and autonomous systems with well-defined interfaces.

We are careful not to fall into the trap of *premature design and implementation*; our main concern here is to understand the main data flow in the system and how the different systems partaking in the context diagram cooperate to realize that data flow.

System context diagram

The first step in understanding, scoping, and documenting a software system is to describe the system (henceforth called the *SUD* (*system under discussion*)) in terms of external systems – where the inputs come from and where the outputs go to.

We define the crucial concepts of *source* and *sink boxes* (DeMarco, 1978):

> A *source* or *sink* is a person, organisation or software system, that lies outside the context of the SUD system and that is a net originator or receiver of system data.

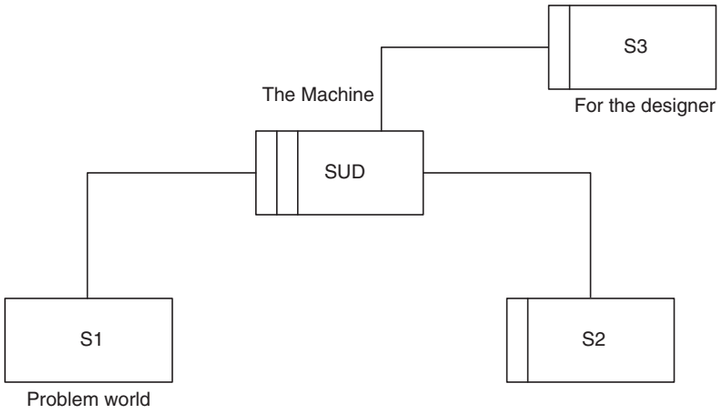
It is obvious that we need to understand what a system's major sources and sinks are before proceeding.

We represent the system context diagram by a drawing in which each system is a rectangular box with lines connecting the SUD with its sources and sinks. Even at this stage, we would like to distinguish between those external systems that must be included in the design (that is, those that must be implemented by us) and those that we do not design and for which we are only interested in the services that they provide or require with respect to the SUD. To this end, we add a single vertical line to those boxes representing the former category and no line for the latter category. The SUD is a special case by definition, and it will be annotated by two vertical lines. In this way we are able to distinguish between the *problem world* (the world outside the computer system) and the *solution world* (which is located in the computer). A generic example of a system context diagram is shown in Figure 1 (using the notation from Jackson, 2001).

Summarizing, we describe the SUD as a *black box* that is surrounded by other systems, which cooperate with the SUD to ensure that the system goals can be achieved. The discovery of the context diagram in applications is very important, because it is a foundation for the discovery of other artefacts such as stakeholders, viewpoints, requirements, and contractual interfaces between the SUD and its *satellite systems*. Furthermore, it is an indispensable tool for project managers who must determine the project size and risk, not to mention monitoring system evolution once the development team has started on the software design.

Arriving at a stable context diagram is an iterative process in general, and it can be achieved by analyzing system requirements, interviewing domain experts, and

Figure 1: Example of context diagram (Jackson approach)



creating software prototypes. It is important to discover the most critical external systems in the early stages of the software process, because they are sources of requirements.

Having created the system context diagram, we now find the data and information exchange between the SUD and its external systems. This is a well-known technique in *Structured Analysis* (DeMarco, 1978), and it is described as a *data-flow diagram* (*DFD*). A DFD is:

> A network representation of a system. The system may be automated. The DFD portrays the system in terms of its component pieces, with all interfaces among the components indicated.

At this early stage we can use DFDs to model the high-level data flow between the SUD and its external systems. It is possible to check whether we have discovered the required systems and their requirements. We also assume that we have a stable context diagram and that we are able to trace the data flow through the system. We now address the issue of determining what the *responsibilities* of each system are. In other words, what does each system need from other systems and what does it deliver to other systems? More precisely, we define a *system service* as a discrete capability or behavior that a system exhibits. A service can be an operation, function, or transformation. It could also be an algorithm, transaction, or a function to monitor external systems and devices. The characteristics of a service are:

- Its name. We can use a verb/noun combination, for example, *CreateTransaction*.
- Its input and output parameters. We define a complete and consistent set of arguments that are syntactically correct and that can be refined during detailed design.
- *Preconditions*: these are the conditions that must be true in order for the service to execute. If a precondition evaluates to false, then the service will not execute.
- *Postconditions*: these are the exit criteria for the service. In general, a postcondition is the state or condition that must be achieved or be valid when the service has finished execution.

We can consider this part of the project to be complete when each service has been scoped, documented, and also assigned to a system. Ideally, each service should be assigned to a single system.

The model example: Monte Carlo simulation

We have examined Monte Carlo option pricing from an object-oriented approach based on software design patterns in Duffy and Kienitz (2009). The essential features of the design consisted of the use of classes and class hierarchies to model mathematical concepts and the application of software design patterns to help promote the extendibility of the application.

In general, the traditional object-oriented approach tends to focus on the creation of classes and class hierarchies in the early stages of software development, even when the problem requirements have not yet been agreed on or understood. Even if the problem is understood, there are a number of consequences associated with this choice. First, it excludes non-object-oriented solutions or solutions that are based on other programming styles. Second, it represents a bottom-up approach and the code tends to become unmaintainable as the code base expands. Third, it is difficult to see how to port object-oriented code to run on multi-processor hardware. Finally, the viewpoint can be taken that the use of objects and classes results in *premature optimization* of code.

The new approach that we adopt is to use system decomposition techniques to break the problem into loosely coupled *logical components* whose black-box interface specifications we wish to identify. In a sense, we postpone having to make implementation decisions for as long as possible. Having successfully created a *logical model*, we can then investigate the different scenarios for a detailed design of the problem. Some choices and special cases are:

- S1: The traditional object-oriented solution based on inheritance and subtype polymorphism.
- S2: The same approach as in case S1, except that we use static polymorphism and the *curiously recurring template pattern* (*CRTP*).
- S3: Using universal function wrappers (the C++11 std::function) to implement polymorphic functions without the need to create class hierarchies.
- S4: Creating a parallel solution using threads, tasks, or message passing (Campbell and Miller, 2011; Williams, 2012).
- S5: A combination of cases S1 to S4.

We give a short overview of some of these design techniques in this article. A complete discussion is given in Duffy (2018).

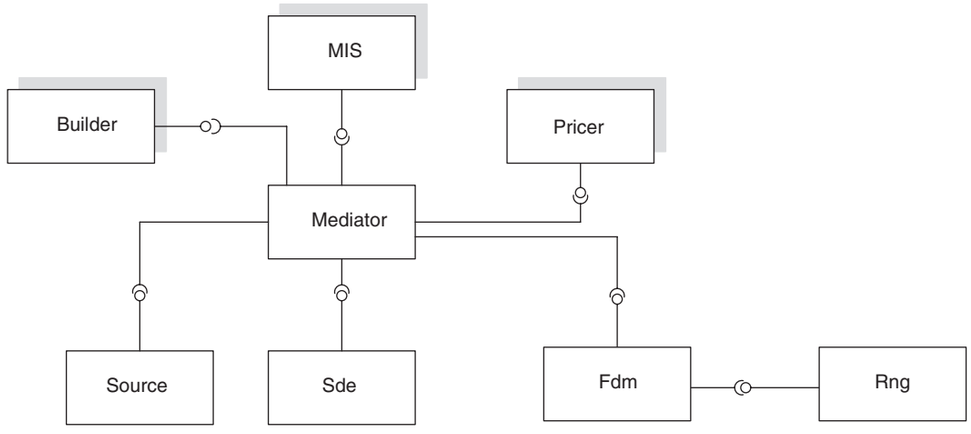
In this article we decompose the system into loosely coupled and cohesive components having well-defined interfaces. The discovery of these components is sometimes by trial and error, and this can be a time-consuming process. We realize that the current application is a special case of a RAT domain category, as discussed in Duffy (2004). The applications in this category share the common characteristic that they process some kind of request and produce a result relating to the status of the request. The best example of a RAT instance is a helpdesk system (everyone understands this problem). The input is a user request and the output is a report (or several reports) describing the status of the request in time and space. For example, a user has placed an order to purchase a book online and she would like to know how long it will take to arrive on her doorstep. In the same vein, we see the Monte Carlo engine as having a similar structure and data flow:

1. Determine the payoff and kind of option to be priced.
2. Choose the stochastic differential equation (SDE) that models the behavior of the underlying asset.
3. Determine how the SDE is approximated by a finite difference scheme.
4. Configure system parameters and define the management system that stores the audit trail data of the running engine.
5. Define the option pricers; for example, it is possible to configure the system to price several kinds of options.
6. Configure the object network using the *Builder* design pattern.

The context diagram for the current application is shown in Figure 2. It is a special case of the context diagram for applications that belong to the RAT category. In general, RAT systems track requests in time and space and they produce the corresponding reports relating to the status of these requests. In this case, the request is to compute the price of one-factor plain, barrier, lookback, and Asian options using the Monte Carlo method. The request is processed in a series of steps to produce the final output by the modules in the context diagram in Figure 2:

- *Source*: The system containing the data relating to the request, for example to price a plain one-factor option with given data. It contains data that is needed by other modules in Figure 2.
- *Sde*: The system that models SDEs. In this case we use geometric Brownian motion (GBM) and its variants. In particular, we are interested in modeling the drift and diffusion of some underlying variable such as the stock price or interest rate, for example.
- *Fdm*: The family of finite difference schemes that approximate the SDEs in the Sde system. In this case we use one-step difference schemes to advance the approximate solution from one time level to the next, until we reach the

Figure 2: New context diagram for Monte Carlo engine



desired solution at expiration. The finite difference schemes require the services of a module that computes random numbers and standard Gaussian variates.

- *Pricer*: This system contains classes to price one-factor options using Monte Carlo simulation. The classes process path information from the *Mediator*, and each pricer class processes this path information in its own way. For example, for a plain option the pricer uses the path data at expiration, uses it to compute the payoff, adds the result to a running total, and then discounts the result to compute the option price. Other path-dependent option types can also be modeled.
- *MIS*: This is the statistics-gathering system that receives status information concerning the progress of computation. For example, this system could display how many paths have been processed at any given time or what the "percentage complete" is.
- *Builder*: This system implements a configuration/creational pattern (based on GOF, 1995) that creates and initializes the systems and their structural relationships in Figure 2.
- *Mediator*: This is the central coordinating entity that manages the data flow and control flow in the system. It is the driver of the system as it were, and it contains the state machine that computes the paths of the Sde. It also informs the other systems of changes that they need to know about. It plays the role of *client* in the *Builder* pattern (GOF, 1995).
- *Rng*: A system to generate random numbers. In this case we use the C++11 <random> library and avoid the use of proprietary libraries.

Designing the application: How do we proceed?

Having created a high-level *design blueprint*, we need to start thinking about a detailed design using a combination of C++ functionality, libraries, and design patterns. In this phase we are really involved in project management, because we build a series of software prototypes, each of which satisfies certain accuracy, efficiency, and maintainability requirements. One approach is to extend the range of components that participate in Figure 2. There are several ways to achieve this end, and we reduce the scope to a discussion of one solution based on the *policy-based design* idiom (Duffy, 2018). It is not necessarily our method of choice, but we include a discussion of it because it contains many of the concepts that we wish to introduce.

Policy-based design: Dynamic polymorphism

In this version we wish to support a range of SDEs, random number generators, and finite difference schemes. We still focus on one-factor plain options. We reduce the risk by not trying to do too much in this cycle, as it would entail having to write and debug a lot of code, even though we do not yet have a stable design.

The goal is to decompose the system into subsystems and in this case we follow one approach taken in Duffy (2018) (policy-based design using templates and private inheritance). In the current case the mediator/SUD class is composed of its template parameters (using shared pointers). The *mediator class* has the form:

```
template <typename Sde, typename Pricer, typename Fdm, typename Rng>
    class SUD : private Sde, private Pricer, private Fdm, private Rng
{ // System under discussion

private:
    // Four main components
    std::shared_ptr<Sde> sde;
    std::shared_ptr<Fdm> fdm;
    std::shared_ptr<Rng> rng;
    std::shared_ptr<Pricer> pricer;

private:
    // Other MC-related data
    int NSim;
    std::vector<double> res;        // Generated path per simulation

public:
    SUD() {}
    SUD(const std::shared_ptr<Sde>& s,
    const std::shared_ptr<Pricer>& p,
    const std::shared_ptr<Fdm>& f,
    const std::shared_ptr<Rng>& r,
    int numberSimulations, int NT)
          : sde(s), pricer(p), fdm(f), rng(r)
    {
       NSim = numberSimulations;
       res = std::vector<double>(NT + 1);
    }

    void start()
    { // Main event loop for path generation

       double VOld, VNew;

       for (int i = 1; i <= NSim; ++i)
       { // Calculate a path at each iteration

          if ((i / 100000) * 100000 == i)
          { // Give status after a given numbers of iterations

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

          VOld = sde->InitialCondition(); res[0] = VOld;

          for (std::size_t n = 1; n < res.size(); n++)
          { // Compute the solution at level n+1

             VNew = fdm->advance(VOld, fdm->x[n - 1], fdm->k,
                          rng->GenerateRn(), rng->GenerateRn());
             res[n] = VNew; VOld = VNew;
          }

          // Send path data to the Pricer(s)
          // This step can be optimised
          pricer->ProcessPath(res);
          pricer->PostProcess();
       }

    }

};
```

We notice the presence of a `start()` method that implements the Monte Carlo path evolver.

In this case we have modeled the classes `Pricer` and `Rng` using subtype polymorphism:

```
class Pricer
{
public:
    // Create a single path
    virtual void ProcessPath(const std::vector<double>& arr) = 0;

    // Notify end of simulation
    virtual void PostProcess() = 0;

    // Discounting, should be a delegate/signal
    virtual double DiscountFactor() = 0;

    // Option price
    virtual double Price() = 0;

    std::function<double (double)> m_payoff;
    std::function<double()> m_discounter;

    Pricer() = default;
    Pricer(const std::function<double(double)>& payoff,
           const std::function<double()>& discounter)
    {
```

```
      m_payoff = payoff;
      m_discounter = discounter;
   }
};

// Pricing Engines
class EuropeanPricer : public Pricer
{
private:
   double price;
   double sum;
   int NSim;

public:
   EuropeanPricer() {}
   EuropeanPricer(const std::function<double(double)>& payoff,
            const std::function<double()>& discounter)
            : Pricer(payoff, discounter)
   {
      price = sum = 0.0; NSim = 0;
   }

   void ProcessPath(const std::vector<double>& arr) override
   { // A path for each simulation/draw

      // Sum of option values at terminal time T
      sum += m_payoff(arr[arr.size() - 1]); NSim++;
   }

      double DiscountFactor() override
      { // Discounting

         return m_discounter();
      }

      void PostProcess() override
      {
         price = DiscountFactor() * sum / NSim;
      }

      double Price() override
      {
         return price;
      }

   };
```

and

```
#include <random>

class Rng
{
public:
   virtual double GenerateRn() = 0;
};

class PolarMarsaglia: public Rng
{ // Only for educational/pedagogical purposes

private:
   std::default_random_engine dre;
   std::uniform_real_distribution<double> unif;
public:
   PolarMarsaglia (): dre(std::default_random_engine()),
      unif(std::uniform_real_distribution<double>(0.0, 1.0)) {}

   double GenerateRn() override
   {

      double u, v, S;

      do
      {
         u = 2.0 * unif(dre) - 1.0;
         v = 2.0 * unif(dre) - 1.0;
         S = u * u + v * v;
      } while (S > 1.0 || S <= 0.0);

      double fac = std::sqrt(-2.0 * std::log(S) / S);
      return u * fac;
   }
};

class CPPRng : public Rng
{ // C++11 versions

private:
   // Normal (0,1) rng
   std::default_random_engine dre;
   std::normal_distribution<double> nor;
public:
   CPPRng() : dre(std::default_random_engine()),
   nor(std::normal_distribution<double>(0.0, 1.0)) {}

   double GenerateRn() override
   {
      return nor(dre);
   }

};
```

In a later version we would need to modify the code to allow *seeding* of these random number generators.

Subtype polymorphism allows us to define various kinds of pricer and random number generator. The classes for SDEs and finite difference methods (FDMs) are hard-coded in this prototype. The pricer hierarchy is set up in such a way that it can be incorporated into a design in which the SUD sends messages to multiple pricers in a transparent manner using Boost *signals2* or the *Asynchronous Agents library* (Campbell and Miller, 2011), for example.

The SDE and FDM classes are:

```
// Instance Systems
class GBM
{ // Simple SDE
private:
   double mu;      // Drift
   double vol;     // Constant volatility
   double d;       // Constant dividend yield
   double ic;      // Initial condition
   double exp;     // Expiry

public:
   GBM() = default;
   GBM(double driftCoefficient, double diffusionCoefficient,
        double dividendYield, double initialCondition,
        double expiry)
   {
      mu = driftCoefficient;
      vol = diffusionCoefficient;
      d = dividendYield;
      ic = initialCondition;
      exp = expiry;
   }

   double Drift(double x, double t) { return (mu - d) * x; }
   double Diffusion(double x, double t) { return vol * x; }

   double DriftCorrected(double x, double t, double B)
   {
      return Drift(x, t) - B * Diffusion(x, t)*DiffusionDerivative(x, t);
   }

   double DiffusionDerivative(double x, double t)
   {
      return vol;
   }

   // Property to set/get initial condition
   double InitialCondition() const
   {
      return ic;
   }


   // Property to set/get time T
   double Expiry() const
   {
      return exp;
   }

};
```

and

```
template <typename Sde>
    class EulerFdm
    {
private:
    std::shared_ptr<Sde> sde;
    int NT;
public:
    std::vector<double> x;          // The mesh array
    double k;                       // Mesh size

    double dtSqrt;
public:
    EulerFdm() = default;
    EulerFdm(const std::shared_ptr<Sde>& stochasticEquation,
             int numSubdivisions)
             : sde(stochasticEquation), NT(numSubdivisions)
    {

       NT = numSubdivisions;
       k = sde->Expiry() / static_cast<double>(NT);
       dtSqrt = std::sqrt(k);
       x = std::vector<double>(NT + 1);

       // Create the mesh array
       x[0] = 0.0;
       for (std::size_t n = 1; n < x.size(); ++n)
       {
          x[n] = x[n - 1] + k;
       }
    }

    double advance(double xn, double tn, double dt,
             double normalVar, double normalVar2) const
    {
       return xn + sde->Drift(xn, tn) * dt
             + sde->Diffusion(xn, tn) * dtSqrt * normalVar;
    }
};
```

In principle, we could have created SDE and FDM class hierarchies for this version, but we have resisted the temptation for the moment due to lack of project time and the fact that they are not yet essential to aid our understanding of the current design.

We give a test case that prices a call option and a put option on the same underlying data. One pricer uses the polar Marsaglia algorithm, while the other uses functionality from the C++11 <random> library:

```
int main()
{
    int NSim = 1'000'000;
    int NT = 500;

    double driftCoefficient = 0.08;
    double diffusionCoefficient = 0.3;
    double dividendYield = 0.0; double initialCondition = 60.0;
    double expiry = 0.25;

    auto sde = std::shared_ptr<GBM>
                (new GBM(driftCoefficient, diffusionCoefficient,
                dividendYield, initialCondition, expiry));

    double K = 65.0;

    // Factories for objects in context diagram
    std::function<double(double)> payoffPut = [&K](double x)
                    {return std::max<double>(0.0, K - x); };
    std::function<double(double)> payoffCall = [&K](double x)
                    {return std::max<double>(0.0, x - K); };
    double r = 0.08; double T = 0.25;
    std::function<double()> discounter = [&r, &T]()
                    { return std::exp(-r * T); };

    auto pricerCall = std::shared_ptr<Pricer>
                (new EuropeanPricer(payoffCall, discounter));
    auto pricerPut = std::shared_ptr<Pricer>
                (new EuropeanPricer(payoffPut, discounter));

    auto fdm = std::shared_ptr<EulerFdm<GBM>>
             (new EulerFdm<GBM>(sde, NT));

    auto rngPM = std::shared_ptr<Rng>(new PolarMarsaglia());
    auto rngCPP = std::shared_ptr<Rng>(new CPPRng());

    StopWatch<> sw;
    sw.Start();

    SUD<GBM, EuropeanPricer, EulerFdm<GBM>, CPPRng>
          s(sde, pricerPut, fdm,rngPM, NSim, NT);

    s.start();

    std::cout << "\n C++11 Rng: " << pricerPut->Price() << '\n';

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

    sw.Start();
    SUD<GBM, EuropeanPricer, EulerFdm<GBM>, CPPRng>
          s2(sde, pricerCall, fdm, rngCPP, NSim, NT);

    s2.start();

    std::cout << "\n Polar Marsaglia: " << pricerCall->Price();

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

    return 0;
}
```

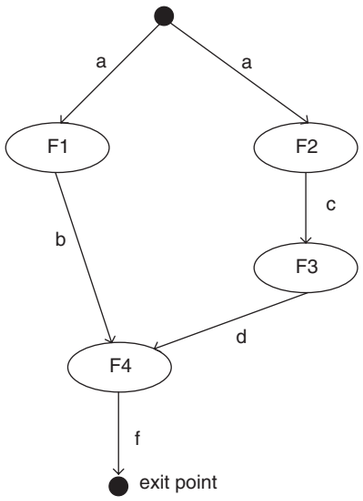
This is the baseline program that can later be extended and generalized.

Finding concurrency in applications

Concurrency refers to the existence of multiple threads of execution, each one receiving a slice of time to execute before being pre-empted by another thread. Concurrency is a requirement for a program that must react to external events and to stimuli. Typical examples of concurrent programs are operating systems and computer games. We note that a program can be concurrent even on a one-core machine. *Parallelism* involves several threads executing at the same time on multiple cores. The main goal of *parallel programming* is to improve the performance of compute-intensive applications that should not be interrupted when running on multiple cores. A *task* is the smallest exploitable unit of concurrency.

The two main decomposition patterns to break a problem into subproblems that can execute concurrently are based on data and tasks. *Task parallelism* (also known as *function parallelism* and *control parallelism*) is based on the model that a problem is a stream of instructions that can be broken down into a sequence of tasks that run simultaneously. This approach is similar to the way we decomposed systems in the single-threaded case. Task parallelism is concerned with running different tasks on

Figure 3: My first data dependency graph



the same or different data at the same time. Communication between the tasks takes place by passing data from one task to the next as part of a *workflow*.

Data decomposition involves decomposing data into *chunks*, and these chunks are input to tasks. Ideally, it should be possible to operate on the chunks relatively independently of each other. An example of where data decomposition can be used is the addition and multiplication of matrices. In this case we decompose a matrix into blocks (submatrices) in such a way that each block fits into cache memory. Another example is *loop-level parallelization*, in which an array is broken into chunks and a single operation is applied to each chunk. Examples of data parallel programming environments are the *Message Passing Interface* (*MPI*) and *Open Multi-Processing* (*OpenMP*) (see Gropp *et al.*, 1997; Chapman *et al.*, 2008). This data model is used extensively in graphics processing unit applications.

A *data dependency graph* or *task graph* is a directed graph in which each vertex represents a task to be completed and an edge from vertex *a* to vertex *b* means that task *a* must complete before task *b* begins. We are now in a position to draw these graphs. To help us get started, we view the graph as a data flow diagram that processes *primary input data* and transforms it in some way to produce *major output data*. We take an example in Figure 3 in which functions receive data and produce data. At this stage we are assuming that there is no shared data, in which case the functions `F1`, `F2`, `F3`, and `F4` are considered to be *pure functions*, that is:

- The functions always evaluate the same result given the same argument value(s). The function result cannot depend on any hidden information or state that may change while program execution proceeds, or between different executions of the program, nor can it depend on any external input from I/O devices.
- Evaluation of the result does not cause any semantically observable *side effects* or output, such as mutation of mutable objects or output to I/O devices.

Having created a task graph, we are in a position to think about the detailed design. There are categories of parallel programs that have been documented as *parallel design patterns* (Mattson, Sanders, and Massingill, 2005). The design tactics in that book provide guidelines on how to create scalable and efficient parallel code.

Steps to parallelization

We define a process to analyze, design, and implement programs that execute on multi-processor hardware. The code must give accurate results and it must be optimized for performance. To this end, a summary of the main steps is as follows:

1. *Decomposition* – break the program into tasks.
2. *Assignment* – assign tasks to processes.
3. *Orchestration* – data access, communication, and synchronization of processes.
4. *Mapping* – bind processes to processors.

This is an iterative process (sometimes combined with trial-and-error experimentation), and we can spend much time making the program correct, ensuring that it is free of race conditions and improving its speedup.

There are several ways to improve the runtime performance of the sequential code in the previous sections. We mention some options:

- Threading in the *C++ Concurrency* library.
- Tasking in the *C++ Concurrency* library using the class std::future.
- Tasking in the PPL library (Campbell and Miller, 2011).
- The Asynchronous Agents library (Campbell and Miller, 2011).

Tasks versus threads

It is possible to use *logical* threads to create multi-threaded code. The alternative is to employ *asynchronous tasks*. The question now is to decide which choice is "better" in some sense. In general, we are interested in performance, scalability, and maintainability of applications. Logical threads are mapped to physical threads, usually in a one-to-one mapping for higher efficiency. Mismatches and lower efficiency can occur due to *undersubscription*, in which there are not enough logical threads to keep the hardware busy; *oversubscription* occurs when there are more logical threads than physical threads. Oversubscription leads to *time-sliced* execution of logical threads, thus causing overheads.

We give a summary of some of the advantages of tasks:

1. Improved *load balancing*. The scheduler uses the correct number of threads and it distributes work evenly across those threads. A good task decomposition algorithm will help the scheduler assign tasks to threads and hence balance the load. On the other hand, it is the responsibility of the developer to perform load balancing by hand when writing multi-threaded code.
2. Faster task startup and shutdown. Tasks are lighter than logical threads. Tasks can be anywhere from 18 to 100 times faster to start and stop than the corresponding thread operations.
3. More efficient evaluation order. Thread schedulers are democratic in the sense that they typically distribute time slices to threads in a *round-robin* fashion. Thus, each logical thread gets a fair share of the allocated time. Task schedulers, however, have some higher-level information and they can delay starting a task until it can make useful progress.
4. We can concentrate on the logical dependencies between tasks (using dependency graphs, for example) and we can let the scheduler do what it does best, namely task scheduling.
5. For novice (and experienced) developers, tasks are easier to understand and to implement than threads. Multi-threaded code can be difficult to debug (are there race conditions and performance issues?), and also difficult to scale.

As a final example, we show how to parallelize Monte Carlo using C++ and PPL tasks. We create four instances of the program:

```
int NSim = 1'000'000;
int NT = 500;

double driftCoefficient = 0.08; double diffusionCoefficient = 0.3;
double dividendYield = 0.0; double initialCondition = 60.0;
double expiry = 0.25;
auto sde = std::shared_ptr<GBM>
    (new GBM(driftCoefficient, diffusionCoefficient, dividendYield,
          initialCondition, expiry));

double K = 65.0;

// Factories for objects in context diagram
std::function<double(double)> payoffPut
    = [&K](double x) {return std::max<double>(0.0, K - x); };
std::function<double(double)> payoffCall
    = [&K](double x) {return std::max<double>(0.0, x - K); };
double r = 0.08; double T = 0.25;
std::function<double()> discounter = [&r, &T]()
            { return std::exp(-r * T); };

auto pricerCall = std::shared_ptr<Pricer<EuropeanPricer>>
          (new EuropeanPricer(payoffCall, discounter));
auto pricerPut = std::shared_ptr<Pricer<EuropeanPricer>>
          (new EuropeanPricer(payoffPut, discounter));

auto fdm = std::shared_ptr<EulerFdm<GBM>>(new EulerFdm<GBM>(sde, NT));

// Random number generators
auto rng1 = std::shared_ptr<Rng<PolarMarsaglia>>(new PolarMarsaglia());
auto rng2 = std::shared_ptr<Rng<MTRng>>(new MTRng());
auto rng3 = std::shared_ptr<Rng<MTRng64>>(new MTRng64());
auto rng4 = std::shared_ptr<Rng<FibonacciRng>>(new FibonacciRng());


SUD<GBM, EuropeanPricer, EulerFdm<GBM>, PolarMarsaglia>
            s(sde, pricerPut, fdm, rng1, NSim, NT);
SUD<GBM, EuropeanPricer, EulerFdm<GBM>, MTRng>
            s2(sde, pricerPut, fdm, rng2, NSim, NT);
SUD<GBM, EuropeanPricer, EulerFdm<GBM>, MTRng64>
            s3(sde, pricerCall, fdm, rng3, NSim, NT);
SUD<GBM, EuropeanPricer, EulerFdm<GBM>, FibonacciRng>
            s4(sde, pricerCall, fdm, rng4, NSim, NT);
```

We create four lambda functions, which when called will run the simulator:

```
auto fn1 = [&]()
    { s.start(); std::cout << "\n Polar Mar: " << s.Price(); };
auto fn2 = [&]()
    { s2.start(); std::cout << "\n MT: " << s2.Price(); };
auto fn3 = [&]()
    {s3.start(); std::cout << "\n MT64:" << s3.Price(); };
auto fn4 = [&]()
    { s4.start(); std::cout << "\n Fibonacci: " << s4.Price(); };
```

In sequential code these functions are run one after the other:

```
fn1();
fn2();
fn3();
fn4();
```

These functions can be run in parallel without incurring race conditions, assuming that shared data is read-only or that there is no shared data.

We now proceed to give some code examples in C++ and PPL of how to parallelize the above sequential code. This is one possible scenario and it is easy to parallelize, because the data that the threads share is read-only and hence race conditions cannot occur. Each solution is a special case of a *fork-and-join* programming model (Mattson, Sanders, and Massingill, 2005; Chapman *et al.*, 2008). Under this model, a program starts as a single thread of execution (this is the *initial thread*). This thread then creates a team of threads (this is the *fork*) and it becomes the *master* of the team. It also communicates with the other members of the team. At the end of the fork–join construct only the master thread is active, because the other threads have finished (this is the *join*). In fact, the master thread waits on the other threads to terminate at a *barrier*, which is a point in the execution of a program where threads wait on each other; no thread in the team may proceed beyond a barrier until all threads in the team have reached that point.

We now see how the *fork-and-join* model is implemented by C++11 and related libraries.

- C++ threads
  A *thread* is a runtime entity that is able to independently execute a stream of instructions. The runtime scheduler allocates a fixed amount of execution time to a thread to give other threads a chance to execute:

```
std::thread t1(fn1);
std::thread t2(fn2);
std::thread t3(fn3);
std::thread t4(fn4);

// No shared data so we define 1 barrier/rendez vous point


t1.join();
t2.join();
t3.join();
t4.join();
```

- C++ futures
  A *task* is a program in local memory in combination with a collection of I/O ports. Tasks send data to other tasks through their output ports and they receive data from other tasks through their input ports. A *future* represents a read-only placeholder view of a variable, while a *promise* is a writable, single-assignment container that sets the value of the future. In short, the future is the value and a promise is the (asynchronous) function that sets the value. We can define a future without having to specify which specific promise will set its value.

```
std::future<void> fut1(std::async(fn1));
std::future<void> fut2(std::async(fn2));
std::future<void> fut3(std::async(fn3));
std::future<void> fut4(std::async(fn4));

// Wait for results to become available
fut1.wait();
fut2.wait();
fut3.wait();
fut4.wait();

fut1.get();
fut2.get();
fut3.get();
fut4.get();
```

- PPL parallel invoke
  The function `parallel_invoke` executes a set of tasks in parallel. It returns when each task has finished. This algorithm is useful when we have several independent tasks that must execute in parallel.

```
// Start the work functions
concurrency::parallel_invoke
(
    fn1,
    fn2,
    fn3,
    fn4
);
```

- PPL tasks
  PPL supports *task groups*. We create tasks, add them to a group, run the tasks, and wait for all of them to terminate.

```
// Parallel task using task group
concurrency::task_group tg;

tg.run(fn1);
tg.run(fn2);
tg.run(fn3);
tg.run(fn4);

tg.wait();
```

Unfortunately, a discussion of the performance gains that we can achieve using the above code is beyond the scope of this article.

Conclusions and applicability

We have given an introduction to a defined and repeatable process to analyze and design mathematical problems in computational finance, and implemented it using the multi-paradigm programming styles and new libraries that C++11 supports. The approach taken here is to understand the problem, then decompose the resulting system into subsystems, and finally design each subsystem. We postpone the choice of a traditional object-oriented design for as long as possible, because it is a kind of premature optimization.

For readers who do not yet have hands-on experience with multi-threading, we recommend *OpenMP* as a good way to learn the basic principles from both a theoretical and a practical perspective (Chapman *et al.*, 2008) before moving on to more advanced libraries. We note that Visual Studio C++ supports *OpenMP* up to version 2.0. The latest version is 4.5.

Daniel J. Duffy is the founder of Datasim Financial and has been involved with C++ and its applications since 1989. More recently, he has been involved with the design of computational finance applications in C++11. He has a PhD in numerical analysis from Trinity College (Dublin University) and can be contacted at dduffy@datasim.nl.

References

Boehm, B. 1986. A spiral model of software development and enhancement. *ACM SIGSOFT Software Engineering Notes* 11(4), 14–24.
Campbell, C. and Miller, A. 2011. *Parallel Programming with Microsoft Visual C++*. Redmond, WA: Microsoft Press.
Chapman, B., Jost, G., and van der Pas, R. 2008. *Using OpenMP*. Cambridge, MA: MIT Press.
DeMarco, T. 1978. *Structured Analysis and System Specification*. Cambridge, MA: Yourdon Press.
Duffy, D. J. 2004. *Domain Architectures*. Chichester: John Wiley & Sons, Ltd.
Duffy, D. J. 2017. A PDE software framework in C++11 for a class of path-dependent options. *Wilmott magazine* 91, 48–57.
Duffy, D. J. 2018. *Financial Instrument Pricing using C++*, 2nd edn. Chichester: John Wiley & Sons, Ltd.
Duffy, D. J. and Kienitz, J. 2009. *Monte Carlo Frameworks*. Chichester: John Wiley & Sons, Ltd.
Duffy, D. J. and Palley, A. R. 2017. C++11, C++14 and C++17 for the impatient: Opportunities in computational finance. *Wilmott magazine* 90, 38–47.
Gamma, E., Helm, R., Johnson, R., and Vlissides, J. J. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.
Gropp, W., Lusk, E., and Skjellum, A. 1997. *Using MPI*. Cambridge, MA: MIT Press.
Jackson, M. 2001. *Problem Frames*. New York, NY: Addison Wesley.
Mattson, T. G., Sanders, B. A., and Massingill, B. L. 2005. *Patterns for Parallel Programming*. New York, NY: Addison-Wesley.
Meyer, B. 1997. *Object-Oriented Software Construction*. New York, NY: Prentice Hall.
Pólya, G. 1990. *How to Solve It*. Harmondsworth: Penguin Books.
Williams, A. 2012. *C++ Concurrency in Action*. New York, NY: Manning Publications.
Wilmott, P., Lewis, A., and Duffy, D. 2014. Modelling volatility and valuing derivatives under anchoring. *Wilmott magazine* 73, 48–57.

Book Club

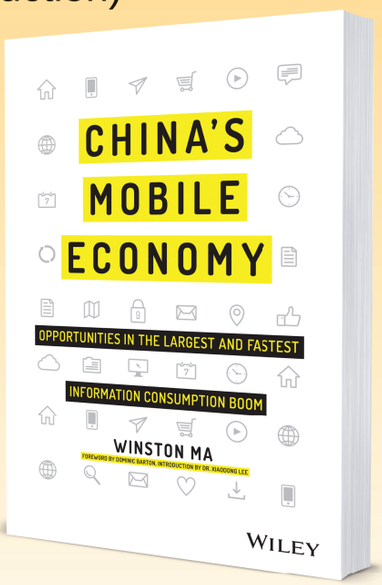
Exclusive discounts on the latest books from Wiley... 30% off for *Wilmott magazine* readers!

China's Mobile Economy

Opportunities in the Largest and Fastest Information Consumption Boom

Winston Ma, Dominic Barton (Foreword), Xiaodong Lee (Introduction)

- **978-1-119-12723-9**
- **368 pages**
- **December 2016**
- **Paperback**
- ~~**£20.99 / €25.20 / $27.50**~~
- **£14.69 / €17.64 / $19.25**



Explore the world-changing digital transformation in China

China's Mobile Economy: Opportunities in the Largest and Fastest Information Consumption Boom is a cutting-edge text that spotlights the digital transformation in China. Organised into three major areas of the digital economy within China, this ground-breaking book explores the surge in e-commerce of consumer goods, the way in which multi-screen and mobile Internet use has increased in popularity, and the cultural emphasis on the mobile Internet as a source of lifestyle- and entertainment-based content. Targeted at the global business community, this lucid and engaging text guides business leaders, investors, investment banking professionals, corporate advisors, and consultants in grasping the challenges and opportunities created by China's emerging mobile economy, and its debut onto the global stage.

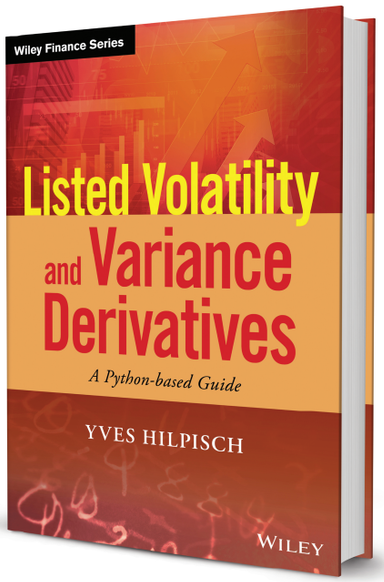
Year 2014-15 marks the most important inflection point in the history of the internet in China. Almost overnight, the world's largest digitally-connected middle class went both mobile and multi-screen (smart phone, tablets, laptops and more), with huge implications for how consumers behave and what companies need to do to successfully compete. As next-generation mobile devices and services take off, China's strength in this arena will transform it from a global "trend follower" to a "trend setter."

Listed Volatility and Variance Derivatives

A Python-based Guide

Yves Hilpisch

- **978-1-119-16791-4**
- **368 pages**
- **November 2016**
- **Hardback**
- ~~**£60.00 / €72.00 / $75.00**~~
- **£42.00 / €50.40 / $52.50**



Leverage Python for expert-level volatility and variance derivative trading

Listed Volatility and Variance Derivatives is a comprehensive treatment of all aspects of these increasingly popular derivatives products, and has the distinction of being both the first to cover European volatility and variance products provided by Eurex and the first to offer Python code for implementing comprehensive quantitative analyses of these financial products. For those who want to get started right away, the book is accompanied by a dedicated Web page and a Github repository that includes all the code from the book for easy replication and use, as well as a hosted version of all the code for immediate execution.

Python is fast making inroads into financial modelling and derivatives analytics, and recent developments allow Python to be as fast as pure C++ or C while consisting generally of only 10% of the code lines associated with the compiled languages. This complete guide offers rare insight into the use of Python to undertake complex quantitative analyses of listed volatility and variance derivatives.

- Learn how to use Python for data and financial analysis, and reproduce stylised facts on volatility and variance markets
- Gain an understanding of the fundamental techniques of modelling volatility and variance and the model-free replication of variance
- Familiarise yourself with micro structure elements of the markets for listed volatility and variance derivatives
- Reproduce all results and graphics with IPython/Jupyter Notebooks and Python codes that accompany the book

Listed Volatility and Variance Derivatives is the complete guide to Python-based quantitative analysis of these Eurex derivatives products.

SAVE 30%

When you subscribe to *WILMOTT magazine* you will automatically become a member of the **Wilmott Book Club** and you'll be eligible to receive 30% discount on specially selected books in each issue when you order direct from **www.wiley.com** - just quote promotion code **WILMO** when you order. The titles will range from finance to narrative non-fiction.
For further information, contact our **Customer Services Department on +44 (0) 1865 778315**, or Email: *cs-journals@wiley.com*