# 

# 

# 

# **Software Design Document (SDD)**

## **Monte Carlo Option Pricing Engine**

**Author:** Kayla Sermini  
 **Language:** C++17  
 **Project Type:** Portfolio | Quantitative Finance | High Performance Computing

[**Software Design Document (SDD) 1**](#_544fljmejpuj)

[Monte Carlo Option Pricing Engine 1](#_osa0841umdqx)

[1. Introduction 3](#_jfamcge2twjj)

[1.1 Purpose 3](#_kvd59o75z2zg)

[1.2 Scope 3](#_6umt4bmfnlwa)

[2. Architecture 3](#_q646qaoi1czb)

[2.1 High-Level Overview 3](#_v6tif39n5kqh)

[2.2 Component Diagram 3](#_fow7rbwy854k)

[3. Class Design 4](#_9hpms8n4ivy0)

[3.1 Class Diagram 4](#_uhqyeit3oy3c)

[4. Execution Flow 5](#_jb29t6pfi1f4)

[4.1 Sequence Diagram 5](#_l9xuz2vo1i6)

[Diagram Overview 5](#_b53ujswu2a0)

[5. Design Decisions 7](#_onaldkeml8nk)

[5A. Design Patterns Used 8](#_hdsrbw7degpb)

[1. Factory Pattern (Manual version) 8](#_pzkajk66ilbd)

[2. Strategy Pattern (Future-proofing) 8](#_a3mpj6loerpa)

[3. Template Method Pattern 9](#_2ne86a22yo4)

[4. Single Responsibility Principle (SRP) 9](#_4pe36pf0ls48)

[5. Thread Pool Potential (Command Pattern upgrade) 9](#_xswg272hua8s)

[Summary Table 10](#_8s6ctkl4zc6k)

[Design Justifications 10](#_dajsdyjxnnht)

[Resilience and Performance Alignment 10](#_l4ogkbidy97l)

[6. Multithreading Strategy 11](#_bk1eued0woc4)

[7. Usage 11](#_kkbvjiep2xcd)

[7.1 Input Parameters 11](#_imdkvcs0xqpp)

[7.2 Run Simulation 11](#_e45eiiix0iuz)

[7.3 Output 12](#_ef2hi1faia5b)

[8. Build Instructions 12](#_91n6cbs5vp8l)

[8.1 Tools Required 12](#_8ob50ncbjbgs)

[8.2 Compile 12](#_lcxklmw8miaa)

[8.3 Run 12](#_ougwgheilrgs)

[8.4 Clean 12](#_6rx2m0g9p8yp)

[9. Future Enhancements 13](#_frozon64nqnx)

[10. Conclusion 13](#_5gu9yh1smif9)

## **1. Introduction**

### **1.1 Purpose**

This document outlines the design and implementation of a Monte Carlo Option Pricing Engine in C++. The project aims to demonstrate advanced C++ programming skills, adherence to OOP principles, and efficient use of data structures and algorithms pertinent to quantitative finance applications.

### **1.2 Scope**

The engine supports pricing European-style Call and Put options using the Monte Carlo simulation method. It incorporates multithreading for performance optimization and is designed with extensibility in mind, allowing for future integration of additional financial models and instruments.

Future extensions may include:

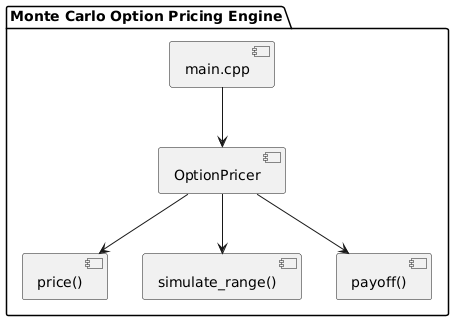
* Support for Greeks (Delta, Gamma, etc.)
* Heston model for stochastic volatility
* Asian/barrier options

## **2. Architecture**

### **2.1 High-Level Overview**

The Monte Carlo Option Pricing Engine simulates numerous possible future asset prices and calculates expected option payoff using a stochastic process.Multithreading enables efficient utilization of CPU resources. The system is built in a modular, testable, and performance-oriented way.

### **2.2 Component Diagram**

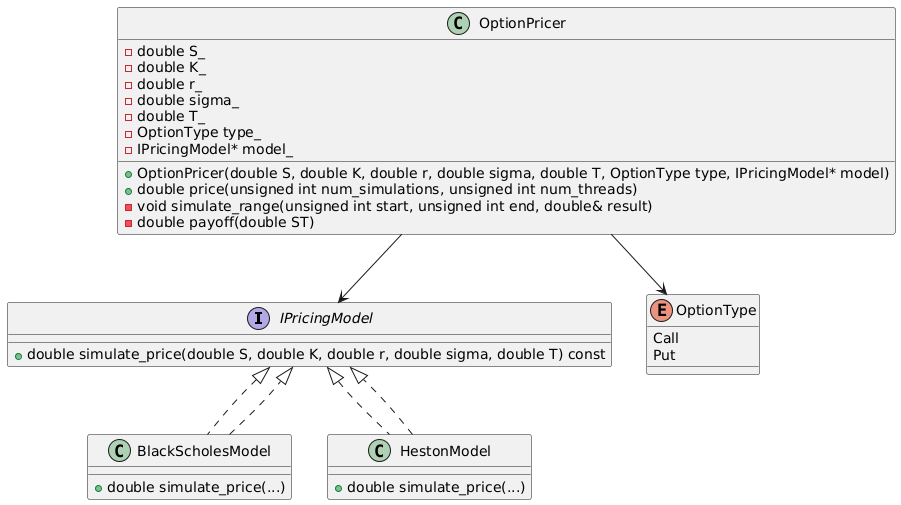


#### **Explanation**

* main.cpp: Entry point, receives user inputs and runs simulation.
* OptionPricer: Orchestrates simulations, aggregation, and pricing.
* IPricingModel: Abstract interface for pricing algorithms.
* BlackScholesModel: Concrete pricing implementation.

## **3. Class Design**

### **3.1 Class Diagram**



### **Class Roles and Responsibilities**

| **Class** | **Responsibility** |
| --- | --- |
| **OptionPricer** | Coordinates simulation, workload distribution, aggregation, and pricing. Acts as the controller in the simulation pipeline. |
| **IPricingModel (interface)** | Abstracts away pricing logic. Allows flexible injection of different stochastic models (e.g., Black-Scholes, Heston). |
| **BlackScholesModel** | Implements IPricingModel with logic specific to the Black-Scholes pricing formula. |
| **HestonModel** *(optional future)* | Provides a different pricing strategy using stochastic volatility. |
| **OptionType (enum)** | Encodes option variety (Call or Put) as a type-safe enum, improving readability and reducing logic errors. |

### **Design Principles Applied**

| **Principle** | **Application** |
| --- | --- |
| **Encapsulation** | OptionPricer hides pricing internals and exposes only high-level simulation APIs. |
| **Single Responsibility Principle (SRP)** | Each class handles one concept: pricing logic (model), orchestration (pricer), or option metadata (enum). |
| **Open/Closed Principle** | OptionPricer can use any model implementing IPricingModel without modification. |
| **Dependency Inversion Principle** | Pricing logic depends on the abstract IPricingModel, not concrete implementations. |
| **Polymorphism** | Allows runtime injection of pricing strategies using interface-based inheritance. |

### **Performance Alignment**

| **Concern** | **Design Feature** |
| --- | --- |
| **Memory locality** | All pricing logic is encapsulated per-thread; models can be reused statelessly. |
| **Low overhead** | Interface method calls are optimized by compiler via inlining where applicable. |
| **Safety** | Type-safe enum prevents misuse of option type conditions. |
| **Parallel compatibility** | No shared mutable state within pricing model or pricer logic. Thread-safe by design. |

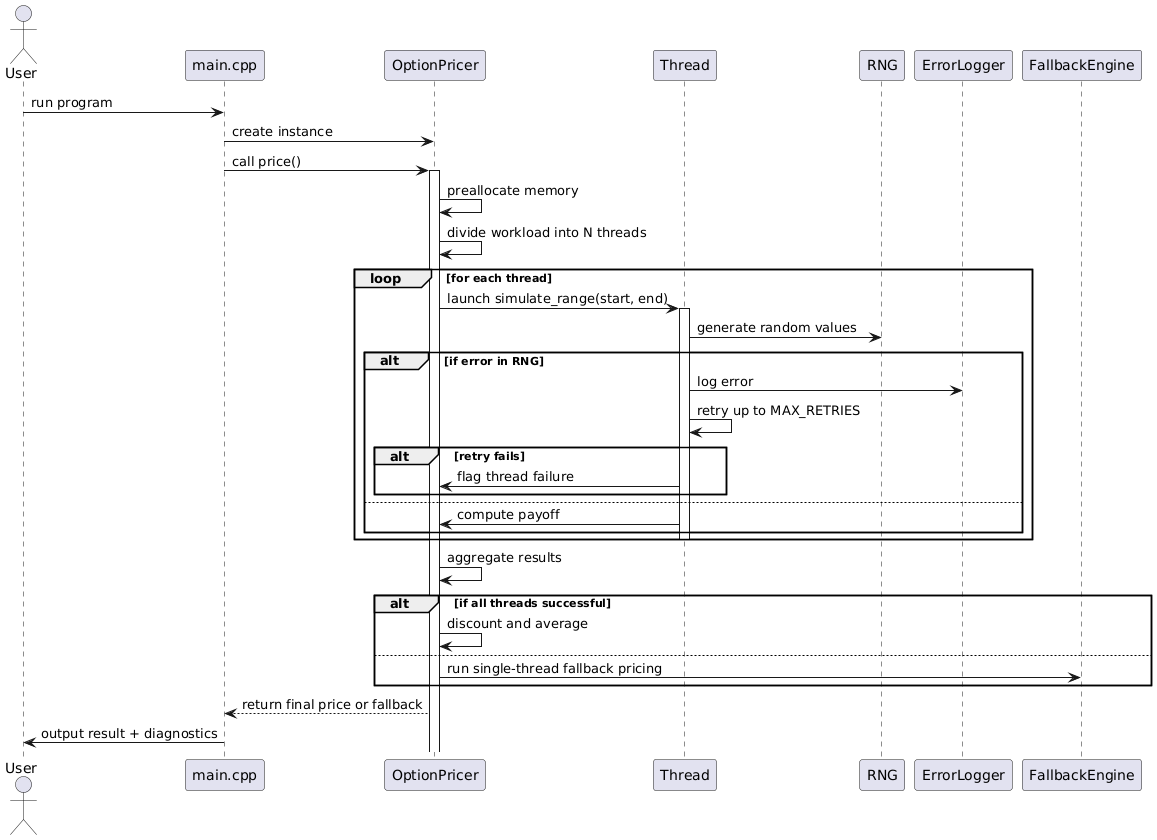
## **4. Execution Flow**

### **4.1 Sequence Diagram**

### **Diagram Overview**

The following sequence diagram illustrates the end-to-end execution flow of the Monte Carlo Option Pricing Engine, including low-latency optimization techniques, multithreaded simulation, error handling, and fallback mechanisms.

This diagram extends the basic flow to cover more realistic and production-grade requirements, such as fault tolerance and computational resilience under high-load environments—common in quantitative finance applications.



#### **Explanation**

| **Step** | **Description** |
| --- | --- |
| **1** | The user initiates execution via the program entry point in main.cpp. |
| **2** | A new instance of OptionPricer is created, initialized with pricing parameters (e.g., spot price, strike price, maturity). |
| **3** | The price() method is invoked. This is the central method coordinating all simulation logic. |
| **4** | The engine begins by preallocating memory buffers and initializing thread-local resources to avoid latency introduced by dynamic memory operations during execution. |
| **5** | The simulation workload is divided across N threads, where N is either provided by the user or dynamically determined by querying system hardware concurrency. |
| **6** | Each thread is launched independently, executing the simulate\_range() function for its assigned subrange of simulations. |
| **7** | Within each thread, a thread-local instance of a random number generator (RNG) is used to generate asset price paths. This avoids contention and ensures reproducibility. |
| **8** | If a thread encounters an error during simulation (e.g., RNG failure, numerical instability), it logs the error using a centralized ErrorLogger utility. |
| **9** | The thread then retries the simulation up to a predefined threshold (MAX\_RETRIES). If retries are exhausted, the thread flags itself as failed and halts further computation. |
| **10** | If all threads succeed, the main engine aggregates all partial payoffs and computes the final discounted expected value of the option. |
| **11** | If any thread fails irrecoverably, the system invokes a fallback engine (e.g., single-threaded simulation) to recompute the price in a more controlled environment. |
| **12** | The final output (either from multithreaded or fallback execution) is returned to main.cpp, which displays the result to the user along with diagnostic or timing information. |

## **5. Design Decisions**

| **Design Element** | **Reasoning** |
| --- | --- |
| **C++ Language** | Used for performance and system-level control. |
| **Multithreading** | Makes simulation fast and scalable. |
| **Object-Oriented Design** | Encapsulates logic for reuse and testing. |
| **Stateless Threads** | Each thread works independently = no race conditions. |
| **RNG per thread** | Prevents identical results across threads. |
| **Manual CLI setup** | Keeps focus on performance and design (can add GUI later). |

## **5A. Design Patterns Used**

This project uses key **object-oriented design patterns** that are simple, but powerful. Each one helps keep the code modular, flexible, and easy to expand later without changing the entire system.

### **1. Factory Pattern (Manual version)**

We don’t use a separate factory class, but our OptionPricer constructor behaves like a **manual factory**:

#### **What it means:**

You can create different pricing "products" (Call or Put) by just changing the OptionType parameter. This decouples the pricing logic from the input type, which makes your class reusable and configurable.

### **2. Strategy Pattern (Future-proofing)**

Right now, the engine supports only the **Black-Scholes pricing strategy**, but it’s structured in a way that lets you later inject new pricing strategies (like Heston) without changing the rest of the code.

#### **If extended:**

class PricingStrategy {

public:

virtual double simulate(double S, double K, double r, double sigma, double T) = 0;

};

class BlackScholes : public PricingStrategy { ... }

class HestonModel : public PricingStrategy { ... }

Then OptionPricer would take a pointer to a PricingStrategy object and call simulate(). This lets you swap in new logic without touching the engine—this is the Strategy Pattern.

### **3. Template Method Pattern**

Your price() function acts as a **template method**:  
 It defines the skeleton of the simulation process (loop over threads, call simulate\_range), but the core logic of the payoff is handled separately in payoff().

double OptionPricer::price(...) {

// Shared setup logic

simulate\_range(...);

// Shared cleanup logic

}

This separation of control flow and computation details makes it easy to test and update individual parts without breaking others.

### **4. Single Responsibility Principle (SRP)**

Each function and class has **one job**:

* OptionPricer: manages simulation config and controls flow
* simulate\_range: does the simulation work
* payoff: handles option-specific logic

This makes your code easier to debug, extend, and reason about—because each part has a clear purpose.

### **5. Thread Pool Potential (Command Pattern upgrade)**

In future enhancements, you could refactor the thread launching to use a **thread pool**, where each simulation range is packaged into a "task" (command) and executed by a worker.

This sets you up for the **Command Pattern**:

* Each task is an object.
* Each thread takes and executes tasks.

This would allow for dynamic task scheduling, retries, cancellation, etc.

## **Summary Table**

| **Pattern** | **Purpose / Role in Project** |
| --- | --- |
| **Factory (manual)** | Simplifies creation of pricers for different option types |
| **Strategy (future)** | Makes pricing logic modular and extendable (e.g., Heston) |
| **Template Method** | Defines high-level steps while letting low-level vary |
| **SRP** | Keeps each method/class focused on a single responsibility |
| **Command (optional)** | Could be used in future to manage task-based threading |

### **Design Justifications**

| **Design Element** | **Justification** |
| --- | --- |
| **Thread-local RNGs** | Reduces latency and eliminates contention from shared random sources. |
| **Preallocation** | Avoids performance hits from heap allocation during simulations. |
| **Retry Mechanism** | Provides resilience in environments with numerical edge cases. |
| **Fallback Engine** | Ensures system reliability by preventing total failure on partial thread errors. |
| **Centralized Error Logging** | Enables post-analysis and supports profiling in production systems. |

### **Resilience and Performance Alignment**

This sequence aligns with real-world quant engineering constraints:

* **Low-latency design** via preallocated memory and concurrent threading.
* **Fail-safe simulation** under rare but possible numeric or memory faults.
* **Auditability** through error logging and retry visibility.
* **Maintainability** via clear separation of concerns and extension points.

## **6. Multithreading Strategy**

* Uses std::thread to parallelize simulation.
* Number of simulations divided equally across threads.
* Each thread uses its own RNG instance.
* Final result is the discounted average of all thread-level payoff sums.
* Thread-safe because each thread writes only to its own memory slot.

## **9. Future Enhancements**

| **Feature** | **Description** |
| --- | --- |
| **Greeks Calculation** | Add Delta, Gamma, Vega using finite differences |
| **Heston Model** | Simulate stochastic volatility |
| **Path-dependent Options** | Asian, barrier, and lookback support |
| **Command-Line Interface** | Add argument parsing |
| **Unit Tests** | Use GoogleTest for regression checks |
| **Benchmarking** | Track performance impact of thread scaling |
| **Python Visualization** | Export CSVs and plot payoffs |

## **10. Conclusion**

This engine is a high-performance simulation built to demonstrate:

This engine demonstrates mastery of C++, OOP principles, high-performance computing, and quant modeling techniques. It is extensible, fault-tolerant, and engineered to simulate real-world financial scenarios reliably and efficiently.