Queens College, CUNY, Department of Computer Science

Computational Finance

CSCI 365 / 765

Fall 2017

Instructor: Dr. Sateesh Mane

October 4, 2017

2 Project Part 2

2.1 Sample client request

- Here is a very typical calculation a calling application might wish to perform.
- Calculate the fair value of an option in a loop, from $S = 1$ to $S = 100$ in steps of 1, where S is the stock price. That is a total of 100 calculations.
- **How would we support it?**

2.2 Question(s) to ponder

- Here is one possible answer, to calculate option fair values in a loop.

  1. The calling application instantiates an `Option` object (call it `opt`), which is one of our derived classes.
  2. The calling application executes a stored procedure database call, etc. and obtains indicative and market data.
  3. The calling application calls `opt.getIndicativeData()` and `opt.getMarketData()` to initialize `opt`. All the validation checks are successful.
  4. Next the calling application runs a loop from $S = 1$ to $S = 100$ in steps of 1.
     (a) On each pass, the calling application overwrites the value of the relevant data member in `opt`, to set the stock price.
     (b) Next, the calling application calls `opt.FairValue()` and reads the output data object to read the option's fair value.
  5. After the loop completes, `opt` goes out of scope. The calculation is over.

- **Question: would this work?**
- **Question: is it a good design?**
- Suppose `FairValue()` allocates memory internally, before calling `calc()`, and releases the memory after calling `calc()`.
- That seems to be basically what would happen in our current design.
- **Question: is it a good idea to allocate and deallocate memory 100 times, for what is basically the same mathematical calculation? (It is only the stock price which changes.) And we can easily increase the loop to 1000 steps.**
- **Question: is there a better way to support the client request?**
- ***Question: how?***
- **These are questions to think about. It is not a final software design.**
- ***Use your imagination.***

2.3 Alternative interface

- As I have written it, `FairValue()` takes only one function argument, which is an output data object. *Why is that?*

- Why not design `FairValue()` to take three function arguments, one for the indicative data, one for the market data, and one for the output data?

- The revised function signature would be this.

```
int FairValue(const IndicativeData & indData,
              const MarketData & mktData,
              OutputData & outData);
```

- Then `FairValue()` would internally call both `getIndicativeData()` and `getMarketData()`, and pass them the relevant input data objects.

- In this revised architecture, both `getIndicativeData()` and `getMarketData()` could be declared as protected class methods. *What is wrong with this idea?*

- **There is nothing wrong with this idea. It would work and is a good idea.**

- *Then why not adopt it?*

  1. Suppose `FairValue()` accepted the `IndicativeData` and `MarketData` objects as inputs, and passed them to `getIndicativeData()` and `getMarketData()` and validated them and proceeded with the rest of the steps described above.
  2. Then consider the following:
     (a) ***Where/when would the calling application have an opportunity to perform a client ovewrite of the input data?***
     (b) Once `FairValue()` is called, the flow of program execution lies entirely within `FairValue()`, until it exits and returns the output to the calling application.
  3. In our current architecture, both `getIndicativeData()` and `getMarketData()` are called first, by the calling application. Then the calling application has an opportunity to overwrite the input data. After that, the calling application calls `FairValue()` to do a calculation and return output. There are weak points with this design.
     (a) The calling application could forget to call `getIndicativeData()` and/or `getMarketData()` before calling `FairValue()`.
     (b) Where/how do we enforce that our functions are called in the correct sequence?

- If `FairValue()` accepted the indicative and market data objects as inputs, it could guarantee that both `getIndicativeData()` and `getMarketData()` would be called, and at the correct point in the flow of program execution. Our current design does not guarantee this.

- **We can discuss this. There are merits and drawbacks to both schemes.**

- **Use your imagination. The design can be changed.**

2.4 Multiple inputs

- Return to the question of a loop of calculations, stock price $S = 1$ to 100 in steps of 1.
- As I have written it, we input one `const IndicativeData` object and one `const MarketData` object and output one (not `const`) `OutputData` object.
- *Why not an array of objects, or a vector?*

  Maybe 100 market data objects, each with a different stock price inside?
- Maybe a function signature like this.

```
int FairValue(const std::vector<IndicativeData> & indData,
              const std::vector<MarketData> & mktData,
              std::vector<OutputData> & outData);
```

- ***Question: check that the "const" has been applied correctly.***
- This version of `FairValue()` would know that there were 100 calculations to do.
  It might be able to make better decisions about memory allocation and deallocation.
- ***Could the client overwrite problem be solved this way?***
  1. The calling application overwrites the input data objects, with client overwrites.
  2. It is not a problem for us if the calling application overwrites the input data objects.
  3. What *is* a problem is that **we must not overwrite** the input data objects.
- **Question: do you understand the difference: who overwrites the input data?**
- ***Question: what if the input vectors are not the same length?***
  1. This could get messy.
  2. Think about it.

Think about it.

2.5 Client Data

- Our inputs are not restricted to only a class for the indicative data and market data.
- There could be a class for client overwrites, call it `ClientOverwrites`.
- We would receive the indicative data and market data and a list of client overwrites.
- **Question: how would we implement it?**

Think about it.