PDE Models

PDE Models

- Goal is to set up design software frameworks
- Determine *scope* (“what kinds of PDE supported?”)
- ISO 9126: Accuracy, Efficiency, Functionality
- In the long term Maintainability paramount
- (Creeping featuritis ☺, evolving requirements)
- Reusability, assembly from prebuilt libraries (e.g. NAG, IMSL; Fortran)

Challenges

- Types of PDEs in computational finance
- Categories of PDEs
- Finite difference methods for PDEs
- Libraries, frameworks, applications in C++11
- (Multiparadigm design patterns)

PDEs in Computational Finance

- Equities (Black Scholes, Heston, Asian,…)
- Interest rate (CIR, HW, Cheyette,…)
- Fixed income (caps, floors, swap, swaptions,…)
- Specials (UVM, Anchoring (Wilmott, Lewis and Duffy))
- Nonlinear PDEs (Hamilton-Jacobi-Bellman (HJB))
- Fokker-Planck, Kolmogorov

Dimensions of PDEs

- D1: Degree of nonlinearity
- D2: One-factor and multi-factor
- D3: Domain (bounded, semi-infinite, infinite)
- D4: Fixed, free and moving boundaries
- D5: Known/unknown parameters
- D6: Time-dependence, time-independence
- D7: PDEs in conservative and non-conservative forms

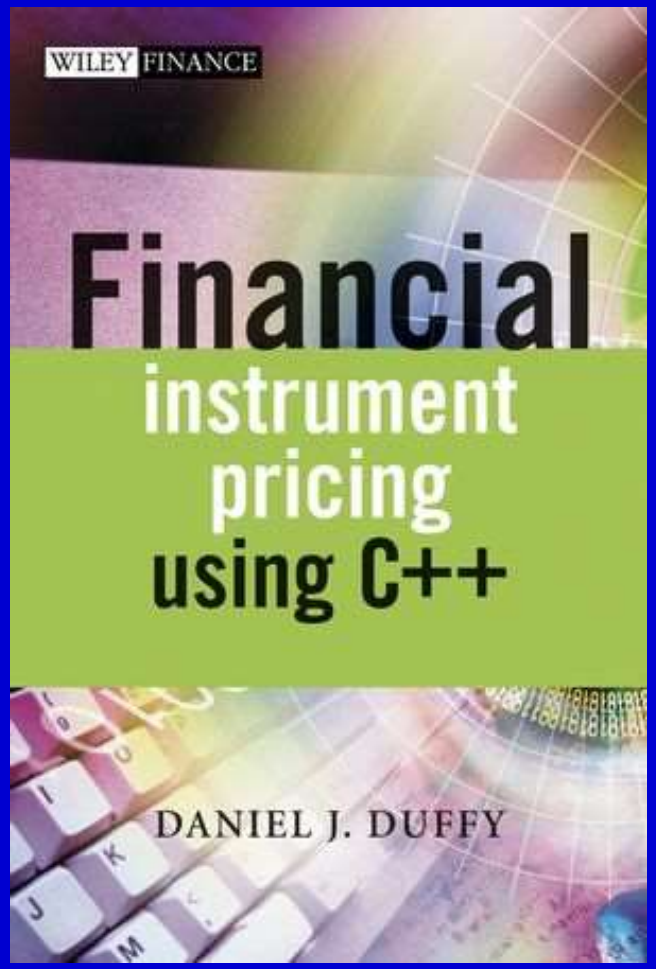
Finite Difference Methods

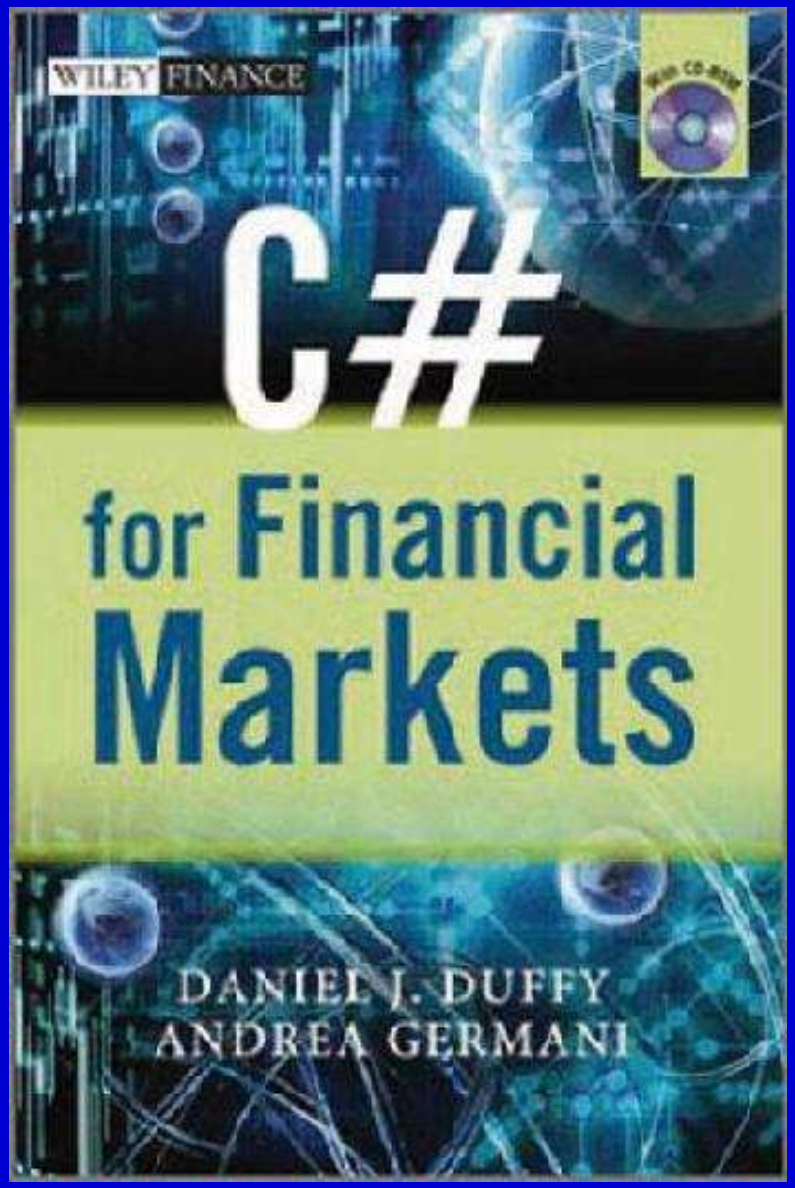
- F1: Suitability for linear and nonlinear PDE
- F2: One-step methods, multi-step methods
- F3: Facility with multi-factor models
- F4: Reusability of PDE and FDM models
- F5: Using numerical libraries to promote productivity

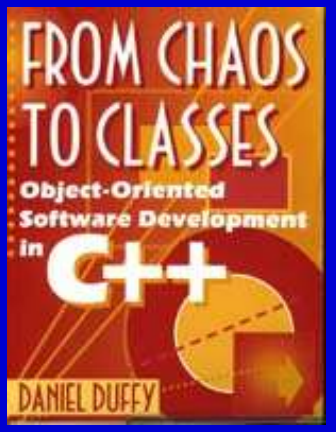
How to write a FD Solver?

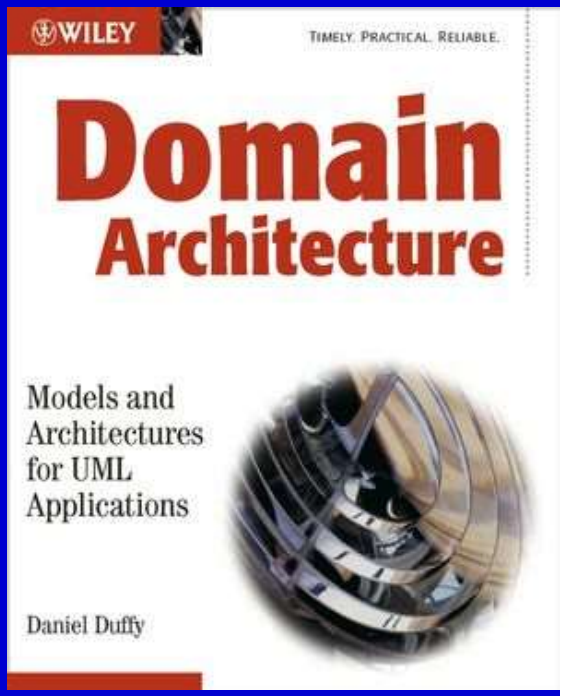
- L1: Hard-coded (e.g. Crank Nicolson for Black Scholes PDE) (Duffy 2004)
- L2: GOF design patterns (subtype polymorphism, CRTP pattern) (Duffy 2006, Duffy 2009)
- L3: Layered approach (POSA pattern)
- L4: Domain Architectures (incorporates designs L1, L2, L3) (Duffy 2004, Duffy 2015)

A Word from Sponsor









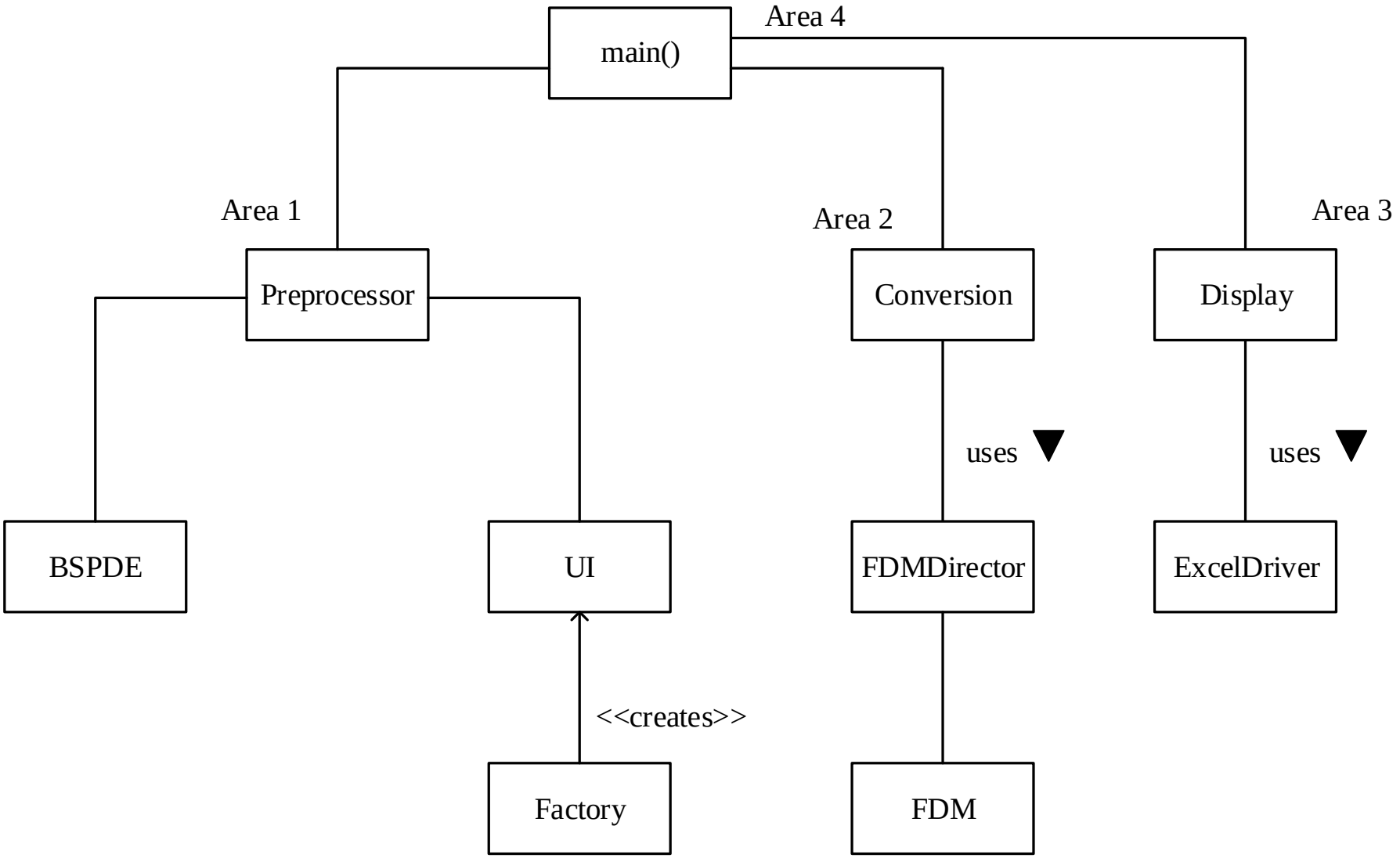
System Architecture (1)

- Which approach to use? (L1 to L4)
- Each approach has its own level of difficulty and consequences
- “Mental model” varies between bottom-up programming and top-down system decomposition
- Many implicit assumptions in developer’s head ☺

System Decomposition

- Break a system into loosely-coupled and cohesive components
- Choose between task and data decomposition
- Single Responsibility Principle (SRP)
- Components have well-defined *provides* and *requires* interfaces
- Component internals can be implemented using OOP and design patterns

Solution 1


main()
Area 4
Area 1
Area 2
Area 3
Preprocessor
Conversion
Display
uses
uses
BSPDE
UI
FDMDirector
ExcelDriver
<<creates>>
Factory
FDM

Remarks

- Separation of concerns (SRP)
- Object-oriented approach
- Based on PAC (Presentation-Abstraction-Control) model
- Not very flexible (nor maintainable)
- Useful for throwaway prototypes

Code: Initialisation

```
using namespace BlackScholesOneFactorIBVP;

// Assignment of functions
sigma = mySigma;
mu = myMu;
b = myB;
f = myF;
BCL = myBCL;
BCR = myBCR;
IC = myIC;

double T = 1.0; int J= 200; int N = 4000-1;
double Smax = 100.0;
FDMDirector fdir(Smax, T, J, N);
fdir.Start();
```

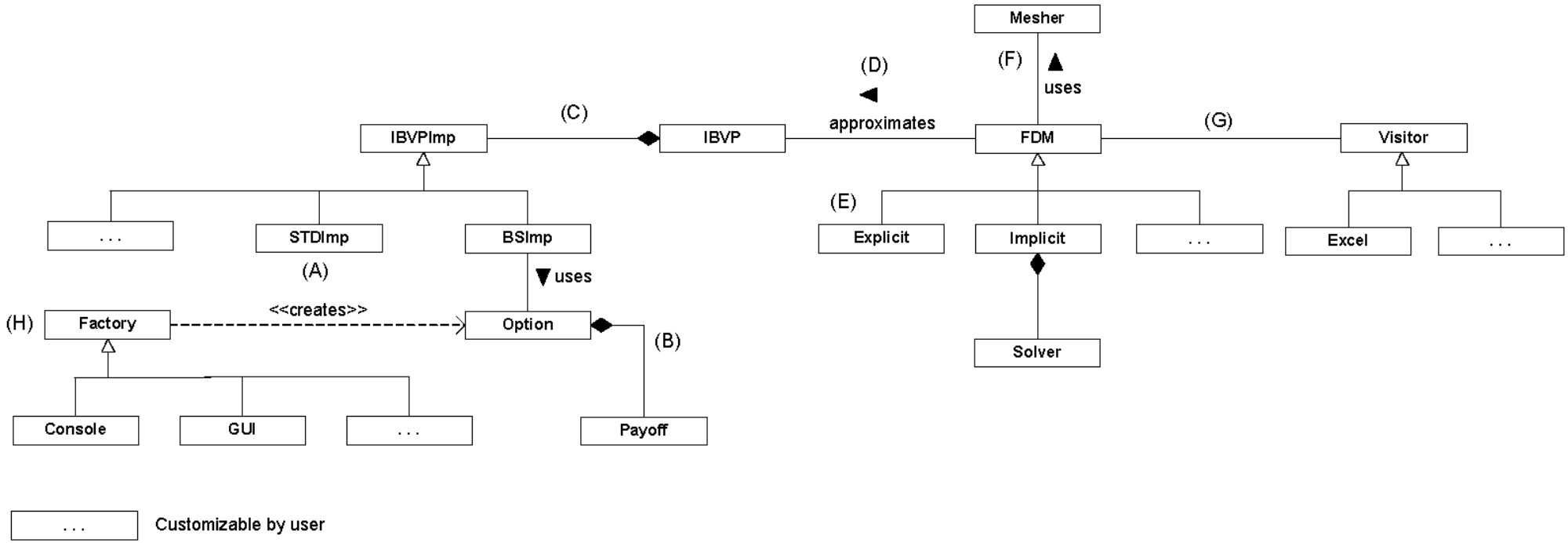
Code: State Machine

```
L1:
        fdir.doit();

        if (fdir.isDone() == false)
                goto L1;

        Vector<double, long> xresult(J+1, 1);
        xresult[xresult.MinIndex()] = 0.0;
        double h = Smax / (double) (J);

        for (long j = xresult.MinIndex()+1;
                                j <= xresult.MaxIndex(); j++)
        {
                xresult[j] = xresult[j-1] + h;
        }


        printOneExcel(xresult, fdir.current(), string("Value"));
```

Solution 2

- OOP design pattern approach
- Class hierarchies and subtype polymorphism (virtual functions))
- Variant: use CRTP
- Patterns: Template Method, Bridge, State, Mediator

Model



UML Class Diagram of Finite Difference Solution

Remarks

- Maintainability issues due to inheritance hierarchy
- Possible performance issues
- Reasonably stable (new schemes only need to implement couple functions)
- Restricted to linear convection-diffusion-reaction PDE with Dirichlet boundary conditions
- Object-oriented interfaces

Code: PDE Class

```
class IBVP { // 'Device-independent' class

        Range<double> xaxis;                // Space interval
        Range<double> taxis;                // Time  interval

        IBVPImp* imp;                       // Bridge implementation

public:
        // Coefficients of parabolic second order operator
        double diffusion(double x, double t) const; // Second derivative
        double convection(double x, double t) const;
        double zeroterm(double x, double t) const;
        double RHS(double x, double t) const;

        // Boundary and initial conditions
        double BCL(double t) const;
        double BCR(double t) const;
        double IC(double x) const;

        double Constraint(double x) const;
};
```

Code Bridge Abstraction

```
class IBVPImp
{

public:

        // Selector functions
        // Coefficients of parabolic second order operator
        virtual double diffusion(double x, double t) const = 0;
        virtual double convection(double x, double t) const     = 0;
        virtual double zeroterm(double x, double t) const = 0;
        virtual double RHS(double x, double t) const = 0;


        // Boundary and initial conditions
        virtual double BCL(double t) const = 0;
        virtual double BCR(double t) const = 0;
        virtual double IC(double x) const = 0;

        virtual double Constraint(double x) const = 0;

};
```

Code Bridge Implementation

```
class BSIBVPImp : public IBVPImp
{
public:
        Option* opt; // BS data

        BSIBVPImp(Option& option);

        double diffusion(double x, double t) const;
        double convection(double x, double t) const;
        double zeroterm(double x, double t) const;
        double RHS(double x, double t) const;


        // Boundary and initial conditions
        double BCL(double t) const;
        double BCR(double t) const;
        double IC(double x) const;

        double Constraint(double x) const;
};
```

Code Base Class FDM

```
class IBVPFDM
{ // Set of finite difference to solve scalar initial
  // boundary value problems

protected:

          IBVP* ibvp;          // Pointer to 'parent'

// more

          Mesher m;

public:
          NumericMatrix<double, long>& result();            // The result of the calculation
          Vector<double, long>& resultVector();

          boost::tuple<Vector<double,long>, NumericMatrix<double, long>> ManagementInformation()
const;

          const Vector<double, long>& XValues() const;                // Array of x values
          const Vector<double, long>& TValues() const;                // Array of time values

          // Hook function for Template Method pattern
          virtual void calculateBC() = 0; // Tells how to calculate sol. at n+1
          virtual void calculate() = 0; // Tells how to calculate sol. at n+1

};
```

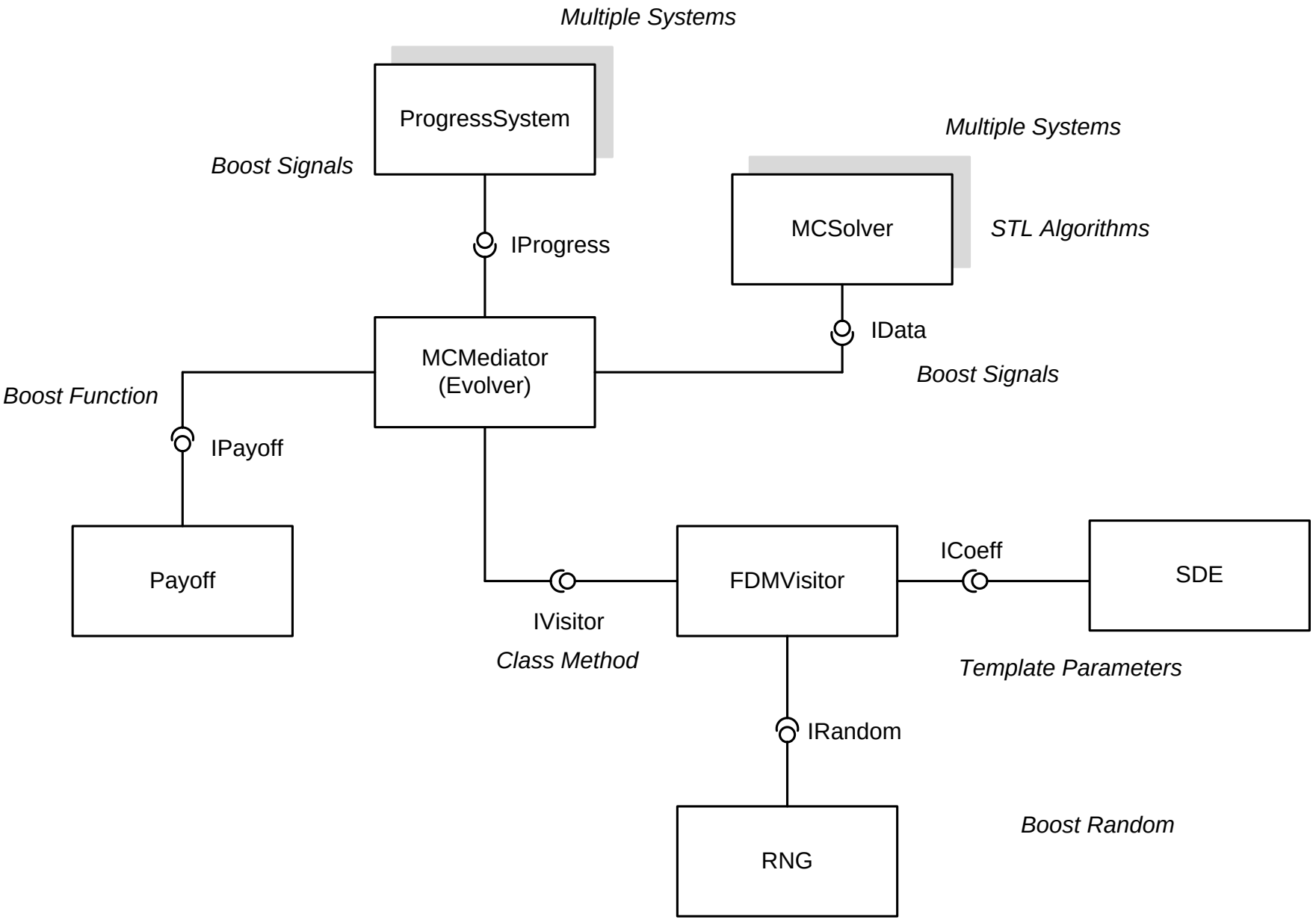
Code ADE Solver

```
class ADE: public IBVPFDM
{
private:

        // Intermediate values
        Vector<double, long> U;
        Vector<double, long> V;
        Vector<double, long> UOld;
        Vector<double, long> VOld;

public:
        ADE();
        ADE(IBVP& source, long NSteps, long JSteps);

        // Hook function for Template Method pattern
        void calculateBC();
        void calculate();

        double fitting_factor(double x, double t);
};
```

Solution 3

- Based on Domain Architectures and layering principles
- A domain architecture is a family of applications (meta pattern)
- Can be used as a *reference model* for several *similar* kinds of applications
- Used in Monte Carlo (C++ (V1) and C# (V2))

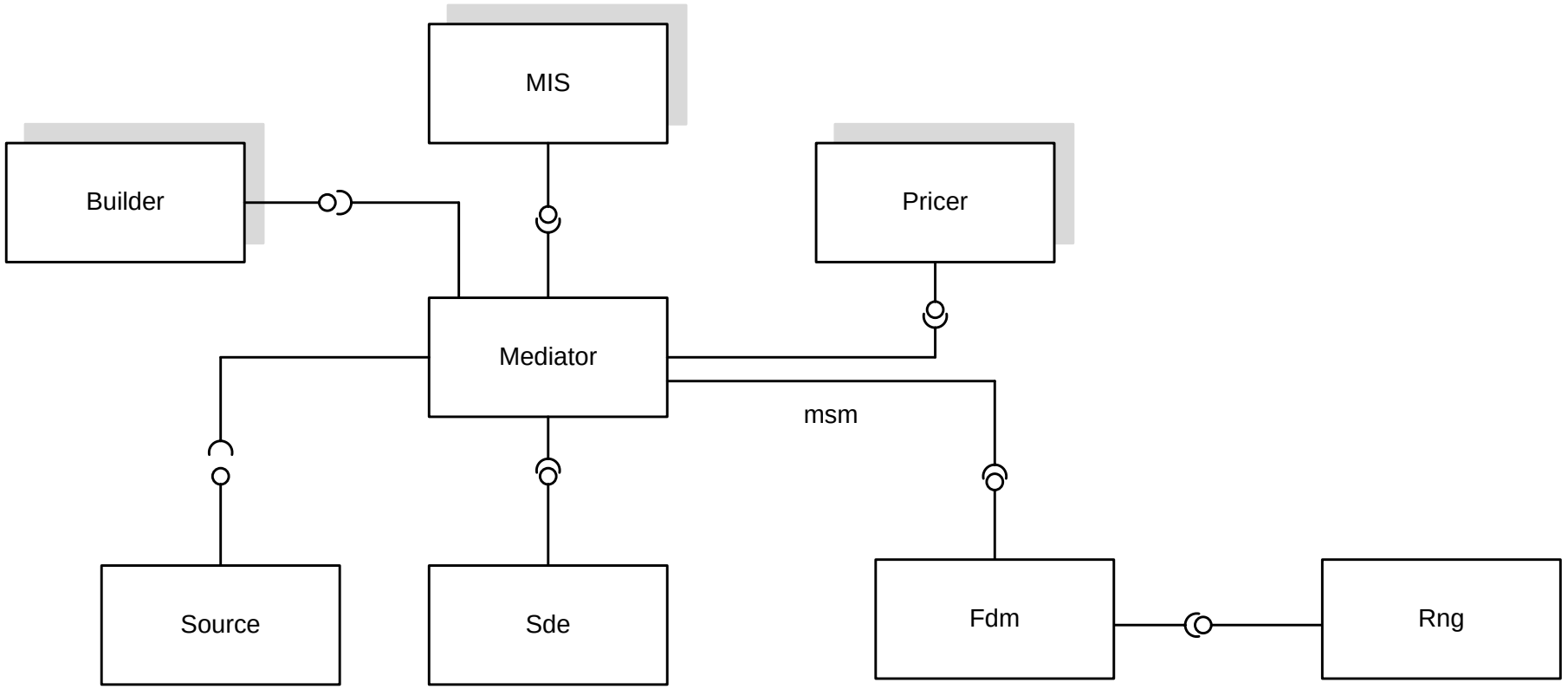
Original Model



Remarks

- Hybrid model (OOP, functional)
- More focus on decomposition and narrow interface design
- No (less) class hierarchies needed
- Support for both provides and requires interfaces
- Event notification using Boost signals2 (or .NET delegates)

Revised Model



C++ Classes for PDE

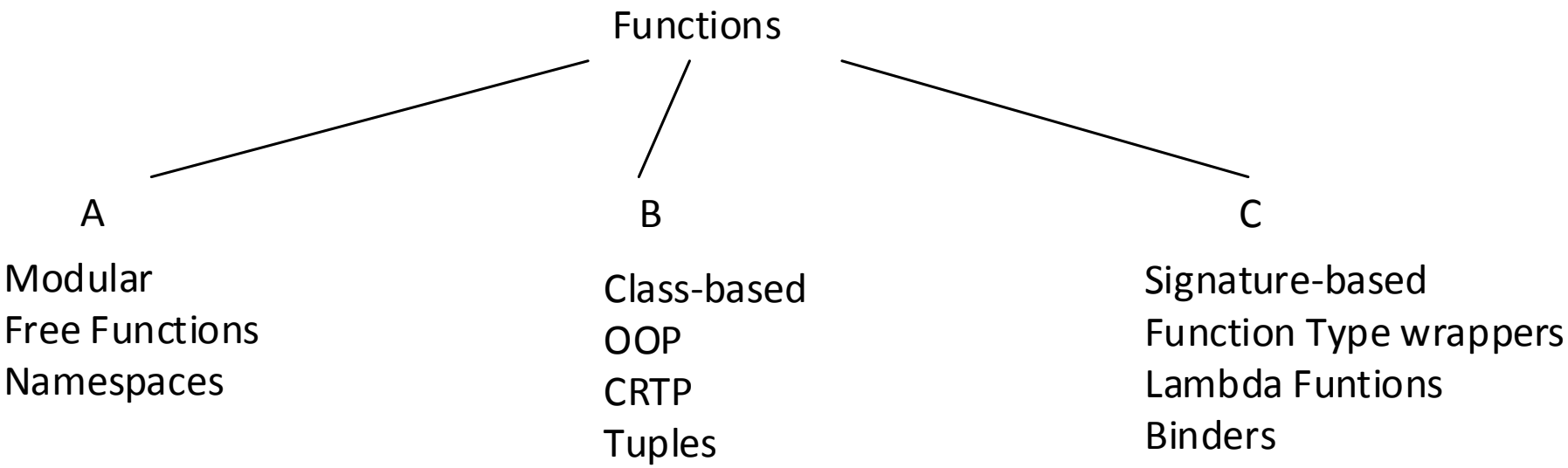
C++ Classes for PDE

- Discussion of how to design PDEs in C++ (non-OOP approach)
- Structure, creation and behaviour (in an application)
- How can we use the new features in C++?
- Let’s look at all the options!

Modelling Functions

- A. Modular, non-OO
- B. Class/struct-based
- C. Signature-based
- (It's a design decision to decide which option or combination of options to choose from)

Decisions, Decisions



Which Choice?

	Maintainability	Efficiencey	Portability	Interoperability
A	3	1	3	3
B	2	3	2	2
C	1	2	1	1

Critical Components

- C1: Function type wrappers
- C2: Tuples
- Combining C1 and C2 in different ways
- Composition structure
- Three-fold advantage a) group functions, b) functions are signatures, c) functions with tuples as input or output

Grouping Functions

- C++ does not support interfaces (compare C#, Java)
- Need some way to group functions into a single entity
- This entity can be tightly coupled or loosely coupled
- Use functional type wrappers (then entity become a Bridge instance)

Ex: Functions in a Class

```
template <typename T> class TwoFactorPde
{
public:
// U_t = a11U_xx + a22U_yy + b1U_x + b2U_y + cU_xy + dU + F;
// f = f(x,y,t)

        boost::function<T (T,T,T)> a11;
        boost::function<T (T,T,T)> a22;
        boost::function<T (T,T,T)> c;
        boost::function<T (T,T,T)> b1;
        boost::function<T (T,T,T)> b2;
        boost::function<T (T,T,T)> d;
        boost::function<T (T,T,T)> F;

        TwoFactorPde() {}

};
```

Ex: Tuples of Functions (1)

```
double diffusion (double x)
{
        double s = 0.2;
        return 0.5 * s * s * x *x;
}

double convection (double x)
{
        double r = 0.05;
        return r*x;
}
std::tuple<FunctionTypeI, FunctionTypeI>
        pde = std::make_tuple<FunctionTypeI, FunctionTypeI>
                (diffusion, convection);
```

Ex: Tuples of Functions (2)

```
typedef std::function<double (double)> FunctionTypeI;

// a u' + b u = g (a,b,g are scalar-valued functions)
typedef tuple<FunctionTypeI,FunctionTypeI,FunctionTypeI> RobinBC;

// Numerical approximation of Robin BC
// Input is a) boundary point x, b) mesh h, c) RobinBC
// Output is a tuple of 3 numbers

std::tuple<double,double,double>
       discreteRobinBC(const RobinBC& bc, double x, double h)
{ // One-sided discretisation of continuous Robin BC
  // Returns the coefficients

       double a = std::get<0>(bc)(x) / h;
       double b = std::get<1>(bc)(x) - a;
       double c = std::get<2>(bc)(x);

       return std::make_tuple<double,double,double> (a,b,c);
}
```

Function Composition

- Just like in mathematics
- Applications to PDE when we perform transformation to a unit interval, for example
- We also need to transform the coefficients (and derivatives) of the PDE
- Most PDE solver use domain truncation

Examples (1)

```
// A class to model function composition
struct FunComposer
{
      FunctionTypeI f_;
      FunctionTypeI g_;

      FunComposer(const FunctionTypeI& f,
            const FunctionTypeI& g): f_(f), g_(g) {}
      double operator () (double x) { return f_(g_(x));}
};

// Composition of functions
FunComposer fCom(diffusion, convection);

std::cout << fCom(100.0) << std::endl;
```

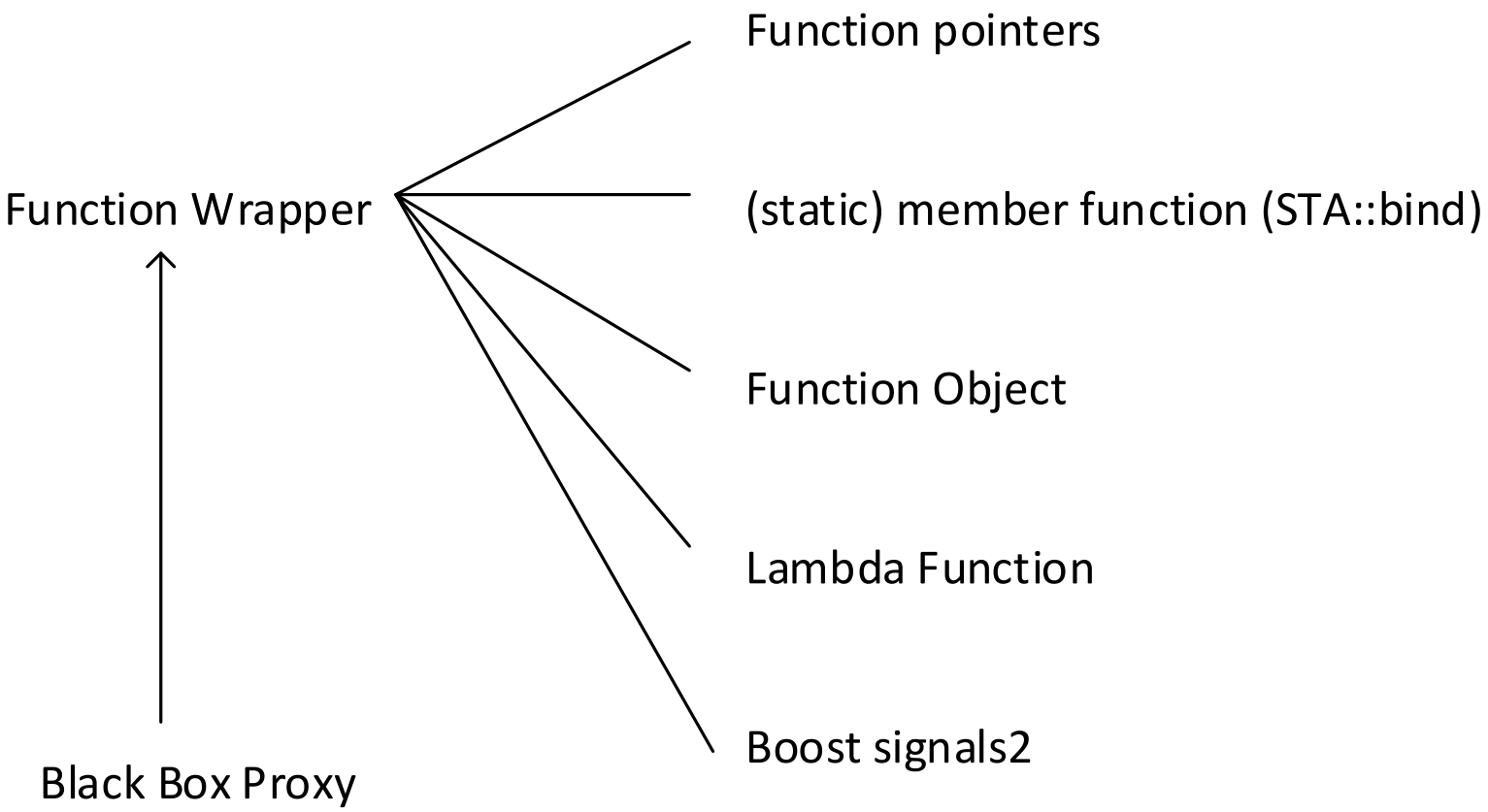
Examples (2)

```
auto Compose = [](const FunctionType& f,
       const FunctionType& g, double x, double t)-> double
{
       return f(x)*g(t);
};

double My(double x)
{ return std::pow(x, 0.5);}
double Another(double x)
{ return std::pow(x, 2.0);}

std::cout << "II " << H(My, Another, 2.0, 2.0);
```

‘Universal’ Function Type Wrappers

- `std::function` (from `boost::function<>`)
- Similar to a .NET delegate type
- Universal black box interface to many kinds of functions in C++
- Can handle various “developer styles”

Function Styles



The Layers Pattern (POSA 1996)

- We need a defined process in order to structure an application
- (Class hierarchies and subtype polymorphism are less critical to success these days …)
- Layering is a common technique (Edsger Dijkstra THE multiprogramming system)
- In finance, e.g. RiskWatch
- Choice: 3-layer versus 5-layer models

THE ("*Technische Hogeschool Eindhoven*") 1966, 5 Layers

- 0: OS multiprogramming; processes, semaphores,…
- 1: Allocating memory to processes(the *pager*)
- 2: Console – OS communication
- 3: All I/O between devices attached to the computer
- 4: User programs. Compilation, execution, printing user programs
- 5: The user!

Example: Lattice Models (3 Layers)

- In previous versions used class hierarchies, procedural, GOF patterns
- Maintenance issues (‘magic number 7, plus or minus 2’)
- Need a model that helps the developer stay in control
- And that allows us to manage projects

Layer 1: Data Layer

- ADT and containers to hold (structured) data
- Generic (node types, axes)
- Implementation (home-grown, Eigen, Boost uBLAS)
- Adapter containers (good information hiding)
- Objects and generics

Lattice ADTS

```
// The Node class contains the values of interest. LatticeType == 2
for binomial,3 for trinomial.
template <class Node, int LatticeType>
                class Lattice
{ // Generic lattice class

        // Implement as a full nested vector class
        std::vector<std::vector<Node> > tree;
};

template <typename TimeAxis, typename Node, int LatticeType>
                class GeneralisedLattice
{ // Generic lattice class


        // Options
        //std::vector<std::vector<Node> > tree;
        //boost::unordered_map<TimeAxis, std::vector<Node> > tree;
        std::map<TimeAxis, std::vector<Node> > tree;
};
```

Layer 2: Functions/Mechanisms

- Components use the ADTs from Layer 1
- e.g. forward and backward induction algorithms
- Algorithms (similar to GOF *Strategy* and *Visitor* patterns)
- Can use C++ 11 function wrapper to implement behavioural ‘variations’
- Generics and functional programming

Algorithms

```
template <class Node> void ForwardInduction
(Lattice<Node, TYPEB>& lattice,
const std::function<std::tuple<Node, Node>
       (const Node& node)>& generator,
const Node& rootValue
)


template <class Node> Node BackwardInduction
(Lattice<Node, TYPEB>& lattice,
const std::function<Node
       (const Node& upper,const Node& lower)>& generator,
const std::function<Node (const Node& node)>& endCond)
```

Layer 3: UI and Configuration

- Code to initialise the components in layers 1 and 2
- “Creational patterns”
- Lambda functions used as in-situ factories
- Combine tuples and application objects in a next generation Builder pattern
- (Multiple entry points to the application *network*)

Customisation

```
// Payoff as a lambda function
double K = 100;
auto Payoff = [&K] (double S)-> double
       {return std::max<double>(K - S, 0.0);};

// The early exercise constraint
auto AmericanAdjuster
       = [&Payoff](double& V, double S)->void
{ // e.g. early exercise

       V = std::max<double>(V, Payoff(S));
};
```

Using Lambda Functions

```
// Down-and-out put
double L = 20.0;
auto BarrierAmericanAdjuster = [&AmericanAdjuster, &L](double& V, double S)->void
{ // e.g. early exercise

          if (S > L)
          {
                    AmericanAdjuster(V,S);
          }
          else
          {
                    V = 0.0;

          }
};

double U = 100.0;
auto DoubleBarrierAmericanAdjuster = [&AmericanAdjuster, &L, &U](double& V, double S)->void
{ // down and out, up and out

          if (S > L && S < U)
          {
                    AmericanAdjuster(V,S);
          }
          else
          {
                    V = 0.0;
          }

};
```

Test Case

```
// Price a plain option
double res = LatticeMechanisms::BackwardInduction<double>
       (lattice, algorithm, Payoff);

std::cout << "Plain Option price, BM classic #1: " << res;

// Price an early exercise option
double res2 = LatticeMechanisms::BackwardInduction<double>
       (lattice2, algorithm, Payoff,
BarrierAmericanAdjuster);

std::cout << "Early execise option price" << res2;
```

Idea: 5-Layer PDE/FDM Model

5	User
4	Vieuws (FDM, FEM, MC)
3	Ínternal Functionality’
2	Lattice/Pde/Sde/Ode
1	Specific Pdes etc.