9. An Introduction to Unified Software Design (USD)

No two languages are ever sufficiently similar to be considered as representing the same social reality. The worlds in which different societies live are distinct worlds, not merely the same world with different labels attached. Edward Sapir

9.1 Introduction and Objectives

The first eight chapters of this book were devoted to the syntax and multi-paradigm language features in C++11 and C++14. But syntax alone is not sufficient if we wish to create flexible and maintainable code. To this end, we introduce a new design approach that subsumes and integrates a number of popular design methodologies to give us a *defined process* that can be applied to the creation of small to complex libraries, frameworks and applications. We call it *Unified Software Design* (*USD*) for convenience and we shall apply it in many of the remaining chapters of this book to a range of applications such as Monte Carlo simulation, option pricing using PDE methods and to the creation of supporting libraries and frameworks. Our goal is to have a standardised design approach to software development and this approach can also be applied by the reader when creating her own applications in computational finance.

The main goal of this chapter is to describe the underlying principles of USD, how it complements, subsumes and extends other design methodologies and how to analyse, design and implement small, medium-sized and complex applications. We give simplified (but not simple) generic examples in C++ to show the usefulness of the approach. Understanding the fundamental principles and the examples will help you make the transition to larger problems.

Some of the principles underlying our design approach can be summarised by the steps that György Pólya describes when solving a mathematical problem (Pólya 1990):

1. First, you have to *understand the problem*.
2. After understanding, then *make a plan*.
3. *Carry out the plan*.
4. *Look back* at your work. How could it be better?

We see these steps as being applicable to the software development process in general and to the creation of software for computational finance in particular. Getting each step right saves time and money. In short, we adopt the following tactic: *get it working, then get it right and only then get it optimised* (in that order).

9.2 Background

In this section we discuss the methods that have shaped and influenced USD. It is a multiparadigm method to analyse, design and implement applications in a range of domains, for example, process control, logistics and computer-aided design (CAD). A discussion of these kinds of applications is outside the scope of this book. Instead, we focus on applications related to computational finance and how they are implemented in C++.

USD can best be described as a defined process to analyse and design software systems. It uses features from a number of well-known system design approaches while at the same time it tries to avoid features that are more difficult to apply. The main phases in USD reflect Pólya's steps in section 9.1 when we solve a problem:

- Phase I: System Scoping and Initial Decomposition.
- Phases II: Identify System Components and Interfaces.
- Phase III: Detailed Design.
- Phase IV: Implementation, Review and Maintenance.

We discuss these phases in this and following chapters. The execution of each phase can be seen as a process that maps input to output and we describe the process as a sequence of (computable) *activities* that tie the process' output to its input. Ideally, we define a low-risk and seamless process to take user requirements and map them into code. To this end, we have used a number of established methods to analyse and design software systems. We now give a short description of each one to provide the reader with some background information.

9.2.1 Jackson Problem Frames

Michael Jackson introduced the idea of a problem frame (see Jackson 2001). A *problem frame* is defined as:

> *a kind of pattern. It defines an intuitively identifiable problem class in terms of its context and the characteristics of its domains, interfaces and requirements.*

Some of the reasons to discuss problem frames in this book are:

- They provide us with ideas on how to document domain architectures.
- They are based on a small set of very generic concepts.
- They provide reference models for classes of applications.

In general terms, we say that the *concern* of a problem frame captures the fundamental criteria of successful analysis for problems that fit that frame. Quoting Jackson 2001, the frame 'specifies what descriptions are needed, and how they must fit together to give a convincing argument that the problem has been fully understood and analysed'. Jackson has identified five basic reusable frames and these can be considered as being instances of a *problem frame model*.

The five basic forms are:

- *Required behaviour*: some part of the physical world whose behaviour must be controlled until some conditions are satisfied. An example of such a system is a Home Heating System or an environmental controller.
- *Commanded behaviour*: the physical world is controlled by the actions of an operator. An example is a sluice gate problem; this problem is a model of a simple irrigation system.
- *Information display*: systems where we need constant information about the real world. The information must be presented in the required place in the required form. An example is a one-way traffic light system.
- *Simple workpiece*: a tool to allow users to control and edit text or graphics objects on a screen. The objects can subsequently be copied, printed or transformed in some way. An example is a CAD (Computer Aided Design) application.
- *Transformation*: computer-readable input files whose data must be transformed to give certain required output files. The output data must be in a particular format. An example of such a frame is a management information system (MIS).

9.2.2 The Hatley-Pirbhai Method

This is a popular method for analysing real-time systems. It is an approach that combines data flow decomposition in which model components are constructed in control and information-space (see Hatley and Pirbhai 1988). The authors take a hierarchical and iterative view of systems development. In particular, the method captures system requirements by viewing a system from three major perspectives:

- The process (functional) model.
- The control (state) model.
- The information (data) model.

The *Data Flow Diagram* (DFD view) is the primary tool for depicting functional requirements in the Hatley–Pirbhai method. It partitions the requirements into component processes or functions. These functions are connected by data flows to form a network. The second view is called the *control model* and describes the circumstances under which the processes from the process model are performed. The control model examines the external events in a system and is documented using finite state machines (FSM). Finally, *information modelling* is the third perspective and it is not documented in Hatley and Pirbhai (1988).

We have been influenced by the Hatley–Pirbhai approach in a number of ways. First, it discusses the *context process* consisting of a single process, *terminators* (entities outside the context of the system) and *data flows*. Second, the main process is decomposed or levelled into subprocesses, thus promoting *separation of concerns*. Finally, the usefulness of this method for us is the realisation that the context diagram, data flow and architecture are indispensible when analysing software systems.

9.2.3 Domain Architectures

A *domain architecture* (Duffy 2004) is a reference model for a range of applications that share similar structure, functionality and behaviour. A domain architecture is a kind of *meta model* that describes more specific instance systems.

We discuss five basic forms and one 'composite' form:

- *MIS* (Management Information Systems): Produce high-level and consolidated decision-support data and reports based on transaction data from various sources.
- *PCS* (Process Control Systems): Monitor and control values of certain variables that must satisfy certain constraints. We are primarily interested in exceptional events.
- *RAT* (Resource Allocation and Tracking) systems: Monitor a request or some other entity in a system. The request is registered, resources are assigned to it, and its status in time and space is monitored.
- *MAN* (Manufacturing) systems: Create finished products and services from raw materials.
- *ACS* (Access Control Systems): Allow access to passive objects from active subjects. They are similar to security systems and the Reference Model in large computer systems.
- *LCM* (Lifecycle Model): A 'composite' model that describes the full lifecycle of an entity; an aggregate of *MAN*, *RAT* and *MIS* models.

We discuss examples of these models in later chapters.

9.2.4 Garlan-Shaw Architecture

An *architectural style* is a description of an architecture of a specific system as a collection of computational components together with a description of the interactions among these components, known as the *connectors* (see Shaw and Garlan 1996). Examples of components are databases, layers, filters and clients. Examples of connectors are procedure calls, event broadcasts and database protocols. Shaw and Garlan propose several common architectural styles:

- *Dataflow systems*: for example, batch sequential, pipes and filters.
- *Call-and-return systems*: object-oriented systems, hierarchical layers, main program and subroutine.
- *Data-centred systems* (repositories): databases, hypertext systems, blackboards.
- *Independent components*: communicating processes, event systems.
- *Virtual machines*: interpreters, rule-based systems.

These styles are *reference models* that we can apply in the detailed design stage of the software lifecycle. They are discussed in Shaw and Garlan 1996 and a more detailed account of a number of these styles can be found in POSA 1996 where they are documented in handbook form. A discussion of architectural styles is unfortunately outside the scope of this book.

9.2.5 System and Design Patterns

We discuss the system and design patterns of POSA and GOF, respectively (see POSA 1996, GOF 1995). The patterns in POSA are concerned with large-scale system design while the GOF patterns are more finely grained. For example, POSA describes how to design large systems in terms of patterns such as:

- Presentation–Abstraction–Control (PAC).
- Layers.
- Blackboard.
- Model–View–Controller.
- Pipes and Filters.
- Microkernel.
- Proxy.
- Publisher–Subscriber.

The GOF patterns are concerned with the lifecycle of objects (instances of classes):

- *Creational patterns*: flexible ways of creating objects:
  - Abstract Factory (creating related instances of classes in a class hierarchy).
  - Factory method (creating instances of a given class).
  - Prototype (creating objects as 'clones' of some typical object).
  - Builder (create a complex object step-by-step).
- *Structural patterns*: defining relationships between objects:
  - Composite: recursive aggregates and tree structures.
  - Proxy: indirect access to a resource via a 'go-between' object.
  - Bridge: separate a class from its various implementations.
  - Facade: create a unified interface to a logical grouping of objects.
  - Adapter: convert the interface of one class into the interface of another class.

- *Behavioural patterns*: how objects send messages to each other:
  - Visitor: extend the functionality of a class hierarchy (non-intrusively).
  - State: implement a Harel/UML statechart (see Harel and Politi 2000).
  - Strategy: create flexible, interchangeable algorithms for object methods.
  - Observer: define synchronising procedures between objects.
  - Mediator: define a single communication 'hub' in a star of objects.
  - Command: encapsulate a function as an object.

We remark that the GOF patterns can be viewed as a special case of a Lifecycle Model (LCM) because we are interested in object lifetime; the main phases are the creation of an object (MAN), placing the object in some structure (RAT) and then monitoring how the object interacts with other objects (MIS).

9.3 System Scoping and Initial Decomposition

In general, we try to understand as much of the problem as possible before delving into detailed design and coding work. 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 by us. Second, we decompose the initial amorphous and monolithic system into loosely-coupled and autonomous systems with well-defined interfaces.

We need to be careful not to fall into the trap of premature design; our main concern here is to understand the main data flow through the system and how the different systems partaking in the context diagram cooperate to realise that data flow and the system's core process.

9.3.1 System Context Diagram

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

We define the 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.

We represent the system context diagram by a drawing in which each system is a rectangular box with lines connecting SUD with its sources and sinks. Even at this stage we would like (if possible) 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 SUD. To this end, we add an extra vertical line to those boxes representing the former category. The SUD is a special case by definition and its box 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 and software). A generic example of a system context diagram is shown in Figure 9.1 (using the notation based on Jackson 2001).

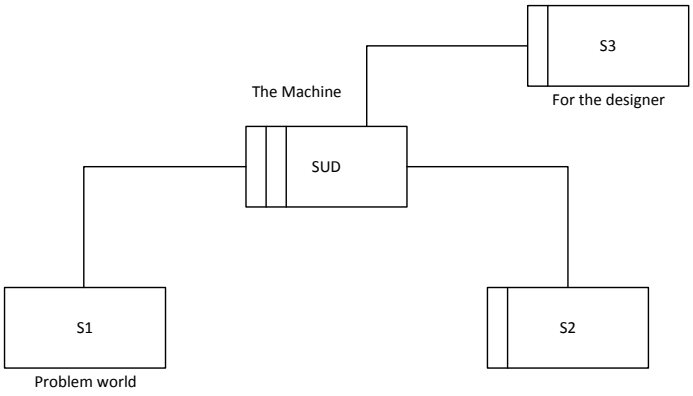

Figure 9.1 Example (Jackson approach)

Summarising, we describe the system under discussion (SUD) as a black box that is surrounded, as it were, by other systems that cooperate with SUD to ensure that 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 SUD and its *satellite systems*. Furthermore, it is an indispensable tool for project managers who must determine project size and risk, not to mention monitoring system evolution.

Arriving at a stable context diagram is an iterative process and it can be created by analysing system requirements, interviewing domain experts and 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.

9.3.2 System Data Flow

Having created the system context diagram we now find the data and information exchange between SUD and its external systems. This is a well-known technique in *Structured Analysis* (DeMarco 1978, Hatley and Pirbhai 1988) and is best described as a *data-flow diagram* (DFD). A DFD is:

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

This is a useful definition and we adapt it to modern design as discussed in this book. At this stage we can use DFDs to model the high-level data flow between SUD and its external systems. It is possible to check whether we understand what is going on and whether we have discovered the required systems and requirements. We shall see that they will allow us to determine the responsibilities of systems.

We take a well-documented example to motivate the use of DFDs, namely the *Drink Vending Machine* (DVM) (see Hatley and Pirbhai 1988). The problem is understandable to most readers while it is also challenging to analyse. Furthermore, we can generalise DVM to other applications and domains. In fact, we know that it is an instance system of the ACS domain architecture type (see Duffy 2004). Those readers who develop interactive applications will hopefully find the solution of DVM to be useful because there are many similarities between the features of DVM and those of other interactive applications.

We include the list of features and requirements from Hatley and Pirbhai 1988:

- F1: Accept objects (candidate coins) from the customer as payment.
- F2: Check for slugs (not real coins), for example by validating size, weight, thickness and serrated edges.
- F3: Accept Euros only.
- F4: The system cannot be tricked by conniving people.
- F5: The customer should be able to select a product.
- F6: Check product availability: if not available, return coins to customer.
- F7: Products and their prices may change from time to time.
- F8: Return the customer's coins on request if she decides not to go through with the transaction.
- F9: Dispense product if it is available and enough coins have been inserted.
- F10: Return the correct change if the amount deposited is greater than the product price.
- F11: Disable the product selection after the product has been dispensed and until the next validated coin is received.
- F12: Make deposited coins available for change.

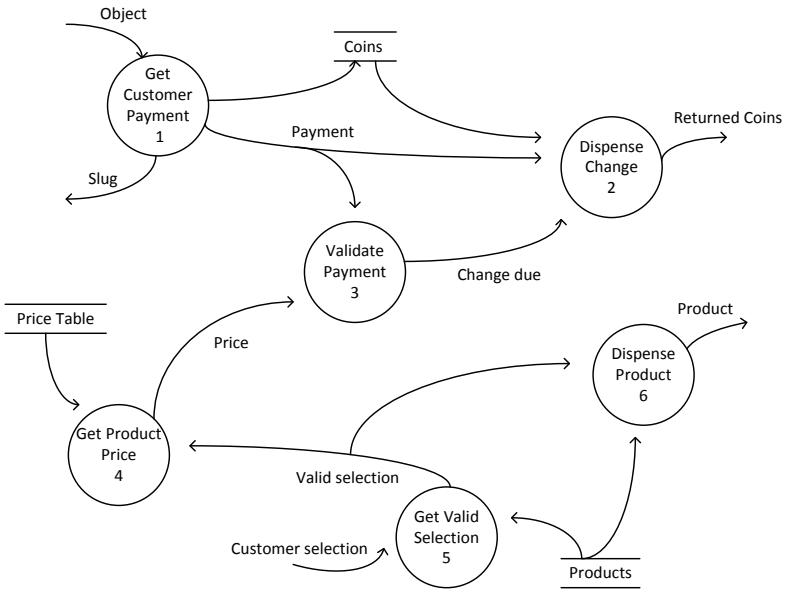

Figure 9.2 Traditional DFD

The traditional DFD is shown in Figure 9.2. In this diagram we note that sources and sinks (for example, *Products, Price Table* and *Coins*) are known as *data stores* and processes that consume or produce data (or both) are shown as circles. This informal diagram adds to our understanding of the problem but it can be difficult to map to modern tools and languages in our opinion. Instead, we reengineer this problem as a special case of an *Access Control System* (*ACS*) as discussed in Duffy 2004. The canonical context diagram is shown in Figure 9.3. We ignore the discussion about which external systems need to be designed:

- *ACS:* the central system (we call it DVM).
- *Source*: the system corresponding to active users that initiate all requests.
- *Authentication*: the system for coin insertion and validation.
- *Resources*: the system containing drinks.
- *Sink*: recipient of status reports on transaction completion (there may be multiple systems).
- *MIS*: (possibly) remote management systems that communicate with DVM.

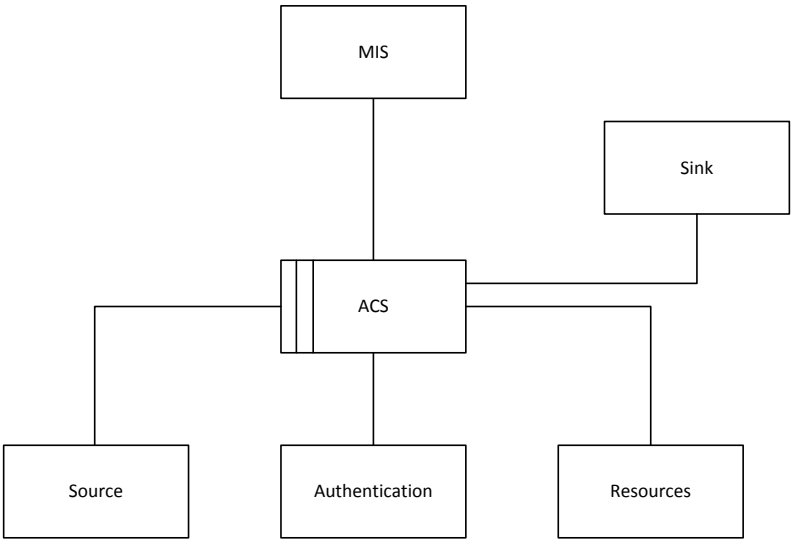

Figure 9.3 (Modern) Context Diagram

DFDs are useful brainstorming tools when we wish to discover the information flow between SUD and its external systems but further decomposition of the top-level DFD (for example, the case in Figure 9.2) is not desirable for a number of reasons. Instead, we try to formalise the top-level DFD by focusing on data produced

and consumed by the systems in the context diagram. In particular, analysing a problem by first determining what its *core process* is leads us to a more robust solution:

> *A core process is one whose deliverables are visible to external customers and it usually spans the whole organisation because several organisational units are involved in its execution*.

Translated into our process this means that we now start to concentrate on the major data flow, in particular the data that the critical Sink systems receive. In other words, we adopt Pólya's heuristic H6 (*work backward*) as it is a good test of how well we understand the problem before proceeding.

9.3.3 System Responsibilities and Services

At this stage we 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 behaviour that a system exhibits. A service can be an operation, function or transformation. It could even 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 used and 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 run its course.

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

9.3.4 Optimisation: System Context and Domain Architectures

It is possible to design software systems using an iterative approach, especially if the requirements have not yet been pinned down when a project begins. To complicate things, *emerging requirements* can force a rewrite of the software system or they can even cause project cancellation due to budget overruns. One way to mitigate project risk is to draw on one's experience (or on the experience of others). The approach that we take here is to determine if the current project is a special case of one or more of the domain architectures that we introduced in section 9.2.3. The styles have their differences but they have one thing in common, namely they all tend to have the same context diagram in their most general form. We show this in Figure 9.4 and it can be used as a template for any application. In particular, the generic names SUD as well as the names S1 to S5 need to be renamed and their responsibilities must be found (as in Figure 9.3 for the case of the DVM system) for example. As motivation, we describe the general responsibilities of each system:

- S1: The system without which there would be no input data. All major events and data originate in this system.
- S2: This is a kind of help/classifier system that transforms the data from S1 to entities and objects that can be used by system S3. In a sense, it is a kind of *object factory*.
- S3: The system that transforms the modified input data to the desired output form.
- S4: The external system that receives the output data. S4 could be a large system in its own right (with its own context diagram) and it can be designed in the same way as the current SUD.
- S5: There are (possibly optional) systems that monitor and control the SUD by exchanging information with it, for examples downloading settings or performing watchdog duties.
- SUD: The central *mediator* system that coordinates the data and control flow among the systems in the context diagram.

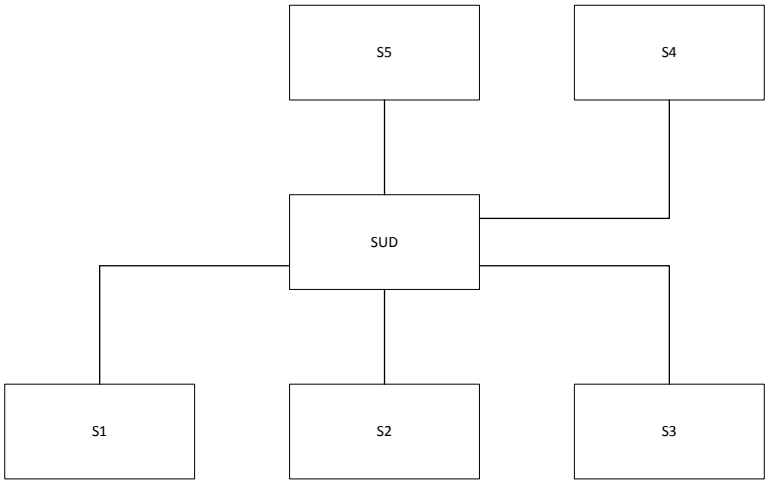

Figure 9.4 Canonical Context Diagram

The essential systems to design as early as possible are S1, S3, S4 and of course SUD. This is the minimal set to help kick-start the design process. If you can identify these systems then you will have considerably reduced project risk.

It is important to distinguish between the different domain categories. The most important and fundamental distinguishing feature or 'separator' for a domain category is the type of data that it produces. The reader can use the following initial separation mechanisms to classify applications:

- MIS: produce decision-support information from transaction data. This could take the form of multi-dimensional data that is processed by *reporting systems*.
- MAN: create products and services from raw materials.
- RAT: track requests in time and space.
- PCS: satisfy certain exceptional conditions at all times. Monitor exceptional events.
- ACS: provide active authorised subjects with access to passive objects/resources.

We now take a toy example in C++ of a system based on the context diagram in Figure 9.4. This choice is for motivational purposes only. We design the system using systems S1 (Input), S3 (Processing), S4 (Output) and SUD (mediator). The core process is to create a string, trim it (remove leading and trailing blanks using functionality from the Boost C++ string algorithm library), convert it to upper case and then finally display it on the console. The system can be seen as a very simple RAT system.
We implement the system in C++ in which SUD is a template class whose parameters correspond to the systems S1, S3 and S4. We agree on certain functions that these systems should implement so that SUD can communicate with them. We use *private inheritance* as a trick to implement interfaces.
It is a simple *push model* in the sense that data (in this case a string) is pushed from one system to another one.

The class definition is:

```
#include <string>
#include <iostream>
#include <boost/algorithm/string.hpp>

template <typename I, typename O, typename Processing>
	class SUD : private I, private O, private Processing
{ // System under discussion

private:
	// Define the requires interfaces that SUD needs
	using I::message;		// Get input
	using Processing::convert;	// Convert input to output
	using O::print;			// Produce output
	using O::end;			// End of program

public:
	void run()
	{
```

```
            // Core process, showing mediating role of SUD
            auto s = message();       // I, input
            convert(s);               // P, processing
            print(s);                 // O. output

            end();                    // O, signals end of program
        }
};
```

We now specialise this class as follows; we create single input and output objects and two converter objects:

```
// Instance Systems
class MyInput
{
public:
        std::string message() const
        {
            return std::string(" Good morning ");
        }

};

class MyConverter
{
public:

        void convert(std::string& s) const
        {
            // Process string using Boost String Algorithm library
            boost::trim(s);
            boost::to_upper(s);
        }
};

class YourConverter
{
public:

        void convert(std::string& s) const
        {
            s = std::string("Sorry, it's a secret");
        }
};


class MyOutput
{
public:

        void print(std::string& s) const
        {
            std::cout << s << std::endl;
        }

        void end() const
        {
            std::cout << "end" << std::endl;
        }

};
```

A test program now follows:

```
    // First instantiation
    SUD<MyInput, MyOutput, MyConverter> sud;
    sud.run();

    // Second instantiation
    SUD<MyInput, MyOutput, YourConverter> sud2;
    sud2.run();
```

Summarising, this code is an example of *policy-based design* (PBD) that we introduce in section 9.5. Properties of this solution are that specialisations of `SUD<I,O,P>` are classes and we cannot switch between different converters at run-time. Exercise 2 of this chapter discusses some extensions and modifications to the code.

9.4 Checklist

Before proceeding to detailed design we should look back at what we have achieved. We reconsider and reexamine the result and the steps that led to it. Some checklist questions are:

- Can we check the result? Can we see it at a glance?
- Can we write a C++ prototype (proof of concept) to motivate the system's core process?
- Can you use the result for some other problem?
- Can you check system responsibilities by working backward from the output to the input (Pólya's heuristic H6).

Of course, we cannot hope to discover every detail and what-if scenarios at this early stage but we must feel confident that we are not in for some unpleasant surprises, budget overruns and code rewrites in the later phases of the software development lifecycle. This is consistent with Jackson's observations in section 9.2.1.

9.4.1 A Special Case: Defining the System's Operating Environment

We conclude this section with a discussion of some topics that are relevant when systems are developed that must fit into existing software and hardware environments. In such cases we can fast forward to detailed design because of software dependency, portability and interoperability constraints. Not only do we need to find system interfaces but we may also need to add an extra level of indirection between SUD and these external systems by introducing *virtual machines* that can be realised by well-known design patterns such as *Proxy* and *Bridge* (see GOF 1995). The sooner we know what these dependencies are the more maintainable the resulting code will be because we know that the operating environment will change over the lifetime of the system. In particular, the boundary between specification and design deserves attention and then system modelling will become part of the design process.

We summarise this section by listing the kinds of systems that can emerge in practice. This activity is part of *system scoping* (Sommerville and Sawyer 1997):

- Systems that directly interface to the SUD. This includes systems that already exist as well as planned systems that are being developed simultaneously with SUD.
- Other systems (where known) that may coexist with the current system and defining the interactions between them.
- Related business processes.

It is worthwhile to examine the problem from these perspectives before moving to detailed design. We may discover functionality and requirements that would otherwise be difficult to realise in later stages of the software lifecycle.

9.5 Variants of the Software Process: Policy-Based Design

Policy-Based Design (*PBD*) is a compile-time programming idiom in C++ based on policies. A *policy* defines a class interface or a class template interface. A policy can contain member functions, member variables and inner type definitions. Policies focus on behaviour and somewhat less on structure. In other words, a policy is *syntax oriented* by which we mean that is specifies the names and input/output parameters that conforming classes must implement. Universal function wrappers on the other hand, are *signature oriented*. A common remark concerning policies is that some developers see them as the compile-time variant of *Strategy* design pattern (GOF 1995). More precisely, policies employ parametric polymorphism while GOF design patterns employ subtype polymorphism.

Before going into detail, we give an initial example of a GOF *Strategy* pattern, its implementation using subtype polymorphism and the corresponding implementation as a policy. We recall the definition of this design pattern:

Strategy defines a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

To this end, we create a function and a class that generate arrays of random numbers based on a number of generators, for example the infamous `rand()` function, the lagged Fibonacci generator in Boost and the Mersenne Twister algorithm in the C++ `<random>` library. The approach taken when applying the traditional *Strategy* pattern is to create an abstract base class containing a pure virtual member function that defines the

contract between the strategy and its clients. As usual, concrete derived classes implement this member function.

The base class representing the strategy interface is:

```
class Rng
{
public:
    virtual double rng() = 0;

    // Function call operator; Template Method pattern
    double operator () ()
    {
       return rng();
    }
};
```

Some implementations are:

```
#include <random>
#include <cstdlib> // rand()
#include <memory>
#include <vector>
#include <algorithm>
#include <iostream>
#include <ctime>
#include <boost/random.hpp>

class InfamousRng : public Rng
{
public:
    InfamousRng()
    {
       srand(std::time(0));
    }

    double rng() override
    {
       return rand();
    }
};


class MTRng : public Rng
{ // Mersenne Twister
private:
    std::mt19937 eng;
    std::uniform_real_distribution<double> d;

public:
    MTRng() : eng(std::mt19937(std::time(0))),
    d(std::uniform_real_distribution<double>(0.0, RAND_MAX))
    {
    }

    double rng() override
    {
       return d(eng);
    }
};


class LFRng : public Rng
{ // Lagged Fibonacci
private:
    boost::lagged_fibonacci607 eng;
    boost::uniform_real<double> d;

    boost::variate_generator<boost::lagged_fibonacci607,
          boost::uniform_real<double>> gen;

public:
LFRng() :
    eng(boost::lagged_fibonacci607(std::time(0))),
    d(boost::uniform_real<double>(0.0, RAND_MAX)),
    gen(boost::variate_generator<boost::lagged_fibonacci607,
```

```
        boost::uniform_real<double>>(eng, d))
        {

        }

           double rng() override
        {
           return gen();
        }
    };
```

This class hierarchy uses subtype polymorphism and new generators will be derived classes of `Rng`.
The next step is to define client code that uses this hierarchy. To this end, we create a class and a free function that both generate arrays of random numbers. These two entities can be seen as clients of the strategy algorithms:

```
    class ArrayGenerator
    {
    private:
        std::shared_ptr<Rng> gen;
    public:
        ArrayGenerator(const std::shared_ptr<Rng>& generator) : gen(generator) {}

        std::vector<double> randomArray(unsigned N) const
        {
           std::vector<double> result(N);
           for (std::size_t i = 0; i < result.size(); ++i)
           {
              result[i] = gen->rng();
           }

           return result;
        }

        void generator(const std::shared_ptr<Rng>& generator)
        { // Choose another rn generator

           gen = generator;
        }
    };

    // C-style function
    std::vector<double> CreateRandomArray(unsigned N, Rng& rng)
    { // Using Principle of Substitution

        std::vector<double> result(N);
        for (std::size_t i = 0; i < result.size(); ++i)
        {
           result[i] = rng();
        }

        return result;
    }
```

Test code (including the ability to change the generator at run-time) is:

```
        std::shared ptr<Rng> rng(new MTRng);
        ArrayGenerator gen(rng);

        auto rnArr = gen.randomArray(20);

        // Now switch to another generator
        std::shared ptr<Rng> rng2(new LFRng);
        gen.generator(rng2);
        rnArr = gen.randomArray(20);


        // C function as client of GOF Strategy
        std::size t sz = 20;
        InfamousRng rng3;
        auto rnArr3 = CreateRandomArray(sz, rng3);
```

We now look back and examine the consequences of using this design pattern (see GOF 1995). These consequences can be seen as advantages or as disadvantages:

1. We create families of algorithms that can be reused in multiple contexts. It avoids inheritance because classes use composition with client context classes containing a reference to a base *Strategy* class. We encapsulate algorithms in separate classes and we can vary each algorithm independently of its context.
2. Strategies provide different implementations for the *same* behaviour. We can thus use that particular strategy that best suits our efficiency and accuracy requirements. In this sense we call *Strategy* a *service-variation* pattern.
3. Clients must be aware of strategies. This introduces coupling because the client needs to know the name of the base class (in the above case it is called `Rng`). Each new algorithm must be implemented as a derived class of `Rng` which precludes functionality from other sources. This can be a disadvantage as it can lead to our having to use (multiple) inheritance.
4. Strategies increase the number of objects in an application. In many cases these objects must be created on the heap and then the application needs to define an object lifetime policy for them. This is an added complication and will add to development time.
5. Performance; the above code uses virtual member functions to vary behaviour which may have a performance impact in an application.

We now take this code and we adapt it to produce a solution based on the principles of PBD. In a sense it is a variation of the OOP approach. In global terms, it is a compile-time trick (based on *parametric polymorphism*) while the above solution is based on *subtype polymorphism*.

The central idiom in PBD is a class template (called the *host class*) that takes one or more type parameters as arguments. These types are specialised by *policy classes* and each one implements a particular implicit interface called a *policy*.

We now have defined enough terms to design the above problem using PBD. First, the host class in this case is:

```
template <typename RngPolicy>
      class RandomGenerator : public RngPolicy
{ // Policy-based design random number generator.
  // This is a host class.

public:
    // Function call operator, Template Method pattern
    double operator () ()
    {
       return rng();
    }

    std::vector<double> randomArray(unsigned N)
    {
       std::vector<double> result(N);
       for (std::size_t i = 0; i < result.size(); ++i)
       {
          result[i] = rng();
       }

       return result;
    }

};
```

In this case we see that the host class includes no hard-coded class names, only a template parameter that implements an *implicit interface*, in this case the implementation of the *function call operator* (this entails that the class becomes a function object). *Any* class that implements this operator can be used as a policy. This adds to the reusability and customisabilty of the host class. In fact, a library can be created consisting of the host class and a set of already *predefined implementations* for each policy. For example, we can define a policy that implements the `knuth_b` engine adaptor that returns shuffled sequences generated with the simple pseudo-random number generator engine `minstd_rand0`:

```
class KnuthRng
{ // knuth_b algorithm as a policy class
private:
    std::knuth_b eng;
    std::uniform_real_distribution<double> d;

public:
    KnuthRng() : eng(std::knuth_b(std::time(0))),
```

```
            d(std::uniform_real_distribution<double>(0.0, RAND_MAX))
    {
    }

    double rng()
    {
       return d(eng);
    }

    // Function call operator,Template Method pattern
    double operator () ()
    {
       return rng();
    }

};
```

An example of using PBD in the current context is:

```
// Policy-based classes
RandomGenerator<KnuthRng> policyRng;

auto rnArr4 = policyRng.randomArray(100);


// Use OOP classes with policy RNG
RandomGenerator<LFRng> policyRng2;

auto rnArr5 = policyRng2.randomArray(100);
```

We note that it is *not* possible to change a policy of the host class at run-time, in contrast to the design based on the GOF *Strategy*.
Incidentally, the code example in section 9.3.4 was an example of PDB with three policy classes for input, processing and output. We recall that the host class in this case is:

```
template <typename I, typename O, typename Processing>
    class SUD : private I, private O, private Processing
```

9.5.1 Advantages and Limitations of PBD

The term *policy-based design idiom* is discussed in Alexanderscu 2001 where it is informally defined and a number of examples are given to show how to use it. The summary of PBD from Wikipedia reads:

A key feature of the policy idiom is that, usually (though it is not strictly necessary), the host class will derive from each of its policy classes using (public) multiple inheritance. (Alternatives are for the host class to merely contain a member variable of each policy class type, or else to inherit the policy classes privately; however inheriting the policy classes publicly has the major advantage that a policy class can add new methods, inherited by the instantiated host class and accessible to its users, which the host class itself need not even know about.) A notable feature of this aspect of the policy idiom is that, relative to object-oriented programming, policies invert the relationship between base class and derived class - whereas in OOP interfaces are traditionally represented by (abstract) base classes and implementations of interfaces by derived classes. In policy-based design the derived (host) class represents the interfaces and the policy classes implement them. It should also be noted that in the case of policies, the public inheritance does not represent ISA relationship between the host and the policy classes. While this would traditionally be considered evidence of a design defect in OOP contexts, this does not apply in the context of the policy idiom.

We can see that the veracity of this description can be checked against examples. With this knowledge, we can take a critical look at PBD from both a language and design process point of view:

- The policy interface does have a direct, explicit representation in C++ code. This interface is defined implicitly (via *duck typing)*; hence it is documented separately and manually using comments. Duck typing is a layer of the programming language on top of typing and it is concerned with establishing the suitability of an object for some purpose. This suitability is determined by the presence of certain methods and properties (with the appropriate semantic meaning). In other words, a programmer is only concerned with ensuring that *objects behave as demanded of them in a given context* rather than ensuring that they are of a specific class. We note that duck typing can be defined for run-time ad compile-time scenarios but a discussion is outside the scope of this book.

Templates apply duck typing in a static typing context. This brings on a discussion regarding the advantages (or disadvantages) of static versus dynamic type checking that we postpone for the moment.

- Policy-based design may not be the best approach for all applications. It seems to be closely identified with C++ and with C++ templates. Other languages, for example C# generic interfaces with *constraints* provide a possibly more elegant solution. For example, a simple system in C# with two policies, representing input and output is:

```
interface IInput
{
   string message();
}

interface IOutput
{
   void print(string s);
}

// I/O stuff
class SUD<I,O>
   where I : IInput
   where O : IOutput
{

private I i_;
private O o_;

   public SUD(I i, O o)
   {
         i_ = i;
         o_ = o;
   }

   public void run()
   {
      o_.print(i_.message());
   }
}
```

- At the moment of writing, C++ supports neither interfaces nor constraints (or *concepts* as they are known in C++). Hence the above desirable property is not yet possible in C++. Consequently, the only solution seems to be to use public or private inheritance instead of composition.
- It is not easy to ensure (beyond trial-and-error and iteration) to create a good set of orthogonal policies.
- No support for *required interfaces*. This problem is caused by the fact that C++ does not have support for *events* and *delegates*. A workaround is to use the GOF *Observer* design pattern that implements *callbacks*. We have not tried to implement this pattern from a PBD perspective, preferring to use the Boost C++ *signals2* library which offers a more elegant solution.
- It is not clear how to apply PBD to design parallel software systems.

9.5.2 A defined Process for PBD

The Unified Design Process contains PBD as a special case. This means that we can resolve the issues surrounding PBD by first analysing and designing a problem. We now give some useful guidelines:

1. Find host class and policies. Our starting point is to create the system context diagram. In general, the host class corresponds to the SUD system while policies correspond to the SUD's satellite systems.
2. Does each system in the context diagram satisfy SRP (*Single Responsibility Principle*)? Is there overlap in responsibilities between the systems? Determine how to remedy these problems, for example by ensuring that each system processes its own kind of data. This becomes easier if you can determine which domain category your application is an instance of because the systems have orthogonal responsibilities in those cases.
3. Determine the *provides-requires* interfaces of each system in the context diagram. Referring to Figure 9.4 we can say that systems S1, S2 and S3 provide services to SUD while SUD provides services to systems S4 and S5 (or we can say that S4 and S5 require services from SUD). In some cases system S5 could provide services to SUD, for example the downloading of reference data that is needed by SUD.
4. Decide how you will design the system as a prototype in C++. Review the prototype and determine if it satisfies the requirements.

We continue with this topic in chapter 10 when we discuss detaiedl design.

9.6 Using Policy-Based Design for Drink Vending Machine (DVM) Problem

We now discuss how to create a simple prototype in C++ for the problem that we introduced in section 9.3.2. It is easy to create the initial context diagram in Figure 9.3 because DVM is a special case of an *Access Control System* (ACS) whose context diagram is known. From a project management perspective we have hopefully removed a major source of risk as we have created a stable design that we can extend to satisfy current and emerging requirements.

We choose to implement the system in Figure 9.3 using C++ templates as follows:

```
// Notation: G{Sys} == Generic system, not instantiated
template <typename GSource, typename GAuth, typename GResource,
    typename GSink, typename GMIS>

class DVM : private GSource, private GAuth, private GResource, private GSink, private GMIS
{ // System under discussion
private:

    // C++ does not support interfaces NOR concepts (constraints)
    using GSource::message;

    using GAuth::amount;
    using GAuth::increment;
    using GAuth::decrement;

    using GResource::displayProducts;
    using GResource::getProduct;
    using GResource::updateProduct;

    using GSink::notify;

    // Data, the amount left to spend
    int amt;

    // Other members

};
```

We see that policy interfaces have been formalised and these are implemented by policy classes. We design the SUD (*mediator)* that coordinates interactions with the policy classes as the following code shows:

```
void run()
{

    // 1. Get customer payment, Auth; Idle State,wait for coins
    std::cout << "Amount to insert: "; std::cin >> amt;
    GAuth::increment(amt);
    std::cout << "Amount in machine: " << amount();

    // 2. Choose product, Resource; Waiting for Selection State
    GResource::displayProducts();
    auto prod = getChoice();

    int change;
    try
    {
       auto info = GResource::getProduct(prod);
       std::cout << prod << " costs " << std::get<0>(info);
       change = amt - std::get<0>(info);
    }
    catch (int ex)
    {
       std::cout << "Product not found, bye\n";
       return;
    }

    // 3. Commit transaction (dispense product) and update database, Resource
    GResource::updateProduct(prod, 1);
    GResource::displayProducts();

    // 4. Inform Sinks and MIS
    GSink::notify(prod, true);

    // 4A. Give change
    std::cout << "The amount of change is: " << change;
```

```
}
```

This simple *get it working* code realises the core process in the system. One approach is to model this code as a state transition table or a UML statechart (Hatley Pirbhai 1988, Harel and Politi 2000).
This concludes the design of the generic framework. In order to test it we need to create a number of policy classes. The first group is given by the following policy classes:

```
class Auth
{ // This is where money is placed

private:
    int money_;
public:
    Auth(int money=0) : money_(money) {}

    int amount() const { return money_; }

    void increment(int amount) { money_ += amount; }
    void decrement(int amount) { money_ -= amount; }

};

using Product = std::tuple<int, int>;

class Resource
{ // (Simple) product database
private:
    // Name, price and supply
    std::map<std::string, Product> db;

public:
    Resource()
    {
       std::tuple<int, int> tup(2, 10);
       db.insert(std::pair<std::string, Product> ("COLA", tup));

       std::tuple<int, int> tup2(2, 20);
       db.insert(std::pair<std::string, Product>("ORANGE", tup2));
    }

void displayProducts() const
{
    std::cout << "**** Product Database **************\n";
    for (auto i = std::begin(db); i != std::end(db); i++)
    {
       std::cout << "Product: " << i->first << ", price: "
          << std::get<0>(i->second)
          << ", supply: " << std::get<1>(i->second);
    }
    std::cout << "*************\n";
}

std::tuple<int, int> getProduct(const std::string& product)
{
    auto prod = db.find(product);
    if (prod != db.end())
    {
       return prod->second;
    }

    throw - 1;
}

void updateProduct(const std::string& product, int quantity)
{ // Decrement supply by a given amount

    auto prod = db.find(product);
    if (prod != db.end())
    {
       std::get<1>(prod->second) -= quantity;
    }
}
};

class Sink
{ // Final recipient of end of transaction
private:
```

```
public:
    void notify(std::string prod, bool status) const
    {
       std::cout << "Transaction " << prod << ", status "
          << std::boolalpha << status << std::endl;
    }

};


class MIS
{
public:

       // TBD $$

};
```

In this case we see that `Resource` is a very simple database and that there is only a single sink class, namely `Sink`.

An example of use is:

```
// Single sink class
DVM<Source, Auth, Resource, Sink, MIS> dvm;
dvm.run();
```

9.6.1 Introducing Events and Delegates

In some cases we wish to broadcast the end of a transaction to several sinks. C++ does not support this functionality directly and we then propose alternative solutions:

a) The GOF *Observer* pattern (see GOF 1995).

b) Using collections of C++11 universal function wrappers (class *std::function*) (Demming and Duffy 2010).

c) Using Boost C++ *signals2* library (Demming and Duffy 2010).

We discuss how to apply option c) to the current problem. To this end, we send information to the console, to a simulated local database and to a simulated remote database. The corresponding *slot functions* are:

```
void Print(const std::string& prod, bool status)
{ // Free function

    std::cout << "I am a printer " << prod << std::endl;
}


// Lambda function
auto database = [](const std::string& prod, bool status)-> void
{ std::cout << "In due time,I save " << prod << "to database"; };

auto databaseBackup = [](const std::string& prod, bool status)-> void
{ std::cout << "Just backuped..."; database(prod,status); };
```

We now define a sink function object that plays the role of a *signal*. It triggers its connected slots when fired:

```
class Sink2
{ // Final recipient(s) of end of transaction, now a signal;
  //configure beforehand
private:
    boost::signals2::signal<void(std::string& prod, bool status)> sig;

public:
    Sink2()
    {
       sig.connect(2,Print);
       sig.connect(0,database);
       sig.connect(1,databaseBackup);
       // add

    }

    void notify(std::string prod, bool status) const
    {
       std::cout << "\n*** Notifications ***\n";
```

```
        // Emit signal and broadacast to slots
        sig(prod, status);
        std::cout << "*** Notifications, end ***\n\n";
    }

};
```

An example of use is:

```
// Multiple sinks in Boost signals2
DVM<Source, Auth, Resource, Sink2, MIS> dvm2;
dvm2.run();
```

9.7 Advantages of Uniform Design Approach

We summarise the findings and results. Some of the major advantages as we see them are:

- Separation of concerns: we avoid premature coding by decomposing the problem into loosely-coupled and cohesive systems. In other words, each system satisfies Single Responsibility Principle (SRP) and it has narrow interfaces with other systems.
- We perform up-front analysis to create maintainable and reusable software systems and code.
- A common vocabulary that all team members can understand. Improved communication between team members, especially during system evolution.
- Using the design artefacts as input to project management and software risk management.
- Adopting an assembly mindset to glue software components to form larger systems. Deliver software incrementally in a sequence of stable prototypes.

9.8 Summary and Conclusions

In this chapter we have introduced a process that allows us to analyse, design and implement software systems of any size and of any complexity. This process has been influenced by methods from *Structured Analysis*, *analogical reasoning* (techniques using domain architectures), developer experience in combination with trial and error. It allows us to design software as a sequence of stable prototypes. As special case we formalise the policy-based design approach in C++ and we showed how our approach subsumes it. We also gave some concrete examples.

In chapter 10 we discuss the process in more detail and we give guidelines and examples relevant to computational finance. We note that the process in this chapter is *data-driven* in the sense that for each system we find the data that it processes and that it produces. This leads us to chapter 10 that discusses how to design and implement these deliverables in C++.

9.9 Exercises and Projects

1. (Brainstorming Question) Consider the steps 1,2,3 and 4 in section 9.1. Answer the following questions from the perspective of software development (preciseness is not a concern at the moment; the questions get you thinking about how to analyse and solve problems):

a) Which steps would you execute (and how much time would you spend on each step) for the following kinds of projects?:
   - Adding/modifying functionality in an existing application.
   - A project to test the feasibility of a new algorithm or model.
   - A new project that has no precedence (has never been done before) in your organisation.
   - Integrating third-party software into your application.

b) When working on software projects whose functional requirements are missing, incomplete or even wrong determine how the following *heuristics* are useful in getting a better understanding of what needs to be designed (see Pólya 1990 and Pólya 1990A for more discussions and more examples of heuristics):
   - H1 *Analogy*: find a problem that is similar to the current problem and solve it.
   - H2 *Variation of the Problem*: vary or change the problem to create a new problem whose solution helps you to solve the original problem.
   - H3 *Auxiliary Problem*: reduce the scope by finding a subproblem or side problem whose solution helps you solve the original problem.
   - H4 *Precedence*: find a problem that is related to the current problem and that has been solved before.
   - H5 *Diagrammatic Reasoning*: can you draw a picture of the problem?
   - H6 *Working backward*: start with the goal and work backward to something that is already known.

- H7 *Decomposing and Recombining* : decompose the problem and recombine its elements in some new way.
- H8 *Auxiliary Elements* : add a new element to the problem in order to get closer to a solution.
- H9 *Generalisation* : find a problem that is more general than the current problem.
- H10 *Specialisaton* : find a problem that is more specific than the current problem.
- H11 *Induction* : solve the problem by deriving a generalisation from some specific examples.

The goal of applying these heuristics is to reduce risk and to get a better understanding of the problem that we are attempting to design. In general, you oscillate as it were between more general and more specialised versions of the current problem.

c) Consider your last software project that you worked at. Which of the heuristics in part b) did you use – either consciously or unconsciously? Which single heuristic would you have liked to use but did not?

d) Which of the heuristics in part b) are addressed by the Gamma (GOF*) design patterns*? We recall the definition of a design pattern (see GOF 1995):

In software engineering, a design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. A design pattern is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations.

Which heuristics are not addressed or subsumed by design patterns? As a guideline, take a pattern that you are familiar with (for example, *Strategy*) and compare it with each heuristic in turn.

2. (Extending the PBD design of Section 9.3.4)

We ask some questions regarding the code in section 9.3.4. The objective is to extend and modify the code in order to get some more hands-on experience with a small system before moving to more complex applications in computational finance that we discuss in later chapters. Answer the following questions:

a) Generalise the code so that it works with any data type and not just with strings. Furthermore, the input data type `T1` is not necessarily the same as the output data type `T2` and hence the converter becomes a class that maps instances of `T1` into instances of `T2`. You will need to use *template template parameters*:

```
template <typename T1, typename T2,
      template <typename T1> class Source,
      template <typename T2> class Sink,
      template <typename T1, typename T2> class Processing>

class SUD : private Source<T1>,private Sink<T2>, private Processing<T1, T2>
{ // System under discussion

      // TBD
}
```

Implement the code and test with a some scalar and composite data types.

b) In this part we implement the converter using a universal function wrapper that will be a data member of SUD, namely an instance of `std::function<T2 (const T1& t)>`. The class declaration now becomes:

```
template <typename T1, typename T2,
         template <typename T1> class Source,
         template <typename T2> class Sink> {…};
```

Implement the code. Is this a more flexible solution than in part a) and if so in what respects? In particular, implement the converter functionality using free functions, lambda functions, static member functions, function objects and binded member functions.

c) Modify the original code to support a new requirement: system S1 opens a file and processes each of its records, system S3 converts the strings to trimmed upper case strings and system S3 writes the strings into a new file.

3. (Extending the DVM Code from Section 9.6)

The initial code for the DVM problem needs to be extended to suit a wider range of customers. Some new features and requirements are:

a) Remote management of multiple DVM machines, for example monitoring and control of malfunctioning machines.
b) The ability to pay using credit cards.
c) Different kinds of user input panels to suit different styles.
d) The facility to choose a range of products and then pay for them as a single transaction, for example a lunch packet consisting of a drink, sandwich and chocolate bar.

Answer the following questions:
a) Determine the changes that need to be made to the context diagram in Figure 9.3 in order to accommodate these new features. For example, you may need to design new satellite systems or existing systems may need to be modified in some way.
b) Which of the following design patterns can be used to implement the required features? *Proxy*, *Bridge*, *Remote Proxy*, *State Machine*, *Observer*, *Mediator*.
c) Give a rough estimate (upper bound) of the number of classes that you will need to create in order to realise these features.

4. (Alternative Implementation of the DVM Problem)
The solution of the DVM system in this chapter was based on C++ templates and PBD. This may not be the most suitable approach in all situations. More importantly, developers may feel more comfortable with class hierarchies and subtype polymorphism (which many C++ legacy systems use). For this reason, implement DVM using class hierarchies and `virtual` member functions. To this end, we have implemented DVM in C# using interfaces and generics and the code is almost in the form that we wish to see in C++. The interfaces (these will be *pure abstract classes* in C++) and SUD interface are:

```
// Define the interfaces/contracts
public interface ISource
{

        string message();
}

public interface IAuth
{
       int amount();
       void increment(int amount);
       void decrement(int amount);
}

public interface IResource
{
       void displayProducts();
       Tuple<int, int> getProduct(string product);
       void updateProduct(string product, int quantity);
}

public interface ISink
{
       void notify(string prod, bool status);
}

public interface IMIS
{
       void notify();
}
// ** end of interfaces

// Notation: G{Sys} == Generic system, not instantiated
public class DVM<GSource, GAuth, GResource, GSink, GMIS>
       where GSource : ISource
       where GAuth : IAuth
       where GResource : IResource
       where GSink : ISink
       where GMIS : IMIS
{

       private ISource iso_;
       private IAuth ia ;
       private IResource ir_;
```

```
        private ISink isi_;
        private IMIS im_;


         // More
}
```

Answer the following questions:

a) Implement DVM in C++ using C# as baseline requirements. Test the program and check that the output is the same as before.
b) Compare all solutions in terms of efficiency, maintainability, extendibility and replaceability of components at run-time. The answers will help you decide which option is more suitable in a given context.
c) Consider the option of using C++ universal function wrappers and the Boost C++ *signals2* library as a means to allow SUD to communicate with the external world. What are the advantages and disadvantages when compared to the other solutions? Consider issues such are shared/non-shared state, event-driven programming, maintainability and whether data is pushed or pulled.

5. (Which Category does an Application belong to?)

In this book we discuss a number of numeral applications in computational finance that we implement as we progress in the book. In this exercise we wish to gain *deep insights* into how to understand problems even before we write a single line of code. Consider the following applications:

- A1: Using the binomial method to price one-factor options.
- A2: Creating an interpolation library.
- A3: Affine pricing models for interest-rate derivatives.
- A4: Optimisation algorithms.
- A5: Parameter estimation algorithms.
- A6: One-factor and two-factor Monte Carlo option pricers.
- A7: Computing the mean error of a finite difference approximation of a stochastic differential equation (SDE) (see Kloeden, Platen and Schurz 1997, page 118).
- A8: Option pricing using PDE models.
- A9: Postprocessing routines to test the accuracy of a numeric scheme by comparisons with a method that produces reference values to test against.
- A10: Creating data, modules to be used by other systems.
- A11: Configuring a complete application.

Answer the following questions:

a) Determine which category (or categories) these applications are instances of. In general, the most important categories are MAN, RAT and MIS.
b) Create a basic system context diagram for these applications by identifying the systems and their responsibilities in the diagram (take Figure 9.4 as reference model).
c) Identify the similarities and differences between the system context diagrams in the case of PDE, binomial and Monte Carlo option pricers.
d) Can you identify reusable artefacts such as code, libraries, data types and containers that can possibly be used in several applications?