Object and Components in a Financial System

Anatomy of a Settlement System Development

Mark Collins-Cope
Hubert Matthews

Ratio Group Ltd.
17/19 The Broadway
Ealing W5 3NH

Email: info@ratio.co.uk
Web: www.ratio.co.uk

We Know the Object

Table of Contents

1. INTRODUCTION 3
2. ARCHITECTURE 4
3. COMPONENTS 5
4. PERSISTENCE 8
5. PATTERNS 9
6. SUMMARY 10
7. REFERENCES 10

1. Introduction

This paper is one of a family of four papers (see References) describing different aspects of the design and implementation of a financial settlement system. The system was a financial "back office" settlement system which was developed to replace an existing legacy system. Designed to be both flexible and high performance, the system was developed over a period of three years using a component-based approach and object-oriented techniques. The system comprised over one million lines of C++ source code (of which 75% was machine generated), divided into 1,500 classes spread over 30 component subsystems.

A component, for the purposes of this paper, may be defined as: *a collection of collaborating classes with a well-defined interface that is distributed in binary form.* A component is thus typically made up of a number of classes which interact with each other to provide the functionality accessible to the outside world via its interface.The interface to a component is typically made up of one of two facade (front-end) classes that hide the internals of the component. Internal project distribution in binary form - as libraries in perhaps *.dll*, *.a* or *.obj* form - guarantees that there is no tampering their internals.

This document provides a summary overview of the key elements of its three companion papers (see References), and adds some additional high level discussion on the approaches taken within the project.

The remainder of this document is structured as follows:

- section 2 gives an introductory overview of the underlying architecture of the settlement system;
- section 3 provides summaries key aspects of the component based approach used to develop the system;
- section 4 overviews the persistence (object to relational mapping) componentdeveloped for use with the system;
- section 5 discusses the use of patterns within the system;
- section 6 is a summary of this paper.

2. Architecture

The architecture of a system may be described as defining the structural relationship between the individual components that together create the system as a whole. Architecture was considered a vital element of the settlement system development, and there was a continual focus on system architecture and component structure throughout the project lifecycle. It is important to realise this was not a straighforward task, there were many difficult dilemmas along they way, and many alternative solutions to particular issues to be considered. The focus on architecture did, however, contribute significantly to the success of the component based approach adopted for the development.

Underpinning the architecture of the system is a layered architectural model. The term 'layered architecture' reflects the practice of drawing horizontal bands across diagrams such as that found in Figure 1, grouping components by the degree to which they are specific to any particular application - or conversely, the degree to which they might be potentially re-used in other applications.

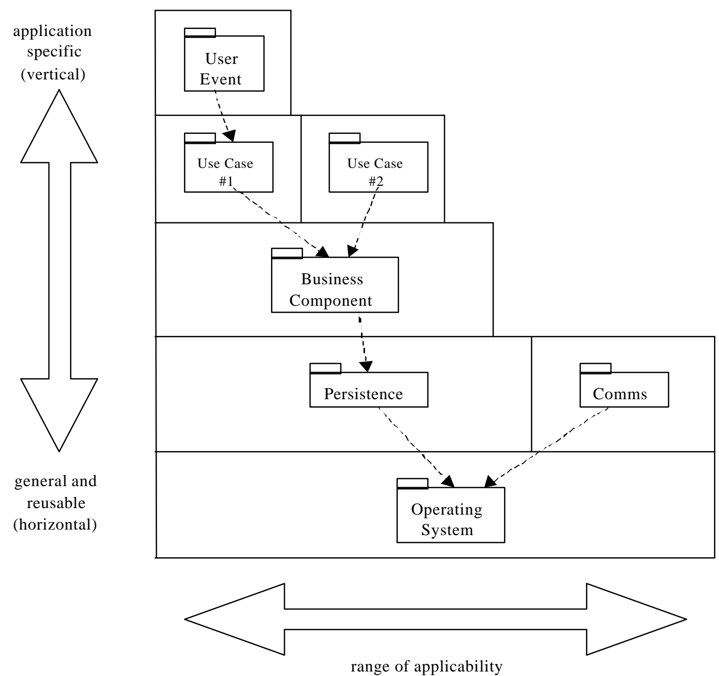

Figure 1 - layered architecture for CBD

A full discussion of the layered architecture presented here is deferred to the companion paper "Components in Financial Systems"**,** however the key benefits derived from the architecture were:

- it assisted in the seperation of the technical and business domain parts of the system, allowing expertise to be applied appropriately,
- it provided a framework for decision making during component design,
- disciplined application of the 'dependencies go downwards' rule ensured bottom up integration and testing was feasible, and
- it has enabled some of the components in the system to be at the least *potentially* re-usable.

Further discussion of the business components and persistence elements of the system can be found later in this document.

RATIO

3. Components

The system discussed in this paper was developed using a component based approach, each component existing solely within a particular layer of the architecture introduced in Figure 1. In the second and third layers in this model (the activity and business component layers) there is a focus on implementing business - rather than technology - related functionality.

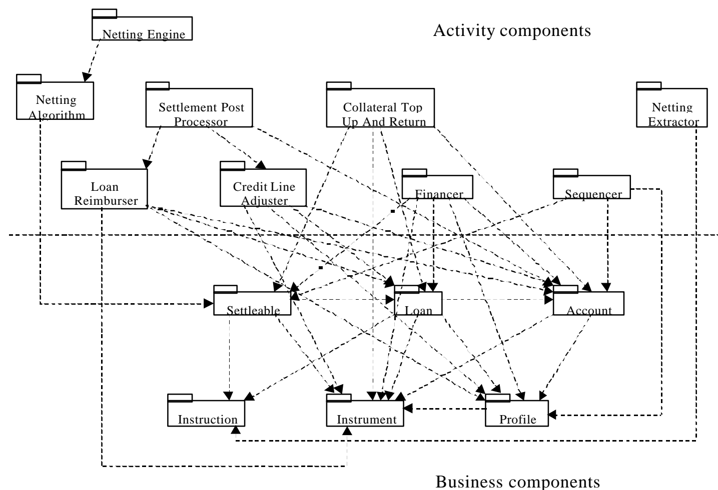

Figure 2 - business components across two layers

Figure2 shows the components that exist within the two business related layers of the system. The dotted lines show how the components depend on each other, and are combined to create the full system.

Perhaps the most important point to note from this diagram is that the higher layer components are fundamentally concerned with the implementation of business control logic and rules (and hence have verb-like names), whilst the lower layer encapsulates business components or objects that would not be unfamiliar to a business user of the settlement system (and hence have noun-like names).

A detailed discussion of the component approach to development can be found in this paper's companion document "Components in Financial Systems." For now, we note that perhaps the most difficult questions to answer in a largely bespoke component based development is: *how do you find (or define) the components in the first place?* There is no simple answer to this question - components obviously have to be designed, and this is clearly a creative engineering activity requiring substantial disucussion and brainstorming. However the experiences of the settlement system development process have led us to the following summary guidelines, which must in practice be mixed with considerable brainstorming and evaluation of the pros and cons of alternatives.

- A component must have a well defined façade (front-end) object, through which all communication to the component passes; component internals must not be visible to the outside world.
- Within the business component layer (third from top), components encapsulate business entities (e.g. account, instrument, etc.).
- Within the activity layer (second from top) components often encapsulate business processes and rules.

- The objectives in creating a component are that it should be, internally, as cohesive as possible, and that externally, it should be as loosely coupled as possible with other components.
- Pay heed to the overall architectural structure of your project (perhaps using a reference architecture such as that described in Figure 1), work within it, but do evolve it *if* necessity dicates; to achieve this it is necessary to have some members of the team focus on the *overall* architecture of the system, not just the particular component they may be working on.
- Follow the architectural princples: no cyclic dependencies, downward dependencies between layers only.

From a "process" perspective, the following sumarises how to converge on a component breakdown:

- start with a definition of the requirements of the system, and a high level business object model;
- first cut component divisions can be found by grouping objects in the business object model;
- from the requirements, extract the key 'services' that the system will provide, and use these to investigate the potential component divisions; do this by developing sequence diagrams at the inter-component level;
- iterate until there is stability in your component breakdown;

One important point of experience is worth emphasising here: don't expect to get the components and their interfaces right first, or even second, time. The more functionality is added to the system, the more likely it is that an initial component breakdown will prove inadequate. The settlement development team commented that they had to rely heavily on the stability of component interfaces, and that changes in interfaces had a wide-reaching effect on the system. For this reason, interface changes in the settlement application had to be approved by the system architects, who had to balance the need for the change (to add new functionality, for instance) with the need to maintain stable interfaces.

This may sound difficult, however the benefits of adopting a component approach can be considerable, as:

- Team structure naturally follows component structure: individual teams work on individual components, and the division of labour is far simpler than on 'procedural' projects.
- Components create a natural structure for testing: unit testing at the component level, and sub-assembly integration testing for groups of components; It is also simple to create test stubs should one component take longer to develop than expected.

- Source code and configuration management becomes much easier, as these naturally follow from the component structure,
- Performance issues, when they arise, can often be traced back to an invidual component; which can be improved without disrupting the rest of the system,
- Comparing with a object development not using a component based approach, having a clear high level focus on a restricted number of component interfaces makes the complexity of the system manageable (imagine trying to manage the relationships between 1500 classes without a super-structure to assist).

In general, the settlement development team considered that the component approach was beneficial for a number of reasons. Since components do not share any data, and encapsulation is strictly enforced by the compiler then there can be no possible interaction between components except via the published interfaces. Therefore all interaction is visible in high-level inter-component sequence diagrams and can be planned at the architectural level. They considered the benefits of this to be far-reaching, in that:

- the system is easier to comprehend as all interaction is visible and documented,
- implementers of a component need only concern themselves with the internals of their own component and the interfaces of collaborating components,
- components are strongly de-coupled, allowing for parallel implementation efforts,
- internal changes to a component are invisible externally,
- interfaces can act as distribution points in a distributed system.

The team also felt that the structure of a component-based system degraded much less during the course of development compared to similar systems built using procedural techniques. One reason for this was because of the strict policing of encapsulation by the compiler, which did not allow for shortcuts.

There was a cost to pay for these benefits. Encapsulation means that data cannot be shared, so programmers must pass all required data through the interfaces of components. This can lead to very long parameter lists on functions (17 parameters in one case!). The balancing argument to this is that it makes all data passing explicit, thereby forcing decisions about who needs what to be made early on in the project. In one case (AccountContext) the amount of data to be passed was sufficiently large that it was gathered together into an object (AccountContextData) which was passed as a single parameter instead.

A final point. Components can be likened to the 'technical libraries' of procedural systems (string handling, time and date handling, etc.), however, there is one major difference that should be apparent from the discussion so far: components are used to sub-divide not only the technical domain but also the business domain. In procedural systems one would not find an

'Account' library, but this is precisely the type of business component defined within this system.

4. Persistence

The settlement system we are discussing here was, for a variety of reasons, built on top of a relational database management system. Accordingly, one of the largest components of the system was the persistence framework. Persistence is a term used to indicate that the lifetime of an object will be longer than the program that created it, a situation true in almost all business-oriented applications today. As well as 'storing' objects somewhere, the persistence framework in question had to deal with more tricky problems such as caching for high performance and automatic write-back issues.

The persistence framework developed for the settlement project is an excellent example of the design principle of clean abstraction applied on a large scale. A clean abstraction is presented by a component when it provides a simple interface and has an associated 'usage' model that requires little or no knowledge of its internal details, and which is described in terms of concepts familiar to its *user* (perhaps the best example of clean abstraction used on a wide scale is the 'desktop' model presented in many GUI environments).

The abstraction presented to application developers by the persistence component of the settlement was that *all objects are in memory all of the time*. They aren't, of course, but that is the whole point about a good abstraction, as far as the user of the persistence mechanism is concerned, objects can be considered to be in memory at all times. In reality, as the settlement system is running, objects are being loaded and stored onto the underlying database all the time, and moved into and out of one of two internal caches. However when writing their code, application developers did not have to concern themselves with most of this necessary but distracting activity. The persistence framework took this burden from them, allowing them to focus on implementing the business functionality required of the system.

Whilst we leave a detailed discussion of the internals of the persistence component to this paper's companion document ("Persistence: Implementing objects over a relational database"), here is a brief summary of how it operates.

- all persistent objects are derived from a common persistence base class,
- application developers accessed persistent business objects via smart pointers,
- the smart pointer mechanism, in conjunction with the persistence framework, works out where the object is (in the application's memory, in a cache, or in a database table), and if necessary, brings it into application memory,
- the application developer works on the object as necessary,
- at the end of the business transaction, any objects which the application developer has modified are automatically written back to the database.

The development team noted that changing the structure of a database table was easy as only one class would be affected, and compared favourably with the pain of trying to do this in a procedural implementation, particularly one where insufficient care was taken to isolate the database – e.g. one with embedded SQL statements.

5. Patterns

In object-oriented software a pattern is a reusable design. It captures a simple and elegant solution to a specific problem and documents it in a succinct and easily applied form. The seminal work on software patterns is the book “Design Patterns: Elements of Reusable Object-Oriented Software” by Gamma, Helm, Johnson and Vlissides [GoF], whose authors are colloquially referred to as the “Gang of Four”. Their book documents in a standard way 23 common design patterns that they distilled from a wide range of sources. Martin Fowler [Fowler] extended the idea of patterns to encompass analysis models in his book 'Analysis Patterns.'

The settlement development team were both consumers and producers of patterns. Consumers, in the sense that at least ten of the classic Gang of Four patterns were used in the design of the system, and producers, in the sense that a number of design and analysis type patterns were 'uncovered' during the development.

Analysis style patterns included:

- a multicurrency conversion pattern, dealing with how to encapsulate currency conversion in a clean, simple and elegant manner,
- an instruction/transaction/movement pattern, dealing with the relationships between these settlement domain entities, and
- a loan pattern, dealing with the seperation and management of liquidity and collateral against loans.

Whilst these may not be patterns in the strict sense (of at least 3 known usages), we feel that they are worthy of the title if only for one simple reason: the sheer amount of effort and brainpower that went into discussing the numerous modelling options that were available in each case. Many alternatives to the instruction/transaction/movement pattern, for example, were discussed by the project team, before the final (and interestingly most simple) model was adopted (for further discussion see the companion paper "Patterns in Financial Systems").

Other 'design' type patterns used by the system included:

- a data driven business rule pattern, which enabled new business rules to be added without having to modify existing code (used in conjunction with the loan pattern), and
- a flexible ordering pattern, used to ensure a flexible approach to the sorting of movements before sequencing; achieved by the addition of new 'comparator' type objects to the system.

The breadth of applicability of the patterns concept to the project demonstrates one thing if nothing else: time and money invested in learning patterns is time and money well spent. The application of patterns to the project added significantly to the quality of the development.

6. Summary

This paper has discussed the architectural, component, pattern and persistence aspects of the development of a large settlement system. The key messages of this paper can be summarised as follows:

- The component based approach to development works, and assists greatly in managing the complexity of large applications.
- Architectural vision underpins good component based design.
- Patterns provide a clearly useful starting point that short-cuts a learning curve that might otherwise have been much slower.
- Hiding the details of the persistence aspects of the development from application developers enables them to focus clearly on solving business rather than technical problems.

At the beginning of this project, the development team - whilst experienced in small object oriented development - had little experience of the component based approach to large OO systems. Judicious investment in training (C++, OOA/D using UML, principles of OOD, design patterns, analysis patterns and framework design), self-learning (the team read numerous books on OO and CBD), use of external consultants at certain key points, and a great deal of talent and hard work lead to the successful delivery of over a million lines of code for a large settlement system - on time and to budget. This was achieved following the "principles" of object and component based design, and has led to the delivery a quality system that can be readily enhanced in the future.

7. References

"Components in Finance Systems", 1999.
"Patterns in Financial Systems", 1999.
"Persistence: Implementing Objects over a Relational Database", 1999.