Components in Financial Systems

Hubert Matthews
Mark Collins-Cope

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

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 developed to replace an existing legacy system. Designed to be both flexible and high performance, the system was developed over a period of 3 years using a component-based approach and object-oriented techniques. The system comprised over 1,000,000 lines of C++ source code (of which 75% was machine generated), divided into 1,500 classes spread over 30 component subsystems.

In this paper, examples of components from the settlement domain are discussed with reference to wider architectural issues such as layering, interfaces and dependency management. The process by which a component-based system may be designed and the implications for team working are also discussed, as are issues of designing components for reuse.

The audience for this paper is intended to be designers and analysts in all fields, particularly those involved in component based design or development. The first part of the paper assumes some knowledge of financial systems, particularly the settlement domain, whereas the remainder of the document is more general in nature. UML object models and sequence diagrams are used to illustrate the packaging and decomposition of the design.

For the purposes of this paper, we offer the following definition of a component:

A component is a collection of collaborating classes exporting a well-defined interface that is distributed in binary form.

This definition is adequate to describe most possible examples of components ranging from a single object file through DLLs to COM objects and Enterprise Java Beans. In this paper we shall be examining components developed in a Unix environment that have been implemented in C++ as shared libraries, although the discussion applies equally well to other types of component.

2. Components in a settlement system

In this section of this paper we discuss the structure, functionality and inter-relationships between components as used in the settlement system under discussion.

The settlement process involves the receipt of matched and validated instructions from clients and making changes to their respective accounts. This may involve the use of credit lines or the creation of loans, for which collateral is required. A process called netting is used to produce a subset of transactions that can be settled together within the constraints of the settlement house and its clients.

2.1. Domain specific components

Figure 1 details the overall structure and inter-relationships between the domain specific components in the system. The components shown are separated into two major layers: the activity (or use case) component layer, which is concerned with issues of overall application control, and the business component layer, concerned with the provision of re-usable business components that encapsulate domain specific objects. Dependencies between components are indicated using arrows.

Examining the dependencies between components in this diagram, we can see that:

- all inter-layer dependencies (those between a component in one layer and a component in another) are in a downwards direction - going from the less stable application specific components in the activity layer to the more stable, more general business components. This helps ensure that the elements of the system most likely to change - the activity or use case components - are not depended upon, helping minimise the likelihood of change propagation throughout the system.
- when considered as a whole, the dependencies between components are acyclic - there is no direct or indirect route by which a component is dependent upon itself. This maximises the independence of the components and ensures that bottom up integration and testing of the system is at least feasible (components between which there is a dependency cycle effectively form one large component, as every component ends up dependent on every other).
- the instrument and the account components are heavily used. This is to be expected, given the nature of the settlement domain, and the rich functionality that these two components encapsulate.

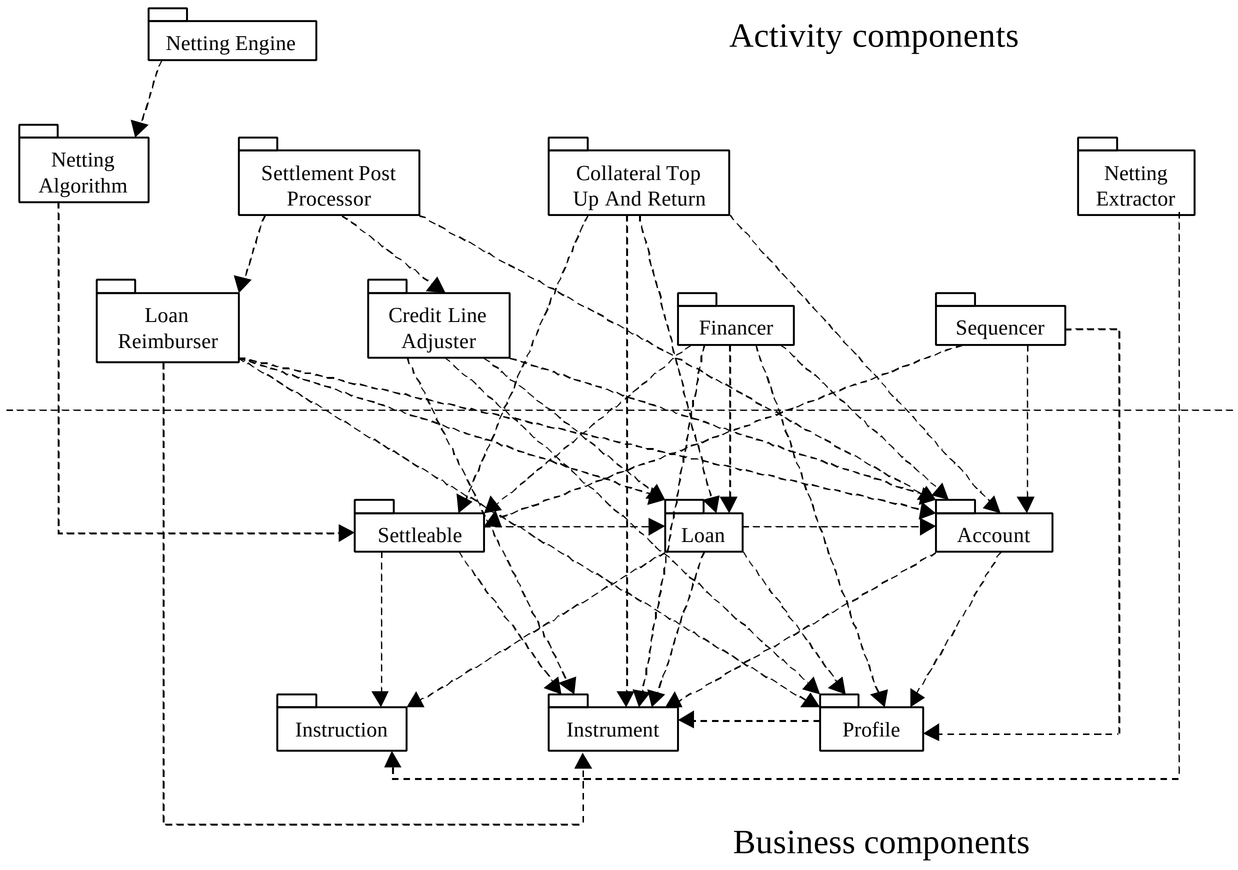


Figure 1 - Domain specific components of the system

2.2. Business layer components

The business layer components of the system encapsulate domain visible objects and their associated business logic. They are heavily used by the components in the higher activity (or use case) layer, and are typically based on well understood entities within the business area. They are independent of the specifics of the application itself, and are reused across different functional areas - for example netting and settlement post processing.

The major business components of the settlement system are as follows:

- Account
- Instruction
- Instrument
- Loan
- Profile
- Settleable

These are discussed in more detail below.

2.2.1. Account

An Account represents customers and depositories within the system. It records the effects of business actions such as buying, selling, borrowing, lending and pledging. Its key responsibilities include: position management; booking; risk limits; collateral and management of account characteristics.

The Account component provides two interfaces to its users: IntAccountFactory and IntAccount. The former interface is used to create and retrieve accounts that are then manipulated by the latter.

2.2.2. Instruction

The Instruction component models the instructions that customers give to the settlement house regarding what they have bought and sold in any given transaction. It represents one party’s view of a transaction. Instructions track their own lifecycle as they pass through the different stages of the settlement process, and also handle issues of validation and matching.

2.2.3. Instrument

The Instrument component models real world financial instruments such as cash and securities. It is also responsible for the quantity and type of that instrument and its instrument group. Both fungible (anonymous/interchangeable) and non-fungible (specifically identifiable/non-interchangeable) instruments can be handled.

Key instrument functionality includes: arithmetic and comparison; valuation; and valuation for use as collateral - including calculation of haircut and uplift.

2.2.4. Loan

The Loan component models the lending of cash and securities. It also handles the principal, credit and collateral for a loan in the form of an instrument quantity and movements that are generated by the setting up of the loan.

There are three interfaces to the Loan component: IntLoanFactory for loan creation, IntLoan for standard manipulation, and IntLoanAction - a sub-interface used for implementing the details of a loan.

The component implements all of the 17 different types of loans currently used within the business area for which the system was developed.

2.2.5. Profile

The Profile component contains global reference data that is used by other components. Its primary use is to hold configurable parameters that are required for example when financing, such as details of the risk taker, the asset taker, the asset provider and associated loan types.

2.2.6. Settleable

The Settleable component implements the core functionality of settlement, providing classes such as movements, transactions and transaction groups to do this. Its functionality includes controlling the processing logic of settlement, i.e. whether provision checking, cash financing etc are performed.

There are four interfaces to this component:

- IntSettleableFactory
- IntPendingAmountHolder
- IntTransaction
- IntMovement (the primary interface).

The factory interface is for creation and retrieval of settleables. The pending amount interface deals with the pending arrival of securities or cash, and the transaction interface handles the grouping of movements and instructions into transactions. The movement interface provides the major functionality of the settlement engine, in particular methods to attempt to settle movements, sequence the order of movements, and status functions to determine why an attempted settlement failed.

2.3. Activity (or use case) layer components

Activity components encapsulate the algorithms, rules and procedures that manipulate the business components discussed in the previous section. In this section, we shall examine the major components in this layer, namely:

- Collateral Top-Up and Returner
- Credit Line Adjuster
- Financer
- Loan Reimburser
- Netting Engine
- Netting Algorithm
- Netting Extractor
- Sequencer

- Settlement Post Processor

Unlike the business components, activity components do not contain persistent objects. Most capture one significant piece of functionality as seen from a user's perspective – and can therefore be thought of as implementing a use case.

2.3.1. Collateral Top-Up and Returner

This component re-evaluates the collateral pledged towards loans and performs either a top-up or a return of collateral as necessary.

2.3.2. Credit Line Adjuster

The Credit Line Adjuster adjusts customer's credit lines following updates made during the settlement process. This involves the following steps:

- checking for changes in a customer's credit lines by querying the Account component
- if the credit line has been over-utilised then loans are moved from one financing type to another
- if the credit line is not fully utilised then additional loans are moved to the cheapest financing type

These steps occur only after loan reimbursement.

2.3.3. Financer

The Financer deals with the creation of cash loans. Although the component interface supports the creation of Securities loans, this functionality is not currently implemented within this component.

2.3.4. Loan Reimburser

Loan reimbursement (as implemented by this component) happens after settlement, and involves trying to reimburse as many loans as possible. This involves the following steps:

- checking the available liquidity on the asset taker's account
- closing loans of a specified type which are ordered by various business rules including age
- returning collateral to the asset taker
- reducing the asset taker's credit line utilisation

Each of these steps is undertaken by the Loan component with the Loan Reimburser sequencing the steps and logging any exceptions.

2.3.5. Netting Algorithm

The Netting Algorithm component takes movements extracted by the Netting Extractor and groups them for settlement.

2.3.6. Netting Engine

The Netting Engine maintains overall control of the netting process. It performs common pre- or post-processing before invoking the Netting Algorithm component.

2.3.7. Netting Extractor

The Netting Extractor marks movements as “extracted” so that they are not updated during the netting process.

2.3.8. Sequencer

The Sequencer orders movements according to both the customer’s and the settlement house’s sequencing options.

2.3.9. Settlement Post Processor

After settlement the post processor attempts to reimburse loans and adjust credit lines for all of the customer accounts. It will:

- reimburse outstanding cash loans on “free held” positions
- adjust credit lines on “free unconfirmed” cash positions with outstanding loans

It uses the Loan Reimburser and the Credit Line Adjuster components to do this.

2.4. Example of component interaction

Having examined a number of components in isolation, we will now look at how these components interact to implement a given requirement. We will look at the collateral top up and return component which re-evaluates the collateral pledged towards loans and performs either a top-up or a return of collateral as necessary.

A simplified object model of the relevant parts of the system is shown below:

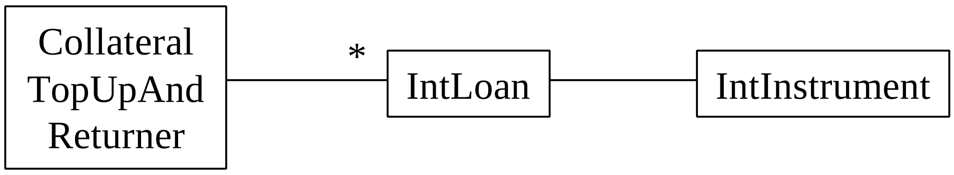


Figure 2 - Component level object model

This shows that we are examining a number of loans, each of which has an associated instrument that has been pledged as collateral for that loan. Note that at this level of abstraction we are interacting with components' interfaces so the object model shows interfaces rather than components. (The "Int" prefix is a naming convention used by the settlement application to denote a interface.)

The high-level sequence diagram for this operation is:

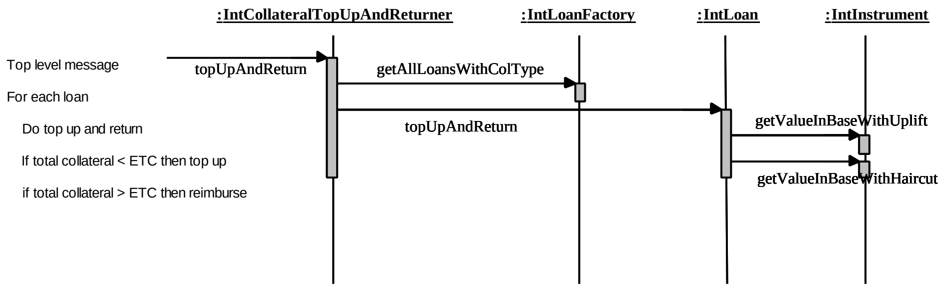


Figure 3 - High-level sequence diagram of component interaction

Here we use the factory interface IntLoanFactory to retrieve the relevant loans, which we then process in turn by calling their topUpAndReturn method. The instrument's collateral valuation methods are then used to determine if the level of collateral needs to be changed either upwards or downwards.

The major point of interest from this diagram is that all component interactions occur through the published interfaces, which guarantees that the encapsulation of the components is preserved and that any interactions are visible at the architectural level. This will be discussed further in the second section of the paper.

3. Architecture and reuse

3.1. Architectural layering

Architectural layering is an important concept in the design of software systems, one that leads to a flexible, re-usable and stable software structure. It is a metaphor which divides the structure of an application, or more precisely the component packages (packages that are intended to be components) from which it is made, into an ordering based firstly on the degree of application/domain specificity of the classes contained within the packages, and secondly on the dependencies between these packages. Figure 4 shows an example of layering showing how the reusability of a component varies according to how specific its area of application is.

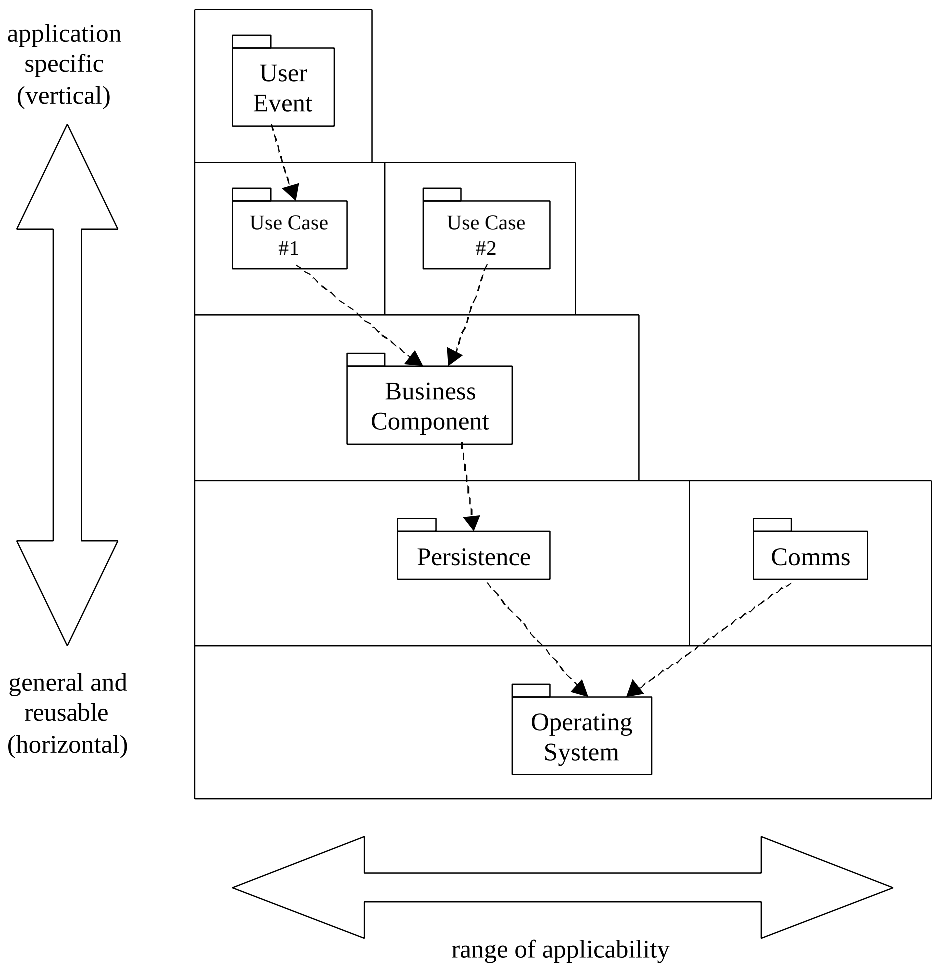


Figure 4 - Vertical v. horizontal components

The ordering is generally shown on a package diagram by drawing the higher order (or higher layer) packages - those which contain classes that are inherently application or *vertical* domain specific - towards the top of the package diagram, and the lower order (or lower layer) packages - those which contain classes that have fewer or no domain dependencies - towards the bottom of the diagram.

3.1.1. Motivation for architectural layering

What is the point of architectural layering? Key motivations are:

- *To minimise the mixing of levels of details.* Descending into detail (e.g. individual data formats and structures) when the overall structure of the system is unnecessary.
- *To assist in managing the dependencies between the intended components that will make up a system.* Cross-layer dependencies should be in a downward direction, from one layer to the layer immediately below it. This ensures that the propagation of change within the system is minimised, as dependencies will point from the more unstable specific layer (where requirements changes are more likely to occur) to the more stable general one.
- *To assist in separation of concerns.* Layers such as the highest, which contain GUI dependent classes, and the lowest, which contains database dependent classes, are inherently technology specific. The business object layer in particular should not contain technology dependencies. Such a separation of concerns will assist in both focusing the mind of the designer (divide and conquer), and in managing technology changes (e.g. from Windows to Motif) should these be required. This implies that the interfaces between these layers should themselves be technology independent.

The 'layering' metaphor comes from the practise of dividing package diagrams into horizontal bands, going from the top to the bottom of the diagram, each of which isolates various concerns of the software architecture, such as the user interface, data storage, communication, interfacing to hardware, persistence, etc. Layers are often referred to as being vertically specific (as in a vertical market) or horizontally specific (as in a horizontal market). This depends on whether they are tied to a particular problem domain or not, although it should be noted that there may be a sub-layering (most easily derived from package dependencies) within both the vertical and horizontal areas of a software architecture. Good component-based design relies heavily on these concepts.

An overview of the architectural layering of the settlement system is shown in figure 5. It illustrates four layers and conforms to the conventions discussed above:

- The upper layer contains objects which are highly specific to the application in question, and which are generally not reusable (unless you are building another version of the same application!). This layer is generally made up of classes that either manage user or external system interfaces (boundary classes[1]), or use case or control classes (controller classes). These control the functions or use cases provided by the application by manipulating the business objects in the layer below. In the settlement application this is referred to as the *activity layer*.

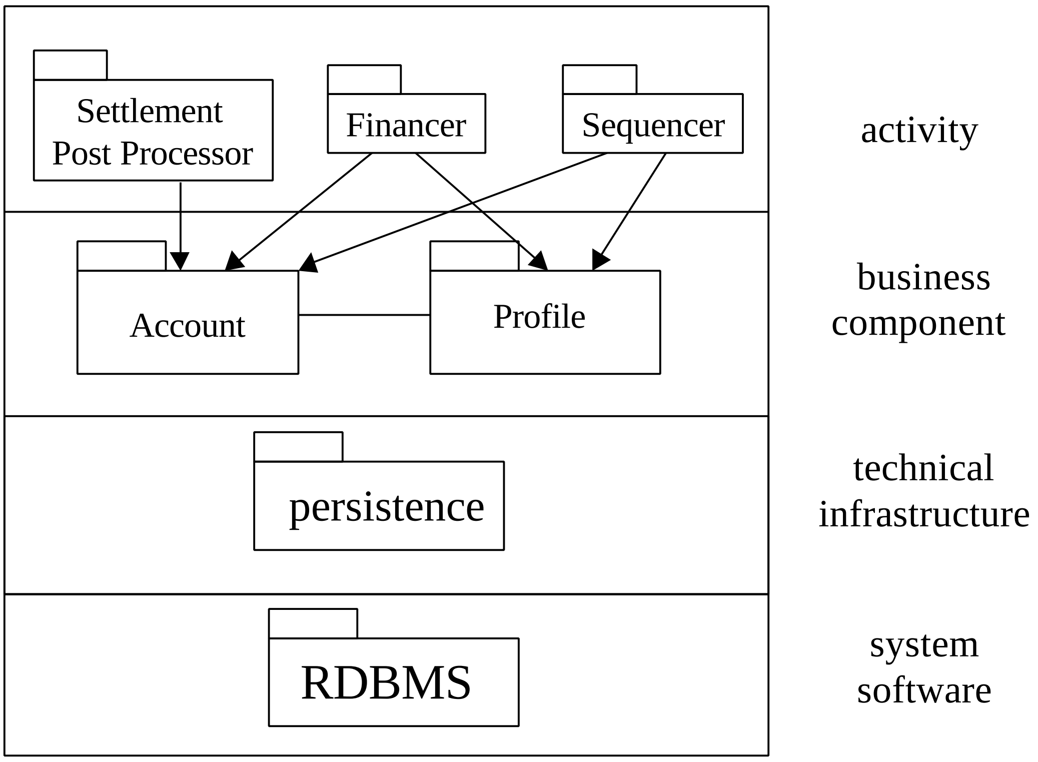


Figure 5 - Layering of settlement system

- The layer below is the *business component layer*, in which components provide façades to groups of co-operating business objects (entity classes) which they contain. Business components encapsulate objects that contain domain specific data, and operations on this data that follow the appropriate business logic or rules. The façade provided by a component may have one or more interfaces. Since different applications in the same

[1] Jacobsen introduced the concept of boundary, controller and entity classes in his book "Object Oriented Software Engineering" (see [Jacobsen]). Boundary classes deal with the interface of the system to other systems or to humans (such as a graphical user interface). Controller classes contain algorithmic code that controls the interaction of the parts of the system, and entity classes are the business objects that represent entities in the problem domain.[1]

problem area may deal with the same sorts of problem domain objects this layer is more reusable and less specific than the upper layer.

- The *persistence layer* is the lowest layer that was written by the developers of the system under discussion. It deals with issues connected with the loading and retrieval of objects to and from a relational database. Further details of the persistence mechanism used in this project are contained in the paper “Persistence: Implementing Objects over a Relational Database”.
- The final layer is a commercial RDBMS. This is generally applicable across a wide range of applications.

3.1.2. Architectural layering works!

Architectural layering provides a solid basis upon which a component based design may be undertaken. In the settlement system development under discussion, the layering with downward acyclic dependencies ensured that the system has remained stable in the face of considerable changes in requirements.

3.1.3. Layering and Deployment

Layering can provide flexibility for how the components of the system are deployed to physical platforms, for instance in a traditional 3 layer client-server system, the use case components might be deployed to the client machine, the business components to the middle tier and the RDBMS on its own dedicated tier. However, in this case the entire application was deployed to a single, enterprise class server.

3.2. Granularity and interfaces

Having seen how components and layering can help us we must next address the issue of designing such a system. A typical approach is to use object-oriented analysis and design (OOA/D). A good starting point is the use cases for the required system. They describe the scope and required functionality and may be cross-referenced against a high-level specification object model of how components and their façade objects relate. Another paper ("The RIS Approach to Use Cases " – see [RIS]) discusses the factoring out of the top level of component 'services' from 'business process' use cases in more detail. We shall discuss more details of the design process below, but for the moment we will assume that we have a list of services to be implemented.

Given a list of services to implement we are faced with two questions:

- how do we group these services into cohesive interfaces, and
- how do we decide which components implement which interfaces?

Answering these questions is probably the key architectural step in the design of a component system, since this is the foundation of all the subsequent design and implementation effort. It is vitally important to concentrate the right resources on getting component interfaces right early on, and not be tempted to rush into the implementation of individual components. Experience from this project was that investment at this point is rewarded handsomely in the subsequent design and development phases.

3.2.1. Changes to interfaces

In a large system, we wish to avoid changing the components' interfaces as this may mean having to modify a large amount of code. The component-based approach helps prevent changes from propagating across the system as internal modifications are invisible because of the high degree of encapsulation. However, if a component's interface changes then potentially all clients of that component (or that particular interface if it offers multiple interfaces) are subject to change. Therefore we must try to avoid changing interfaces if possible, and if it is not possible then we must consider the knock-on effects on a system-wide basis.

The approach taken in the settlement system was that the system architects had to approve all changes to a component's interface. This enabled them to balance the need for the change (to add extra functionality, for example) against the need to maintain stable interfaces. Taking an example for the system in question a change to either the Account or Instrument components' interfaces would cause 10 other components to have to be changed too (or at least rebuilt), so such a change would have to be absolutely necessary to be approved. The change request procedure ensures that the architects are fully aware of such changes.

3.3. Reuse issues

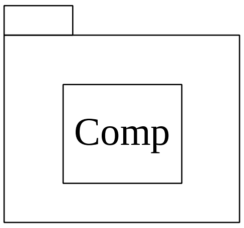
One of the commonly vaunted advantages of object-oriented technology is the possibility of reusing code. Because of inter-class coupling it is often not possible to reuse individual classes. Components, however, encapsulate a number of classes that collaborate to achieve their intended aspect of functionality, and are therefore more much more likely to be reusable.

3.3.1. Extending components externally

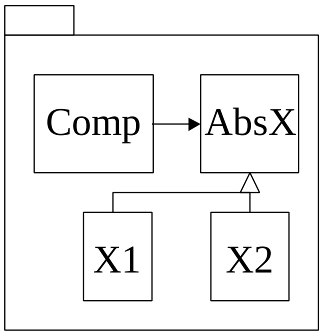
A typical scenario for the life of a vertical component might be as follows. First of all, the component is written and used in one particular context, usually the project as part of which it is developed. The next stage is when the component is first reused. A certain amount of work has to be expended in order to make the component fit into this new context, but it will become more general purpose re-usable. A final stage is when the component is adapted to allow it to be extended without access to its source code (when it conforms to the open closed principle), typically by getting the component to call out to extension classes that are subclassed from a base provided by the package. This will generally occur at a later stage than the first re-use, as only at this time have the designers had sufficient exposure to different contexts to make it truly general. After this final stage is reached then the component can be used and extended to work in almost any environment where its core functionality is useful.

In the project under discussion, the vertical components are at the second stage, the number of functional changes so far requested constituting an effective first re-use. They can currently be configured using database tables to vary the functionality within certain limits. Considerable effort has gone into defining internal extension points. At some point, when the demand for re-use of a particular component is apparent, the internal extension points will be externalised.

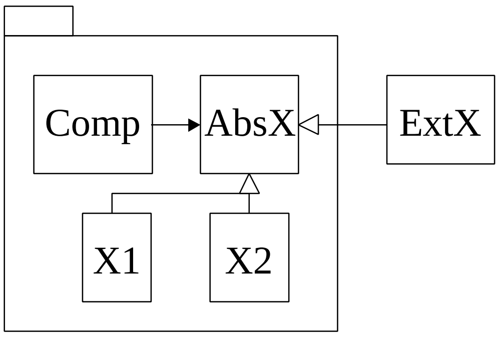
To illustrate this lifecycle, diagrams showing the three stages of a component's life are shown below:



Stage 1: monolithic internals



Stage 2: internally extensible



Stage 3: externally extensible

Figure 6 - Stages in component reuse

In stage 1 the internals of the component are often monolithic with little effort having been put into making the component flexible or extensible in any way. When requirement changes force the component to be updated what typically happens is that the internals of the component are modified to implement the new functionality by factoring out common behaviour into superclasses and by the use of internal polymorphism. One particularly common method is the use of strategy objects that act as plug-in algorithms for variations in behaviour. This is shown as stage 2 in figure 10, where AbsX is an abstract base class that defines the interface for strategy objects X1 and X2.

Once the component has reached this stage, it is simple to allow for external strategy objects such as ExtX shown as stage 3 in the diagram. We have now reached a stage where we can vary the behaviour of the component without having to modify its source code in any way. Hence we can distribute the component in binary form, which has advantages in terms of reduced build times, security of intellectual property (users don't have access to the source code), simplicity of distribution, etc. Stage 3 was not a require of the Settlement project and as such no such implementation has occurred.

We have also gained in that the component is now more general, because we have removed the application specific part of the behaviour, and more flexible by allowing users to provide their own versions of the extracted behaviour. In architectural terms, we may be able to move the component down a layer whilst keeping the application specific behaviour at the correct layer, as shown in figure 11.

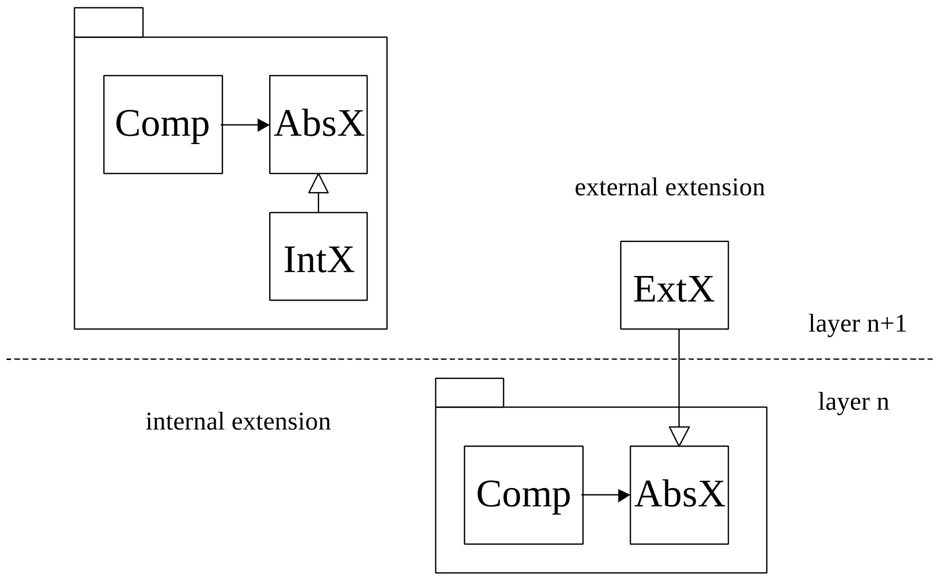


Figure 7 - Internal v. external strategy objects

Here the architectural layer boundary is shown with a dotted line. On the left we see that because the extension behaviour (which is application specific) is within the component then the component as a whole becomes application specific and therefore less reusable. On the right, however, the application specific part has been separated from the common core functionality of the component, so the common part (which is now application neutral and so more reusable) can be moved down a layer and the application specific part remains at the same layer.

As a concrete example of this process applied to the settlement application, consider risk limit checking for an account. The Account component currently contains within itself an abstract interface from which all risk limit checkers are derived. This is an example of the use of the

Strategy pattern (see [GoF] for further details). Which checker should be used for a particular account is then chosen by the use of configuration data. It would be very easy to export this abstract risk checker interface and have an external class derive from it. All that would then be needed would be a way of connecting this external checker object to the Account component and we would have a fully extensible component, like this:

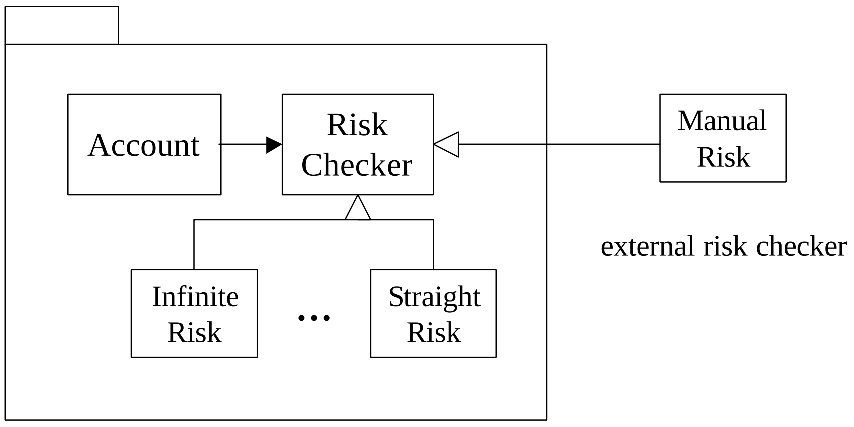


Figure 8 - Settlement example of external strategy object

A key lesson here is that on larger systems such as this one it is vitally important to concentrate on getting component interfaces right early on!

3.3.2. Reuse of business components

The settlement system has undergone extensive change during its development because of requirement changes. Although the business components have not been reused in other applications one could argue that these components have been reused within the application itself. This is a testament to the effectiveness of component-based development.

3.3.3. Reuse of the persistence layer

In contrast to the vertical components, the strict architectural layering of the system has ensured that the persistence layer contains no application specific code. This means that it is a generally applicable, horizontal component that could be reused in other non-settlement applications. Indeed there are currently plans to reuse it in a number of other fields.

3.3.4. Reuse of other business classes

A number of other classes were reused in the settlement application. These were mostly general non-business specific classes such as date and time classes, strings and collections. One very useful business class that was reused though is the TiCash class. This class handles all of the multi-currency aspects of cash, handling arithmetic, conversion, rounding and all of the legal procedures for Euro conversion, etc. More details of this class can be found in the "Patterns in Financial Systems" paper.

4. Process and team organisation

4.1. Design process

In this section we briefly describe a design process for component-based development. It is intended to be illustrative of the general approach rather than as a comprehensive and prescriptive sequence of steps to be followed.

The process involves the following steps:

1) extract essential functionality from a written description of the requirements as service use cases
2) decide to implement these services as either use case objects or as methods on component interfaces
3) decide how to allocate these methods and interfaces to components
4) use inter-component level sequence diagrams to verify the decomposition and to discover lower level methods
5) drill down using intra-component level sequence diagrams to decide how to implement low-level methods and component internals whilst keeping the higher level design under rigorous change control.

Step 1
The usual place to start a design is from the requirements specified by the customer of the system. These are usually specified in English or some other natural language. From this written description we can extract the required functionality of the system in the form of a list of functions to be implemented, sometimes called "service" use cases. One particular process for discovering these services is the RIS method described more fully in "The RIS Approach to Use Cases" (see [RIS]).

Step 2
Given this set of services we can now decide how to implement them. Possible approaches are to:

- implement the steps of a use case within the use case object itself;
- have a use case simply forward to a method on a component interface;
- call the component interface directly.

The first of these choices (a "fat" use case object) is appropriate if we require "do" and "undo" facilities (as in Command pattern [GoF]), or if the use case's functionality will not be reused elsewhere. This is often true as use cases represent the most application specific functionality within the system. Also, methods such as RIS factor out common functions as services to be implemented in their own right. A forwarding use case object would be used if the code must run in a multi-user environment where all user code must execute as part of a transaction, whereas we would call the component interface directly only if lower level components offer the correct functionality. The sequence diagram in figure 13 gives an example of a "fat" use case object from the settlement domain called CollateralTopUpandReturner that implements the "Collateral Top-up and Return" use case.

Now that we have made our first level of implementation decisions we have started to populate the top layer of our reference architecture (defined in section 3.1) with either use case objects or top-level components.

Step 3
The next step is to group our services into interfaces and to assign these interfaces to components. Considerations of granularity, coupling, cohesion, and component level polymorphism as discussed in section 3.2 now drive our design. At this point we will also consider issues such as the need to modify an existing component to allow it to be extended (as in section 3.3.1), whether we should buy a commercial component and adapt it, or build our own. Typically these business components will be in the second highest layer of the reference architecture.

Step 4
In step 4 we use an inter-component sequence diagram to verify that the interfaces and components identified in step 3 can support the desired functionality. The diagram also allows us to discover lower level methods that we must design. Figure 13 shows an example of such a high-level sequence diagram taken from the settlement domain.

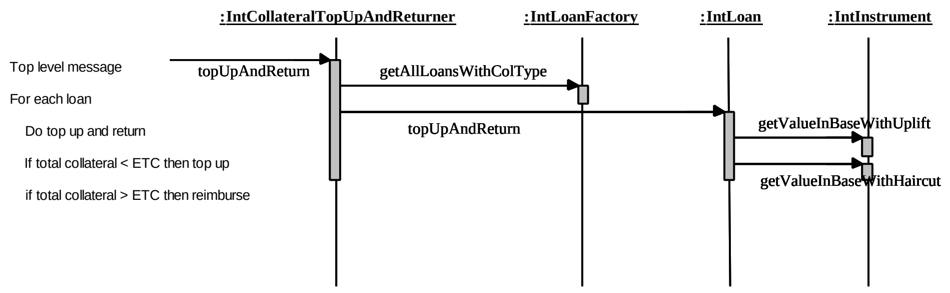


Figure 9 – Inter-component sequence diagram

Step 5
Having verified that our top-level decomposition is not missing any essential details we can then drill down by drawing sequence diagrams for each method in the inter-component diagrams. This leads us towards designing the internals of each component method. To illustrate this, figure 14 shows the sequence diagram for the getValueInBaseWithHaircut method and figure 15 shows a simplified specification object model for the Instrument component of the settlement system.

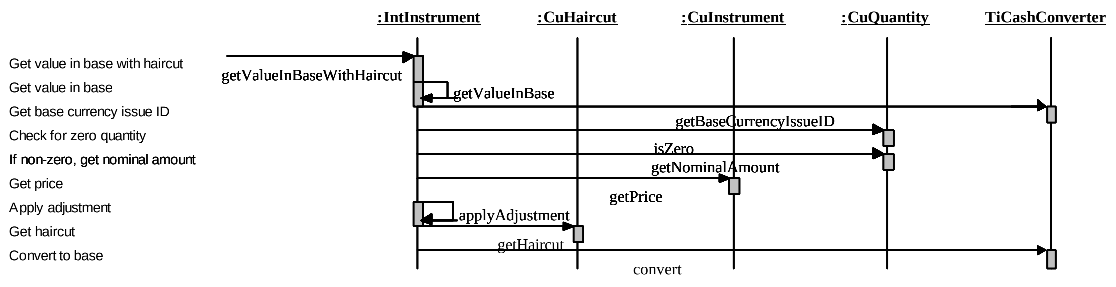


Figure 10 – Implementing IntInstrument::getValueInBaseWithHaircut

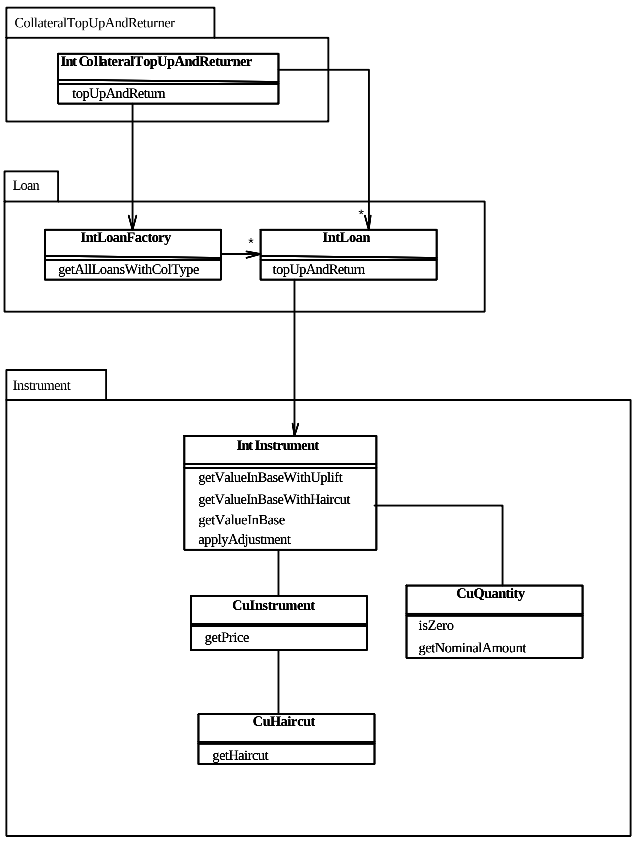


Figure 11 - Specification object model for Instrument

4.2. Practical aspects of development process

The above description is just a general overview of one particular approach to designing using components. In practice the process is more complex. For example, for the settlement application the functional requirements were predominantly extracted using workflow designs. Analyses of the variability and configurability of the design were also undertaken.

4.3. Team organisation

Ensuring that components are as independent as possible has significant advantages for team working. Designing strongly encapsulated components that interact only at the architectural level and only then in visible and planned ways has significant benefits when delivering a project to tight timescales. Specifically it allows a number of relatively independent design and development cells to be mobilised, which can work concurrently and also provides a very obvious strategy for initial “assembly” level testing. This is a positive example of the software management maxim: *the structure of the software reflects the structure of the team that built it and vice versa.*

Another advantage of the component approach is that in languages such as C++ and Java the compiler enforces the encapsulation that allows for independent teams. All access to the component is via its interfaces, with no back doors being allowed. This adds significantly to the intelligibility of the system as the components will be independent as far as is possible. Any coupling they do have is clearly documented and is visible at the architectural level. This contrasts with the traditional structured programming approach where encapsulation is a matter of good practice and programmer discipline rather than being strictly policed by the compiler.

5. Component implementation issues

Up to now we haven't mentioned anything specific about how components are implemented. We will now give details of the internal anatomy of a component and show how the logical package structure that we have so far seen is mapped to source files (the physical view) and to binary objects (the release view).

5.1. Detailed anatomy of a component

We must now confess to having simplified the internal structure of the instrument component shown above. Certain implementation details were glossed over in the implementation model to avoid muddying the picture. However it is now appropriate to examine these implementation specific details.

5.1.1. Logical View

One of the weaknesses of C++ is that private implementation details are visible within the header files. To avoid this leakage of implementation information, and therefore to avoid unnecessary rebuilds because of changes to that private information, the settlement project used a technique variously known as pimpl, Cheshire Cat, or handle/body (see [Meyers] for further details). This involves inserting an intermediate object that acts as a façade object for the component that merely passes all requests on to an internal object thus hiding all implementation detail and exposing only a single simple object to users.

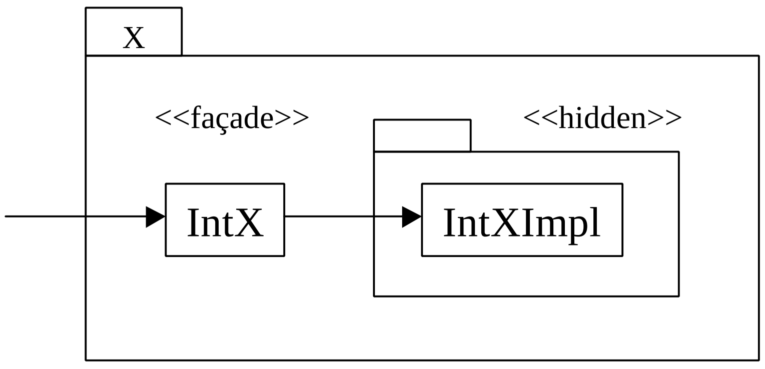


Figure 12 - Implementation of facade object

Having enriched our packages by adding in this façade object, we can now widen our view to encompass three views of a component: the logical, physical and release views. The logical view is the only one we have seen up to now, and gives the package structure. This is the view we design with.

5.1.2. Physical View

The next view is the physical view: how we map our logical package structure onto physical source files. For component X shown above we would typically have a source structure something like this:

Header files	Implementation files
X.h (exported)	
IntX.h	IntX.cc
IntXImpl.h	IntXImpl.cc

plus any additional implementation/header file pairs for classes within the hidden implementation package. X.h is the only file accessible to users of component X and defines the interface to X. The directory structure to hold these various files can vary from project to project and, whilst vitally important for successful configuration management, is not discussed in any further detail in this paper.

5.1.3. Release View

The release view is what we give to users of the component. All they need is the binary code for the component, as object files or as a shared library, and the header file giving the component's interface:

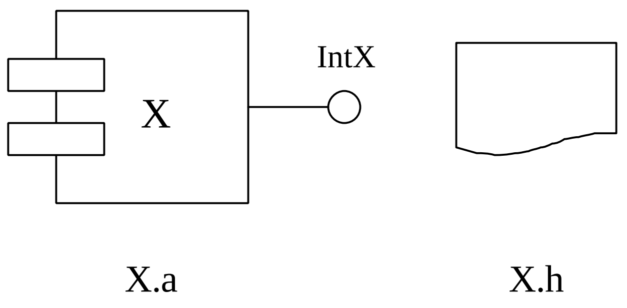


Figure 13 - Release view of component X

Here we are distributing component X as a Unix object library (.a stands for archive), plus the header file needed to use it.

For the settlement system the full inventory of files for the physical view of the instrument component is too long to include, but it follows roughly the scheme laid out above, with the header files (*.h) and the implementation files (*.cc) being held in different directories. The release view is also similar except that the components are distributed as Unix shared libraries in .so format instead.

6. Summary

This paper has described the use of component-based development methods on a large financial settlement system. The functionality of certain key components within the domain has been described and it was shown how these components, along with their supporting technical infrastructure, form a layered structure. This lead to a discussion of general architectural principles such as layering and granularity, and issues concerned with reuse and extension of components. We also have given a brief overview of designing component-based systems and how this affects team working.

Finally, implementation issues such as façade objects and the three views of a component (logical, physical and release views) were discussed.

The general feeling of the development team was that the use of a component-based approach was beneficial to the project. The areas where it particularly helped are in allocating the work into teams, as each component was self-contained and relied only on the interfaces of other components, and in the control of the propagation of change – most changes were contained within a single component.

The control of interface change through the use of change requests definitely helped in maintaining the architecture of the system. Team members commented that with previous comparable systems developed using a structured procedural approach the system architecture tended to degrade during development, primarily because of the ease of taking shortcuts. With a component-based system, encapsulation is policed by the compiler rather than relying on programmer discipline so it is very difficult to use a back door into a component – going via the public interface is really the only viable option. Therefore to achieve these benefits the project must have a strong emphasis on architecture.

One point worthy of note is that the system achieved its performance targets without breaking the problem domain model. Profiling of the system showed that the object aspects of the design (the use of virtual functions, two and three layer interfaces, design patterns, etc) took up a noticeable amount of CPU time compared to a procedural system, but not sufficient to compromised the system's performance.

The project team also felt that one of the most important achievements was to get the component interfaces correct at an early stage and then to control rigorously changes to them. This prevents the system architecture degrading over time and reduces the need to rework existing code because of interface changes.

7. References

[Meyers]	Scott Meyers, Effective C++, 1992
[GoF]	Erich Gamma, Helm, Johnson, Vlissides, Design Patterns, 1995
[RIS]	Mark Collins-Cope, "The RIS Approach to Use Cases" (available from Ratio)
[Jacobsen]	Jacobsen, Object-Oriented Software Engineering

"Object and Component Overview," (available from Ratio)
"Patterns in Financial Systems," (available from Ratio)
"Persistence: Implementing Objects over a Relational Database" (available from Ratio)