Making Use Cases precise: a model driven approach

Abstract

This paper presents a method of object-oriented analysis (OOA) that combines the informality and user-friendliness of use cases, with the formality and precision of the Unified Modelling Language (UML), augmented with constraints and definitions written in the Object Constraint Language (OCL). The method is illustrated with an example. The benefits of producing precise functional specifications before coding starts are discussed.

Introduction

Use cases are probably the best known and most popular technique for capturing functional requirements. However use cases suffer from the drawbacks that all narrative specification techniques suffer from, that is: possible misinterpretation due to their inherent imprecision, difficulty to uncover inconsistencies, and difficulty to check for missing specifications, such as a missing flow or a missing outcome.

The analysis approach presented in this paper increases the quality of use case specifications, making it far easier to uncover problems while the specifications are being built. Our approach uses the full descriptive power of UML [Fowler04] and, optionally, OCL [Warmer03], to capture functional requirements.

The fundamental principles of our approach can be summarised as follows:

- The system under construction is modelled as an object. Like all objects, it has behaviour and state, described in its class. The behaviour is captured as operations. The state is captured with a class model.

- The operations that implement a use case are identified by formalising each meaningful flow in the use case as a system dialogue, which is represented as a sequence diagram showing the interactions between the actor(s) and the system object.

- Each operation is further specified in a declarative style, stating its pre- and postconditions, based on a UML model of the system state and on the operation input and output arguments. They can, optionally, be written in OCL for added precision.

Target audience

This paper is aimed primarily at use case analysts. It should also be useful to test analysts, designers and developers who need to understand use case specifications and associated models.

A working knowledge of object-oriented concepts and UML is a pre-requisite. No previous knowledge of OCL is assumed. OCL is an optional ingredient in this method and is confined to a single sub-section of this paper.

Example system

A simple, fictitious example system is used to illustrate the method. An informal description of the system is provided in the box below.

A **Mortgage Manager** system is required by a financial institution to keep track of residential mortgage accounts, i.e. loans secured on a residential property. At a very high level, the system requirements are:

- to capture information and provide on-demand statements on mortgage accounts and mortgage products
- to manage the details and life of each account

The users (actors in UML parlance) of the system are:

- account administrators
- account holders (through a Web interface)

Sources and References

Our modelling approach draws heavily on the principles of the Catalysis method [D'Souza99]. The rigorous approach of modelling behaviour using operation specifications based on an underlying type model is based on both Catalysis and Syntropy [Cook94]. The first OO method that modelled the system as an object was Fusion [Coleman94].

2 Method Description

One of our key modelling principles, initially proposed in the Fusion [Coleman94] method and elaborated further in Catalysis [D'Souza99], is that the system is itself modelled as an object, instance of a <<system>> type, and that system functionality is modelled as the behaviour and state of that type.

Figure 1 shows the MortgageManager system type. To start with, the type has no attributes and no operations. At the end of our use case analysis, the type will have been populated with all the system operations required to support the use cases in scope, and all the types required to write precise specifications of the behaviour of those operations.

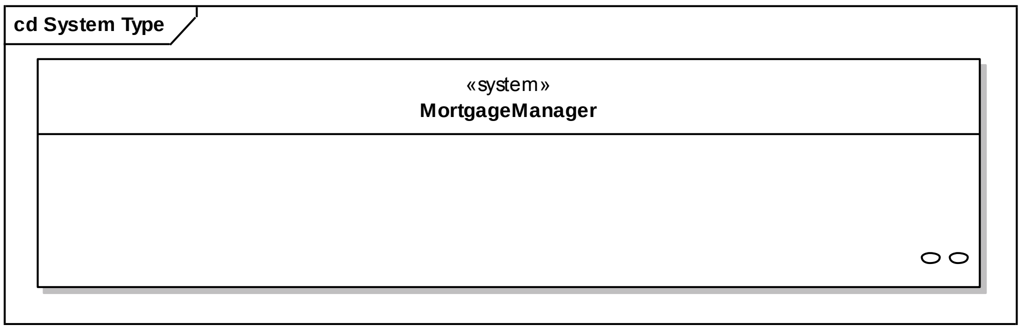


Figure 1: The System Type

Our analysis method starts with an informal description of the system functionality based on use cases, and ends up with formal and precise system operation specifications, written in English and (optionally) in OCL. These are the main steps of our method:

1. **Describe Use Case Flows**

   Identify actors and use cases and produce informal descriptions of use case flows.

2. **Identify system operations**

   Capture the system-actor(s) dialogue in each use case flow using a sequence diagram and identify system operations required to support that flow.

3. **Specify system operations**

   Represent each system directed interaction identified in step 2 as an operation on the system type. Formalise its input and output arguments in UML, and provide a specification of its precondition and its outcome (postcondition). In parallel, build an abstract model of the system state. We call this the System Type Model (STM), since the classes that appear within it are not software classes with methods and instance variables. Instead, their purpose is to provide a business-oriented vocabulary, free from design and implementation details, to specify the behaviour of the system operations.

   Ensure that the operation outcome is specified using only terms from the system type model. Anything that cannot be described in these terms should be considered wishful thinking, and not an implementable specification of behaviour.

4. **Add precision with OCL**

   If more precision is required, write a specification of each system operation in OCL. Invariants and definitions in the System Type Model can also be described in OCL.

Step 1 - Describe use case flows

Let us assume we have a simple use case, where the actor is the Account Administrator, and his/her goal is obtain a breakdown of money on loan by mortgage product.

Let us also assume that an initial sketch of the main use case flow has already been agreed with the business users, with the aid of a user interface prototype, and is as follows:

1. The actor asks the system for the mortgage balance breakdown by MortgageProduct
2. The system calculates the balance on loan for all the MortgageAccounts, broken down by MortgageProduct, and displays the result to the actor (Screen 1)
3. The actor selects one of the MortgageProducts from the list and asks the system to show the balance composition for that MortgageProduct
4. The system calculates and shows the composition of the selected balance (Screen 2)

The two screens' logical contents may be as follows, with the product/balance table ordered by descending balance value:

Screen 1 - Balance Breakdown by Product

Product	**Balance**
□ 2 Year Fixed	£25.000,000
□ Flexi Tracker	£15.000,000
□ 3 Year Fixed	£13.000,000
Total	£53.000,000

Show Balance Composition

Screen 2 - Product Balance Composition for Product: Flexi Tracker

Approved	**Completed**	**Secured**	**Unsecured**	**Accounts**	**Acc**
Elements					
£10.000,000	£15.000,000	£23.200,000	£1.800,000	254	367

Step 2 - Identify system operations

Each use case flow is modelled as a system dialogue, i.e. a sequence of interactions between the actor and the system, and illustrated in a sequence diagram (Fig. 2).

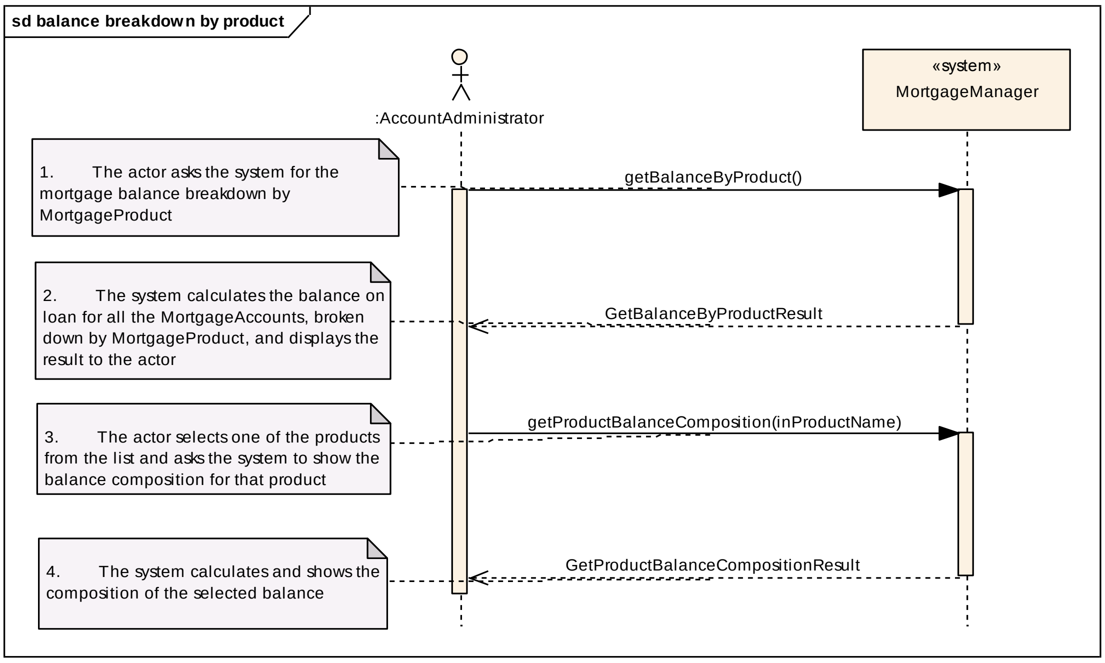

sd balance breakdown by product
:AccountAdministrator
«system»
MortgageManager
1. The actor asks the system for the mortgage balance breakdown by MortgageProduct
getBalanceByProduct()
2. The system calculates the balance on loan for all the MortgageAccounts, broken down by MortgageProduct, and displays the result to the actor
GetBalanceByProductResult
3. The actor selects one of the products from the list and asks the system to show the balance composition for that product
getProductBalanceComposition(inProductName)
4. The system calculates and shows the composition of the selected balance
GetProductBalanceCompositionResult

Figure 2: System Dialogue

The requests from the actor to the system are formalised as operations getBalanceByProduct() and getProductBalanceComposition() on the system object, as shown in the diagram.

Note how the diagram does not introduce new classes to support system functionality. The MortgageManager system supports all the functionality that the use case requires. However, as we will see, the specification of that functionality can be factored out into the types in the System Type Model, by attaching definitions to individual types. For example, the rule that calculates the balance for a MortgageProduct, productBalance(), is attached to the MortgageProduct type.

Step 3 - Specify system operations

Each interaction initiated by the actor is modelled as an operation on the system type. The input arguments of the operation are the data entered by the actor, if any. The result is the data, together with any errors, returned by the system.

The next step is to write a specification for the operations, containing definitions of their input and output data, and the rules for producing the output data. The most useful technique to write implementation-independent operation specifications is to use pre- and postconditions. The box below provides some details of this technique.

The postcondition of an operation is a boolean expression that is guaranteed to be true when the operation completes, provided that the precondition, itself a boolean expression, is true when the operation is invoked.

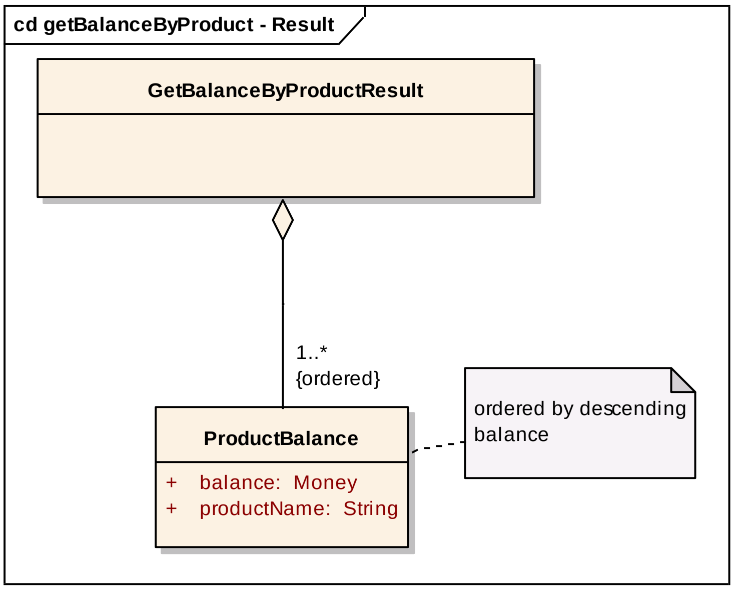
A precondition must only refer to properties of the input arguments and of the system state at the start of the operation. A postcondition can use properties of the input and output arguments and of the system state, before and after the execution of the operation.

The postcondition of an operation expresses how the output arguments are derived from the input arguments and from the system state, and how the system state after the operation is derived from the state before the operation and the input arguments.

A postcondition cannot say anything about the state of the system while the operation is being executed. It can only mention system state before and after the execution. Hence writing a postcondition forces the analyst to consider the effect of an operation without considering its internal algorithm, a useful discipline to avoid mixing design considerations with business results.

Both operations in hand are of the *query* type, which means they do not change the state of the system. All we need to say is how the result of each operation is derived from its input data and from the system state. But so far the system has no state in our model! (see Fig 1). Let us write specifications for these operations to see how the state can be derived from the behavioural requirements.

System Operation: getBalanceByProduct()
Input arguments: None
Result: GetBalanceByProductResult (see Figure 3)
Precondition: None
Postcondition: The result of the operation is a collection of ProductBalances, one for each MortgageProduct known to the system. Each consists of the name of the MortgageProduct and the total amount on loan for all MortgageAccounts associated with that MortgageProduct. The balances must be sorted by descending value, i.e. higher balances first.


cd getBalanceByProduct - Result
GetBalanceByProductResult
1..*
{ordered}
ProductBalance
+ balance: Money
+ productName: String
ordered by descending balance
Figure 3: getBalanceByProduct() result type

As shown in figure 3, the result type of the operation is represented as a UML class. From the postcondition, we can start forming a picture of the information the system needs to hold to answer the query:-

- all MortgageProducts for which MortgageAccounts can exist, with their product name
- all MortgageAccounts linked to each MortgageProduct
- all MortgageAccountElements linked to each MortgageAccount, with their balance.

The fact that the balance is held within each MortgageAccountElement, and not in each MortgageAccount, is assumed to be part of domain knowledge[1].

Figure 4 shows the resulting System Type Model, with explanatory notes to show where the model elements come from:

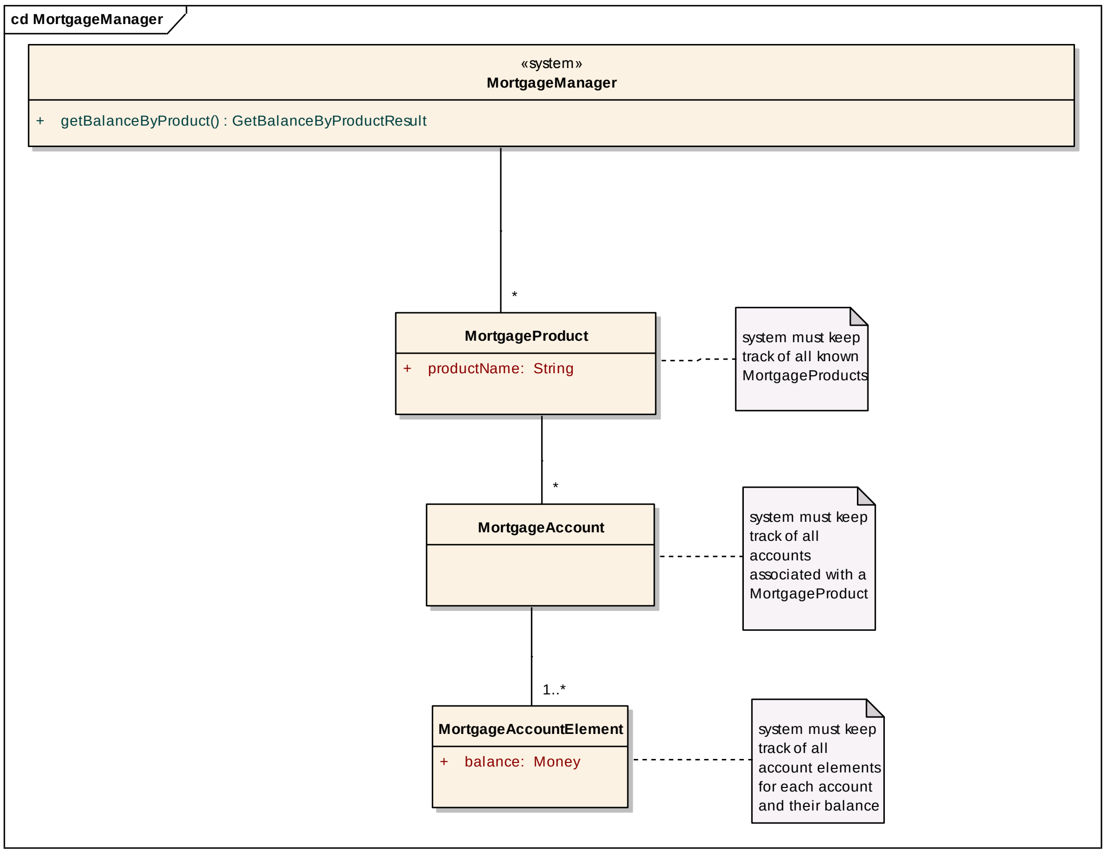

cd MortgageManager
«system»
MortgageManager
+ getBalanceByProduct() : GetBalanceByProductResult
*
MortgageProduct
+ productName: String
system must keep track of all known MortgageProducts
*
MortgageAccount
system must keep track of all accounts associated with a MortgageProduct
1..*
MortgageAccountElement
+ balance: Money
system must keep track of all account elements for each account and their balance

Figure 4: System Type Model with first operation and supporting state

An informal review of the operation postcondition against the model should be sufficient to convince us we have all the required attributes and associations. The system can navigate to its attached MortgageProduct instances. For each, it can navigate to its MortgageAccounts, and add up the balances of all the MortgageAccountElements linked to these MortgageAccounts. The result can easily be constructed from these objects and their attributes.

[1] Even though we are pretending to start with no model of the system, in most situations a Domain Model will exist, and the types, attributes and associations required to write our postconditions will be in that model. However for now we will stick with the assumption that we are building the System Type model from scratch.

The second operation takes a product name as input argument and returns an instance of GetProductBalanceCompositionResult (Figure 5)

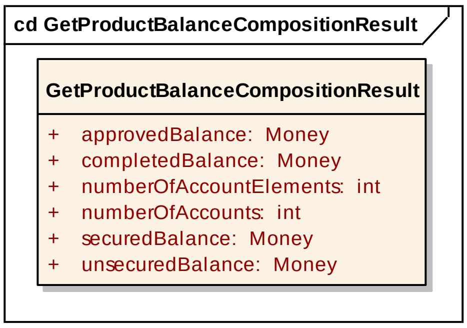

cd GetProductBalanceCompositionResult
GetProductBalanceCompositionResult
+ approvedBalance: Money
+ completedBalance: Money
+ numberOfAccountElements: int
+ numberOfAccounts: int
+ securedBalance: Money
+ unsecuredBalance: Money

Figure 5: result type of getProductBalanceComposition()

System Operation: getProductBalanceComposition()
Input arguments: inProductName : String
Result: GetProductBalanceCompositionResult (see Figure 5)
Precondition: A MortgageProduct by the given product name (inProductName) must be known to the system
Postcondition: The result of the operation contains:-

- approvedBalance - sum of all balances of MortgageAccountElements for the given MortgageProduct, considering only MortgageAccountElements with status=Approved
- completedBalance - sum of all balances of MortgageAccountElements for the given MortgageProduct, considering only MortgageAccountElements with status=Completed
- securedBalance - sum of all balances of MortgageAccountElements for the given MortgageProduct, considering only MortgageAccountElements with status=Approved or Completed, and isSecured=true
- unsecuredBalance - sum of all balances of MortgageAccountElements for the given MortgageProduct, considering only MortgageAccountElements with status=Approved or Completed, and isSecured=false
- numberOfAccounts - counts how many MortgageAccounts are linked to the given MortgageProduct, considering only MortgageAccounts with at least one MortgageAccountElement with status=Completed or Approved
- numberOfAccountElements- counts how many MortgageAccountElements exist for MortgageAccounts linked to the given MortgageProduct, considering only MortgageAccountElements with status=Completed or Approved

Note how writing the postcondition in terms of types and attributes in the system type model has forced into the open the issue of the status of MortgageAccountElements, easily missed if the analysis had stopped at the narrative level in the use case flows. Further investigation with the business users reveals that a MortgageAccountElement can be in one of three states, as depicted in the state diagram in Figure 6[2]. The balance of elements in the Redeemed state should not be included in the total product balance, hence the postcondition only includes those in the Completed and Approved state.

[2] State models such as this one should be produced for all types with an interesting lifecycle - the presence of a status attribute is a good indicator of a need for this.

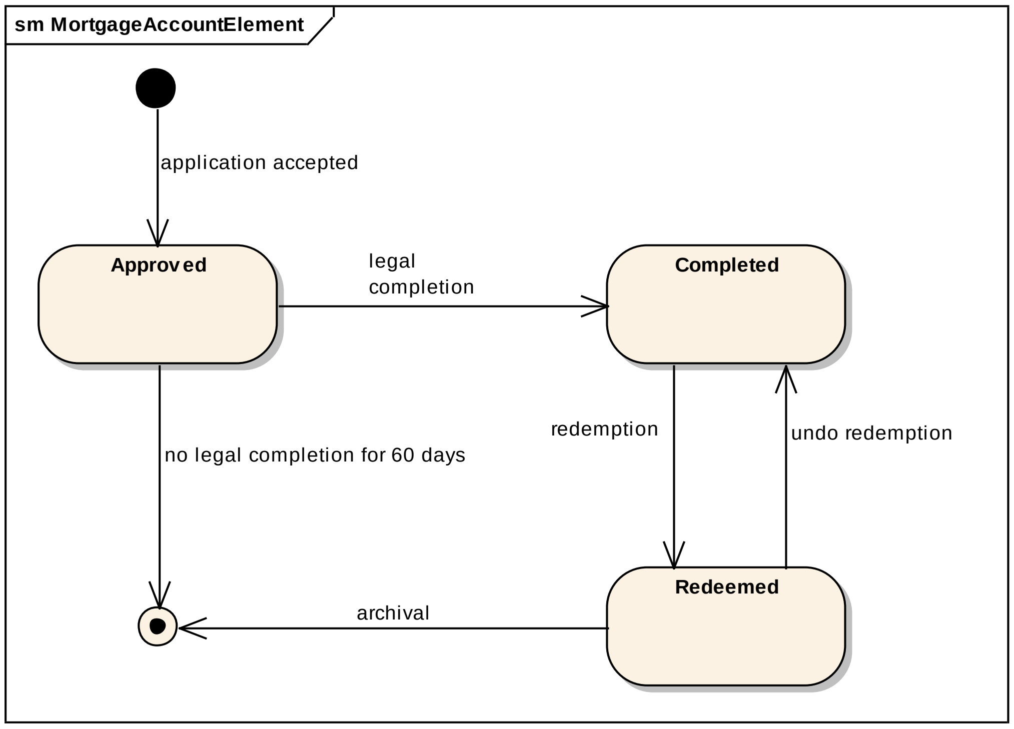


Figure 6: state model of a MortgageAccountElement

Note that the first operation, getBalanceByProduct(), should have also ignored such account elements. The operation specification can be rewritten to reflect this (new clause in **bold**):

System Operation: getBalanceByProduct()
Input arguments: None
Result: GetBalanceByProductResult (see Figure 3)
Precondition: None
Postcondition: The result of the operation is a collection of ProductBalances, one for each MortgageProduct known to the system. Each consists of the name of the MortgageProduct and the total amount on loan for all MortgageAccounts associated with that MortgageProduct, **excluding MortgageAccountElements in the Redeemed state.** The balances must be sorted by descending value, i.e. higher balances first.

In order to support the second operation, MortgageAccountElement needs two new attributes - see Figure 7.

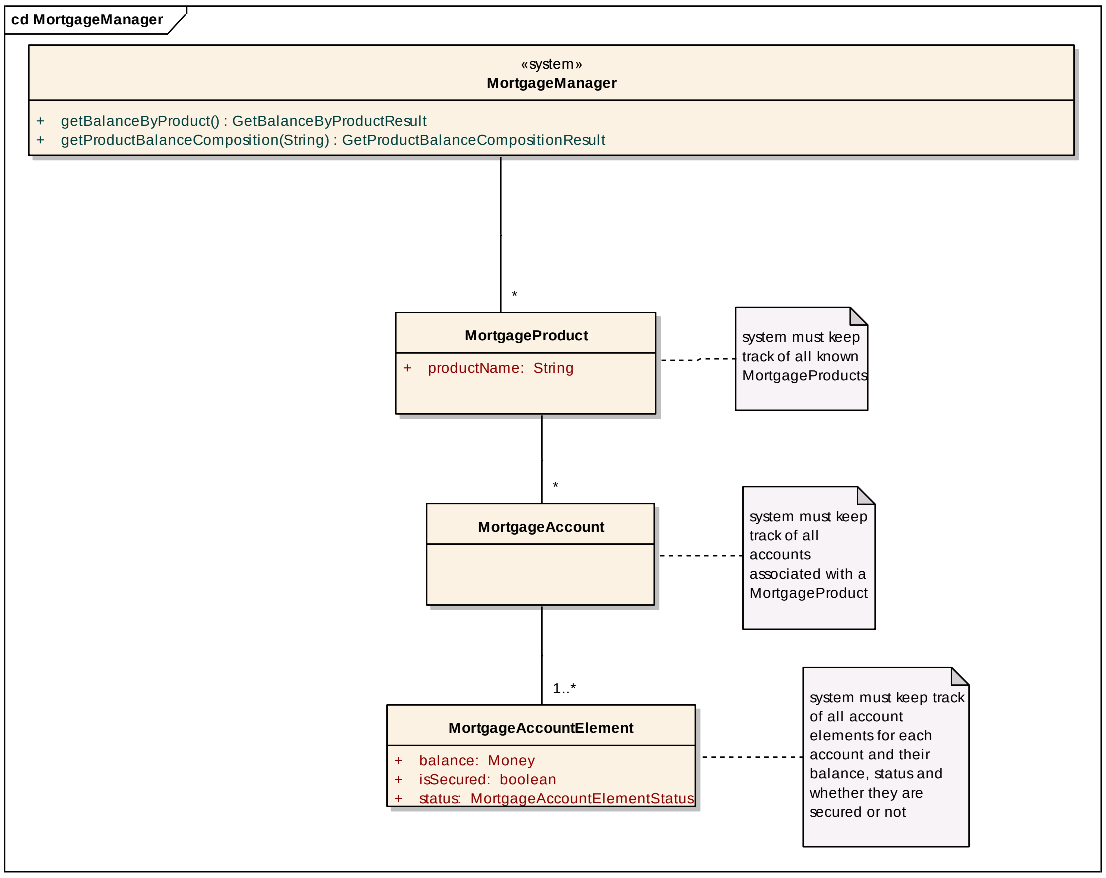

cd MortgageManager
«system»
MortgageManager
+ getBalanceByProduct() : GetBalanceByProductResult
+ getProductBalanceComposition(String) : GetProductBalanceCompositionResult
*
MortgageProduct
+ productName: String
system must keep track of all known MortgageProducts
*
MortgageAccount
system must keep track of all accounts associated with a MortgageProduct
1..*
MortgageAccountElement
+ balance: Money
+ isSecured: boolean
+ status: MortgageAccountElementStatus
system must keep track of all account elements for each account and their balance, status and whether they are secured or not

Figure 7: System Type Model with both operations and supporting state

The use case flow can now be improved, adding two business rules and inserting references to the operation specifications:

1. The actor asks the system for the mortgage balance breakdown by MortgageProduct.
2. The system calculates the balance on loan for all the MortgageAccounts, broken down by MortgageProduct, and displays the result to the actor (Screen 1) (B1)
3. The actor selects one of the products from the list and asks the system to show the balance composition for that product
4. The system calculates and shows the composition of the selected balance (Screen 2) (B2)

Business Rules

B1. the balance of account elements with status=redeemed must not be included (see getBalanceByProduct() operation specification)

B2. see the getProductBalanceComposition() operation specification for the derivation of the data shown to the user

As use case flows are brought into scope and analysed, the System Type Model grows by including all the types, attributes and associations needed to support the operation specifications, and the use case specifications updated to reflect the new, improved understanding.

The reader is reminded that the types in this model are not meant to stand for software classes with executable methods. Their role is to provide the vocabulary to support the behavioural specifications, hence they are stereotyped as <<specification>>. There is no implication that they will exist as classes in the implementation, although it would be a poor implementation that did not follow the structure of the specification.

Step 4 - Add precision with OCL

To increase our confidence that the desired behaviour is correctly and fully specified, we can write system operation specifications in OCL. This is an optional step. A reader unfamiliar with the basic concepts of OCL may wish to omit this sub-section.

We only provide an OCL specification for the first operation, getBalanceByProduct(), to demonstrate the approach.

As a convention, we write reserved OCL words in **bold**, and auxiliary definitions in *italics*. All OCL text is written in the `Courier` font.

To make our OCL operation specification simpler, we attach some definitions to the types in the model. We could have done this even if the specification was written only in English, however the discipline of writing OCL aids the discovery of these additional model elements. We define a query, *accountBalance*(), on the MortgageAccount type, to return the total balance of all its elements, as follows:

```
context MortgageAccount
def: accountBalance () : Money =
      self.MortgageAccountElement->
      select(status <> MortgageAccountElementStatus::Redeemed).
            balance->sum()
-- end definition
```

Writing the definition in OCL brings to light the potential problem of a MortgageAccount whose elements are all in the Redeemed state. In this case we want the balance to be zero, but the select operator would return an empty collection, and attempting to sum the balances of its elements would yield an undefined result. Let us rewrite the definition to cover this case explicitly:

```
context MortgageAccount
def: accountBalance () : Money =
      let
      activeElements = self.MortgageAccountElement->
            select(status <> MortgageAccountElementStatus::Redeemed)
      in
      if activeElements()->isEmpty()
      then Money (0.0)
      else activeElements.balance->sum()
      endif
-- end definition
```

We also attach a definition *productBalance*() to the MortgageProduct type, to return the total balance for all its accounts. Note how this definition makes use of the previous one on MortgageAccount.

```
context MortgageProduct
def: productBalance() : Money =
      self.MortgageAccount->accountBalance ->sum()
-- end definition
```

Finally, we attach a definition to ProductBalance, (the type used in the result of the operation, see figure 8). The definition, *derivedFromProduct*(), returns true if the ProductBalance is derived from the given MortgageProduct. Note how this definition uses the previous one on MortgageProduct.

```
context ProductBalance
def: derivedFromProduct(mp : MortgageProduct) : boolean =
self.productName = mp.productName
and
self.balance = mp.productBalance()
-- end definition
```

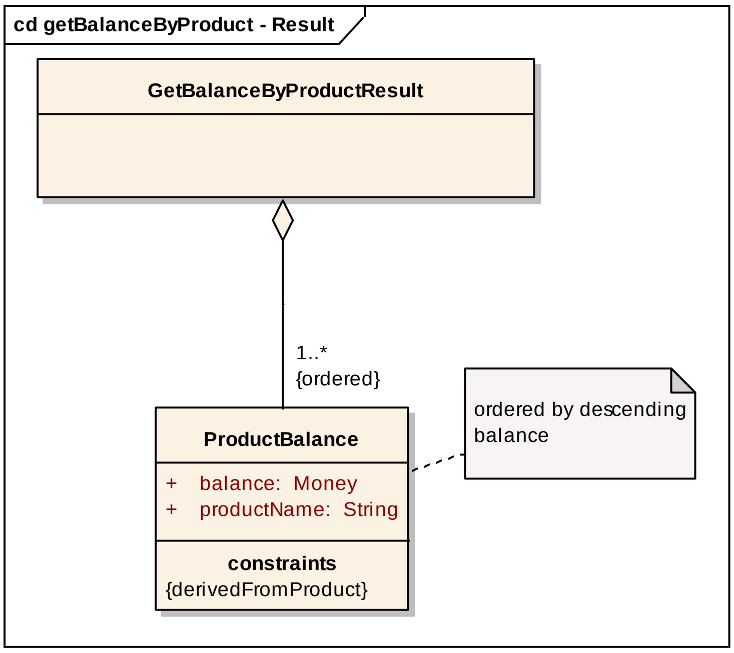


Figure 8: ProductBalance with the new definition

It is important to note that the definitions we have added are not UML methods, just like attributes are not instance variables. Definitions return values but do not alter the state of the objects. Writing them in OCL enforces this discipline since OCL expressions are side-effect free.

With these three auxiliary definitions, the first operation can be specified in OCL as follows:

```
-- signature
context MortgageManager::getBalanceByProduct ()
      : GetBalanceByProductResult
pre:
      true -- i.e. no precondition

-- postcondition starts here
post:

-- local definition (let … in) to select a product in the system by name
let
productWithName(name:String) : MortgageProduct =
      self.MortgageProduct->select(productName=name)
in
-- 'result' is an instance of GetBalanceByProductResult
-- and must contain one ProductBalance for each MortgageProduct
result.ProductBalance->size() = self.MortgageProduct->size()
and
-- the names in the result must come from the MortgageProducts
result.ProductBalance.productName =
      self.MortgageProduct->collect(productName)
and
```

```
-- each object within the result
-- must be derived from the MortgageProduct with the same name
result.ProductBalance->
      forAll(derivedFromProduct(productWithName(productName)) )
and
result.ProductBalance->sortedBy(-balance) --  descending order
-- end postcondition
```

Figure 9 shows the System Type model with the new definitions (OCL constraints) on MortgageProduct and MortgageAccount

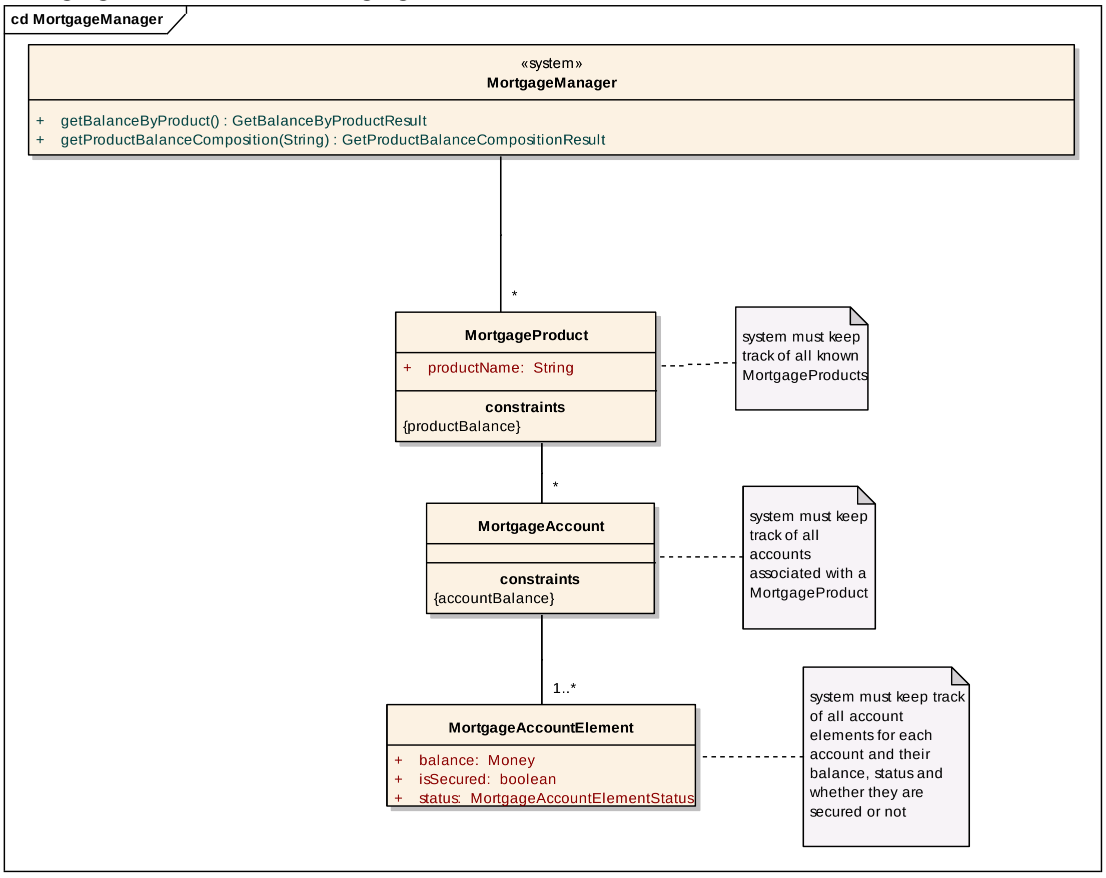

cd MortgageManager
«system»
MortgageManager
+ getBalanceByProduct() : GetBalanceByProductResult
+ getProductBalanceComposition(String) : GetProductBalanceCompositionResult
*
MortgageProduct
+ productName: String
constraints
{productBalance}
system must keep track of all known MortgageProducts
*
MortgageAccount
constraints
{accountBalance}
system must keep track of all accounts associated with a MortgageProduct
1..*
MortgageAccountElement
+ balance: Money
+ isSecured: boolean
+ status: MortgageAccountElementStatus
system must keep track of all account elements for each account and their balance, status and whether they are secured or not

Figure 9: System Type Model showing two new OCL constraints

OCL is not a mandatory part of our process, partly because it is not yet well supported by modelling tools. The value of writing OCL constraints lies in the positive effect that this formal discipline has on the quality of the UML model, uncovering behaviour and data that had been incompletely or wrongly specified, and providing a precise, unequivocal representation for business rules.

3 Review and conclusions

Our method offers a well defined process for moving from informal, imprecise descriptions to formal, precise specifications, providing traceability between the two levels. The method has been used with a high degree of success at RDF Group, a UK software house, for several years. A number of complex and mission critical financial applications have been delivered on time and within budget utilising the analysis method described in this paper, within an iterative and agile process loosely based on the Rational Unified Process [Jacobson99], where functionality is specified and moved into design, coding and testing in tight time-boxed timescales.

The main additional deliverable of our method is the System Type Model, with the associated system dialogues and system operation specifications. While it is undeniable that these artefacts need to be kept up-to-date as changes occur, this cost is far outweighed by the many tangible benefits:-

- The analysts writing use case descriptions can produce far better documents as a result of producing precise UML models in parallel with the narrative text. Issues can be identified and resolved earlier, and the impact of proposed changes assessed more accurately.

- Traceability is improved and design is simplified. Use case realisations can now be produced for each system operation, which are far more precisely specified and easier to implement correctly than textual narratives. In a multi-tier component-based implementation, component operations are linked to the system operations they implement, providing a guide to the designer, developer and maintainer. Business rules attached to specification types can be implemented as methods on the classes that implement those types. Value Object classes are traceable to the argument types used by the system operations. Database tables and fields are traceable to specification types.

- The added effort in analysis is also more than compensated by the savings achieved in downstream activities, as fewer queries and issues are raised in design, coding and testing.

References

[Coleman94] Derek Coleman, P.Arnold, S.Bodoff, C. Dollin, H.Chilchrisr, F. Hayes and P. Jeremaes, *Object-Oriented Development: the Fusion Method*, Prentice-Hall, 1994

[Cook94] Steve Cook and John Daniels, *Designing Object Systems - Object Oriented Modelling with Syntropy*, Prentice-Hall, 1994

[D'Souza99] Desmond D'Souza and Alan Wills, *Objects, Components and Frameworks with UML: the Catalysis Approach*, Addison-Wesley, 1999

[Fowler04] Martin Fowler, *UML Distilled Third Edition - A Brief Guide to the Standard Object Modeling Language*, Addison-Wesley 2004

[Jacobson99] Ivar Jacobson, Grady Booch and James Rumbaugh, *The Unified Software Development Process*, Addison-Wesley 1999

[Warmer03] Jos Warmer and Anneke Kleppe, *The Object Constraint Language, Second Edition - Getting Your Models Ready for MDA*, Addison-Wesley, 2003

Author

Franco Civello is Lead Analyst at RDF Group, Brighton, UK. He received a PhD in Software Engineering from the University of Brighton, and is a Member of the BCS and the IEEE and a Chartered Engineer. Contact: franco.civello@rdfgroup.com