# RSI - A Structured Approach
to
# Use Cases and HCI Design

Mark Collins-Cope,
Ratio Group Ltd.

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

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

We Know the Object

# Table of Contents

**ABSTRACT** .......... **4**

**1. INTRODUCTION** .......... **5**

1.1. USE CASE ANALYSIS - IVOR JACOBSON .......... 5
1.2. GOAL ORIENTED USE CASE ANALYSIS - ALISTAIR COCKBURN .......... 6
1.3. UML 1.3 «INCLUDES», «EXTENDS» AND SPECIALISATION .......... 7
1.4. DESIGN BY CONTRACT - BERTRAND MEYER .......... 8
1.5. LEVELS OF ABSTRACTION IN OBJECT MODELLING - COOK AND DANIELS .......... 9
1.6. HUMAN COMPUTER INTERFACE VIEW - VAN HARMELEN .......... 9

**2. MOTIVATION FOR THE RSI APPROACH** .......... **10**

2.1. EXPERIENCES AS TRAINERS AND MENTORS AT RATIO .......... 10
2.2. USE CASES IN PRACTICE .......... 10
***Driver : Enter Parkade (Car Park) with Ticket*** .......... *11*
***Success End Condition*** .......... *11*
***Failure End Condition*** .......... *11*
***Primary Success Scenario*** .......... *12*
***Exceptions and Recovery Steps*** .......... *12*

**3. THE RSI APPROACH** .......... **14**

3.1. OBJECTIVES .......... 14
3.2. CLASSIFICATION OF LEVELS OF GRANULARITY AND ABSTRACTION .......... 14
*3.2.1. Requirement Use Cases* .......... *14*
*3.2.2. Interface Use Cases* .......... *15*
*3.2.3. Service Use Cases* .......... *16*
3.3. INTER-RELATIONSHIPS BETWEEN THE MODELS .......... 17

**4. THE RSI PROCESS** .......... **19**

4.1. PROCESS OVERVIEW .......... 19
4.2. INCREMENTAL AND ITERATIVE DEVELOPMENT .......... 19
*4.2.1. Iterating* .......... *19*
*4.2.2. Increments* .......... *20*
4.3. DEVELOPING THE REQUIREMENT USE CASE MODEL .......... 20
*4.3.1. Inputs:* .......... *21*
*4.3.2. Outputs:* .......... *21*
*4.3.3. Sub-process:* .......... *21*
*4.3.4. Example - hotel reservation system - requirement use case model* .......... *21*
4.4. DEVELOPING THE INTERFACE USE CASE MODEL .......... 24
*4.4.1. Inputs:* .......... *24*
*4.4.2. Outputs:* .......... *24*
*4.4.3. Sub-process:* .......... *24*
*4.4.4. Example (continued) - hotel reservation system - interface use case model* .......... *25*
***interface model use case summary diagram*** .......... *25*
4.5. DEVELOPING THE SERVICE USE CASE MODEL .......... 29
*4.5.1. Inputs:* .......... *29*
*4.5.2. Outputs:* .......... *29*
*4.5.3. Sub-process:* .......... *29*

*4.5.4. Example (continued) - hotel reservation system service use case model* .......... *30*
4.6. CONSOLIDATION .......... 34
*4.6.1. Inputs:* .......... *34*
*4.6.2. Outputs:* .......... *34*
*4.6.3. Sub-process:* .......... *34*
*4.6.4. Example (continued) - hotel reservation system consolidated model* .......... *35*
4.7. PROCESS VARIATIONS .......... 37

**5. ONWARDS TO DEVELOPMENT .......... 38**

5.1. USE CASE REALISATION .......... 38
5.2. INTERFACE SEQUENCE DIAGRAMS .......... 38

**6. SUMMARY .......... 40**

**7. CREDITS .......... 42**

**8. REFERENCES .......... 42**

# Abstract

*Use case analysis is the de-facto standard technique for requirements capture in the OO development world. When engineers first undertake use case analysis, a number of issues are raised for which easy answers can't be found in the text books. These include:*

- *How does user-interface design fit into the use case analysis process?*
- *Should user interface dynamics be included?*
- *What is the appropriate level of granularity (size) and abstraction (detail) for use cases?*
- *If large grained use cases are used, should they be decomposed into 'lower level' use cases?*
- *If so, at what point should this decomposition stop, and how should these sub-use cases be used?*
- *What are the 'relationships' between these use cases?*
- *How should this decomposition be approached from a process perspective?*
- *How are used cases related to object models? Should the 'things' referred to in use case descriptions be cross-referenced against object models?*

*This paper describes the «requirements»/«service»/«interface» (RSI) approach to use case analysis. This approach provides a framework for analysing and understanding potential use case deliverables and their inter-relationships, with a view to answering the questions such as those detailed above.*

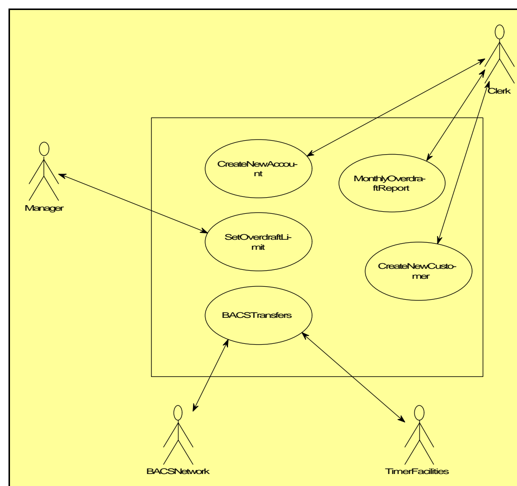
*In particular it puts user-interface design centre stage within the use case analysis process, provides a set of standard classifications for use case granularity, provides placeholders for other interface deliverables, and gives clear traceability from high level requirements to system interfaces.*

# 1. Introduction

## 1.1. Use case analysis - Ivor Jacobson

Use case analysis is a requirements capture technique that is most often used in the early stages of OO and component development projects.  It was first introduced by Ivar Jacobson in his book "Object Oriented Software Engineering" [1],  although the description of use case analysis presented in this book has, in practise, been interpreted in many different ways (one of the consequences of which is that, seemingly at least, no two use case analysis projects ever deliver the same information).

Broadly speaking, use cases assist in defining the functional requirements of computer systems. Following is an example use case diagram:




The use case diagram above is made up of the following elements:

- a bounding box - showing the scope of the system under specification (the inner box in the diagram above).
- a number of actors (stick men) representing the categories of users or systems which interact with the system under specification, in this case:
  - the manager of the bank
  - the clerks at the bank
  - the BACS network for inter-bank transfers, and
  - automated timing facilities
- a number of use cases (ovals) representing the 'business functions/processes' provided or supported by the system, in this case:
  - functions to set overdraft limits
  - functions to undertake BACS (automated inter-bank) transfers


RATIO

  - functions to create new customers and accounts, and
  - functions to produce a monthly overdraft report
- interaction flows (double ended arrows), showing which actors interact with which use cases.

Use case diagrams are typically supplemented with additional documentation, describing the detail of each use case.

Jacobson's use cases are structured around business processes for which automated support is required. Page 350, section 13.3.1 of *Object Oriented Software Engineering* [1] describes the use case 'Transfer goods between warehouses'. The description of this use case contains both individual 'commands' ("Foreman issues command to transfer goods between warehouses", "Truck driver notifies system that he has arrived at the warehouse", etc.), and detailed user interface designs (diagrams of dialogs are shown). Much of the description is concerned with describing user interface dynamics (e.g. "the foreman may move between fields by using the tab key.").

Jacobson introduced two relationships *between* use cases: «uses» and «extends». As the semantics of these relationships has now been superseded by the UML 1.3 specification further discussion of use case inter-relationships is left until later.

## 1.2. Goal oriented use case analysis - Alistair Cockburn

Cockburn's work ("Structuring Use Cases with Goals" [2]) recognised that use cases had "improved the situation that requirements documents [are] often an arbitrary collection of paragraphs, having a poor fit with both business re-engineering and implementation." He noted, however, that he had personally encountered 18 different definitions of use cases given by different, but expert teachers and consultants.

To remedy this, Cockburn went on to describe a goal-oriented approach to use case analysis, in which a use case is described in terms of a goal oriented sequence of business or work steps. The following use case extract is taken from Cockburn's paper [2]:

**Use case: get paid for car accident (insurance system):**
Actor - Claimant
Actor goal - to get paid for a car accident
1. Claimant submits claim with substantiating data;
2. Insurance company verifies claimant owns a valid policy
3. Insurance company assigns agent to examine case
4. Agent verifies all details are within policy guidelines
5. Insurance company pays claimant

<u>Extensions</u>

1a. Submitted data is incomplete
1a1 Insurance company requests missing information
1a2 Claimant supplies missing information
… etc.

Note that some steps of this description (e.g. step 1) are unlikely to be automated. Cockburn goes on to describe how each step within the description can be viewed as a mini 'use case' in its own right - with its own goals. Thus use cases are decomposed into sub-use cases. Cockburn describes the classification of use cases into 'summary goals', 'user goals', 'subfunctions' and 'dialog interactions' (at which level Cockburn describes goals such as 'move to next field using the tab key').

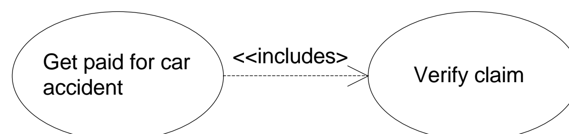
Cockburn also clarified the difference between a *scenario* and a *use case*. Essentially, a scenario can be thought of as a particular occurrence (or instance) of a use case - so for example: "Elizabeth Cullinan claiming for the car accident on 3 September 1998 at Chiswick Roundabout in London" would be a scenario generalised in the above use case.

The work presented here builds on Cockburn's approach. It differs in its classification of use case types; and in targeting the decomposition of use cases into interface and service sub-types.

## 1.3. UML 1.3 «includes», «extends» and specialisation

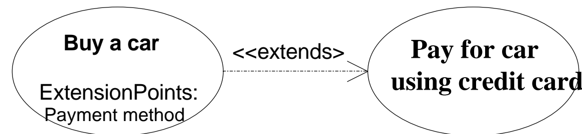
UML is a de-jure notational standard for object-oriented analysis and design notations (notations include use cases, class models, sequence diagrams, state models, etc.), administered by the Object Management Group (OMG). The current version (1.3) [15] defines three relationships between use cases:

- one use case «includes» another
  «includes» is used to indicate that one use case is effectively a 'subroutine' of another (B is a subroutine of A, in this case). For example, Cockburn's use case 'get paid for car accident' might «include» a use case 'verify claim' for step 2.



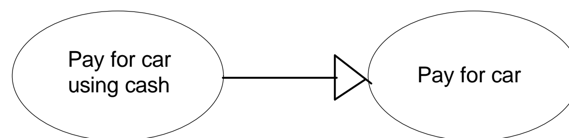

- one use case «extends» another
  «extends» is used to indicate one use case conditionally extending the behaviour of another. For example, a use case 'buy a car' might contain a sub-step 'pay for car with

cash' An extending use case might replace 'pay for car with cash' with 'pay for car using credit card.'




The use of «extends» is considered by some to be something of an anomaly. Anthony Simmons' paper *Use cases considered harmful* [3] (which perhaps ought to have been named "«extends» considered harmful") discusses the ever changing and somewhat peculiar 'come-from' and 'go-to' semantics of the «extends» structure.

- One use case is a specialisation of another
  Specialisation is used to indicate that one use case specialises the behaviour of another (one specialising - or making more 'concrete' - the behaviour of another). For example, a use case 'pay for car' could be specialised into 'pay for car using cash' or 'pay for car using credit card.'




RSI focuses on the use of «includes», though it does not preclude the use of other constructs.

## 1.4. Design by contract - Bertrand Meyer

Bertrand Meyer introduced design by contract in his seminal work *Object oriented software construction* [4]. Design by contract involves use of pre- and post-conditions to specifying the behaviour of operations, and use of invariants for restricting the possible behaviour of the types within a type model.

Pre-conditions specify the conditions which must apply before a particular operation (or use case) can be invoked. It is the responsibility of the user of the operation to ensure they are adhered to. Post-conditions specify the state of a system after an operation has completed - they effectively define the functionality of the operation. Invariants specify conditions that must always be true in a system, and are often used to capture business rules in a formal manner.

Pre-conditions, post-conditions and invariants are used extensively (and certainly not exclusively) within RSI to specify the behaviour low-level use cases against a corresponding core specification model.

## 1.5. Levels of abstraction in object modelling - Cook and Daniels

Steve Cook and John Daniels [5] noted that there are three perspectives you can use in modelling, most visible in their application to type or class models:

- the *conceptual* or *essential* view - whereby the objects in a diagram correspond to the objects and inter-relationships between the objects in the user's view of the world as whole (that is: not just those objects which are relevant to a system under specification or implementation),
- the *specification* view, which focuses on objects and interfaces to objects are used in the system, and
- the *implementation* view, which exposes all aspects of the objects bare.

RSI focuses on *conceptual* and *specification* models.

## 1.6. Human computer interface view - van Harmelen

The HCI (Human Computer Interface) and UML/object modelling worlds have, it appears, until recently retained an unhelpful degree of isolation from each other. In "Object-Oriented Modelling and Specification for User Interface Design" [6], Mark van Harmelen set about to break down some of these barriers in his description of Idiom, a user-interface design methodology.

Van Harmelen described Idiom's design artefacts as comprising of course and fine grained tasks (tasks are the HCI world's equivalents of use cases), which specify user-oriented activities in terms of a domain model, a core model, and a interaction model. Idiom also used pre- and post-conditions in specifications.

The domain model describes the user's world in terms of domain objects and their inter-relationships. The core model describes those domain objects which form part of the system to be implemented. The interaction model augments the system model by specifying view structures and transactions in the interactive application.

# 2. Motivation for the RSI Approach

## 2.1. Experiences as trainers and mentors at Ratio

Ratio is a training and consultancy company specialising in object and component based software development (see www.ratio.co.uk). As such, we have much experience in teaching and mentoring use case analysis to students new to the OO world, and to others who whilst familiar with OO programming, are new to UML and object modelling.

We have found that, whilst superficially a simple and easy to grasp technique, a number of questions (detailed in the abstract of this paper) repeatedly came up as use cases were put into action. These questions were key motivators in developing the RSI approach.

## 2.2. Use cases in practice

With these questions in mind, Ratio undertook an informal research project in early 1998 to see just how use cases were being used in practise. We collected around 10 examples of use cases from a self-selecting group of participants from around the world (on a commercial in confidence basis via the comp.object Usenet newsgroup), which we have since updated with input from more recently available literature.

The findings of this informal study were as follows:

- user interface was often omitted from use case descriptions - the notable exceptions to this being Jacobson's larger worked examples in *Object Oriented Software Engineering* [1] and Schneider and Winters examples in their book *Applying Use Cases - A Practical Guide* [7];

- Elements of user interface navigation and design were often implicit in use case descriptions. Descriptive text such as following (taken from a sample use cases obtained via comp.object) is illustrative of this:

1. The actor selects a customer.
2. The actor gets a list of the customer's accounts.
3. The actor selects an account to view.
4. The actor examines the details of the customer's account.
5. The actor continues with steps 2-4 until the correct account is chosen.
6. The actor adds creates an order on the account.

Or:

…
5) The system now displays these costs to the user using the cost display form. The total cost and the required user's contribution are both shown.
6) The system creates an invoice and adds this to the user's account.
7) The user elects to pay the balance using the 'pay now' button.
…

One concern regarding such descriptions is that the implicit user interface navigation may not be appropriate when user interface design is considered in more detail.

- different levels of granularity of use case were used. These varied from the 'business process' level - as described by Cockburn (see above) and as described by Jacobson et al. in *Software Reuse* [8] (page 220):

> A **business use case** is a sequence of work steps performed in a business system that produces a result of perceived and measurable value to an individual actor of the business.

- to the 'system interaction' level - e.g. the use of actions (the Catalysis equivalent of use cases) in Wills and DeSouza's *Catalysis* [10] (see example below), and "define a style", "change a style" as described in Fowler's *UML Distilled* [9] (see page 44 in which Fowler discusses the difference between user goals and system interactions).

- some use case descriptions were accompanied by a type or class diagram, as best exemplified throughout Wills and DeSouza's *Catalysis.* We found these greatly increased the clarity of use cases when considered from a system designers perspective;

- Cockburn's guidelines [2] for use cases at the 'business process' level were the most consistently applied, as exemplified by the following example from Pete McBreen of McBreen Consulting in Canada (petemcbreen@acm.org).

**Driver : Enter Parkade (Car Park) with Ticket**

Driver requests entrance to parkade, receives ticket and then enters parking lot.

**Success End Condition**

Driver is allowed entrance to Lot.
A Ticket is issued that records the date/time of entrance.
The available spaces in the parkade are decremented by 1 (Available Spaces >= 0 )
Lot Full signs illuminated if Available Spaces == 0

**Failure End Condition**

Driver is not allowed entrance to Parkade.

**Primary Success Scenario**

| Step | Actor | Action |
|---|---|---|
| 1 | Driver | Request Entrance to Parkade |
| 2 | System | Dispense Ticket |
| 3 | Driver | Take Ticket |
| 4 | System | Admit Vehicle |
| 5 | System | Decrement Available Spaces by 1 |

**Exceptions and Recovery Steps**

| Step | Condition/Action |
|---|---|
| 1a | Parkade Full |
| 2a | No Tickets |
| 2a1 | System: Display Entrance out of Order - Failure |
| 3a | Ticket not taken after 20 Secs |
| 3a1 | System: Rescind Ticket - Failure |
| 4a | Vehicle does not go through entrance after 20 seconds |
| 4a1 | System: Rescind Ticket - Failure |

- the target audiences for use case descriptions clearly varied: some being targeted at end users, some at system developers. Cockburn's use cases (see above) are clearly 'user friendly', whilst Wills and DeSouza's actions (see example below) in *Catalysis* have a more formal nature targetted at software engineers.

- with the exception of Wills and DeSouza's [10] approach, none provided detailed descriptions of exactly *what* information was passing to and from the system to the outside world. Wills and DeSouza do this in the following manner:

**use case** buy_thing (purchaser, vendor, thing)
**post:** vendor.possessions -= thing -- *vendor no longer owns thing*
**and** purchaser.possesions += thing -- *purchaser now owns thing*

In this example, the use case requires a purchaser, vendor and a thing to be identified before it can act. Note that the use case parameters in this instance have clear implications on the user interface.

- the structure of the decomposition of use cases was often fairly arbitrary, with «includes» and «extends» being used as text management aids, rather than being structured to assist in the ensuing system development.

- only Wills and DeSouza's *Catalysis* [10] *clearly identified* a set of atomic system functions that the system would offer (the actions), and even *Catalysis* seems to focus only on functions which change system state.

Combining the results of this informal study with our own experience, we reached the following conclusions:

- the key activity of user interface design was being neglected in the use case analysis process, often being relegated to that ubiquitous black hole: 'supplementary documentation' (for which read: no real focus on this activity);
- although the interpretation of what constituted a use case varied from project to project, using some form of use case analysis was considered beneficial in all cases;
- use cases could be used effectively at *varying* levels of granularity and abstraction;
- different target audiences required a different form of use case structure and description;
- informal classifications of use case granularity were already in use, and these could be built upon;
- type modelling added greatly to the clarity of use case descriptions when targeted at system developers;
- more structure could and should be applied to assist in the decomposition process for use cases;

This work culminated in the initial development and publication of the RSI approach [11,12], and the inclusion of the principles of the RSI approach on a large (hundreds of man-year) component development system development being undertaken by Andersen Consulting for a major finance sector customer, the experiences of which are discussed as appropriate throughout this paper.

# 3. The RSI Approach

## 3.1. Objectives

The objectives of the RSI approach to use case analysis are as follows:

- to provide a *guideline framework* for analysing and considering: what levels of granularity/abstraction of use cases can be used within a use case analysis process; under what circumstances the different levels of granularity/abstraction should be used; how the different levels of granularity/abstraction should be documented; and to whom descriptions should be targeted:

- to provide a clear place for *user interface design* in use case analysis process, whilst maintaining a clear separation between interface (dialog box, button) and core domain (bank, account, customer) concerns;

- to provide a clearly defined scalable *process* linking together the varying levels of use case described above, that enables initial descriptions of requirements to be traced through to low level use cases.

- to encourage use case descriptions to be *cross-referenced against a type model* when targeted at system developers.

- to *structure* the deliverables of the use case analysis process to assist in the ensuing development process.

These objectives relate directly to the questions posed in the abstract to this paper.

## 3.2. Classification of levels of granularity and abstraction

The RSI approach provides three classifications for use case granularity, represented by the UML stereotypes «requirement», «interface» and «service».

### 3.2.1. Requirement Use Cases

A requirement use case defines a business or work process for which some automated system support may be required (e.g. "sale of goods", "resource a course", "open a new account", etc.). Requirement use cases may be documented in the manner described by Cockburn [2], detailing:
- the actor,
- the objective of the use case,

- a goal oriented decomposition of the use case into a sequence of steps - labelled according to whether they are to be automated or not
- a list of any extensions dealing with deviations and exceptions in the normal sequence of events (Cockburn's exception numbering convention is particularly useful in these descriptions)

Requirement use cases provide the starting point from which decisions regarding the scope and phasing of the functionality can be made. The individual steps (sub-goals) within the description are candidates for potential automation.

The objectives of developing the requirements use case model are to clearly document the business drivers for a system in as concise a manner as possible. Requirement use cases may be decomposed (using the «includes» relationship) for the purposes of removing duplication of text across multiple use case descriptions.

The target audience for requirement use cases is primarily end users. Developing requirement use cases falls within the remit of a typical business analyst's role.

### 3.2.2. Interface Use Cases

Interface use cases describe functionality that is concerned with managing the interface between the actors of a system and the underlying services it offers. Interface use cases undertake the role of 'translating' the information provided by an actor into a form acceptable to a set of underlying service use cases. To do this, they factor out the elements of functionality that are more related to the interface than to the underlying system. Interface use cases may be documented as follows:

- the objectives of the interface
- a detailed description of any interface formats used. This might include user interface design (dialogs, etc.), report layouts, file formats, etc.
- a step by step description of the functional aspects of the interface such as user interface dynamics (e.g. "when the 'select customer' button is clicked, the 'accounts' list-box is refreshed"), file processing, how report contents are made up, etc.

An alternative to the latter two documentation items is to develop an interface prototype.

The objectives of developing the interface use case model are to clearly document (or demonstrate in the case of an interface prototype) the interfaces used by a system in as concise a manner as possible, providing traceability to antecedent requirement use case descriptions. Interfaces may be decomposed using the «includes» stereotype to show use of one interface use case (and user interface element) by another.

The target audience for interfaces use cases is primarily end users. Developing interfaces falls within the remit of the user-interface designer's role.

### 3.2.3. Service Use Cases

Service use cases define the underlying functions offered by the functional core of a system in a manner independent of any particular interface concerns.

Service use cases are specified as follows:

- a list of input parameters detailing the information (if any) that is passed to the use case from its calling environment (e.g. customer, account, date, etc). Inputs are instances of types in the service use case model's association 'core domain specification type model' (thankfully abbreviated to *core specification model).*
- a list of output parameters, detailing the information (if any) that is passed from the use case to its calling environment upon termination (e.g. set of overdrawn customers, all new accounts, etc.). Outputs are instances of types in the service use case models association core specification model.
- the pre-conditions of the use case, describing what conditions must be true of the system before the use case can be initiated. Pre-condition descriptions cross-reference the core specification model, and when circumstances dictate may be formally specified using the UML Object Constraint Language (OCL) [7].
- the post-conditions of the use case, describing any changes that will be made to the internal state of the system once the use case has been completed. Post-condition descriptions cross-reference the core specification model, and may also be formally specified in OCL.

The target audience for service use cases is primarily system developers (rather than end users). Hence a more formal and concise format for the documentation is considered appropriate.

It may sometimes be appropriate to indicate that one service may be built up from another that exists *independently in its own right* using the «includes» stereotype and relationship. However service use cases should not in general be decomposed further - as this would be to intrude into the realm of the software design process.

Two flavours of service use cases emerge: update services - those that change system state, and query services - those that retrieve information without changing system state.

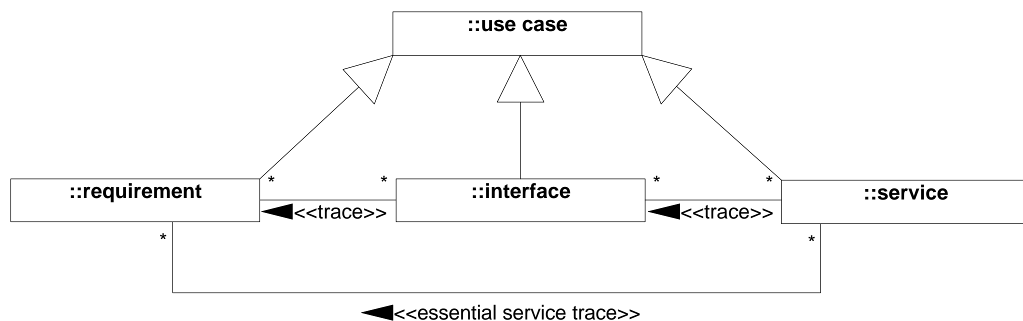
The input parameters to query service use cases are typically used to provide a selection criteria by which sets of objects within the system will be identified. These are then returned in the output parameters. The input parameters to update service use cases are typically objects returned by a previous invocation of a query service. update service use cases do not deal with selection criteria. It is the job of the interface use cases to tie these two threads of activity together. This distinction is important in that it encourages re-use of update service use cases independently of any selection mechanism that might be used.

Another sub-classification of services also proves useful: those that are part of the *essential* service set, and those that are only part of the consolidated service set (the essential set plus any others). The essential service set is made up of those services use cases that are directly mandated by the requirement use cases regardless of any interface design decisions. This distinction is important in analysing the impact in changes to requirements using the traceability model, and in that the essential service use case set provides a clear definition of the atomic functions the system will implement.

The target audience for service use cases is system architects, designers and developers. Developing the service use case model falls within the remit of the system designer's role.

## 3.3. Inter-relationships between the models

The following diagram summarises the relationships between the categorisations of use cases at a high level (the full meta-model for the RSI approach is shown in section 3.6.3 of this document):

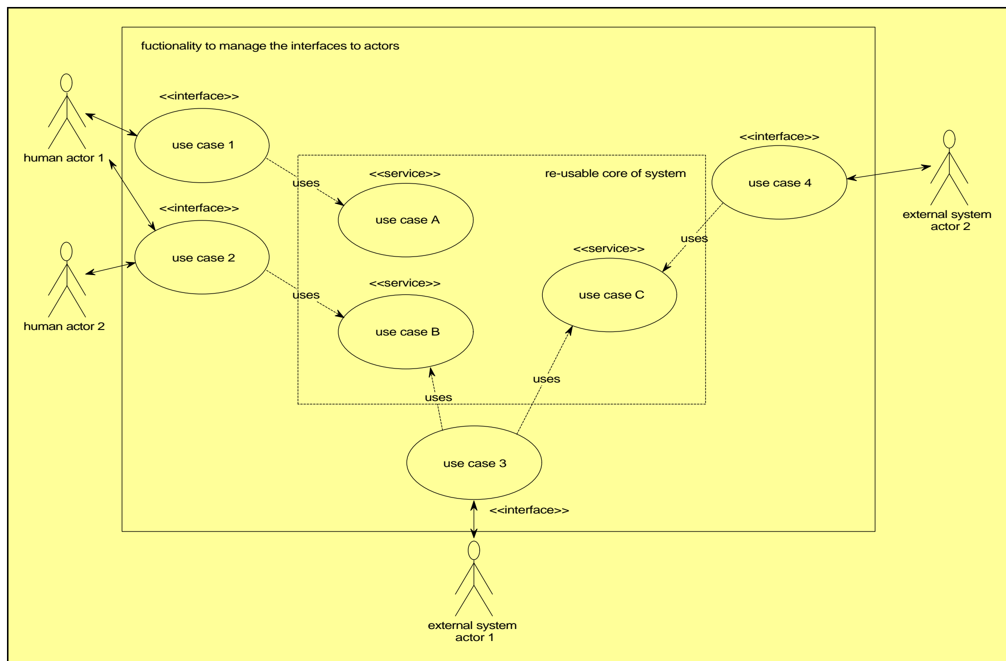



Requirement use cases provide the starting point for the use case analysis process. They are refined into service and interface use cases necessary to implement those parts of the requirement it is decided to automate. The inter-relationships between requirement and interface/service use cases is shown using «trace» dependencies on a use case traceability diagram.

Interface use cases use service use cases to gain access to the functional core of the system. The update/query subdivision of services is used in the following manner:

- queries are used by interface use cases to provide information necessary for the construction of (user) interface dynamics. For example, in a banking system double-clicking on a particular customer in a ‘customers' list-box may cause an associated ‘accounts’ list-box to be refreshed with the chosen customer's accounts. To retrieve this information, a query use case would be used ('select accounts by customer' in this case).

- update service use cases are used by interface use cases to change the internal system state (to actually "do" something). The parameters of the update service use cases will have typically been returned by a previous invocation of a query service use case. The inter-relationships between interface and service use cases may be shown using a «trace» on the traceability model.




The separation of service and interface use cases partitions functionality into the elements that are primarily concerned with manipulating the peculiarities of an interface, and the elements that are fundamental to the system in question. The motivations for this separation are twofold: to encourage the re-use of service use cases (from a variety of interface use cases), and to enable a clear factoring out of interface concerns in the specification process (particularly as RSI encourages a semi-formal or formal approach to service use case specification).

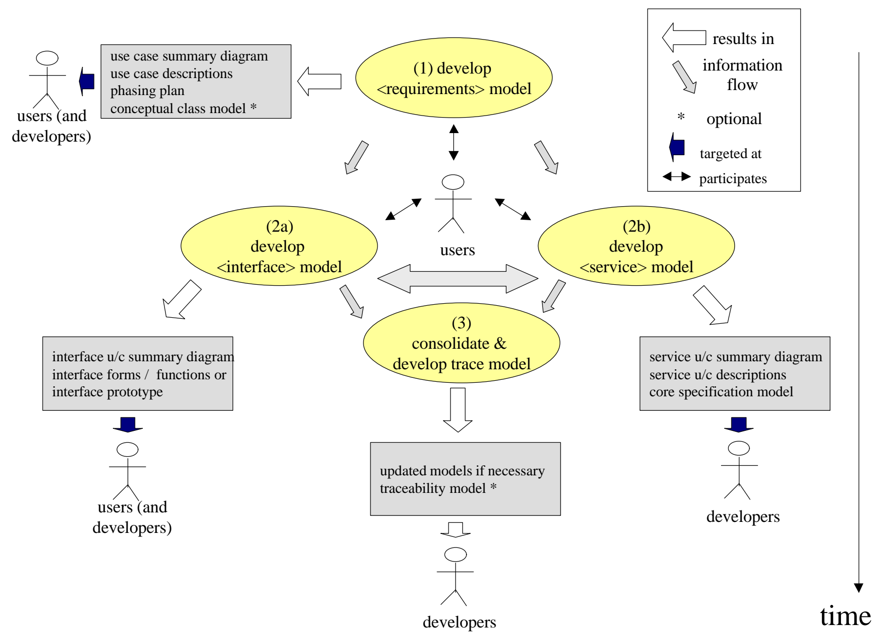
Often the service/interface separation is clear-cut, most often when the actor is a human user. Sometime it is not entirely clear where the dividing line between them should be. When this occurs it is necessary to consider what aspects of functionality would change if the nature or the details of the interface changed. The aspects that remain constant constitute part of the service use case set; the aspects that vary constitute part of the interface use case set.

# 4. The RSI Process

## 4.1. Process overview

The following description breaks down the overall process of developing an RSI model into four stages - two of which (stages 2a. and 2b.) are often undertaken in parallel.

1. developing the requirements use case model (conceptualisation);
2a. developing the interface use case model (specification);
2b. developing the service use case model (specification);
3. Consolidating the models and producing the traceability model (specification).




## 4.2. Incremental and iterative development

### 4.2.1. Iterating

The term iteration can be used to indicate going back and reworking existing project deliverables. This type of iteration occurs on projects for a number of reasons: mistakes may have been made in earlier deliverables which need correcting, a second project increment will often require existing deliverables to be revisited, or requirements may change mandating a rework of existing deliverables.
An alternative use of the term iteration is to indicate that certain steps with a process continue until certain completeness and correctness (or quality) criteria are met. This may occur when



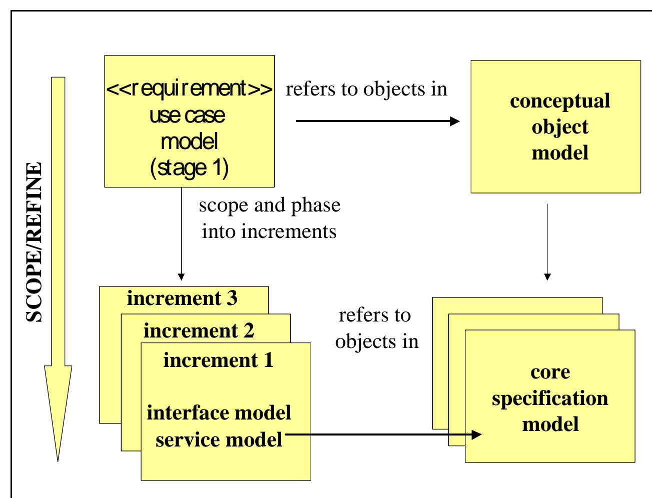
RATIO

two activities are so tightly bound that there appears to be a dependency link in both directions between the activities.

Both forms of iteration occur with RSI, the latter being the relationship between the creation of the interface use case model and the service use case model.

#### 4.2.2. Increments

An incremental approach to development divides the delivery of a system up into a number of phases. The criteria by which this division is made may vary, but a functional split is often used.




RSI favours the functional division of a system at the end of the requirements stage of the process. The functionality of a particular phase may be determined on the basis of priority ("we just have to have this function.") or risk ("we're not quite sure what we want here.") or both. Other criteria for incremental phasing (such as managing technical risk) are beyond the scope of this discussion, and so not considered further here (although in project management terms they are very important).

### 4.3. Developing the requirement use case model

The requirement use case model is the first part of the model to be developed, and provides the starting point from which all other models can be traced.

The process description here assumes some form of ad-hoc project start-up document. On small projects it may be possible to develop a requirement model from scratch based on discussions with users. On larger projects, some form of antecedent document will almost certainly be necessary.

#### 4.3.1. Inputs:

- ad-hoc requirements or project start-up document;

#### 4.3.2. Outputs:

- requirement use case summary diagram(s);
- requirement use case textual descriptions
- requirement use case scoping and phasing plan;
- (optionally) conceptual domain object model;

#### 4.3.3. Sub-process:

- Review ad-hoc requirements document with users:
  – identify any business processes implied by the requirements document;
  – add these to the candidate requirement use case set;
- Consider some or all candidate requirement use cases in conjunction with users (if there are a very large number, some pre-requirements scoping may be necessary):
  – identifying the main flows;
  – develop exception flows asking: "what could go wrong?"; "what might vary?";
  – develop exception flows to exception flows in the same manner, until no further exceptions are found;
  – if a conceptual object model is being developed, consider the impact of the above on it;
- Finalise and review all deliverables, iterating as necessary;

The requirement use case model is necessarily low in detail as it is the basis upon which the project will be scoped and phased. Including more detail would lead to a great deal of wasted efforts documenting steps that may never get automated!

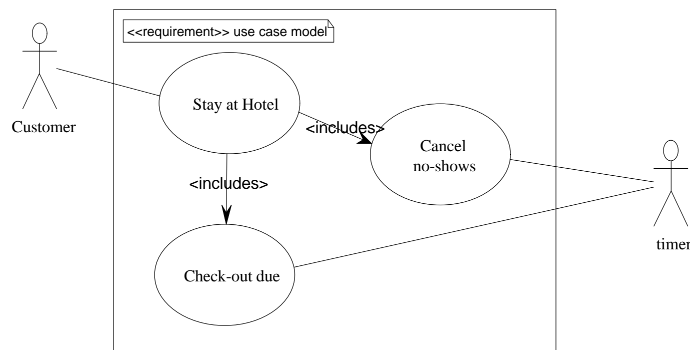
Experience using requirement use cases on the Andersen Consulting project mentioned above suggests that the exception flow analysis is particularly useful in ensuring a complete model of the system is generated. The project also demonstrated that accurate estimates can be made at the end of this phase.

#### 4.3.4. Example - hotel reservation system - requirement use case model

The worked example shown is based on the following ad-hoc requirements:

| Reference | Problem Statement |
|---|---|
| A. | The guest makes a reservation. |
| B. | The hotel will take as many reservations as it has rooms available. |

| | |
|---|---|
| C. | When a guest arrives, he or she is processed by the registration clerk. |
| D. | The clerk will check the details provided by the guest with those that are recorded. |
| E. | Sometimes, guests do not make a reservation before they arrive. |
| F. | Some guests want to stay in non-smoking rooms. |
| G. | When a guest leaves the Hotel, he or she is again processed by the registration clerk. |
| H. | The clerk checks the details of the guest staying and prints a bill. |
| I. | The guest pays the bill, and leaves the Hotel and the room becomes unoccupied. |




**requirement use case model - summary diagram**

It is worth noting that although they are low in detailed information content, use case diagrams can provide an excellent overview of a system. On large projects such as the Andersen Consulting development discussed in this it is necessary to develop multiple use case diagrams, generally subdivided by business area.

The 'stay at hotel' use case is described as follows:

*Actor: Customer*
*Objective: To book and stay in a room.*

Normal flow: [assumes everything goes fine]

1. The customer contacts the hotel and requests a booking (A)
2. The clerk processes the booking (automated) (B - must check availability) (F - must check if smoking preference available).

3. The customer turns up at the hotel to check in. (C)
4. The clerk processes the customer (automated) (C) (D - checks details against reservation) (E - guest may not have reservation) (F - check smoking preferences) confirms the customer has arrived.
5. The customer stays in the room.
6. The customer requests check out. (G)
7. The clerk prints the bill showing the cost, date, and a description of each item, (G) (H) and the room is freed up. (automated)
8. The customer checks the bill and pays it (automated). (I)

Extensions:

[ These are generated by taking each step in the above normal flow, and asking: what could possibly go wrong? 1a: indicates something is wrong with step 1; 1a1, etc. indicates alternative steps taken, etc. ]

2a. No room is available for the required dates (B)
2a1 Use case terminates. No further action. No record on interaction on system.

2b. The customers smoking preference is unavailable, and they will compromise.
2b1. Continue normal flow.

2c. The customers smoking preference in unavailable, and they won't compromise.
2c1. Bye bye customer. Use case terminates. No further action.

3a. The customer doesn't turn up.
3a1. The system checks reservations every midnight, and cancels no-shows (automated - see requirement use case 'cancel no-shows')

3b. The customer cancels the reservation.
3b1. The clerk cancels the reservation (automated)

4a The customer doesn't have a reservation.
4a1. The clerk checks availability and creates a reservation as per step 2, normal flow continues. [note: step 2 error conditions may apply]

4b. The customers details as per the reservation are incorrect (but smoking preference is ok)
4b1. The clerk updates them to be correct (automated).

4c. The customer smoking preference is wrong.
4c1. The clerk cancels the existing booking (automated) and makes a new reservation as per step 2.

7a. The customer doesn't check out.

7a1. The system alerts the clerk after check-out time that the customer hasn't checked out (automated - see requirement use case 'check-out due')
etc.

**requirement 'stay at hotel'**

Arguably, the above business process could have been structured as two: 'make a reservation' and 'arrive at hotel.' This would have been appropriate if further requirements could share them - and would have been shown using the «includes» trace on the use case diagram.

A full description of the requirement use case model for this system can be found in the companion document to this paper on www.ratio.co.uk/rsi-info.htm.

## 4.4. Developing the interface use case model

The interface use case model documents the user-interface of the proposed system (it can also be used to document external system interfaces - however these are not the focus of the discussion here). Unlike the other models that form part of the RSI deliverable set, it is inherently dependent on technical domain decisions such as the GUI chosen for development.

### 4.4.1. Inputs:

- requirement use case summary diagram(s);
- requirement use case textual descriptions;
- core specification domain model(s);
- service use cases definition (developed in parallel).

### 4.4.2. Outputs:

- interface use case summary diagram(s);
- interface use case textual/diagrammatic descriptions or interface prototype;
- updated core specification domain model;

### 4.4.3. Sub-process:

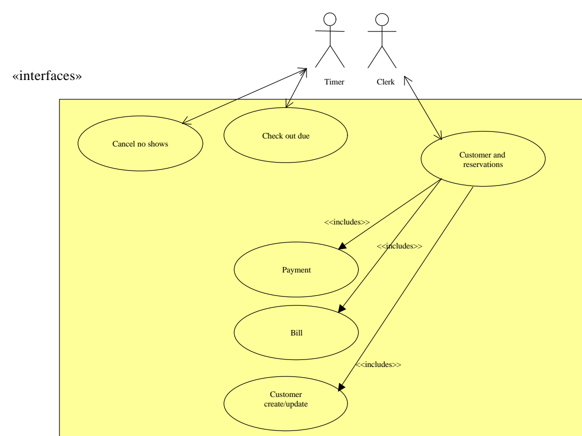
- Consider the requirement use case model with users, taking into account typical volumes of candidate objects;
- Develop and refine the proposed interface (either on paper or as a prototype) with users;
- Check completeness by ensuring all automated requirement steps can be traced from at least one interface;
- Finalise and review all deliverables, iterating as necessary;

The essential elements of the service model (under parallel development) may help in user-interface design:

- service descriptions may assist in that:
  - inputs typically indicate information that must be collected from the user;
  - outputs indicate information that must be displayed to the user;
  - pre-conditions indicate conditions which the interface must adhere to before invoking an underlying service, and may suggest validation rules for the interface, or preferably, ways of presenting information to the user which ensure that they are always met - which don't allow the user to make a mistake;
  - post-conditions may indicate information which may need to be fed back to the user;

- the core specification model may assist in that:
  - associations suggest intermediate routes by which object may be identified and selected on the user-interface
  - association multiplicitys suggest appropriate user-interface constructs: fixed multiplicitys (1 to 2, 1 to 4, etc.) hint at the use of constructs such as radio-buttons, check-boxes etc.; variable multiplicitys (1 to *, etc.) hint at the use of list-boxes and drop-down lists, etc.
  - the *relative* volumes of its types indicate whether it will be appropriate to navigate directly between them on the interface;

4.4.4. Example (continued) - hotel reservation system - interface use case model




**interface model use case summary diagram**

Following is an example interface use case ('customer and reservation management'):

Actor: *Clerk*

Objective: *to enable the management of customers and reservations.*

**Format**

Customer/Reservation Management

Customer

surname collins-c

find new...

Collins-Cope, Liz S
Collins-Cope, Nathan NS
Collins-Cope, Mark S
Collins-Cope, Lewis NS

details... print bill

payment... reservations

Reservation

start date 21-1-00

end date 22-1-00

availability existing

21 20/1/00 24-1-00 S --------
24 17/1/00 04-2-00 S ---------
01 20/1/00 24-1-00 S ---------
22 20/1/00 24-1-00 S ---------

delete customer

check-in check-out reserve

**interface use case 'customer/reservation management'**

The impact of the essential service use case model (under parallel development - see section 4.5.4) on this interface design is as follows:

- the association between Hotel and rooms in the core specification model is one to many, so a listbox has been chosen to show the rooms.

- the pre-conditions to service use case 'make reservation' state that the room being reserved must be available for the dates specified. The interface guarantees that this is always the case by disabling the 'reserve' button except when a room has been selected in the reservations list-box and the reservation list-box has been populated using the 'availability' button (note: a customer must also be selected).

- the specification model contains associations between Hotel and Customer, and between Customer and Reservation. Hotel is the anchor point on the specification model. This route is exploited on the interface: having selected a customer, clicking the 'reservations' button will refresh the listbox in the 'Reservations' section of the interface with the Customer's Reservations.

A full description of the interface semantics of this dialog (including cross-references to underlying service use cases (not normally shown) is shown in the functionality section for this interface (see below). A diagrammatic format is used here to show the dynamics and operation of the interface - text annotations are attached to the relevant parts of the interface dialog. An alternative approach is to list these annotations separately in a purely textual description. The most preferable approach is to demonstrate the dynamics in action using a screen prototype of the system. Such an approach may use a set of dummy services that return pre-defined (possibly configurable) sets of objects to be shown in the interface.

*find* button is disabled if no characters in *surname* field

*New* button always available. Clicking *new* button pops up <<interface>> Customer details with all fields blank. On return, customer is entered into list box and selected.

If a *listbox* contains one item, the item is selected by default.

Clicking the *existing* button causes reservations between dates specified above to be displayed in the listbox. Uses <<query>> find reservations by dates.

Clicking the *availability* button causes room between the given dates to be displayed in the listbox. Uses <<query>> find availablity by dates

This group of buttons is disabled unless two dates are filled in above

Clicking *find* button populates listbox with matching names. Uses <<query>> Find customers by name

Customer/Reservation Management

Customer

surname: collins-c

find | new...

| | |
|---|---|
| Collins-Cope, Liz | S |
| Collins-Cope, Nathan | NS |
| Collins-Cope, Mark | S |
| Collins-Cope, Lewis | NS |

details... | print bill

payment... | reservations

Reservation

start date: 21-1-00

end date: 22-1-00

availability | existing

21 20/1/00 24-1-00 S --------
24 17/1/00 04-2-00 S ---------
01 20/1/00 24-1-00 S ---------
22 20/1/00 24-1-00 S ---------

delete | customer

check-in | check-out | reserve

See main document.

This group of buttons is disabled unless a name is selected in the listbox above.

Clicking the *deletei* button causes the selected Reservation to be deleted. Uses <<update>> delete reservation

Clicking the *details* button causes the <<interface>> Customer details to be poed-up, with details as per the selected Customer.

Clicking the *customer* button causes the selected Reservation's customer to be displayed opposite Uses <<query>> Get customer by reservation

Clicking the *payment... button* causes the Payment <<interface>> to be pop-up, with the selected Customers details showing.

Clicking the *print bill …* button causes the bill to be printed as per the print bill <<interface>> Uses <query> Get bill details.

This group of buttons is disabled unless a customer and reservation are selected

Clicking the *reservations* button and Reservation listbox is filled with selected Customers details.

Creates a reservation using <<update>> make reservations

Clicking *check-in* button conforms check-in. Uses <<update>> confirm check-in.

*Check-out* confirms customer check-out. Uses <<update>> delete reservation.

We Know the Object

Following is an example of an interface use case being used to specify a report layout:

**Format:**

**Bill for <surname>, <first name>**

**Printed: dd/mm/yy**

| Item | Date | Value | Balance |
|---|---|---|---|
| ccccccccccccccccccc | dd/mm/yy | #####.## | ######.## |
| … | | | |
| Total: | | | ######.## |

**Functionality:**

1. Lines within the report are filled in using '«query» get bill details'.

**interface use case 'bill'**

A full description of the requirement use case model for this system can be found in the companion document to this paper on www.ratio.co.uk/rsi-info.htm.

### 4.5. Developing the service use case model

The service use case model is developed on a per increment basis.

The consolidated service use case model is comprised of the essential service use cases plus additional query use cases mandated by interface design.

4.5.1. Inputs:

- requirement use case summary diagram;
- requirement use case textual descriptions;
- (optionally) conceptual domain object model;
- interfaces use case model (developed in parallel);

4.5.2. Outputs:

- service use case summary diagram (essential and full models);
- service use case textual descriptions (essential and full models);
- core specification model;

4.5.3. Sub-process:

- Consider the requirement use case model with users

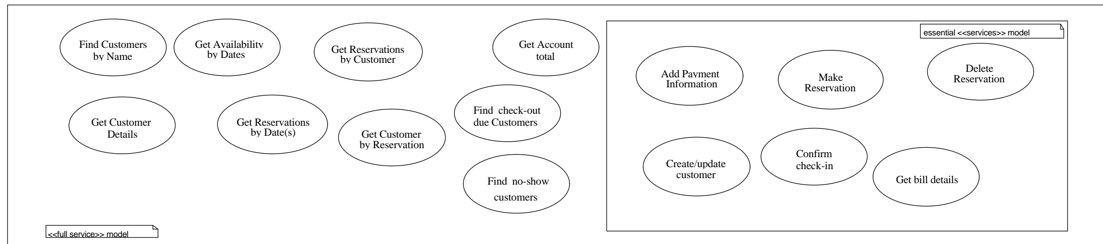
  - identify any service use cases directly implied by the requirement use case model
  - add them to the candidate service use case list
- Consider the emerging interface use case model
  - identify any queries implied by it that are not in the essential service use case set.
  - add them to the candidate service use case list
- Consider the candidate service use case list:
  - undertake a preliminary analysis of service use case inputs, outputs, pre- and post-conditions, forming a candidate object list
  - update the core specification model accordingly
  - refine model and add any invariants.
- When the core specification model is stable, document the service use cases against it
- Finalise and review all deliverables, iterating as necessary.

The interface use case model (under development in parallel) is required to develop the majority of the query parts of the service use case model. Much of a user interface will be concerned with functionality to allow the user to locate objects upon which to subsequently apply updates. This functionality will be provided by queries - hence the dependency to the interface use case model. Updates, on the other hand, are simply given objects on which to act - finding those objects is the role of queries. Consequently they can be inferred directly from the requirement use case model - assuming it is complete!

The impact of the interface use case model on the service use case model can be summarised as follows. A user-interface will have associated with it a number of actions (dynamics), some of which will require support from the functional core of the system:

- those which simply populate other parts of the user interface as an aid to helping the user locate another underlying object are candidates for the provision of queries;
- intermediate objects (those which are used to populate user interface controls such as list-boxes) are candidates for types in the core specification model, and are likely to be have direct or indirect associations to the object which the user is trying to locate;
- the type of user-interface control used (listbox, radio-button, etc.) will be indicative of the multiplicities of such associations;
- those which change system state are candidates for updates;

### 4.5.4. Example (continued) - hotel reservation system service use case model




Note that, with the exception of 'Get bill details' the essential service use case model is comprised of only updates. The consolidated service use case model contains all the queries, of course.

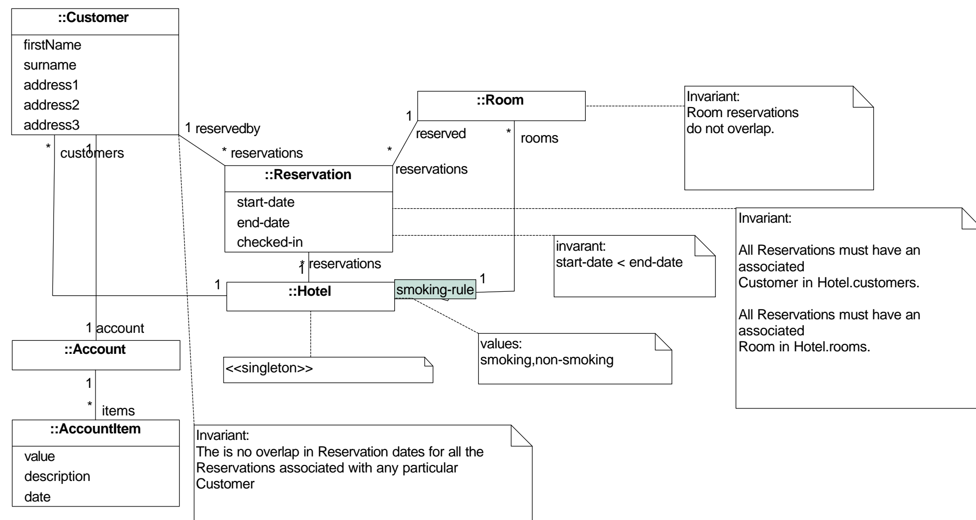
'Get bill details' is derived from step 7 of requirement 'Stay at Hotel':

| | |
|---|---|
| 7. | The clerk prints the bill showing the cost, date, and a description of each item, (G) (H) and the room is freed up. (automated) |

This mandates the contents of the bill to be the cost, date and description of each item, and so this query can be considered interface independent. Even this, however, may be considered a premature addition of detail.

Another potential query candidate for the essential service use case set might have been 'find no-show customers.' The interface use case corresponding to this service has no associated formats, and so it might be considered that this query is independent of the interface. This would be a mistake, however, as the decision not to have any interface formats is not mandated in the requirement use case model, and may be subject to change during later revisions of the interface use case model.

Service use cases descriptions cross-reference the following core specification model:




The model shown contains a number of invariants (business rules) which must be adhered to at all times (e.g. "Room reservations do not overlap"). Using invariants reduces the complexity and removes redundancy in pre- and post-condition descriptions (which are, conceptually at least, implicitly 'and'-ed with them at the beginning and end of each use case).

Note that the core specification model must contain at least one 'anchor point' (Hotel in this example). Pre- and post-conditions are always described in a manner that is relative to some known point within the model (e.g. a post-condition stating that 'aCustomer.reservations is empty' is relative to aCustomer). Without a well known anchor point, it would not be possible to find a way into the model!

Two example service use case specifications are shown below. A full description of the service use case model for this system can be found in the companion document to this paper on www.ratio.co.uk/rsi-info.htm.

**in:- [what information is passed from the actor to the system]**
*start-date, end-date, aRoom, aCustomer;*
**out:- [what information does the system pass back to the actor]**
*status [shorthand for ok/fail]*
**pre:-conditions:- [what must be true before this service can be invoked]**
*start-date <= end-date;*
*aRoom is available from start-date to end-date inclusive;*
*aCustomer does not have any existing Reservations between start-date and end-date inclusive;*
**post:-conditions:- [what state changes will have taken place when this use case ends]**
*aReservation is added to Hotel.reservations, such that*

- *aReservation.start-date = start-date*
- *aReservation.end-date = end-date*
- *aReservation.reservedby = aCustomer*
- *aReservation.reserved = aRoom*

**update service use case 'make reservation'**

The descriptions above use a number of semi-formal conventions (based loosely on OCL), which experience shows assist in removing ambiguity:

- Names that correspond to types in the specification model are capitalised;
- Use of an instance of a type in the specification model is indicated by pre-fixing by 'a' or 'an' onto the type name. Additional qualification (aFirstCustomer, etc.) may be required if more than one instance needs to be identified.
- Use of attributes is shown using a '.' between the type name and the attribute name of the model. So 'aReservation.start-date' indicates the start-date attribute of aReservation.
- Navigation via associations is shown the same convention: so 'aReservation.reservedby' is the Customer who has reserved the Room identified by Reservation.reserved.
- All associations on the model are, at this stage, assumed to be bi-directional, so if aReservation.reservedby = aCustomer, aCustomer.reserves = aReservation.

Note that 'Make reservation' adds a new instance of a Reservation to the model. When an instance is added (or deleted or updated) it must be added *to* something (or deleted *from* or updated *in*). In this case, a Reservation instance is added to the set of reservations associated with (the one and only) Hotel. Hotel provides the context within which the Reservation instance is added - hence its inclusion in the post-condition description.

Fans of OCL, or those on projects which demand a high degree of formality, may choose to document the above example in a more formal manner as follows [7]:

```
Hotel::MakeReservation(start:Date, end:Date, aRoom:Room, aCustomer:Customer)
pre:  ( start <= end )  and
      ( aRoom.reservations->forAll(  (end < start-date) or (start  > end-date)  ) ) and
      ( aCustomer.reservations->forAll(  (end < start-date) or (start > end-date)  ) )

post:  ( self.reservations->includes(aReservation) ) and
       ( aReservation.start-date = start ) and
       ( aReservation.end-date = end ) and
       ( aReservation.reservedby = aCustomer ) and
```

| ( aReservation.reserved = aRoom ) |
|---|

Note, however, that this description makes the possibly premature decision that the 'make reservation' service use case will be implemented as a method on the Hotel type, which isn't necessarily the case.

**in:-**
none.
**out:-**
*aNoShowCustomerSet*
**pre:-**
*none*
**post:-**
*aNoShowCustomerSet contains all the Customers in Hotel.customers with an associated Reservation in which start-date is today, and checked-in is set to false.*

**query service use case 'find no-show customers'**

## 4.6. Consolidation

### 4.6.1. Inputs:

- all previous models

### 4.6.2. Outputs:

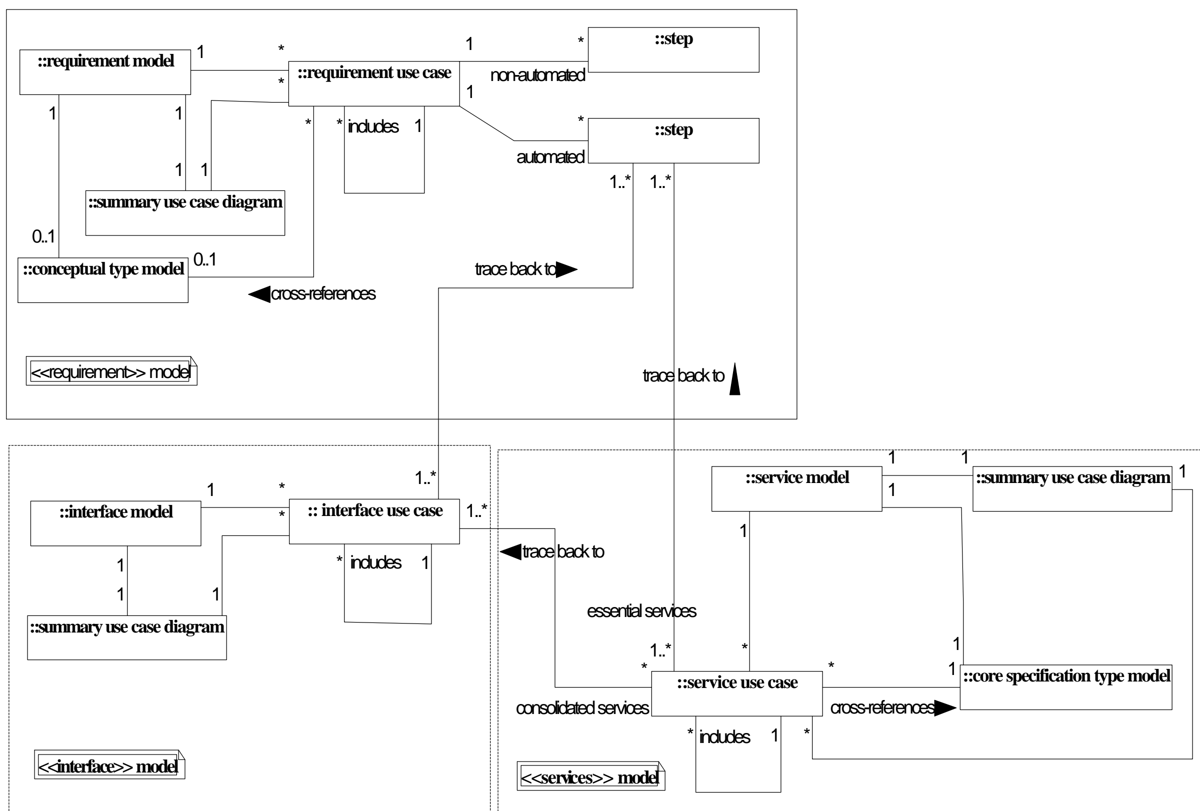
- updated models as required
- RSI trace model

### 4.6.3. Sub-process:

- Develop a trace model showing the dependencies:
  - from *essential* service use case set to requirements service use case set
  - from all service use cases to interface use cases
  - from interface use cases to requirement use case steps
- Ensure that:
  - for each service use case to automated requirement use case step dependency, there is a corresponding pair of service use case to interface use case and interface use case to automated requirement use case step dependencies

  - each service use case can be traced to at least one interface use case, and each essential service use case can be traced back to at least one automated requirement use case step.
  - each interface use case can be traced to at least one automated requirement use case step, and each automated requirement use case step can be traced to at least one interface use case step.
- Iterate, updating models if any trace deficiencies are found, until model is complete.

The RSI meta-model shows trace dependencies in their broader context:




## 4.6.4. Example (continued) - hotel reservation system consolidated model

The consolidated model for the hotel reservation is as described previously, with the addition of the following traceability diagram:

«requirements»
<<includes>>
Check out due
Stay at hotel
Customer
Timer
<<includes>>
Cancel no shows

Timer
Clerk

Cancel no shows
Check out due
Customer and reservations
Make reservation <<update>>
Delete reservation <<update>>
<<includes>>
<<includes>>
Payment
essential «services»
Add payment information <<update>>
Bill
Get bill details <<query>>
<<includes>>
Customer create/update
Create customer <<update>>
Update customer <<update>>
«interfaces»

Find customers by name <<query>>
Get availability by dates <<query>>
Find no show customers <<query>>
Get account total <<query>>
Get customer details <<query>>
Get reservations by dates <<query>>
Find check-out due customers <<query>>
Get reservations by customer <<query>>
Get customer by reservation <<query>>
consolidated «services» = essential + these

The traceability model shown here is useful in establishing the impact of change on the system. If a given requirement is varied in some way, the model enables the potential impact of this change to be traced through to the interface and service use case models. Our experience of using RSI has shown, however, that maintaining such models requires case tool support for any sizeable project.

## 4.7. Process Variations

A process should at least reflect the cause and effect dependencies that exist between its deliverables. In the case of RSI, these are that:

- an essential service use case is dependent on one or more steps within the requirement use case model;
- an interface use cases is dependent on one or more steps within the requirement use case model;
- a non-essential service use case (those in the consolidated services use case model) is dependent on one or more interface use cases;

So theoretically at least, we could envisage a project in which the service use cases and interface use cases were developed as each step within the requirement use case emerged. The downside of this approach is that it would lead to piecemeal development of service use cases and interface use cases, which did not consider the more holistic aspects of the analysis and design process. Hence the emphasis on project increments within RSI. A more holistic view of the dependencies between RSI deliverables is thus that:

- essential service use cases are dependent on steps within the current increment's requirement use cases;
- interface use cases are dependent on one or more steps within the current increment's requirement use cases;
- non-essential service use cases (those in the consolidated service use case model) are dependent on the current increment's interface use case;

Given this, the following processes could be used for developing the RSI models:
*Variation 1:*

1. The current increment's requirement use cases are first developed in full;
2. The essential service use case model is developed;
3. The interface use case model is developed;
4. The consolidated service use case and traceability model is developed;

*Variation 2:*

1. The current increment's requirement use cases are first developed in full;
2. The interface use case model is developed;
3. The consolidated service use case and traceability model is developed;

Both of these approaches maintain the holistic dependencies chain detailed earlier. Experience has shown that the downside of both these approaches is that they loose the potential synergy of developing the interface use case and service use case model together (generally in a small team), and so they should not be considered unless circumstances dictate that the preferred process is not viable.

# 5. Onwards to development

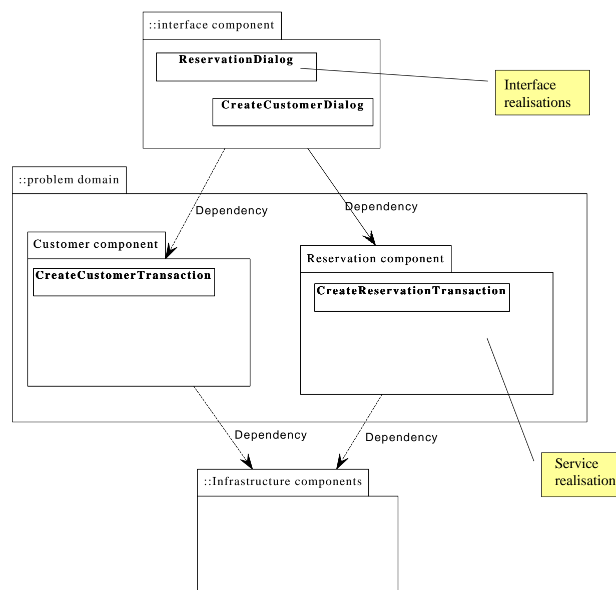
A full understanding of the RSI approach (and the benefits of the structural decomposition RSI gives) requires a brief discussion of the development process which follows RSI.

## 5.1. Use case realisation

Both interface and service use cases provide a good starting point for software design. Interface use cases will generally be initiated on the application menu, and are generally realised as specialisations of GUI classes (e.g. the customer and reservation management dialog may be implemented as a subclass of a GUI class library's dialog box class). Such classes are akin to Jacobson's boundary classes [1].

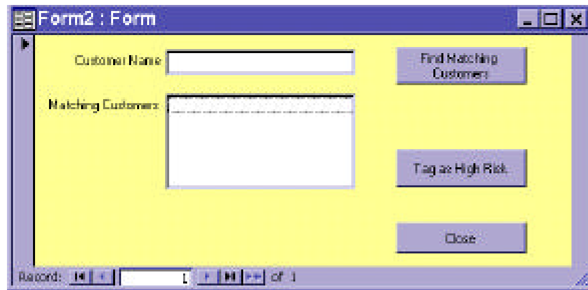
Service use cases are realised as classes or methods on classes on the façade of the components (as in component based development) that make up the system. For example 'make reservation' may well form a method on the façade of a reservation component interface. In a transaction oriented database system, 'make reservation' may well end up of a sub-class of a use case class that encapsulates the (database) transaction begin and commit commands. In general, service use case provide a good starting point for a component based decomposition of a system.

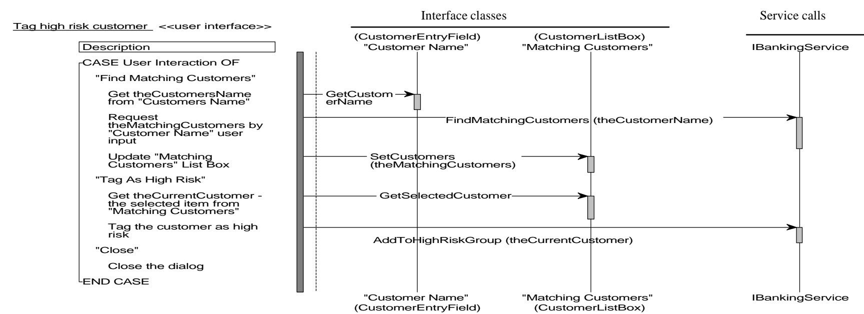
From an architectural perspective, RSI typically leads to the following application structure:




## 5.2. Interface sequence diagrams

The underlying software design of interface classes may be verified using sequence diagrams, although in practise we have not undertaken this activity most of the time.




Note in particular the interface/service boundary.

# 6. Summary

RSI structures use cases into three categories that reflect different levels of granularity (size of activity described in the use case) and abstraction (level of detail described in the use case).

RSI provides a *guideline framework* for analysing and considering: what levels of granularity and abstraction of use cases can be used within a use case analysis process; under what circumstances the different levels of granularity/abstraction should be used; how the different levels of granularity/abstraction should be documented; and to whom descriptions should be targeted.

It does this by structuring use cases into three categories which subdivide use cases by their granularity and the level detail of descriptive detail they contain, all of which reflect aspects of the somewhat varied real world usage of use case (as demonstrated by available literature or examples obtained in our informal survey):

- requirement use cases which:
  - describe business or work processes at a low level of detail,
  - are developed as part of the business analysis process,
  - are primarily targeted towards end users,
  - form the basis of project scoping an phasing;
- interface use cases which:
  - describe atomic system functions with a high level of detail;
  - which are developed by user-interface designers;
  - are primarily targeted at end users;
  - provide a clear placeholder in the use case analysis process for user-interface design.
- service use cases which:
  - describe the functionality offered by the functional core of a system;
  - are developed by system designers, are targeted towards system developers;
  - provide a hook from which a formal or semi-formal specification process can proceed, based on use of an associated type model.

  The *essential* elements of the service model also provides a clearly defined list of the *bounded atomic system functions* that are necessary to implement requirements, something clearly missing in many use case descriptions.

All three levels of use case described in RSI may be appropriate on a project at different stages in the analysis process, which RSI breaks into four stages for larger projects:

1. developing the requirement use case model (conceptualisation) - during which ad-hoc requirements are mapped onto requirement use cases documenting business or work processes.

2a. developing the interface use case model (specification) - during which requirement use cases may be decomposed and refined into interface use cases documenting system interface formats and associated functionality, providing a clear place-holder for *user interface design* in use case analysis process, whilst maintaining a clear separation between interface (dialog box, button) and core domain (bank, account, customer) concerns.

2b. developing the service use case model (specification) - during which requirement and interface use cases are refined into service use cases, which we encourage to be cross-referenced against an associated type model (the core specification model), either formally or informally.

3. consolidating the models and producing the traceability model.

RSI *structures* the deliverables of the use case analysis process to assist in the ensuing development process: Interface use cases clearly defining interface of the system to the outside world and providing a starting point for boundary class definition; service use cases providing a starting point for the allocation of behaviour to component facades, being implemented either as methods on classes in the facades, or as use case (control) objects in their own right.

Use cases are the de-facto standard for object oriented requirements analysis (that is: analysis of requirements that will be implemented in an object oriented fashion). If one wishes to improve the process of requirements and functionality analysis, and to put user-interface design centre stage in this process, then in the OO world at least, it would seem there is no realistic option but to do this within the context of use case analysis. We believe that RSI does this.

*Further information and discussion of the RSI approach to use case analysis can be found by joining the use-case discussion group at eGroups.com. To subscribe, send an email to* use-case-subscribe@eGroups.com.

This work is dedicated to the memory of Michael John Cope.

# 7. Credits

I would like to thank: Thomas Mowbray, Pete McBreen, Steven Kefford, Alan Williams, Hubert Matthews and Greg Gibson for their work in reviewing earlier drafts of RSI papers. Particular thanks are due to Benedict Heal and Hubert Matthews of Ratio for their contribution to the RSI approach, and to Keith Haviland, Andy Vautier and Nigel Barnes of Andersen Consulting for their support of this work.

# 8. References

[1] Ivar Jacobson et al., Object Oriented Software Engineering - A Use Case Driven Approach, Addison-Wesley, 1994.

[2] Alistair Cockburn - Structuring Use Cases With Goals - Sep/Oct and Oct/Nov issues of the Journal of Object Oriented Programming 1997 - also available on the following web site: http://members.aol.com/acockburn/papers/usecases.htm.

[3] Anthony Simmons - Use Cases Considered Harmful - Proceedings of TOOLS 29, 7-10 June, 1999, Nancy, France - published by IEEE Computer Society.

[4] Bertrand Meyer - Object Oriented Software Construction - Prentice Hall, 1994.

[5] Steve Cook and John Daniels - Designing Object Systems: Object-Oriented Modelling with Syntropy - Prentice Hall, 1994.

[6] Mark van Harmelen - Object-Oriented Modelling and Specification for User Interface Design - Proceedings EG Workshop on Design, Specification and Verification of Interactive Systems, Springer-Verlag, 1996, pp199-231.

[7] Geri Schneider and Jason P. Winters - Applying Use Cases - Addison Wesley, 1998.

[8] Ivar Jacobson et al. - The Unified Software Development Process - Addison Wesley, 1999.

[9] Martin Fowler - UML Distilled - Addison-Wesley 1997.

[10] Desmond Frances D'Souza and Alan Cameron Wills - Objects, Components and Frameworks with UML, The Catalysis Approach - Addison Wesley, 1999.

[11] Mark Collins-Cope - UML Use Case Analysis Pattern - October 28/29 1998 - Patterns '98 - published by Eric Leach Marketing.

[12] Mark Collins-Cope - The RSI Approach to Use Case Analysis - Proceedings of TOOLS 29, 7-10 June, 1999, Nancy, France - published by IEEE Computer Society.

[13] The Object Constraint Language - Precise Modelling with UML - Addison Wesley, 1998.

[14] Mark Collins-Cope - Companion document to RSI - A Structured Approach to Use Cases and HCI Design (PDF Format) - www.ratio.co.uk/techlibrary.htm (also available in Word '97 format by email request).

[15] OMG UML v.1.3 - http://www.omg.org/cgi-bin/doc?ad/99-06-09.