Chapter A1

Agate Ltd Case Study—Introduction

Agate Ltd

A1.1 Introduction to Agate

Agate is an advertising agency in Birmingham, UK. Agate was formed as a partnership in 1982 by three advertising executives, Amarjeet Grewal, Gordon Anderson and Tim Eng (the name is a combination of their initials). Amarjeet and Gordon had previously worked for one of the UK's largest and most successful advertising companies in London, but felt frustrated at the lack of control they had over the direction of the campaigns they worked on. As a result, they moved to the West Midlands region of the UK and set up their own business in 1981. Shortly afterwards they were joined by Tim Eng, with whom they had worked on a project in Hong Kong, and Agate was formed.

In 1987, the three partners formed a UK limited company and between them own all the shares in it. Gordon Anderson is Managing Director, Amarjeet Grewal is Finance Director and Tim Eng is Creative Director. They now employ about 50 staff at their office in the centre of Birmingham (see Fig. A1.1) and a further 100 or so at seven offices around the world. Each of these other offices is set up locally as a company with the shares owned jointly by Agate and the local directors.

Figure A1.1 Staff at Agate Ltd UK office.

Initially the company concentrated on work for the UK motor industry, which has declined in scale in recent years (although much of what remains is still located in the West Midlands region). However, as the company has expanded and internationalized, the type of work it takes on has changed and it now has clients across a wide range of manufacturing and service industries.

The company strategy is to continue to grow slowly and to develop an international market. The directors would like to obtain business from more large multinational companies. They feel

that they can offer a high standard of service in designing advertising campaigns that have a global theme but are localized for different markets around the world.

The company's information systems strategy has a focus on developing systems that can support this international business. Not long ago, the directors decided to invest in hardware and software to support digital video editing. This saved money on subcontracting the video-editing work, and with cheap broadband access they have the capability for fast file-transfer of digital video between offices. Now they are considering whether the company should also install its own video streaming servers for use in the growing market for online advertising.

A1.2 Existing Computer Systems

Agate already uses computers extensively. Like most companies in the world of design and creativity, Agate uses Apple Macintosh computers for its graphic designers and other design-oriented staff. The secretaries and personal assistants also use Apple Macs. However, the company also uses PCs to run accounts software in Microsoft Windows. Despite all this, Agate has been slow to install computer systems that support other business processes, such as tracking clients and managing campaigns. Last year, Agate had a basic business system for the UK office developed in Delphi for Windows. However, after the system was developed, the directors of Agate decided that it should have a system developed in Java, the object-oriented language originated by Sun Microsystems Inc. One of the reasons for the choice of Java was that it is portable across different hardware platforms and the company wants software that could run both on the PCs and on the Macs. Unfortunately, the person who developed the Delphi software for the company (and was going to rewrite it in Java) was headhunted by an American software house, because of her skills in Java, and has moved to the USA. Fortunately, this developer, Mandy Botnick, was methodical in her work and has left Agate with some object-oriented system documentation for the system she designed and developed.

This existing system is limited in its scope: it only covers core business information requirements within Agate. It was intended that it would be extended to cover most of Agate's activities and to deal with the international way in which the business operates.

A1.3 Business Activities in the Current System

Agate deals with other companies that it calls clients. A record is kept of each client company, and each client company has one person who is the main contact person within that company. His or her name and contact details are kept in the client record. Similarly, Agate nominates a member of staff—a director, an account manager or a member of the creative team—to be the contact for each client.

Clients have advertising campaigns, and a record is kept of every campaign. One member of Agate's staff, again either a director or an account manager, manages each campaign. Other staff may work on a campaign and Agate operates a project-based management structure, which means that staff may be working on more than one project at a time. For each project they work on, they are answerable to the manager of that project, who may or may not be their own line manager.

When a campaign starts, the manager responsible estimates the likely cost of the campaign, and agrees it with the client. A finish date may be set for a campaign at any time, and may be changed. When the campaign is completed, an actual completion date and the actual cost are recorded. When the client pays, the payment date is recorded. Each campaign includes one or more adverts. Adverts can be one of several types:

- newspaper advert—including written copy, graphics and photographs
- magazine advert—including written copy, graphics and photographs
- Internet advert—including written copy, graphics, photographs and animations
- TV advert—using video, library film, actors, voice-overs, music etc.
- radio advert—using audio, actors, voice-overs, music etc.
- poster advert—using graphics, photographs, actors

- leaflet—including written copy, graphics and photographs.

Purchasing assistants are responsible for buying space in newspapers and magazines, space on advertising hoardings, and TV or radio air-time. The actual cost of a campaign is calculated from a range of information. This includes:

- cost of staff time for graphics, copy-writing etc.
- cost of studio time and actors
- cost of copyright material—photographs, music, library film
- cost of space in newspapers, air-time and advertising hoardings
- Agate's margin on services and products bought in.

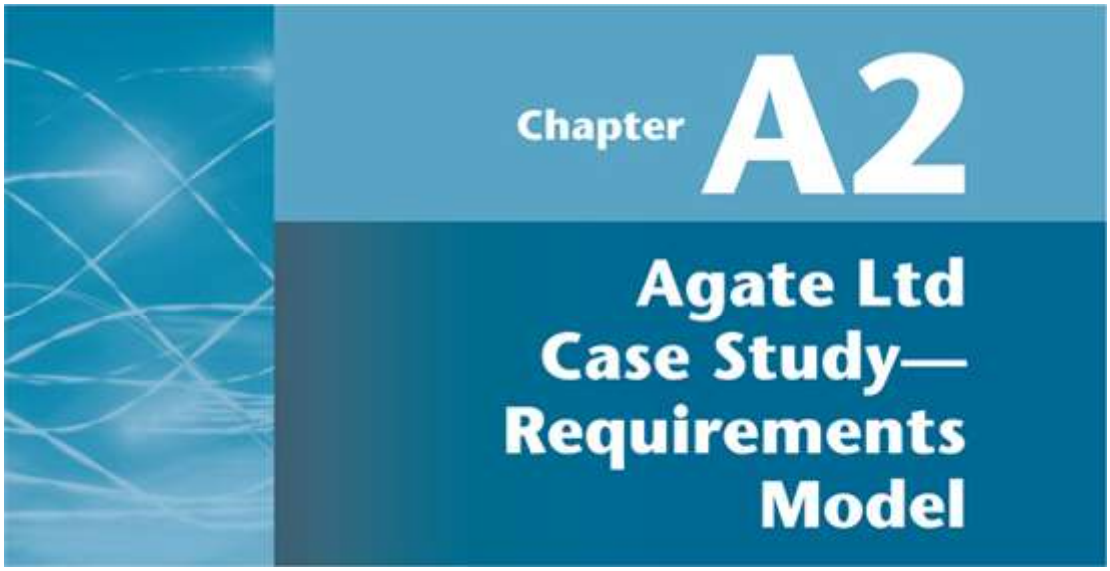
This information is held in a paper-based filing system, but the total estimated cost and the final actual cost of a campaign are held on the new computer system.

The new system also holds the salary grades and pay rates for the staff, so that the cost of staff time on projects can be calculated from the timesheets that they fill out. This functionality has been partially implemented and is not used in the existing system.

A1.4 Summary of Requirements

This section summarizes the requirements for the new system.

1. **To record details of Agate's clients and the advertising campaigns for those clients.**
   - 1.1 To record names, address and contact details for each client.
   - 1.2 To record the details of each campaign for each client. This will include the title of the campaign, planned start and finish dates, estimated costs, budgets, actual costs and dates, and the current state of completion.
   - 1.3 To provide information that can be used in the separate accounts system for invoicing clients for campaigns.
   - 1.4 To record payments for campaigns that are also recorded in the separate accounts system.
   - 1.5 To record which staff are working on which campaigns, including the campaign manager for each campaign.
   - 1.6 To record which staff are assigned as staff contacts to clients.
   - 1.7 To check on the status of campaigns and whether they are within budget.
2. **To provide creative staff with a means for recording details of adverts and the products of the creative process that leads to the development of concepts for campaigns and adverts.**
   - 2.1 To allow creative staff to record notes of ideas for campaigns and adverts.
   - 2.2 To provide other staff with access to these concept notes.
   - 2.3 To record details of adverts, including the progress on their production.
   - 2.4 To schedule the dates when adverts will be run.
3. **To record details of all staff in the company.**
   - 3.1 To maintain staff records for creative and administrative staff.
   - 3.2 To maintain details of staff grades and the pay for those grades.
   - 3.3 To record which staff are on which grade.
   - 3.4 To calculate the annual bonus for all staff.
4. **Non-functional requirements.**
   - 4.1 To enable data about clients, campaigns, adverts and staff to be shared between offices.
   - 4.2 To allow the system to be modified to work in different languages.



Agate Ltd

A2.1 Introduction

In this chapter we bring together the models (diagrams and supporting textual information) that constitute the requirements model. In Chapters 5 and 6 we have introduced the following UML diagrams:

- use case diagram
- activity diagram
- package diagram.

There is not the space in this book to produce a complete requirements model. However, in this chapter we have included a sample of the diagrams and other information. This is done to illustrate the kind of material that should be brought together in a requirements model. We have also tried to illustrate how iteration of the model will produce versions of the model that are elaborated with more detail.

A2.2 Requirements List

The requirements list on the next page includes a column to show which use cases provide the functionality of each requirement. This requirements list includes some use cases not in the first iteration of the use case model.

No.	Requirement	Use Case(s)
1	To record names, address and contact details for each client	Add a new client
2	To record the details of each campaign for each client. This will include the title of the campaign, planned start and finish dates, estimated costs, budgets, actual costs and dates, and the current state of completion	Add a new campaign
3	To provide information that can be used in the separate accounts system for invoicing clients for campaigns	Record completion of a campaign
4	To record payments for campaigns that are also recorded in the separate accounts system	Record client payment
5	To record which staff are working on which campaigns, including the campaign	Assign staff to work on a

	manager for each campaign	campaign
6	To record which staff are assigned as staff contacts to clients	Assign a staff contact
7	To check on the status of campaigns and whether they are within budget	Check campaign budget
8	To allow creative staff to record notes of ideas for campaigns and adverts (concept notes)	Create concept note
9	To provide other staff with access to these concept notes	Browse concept notes
10	To record details of adverts, including the progress on their production	Add a new advert to a campaign. Record completion of an advert
11	To schedule the dates when adverts will be run	Add a new advert to a campaign
12	To maintain staff records for creative and administrative staff	Add a new member of staff
13	To maintain details of staff grades and the pay for those grades	Add a new staff grade. Change the rate for a staff grade
14	To record which staff are on which grade	Change the grade for a member of staff
15	To calculate the annual bonus for all staff	Calculate staff bonuses
16	To enable data about clients, campaigns, adverts and staff to be shared between offices	Not applicable
17	To allow the system to be modified to work in different languages	Not applicable
18	To restrict the ability to create or update data to authorized users in the company.	All use cases that create or update data
19	To limit planned downtime to one hour a week during the night UK time.	Not applicable

A2.3 Actors and Use Cases

Actor	Description
Accountant	The accountant works in the Accounts department and is responsible for the major resourcing issues for campaigns including staffing and related financial matters.
Campaign Manager	Either a Director or an Account Manager (job titles), who is responsible for estimating the campaign cost and agreeing it with the client. They are responsible for assigning staff to the team and supervising their work, managing the progress of the campaign, conducting any further budget negotiations and authorizing the final invoices.
Staff Contact	Member of staff who is the contact for a particular client. They provide a first point of contact for the client when the client wants to contact Agate.
Staff	Any member of staff in Agate.
Campaign Staff	Member of staff working on a particular campaign.

Figures A2.1 to A2.3 show the use cases from the first iteration, with use case descriptions in the tables. The use case diagram in Figure A2.1 has been drawn in a modelling tool, Enterprise Architect from SparxSystems, and is shown as a screenshot.

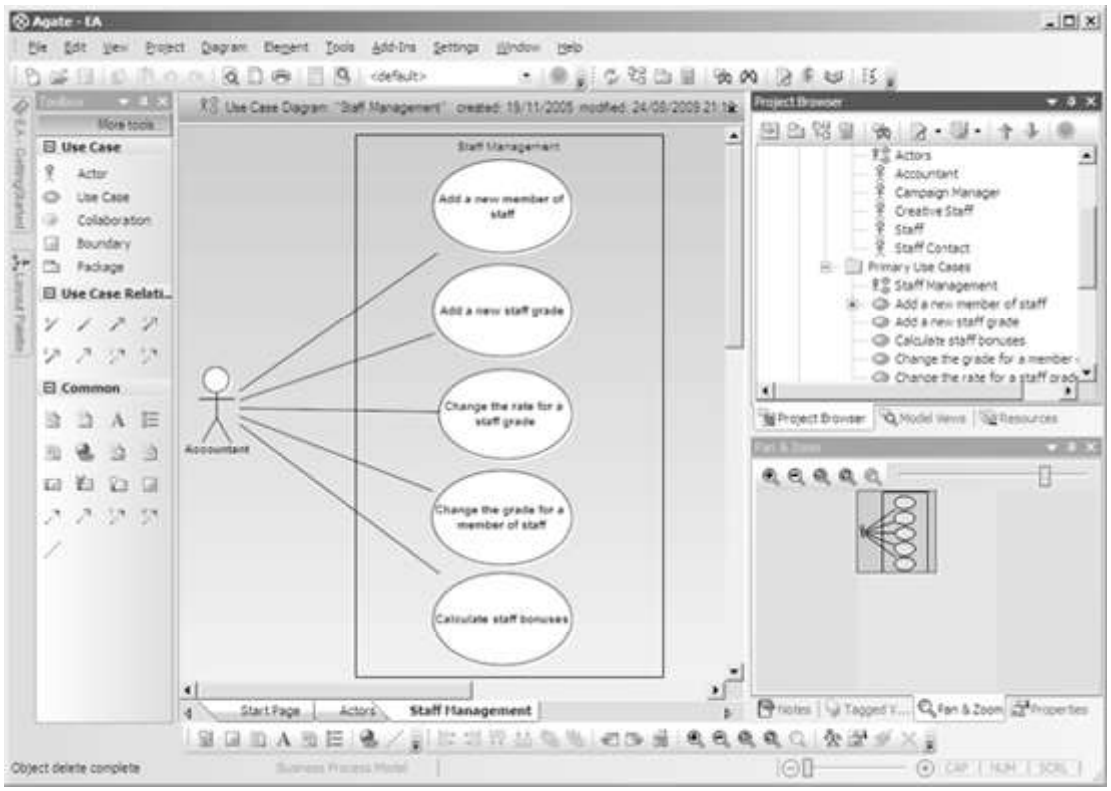


Figure A2.1 Staff Management use cases.

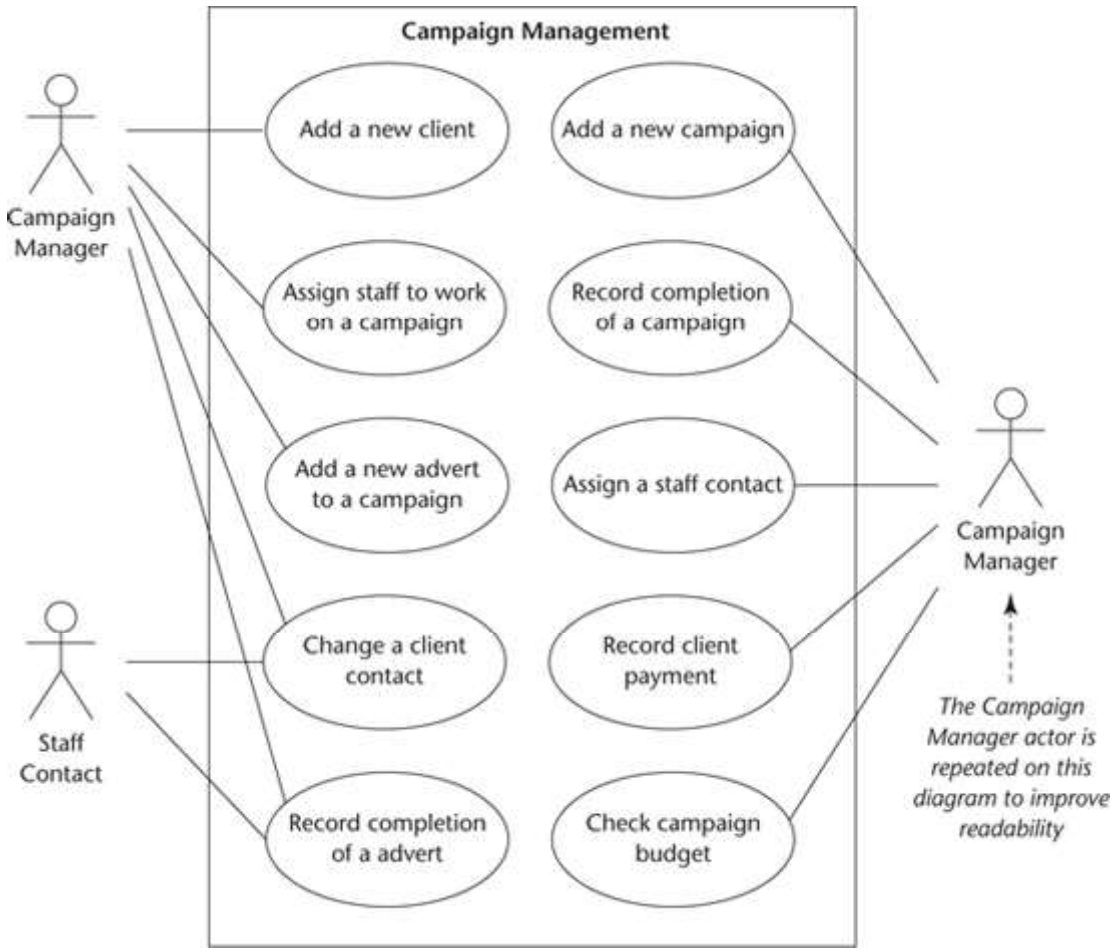


Figure A2.2 Campaign Management use cases.

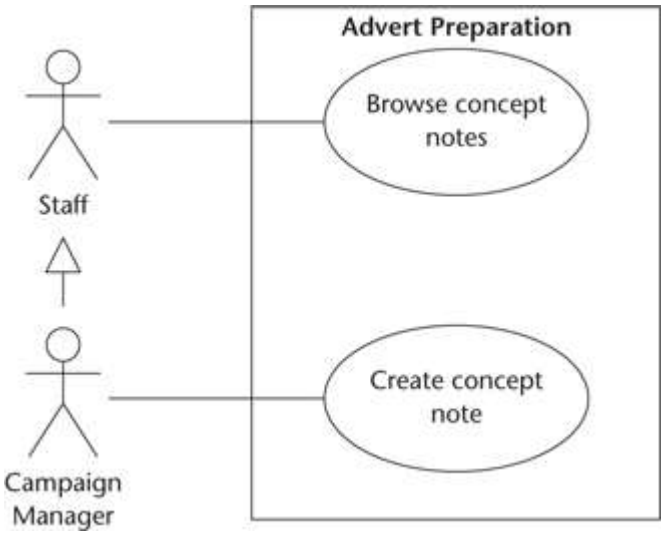


Figure A2.3 Advert Preparation use cases.

Use Case	Description
Add a new staff member	When a new member of staff joins Agate, his or her details are recorded. He or she is assigned a staff number, and the start date is entered. Start date defaults to today's date. The starting grade is entered.
Add a new staff grade	Occasionally a new grade for a member of staff must be added. The name of the grade is entered. At the same time, the rate for that grade and the rate start date are entered; the date defaults to today's date.
Change the rate for a staff grade	Annually the rates for grades are changed. The new rate for each grade is entered, and the rate start date set (no default). The old grade rate is retrieved and the rate finish date for that grade rate set to the day before the start of the new rate.
Change the grade for a staff member	When a member of staff is promoted, the new grade and the date on which they start on that grade are entered. The old staff grade is retrieved and the finish date set to the day before the start of the new grade.
Calculate staff bonuses	At the end of each month staff bonuses are calculated. This involves calculating the bonus due on each campaign a member of staff is working on. These are summed to give the total staff bonus.

Use Case	Description
Add a new client	When Agate obtains a new client, the full details of the client are entered. Typically this will be because of a new campaign, and therefore the new campaign will be added straight away.
Assign staff to work on a campaign	The campaign manager selects a particular campaign. A list of staff not already working on that campaign is displayed, and he or she selects those to be assigned to this campaign.
Add a new advert to a campaign	A campaign can consist of many adverts. Details of each advert are entered into the system with a target completion date and estimated cost.
Change a client contact	Records when the client's contact person with Agate is changed.
Record completion of an advert	The actor selects the relevant client, campaign and advert. The selected advert is then completed by setting its completion date.
Add a new campaign	When Agate gets the business for a new campaign, details of the campaign are entered, including the intended finish date and the estimated cost. The manager for that campaign is the person who enters it.

Record completion of a campaign	When a campaign is completed, the actual completion date and cost are entered. A record of completion form is printed out for the Accountant as the basis for invoicing the client.
Assign a staff contact	Clients have a member of staff assigned to them as their particular contact person.
Record client payment	When a client pays for a campaign, the payment amount is checked against the actual cost and the date paid is entered.
Check campaign budget	The campaign budget may be checked to ensure that it has not been exceeded. The current campaign cost is determined by the total cost of all the adverts and the campaign overheads.

Use Case	Description
Browse concept notes	Any member of staff may view concept notes for a campaign. The campaign must be selected first. The titles of all notes associated with that campaign will be displayed. The user will be able to select a note and view the text on screen. Having viewed one note, others can be selected and viewed.
Create concept note	A member of staff working on a campaign can create a concept note, which records ideas, concepts and themes that will be used in an advertising campaign. The note is in text form. Each note has a title. The person who created the note, the date and time are also recorded.

As part of the second iteration of use case modelling, it is suggested that all the use cases that require the user to select a client, a campaign or an advert should have include relationships with use cases called Find client, Find campaign and Find advert. An example of this is shown in Fig. A2.4.

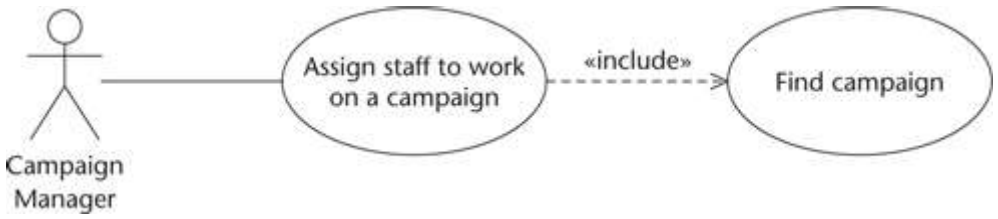


Figure A2.4 Inclusion of Find campaign use case.

In order to test out this idea, prototypes of the user interface were produced in the second iteration. The first prototypes used a separate user interface for these included use cases, as shown in Fig. A2.5.

Figure A2.5 Prototype interface for the Find campaign use case.

However, feedback from the users indicated that this approach was not acceptable. They did not want to have to keep opening extra windows to find clients, campaigns and adverts. The users expressed the view that they should be able to select these from listboxes or dropdown lists that were part of the interface for whatever use case they were in at the time.

In the third iteration of use case modelling, a set of prototypes was produced that uses listboxes. Figure A2.6 shows an example.

Figure A2.6 Prototype interface for the use case Check campaign budget.

In the third iteration, some additional functionality was identified and added to the use case diagrams. As an example of this, Fig. A2.7 shows the use case Check campaign budget extended by the use case Print campaign summary and Print campaign invoice. This additional functionality will also require a change to the prototype interface in Fig. A2.6. Two additional buttons, Print Summary and Print Invoice, need to be added to the row of buttons at the bottom of the window.

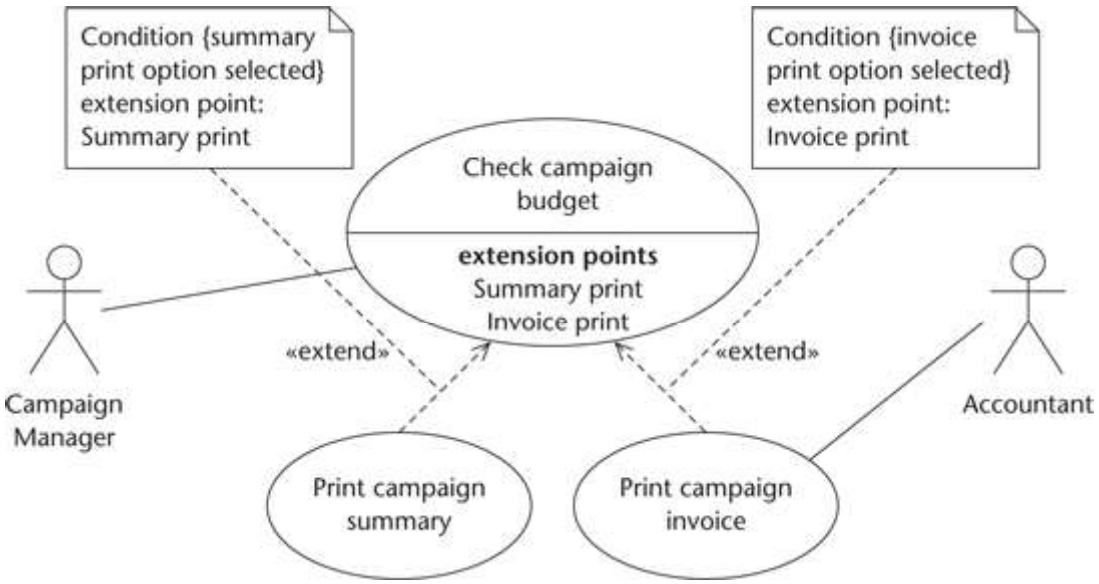


Figure A2.7 Modified use case Check campaign budget with extensions.

Also in the third iteration, the use case descriptions are elaborated to provide more detail about the interaction between the actors and the system. Two examples of these use case descriptions are provided below.

Use case description: Check campaign budget

Actor Action	System Response
1. None	2. Lists the names of all clients
3. The actor selects the client name	4. Lists the titles of all campaigns for that client
5. Selects the relevant campaign. Requests budget check	6. Displays the budget surplus for that campaign
Extensions After step 6, the campaign manager prints a campaign summary. After step 6, the campaign manager prints a campaign invoice.	

Use case description: Assign staff to work on a campaign

Actor Action	System Response
1. None	2. Displays list of client names
3. The actor selects the client name	4. Lists the titles of all campaigns for that client
5. Selects the relevant campaign	6. Displays a list of all staff members not already allocated to this campaign
7. Highlights the staff members to be assigned to this campaign. Clicks Allocate button.	8. Presents a message confirming that staff have been allocated
Alternative Courses None.	

A2.4 Glossary

A glossary of terms has been drawn up, which lists the specialist terms that apply to the domain of this project—advertising campaigns.

Term	Description
Admin Staff	Staff within Agate whose role is to provide administrative support that enables the work of the creative staff to take place, for example secretaries, accounts clerks and the office manager
Advert	An advertisement designed by Agate as part of a campaign. Adverts can be for TV, cinema, websites, newspapers, magazines, advertising hoardings, brochures or leaflets. Synonym: Advertisement
Agate	An advertising agency based in Birmingham, UK, but with offices around the world. The customer for this project
Campaign	An advertising campaign. Adverts are organized into campaigns in order to achieve a particular objective, for example a campaign to launch a new product or service, a campaign to rebrand a company or product, or a campaign to promote an existing product in order to take market share from competitors
Campaign Staff	Member of staff working on a particular campaign
Client	A customer of Agate. A company or organization that wishes to obtain the services of Agate to develop and manage an advertising campaign, and design and produce adverts for the campaign
Concept Note	A textual note about an idea for a campaign or advert. This is where creative staff record their ideas during the process of deciding the themes of campaigns and adverts. Synonym: Note

Term	Description
Creative Staff	Staff with a creative role in the company, such as designers, editors and copy-writers; those who are engaged in the work of the company to develop and manage campaigns and design and produce adverts
Grade	A job grade. Each member of staff is on a particular grade, for example 'Graphic Artist 2' or 'Copywriter 1'
Grade Rate	The rate of pay for a particular grade, for example the Grade 'Graphic Artist 2' is paid £26 170 per year in the UK from 1/1/2010 to 31/12/2010
Staff	Any member of staff in Agate. Synonyms: Staff member, member of staff

A2.5 Initial Architecture

The initial architecture of the system is based on the packages into which the use cases are grouped. These use cases have been grouped into three subsystem packages: Campaign Management, Staff Management and Advert Preparation.

Figure A2.8 shows the initial architecture of these three packages, and a package that will provide the mechanisms for the distribution of the application. At this early stage in the project, it is not clear what this will be, but something will be necessary to meet Requirement 16. At this stage the packages have names that reflect the business context rather than how they might be implemented in Java packages or C# namespaces or an equivalent structure. This will change later.

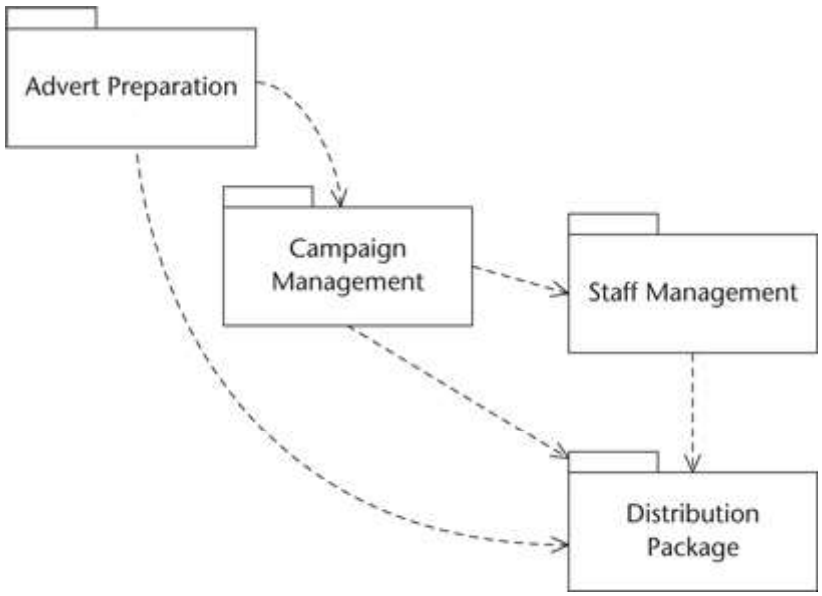


Figure A2.8 Initial package architecture.

A2.6 Activities of Requirements Modelling

In Chapter 5, we outlined the phases and activities of the iterative lifecycle, and in Chapter 6 we included an activity diagram to show the activity Requirements capture and modelling. Figure A2.9 shows the same diagram. This activity can be broken down into other activities, and these are shown in Figs A2.10, A2.11 and A2.12.

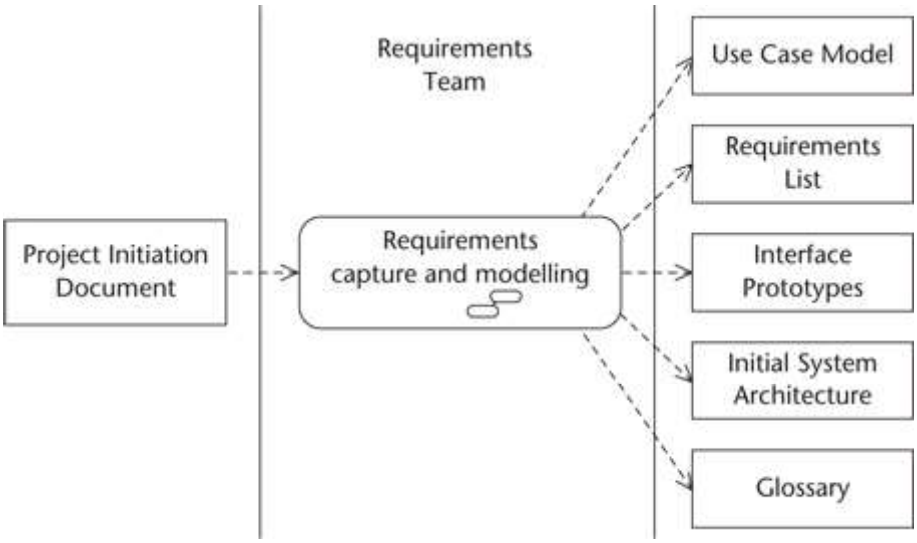


Figure A2.9 Activity diagram for Requirements capture and modelling.

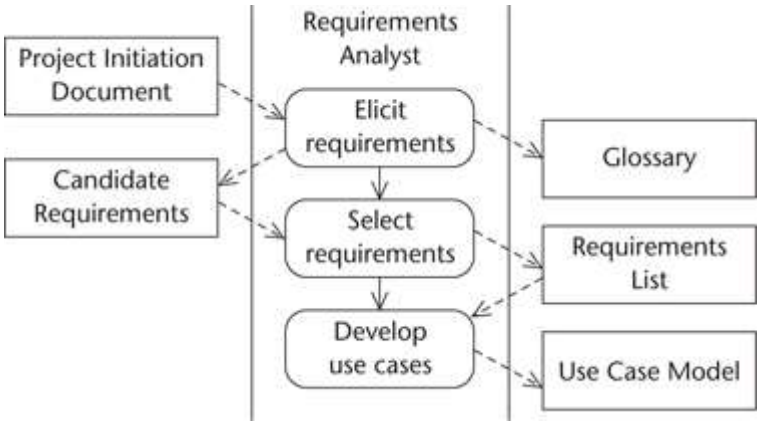


Figure A2.10 Activity diagram to show the activities involved in capturing requirements.

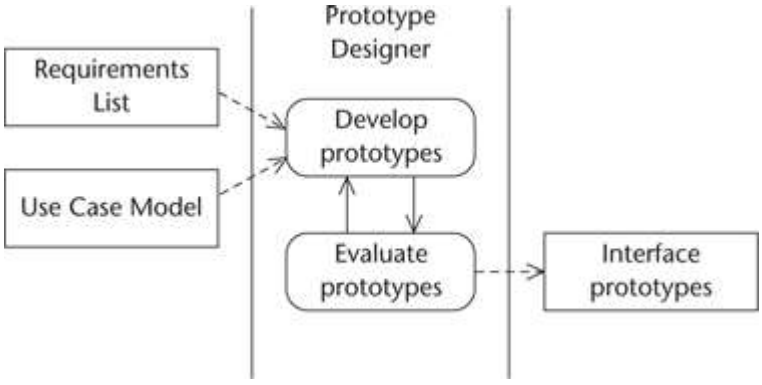


Figure A2.11 Activity diagram to show the activities involved in developing prototypes.

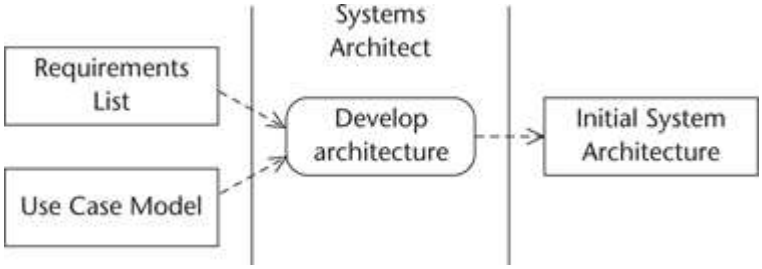
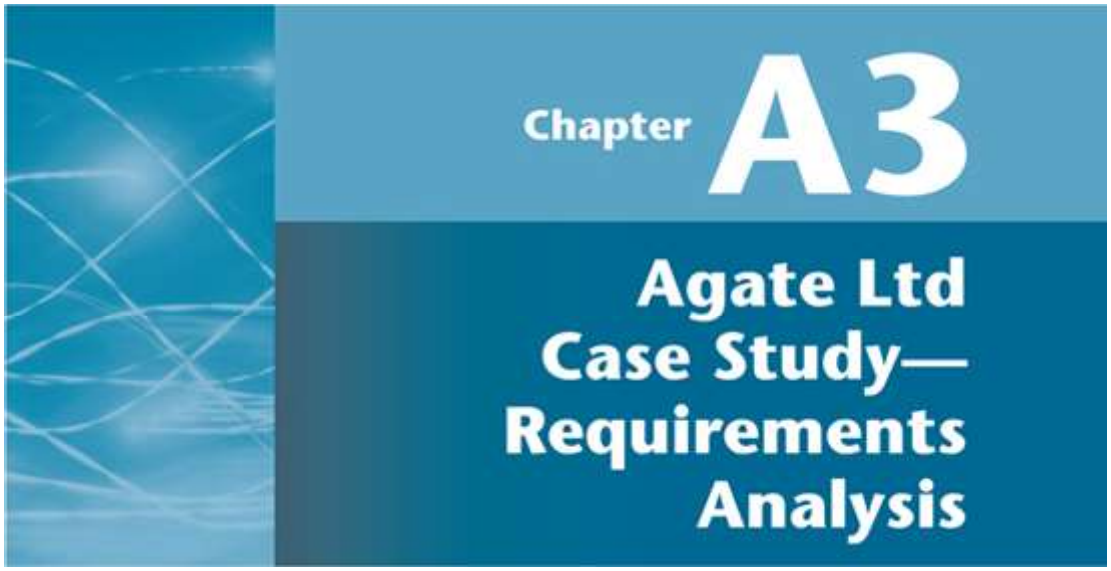


Figure A2.12 Activity diagram to show the activities in developing an initial architecture.

It is important to remember that in a project that adopts an iterative lifecycle, these activities may take place over a series of iterations. In the first iteration, the emphasis will be on requirements capture and modelling; in the second, it will shift to analysis, but some requirements capture and modelling activities may still take place. Refer back to Fig. 5.15. You may also want to look at Fig. 5.7, which illustrates the development of the use case model through successive iterations.



Agate Ltd

A3.1 Introduction

In this chapter we analyse the Requirements Model described in Chapter A2 and produce a number of use case realizations. The activities involved in use case realization are described in Chapter 7 and involve the production of the following UML diagrams:

- communication diagrams
- class diagrams that realize individual use cases
- analysis class model.

Use cases are initially analysed as collaborations and as communication diagrams. This helps to identify classes involved in their realization. After individual use case realizations have been developed, a combined analysis class model is produced from them. A more detailed analysis class diagram is also included to indicate how the model develops as the use cases are analysed.

A3.2 Use Case Realizations

The first use case analysed here is Add a new campaign (all the use cases are specified in Chapter A2). Figure A3.1 shows a collaboration that realizes the use case. Figure A3.2 shows the communication diagram, with boundary and control classes added.

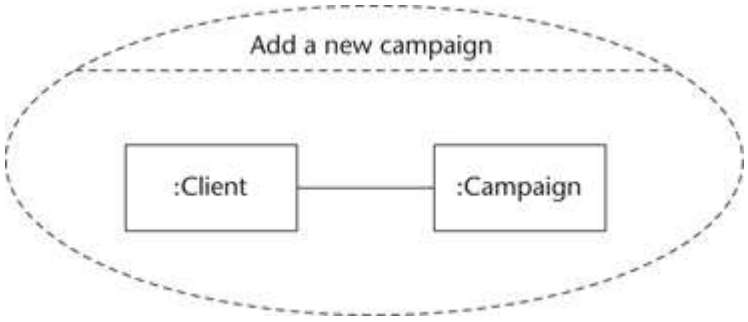


Figure A3.1 Collaboration for the use case Add a new campaign.

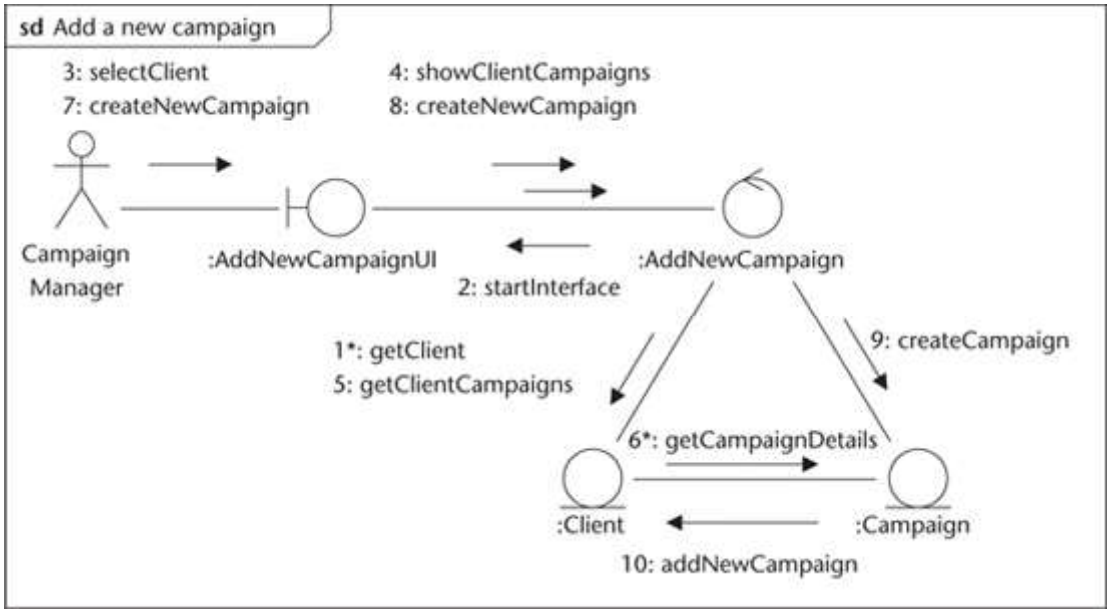


Figure A3.2 Communication diagram for the use case Add a new campaign.

Note that the initiation of the dialogue described by the use case is not modelled explicitly (apart from the inclusion of a startInterface message). Details such as this will be added later for this system, though in some projects it may be important to model them early on. The class diagram that supports this use case (and its collaboration) is shown in Fig. A3.3. Notice that the class Campaign includes only attributes that are required for the use case. The requirements analyst may identify the need for additional attributes (or functionality) while the use case is being analysed, but it is important to confirm any changes with the stakeholders. In these models we have named the constructor operation createCampaign to make it clear where in the interaction a new campaign object is created. If we were preparing a design model, the naming conventions used in object-oriented programming languages would be more appropriate.

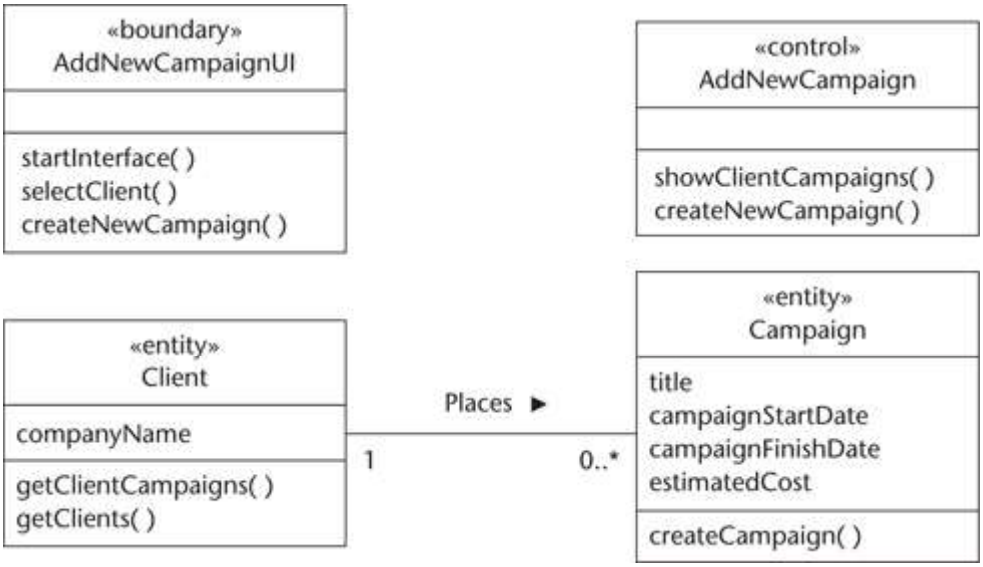


Figure A3.3 Class diagram for the use case Add a new campaign.

Figures A3.4 to A3.12 show the development of the use case realizations for the use cases Assign staff contact, Check campaign budget and Record completion of a campaign. The use case Record completion of a campaign involves the production of a completion note. The

boundary class Completed CampaignPI (we use the suffix PI to stand for printer interface) is responsible for printing the completion note.

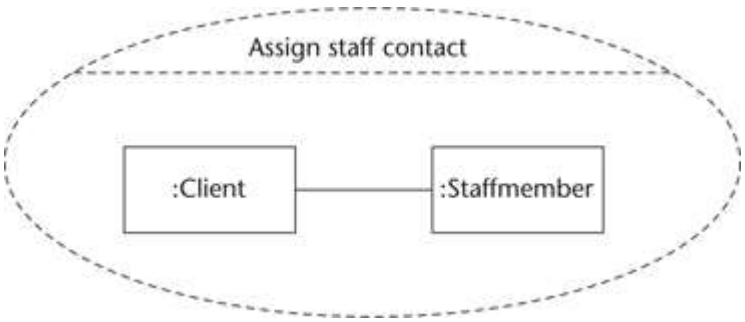


Figure A3.4 Collaboration for the use case Assign staff contact.

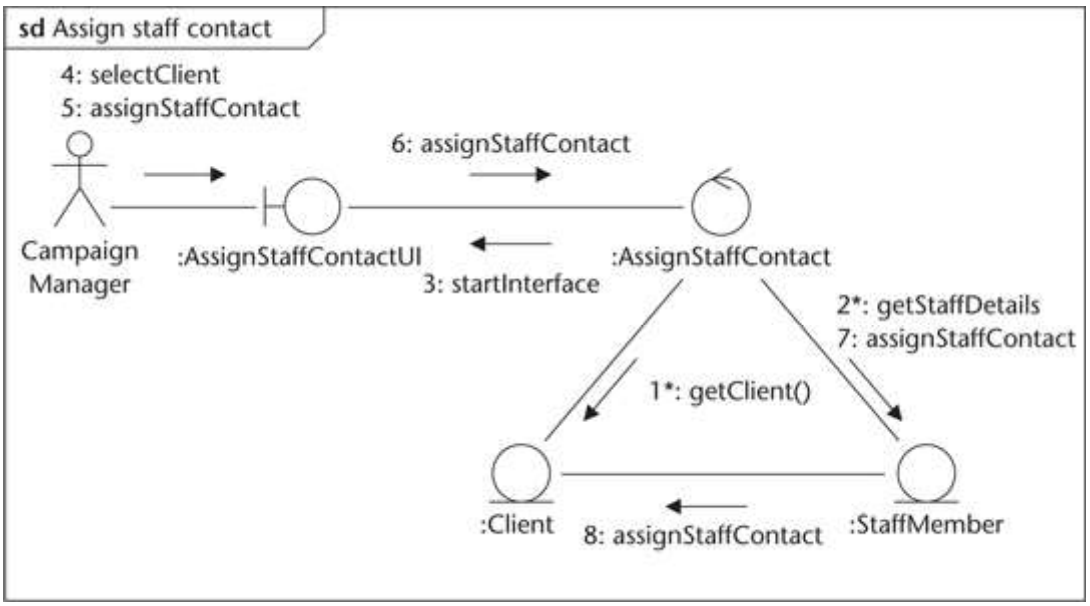


Figure A3.5 Communication diagram for the use case Assign staff contact.

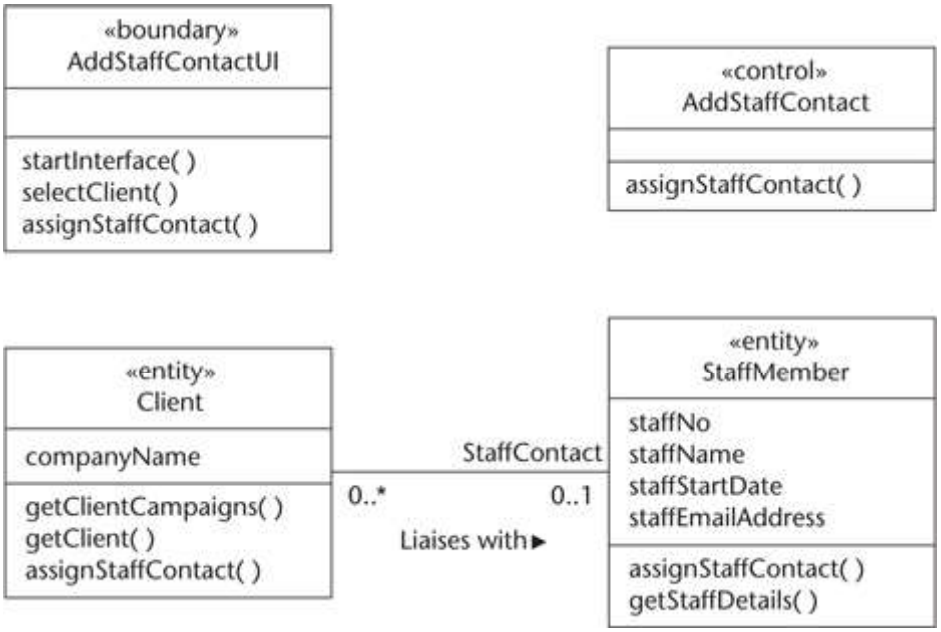


Figure A3.6 Class diagram for the use case Assign staff contact.

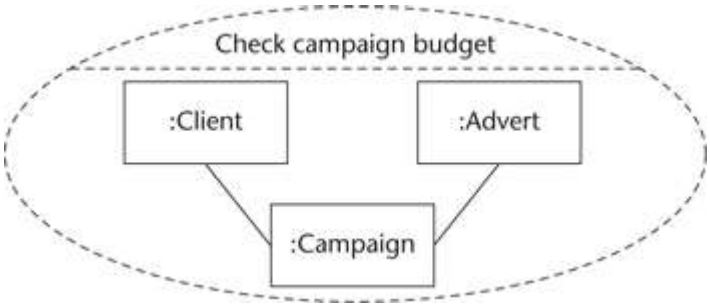


Figure A3.7 Collaboration for the use case Check campaign budget.

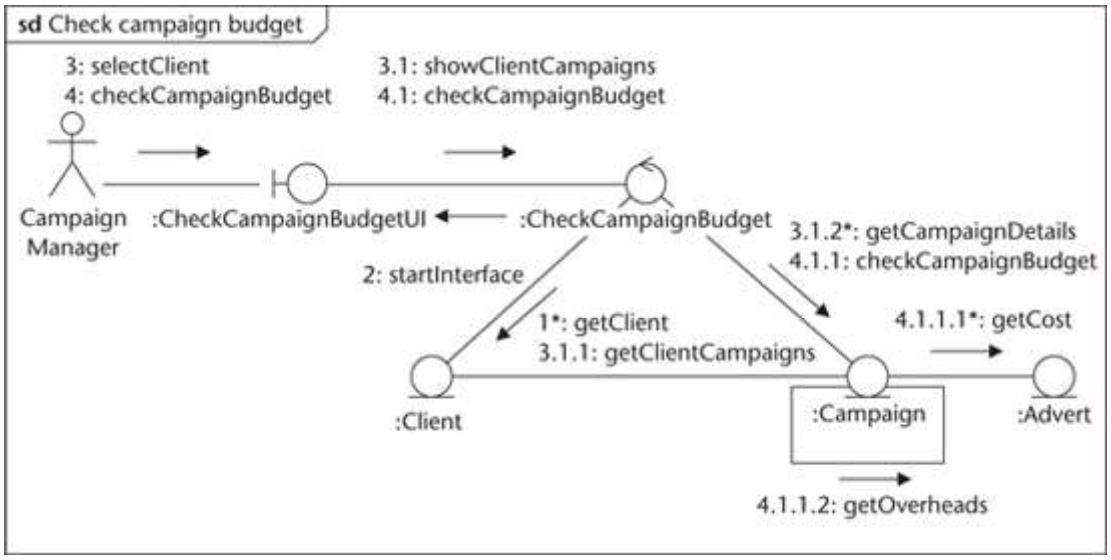


Figure A3.8 Communication diagram for the use case Check campaign budget.

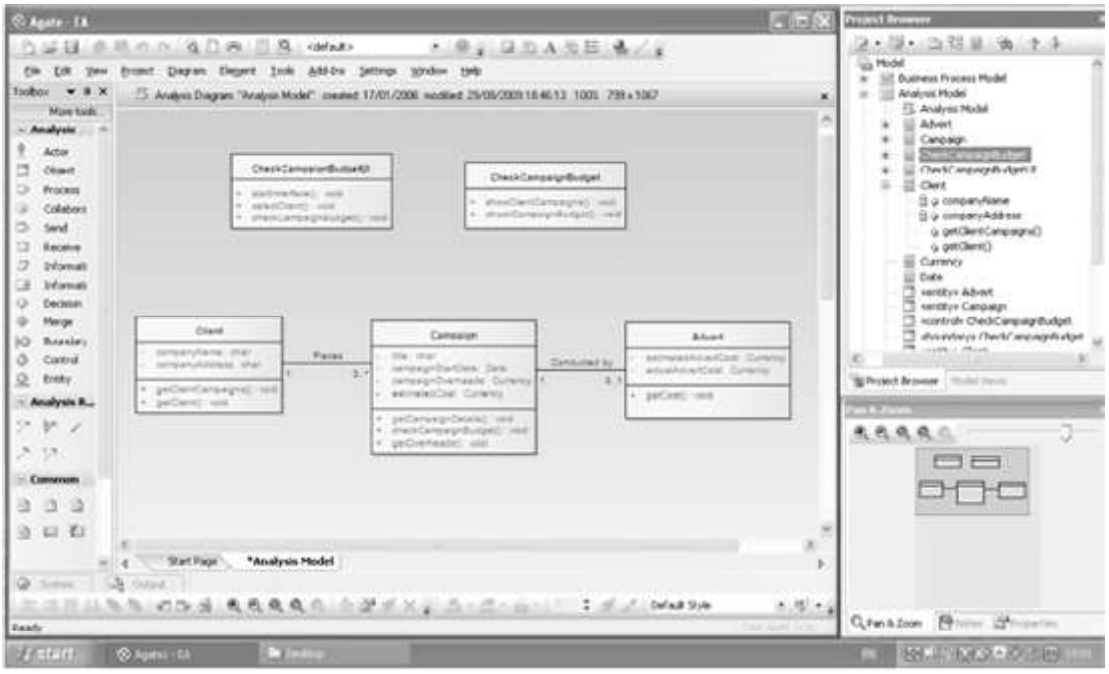


Figure A3.9 Class diagram for the use case Check campaign budget. This version has been drawn in the Enterprise Architect modelling tool.

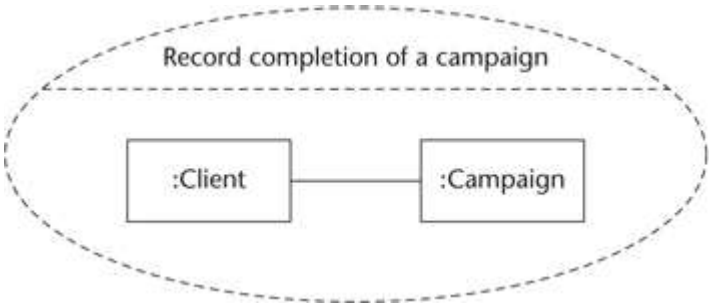


Figure A3.10 Collaboration for the use case Record completion of a campaign.

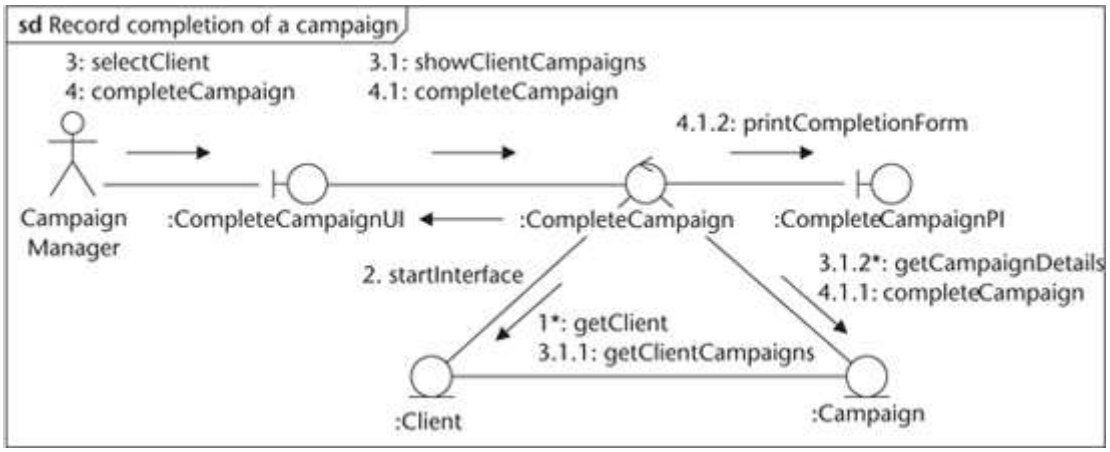


Figure A3.11 Communication diagram for the use case Record completion of campaign.

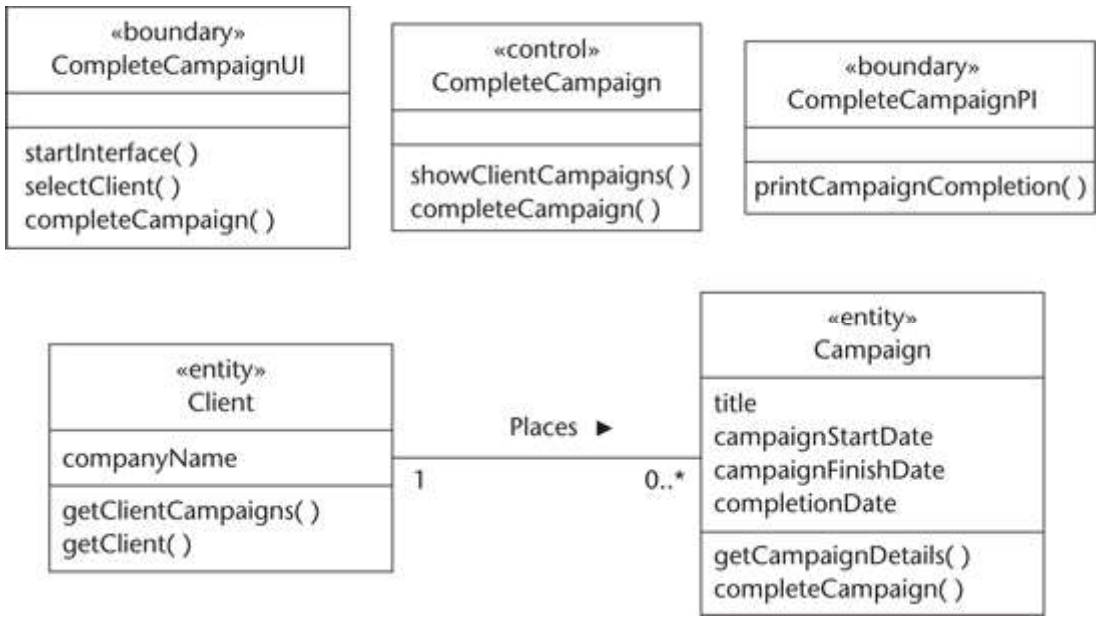


Figure A3.12 Class diagram for the use case Record completion of campaign.

A3.3 Assembling the Analysis Class Diagram

The class diagram in Fig. A3.13 has been assembled from the realizations for Add a new campaign, Assign staff contact, Check campaign budget and Record completion of a campaign.

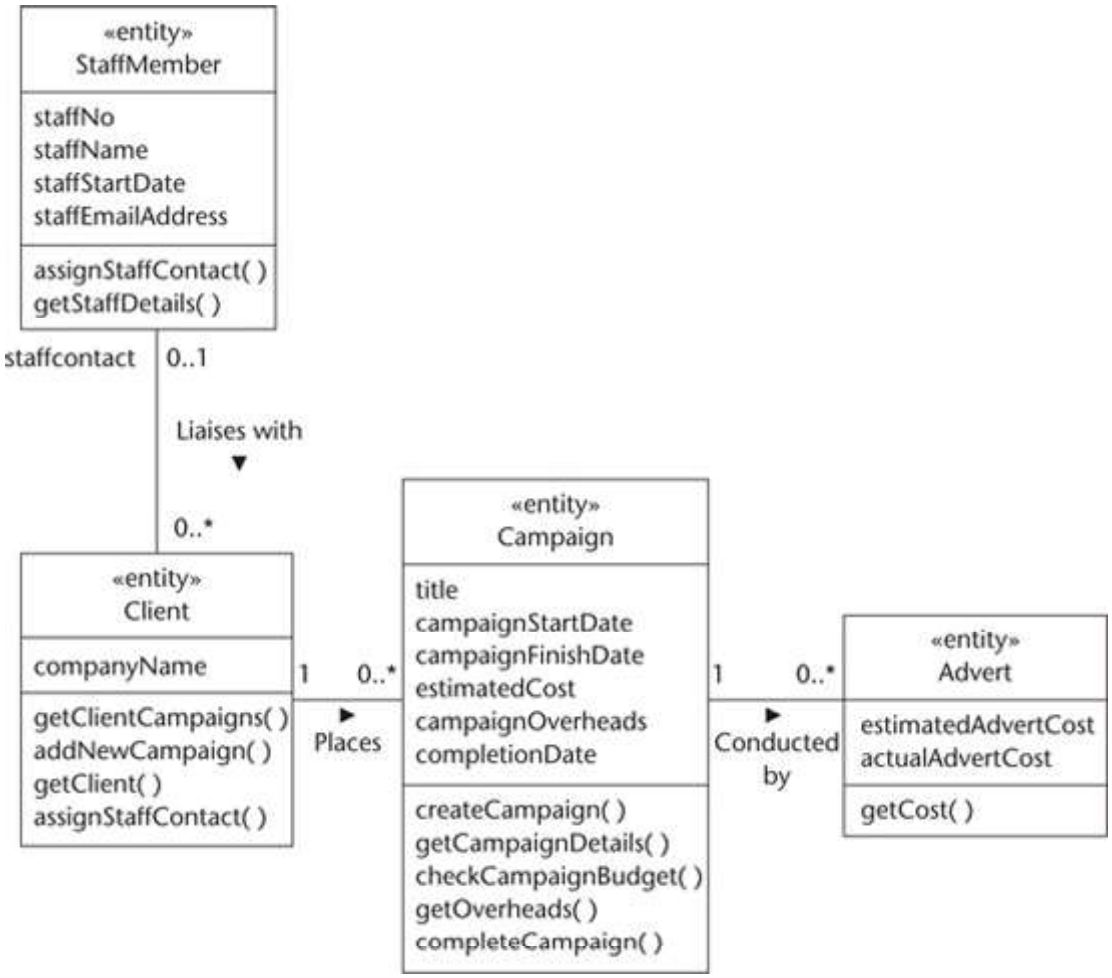


Figure A3.13 Combined class diagram for four use cases.

Figure A3.14 shows a more fully developed class diagram that includes classes, attributes, operations and associations that have been identified from the other use cases in the Campaign Management package. This illustrates how a more detailed and complete picture of the analysis model is developed as the use cases are analysed. The use cases Add a new advert to a campaign and Assign staff to work on a campaign are analysed in Chapter 7. Their realizations are shown in Figs 7.17 and 7.26 respectively.

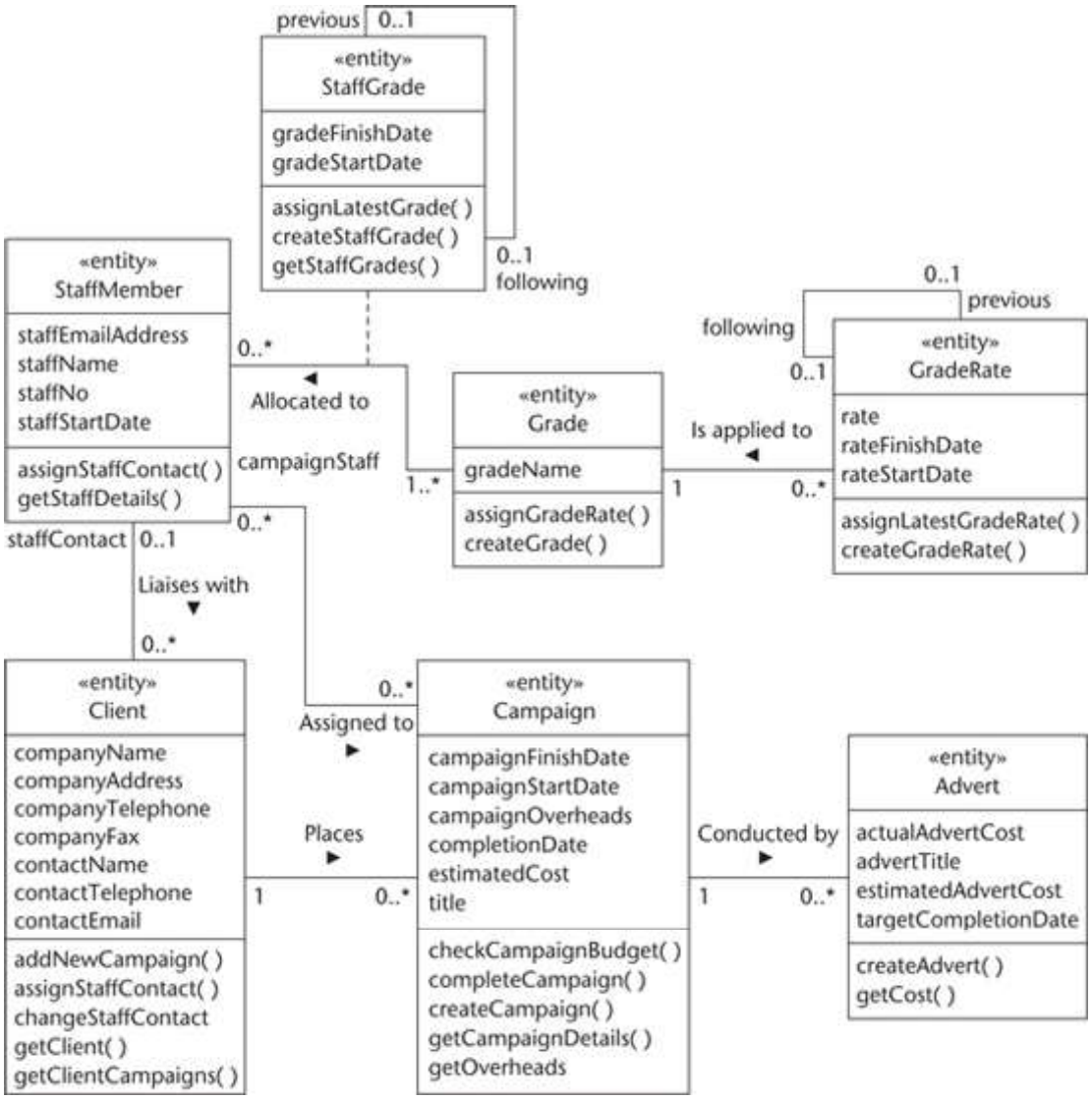


Figure A3.14 Combined class diagram after further requirements analysis.

A3.4 Activities of Requirements Analysis

Figure A3.15 shows an activity diagram that illustrates the relationship between the requirements models and the products of requirements analysis. The activity diagram in Fig. A3.16 shows the main activities involved in use case realization.

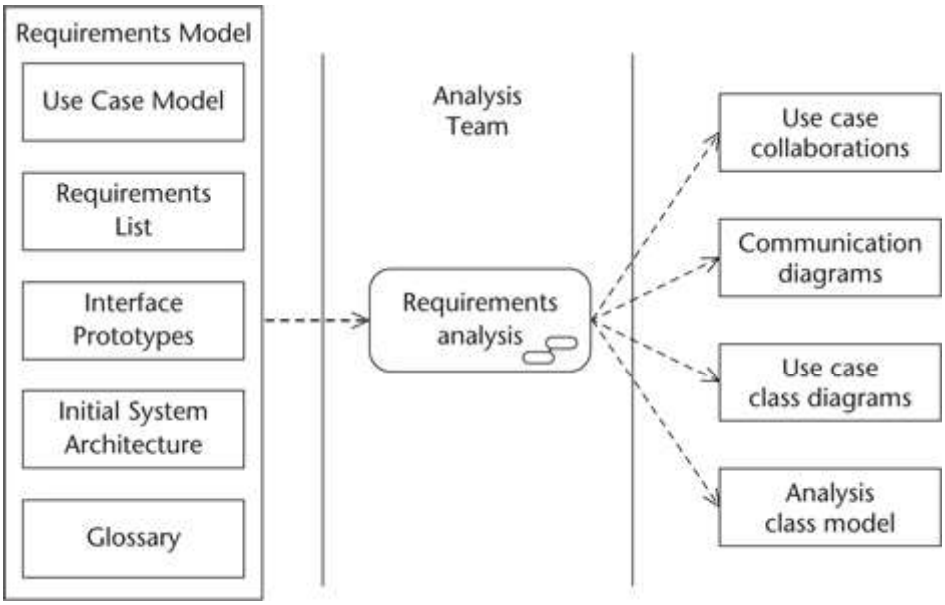


Figure A3.15 High-level activity diagram for Requirements analysis.

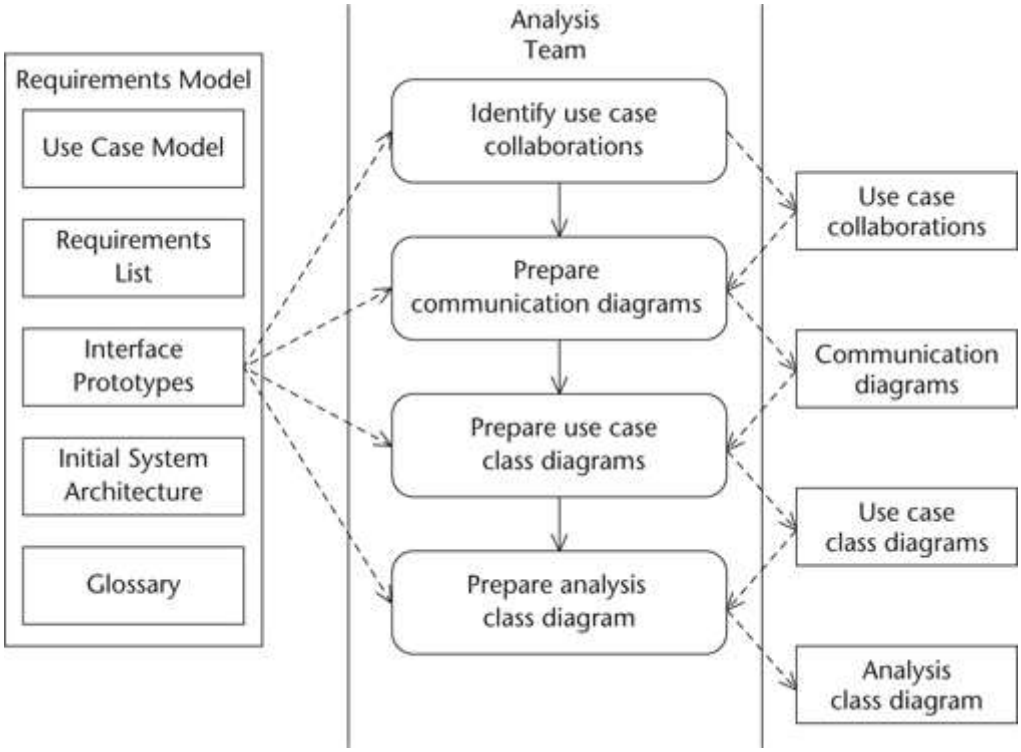
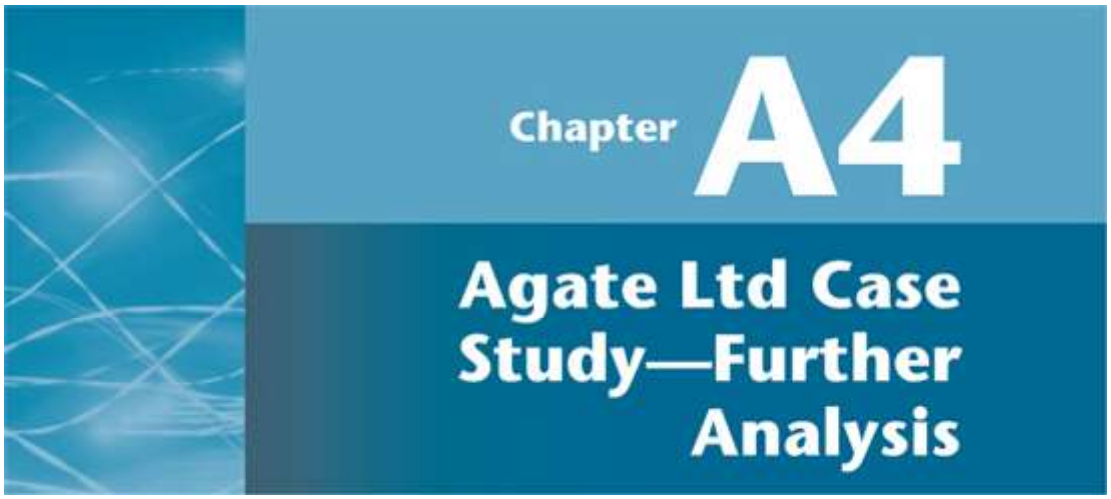


Figure A3.16 Activity diagram describing analysis use case realization.



Agate Ltd

A4.1 Introduction

In this chapter we show how the analysis model presented in Chapter A3 has been refined in a further iteration. The refinement has been carried out with two particular aims in mind.

First we aim to improve our understanding of the domain and thereby increase the general usefulness of the model in a wider context. This essentially means identifying opportunities for reuse through the elaboration of generalization, composition and aggregation structures in the class model, as described in Chapter 8.

Second, we aim to improve the level of detail of the model and also the accuracy with which it reflects user requirements. This is addressed partly through appropriate allocation of behaviour to classes, derived from the analysis of class interaction using sequence diagrams and state machines. We also seek to specify the behavioural aspects of the model in more detail through the specification of operations. The related techniques are described in Chapters 9, 10 and 11.

As a result of these activities, the analysis class model is revised to reflect our greater understanding of the domain and of the requirements.

The following sections include:

- samples of the sequence diagrams and state machines that help us to understand the behavioural aspects of the model;
- specifications for some operations that capture this behaviour and communicate it to the designers;
- a revised analysis class diagram that shows the effects of further analysis on the static structure of the model.

Together, the class diagram and operation specifications comprise an analysis class model.

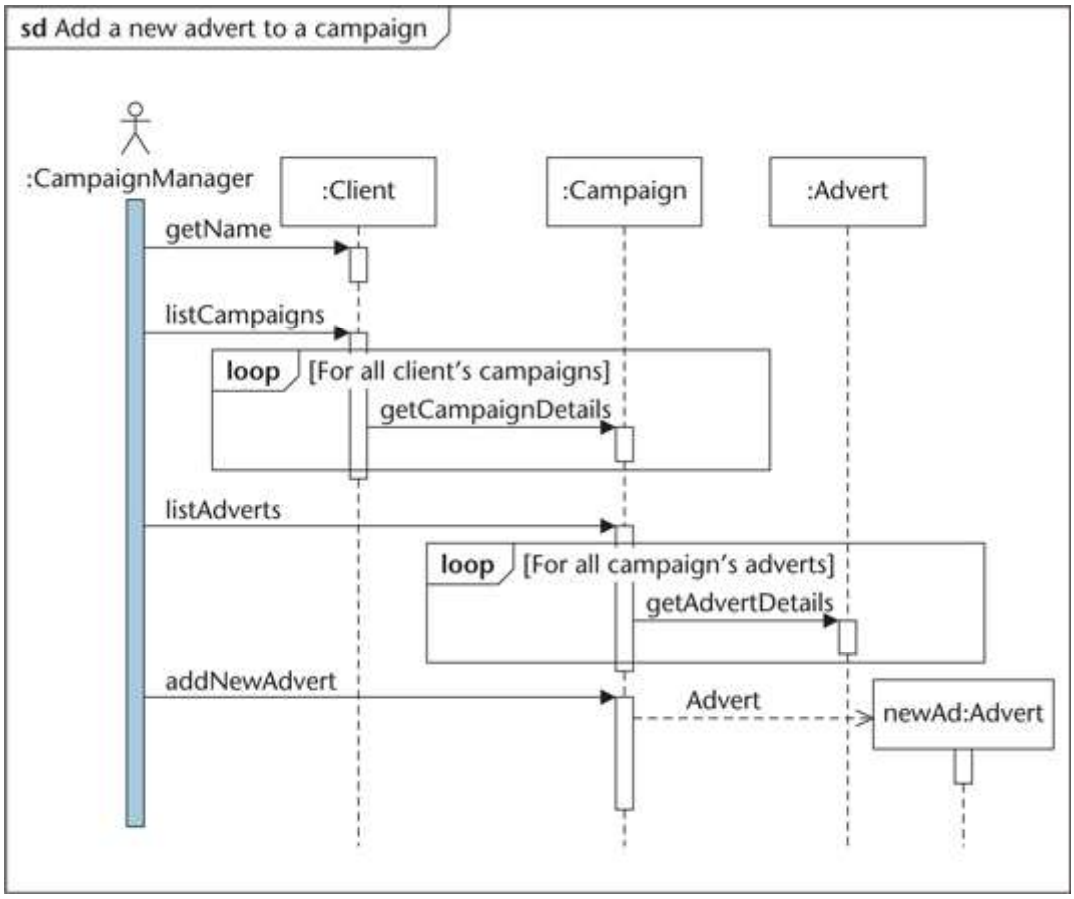


Figure A4.1 Sequence diagram for Add a new advert to a campaign.

A4.2 Sequence Diagrams

The first sequence diagram, shown in Fig. A4.1, is for the use case Add a new advert to a campaign. The second sequence diagram, shown in Fig. A4.2, is for the use case Check campaign budget. Both these sequence diagrams are discussed in some detail in Chapter 9; note that for simplicity we show here the version of Add a new advert to a campaign that does not include boundary and control classes.

Sequence diagrams help the requirements analyst to identify at a detailed level the operations that are necessary to implement the functionality of a use case. It is worth mentioning that, although at this point we are still primarily engaged in analysis—in other words, an attempt to understand the demands that this information system will fulfil—there is already a significant element of design in our models. There is no one correct sequence diagram for a given use case. Instead, there are a variety of possible sequence diagrams, each of which is relatively more or less satisfactory in terms of how well it meets the needs of the use case. The sequence diagrams illustrated here are the product of experimentation, judgement and several iterations of modelling carried out by analysts and users together.

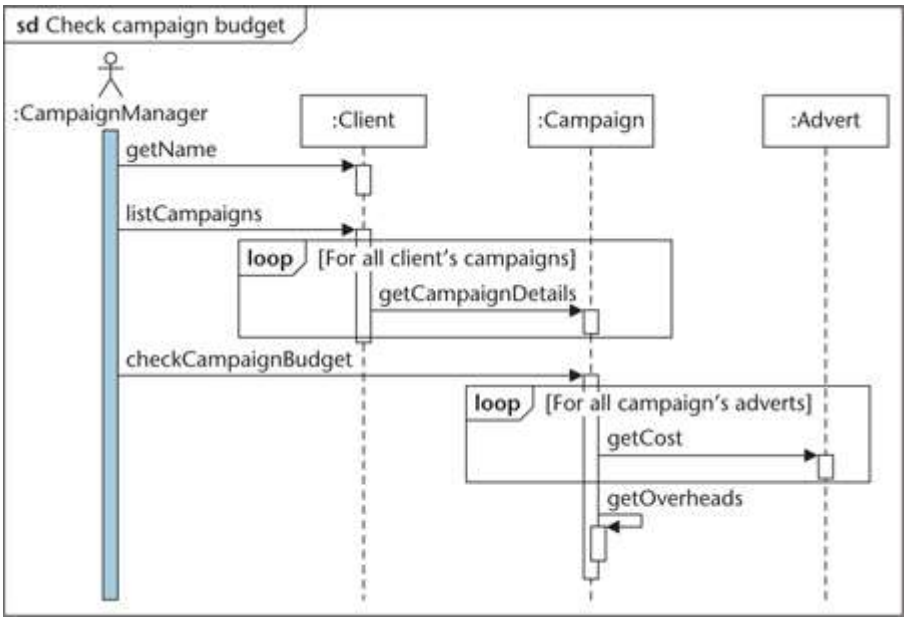


Figure A4.2 Sequence diagram for Check campaign budget.

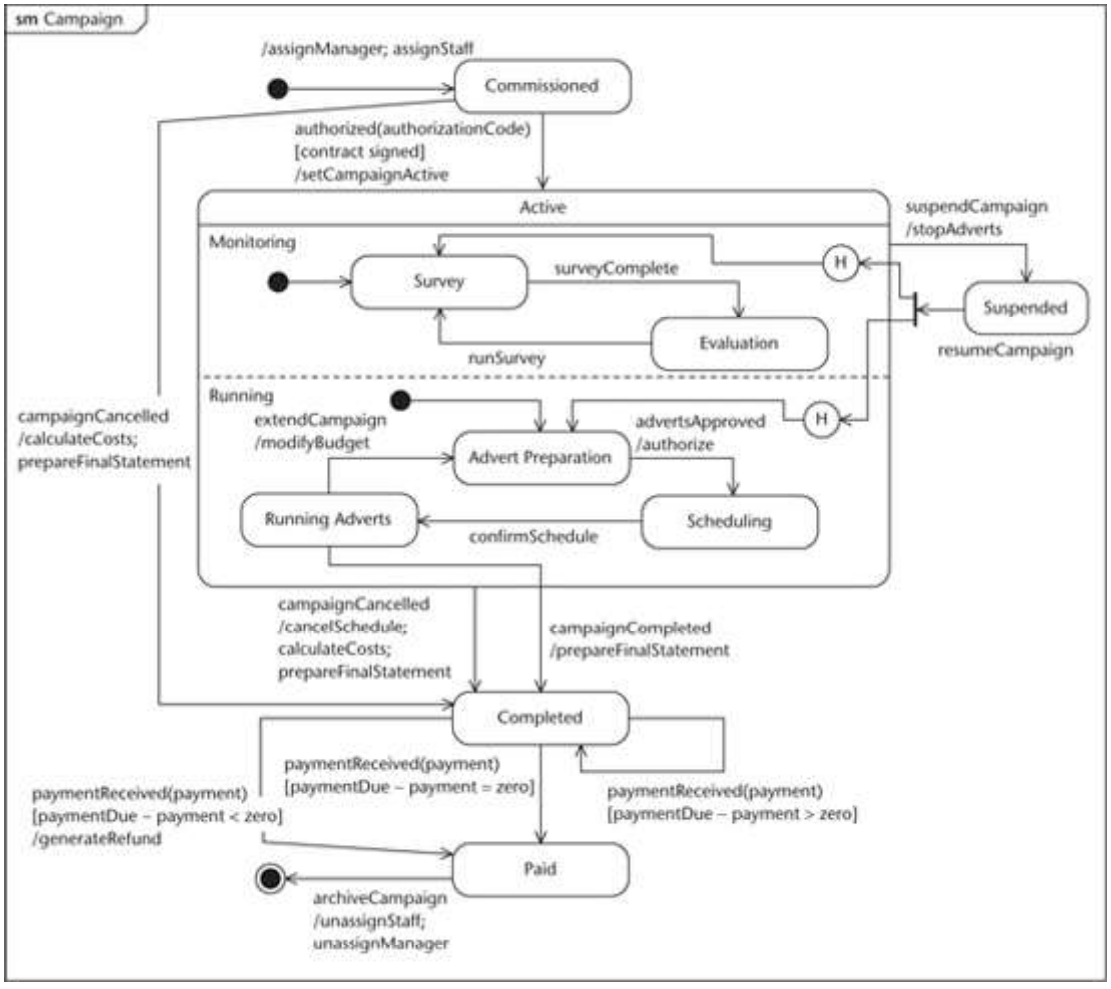


Figure A4.3 State machine for Campaign.

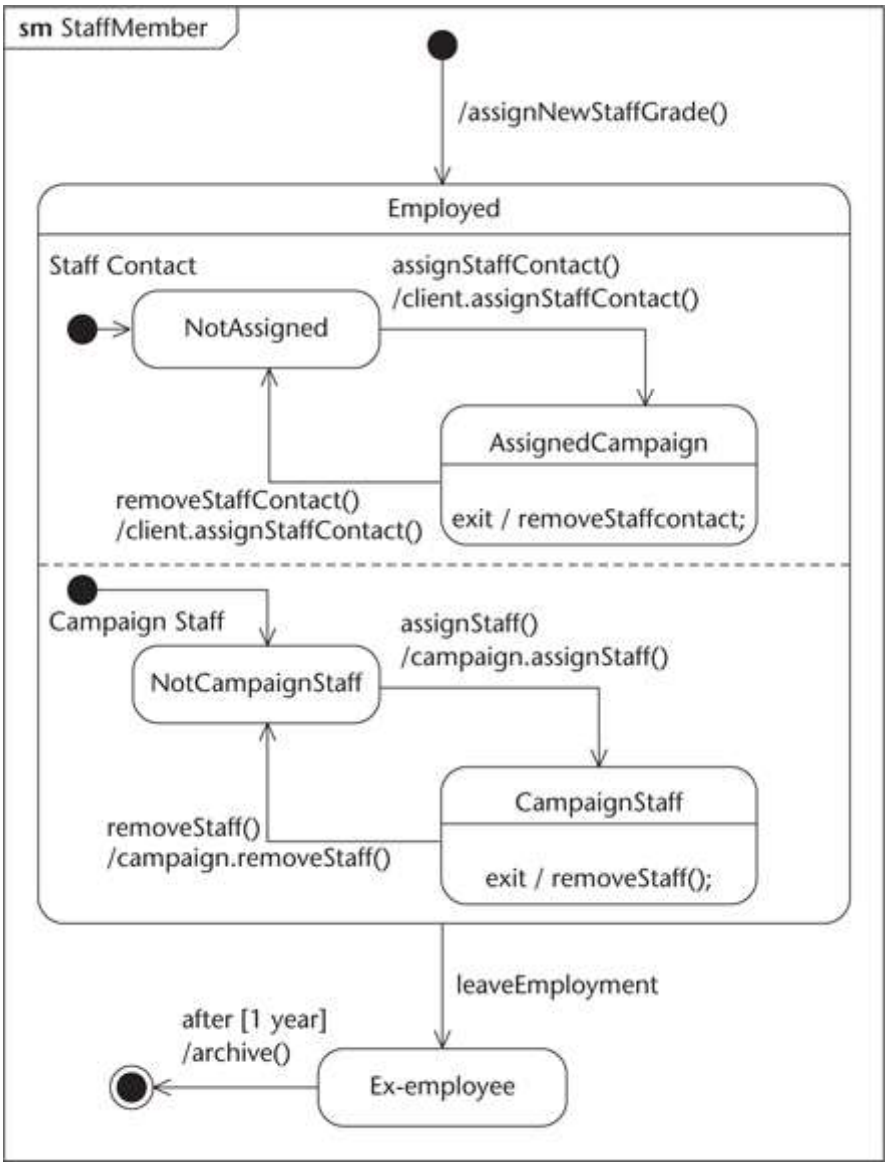


Figure A4.4 Initial state machine for StaffMember.

A4.3 State Machines

In this section we present the final state machine for Campaign (Fig. A4.3), which has already been discussed at some length in Chapter 11, and an initial state machine for StaffMember (Fig. A4.4), which is presented here for the first time. These represent the behaviour of objects of significant classes in the Campaign Management and Staff Management analysis packages, respectively.

In conjunction with sequence diagrams, state machines help to identify the operations that are required and to allocate those operations to appropriate classes. All operations shown on sequence diagrams and state machines are added to the relevant class definitions. Each operation must also in due course be specified, and it is to this that we turn in the next section.

A4.4 Operation Specifications

The operation specifications given below define all operations identified for the sequence diagram Check campaign budget, which is shown above in Fig. A4.2.

Note that in all cases the logic of the operation is very simple; for some it consists of little more than returning the value of an attribute. Each operation, and, indeed, each object, has responsibility for only a small part of the processing required to realize the use case.

By reading the operation specifications in conjunction with the sequence diagram, it is easy to see how the Client, Campaign and Advert objects collaborate to realize this use case.

This view of collaborating objects is simplified to some extent, in that it does not include control and boundary objects and their operations. However, operations in these objects are no more complex than those shown below, since their primary role is simply to call and co-ordinate operations on the entity objects.

Context: Campaign

Operation specification: checkCampaignBudget()

Operation intent: return campaign budget and actual costs.

Operation signature: Campaign::checkCampaignBudget()
budgetCostDifference:Money

Logic description (pre- and post-conditions):

pre: self->exists()

post: result = self.originalBudget-self.estimatedCost **and** self.estimatedCost = self.adverts.estimatedCost->sum()

Other operations called: Advert.getCost(), self.getOverheads()

Events transmitted to other objects: none

Attributes set: none

Response to exceptions: none defined

Non-functional requirements: none defined

Operation specification: getCampaignDetails()

Operation intent: return the title and budget of a campaign.

Operation signature: Campaign::getCampaignDetails() title:String, campaignBudget:Money

Logic description (pre- and post-conditions):

pre: self->exists()

post: result = self.title, self.estimatedCost

Other operations called: none

Events transmitted to other objects: none

Attributes set: none

Response to exceptions: none defined

Non-functional requirements: none defined

Operation specification: getOverheads()

Operation intent: calculate the total overhead cost for a campaign.

Operation signature: Campaign::getOverheads() campaignOverheads:Money

Logic description (pre- and post-conditions):

pre: self->exists()

post: result = self.campaignOverheads

Other operations called: none

Events transmitted to other objects: none

Attributes set: none

Response to exceptions: none defined

Non-functional requirements: none defined

Context: Client

Operation specification: getName()

Operation intent: return the client name.

Operation signature: Client::getName()name:String

Logic description (pre- and post-conditions):

pre: self->exists

post: result = self.name

Other operations called: none

Events transmitted to other objects: none

Attributes set: none

Response to exceptions: none defined

Non-functional requirements: none defined

Operation specification: listCampaigns()

Operation intent: return a list of campaigns for a client.

Operation signature: Client::listCampaigns()titles:String[]

Logic description (pre- and post-conditions):

pre: self->exists

post: result = self.campaign->collect(campaign.title)

Other operations called: Campaign.getCampaignDetails

Events transmitted to other objects: none

Attributes set: none

Response to exceptions: none defined

Non-functional requirements: none defined

Context: Advert

Operation specification: getCost()

Operation intent: return the actual cost for an advert.

Operation signature: Advert::getCost()
actualAdvertCost:Money

Logic description (pre- and post-conditions):

pre: self->exists()

post: result = self.actualAdvertCost

Other operations called: none

Events transmitted to other objects: none

Attributes set: none

Response to exceptions: none defined

Non-functional requirements: none defined

A4.5 Further Refinement of the Class Diagram

Figure A4.5 shows the revised analysis class diagram, after inheritance and aggregation structures have been added. For reasons of space, all attributes and operations have been

suppressed from this view.

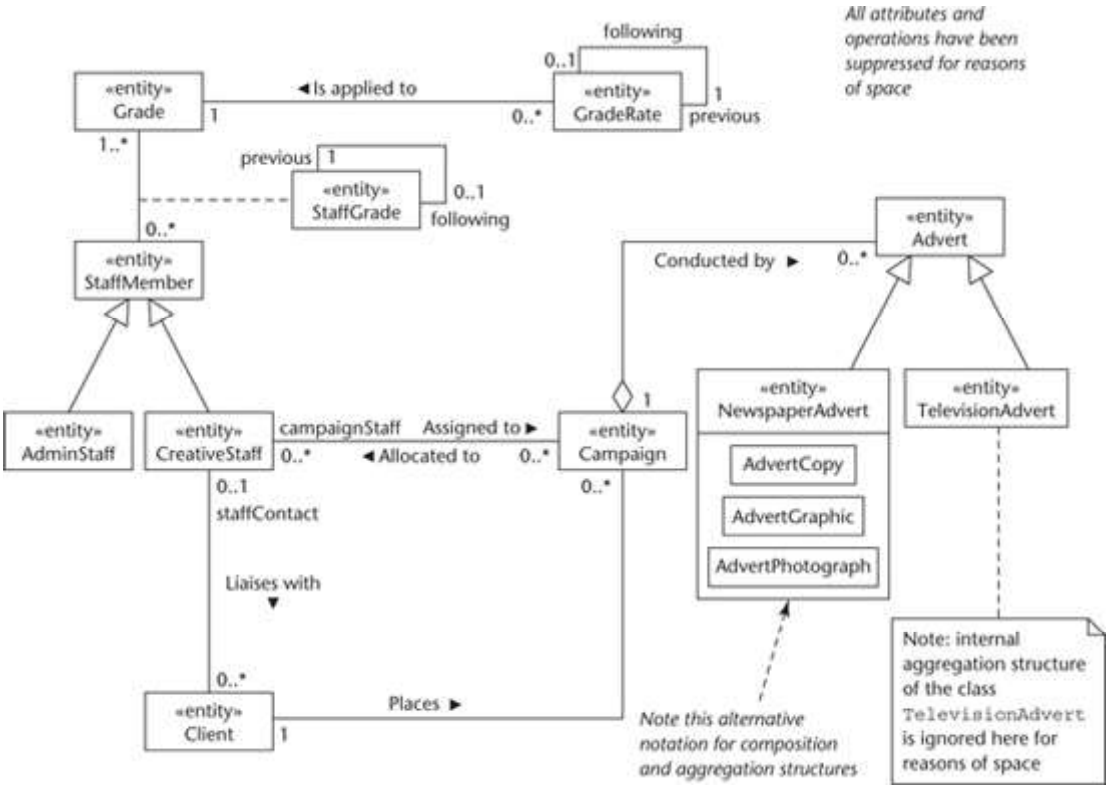


Figure A4.5 Revised analysis class diagram with generalization and aggregation structures.

Figure A4.6 shows an excerpt from the analysis class diagram, detailing the generalization and aggregation structure for Advert with attributes and operations visible. This partial diagram reflects a further iteration of investigation and requirements modelling, which revealed that there is a requirement to keep track of the various elements used to create an advertisement. This is because photographs, music clips and so on can often be used for more than one advertisement in a campaign, and it has been a problem to identify and retrieve these elements when they are needed.

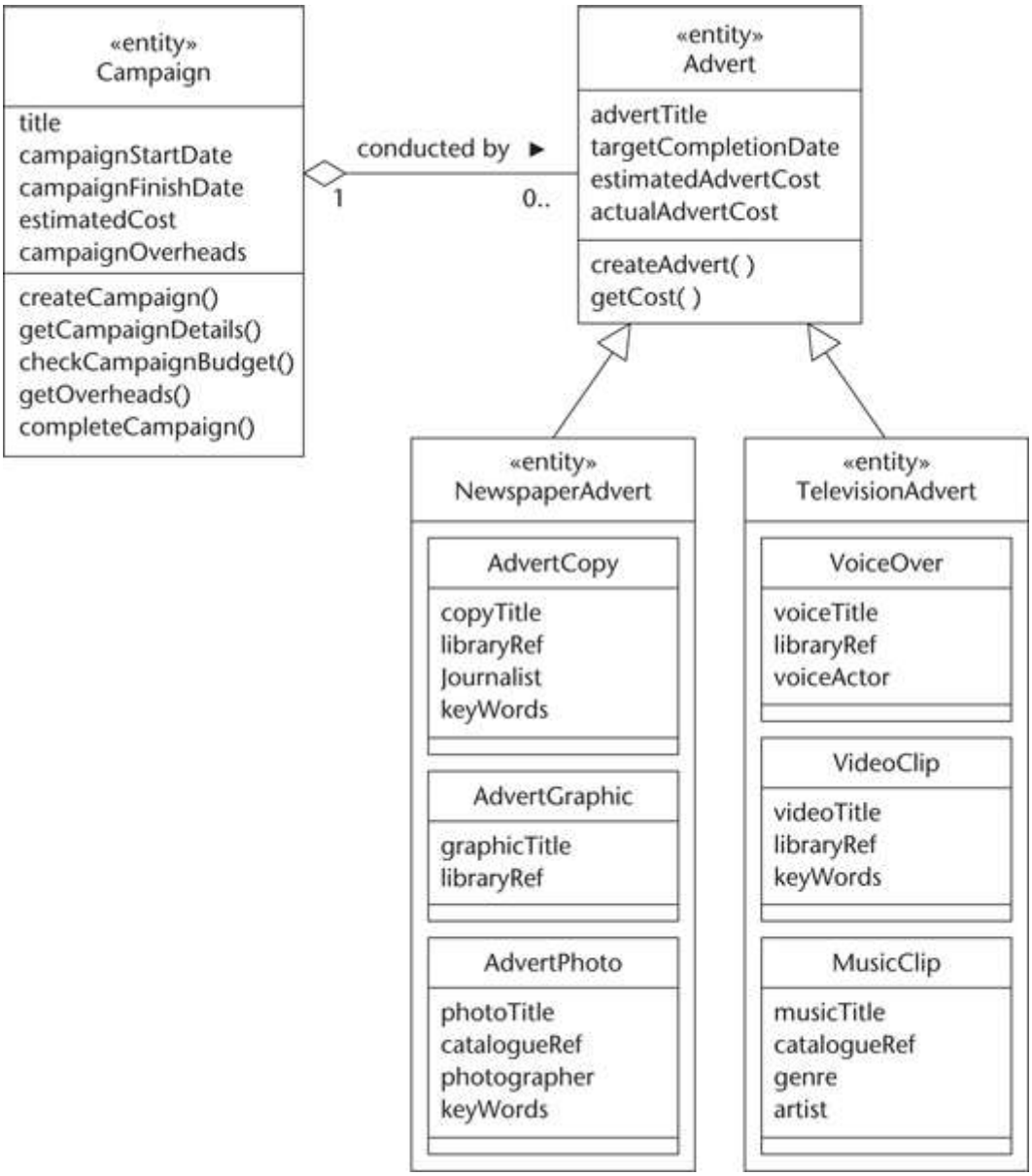


Figure A4.6 Generalization and aggregation structure for Advert.

A4.6 Further Activities of Requirements Analysis

Figure A4.7 shows an activity diagram that illustrates the relationship between the products of the analysis model before and after this iteration of analysis. Some details are worth highlighting.

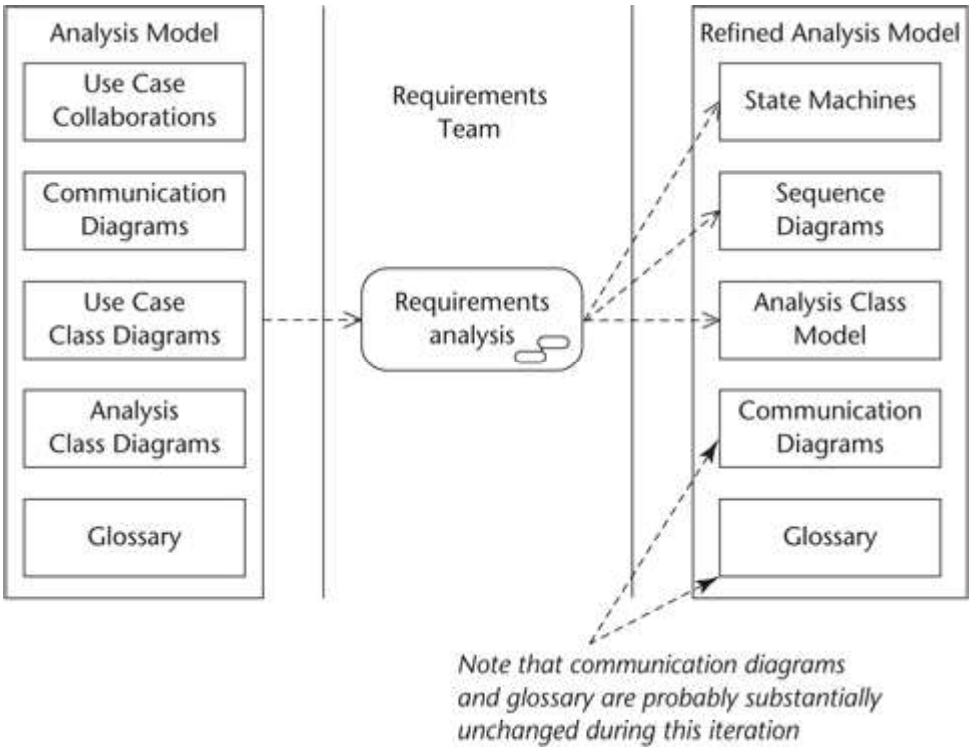
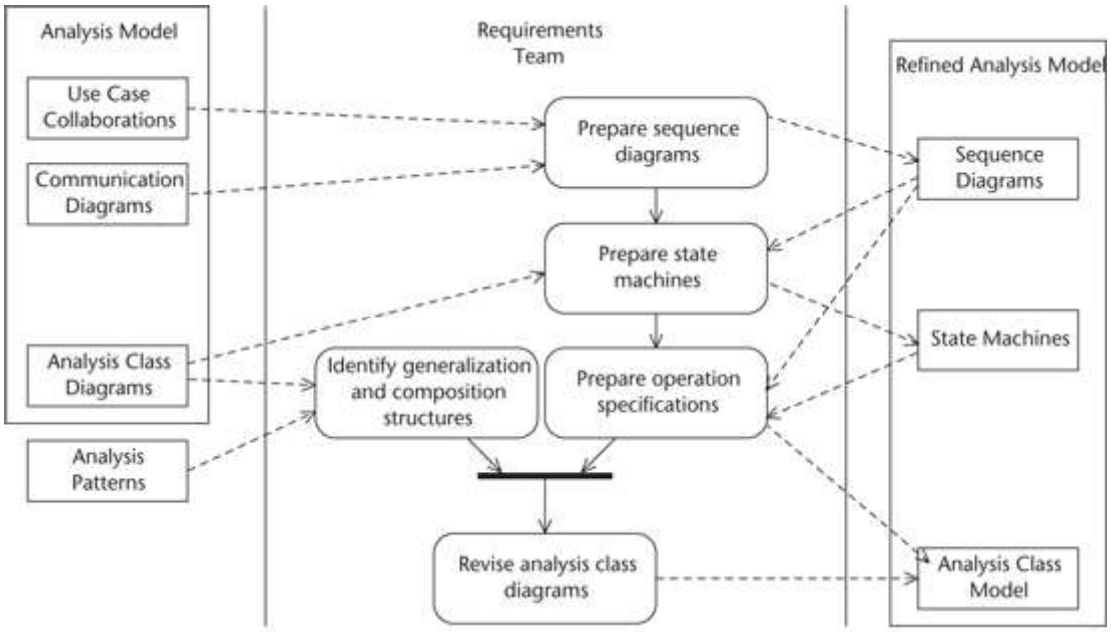


Figure A4.7 High-level activity diagram showing how elements of the analysis model are created or updated during this iteration of analysis.

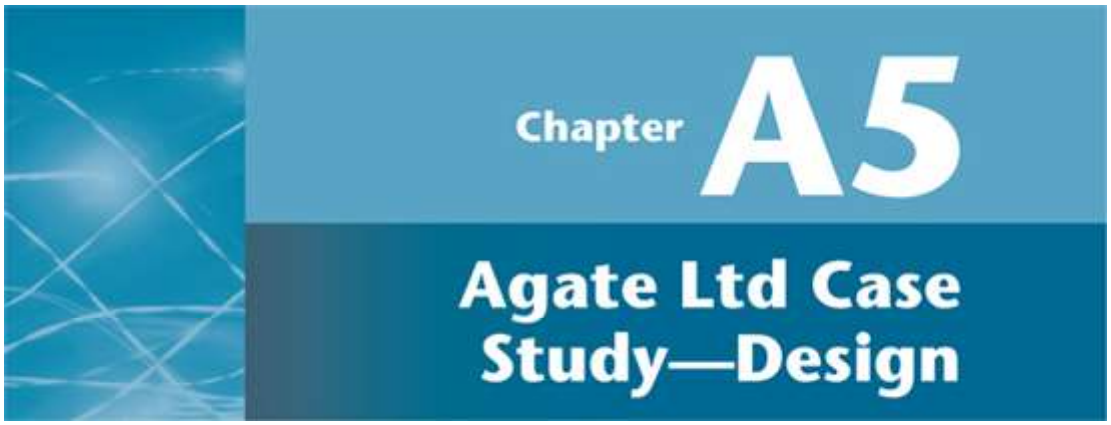
- The analysis class model now includes some detailed class definition. In particular, all operations should be specified at least in outline.
- Some parts of the analysis model may be substantially unchanged during this iteration, for example the communication diagrams and the glossary. Although this is not necessarily the case, we have shown these as unaffected in Fig. A4.7.
- As a result of the operation specification activity, many attributes may also have been specified in more detail. Some, particularly those that are required to provide parameters to operations in other classes, will certainly now be typed. We have not shown this yet, since the typing of attributes is essentially a design activity. But in practice, some design decisions are made in parallel with the more detailed analysis that we describe in this chapter.

Figure A4.8 shows a more detailed view of the activities that are carried out and the products directly used or affected during this iteration. In this diagram, we have tried to suggest a sensible outline sequence for carrying out the various activities. However, it should be noted that this is no more than a guide, and is certainly not meant to be prescriptive. An iterative approach should always be followed that is sensitive to the needs of the project, to the skill of the developers and to the often haphazard manner in which understanding grows during the modelling and analysis of requirements.



For clarity, we have detailed only those activities and products that are most directly involved in this iteration. Note also that the flow of activities is indicative and is not intended to be prescriptive.

Figure A4.8 The activities that are carried out and the products directly used or affected during this iteration of analysis.



Agate Ltd

A5.1 Introduction

In this chapter we show how part of the analysis model presented in Chapter A4 has been modified by the activities of design. The design activities have been concerned with finalizing the software architecture, designing the entity classes, their attributes, operations and associations, designing the boundary classes and the human–computer interaction, designing the mechanisms used for data storage, and designing the control classes. These activities have been explained in Chapters 12 to 18.

The following sections include:

- package diagrams to illustrate the overall software architecture
- class diagrams to illustrate the classes in the design model
- sequence diagrams to illustrate the interaction between instances of classes
- a state machine for the control of the user interface.

A5.2 Architecture

The architecture of the system (shown in Fig. A5.1) has been designed to use Java Remote Method Invocation (RMI) for communication between the client machines and the server.[1] Control classes have been split into two layers. First, there are the control classes that reside on the client machines (in the package Agate Control Client) and manage the interaction between users and the boundary classes. These control classes are essentially those that were designed in Chapter 17. Second, there are control classes that reside on the server. These control classes handle the interaction between the business logic of the application and the entity classes (and the associated data management classes). This helps to decouple the layers: the only communication between the clients and the server will be the communication between the client and server control classes, using RMI.

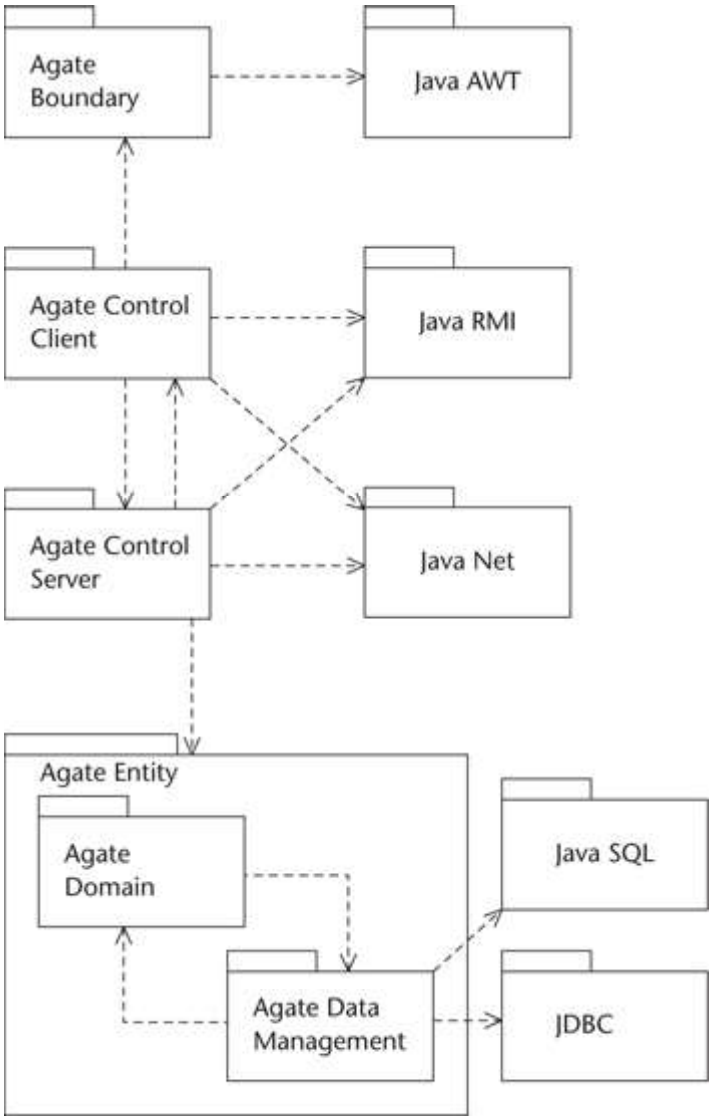


Figure A5.1 Package diagram for software architecture.

Not all control classes will have versions on both the clients and the server. For example, the ListClients and ListCampaigns classes in Figs 17.36 and 17.37 could just exist on the server, where they will have more immediate access to the entity and data management classes. One consequence of this will be visible in the sequence diagrams, where these two classes will no longer be passed references to the boundary class as a parameter, but will return their results to the control class on the client machine, which will set the values in the boundary class. This is shown in Figs A5.11 and A5.12.

On the server, we are using JDBC, and we will map the classes to relational database tables. A design based on the Broker pattern will be used to handle this.

A5.3 Sample Use Case

For the purpose of this case study chapter we are going to present the design of one use case Check campaign budget, for which the boundary and control classes were designed in Chapter 17.

Figure A5.2 shows the design of the user interface for this use case. In the first iteration, we are not concerned with adding the extensions to the use case that handle printing of the campaign summary and campaign invoice.

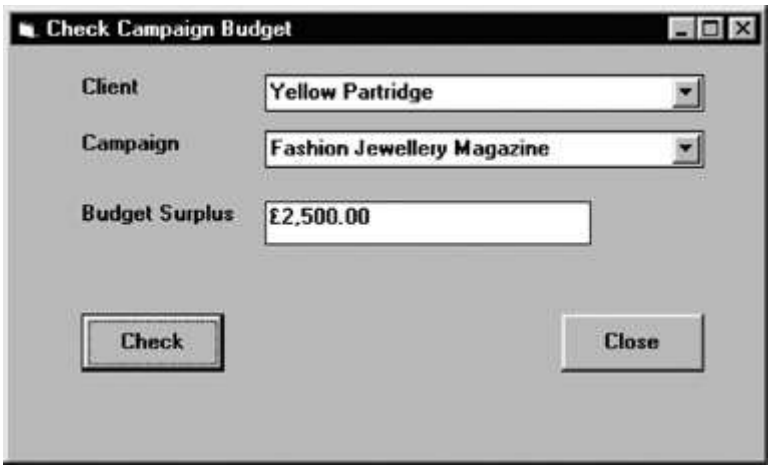


Figure A5.2 Prototype user interface for Check campaign budget.

A5.4 Class Diagrams

The packages on the architecture diagram have been named in a way that will allow us to use the Java package notation for classes. So, for example, the boundary classes will be in the package Agate::Boundary. This is the first package that we are illustrating here, and the classes we are concerned with are shown in Fig. A5.3.

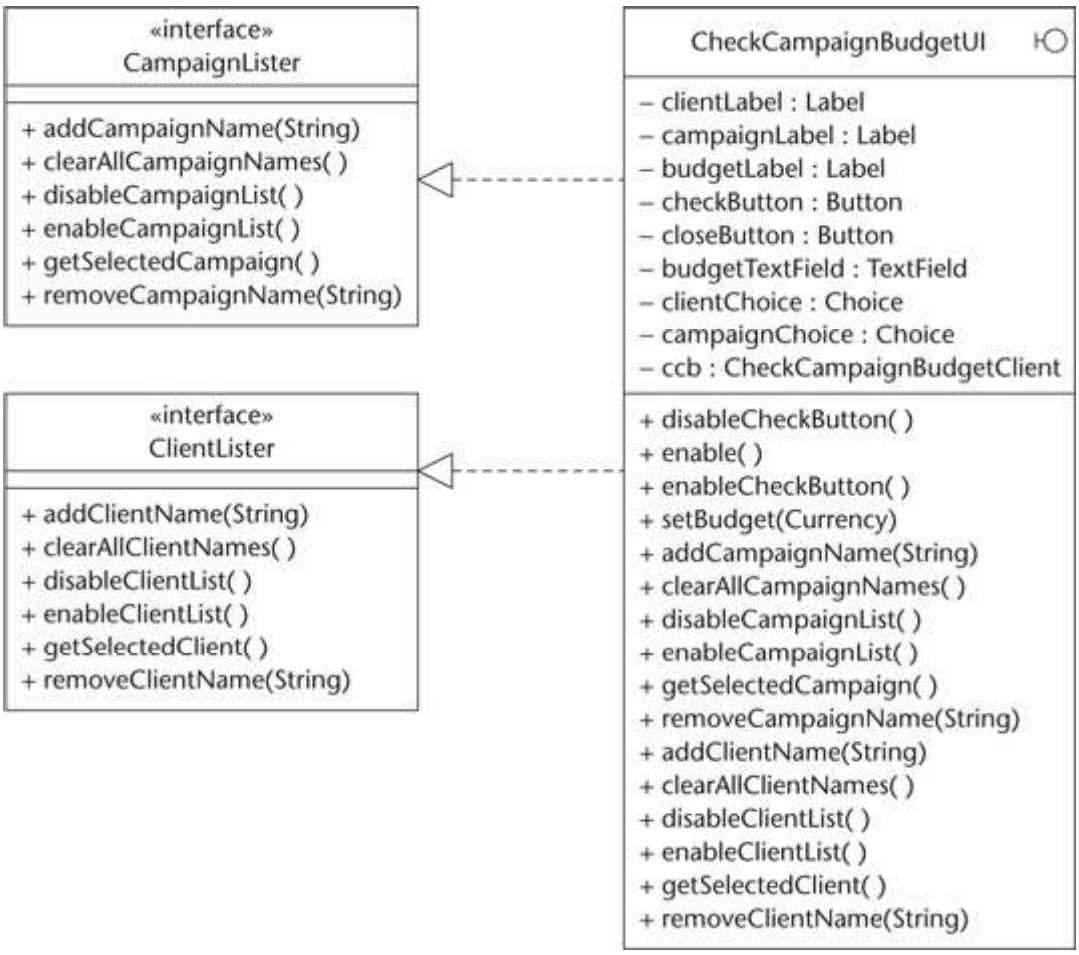


Figure A5.3 Relevant classes in the package Agate::Boundary.

The boundary class CheckCampaignBudgetUI will implement the two interfaces CampaignLister and ClientLister. Note that some of the operations that were included in the class CheckCampaignBudgetUI, such as getSelectedClient(), have been moved into the interfaces, as it is thought that they will apply to any class that implements these interfaces.

Because the control class CheckCampaignBudget will now be split, the version that resides on the client machines (now called CheckCampaignBudgetClient) must be able to respond to the messages addCampaignName() and addClientname(). We have used interfaces for this, because they have to be sent messages remotely by the control classes on the server. This is shown in Fig. A5.4. Note also that this class will need to hold a reference to the version of itself that exists on the server. We have not shown the full package name in the class diagram, but the attribute ccbRemote will in fact be an instance of Agate::Control::Server::CheckCampaignBudgetRemote. In fact, there will be an instance of Agate::Control::Server::CheckCampaignBudgetServer on the server, and for the object on the client to communicate with it via RMI it will have to implement the interface Agate::Control::Server::CheckCampaignBudgetRemote. If ListCampaigns and ListClients only exist on the server, then they will also be in the same package and will implement the interfaces ListCampaignsRemote and ListClientsRemote.

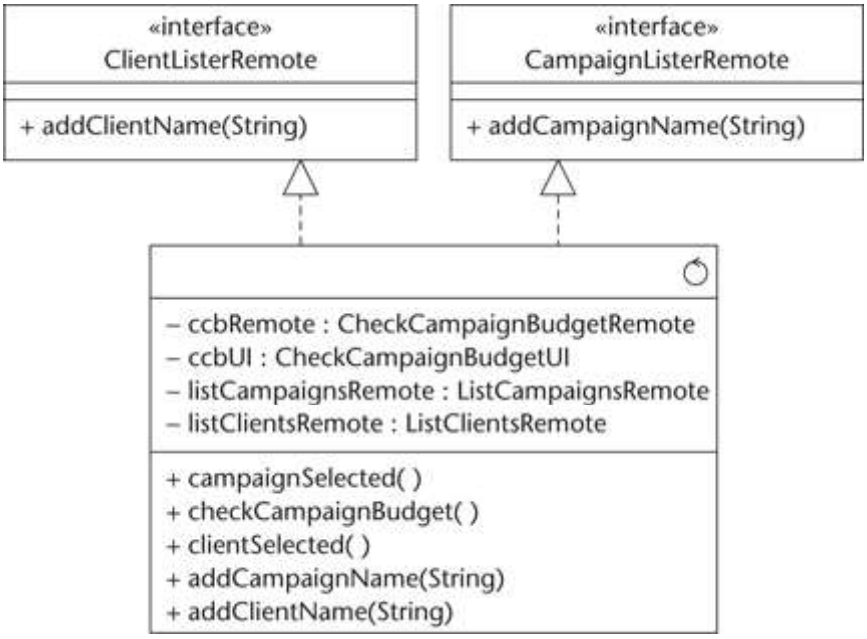


Figure A5.4 The class Agate::Control::Client::CheckCampaignBudgetClient.

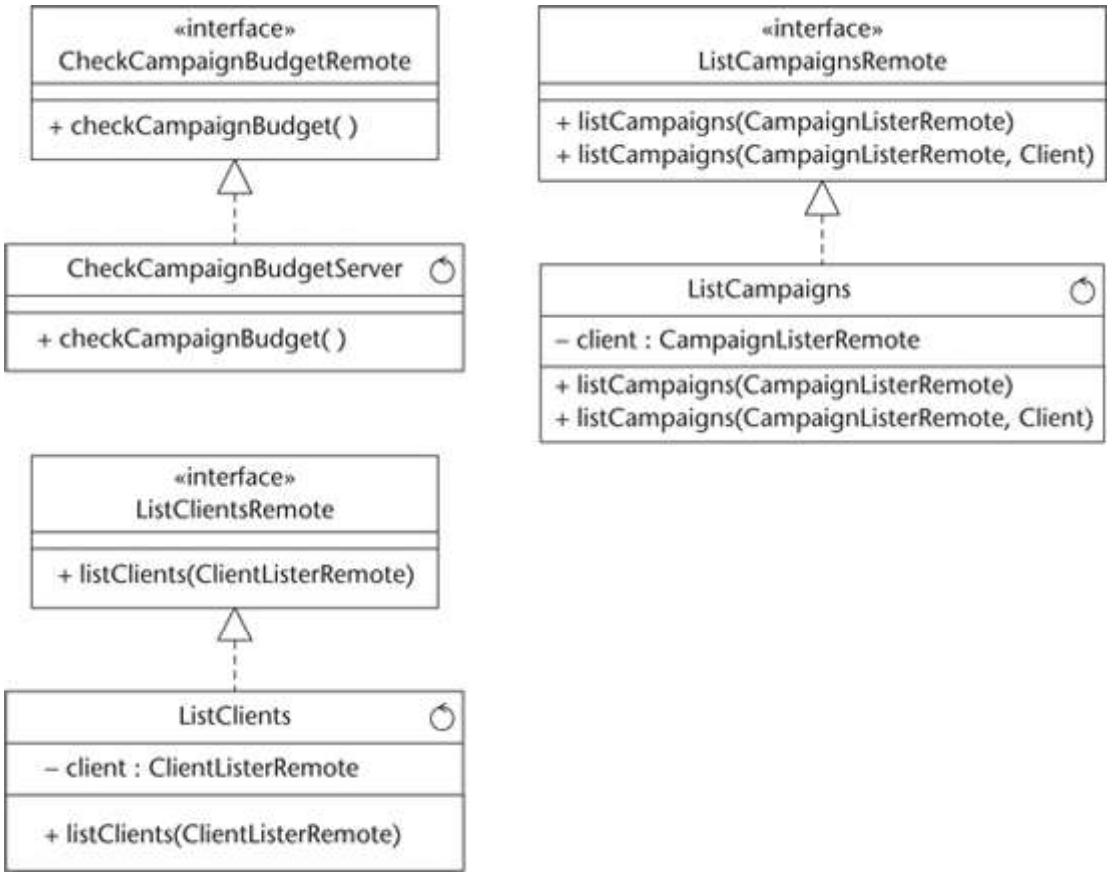


Figure A5.5 Relevant classes in the package Agate::Control::Server.

All the classes that communicate via RMI will need to inherit from the Java RMI package. Rather than being subclasses of the default Java class Object, they will need to be subclasses of java.rmi.server.UnicastRemoteObject.

In Fig. A5.5 we have shown the control classes that reside on the server and the remote interfaces that they must implement. Although we have not shown the full package names, the references to ClientListerRemote and CampaignListerRemote are to the interfaces in the package Agate::Control::Client, shown in Fig. A5.4.

The entity classes that collaborate in this use case are Client, Campaign and Advert. They are shown in a first draft design in Fig. A5.6. However, this design will only work for the kind of application where all the objects are in memory. We need to be able to deal with the process of materializing instances of these classes from the database and, when required, materializing their links with other object instances or collections of object instances. For example, when a particular Client is materialized, we do not necessarily want to re-establish its links with all its Campaigns and the instance of StaffMember that is its staffContact. The Broker pattern, which we discussed in Chapter 18, is a way of making it possible to materialize the objects that are linked to other objects only when they are required.

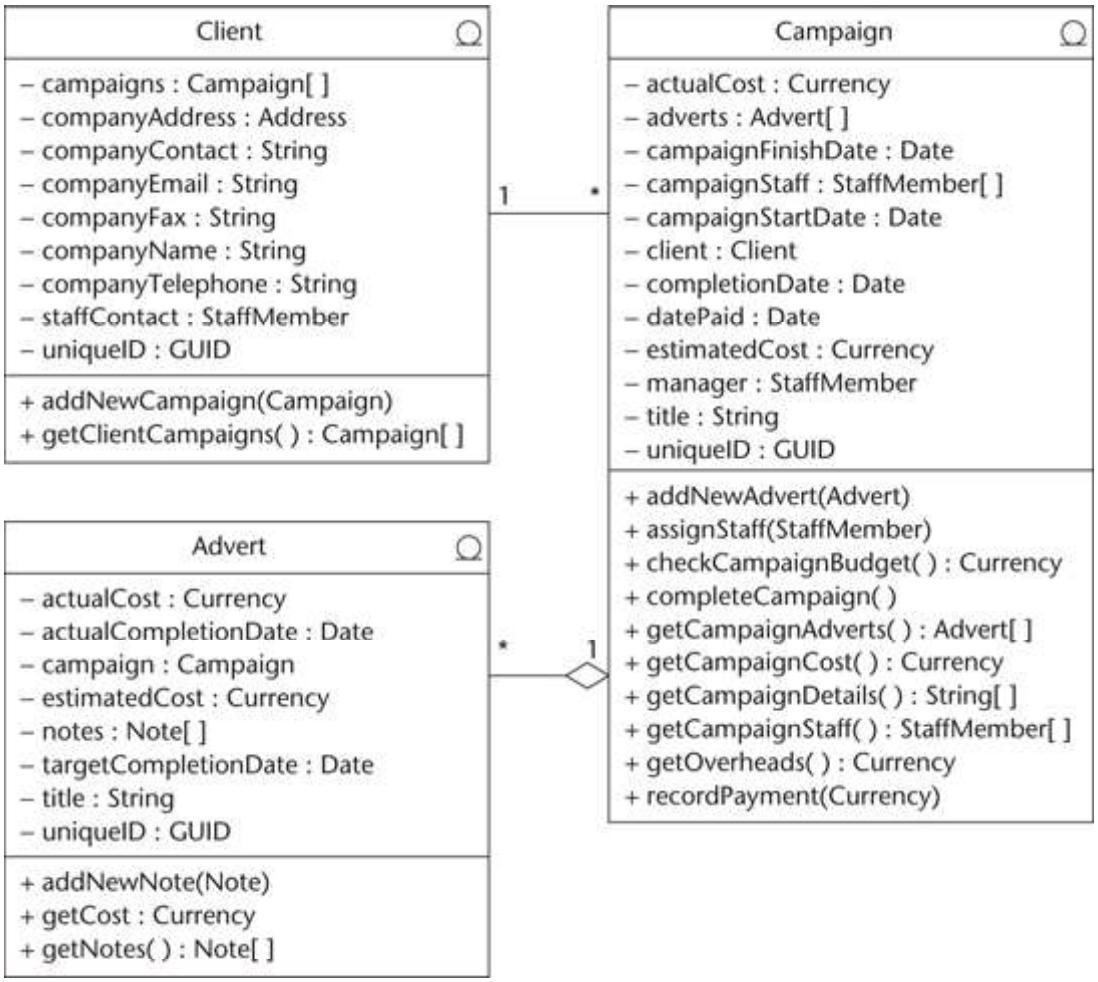


Figure A5.6 First draft design of some classes in the package Agate::Entity::Domain.

In order to achieve this, we can replace the references to the arrays of linked objects with references to the various subclasses of Broker, for example ClientBroker, CampaignBroker and AdvertBroker. Since these are still private attributes, they cannot be referred to directly by other objects and their values can only be obtained by calling one of the operations of the object in which they are contained. The result of this is shown in Fig. A5.7.

The broker subclasses could use the Singleton pattern (see Chapter 15). If this is done, then the design of the operations to return sets of whatever objects they are acting as brokers for will have to be carefully designed to handle concurrent requests from different clients. Alternatively, there could be multiple instances of brokers, and they could be created and destroyed as required, or there could be a pool of brokers available in the server, and when an object needs a broker of a certain type, it would request one from the pool. Figure A5.8 shows the brokers that we are interested in for this use case. We have not used the Singleton pattern in this design.

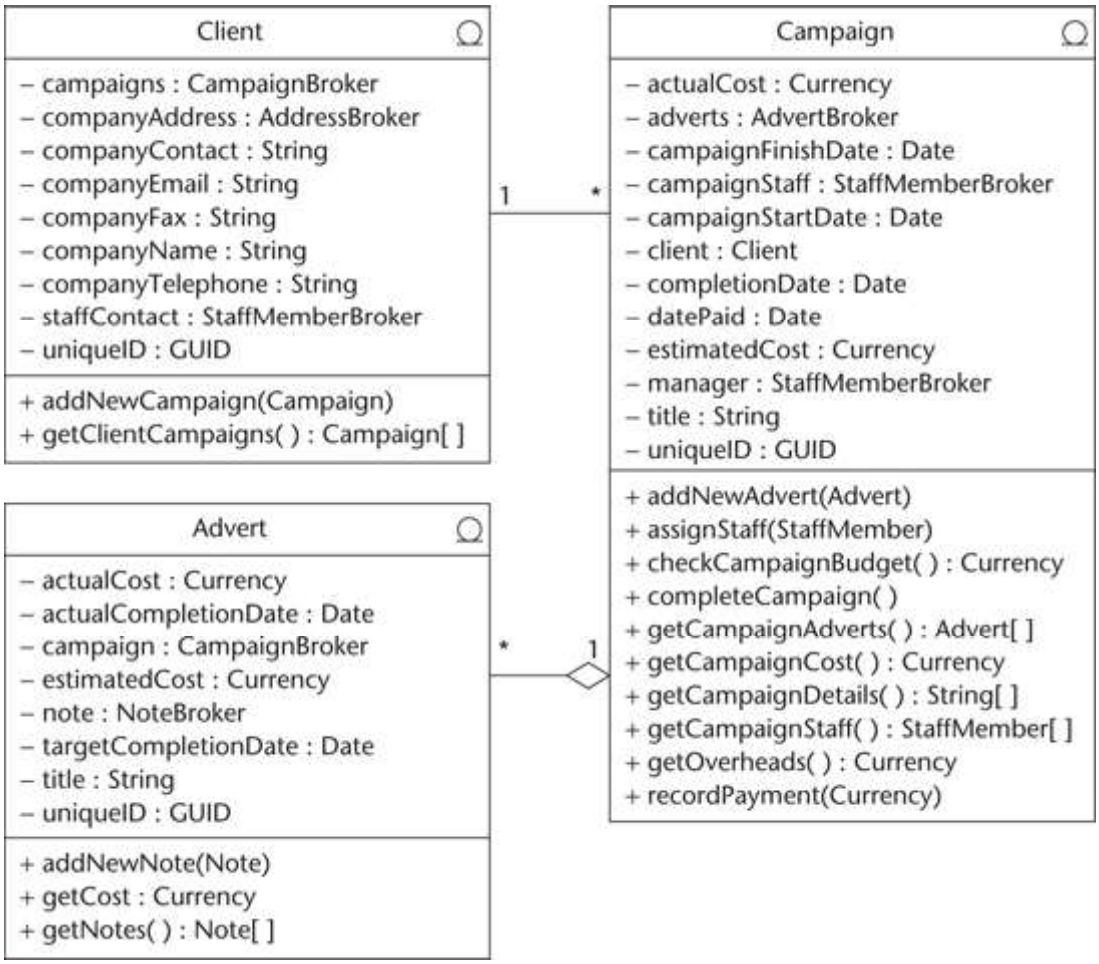


Figure A5.7 Second design of some classes in the package Agate::Entity::Domain.

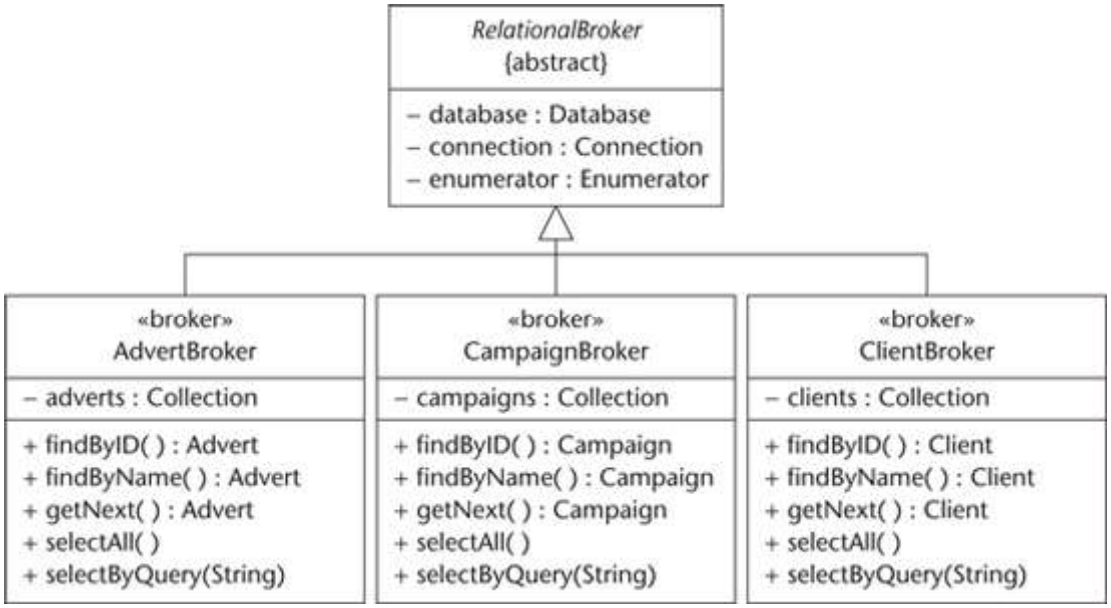


Figure A5.8 Broker classes in the package Agate::Entity::DataManagement.

These brokers will also be used directly by the control classes, for example, when they need to obtain a list of all the objects of a particular class in the system. The brokers have been shown with attributes in which to hold references to the objects necessary for connecting

to the database and issuing queries. We have also assumed that having obtained a list of results, a broker may store it internally in a collection class and allow client objects to iterate through the list of results using an enumerator.[2]

The brokers will be in the package Agate::Entity::DataManagement, together with any other necessary classes to handle the connection to the database. (In this design we are not using proxies or caches, in order to keep it relatively simple.)

The final piece of design necessary to enable the interaction of this use case realization to take place concerns how the control objects on the client machine will obtain references to control objects on the server. For this, we shall use the Factory pattern. A Factory class creates instances of other classes and returns a reference to the new instance to the object that requested it. This is shown in Fig. A5.9.

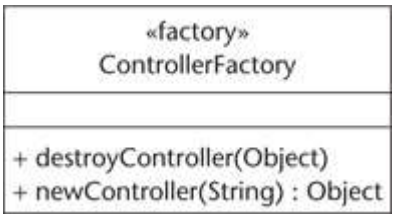


Figure A5.9 Factory class in the package Agate::Control::Server.

So an instance of the control class CheckCampaignBudgetClient on the client machine will request a Factory on the server to provide it with a reference to an instance of CheckCampaignBudgetServer. The Factory will create this instance and pass back a reference to it via the RMI connection with the client. From that point onwards, the client object can make direct requests to the control object on the server. When it is finished with it, it can destroy it, or ask the Factory to destroy it.

In a more sophisticated design, the Factory could hold a pool of already instantiated control classes ready for use. When a client requests an instance of a particular control class, the Factory will take one from the pool if it is available. When the client is finished with the instance, the Factory can put it back into the pool. We are not using pooling in this design, but it is an approach that is commonly used to improve the performance of servers to prevent delays while instances are created and destroyed on demand.

In this design the control class on the server has only one method. This class Agate::Control::Server::CheckCampaignBudgetServer could be designed to hold the business logic for checking the budget of a campaign, but we have taken the decision to leave the responsibility for calculating whether or not the budget is overspent in the Campaign class. There is a case for giving this responsibility to the control class; then, if the business logic changes, it only has to be updated in the control class. However, this makes the entity objects little more than data stores.

Figure A5.10 shows the package diagram with the classes (but not the interfaces) from Figs A5.3 to A5.9. Note that we have not used value objects, which we discussed in Chapter 13 and showed in Fig. 13.24.

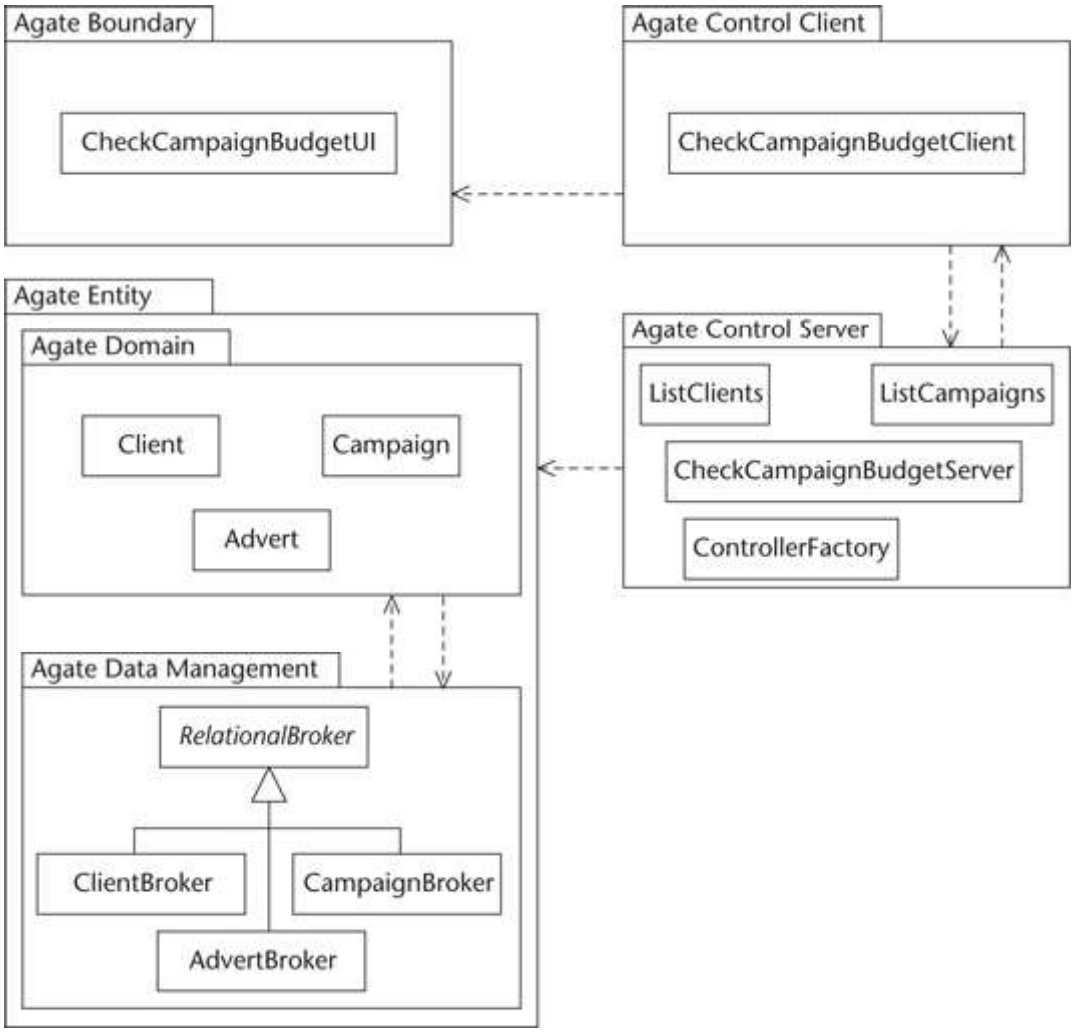


Figure A5.10 Package diagram showing classes.

A5.5 Sequence Diagrams

Figures A5.11 to A5.13 show the sequence diagrams from Chapter 17 revised to take account of the splitting of the control objects and the addition of the Factory class. The package names of objects are also shown.

Although we show the control class on the client as able to directly connect to the instance of ControllerFactory on the server, in reality it would have to request a reference to this object from a naming service or registry on the server: for example, a running instance of the Java rmiregistry.

In Fig. A5.14 we show the interaction between the control class, the brokers and the entity classes. Note how the broker classes perform the tasks involved in retrieving instances or sets of instances from the database.

We have used a simple approach for obtaining the adverts linked to a particular campaign, by having the broker return an array of Adverts. As mentioned above, this could return an enumerator so that the control class could iterate through the collection of Adverts.

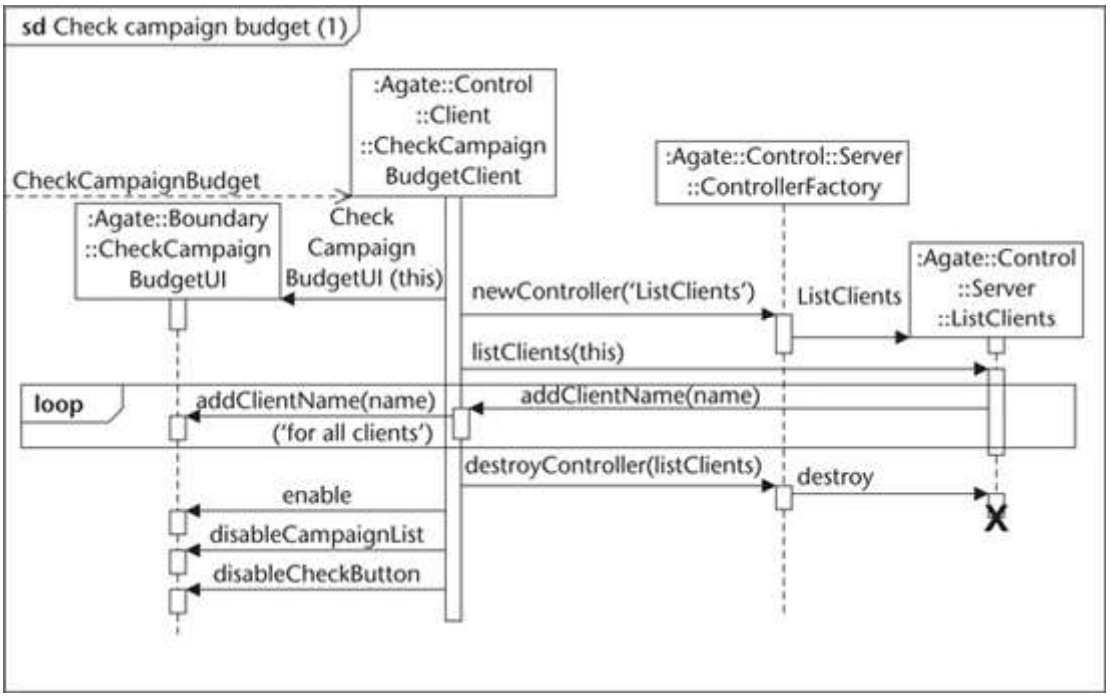


Figure A5.11 First sequence diagram for Check campaign budget.

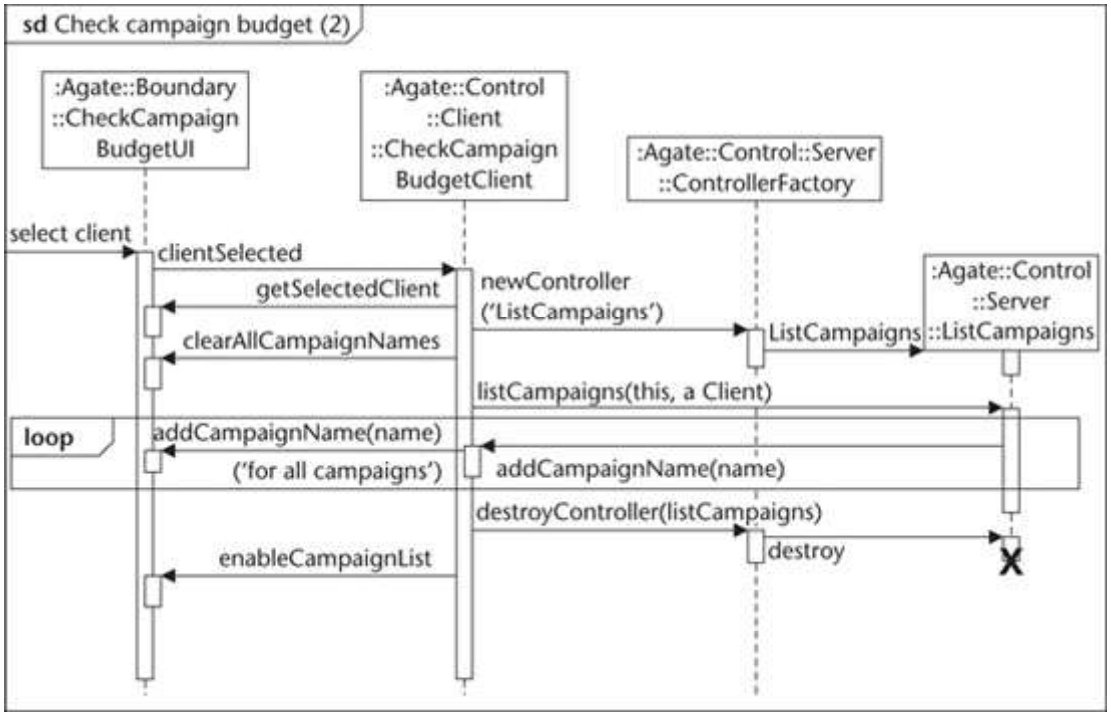


Figure A5.12 Second sequence diagram for Check campaign budget.

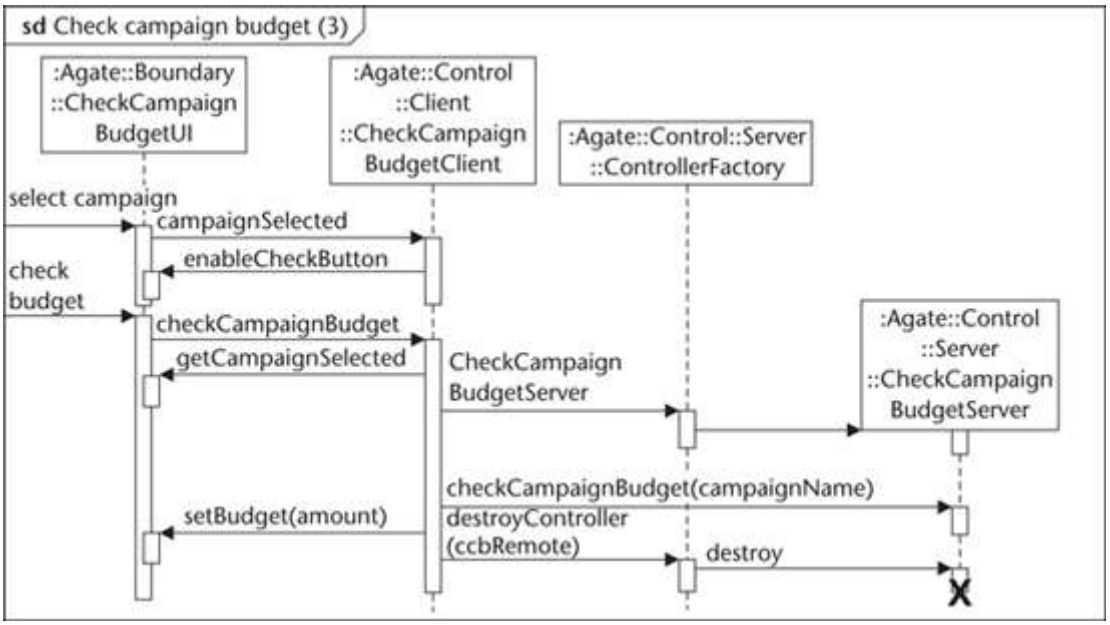


Figure A5.13 Third sequence diagram for Check campaign budget.

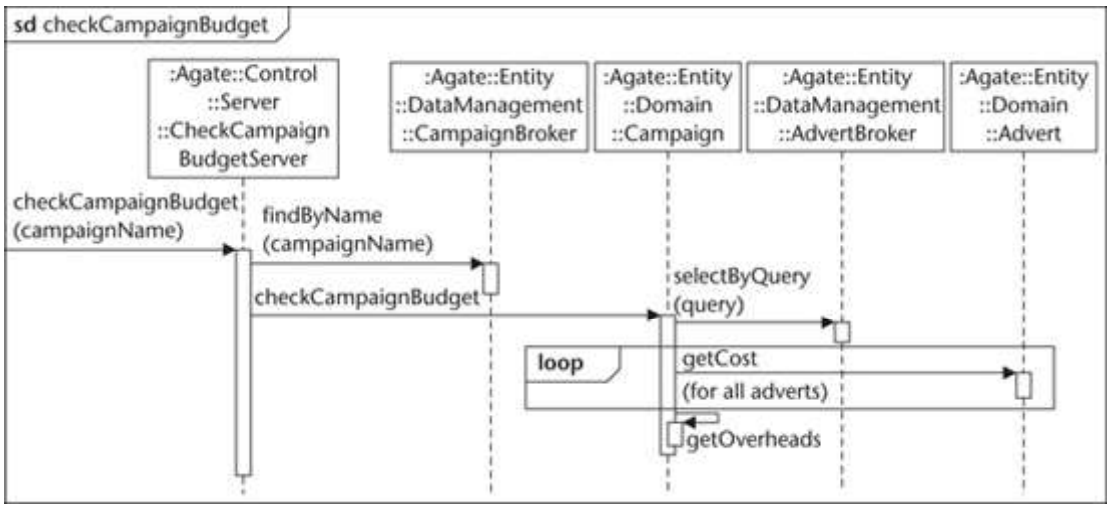


Figure A5.14 Sequence diagram for the operation checkCampaignBudget().

A string named query has been passed to the selectByQuery() operation of the AdvertBroker. The exact format of this will depend on how the object-relational database mapping is set up. If the uniqueID attributes are used in the database as foreign keys, then the SQL statement will be something like:

SELECT * FROM adverts WHERE adverts.campaignID = '123456789';

and the ID of the particular campaign is added into the query string before it is passed to the broker.

A5.6 Database Design

Figure A5.15 shows the SQL to create the tables to map to the classes in Fig. A5.7. The indexes are required to ensure that it is possible quickly to retrieve all the campaigns linked to a client or all the adverts linked to a campaign. A character field has been used to hold the

unique ID for each object. We are assuming that some mechanism will be used to generate these, but have not detailed it here. An alternative would be to use long integer values.

```
CREATE TABLE Clients
  (VARCHAR(30) uniqueID PRIMARY KEY NOT NULL,
  VARCHAR(30) companyAddress,
  VARCHAR(40) companyContact,
  VARCHAR(30) companyEmail
  VARCHAR(30) companyFax,
  VARCHAR(50) companyName NOT NULL,
  VARCHAR(30) companyTelephone,
  VARCHAR(30) staffContactID);
CREATE INDEX client_idx ON Clients (staffContactID, companyName);
CREATE TABLE Campaigns
  (VARCHAR(30) uniqueID PRIMARY KEY NOT NULL,
  FLOAT actualCost,
  DATE campaignFinishDate,
  DATE campaignStartDate,
  VARCHAR(30) clientID NOT NULL,
  DATE completionDate,
  DATE datePaid,
  FLOAT estimatedCost,
  VARCHAR(30) managerID,
  VARCHAR(50) title);
CREATE INDEX campaign_idx ON Campaigns (clientID, managerID, title);
CREATE TABLE Adverts
  (VARCHAR(30) uniqueID PRIMARY KEY NOT NULL,
  FLOAT actualCost,
  DATE actualCompletionDate,
  VARCHAR(30) campaignID NOT NULL,
  FLOAT estimatedCost,
  DATE targetCompletionDate,
  VARCHAR(50) title);
CREATE INDEX advert_idx ON Adverts (campaignID, title);
```

Figure A5.15 SQL to create tables for the classes Client, Campaign and Advert.

A5.7 State Machines

Figure A5.16 shows the event–action table for the state machine of Fig. A5.17. This state machine is the same as the one shown in Chapter 17.

Current State	Event	Action	Next State
–	Check Campaign Budget menu item selected	Display CheckCampaignBudgetUI. Load **Client** dropdown. Disable **Campaign** dropdown. Disable **Check** button. Enable window	1
1	Client selected	Clear **Campaign** dropdown. Load **Campaign** dropdown. Enable **Campaign** dropdown	2
2, 3, 4	Client selected	Clear **Campaign** dropdown. Load **Campaign** dropdown. Clear **Budget** textfield. Disable **Check** button	2
2	Campaign selected	Clear **Budget** textfield. Enable **Check** button	3
3	**Check** button clicked	Calculate budget. Display result	4
3, 4	Campaign selected	Clear **Budget** textfield	3
4	**Check** button clicked	Calculate budget. Display result	4
1, 2, 3, 4	**Close** button clicked	Display alert dialogue	5
5	**OK** button clicked	Close alert dialogue. Close window	–
5	**Cancel** button clicked	Close alert dialogue	H*

Figure A5.16 Event–action table for Fig. A5.17.

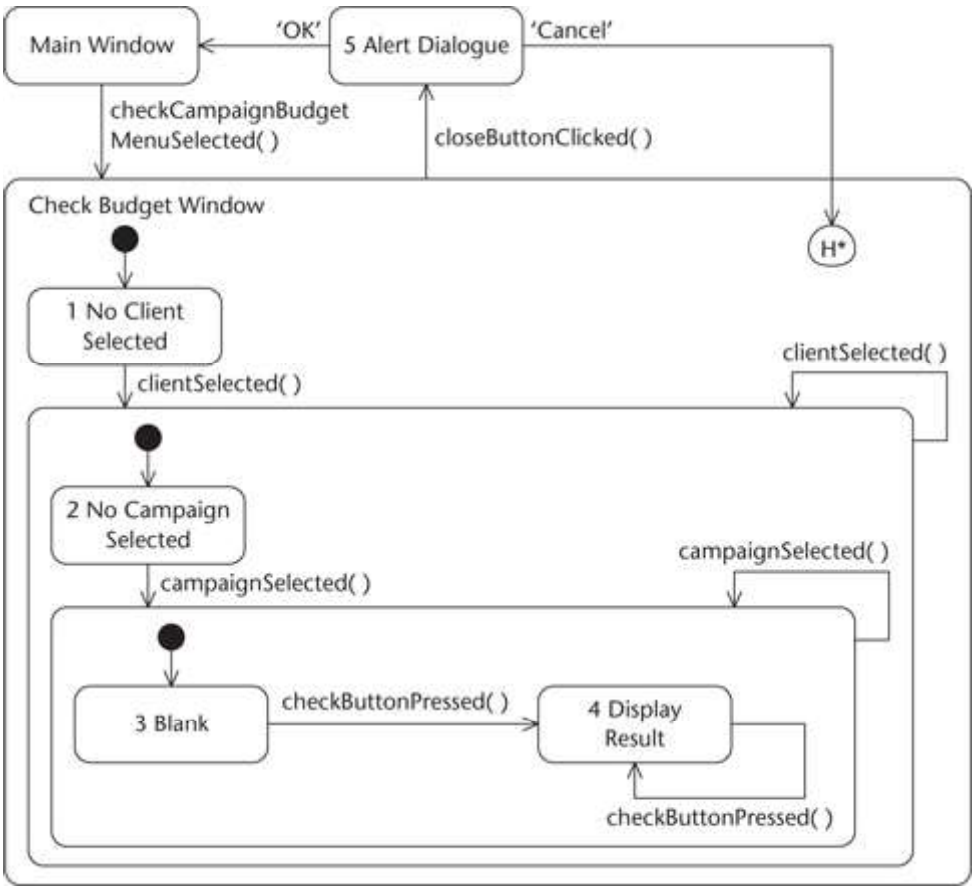


Figure A5.17 State machine for control of the user interface in Check campaign budget.

A5.8 Activities of Design

The activities in the design workflow are shown in the activity diagrams of Figs A5.18 and A5.19.

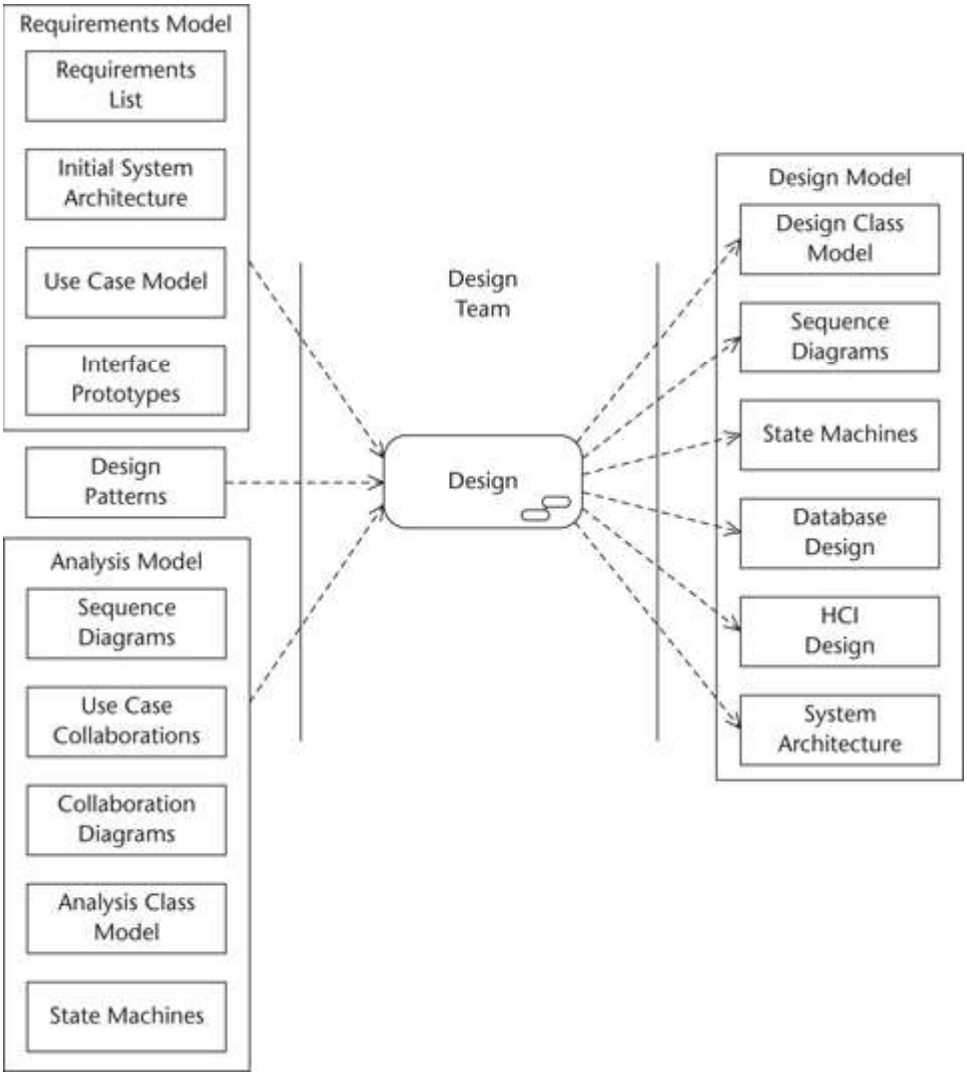


Figure A5.18 Activity diagram for the design workflow.

In order to keep the diagram simple, we have shown the flow of activities in Fig. A5.19 without dependencies on the products that are used and created. Although we have shown a flow through the activities from top to bottom, there will inevitably be some iteration through this workflow even within a major iteration.

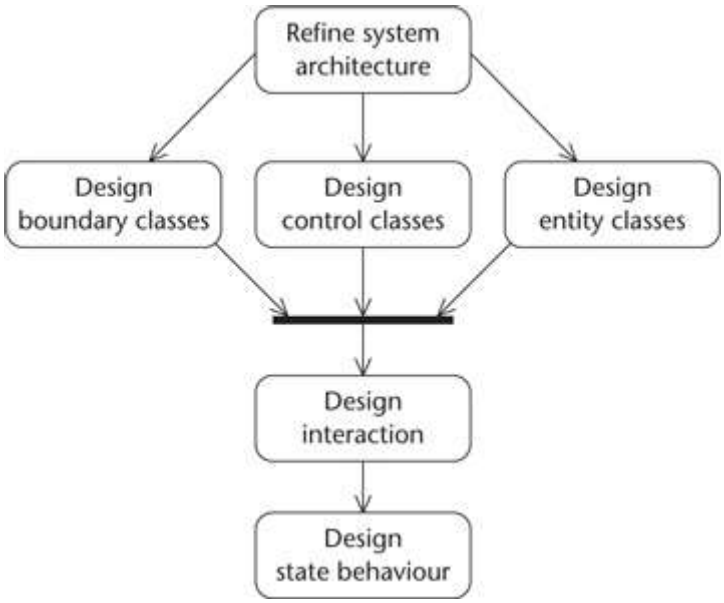


Figure A5.19 Detailed activity diagram for the design workflow.

---

1 To meet the non-functional requirements relating to the distribution of the system, we will need a more complex architecture than this. The eventual solution will probably involve Java 2 Enterprise Edition (J2EE) and Enterprise Java Beans (EJB), and will require the use of application server software. For now we are presenting a design that is not so dependent on an application server, the design for which is beyond the scope of this book.

2 A mechanism for working through a collection dealing with each object in turn.