WHAT IS OBJECT-ORIENTATION ?

Two Approaches of Software development model :

1.Traditional Software development :

In this method overall program is combination of algorithms and Data structures, as it is Algorithmic perspective. In this data is isolated from the process which is major disadvantage, b'cos during the execution it has to locate where required data is available, and if there any changes to data structures we need to change overall processes also.

If system is developed with Structured approach, if customer changes his requirements , its complex to change the overall code b'cos its algorithmic perspective, So we are going for OO approach.

so we are going for Object-oriented software development.

2.Object-Oriented s/w Development model :

It is a way to develop a s/w by building self-contained modules or objects that can be easily replaced, modified and re-used.

In this OO s/w Development method S/W is a collection of descrete objects that encapsulate their data as well as the functionality to model real-world objects.

<u>Reasons for Object-Orientation :</u>

1.Higher level of abstraction

2.Provides transition among different phases of S/W Life cycle.

3. Provides good Programming Techniques

4. Re-usability of Objects.

The following are important features of OO approach-

1.Objects

2.Class and Instance

3.Class membership

4.Generalization

5.Message Passing Mechanisms

6.Polymorphism

7.Object state

1.Objects :

According to Coad and Yourdon object is define as follows.

Object is an abstraction of something in a problem domain, reflecting the capabilities of the system to keep the information about it, interact with it, or both.

Abstraction in this context might be, a form of representation that includes only what is important or interesting from a particular view point.

Eg: a Map is an abstract representation, no map shows every detail of the territory

It covers. The intended purpose of the choice of which details to be given , or which to be suppress.

Mean an Object represents only those features of a thing that are deemed relevant to the current purpose, and hides those features that are not relevant.

According to Rambaugh Object defined as a concept, abstraction, or thing with the boundaries and meaning for the problem at hand.

Objects serves for two purposes.

1.they promote understanding of the real world. 2.Provides a complete practical basis for computer implementation

2.Class and Instance :

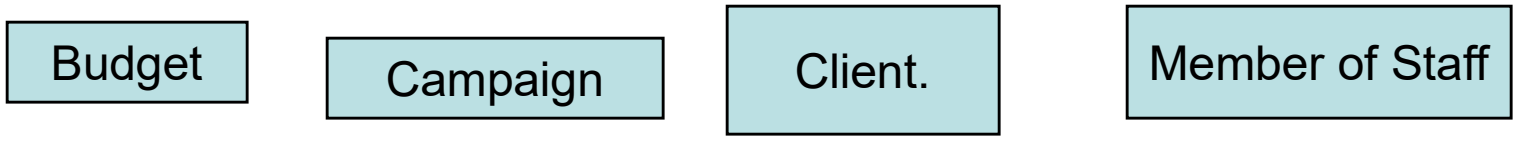
An object represents a particular instance of a class. Objects that are sufficiently similar to each other are said to the same class.

Instance is another word for a single object , but it also carries features of the class to which that object belongs : means every object is an instance of some class.

So like an object , an Instance represents a single person, thing or concept in the application domain.

A Class is a abstract descriptor for a set of instances with certain logical similarities to each other.

The following CAMPAIGN is the class , which is an abstraction that could represent any one of several specific campaigns. This class represents the relevant features that all campaigns have in common. Some examples of campaigns are - A series of magazine adverts for various yellow jewelry products, a national series of TV, cinema, and Magazine adverts. Along with the campaign we have the following classes in the Agate Ltd.. case study which is advertising company.



3. Class membership

the idea that instances belong to a class logically implies that there must be a test that determines to which class an instance belongs. since membership is based on similarity, such a test will also be capable of determining whether any two instances belong to the same class. There are two types of logical similarity which must be tested.

1.Whether All the objects in a class share a common set of descriptive characteristics.

2. Whether all the objects in a class share a common set of valid behaviors or not.

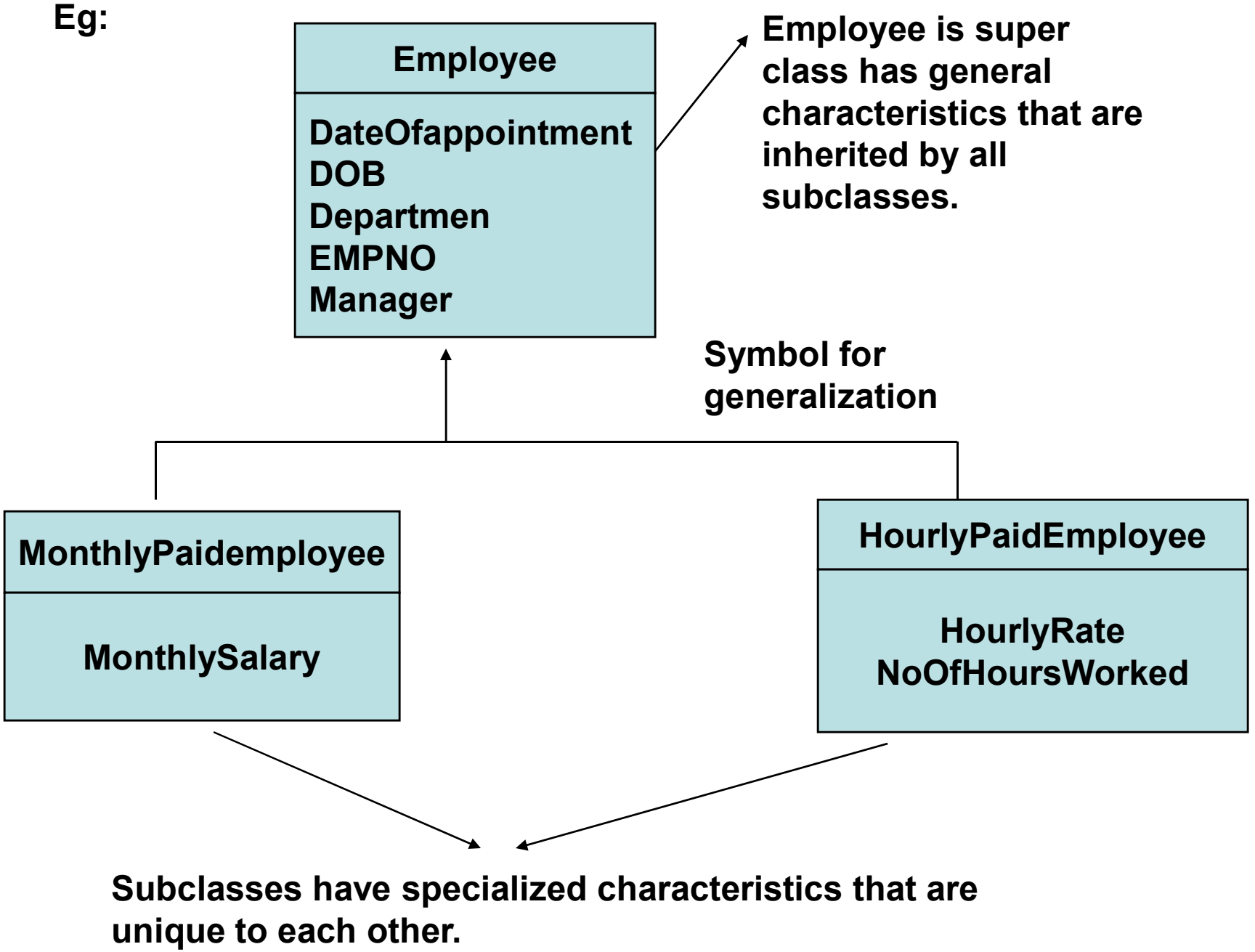
4. Generalization :

In the UML notation Generalization is defined as , it is a taxonomy relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element and may conatain additional information.

Here the Taxonomy means a scheme of hierarchic classification-either an applied set of classifications , or the principles by which that set is constructed.

The main application for Generalization in OO is to describe relationships of similarity b/w classes. Object classes can be arranged into hierarchies.

Eg:


Employee
DateOfappointment
DOB
Departmen
EMPNO
Manager
Employee is super class has general characteristics that are inherited by all subclasses.
Symbol for generalization
MonthlyPaidemployee
MonthlySalary
HourlyPaidEmployee
HourlyRate
NoOfHoursWorked
Subclasses have specialized characteristics that are unique to each other.

Other features of Generalization are

1.Inheritance

2.Transitive Operation of Inheritance

3.Disjoint nature of Generalization.

1.Inheritance : It is used for implementing generalization and specialization in an OO pgming languages. If two classes are related by inheritance , the general class is called Super Class and specialized class is called subclass.

Types : 1.simple Inheritance

2.Multiple Inheritance

3.Multi-Level Inheritance

4.Hybrid Inheritance.

5.Hierarchical Inheritance

2.Trasitive operation : It means that the relationship b/w two elements at adjacent levels of a hierarchy carries over to all more specialized levels.

3. Disjoint nature :

In a hierarchy system , the branches of the tree diverge as they get further away from root and closer to the leaves. they are not permitted to coverage.

5.Message Passing :

In an OO system , Objects communicate with each other by sending messages.

In earlier approaches systems are developed tendency to separate data in a system from the process that act on the data. This method is appropriate but still has some difficulties. That is the process needs to understand the organization of the data that it uses, means process is called dependant on the structure of the data.

This dependency of process on data can also cause, if the data structures were changed for any reason, those processes which uses that data must also be changed.

OO systems avoids these problems by locating each process with the data it uses. Means this is another way of describing an OBJECT : is a data together with process that acts on the data. These Processes are called Operations, and each has a specific signature.

An Operation signature is definition of its interface. In order to invoke an operation, its signature must be given.

In practice its not possible to have all the processes along with data which they access, data and processes are distributed among many different objects.

Message passing is a way of insulating each object from needing to know any of the internal details of the other objects. Essentially the object knows only its own data and its own operations. but in order to for collaboration the objects must to know how to request services from other objects., which may include the retrieval of data. But its is unnecessary for the object services of the another object.

when an object receives a message it can tell instantly whether the message is relevant to it or not . If the message includes a valid signature to one of its operations, the objects can respond. If not, the object does not responds.

So operations residing within an objects, only able to be invoked by a message that gives a valid signature. The complete set of signatures for an object are known as its interface. This is called encapsulation, providing security to the objects by using access specifiers (protocols).

(refer fig for Encapsulation from text book)

6. Polymorphism :

When one person sends a message to another, it is often convenient to ignore many of the differences that exist b/w the various people that might receive the message.

This looks like a Polymorphism, which is important element in OO approaches , defines an ability to appear in many forms, and it refers to the possibility of identical messages being sent to the objects of different classes, each of which responds to the message in a different way.

Polymorphism is a powerful concept for the information systems developer. It permits a clear separation b/w different sub-systems which handles similar tasks in a different manner. This means system can be easily modified or extended to include extra features, since only the interfaces b/w classes need to be known.

7. Object state :

Objects can also occupy different states, and this affects they way that they have responded to messages. Each state is represented by the current values of data within the object , which can in turn be changed by the objects behavior in response to messages.

According to BOOCH Object state is a condition or situation during the life of an object during which it satisfies some condition, performs some activity or waits for event.

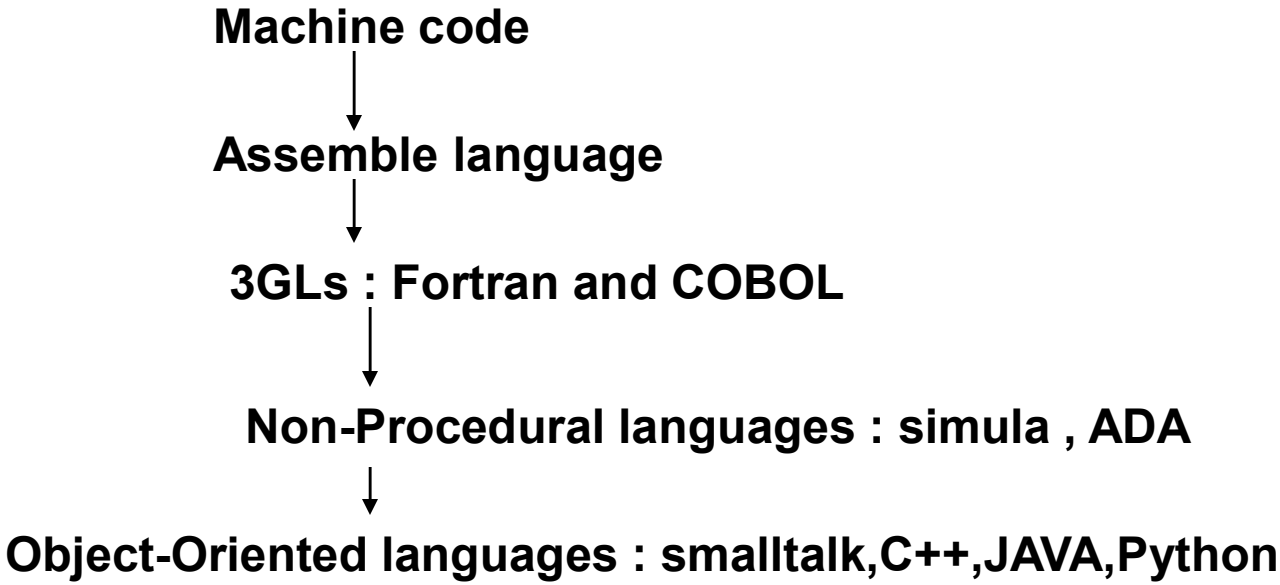
This is fundamental concept to an understanding of the way that the behavior of an OO s/w system is controlled, so that the system responds in an correct way when an external event occurs.

The origins of Object-Orientation :

The following are some strands in the history of computing those have led to OOAD.

1. Increasing Abstraction :

The earlier systems has a steady increase in the level of abstraction at which programmers have ability to operate . The increase in abstraction applies both to the activity of programming itself, and to the computer programs expected to perform. The following path shows increasing abstraction of programming.



2. Event-driven programming :

Early work on systems, simulation led directly to the OO paradigm of independent, collaborating objects that communicate via messages. A typical simulation task is to model the loading of vehicles onto a ship, in order to determine the safe way to do this. This simulation would be run many times under differing assumptions, for eg: the sequence of loading, the arrangement of vehicles on decks, the speed at which the vehicles are driven on to the ship…….etc.

this kind of task is difficult to do in 3GLs, designs of these languages are based on the underlying assumption that the structure controls the flow of execution. If program is written in 3GL, it must to have separate routines that test for , and respond to, a vast no of alternative conditions.

solution for this is, structure the program in a similar way to the problem situation it self: as set of independent s/w agents., each which represents the real-world system that is to be simulated.

In this way the complexity is solved from model of application domain and model of the s/w.

3. Spread of GUIs –

4. Modular s/w – In an OO system, classes have two kinds of definition:

From an External perspective, a class is defined in terms of its interface, which means that other objects need only know the services that offered by objects of that class and the signature used to request each service.

from an Internal perspective , a class is defined in terms of what it knows and what it can do- but only objects of that class need to know anything about this internal definition.

It says that OO system can be constructed so that the implementation of each part is largely independent of the implementation of the other parts., which is what modularity means.

Advantages :

1. It is easy to maintain a system built with Modular design, b'cas as changes to sub system affects very less on remaining system.
2. It is easy to upgrade a modular system.
3. It is easy to build a system which is reliable in use.
4. Each module provides useful and coherent package of functionality.

4. Life cycle Problems :

In structured methodologies ,we have disadvantages, b'cas we apply waterfall life cycle model for designing large engineering projects.

for this in an OO development it tells to apply cyclic development approach , in which there is less difficulty in revisiting and revising earlier stages in an iterative process of product that can repeat.

5. Model transitions :

In structured approach, the models developed during analysis phase –eg: Data flow diagrams have an indirect relationship with the process models developed during Design phase-eg: Structured charts… from these diagrams its hard to get original requirements .

OOAD avoid these transition problems by using a core set of models throughout analysis and design adding more details at each stage. Use-case and Class diagrams constitute backbone of analysis and design in OOAD.

6. Reusable software :

Object-oriented Languages Today - Features

- Strong typing refers to the degree of discipline that a language enforces on the programmer when declaring variables.

-Static type checking is at compile time.

- dynamic type checking is done at rum time.

- Garbage collection is concerned to memory management in systems to create and delete many objects during execution. If objects are not removed from memory when they deleted, the system may run out memory in which to execute.

-Multiple inheritance refers to the capacity of an object acquiring features from more than one hierarchy.

- Languages in which all constructs are implemented as classes or objects are said to be “pure” object oriented languages.

-Dynamic loading refers the ability of a language to load new classes at run-time.

-Standardized class libraries allows programmer to run their programs on different platforms and even on different Operating systems.

-Correctness construct includes pre-conditions and post-conditions on methods, forming a contract b/w any client-supplier pair.

Object-Oriented Languages Today :

No. of object oriented languages are available today, with some significant differences between their features as shown below.

Features	Smalltalk	C++	Eiffel	JAVA
Strong typing	Yes	optional	Yes	Yes
Static/dynamic typing	D	S	S	S+D
Garbage collection	yes	no	yes	yes
Multiple inheritance	no	yes	yes	no
Pure objects	yes	no	yes	no
Dynamic loading	yes	no	no	yes
Standardized class libraries	yes	no	yes	yes
Correctness construct	no	no	yes	no

Limitations of Object-Orientation –

If the application are with the following features those cannot be implemented with OO features.

- If systems are strongly database- oriented , Means both that have a record-based structure of data that is appropriate to a RDBMS, and also that their main processing requirements centre on the storage and retrieval of the data. Such applications cannot be adopted for OO implementation without losing benefits of using a RDBMS for data storage.

-Applications which are strongly algorithmic in their operation are less suited to an OO development.

- Applications concerned to Scientific engineering which involves a large and complex calculations is also not suitable for OO development , if developed with OO features then it may contain few objects but each is with extremely complex.

Agate Ltd Case study – Introduction

Agate is an advertising agency in UK , formed with three advertising executives- Amarjeet grewel , Gordon Anderson and Tim Eng. (Agency name is the combination of their initials).

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/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 actual cost is recorded. When the client pays , the payment date is recorded. Each campaign includes one or more adverts. Adverts can be one of the several types.

- news paper advert

-Magazine advert

- TV Advert

-Radio advert

-Poster advert

-Leaflet

Purchasing assistants are responsible for buying space in news papers and magazines , Space on advertising hoardings , and TV or Radio….etc. The actual cost of a campaign is calculated from range of information. This includes

-Cost of staff time for graphics, copy-writing etc;

- cost of studio time and actors

- cost of copyright material

-Cost of space in news papers…..

this information is held in a paper based filling system, but the total estimated cost and the final actual cost of a campaign are held on the new system. 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.

5. MODELING CONCEPTS

MODELS AND DIAGRAMS :

In any development of projects, the main focus of analysis and design activities is on models, which are usually both abstract and visible. Many of the products are themselves abstract in nature. Most s/w is not tangible for the user. Generally a s/w is usually constructed by teams of people who need to see each others models. So we need to prepare models understandable by everyone in the development team.

MODEL : It is a simplification of reality. They are very useful in the following ways.

- A model is a quicker and easier to build.
- A model can be used in simulations to learn more about things in representation.
- A model gives clear understanding of a problem.
- A model is an abstraction ,mean we can choose which are to be represented, and which are to be suppressed.
- A model can represent real or imaginary things from any domain.
- A useful model has just the right amount of detail and structure.

The main purpose of a model is to represent functional and non-functional requirements. The whole requirement model must be accurate, complete and unambiguous. This does not include premature conditions about how new system fulfils user requests.

DIAGRAM:

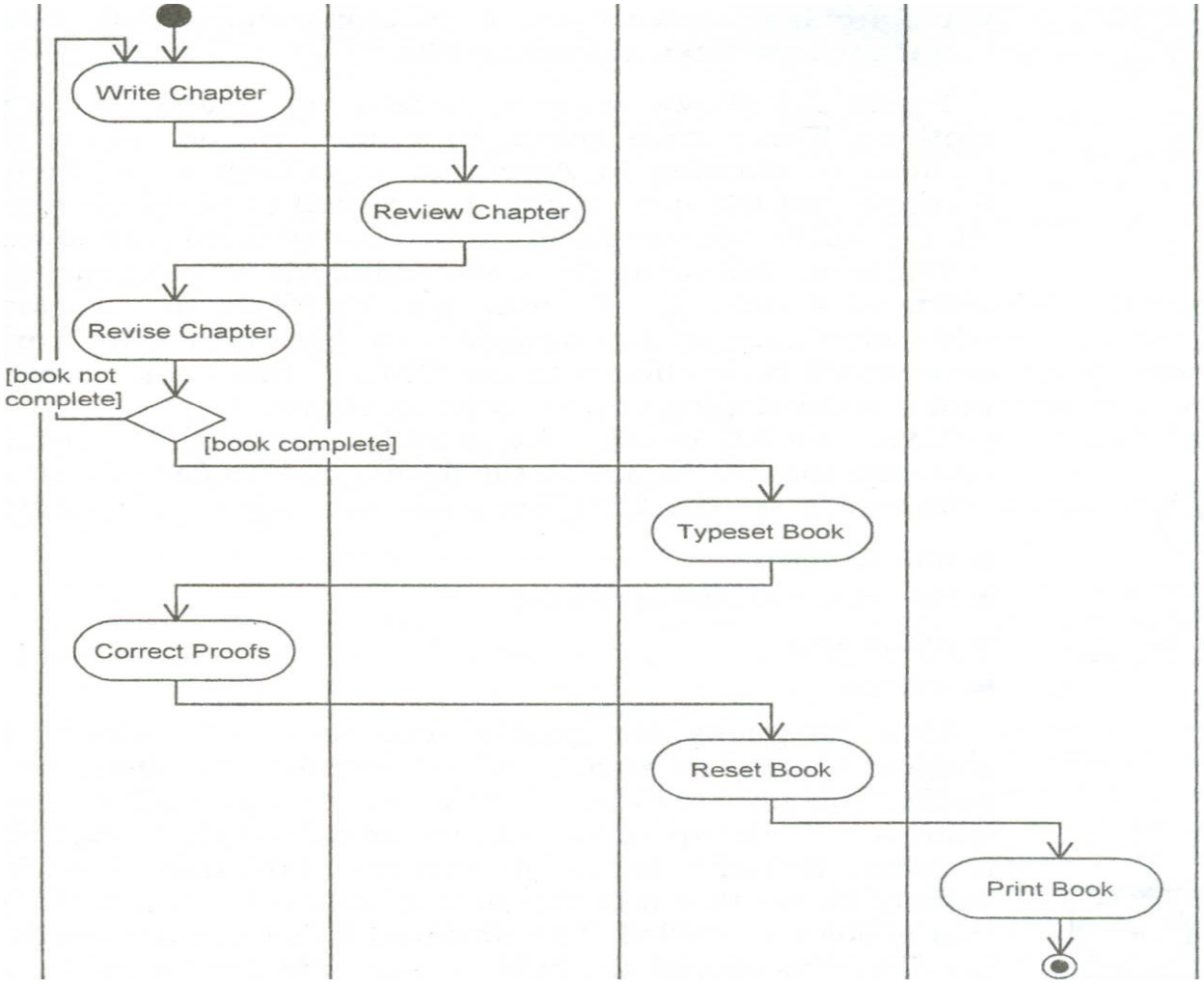
Analysts and designers use diagrams to bulid models of systems in the same Way as architects use diagrams to model buildings.

These diagrammatical models are helpful

- To communicate ideas.
- To generate new ideas and possibilities.
- To test ideas and more predictions.
- To understand structures and relationships.

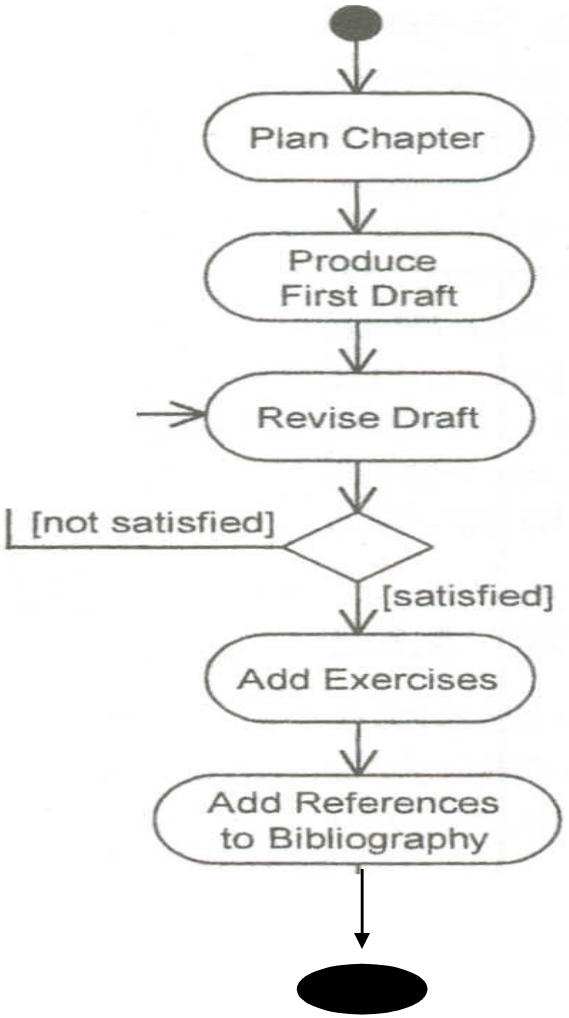
If a system is very simple it may be possible to model it with a single diagram and supporting textual descriptions. But most systems are complete and may require many diagrams fully to capture that complexity.

The following fig shows an example of a diagram(a UML activity diagram) used to show a part of the process of producing book.



Activity diagram for producing a book

The above fig is incomplete ,and can be made complete by expanding each of rounded rectangles into more detail.



shows breaking the activity write chapter into other activities.

The above diagrams are typical kind of diagram used in system analysis and design.

-Abstract shapes are used to represent things or actions from the real world.

-The choice of what shapes to use determined by a set of rules that are available in UML for a particular diagram.

- It is important that we follow the rules about diagram, otherwise the diagrams
may not make sense or other people may not understand them.
- Standards are also important as they promotes communication in the same way as a common language.

The following rules are to be considered for the designers of modeling techniques :

- Simplicity of representation --only showing what needs to be shown.
- Internal consistency --with in a set of diagrams.
- Completeness --showing all the needs to be shown.
- Hierarchical representation --breaking the system down and showing more details at lower levels.

UML consists mainly of a graphical language to represent the concepts that we require in the development of an object-oriented information system. UML diagrams are made up of four elements:

1.Icons,
2.two-dimensional symbols,
3.paths and 4. strings.

UML diagrams are *graphs-composed* of various kinds of shapes, known as *nodes,* joined together by lines, known as *paths.* The activity diagrams in figures illustrate this. Both are made up of *two-dimensional symbols* that represent activities, linked by arrows that represent the transitions from one activity to another and the flow of control through the process that is being modeled.

The start and finish of each activity graph is marked by special symbols-icons-the dot for the initial state and the dot in a circle for the final state. The activities are labeled with *strings,* and strings are also used at the decision points (the diamond shapes) to show the conditions that are being tested.

The UML Semantics section of the UML Specification provides the more formal , grammar of UML-the syntax-and the meaning of the elements and of the rules about how elements can be combined-the semantics.

Differences b/w model and diagram :

A single diagram can illustrate or document some aspect of a system. However, a model provides a complete view of a system at a particular stage and from a particular perspective.

For example, a Requirements Model of a system will give a complete view of the requirements for that system. It may use one or more types of diagram and will most likely contain sets of diagrams to cover all aspects of the requirements. These diagram may be grouped together in models in their own right.

On the other hand a Behavioral Model of a system will show those aspects of a system that are concerned with its behavior-how it responds to events in outside world and to the passage of time. At the end of the analysis stage of a project the Behavioral Model may be quite simple, using Collaboration Diagrams to show which classes collaborate to respond to external events and with informally defined messages passing between them. By the end of the design stage of a project, Behavioral Model will be considerably more detailed, using Interaction Sequence Diagrams to show in detail the interaction between classes within a collaboration, and with every message defined as an event or an operation of a class.

Models in UML :

In UML there are a number of concepts that are used to describe systems and the ways in which they can be broken down and modelled. A *system* is the overall thing that is being modelled, such as the Agate system for dealing with clients and their advertising campaigns.

A *sub-system* is a part of a system, consisting of related elements, example the Campaigns sub-system of the Agate system. A *model* is an abstraction system or sub-system from a particular perspective or *view.* An example would be use case view of the Campaigns sub-system, which would be represented by a model containing use case diagrams, among other things. A model is complete and consist at the level of abstraction that has been chosen. Different views of a system can presented in different models, and a *diagram* is a graphical representation of a set elements in the model of the system.

Different models present different views of the system.

Five views to be used with UML: the use case view, the design view, the process view, the implementation view and the deployment view. The choice of diagrams that used to model each of these views will depend on the nature and complexity of system that is being modelled.

Indeed, you may not need all these views of a system. If the system that you are developing runs on a single machine, then implementation and deployment views are unnecessary, as they are concerned which components must be installed on which different machines.

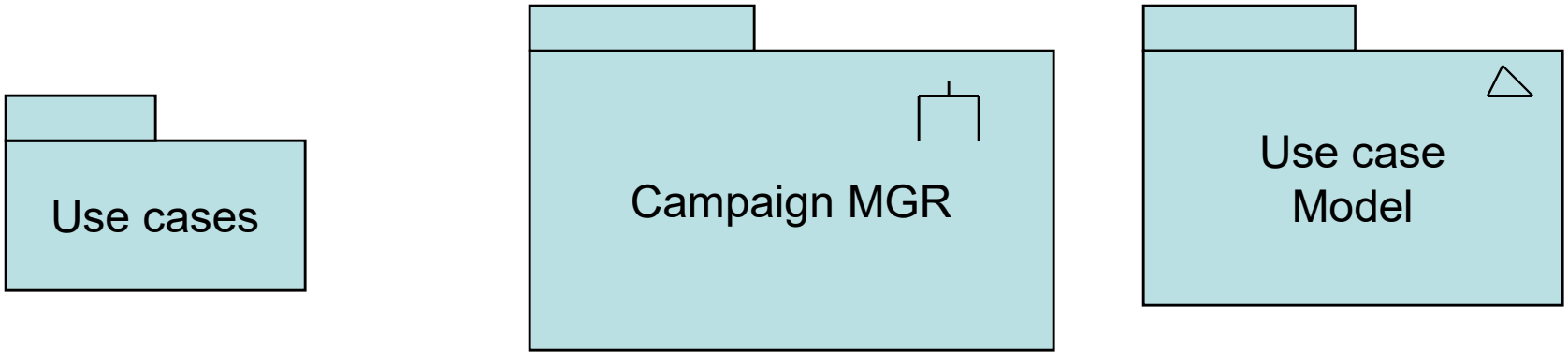
UML provides a notation for modelling sub-systems and models that uses extension of the notation for *packages* in UML. Packages are a way of organizing model elements and grouping them together. They do not represent things in system that is being modelled, but are a convenience for packaging together elements that do present things in the system.

Developing models :

The models that we produce during the development of a system change as the project progresses. They change along three main dimensions:

1.Abstraction 2. formality 3. level of detail.

The following are UML Notations for Package, sub-systems and models



Package

Sub system

Model

In a system development project that uses an iterative life cycle, different models that represent the same view may be developed at different levels of detail as the project progresses.

For example,

Initially, the first use case model of a system may show only the obvious use cases that are apparent from the first iteration of requirements capture.

After a second iteration, the use case model may be elaborated with more detail and additional use cases that emerge from discussion of the requirements. Some prototypes may be added to try out ideas about how users will interact with the system.

After a third iteration, the model will be extended to include more structured descriptions of how the users will interact with the use cases and with relationships among use cases.

It is also possible to produce a model that contains a lot of detail, but to hide or suppress some of that detail in order to get a simplified overview of some aspect of the system.

Drawing Activity diagrams –

Purpose of Activity diagram :

Activity diagrams can be used to model different aspects of a system. At a high level, they can be used to model business activities in an existing or potential system. For this purpose they may be used early in the system development lifecycle.

They can be used to model a system function represented by a use case, possibly using object flows to show which objects are involved in a use case. This would be done during the stage of the life cycle when requirements are being elaborated.

They can also be used at a low level to model the detail of how a particular operation is carried out, and are likely to be used for this purpose late in analysis or during the design stage of a project. Activity diagrams are also used within the Unified Software Development Process (USDP) - to model the way in which the activities of the USDP are organized and relate to one another in the software development lifecycle.

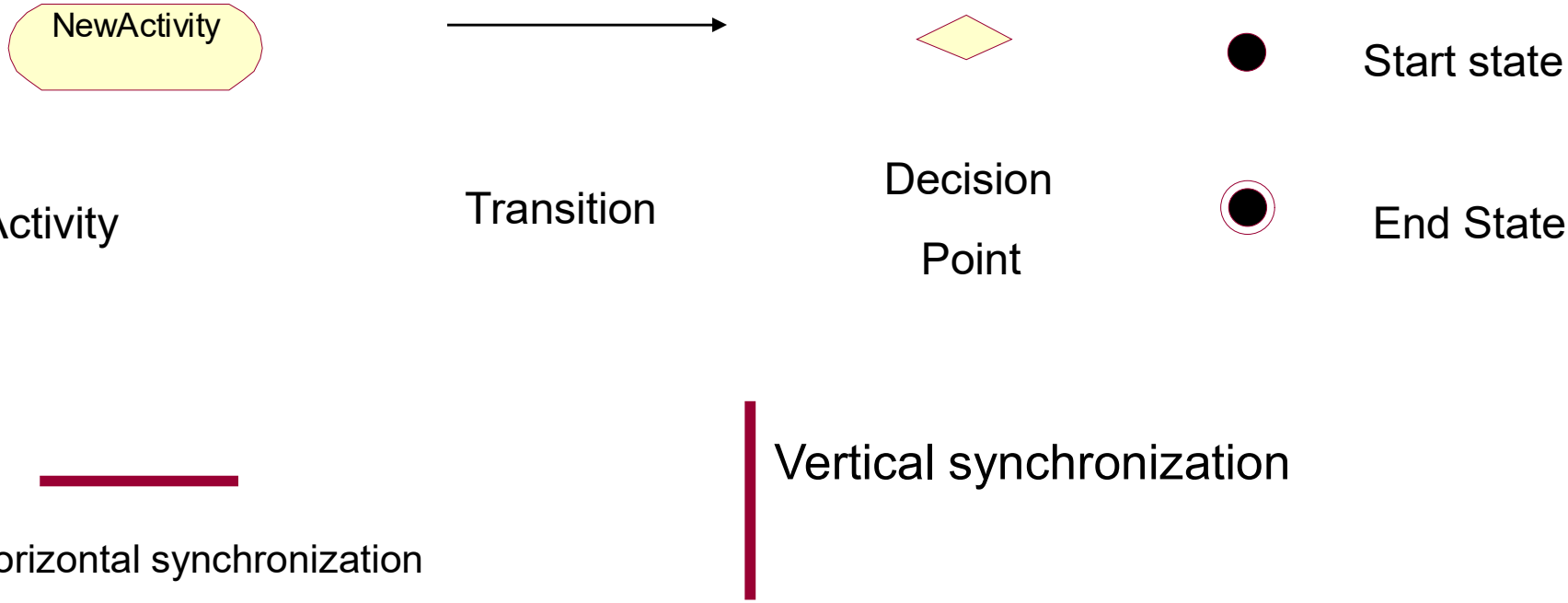
Mainly, activity diagrams are used for the following purposes:

- to model a task (in business modelling for instance);
- to describe a system function that is represented by a use case;
- in operation specifications, to describe the logic of an operation;
- in USDP to model the activities that make up the lifecycle.

Activity diagrams are really most useful to model business activities in the early stages of a project. For modelling operations, interaction sequence diagrams are closer to the spirit of object-orientation. However, there may be occasions when the analyst wants to model the activities that must be carried out, but has not yet identified the objects or classes that are involved or assigned responsibilities to them. In such circumstances, activity diagrams may be an appropriate tool to use.

Notation of Activity Diagrams –

Activity diagrams contain activities, transitions between the activities, decision points, and synchronization bars. In the UML, activities are represented as rectangles with rounded edges, transi-tions are drawn as directed arrows, decision points are shown as diamonds, and synchronization bars are drawn as thick horizontal or vertical bars as shown in the following.



Activities :
An activity represents the performance of some behavior in the workflow.

Transitions :
Transitions are used to show the passing of the flow of control from activity to activity. They are typically triggered by the completion of the behavior in the originating activity.

Decision Points :
When modeling the workflow of a system it is often necessary to show where the flow of control branches based on a decision point. The transitions from a decision point contain a guard condition, which is used to determine which path from the decision point is taken. Decisions along with their guard conditions allow you to show alternate paths through a work flow.

Synchronization Bars

In a workflow there are typically some activities that may be done in parallel. A synchronization bar allows *you* to specify what activities may be done concurrently.

Synchronization bars are also used to show joins in the workflow; that is, what activities must complete before processing may continue.

Means, a synchronization bar may have many incoming transitions and one outgoing transition, or one incoming transition and many outgoing transitions.

Swimlanes

Swimlanes may be used to partition an activity diagram. This typically is done to show what person or organization is responsible for the activities contained in the swimlane.

Initial and Final Activities

There are special symbols that are used to show the starting and final activities in a workflow. The starting activity is shown using a solid filled circle and the final activities are shown using a bull's eye.

Typ-ically, there is one starting activity for the workflow and there may be more than one ending activity (one for each alternate flow in the workflow).

Modeling a workflow in an activity diagram can be done several ways; however, the following steps present just one logical process:

1.Identify a workflow objective. Ask, "What needs to take place or happen by the end of the workflow? What needs to be accomplished?" For example, if your activity diagram models the workflow of ordering a book from an online bookstore, the goal of the entire workflow could be getting the book to the customer.

2.Decide the pre and post-conditions of the workflow through a start state and an end state. In most cases, activity diagrams have a flowchart structure so start and end states are used to designate the beginning and ending of the workflow. Start and end states clarify the perimeter of the workflow.

3.Define and recognize all activities and states that must take place to meet your objective. Place and name them on the activity diagram in a logical order.

4.Define and diagram any objects that are created or modified within your activity diagram. Connect the objects and activities with object flows.

5.Decide who or what is responsible for performing the activities and states through swimlanes. Name each swimlane and place the appropriate activities and states within each swimlane.

6.Connect all elements on the diagram with transitions. Begin with the "main" workflow.

7.Place decisions on the diagram where the workflow may split into an alternate flow. For example, based on a Boolean expression, the workflow could branch to a different workflow.

8.Evaluate your diagram and see if you have any concurrent workflows. If so, use synchronizations to represent forking and joining.

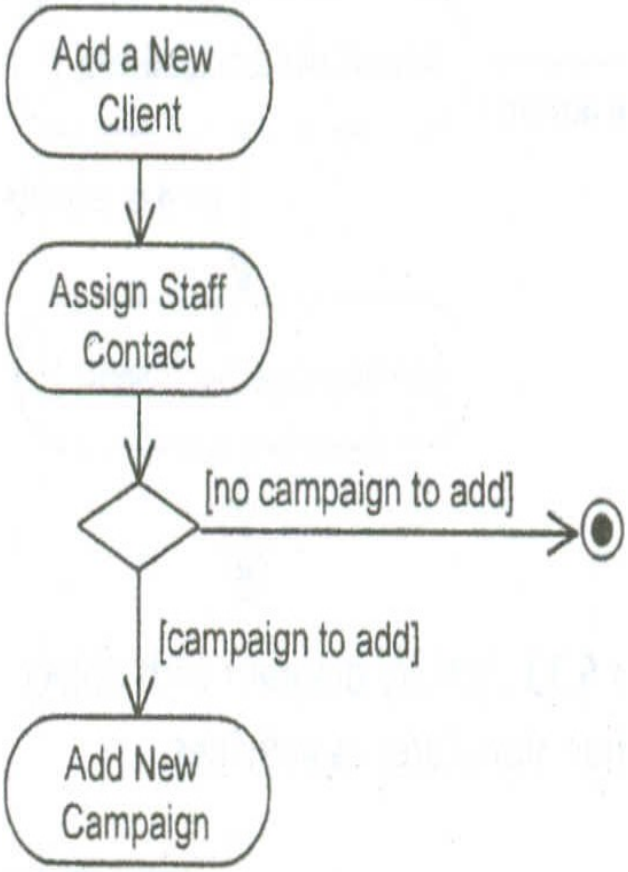
9.Set all actions, triggers and guard conditions in the specifications of each model element.

Activity diagrams at their simplest consist of a set of activities linked together by transitions from one activity to the next. Each activity is shown as a rectangle with rounded ends. The name of the activity is written inside this two-dimensional symbol. It should be meaningful and summarize the activity.

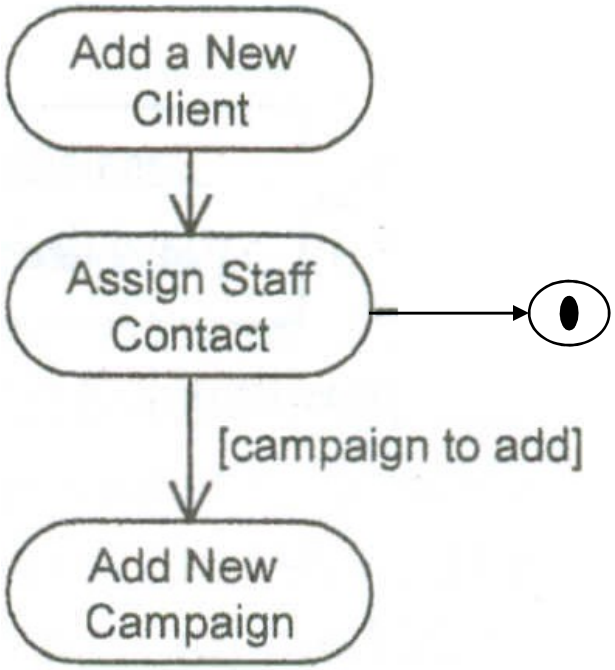
The following fig shows an example of two activities joined by a transition.

Activities exist to carry out some task. In the Fig, the first activity is to add a new client into the Agate system . The transition to the second activity implies that as soon as the first activity is complete, the next activity is started. Sometimes there is more than one possible transition from an activity.

After having a client if there is a campaign immediately we will assign for campaign otherwise there is no activity, b'cos even if client is available there is no sure we always will have campaign, is shown below.



This transcript describes some choices that can be made, and these choices will affect the activities that are undertaken. We can show these in an activity diagram with an explicit decision point, represented by a diamond shaped icon .



Always its not necessary to use explicit decision points, the diagrams shows alternative transitions out of the activity.

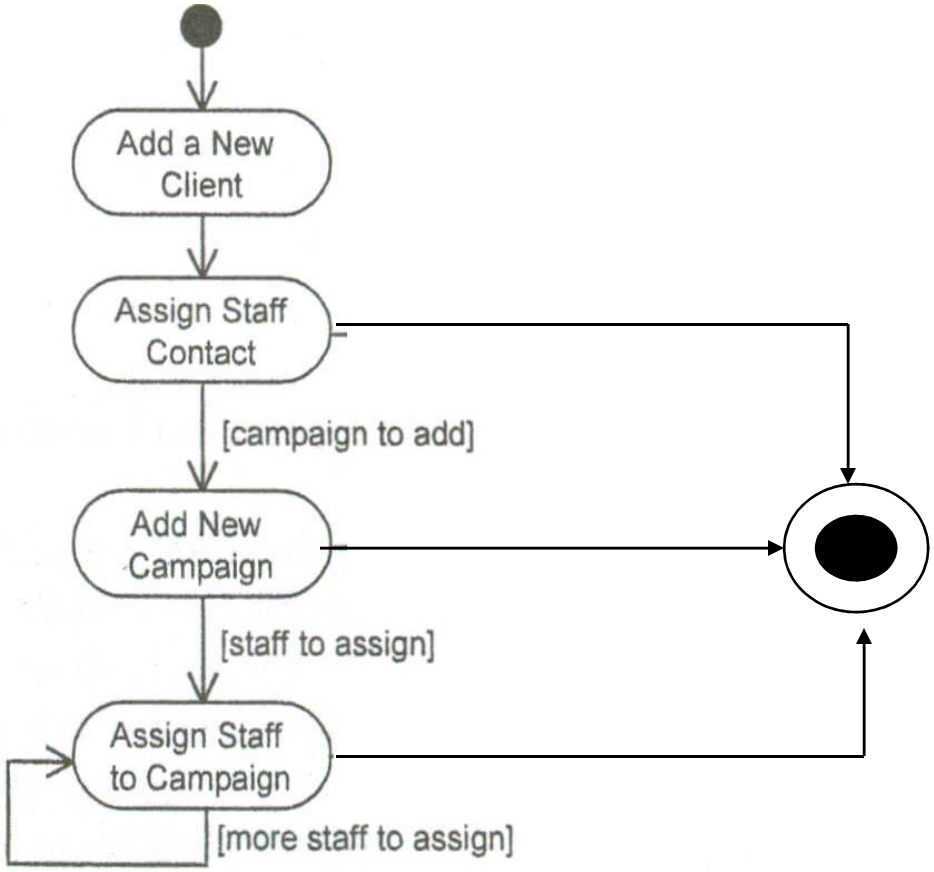


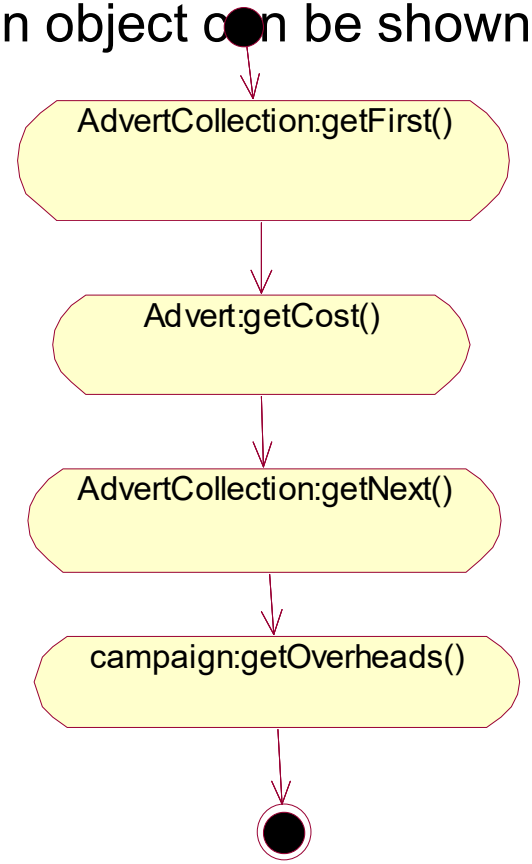
Figure 5.12 Activity diagram with start state.

This diagrams shows Activity diagram along with Start point.

Activity diagrams make it possible to represent the three structural components of all procedural programming languages: se-quences, selections and iterations. This ability to model processes in this way is particularly useful for modelling business procedures, but can also be helpful in modelling the operations of classes.

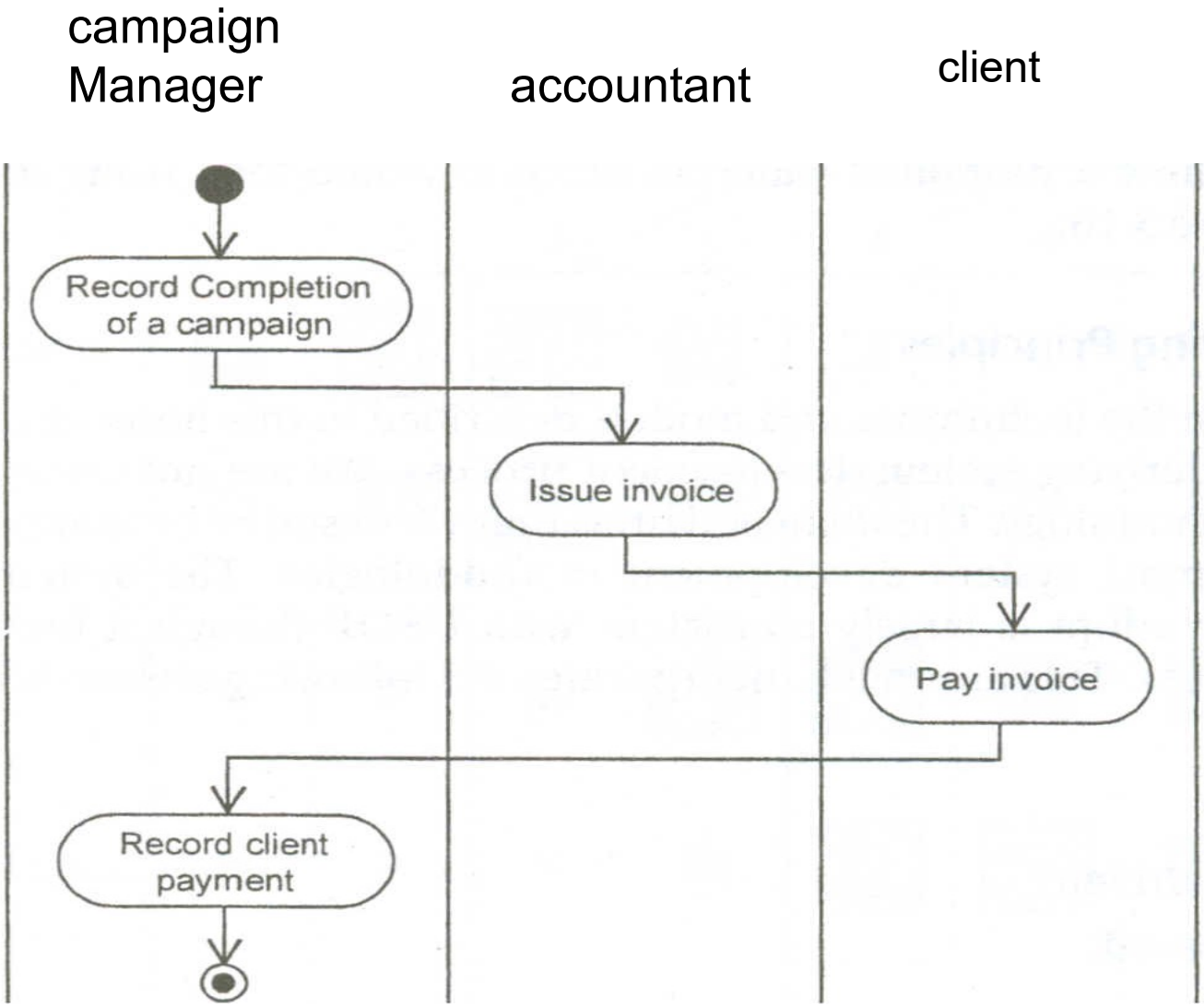
In an object-oriented system, however, the focus is on objects carrying out the processing necessary for the overall system to achieve its objectives. There are two ways in which objects can be shown in activity diagrams:

- the operation signature of an object can be used as the name of an activity;
- an object can be shown as providing the input to or output of an activity.



the figure shows Activity diagram with Object Operation signatures as activities

Activity diagram with swimlanes -



Development Process :

A development process should specify what has to be done, when it has to be done, how it should be done and by whom in order to achieve the required goal.

Project Management techniques are used to manage and control the process for individual projects. One of the software development processes currently in wide use is the Unified Software Development Process (USDP).

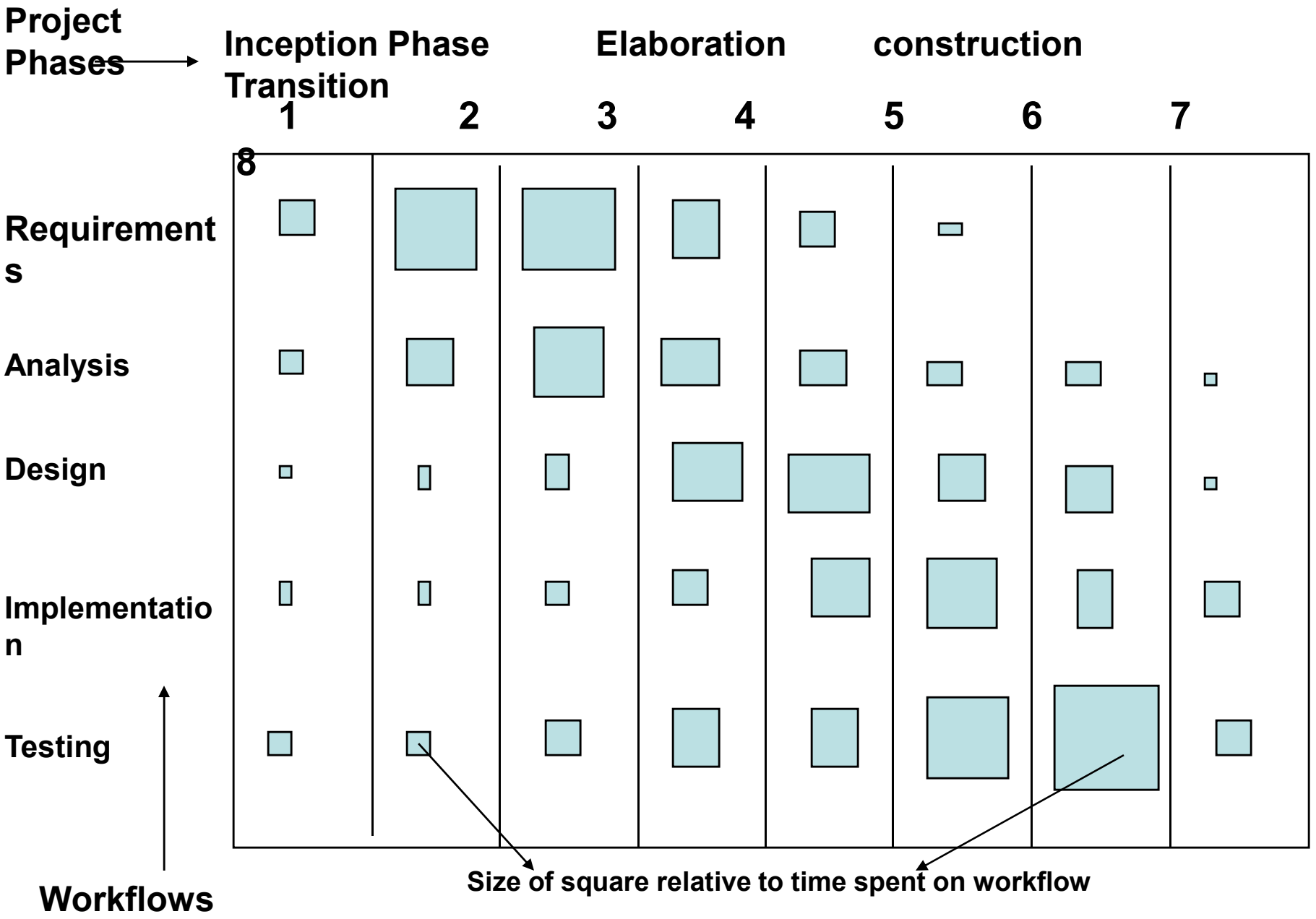
The USDP has been developed by the team that created UML. USDP contains the following development methods for information systems develop-ment.

These include:

- iterative and incremental development;
- component based development;
- requirements driven development;
- architecture centrism;
- visual modelling techniques.

The USDP does not follow the Traditional Life Cycle , but adopts an iterative approach within four main *phases.* These phases reflect the different emphasis on tasks that are necessary as systems development proceeds as shown in the following figure.

Phases and Workflows in USDP



These differences are captured in a series of *workflows* that run through the development process. Each workflow defines a series of activities that are to be carried out as part of the workflow and specifies the roles of the people who will carry out those activities. The difference with the waterfall lifecycle model is activities and phases are one and the same in Waterfall Life cycle Model, where as in iterative lifecycles like the USDP the activities are independent of the phases, and it is the mix of activities that changes as the project proceeds.

Underlying Principles :

USDP approach incorporates the following characteristics. It is
iterative;
incremental;
requirements driven;
component-based;
architectural.

Main activities :
The systems development process contains the following main activities:

1. Requirements Capture and Modelling;
2. Requirements Analysis;
3. System Design;
4. Class Design;
5. Interface Design;
6. Data Management Design;
7. Construction;
8. Testing;
9. Implementation.

These activities are interrelated and dependent upon each other. In a waterfall development process they would be performed in a sequence .This is not the case in an iterative development process, although some activities clearly precede others.

For example, at least some requirements capture and modelling must take place before any requirements analysis can be undertaken.

1. Requirements Capture and Modelling :

Various fact finding techniques are used to identify requirements. Requirements are documented in use cases. A use case captures an element of functionality and the requirements model may include many use cases.

For example, in the Agate case study the requirement that the accountant should be able to record the details of a new member of staff on the system is an example of a use case. It would be described initially as follows. Use Case: 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 recorded. Start date defaults to today's date. The starting grade is recorded.

The use cases can also be modelled graphically. The use case model is refined to identify common procedures and dependencies between use cases. The objective of this refinement is to produce a essential or needed requirements but complete description of requirements.

Prototypes of some key user interfaces may be produced in order to help to understand the requirements that the users have for the system. An initial system architecture may be developed to help guide subsequent steps during the development process. This initial architecture will be refined and adjusted as the development proceeds.

2. Requirements Analysis :

Essentially each use case describes one or more requirement. Each use case is analysed separately to identify the objects that are required to support it.

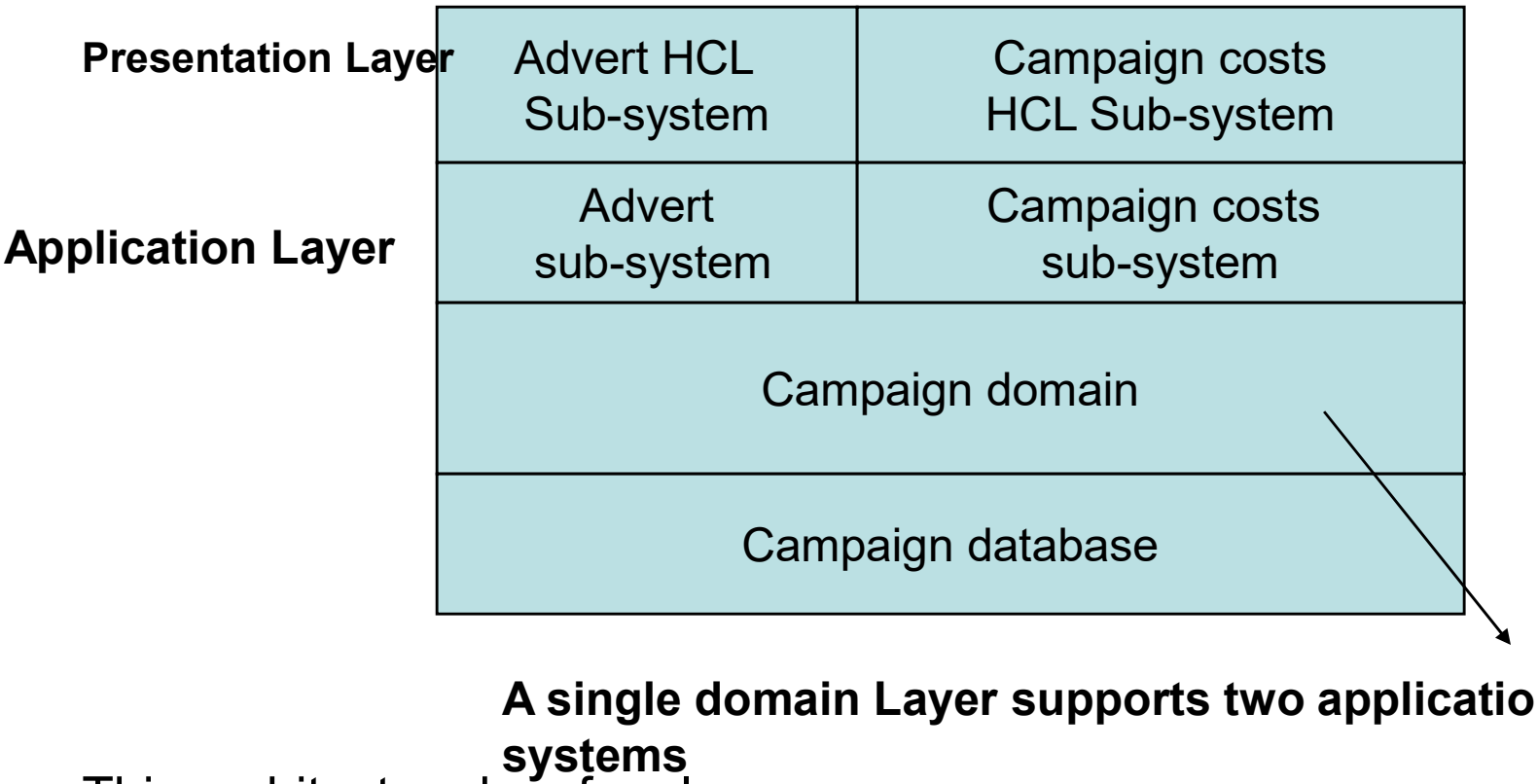
The use case is also analysed to determine how these objects interact and what responsibilities each of the objects has in order to support the use case. Collaboration diagrams are used to model the object interaction. The models for each use case are then integrated to produce an analysis class diagram, The initial system architecture may be refined as a result of these activities.

3. System Design :

During Design stage various decisions concerning the design process are made including the further specification of a suitable systems architecture.

System design is here concerned with identifying and documenting suitable development standards (e.g. interface design standards, coding standards, Quality standards…) for the remainder of the project.

For example, a possible architecture for the system in the Agate case study is shown below.



A single domain Layer supports two application sub-systems

This architecture has four layers.

The two bottom layers provide common functionality and database access for the staffing and advert planning sub-systems. Part of the architectural specification may include the identification of particular technologies to be used. In this case it may be decided to use a client-server architecture with the sub-system interfaces operating through a web browser to give maximum operational flexibility.

4 .Class Design :

Each of the use case analysis models is now elaborated separately to include relevant design detail. Interaction sequence diagrams may be drawn to show detailed object communication and state diagrams may be prepared for objects with complex state behaviour .

The separate models are then integrated to produce a detailed design class diagram. Design classes have attributes and operations specified to replace the less specific responsibilities that may have been identified by the analysis activity .

The detailed design of the classes normally necessitates the addition of further classes to support, for example, the user interface and access to the data storage system.

5. Interface Design :

The nature of the functionality offered via each use case has been defined in requirements analysis.

Interface design produces a detailed specification as to how the required functionality can be realized. Interface design gives a system its look and feel and determines the style of interaction the user will have.

It also includes the positioning and colour of buttons and fields, the mode of navigation used between different parts of the system and the nature of online help.

6. Data Management Design :

Data management design focuses on the specification of the mechanisms suitable for implementation of the database management system being used .

Techniques such as normalization and entity-relationship modelling may be particularly useful if a relational database management system. Data management design and class design are interdependent.

7. Construction :

Construction is concerned with building the application using appropriate development technologies. Different parts of the system may be built using different languages. Java might be used to construct the user interface, while a database management system such as Oracle would manage data storage and handle commonly used processing routines.

***8. Testing** :*

Before the system can be delivered to the client it must be thoroughly tested. Testing scripts should be derived from the use case descriptions that were previously agreed with the client. Testing should be performed as elements of the system are developed.

9. Implementation :

The final implementation of the system will include its installation on the various computers that will be used. It will also include managing the transition from the old systems to the new systems for the client.

This will involve careful risk management and staff training.

6. REQUIREMENTS CAPTURE

USER REQUIREMENTS :

The purpose of developing a new information system must be to produce something that meets the needs of the people who will be using it.

In order to do this, we must have a clear understanding both of the overall objectives of the business and of what it is that the individual users of the system are trying to achieve in their jobs. Unless you are in the rare position of developing a system for a new organization, you will need to understand how the business is operating at present and how people are working now.

Many aspects of the current system will need to be carried forward into the new system, so it is important that information about what people are doing is gathered and documented. These are the requirements that are derived from the 'current system'.

The motivation for the development of a new information system is usually problems with the current system, so it is also essential to capture what it is that the users require of the new system that they cannot do with their existing system. These are the 'new requirements'.

Current system

The existing system may be a manual one, based on paper documents, forms and files; it may already be computerized; or it may be a combination of both manual and computerized elements. Whichever it is, it is reasonably certain that large parts of the existing system meet the needs of the people who use it, that it has to some extent evolved over time to meet business needs and that users are familiar and comfortable with it. It is almost equally certain that there are sections of the system that no longer meet the needs of the business, and that there are aspects of the business that are not dealt with in the existing system.

Here the analyst, who is responsible for gathering information as one of the first steps in developing a new system, gains a clear understanding of how the existing system works means what parts of the existing system will be carried forward into the new one. It is also important, because the existing system will have shortcomings and defects, which must be avoided or overcome in the new system.

It is not always easy or possible to replace existing systems. So called *legacy systems* may have been developed some time ago and may contain millions of lines of program code, which have been added to and amended over a period of time. One approach to dealing with such systems is to create new frontends, typically using modern graphical user interfaces and object-oriented languages, and *wrap* the legacy systems up in new software.

It is not always possible to leave legacy systems as they are and simply wrap them in new code. It was not possible to ignore the problems that faced companies at the turn of the century when it was realized that many systems were in danger of catastrophic collapse as a result of the decision to use two decimal digits to store the year. However, the process of changing the program code in such systems is a matter of understanding the internal working of existing systems rather than gathering information about the way the organization works and the way that people do their jobs.

To develop a new system, we need a case study can be made on the existing system to collect the requirements.

1.Some of the functionality of the existing system will be required in the new system.

2.Some of the data in the existing system is of value and must be migrated into the new system.

3.Technical documentation of existing computer systems may provide details of pro-
cessing algorithms that will be needed in the new system.

4.The existing system may have defects that we should avoid in the new system.
Studying the existing system will help us to understand the organization in general.

5.Parts of the existing system may be retained.

Information systems projects are now rarely 'green field' projects in which manual systems are replaced by new computer-ized systems; more often there will be existing systems with which interfaces must be established

New requirements : Reasons to collect new REQUIREMENTS -

• Most organizations now operate in an environment that is rapidly changing needs of the system.

• The relative strength of national economies around the world can change requirements of an existing system also.

• New technologies are introduced which change production processes, distribution networks and the relationship with the consumer; governments.

• A clear result of responding to a dynamic environment ,most of the organizations change their products and services and change the way they do business. The effect of this is to change their need for information.

• Even in less responsive organizations, information systems become outdated and need enhancing and extending.

• Mergers and demergers create the need for systems to be replaced. The process of replacement offers an opportunity to extend the

Many organizations are driven by internal factors to grow and change the ways in which they operate, and this too provides a motivation for the development of new information systems.

Whether you are investigating the working of the existing system or the requirements for the new system, the information collected for a developing new system will fall into one of three categories:

1. ‘Functional requirements',
2. ‘Non-functional requirements'
3. ‘Usability requirements'.

Functional and non-functional requirements are conventional categories in systems analysis and design, while usability is often ignored in systems development projects.

Functional requirements

Functional requirements describe what a system does or is expected to do, often referred to as its *functionality.* In the object-oriented approach, we shall initially employ use cases to document the functionality of the system. As we progress into the analysis stage, the detail of the functionality will be recorded in the data that we hold about objects, their attributes and operations.

Functional requirements include the following.

1. Descriptions of the processing that the system will be required to carry out.

2. Details of the inputs into the system from paper forms and documents, from inter-actions between people, such as telephone calls, and from other systems.

3. Details of the outputs that are expected from the system in the form of printed documents and reports, screen displays and transfers to other systems.

Non-functional requirements

Non-functional requirements are those that describe aspects of the system that are concerned with how well it provides the functional requirements. These include the following.

1. Performance criteria such as desired response times for updating data in the system or retrieving data from the system.

2. Anticipated volumes of data, either in terms of throughput or of what must be stored.

3. Security considerations.

Usability requirements

Usability requirements are those that will enable us to ensure that there is a good match between the system that is developed and both the users of that system and the tasks that they will undertake when using it.

The International Standards Organization (ISO) has defined the usability of a product as 'the degree to which specific users can achieve specific goals within a particular environment; effectively, efficiently, comfortably and in an acceptable manner'.

In order to build usability into the system from the outset, we need to gather the following types of information.

1.Characteristics of the users who will use the system.

2.The tasks that the users undertake, including the goals that they are trying to achieve.

3.Situational factors that describe the situations that could arise during system use

Fact finding techniques :

There are five main fact finding techniques that are used by analysts to investigate requirements.

1. Background Reading.
2. Interviewing.
3. Observation.
4. Document sampling.
5. Questionnaires .

1 Background reading

If an analyst is assigned within the organization that is the subject of the fact gathering exercise, then he or she will already have a good under standing of the organization and its business objectives.

If, however, he or she is going in as an outside consultant, then one of the first tasks is to try to gain an understanding of the organization.

Background reading or research is part of that process.

The kind of documents that are suitable sources of information include the following:

company reports,
organization charts,
policy manuals,
job descriptions,
Reports and
documentation of existing systems.

Reading company reports may provide the analyst with information about the organization's mission, and so possibly some indication of future requirements, this technique mainly provides information about the current system.

Advantages and disadvantages

+ Background reading helps the analyst to get an understanding of the organization before meeting the people who work there.

+ It also allows the analyst to prepare for other types of fact finding, for example, by being aware of the business objectives of the organization.

+ Documentation on the existing system may provide formally defined information requirements for the current system.

- Written documents often do not match up to reality; they may be out of date or they may reflect the official policy on matters which deals differently in practice.

Appropriate situations

- Background reading is appropriate for projects where the analyst is not familiar with the organization being investigated.

- It is useful in the initial stages of investigation.

2 Interviewing

Interviewing is probably the most widely used fact finding technique; it is also the one that requires the most skill and sensitivity.

Interviews can be used to gather information from management about their objectives for the organization and for the new information system, from staff about their existing jobs and their information needs, and from customers and members of the public as possible users of systems. While conducting an interview, the analyst can also use the opportunity to gather documents that the interviewee uses in his or her work.

Guidelines on Interviewing :

conducting an interview requires good planning, good interpersonal skills and an alert and responsive frame of *mind. These guidelines are the important points while planning and conducting an interview.*

Before the interview :

You should always make appointments for inter-views in advance. You should give the interviewee information about the likely duration of the interview and the subject of the interview.

Being interviewed takes people away from their normal work. Make sure that they feel that it is time well spent.

It is conventional to obtain permission from an interviewee's line manager before interviewing them. Often the analyst interviews the manager first and uses the opportunity to get this per-mission.

In large projects, an interview schedule should be drawn up showing who is to be interviewed, how often and for how long. Initially this will be in terms of the job roles of interviewees rather than named individuals. It may be the manager who decides which individual you interview in a particular role.

At the start of the interview :

Introduce yourself and the purpose of the interview. Arrive on time for interviews and stick to the planned timetable do not overrun.

Ask the interviewee if he or she minds you taking notes or tape-recording the interview. Even if you tape record an interview, you are advised to take notes.

During the interview :

Take responsibility for the agenda. You should control the direction of the interview. This should be done in a sensitive way. If the interviewee is getting away from the subject, bring them back to the point. If what they are telling you is important, then say that you will come back to it later and make a note to remind yourself to do so.

Use different kinds of question to get different types of information. Questions can be open ended

After the interview :

Thank the interviewee for their time. Make an appointment for a further interview if it is necessary. Offer to provide them with a copy of your notes of the interview for them to check that you *have* accurately recorded what they told you.
Transcribe your tape or write up your notes as soon as possible after the interview while the content is still fresh in your mind.

A systems analysis interview is a structured meeting between the analyst and an interviewee who is usually a member of staff of the organization being investigated. The interview may be one of a series of interviews that range across different areas of the interviewee's work or that probe in progressively greater depth about the tasks undertaken by the interviewee. The degree of structure may vary: some interviews are planned with a fixed set of questions that the interviewer works through, while others are designed to cover certain topics but will be open-ended enough to allow the interviewer to pursue interesting facts as they emerge. The ability to respond flexibly to the interviewee's responses is one of the reasons why interviews are so widely used.

Advantages and disadvantages

+ Personal contact allows the analyst to be responsive and adapt to what the user says. Because of this, interviews produce high quality information.

+ The analyst can investigate in greater depth about the person's work than can be achieved with other methods.

+ If the interviewee has nothing to say, the interview can be terminated.

- Interviews are time-consuming and can be the most costly form of fact gathering.

- Interview results require the analyst to work on them after the interview: the transcription of tape recordings or writing up of notes.

- Interviews can be subject to bias if the interviewer has a closed mind about the problem.
- If different interviewees provide conflicting information, it can be

Appropriate situations

Interviews are appropriate in most projects. They can provide information in depth about the existing system and about people's requirements for a new system.

3 Observation

Watching people carrying out their work in a natural setting can provide the analyst with a better understanding of the job than interviews, in which the interviewee will often concentrate on the normal aspects of the job and forget the exceptional situations and interruptions which can occur with the system and how to cope up with those problems.

Observation also allows the analyst to see what information people use to carry out their job. This can tell you about the documents they refer to, whether they have to get up from their desks to get information, how well the existing system handles their needs.

People who are not good at estimating quantitative data, such as how long they take to deal with certain tasks, and observation with a stopwatch can give the analyst lots of quantitative data, not just about typical times to perform a task but also about the statistical distribution of those times.

Observation can be an open-ended process in which the analyst simply sets out to observe what happens and to note it down, or it can be a closed process in which the analyst wishes to observe specific aspects of the job and draws up an observation schedule or form on which to record data.

This can include the time it takes to carry out a task, the types of task the person is performing or factors such as the number of errors they make in using the existing system as a baseline for usability design.

Advantages and disadvantages

+ Observation of people at work provides first hand experience of the way that the current system operates.
+ Data are collected in real time and can have a high level of validity if care is taken in how the technique is used.
+ Observation can be used to verify information from other sources or to look for exceptions to the standard procedure.
+ Baseline data about the performance of the existing system and of users can be collected.

-Most people do not like being observed and are likely to behave differently from the way in which they would normally behave. This can distort findings and affect the validity.

-Observation requires a trained and skilled observer for it to be most effective.

- There may be logistical problems for the analyst, for example, if the staff to be observed work shifts or travel long distances in order to do their job.

- There may also be ethical problems if the person being observed deals with sensitive private or personal data or directly with members of the public.

Appropriate situations

- Observation is essential for gathering quantitative data about people's jobs.

- It can verify or disprove assertions made by interviewees, and is often useful in situations where different interviewees have provided conflicting information about the way the system works.

- Observation may be the best way to follow items through some kind of process from start to finish.

4 Document sampling

Document sampling can be used in two different ways.

First, the analyst will collect copies of blank and completed documents during the course of interviews and observation sessions. These will be used to determine the information that is used by people in their work, and the inputs to and outputs from processes which they carry out, either manually or using an existing computer system. From an existing system, the analyst may need to collect screen shots in order to understand the inputs and outputs of the existing system.

Second, the analyst may carry out a statistical analysis of documents in order to find out about patterns of data. For example, many documents such as order forms contain a header section and a number of lines of detail. The analyst may want to know the distribution of the number of lines in an order. This will help later in estimating volumes of data to be held in the system and in deciding how many lines should be displayed on screen at one time. While this kind of statistical sampling can give a picture of data volumes, the analyst should be alert to seasonal patterns of activity, which may mean that there are peaks and troughs in the amount of data being processed.

Advantages and disadvantages

\+ Can be used to gather quantitative data, such as the average number of lines on an invoice.
\+ Can be used to find out about error rates in paper documents.

\- If the system is going to change dramatically, existing documents may not reflect how it will be in future.

Appropriate situations

The first type of document sampling is almost always appropriate. Paper-based documents give a good idea of what is happening in the current system. They also provide supporting evidence for the information gathered from interviews or observation.

The statistical approach is appropriate in situations where large volumes of data are being processed, and particularly where error rates are high, and a reduction in errors is one of the criteria for usability.

5 Questionnaires

Questionnaires are a research instrument that can be applied to fact finding in system development projects. They consist of a series of written questions. The questionnaire designer usually limits the range of replies that respondents can make by giving them a choice of options.

YES/NO questions only give the respondent two options. If there are more options, the multiple choice type of question is often used when the answer is factual, whereas scaled questions are used if the answer involves an element of subjectivity. Some questions do not have a fixed number of responses, and must be left open-ended for the respondent to enter what they like. Where the respondent has a limited number of choices, these are usually coded with a number, which speeds up data entry if the responses are to be analysed by computer software. If you plan to use questionnaires for requirements gathering, they need very careful design.

Advantages and disadvantages

+ An economical way of gathering data from a large number of people.

+ If the questionnaire is well designed, then the results can be analyzed easily.

-Good questionnaires are difficult to construct.

- There is no automatic mechanism for follow up or probing more deeply, although
it is possible to follow up with an interview by telephone or in person if necessary.

-Postal questionnaires suffer from low response rates.

User Involvement

The success of a systems development project depends not just on the skills of the team of analysts, designers and programmers who work on it, or on the project management skills of the project manager, but on the effective involvement of users in the project at various stages of the life cycle.

The term *stakeholders* was introduced to describe all those people who have an interest in the successful development of the system. Stakeholders include all people who stand to gain (or lose) from the implementation of the new system: users, managers and budget-holders. Analysts deal with people at all levels of the organization. In large projects it is likely that a steering committee with delegated powers will be set up to manage the project from the users' side.

This will include the following categories of people:

1. senior management-with overall responsibility for running the organization,
2. financial managers with budgetary control over the project,

Users will be involved in different roles during the course of the project as:

1. subjects of interviews to establish requirements,
2. representatives on project committees,
3. those involved in evaluating prototypes,
4. those involved in testing,
5. subjects of training courses
6. end-users of the new system.

Documenting Requirements

Information systems professionals need to record facts about the organization they are studying and its requirements. As soon as the analysts start gathering facts, they will need some means of documenting them.

Systems analysts and designers model the new system in a mixture of diagrams and text. The important thing is while implementing a project , it must to employ some set of standards , These may be the agreed standards of the organization carrying out the analysis and design project or they may be a requirement of the organization that is having the work done.

We are using UML to produce models of the system from different perspectives. Computer Aided Software Engineering (CASE) tools are normally used to draw the diagrammatic models and to maintain in a repository the associated data about the various things that are shown in the diagrams.

However, there will also be other kinds of documents, not all of which fit into the UML framework. In large-scale projects a librarian or configuration manager may be required to keep track of these documents and ensure that they are stored safely and in a way that enables them to be retrieved when required.

Such documents include the following:

- records of interviews and observations,
- details of problems,
- copies of existing documents and where they are used,
- details of requirements,
- details of users and
- minutes of meetings.

In many projects, these documents will be stored digitally, using a document management system or a version control system. In this case, many people can access the same document simultaneously. The system enforces control over whether a document can be updated, and ensures that no more than one person at a time is able to 'check out' a document in order to use it.

Not all of the documents listed above represent requirements, and it is necessary to maintain some kind of list or database of requirements. There are software tools available to hold requirements in a database, and some can be linked to CASE tools and testing tools. This makes it possible to trace from an initial requirement through the analysis and design models to where it has been implemented and to the test cases that test whether the requirement has been met.

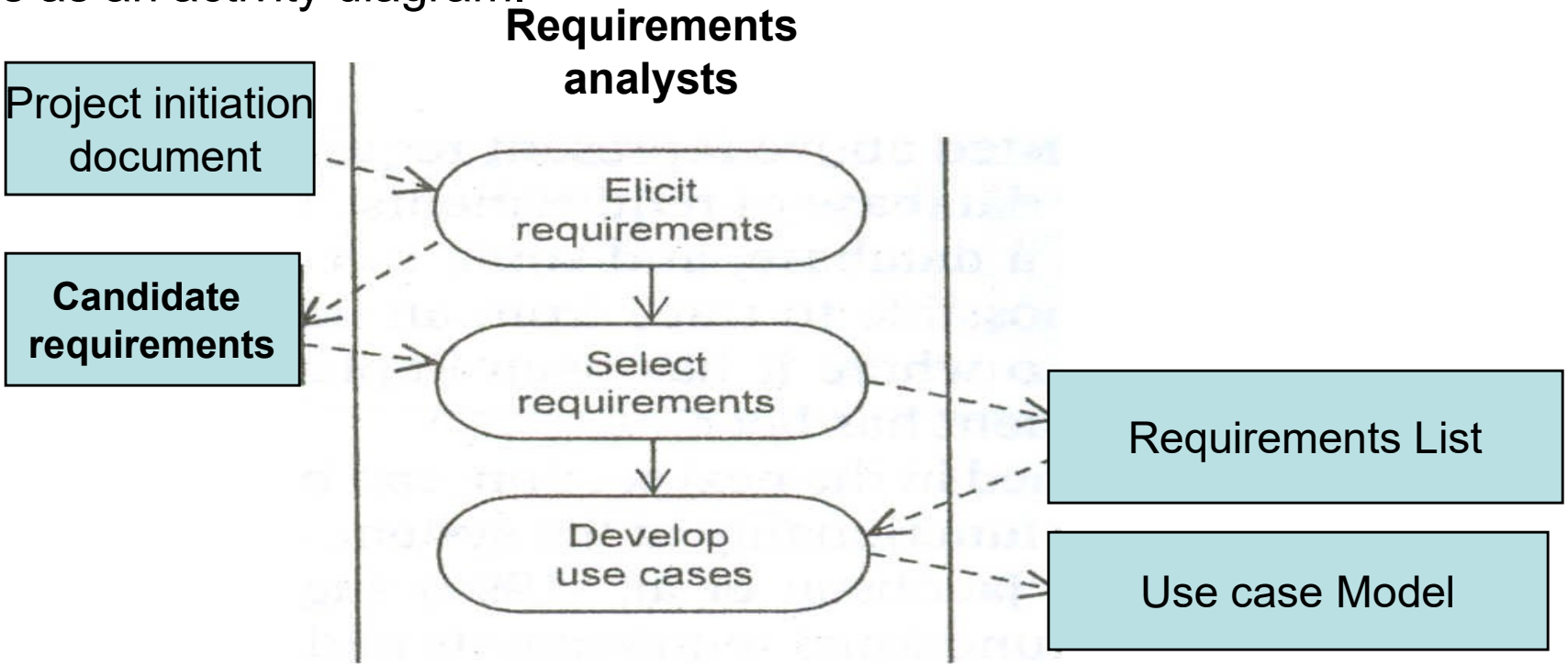
Use cases, can be used to model requirements, but because they focus on the functionality of the system, are not good for documenting non-functional requirements. Jacobson suggest that the use case model should be used to document functional requirements and a separate list of 'supplemen-tary requirements' (those not provided by a use case) should be kept. They say that together, the use case model and the list of supplementary requirements constitute a traditional requirements specification.

finally we can conclude, use cases can be used to model functional requirements, but a separate list of requirements should be kept, containing all requirements functional and non-functional for the system.

Where there is a relationship between a particular use case and a particular requirement, this should be recorded. Moreover, some requirements describe very high-level units of behaviour and may need to be broken down into low-level requirements that describe more precisely what is to be done. Any database of requirements should make it possible to hold this kind **of** hierarchical structure of requirements.

Sometimes the process of requirement gathering throws up more requirements than can be met in a particular project. They may be outside the scope of the project, too expensive to implement or just not really necessary at this point in time.

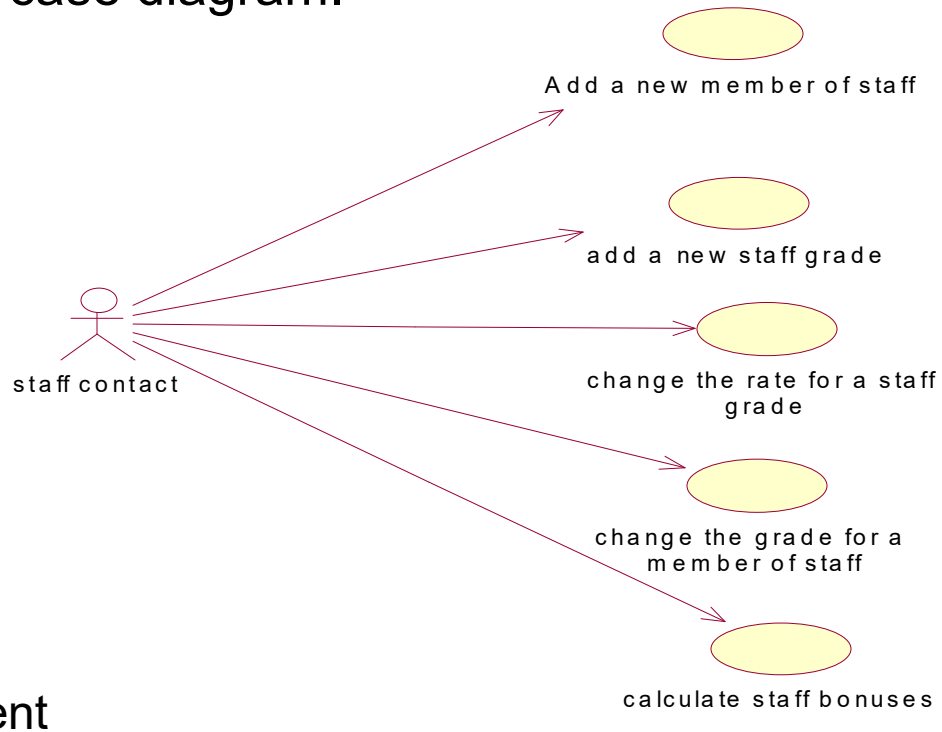
The process of building a requirements model for a system involves going through all the candidate requirements to produce a list of those that will be part **of** the current project. The following Figure shows this as an activity diagram.



USE CASES

Use cases are descriptions of the functionality of the system from the users' pers-pective.

Use case diagrams are used to show the functionality that the system will provide and to show which users will communicate with the system in some way to use that functionality. The following figure shows an example of a use case diagram.



Staff Management

Purpose Of Use case diagrams

The use case model is called as the requirements model; which also include a problem domain object model and user interface descriptions in this requirements model.

Use cases specify the functionality that the system will offer from the users' perspective. They are used to document the scope of the system and the developer's understanding of what it is that the users require.

Use cases are supported by *behaviour specifications.* These specify the behaviour of each use case either using UML diagrams, such as *collaboration diagrams* or *sequence diagrams* or in text form as *use case descriptions.*

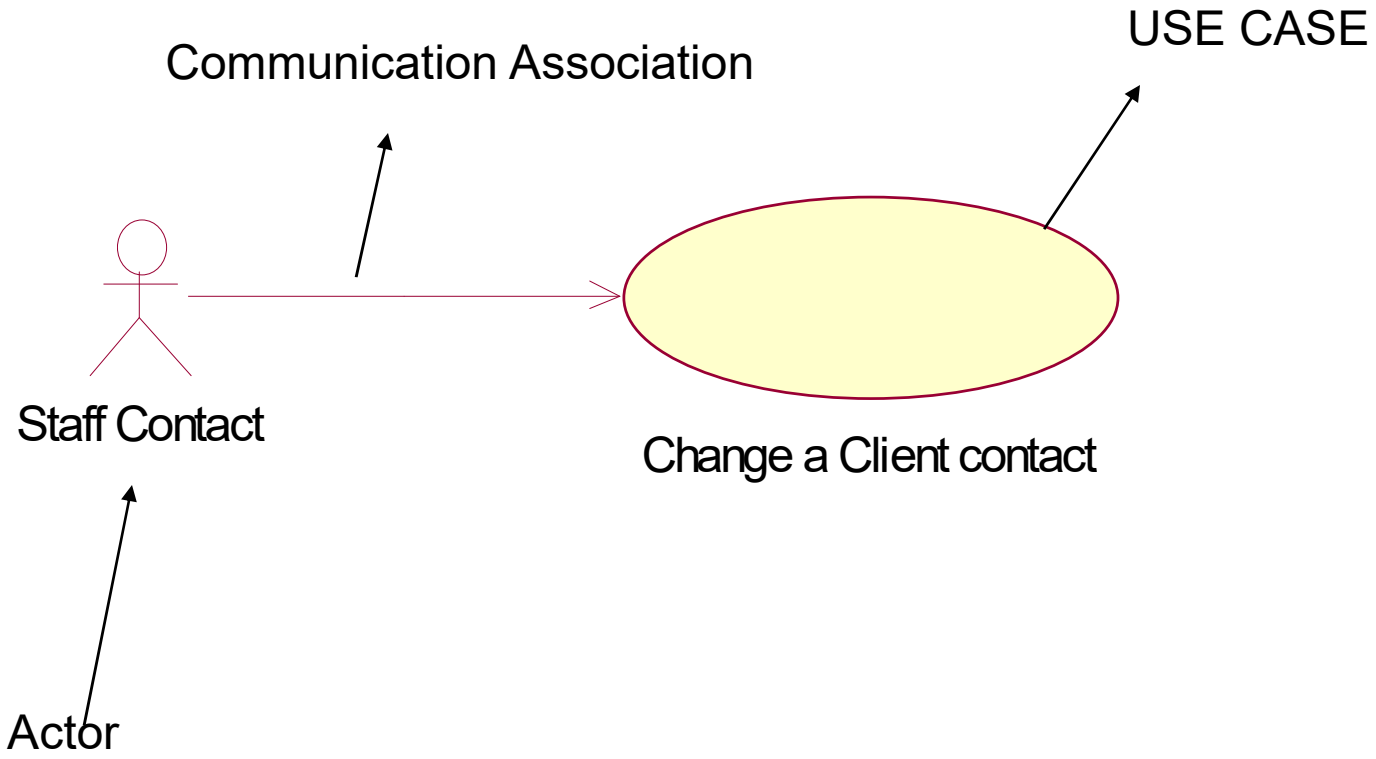
Textual *use case descriptions* provide a description of the interaction between the users of the system called *actors,* and the high level functions within the system called use cases.

These descriptions can be in summary form or in a more detailed form in which the interaction between actor and use case is described in a step-by-step way. In all three representations, the use case describes the interaction as the user sees it, and is not a definition of the internal processes within the system, or some kind of program specification.

Notation of USE CASE Diagrams

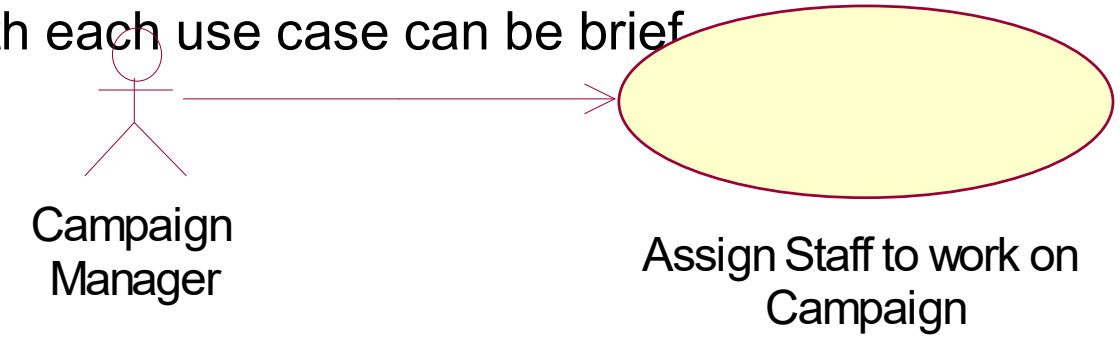
Use case diagrams show three aspects of the system: actors, use cases and relationships of the system or sub-system boundary.

The following Figure shows the elements of the notation.



Actors represent the roles that people, other systems or devices take on when communicating with the particular use cases in the system. In the Figure shows the actor Staff Contact in a diagram for the Agate case study.

In Agate, there is no job title Staff Contact: a director, an account manager or a member of the creative team can take on the role of being staff contact for a particular client company, so one actor can represent several people or job titles. Equally, a particular person or job title may be represented by more than one actor on use case diagrams. This is shown in the above and below together. A director or an account manager may be the Campaign Manager for a particular client campaign, as well as being the Staff Contact for one or more clients. The use case description associated with each use case can be brief.



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.

Alternatively, it can provide a step-by-step breakdown of the interaction between the user and the system for the particular use case. An example of this extended approach is provided below.

Assign staff to work on a *campaign*

Actor Action	System Response
1 . The actor enters the client name.	2.Lists all campaigns for that client.
3. Selects the relevant campaign.	4. Displays a list of all staff members not already allocated to this campaign.
5. Highlights the staff members to be assigned to this campaign.	6. Presents a message confirming that staff have been allocated.

The actor knows the campaign name and enters it directly.

Each use case description represents the usual way in which the actor will go through the particular transaction or function from end to end. Possible major alternative routes that could be taken are listed as *alternative courses.* The term *scenario* is used to describe use cases in which an alternative course is worked through in detail, including possible responses to errors. The use case represents the generic case, while the scenarios represent specific paths through the use case.

As well as the description of the use case itself, the documentation should include the purpose or intent of the use case, that is to say details of the task that the user is trying to achieve through the means of this use case.

One way of documenting use cases is to use a template .This might include the following sections:

- name of use case,
- pre-conditions (things that must be true before the use case can take place),
- post-conditions (things that must be true after the use case has taken place),
- Purpose (what the use case is intended to achieve) and
- description (in summary or in the format above).

Two further kinds of relationships can be shown on the use case diagram itself. These are the *Extend* and *Include* relationships.

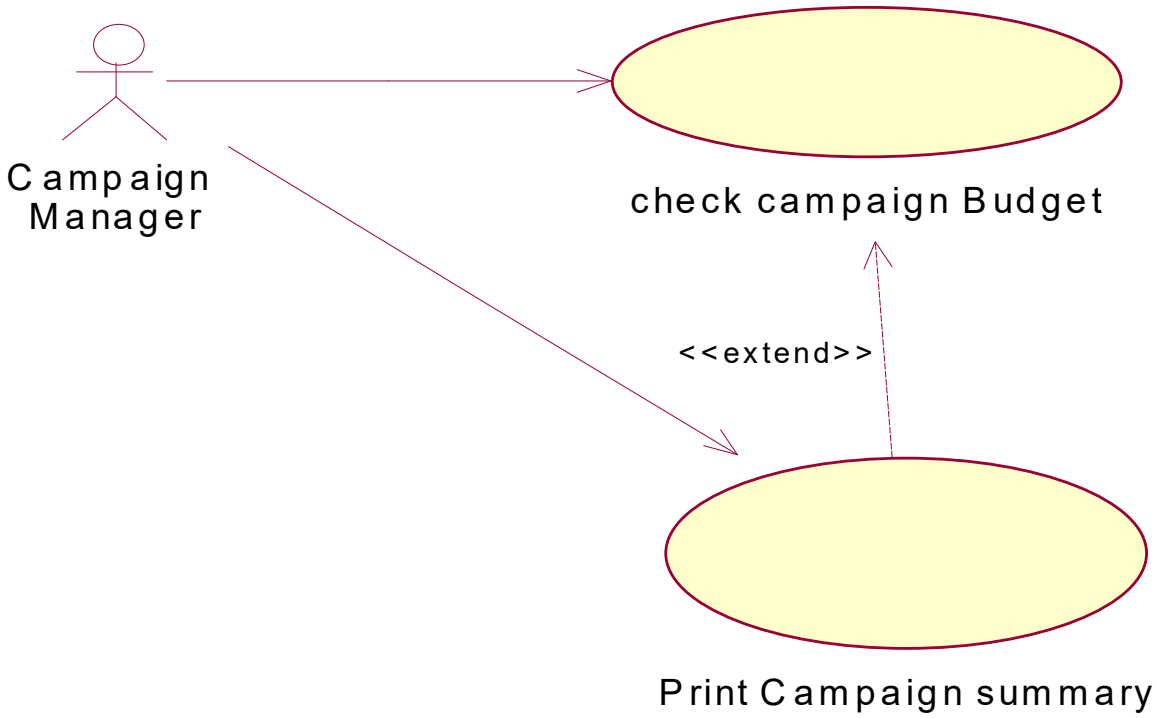
Dependencies - a dependency is a relationship between two modelling elements where a change to one will probably require a change to the other because the one is dependent in some way on the other. A dependency is shown by a dashed line with an open arrowhead pointing at the element on which the other is dependent. There are many kinds of dependencies in UML, and they are distinguished from one another using stereotypes.

Stereotypes - a stereotype is a special use of a model element that is constrained to behave in a particular way. Stereotypes can be shown by using a keyword, such as 'extend' or 'include' in matched *guillemets,* like «extend».

Stereotypes can also be represented using special icons. The actor symbol in use case diagrams is a stereotyped icon-an actor is a stereotyped class and could also be shown as a class rectangle with the stereotype «actor» above the name of the actor. So by stereotyping classes as «actor» we are indicating that they are a special kind of class that interacts with the system's use cases. Note, however, that actors are external to the system, unlike use cases and classes.

1. . «extend» is used when you wish to show that a use case provides additional functionality that may be required in another use case.

In the following Figure , the use case Print campaign summary extends Check campaign budget.



This means that at a particular point in Check Campaign Budget the user can optionally invoke the behaviour of Print campaign summary, which does something over and above what is done in Check campaign budget .

There may be more than one way of extending a particular use case, and these possibilities may represent significant variations on the way the user uses the system. Rather than trying to capture all these variations in one use case, you would document the core functionality in one and then extend it in others.

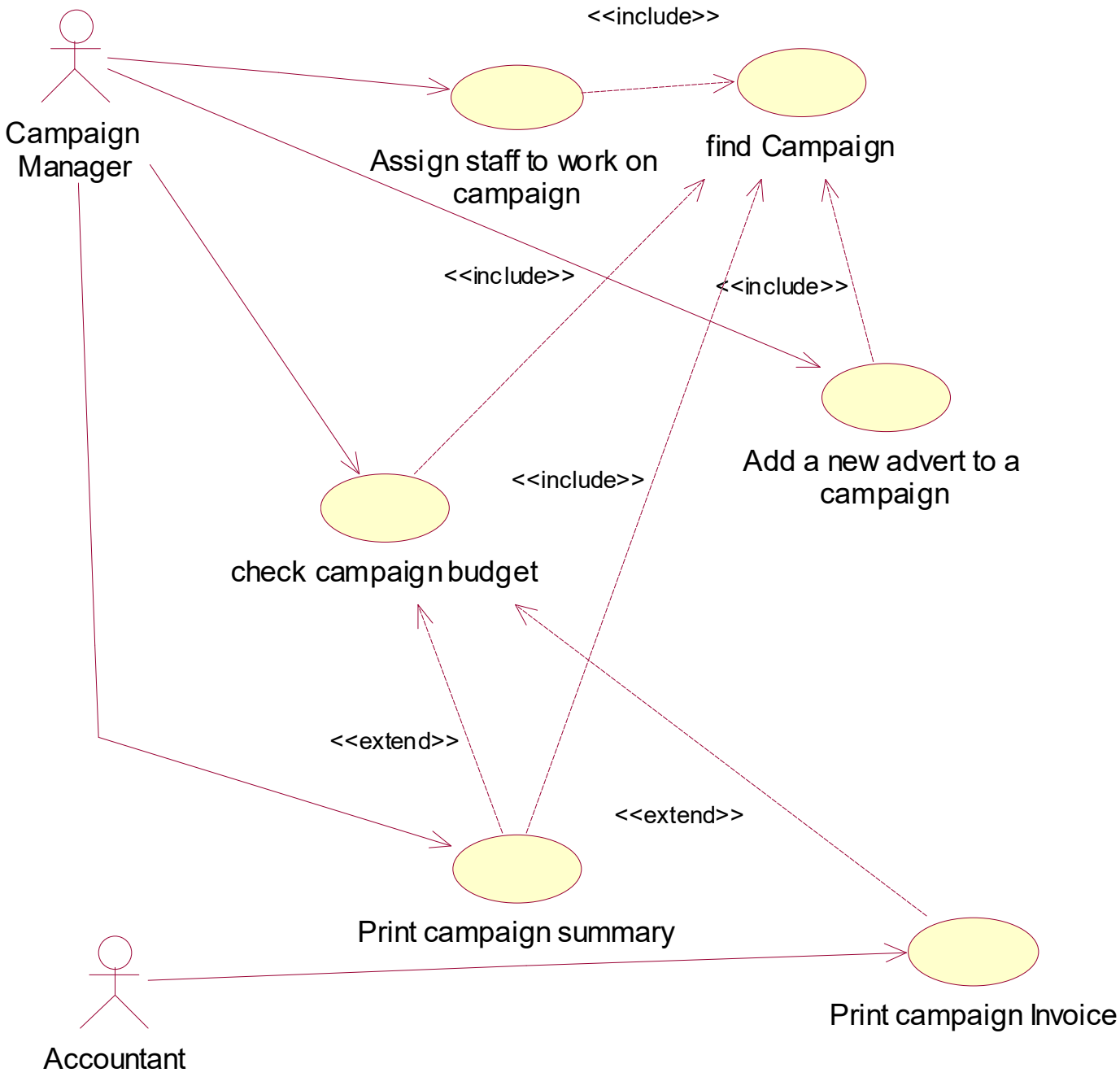
Extension points can be shown in the diagram, They are shown in a separate compartment in the use case ellipse, headed Extension points. The name of the extension point is given and a description of the point in the use case where it occurs. A condition can be shown next to the dependency relationship. This condition must be true for the extension to take place in a particular instance of the use case.

2. «include» applies when there is a sequence of behaviour that is used frequently in a number of use cases, and you want to avoid copying the same description of it into each use case in which it is used. The following Figure shows that the use case Assign staff to work on a campaign has an «include» relationship with Find campaign.

This means that when an actor uses Assign staff to work on a campaign the behaviour of Find campaign will also be included in order to select the relevant Campaign.

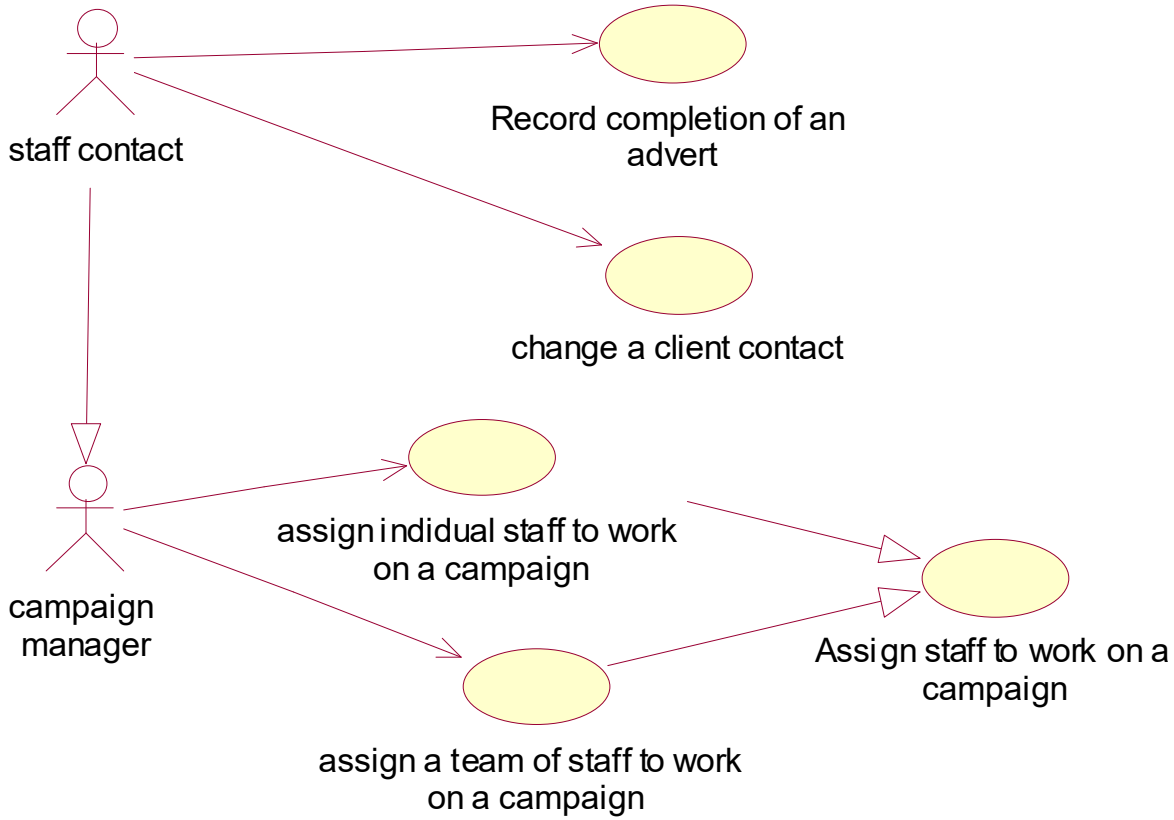
Its clear that actors need not be human users of the system. They can also be other systems that communicate with the one that is the subject of the systems development project, for example, other computers or automated machinery or equipment.

The following Figure shows a use case diagram for the Campaign Management subsystem with both Extend and Include relationships. Note that you do not have to show all the detail of the extension points on a diagram: the Extension points compartment in the use case can be suppressed. if you are using a CASE tool to draw and manage the diagrams, you may be able to toggle the display of this compartment on and off, and even if the information is not shown on a particular diagram, it will still be held in the CASE tool's repository.


Campaign Manager
<<include>>
Assign staff to work on campaign
find Campaign
<<include>>
<<include>>
Add a new advert to a campaign
<<include>>
check campaign budget
<<extend>>
<<extend>>
Print campaign summary
Accountant
Print campaign Invoice

Generalization and Specialization can be applied to actors and use cases.

For example, suppose that we have two actors, Staff Contact and Campaign Manager, and a Campaign Manager can do everything that a Staff Contact can do, and more. Rather than showing communication associations between Campaign Manager and all the use cases that Staff Contact can use, we can show Campaign Manager as a specialization of Staff Contact, as shown below.



Similarly, there may be similar use cases where the common functionality is best represented by generalizing out that functionality into a 'super-use case' and showing separate.

For example, we may find that there are two use cases at Agate Assign individual staff to work on a campaign, and Assign team of staff to work on a campaign, which are similar in the functionality they offer. We might abstract out the commonality into a use case Assign staff to work on a campaign, but this will be an abstract use case. It helps us to define the functionality of the other two use cases, but no instance of this use case will ever exist in its own right. This is also shown in Figure .

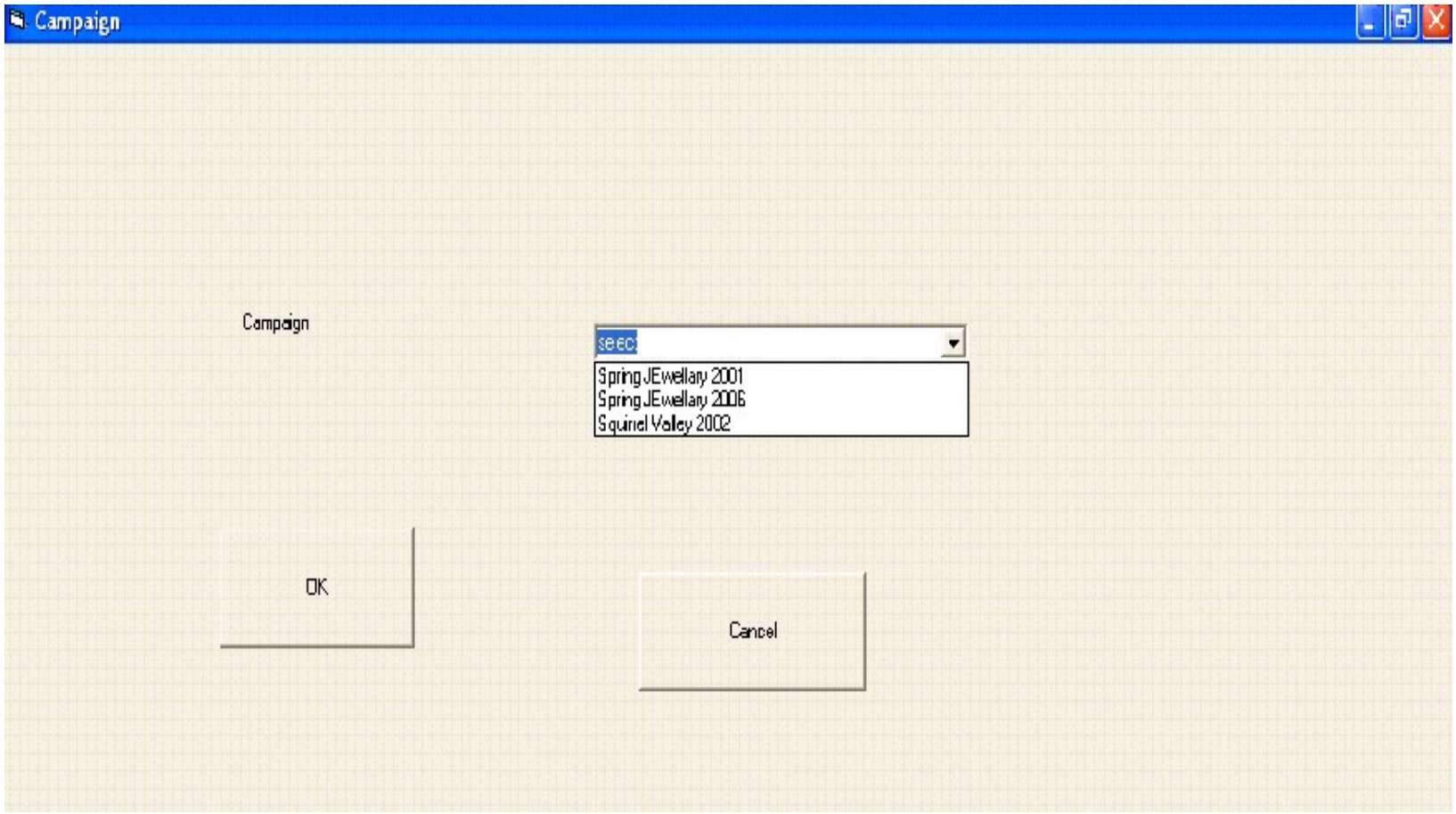
Supporting use cases with prototyping

As the requirements for a system emerge in the form of use cases, it is sometimes helpful to build simple prototypes of how some of the use cases will work. A prototype is a working model of part of the system- usually a program with limited functionality that is built to test out some aspect of how the system will work.

Prototypes can be used to help elicit requirements. Showing users how the system might provide some of the use cases often produces a stronger reaction than showing them a series of abstract diagrams. Their reaction may contain useful information about requirements.

For example, there are a number of use cases in the Campaign Management sub-system for Agate that require the user to select a campaign in order to carry out some business function. The Previous use case diagram reflects this in the «include» relationships with the use case Find campaign. The use case Find campaign will clearly be used a great deal, and it is worth making sure that we have the requirements right.

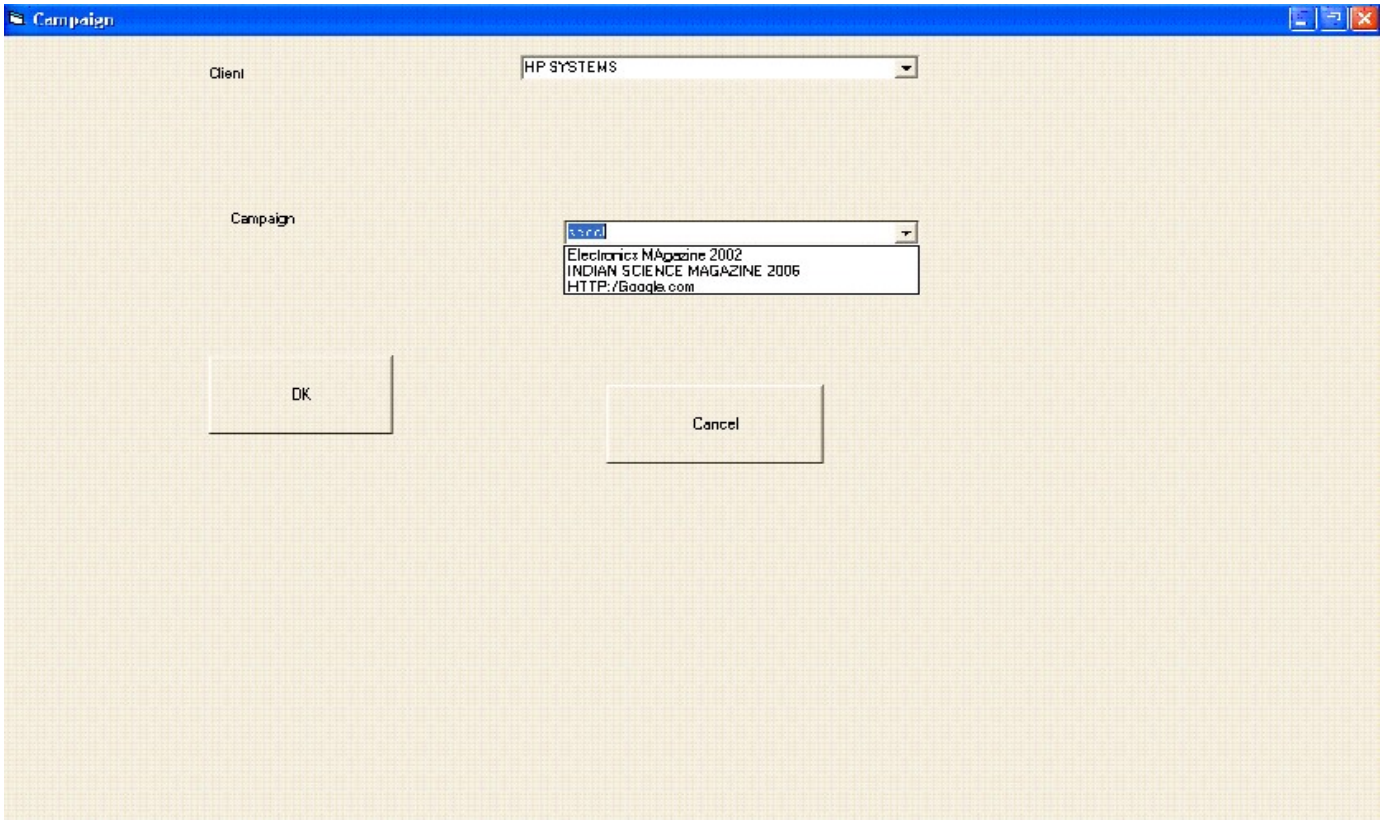
A prototype could be produced that provides a list of all the campaigns in the system. A possible version of this is shown in the following Figure.



Prototype Interface for the Find Campaign USE CASE

Giving this prototype interface design to the users may well produce the response that this way of finding a campaign will not work. There may be hundreds of campaigns in the system, and scrolling through them would be tedious. Different clients may have campaigns with similar names, and it would be easy to make a mistake and choose the wrong campaign if the user does not know which client it belongs to.

For these reasons, the users might suggest that the first step is to find the right client and then only display the campaigns that belong to that client. This leads to a different user interface as shown below.



Revised Prototype Interface for The FIND CAMPAIGN USE CASE

The information from this prototyping model forms part of the requirements for the system. This particular requirement is about usability, but it can also contribute to meeting other, non-functional requirements concerned with speed and the error rate: it might be quicker to select first the client and then the campaign from a short-list than it is to search through hundreds of campaigns; and it might reduce the number of errors made by users in selecting the right campaign to carry out some function on.

Prototypes can be produced with visual programming tools, with scripting languages like TCLITK, with a package like Microsoft PowerPoint@ or even as web pages using HTML.

Prototypes do not have to be developed as programs. Screen and window designs can be sketched out on paper and shown to the users, either formally or informally.

CASE tool support :

Drawing any diagram and maintaining the associated documentation is made easier by a CASE tool.As well as allowing the analyst to produce diagrams showing all the use cases in appropriate subsystems, a CASE tool should also provide facilities to maintain the repository associated with the diagram elements, and to produce reports.

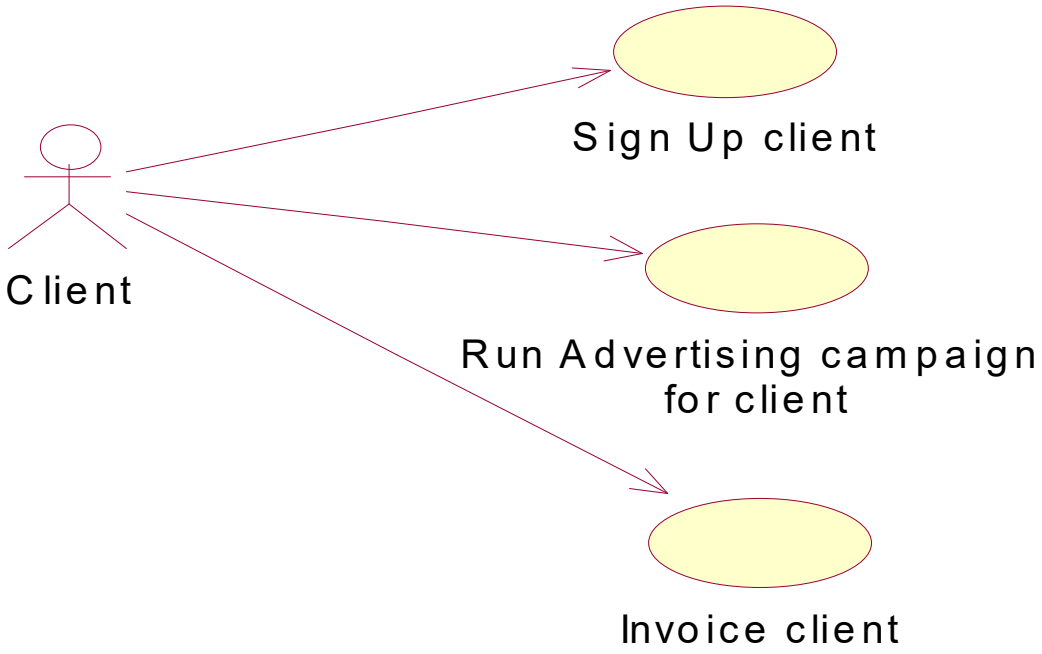
Automatically generated reports can be merged into documents that are produced for the client organization. The behaviour specification of each use case forms part of the requirements model or requirements specification, which it is necessary to get the client to agree with that also.

Business modelling with use case diagrams :

Use case diagrams are used here to model the requirements for a system. They can also be used earlier in the life of a project to model an organization and how it operates. Business modelling is sometimes used when a new business is being set up, when an existing business is being 'reengineered'.

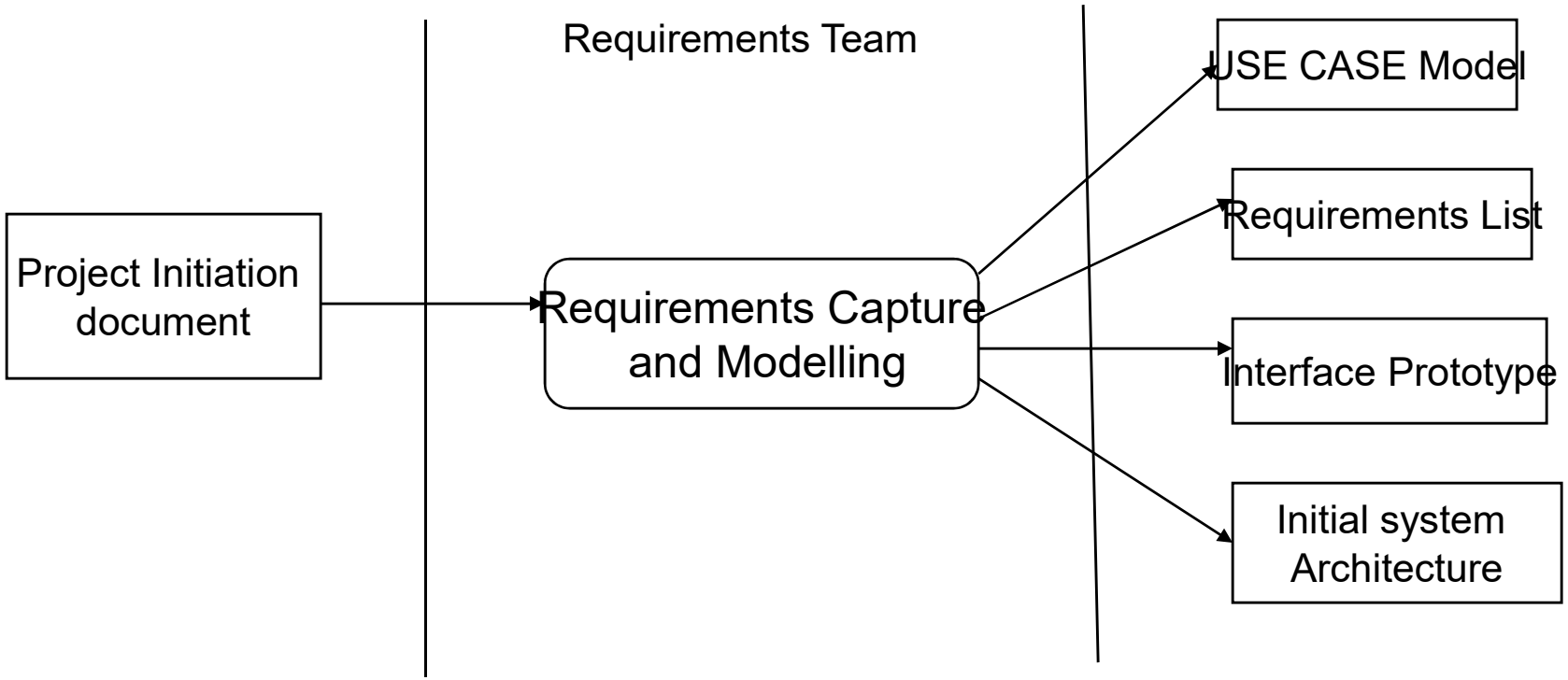
In the examples that we have shown above, the actors have all been employees of the company interacting with what will eventually be at least in part a computerized system. In business modelling, the actors are the people and organizations outside the company, interacting with functions within the company.

For example, The following Figure shows the Client as an actor and use cases that represent the functions of the business rather than functions of the computer system.



Requirements Capture and Modelling :

The first stage of most projects is one of capturing and modelling the requirements for the system. Here , we can include activity diagrams to illustrate the main activities in and products of each phase. The following Figure shows the first such diagram.



In this case we have not broken the activity Requirements capture and modelling down into more detail, though it could potentially be broken down into separate activities for the capture of the requirements (interviewing, observation, etc.) and for the modelling of the requirements (use case modelling, prototyping, etc.).

Here we used object flows to show the documents and models that are the inputs to and outputs from activities, and swimlanes to show the role that is responsible for the activities. In this case, one or more people in the role of Requirements Team will carry out this activity. In a small project, this may be one person, who carries out many other analysis and design activities.

Agate ltd Case Study Requirements Model

Requirements List :

The Requirements List includes a column to show which use cases provide the functionality of each requirement. This Requirements List includes some use cases in the use case model. Use cases are :

1.Add a new client

2. Add a new Campaign

3.Record completion of campaign

4. Record Client Payment

5.Assign staff to work on campaign

6.Assign a Staff contact.

7.Check campaign Budget

8.Create Concept Note

9. Browse Concept Notes

10. Add a new Advert to a campaign,record completion of campaign

11. Add a new Advert to a campaign

12.Add a new member of staff

13.Add a new staff Grade,change the rate for a staff grade

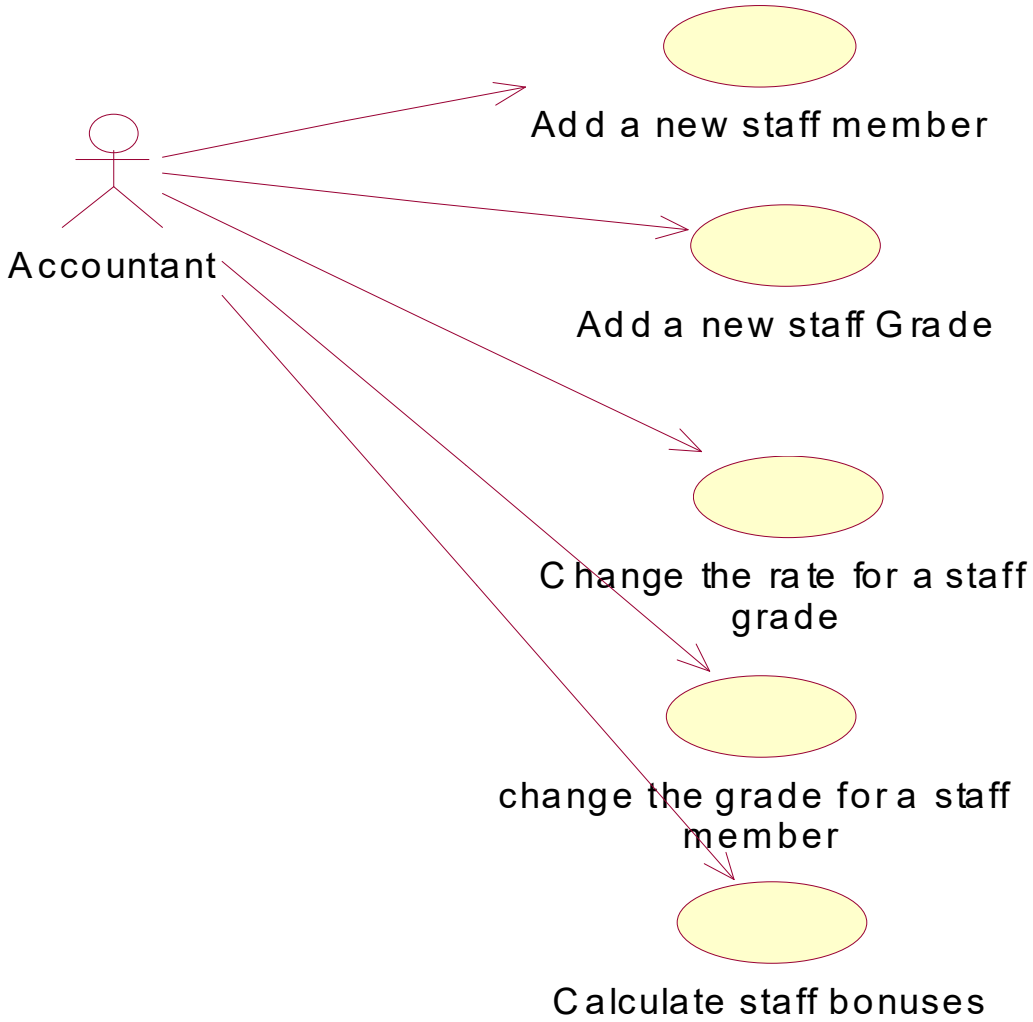
14. Change the grade for a member of staff

15.Calculate staff bonuses

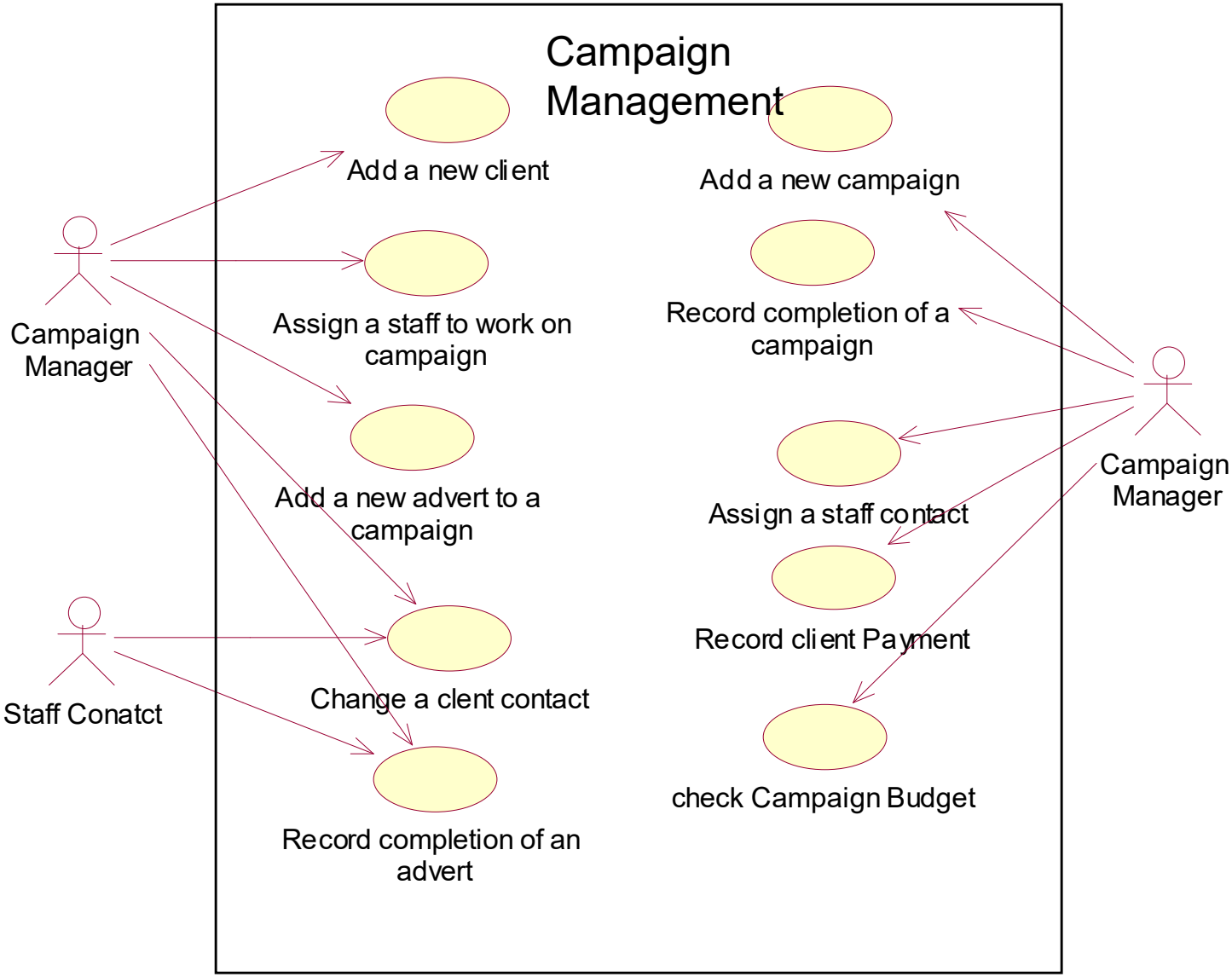
Actors in Agate Ltd.

1. Accountant
2. Campaign Manager
3. Staff contact
4. Staff
5. Campaign staff

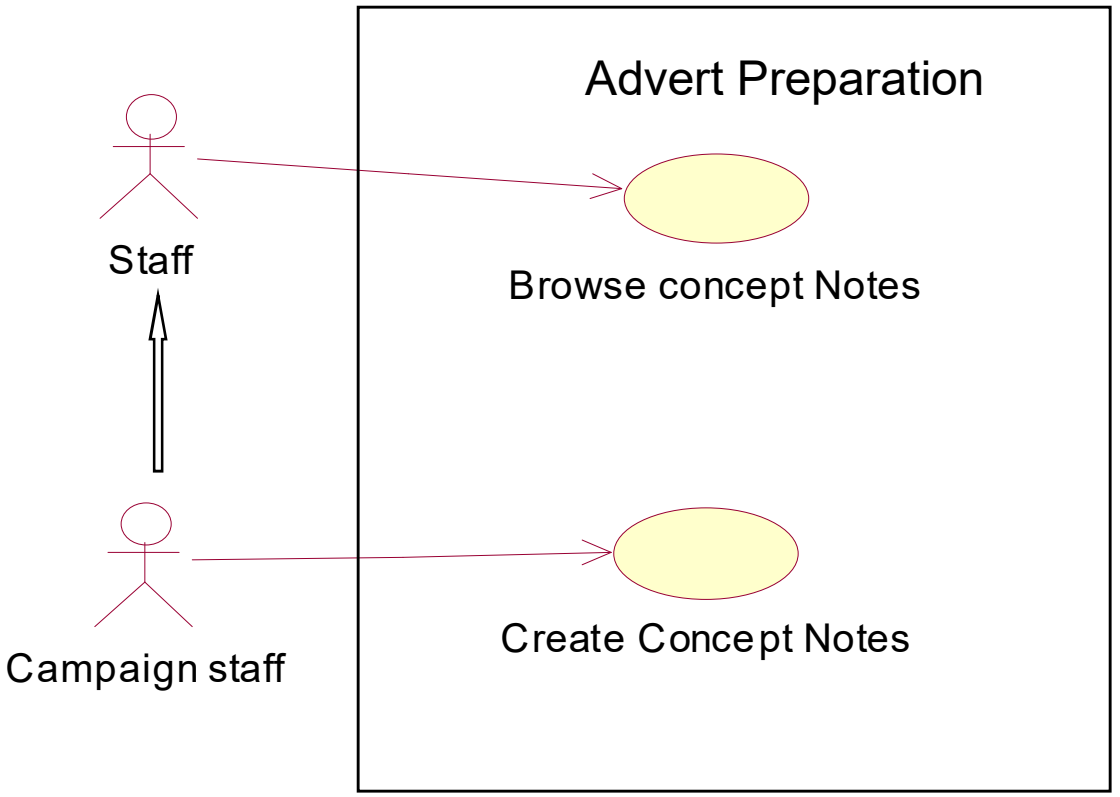
Use case Diagram for Staff Management



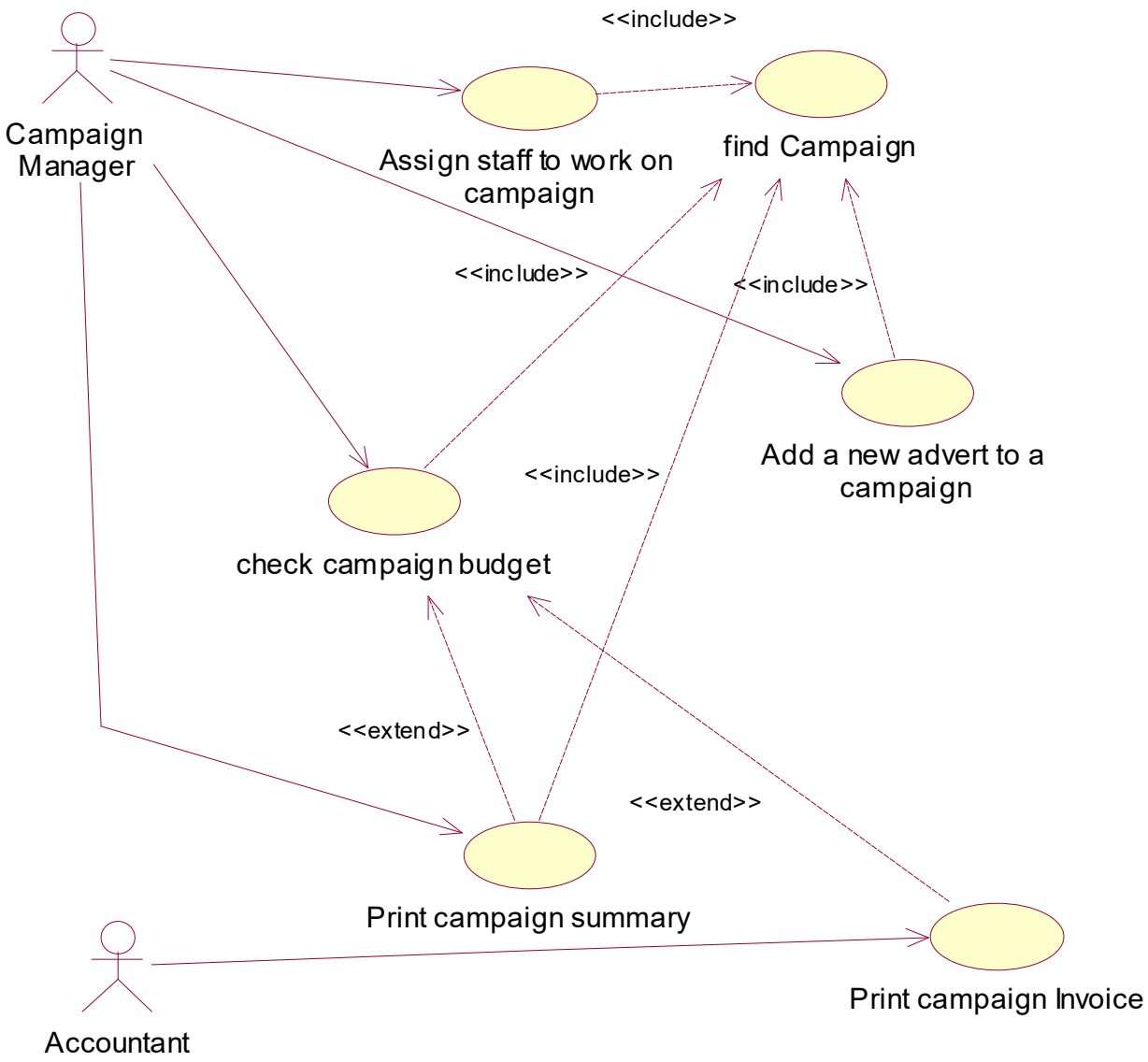
USE CASE Diagram for a Campaign Management



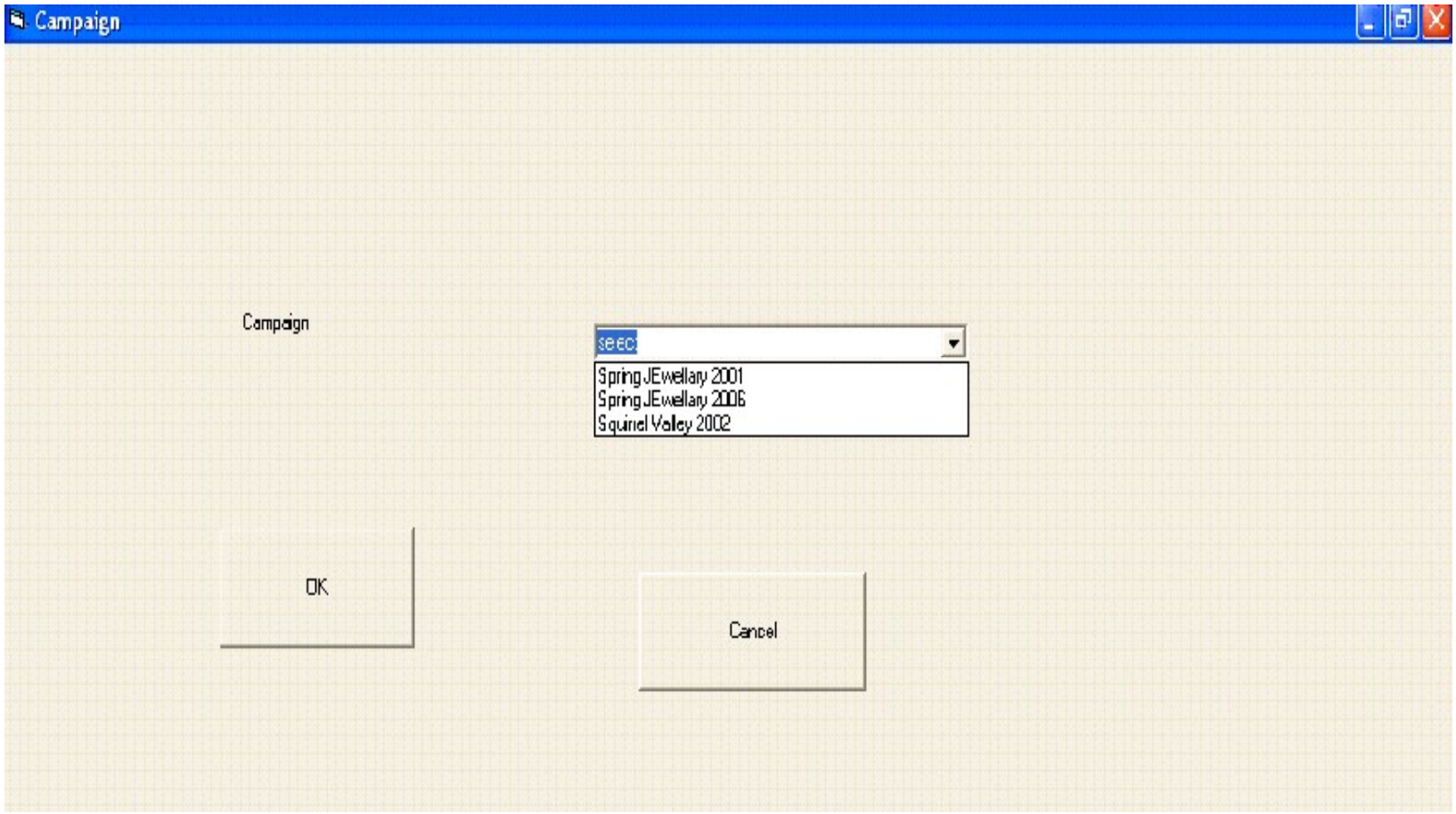
Use Case Diagram for Advert Preparation



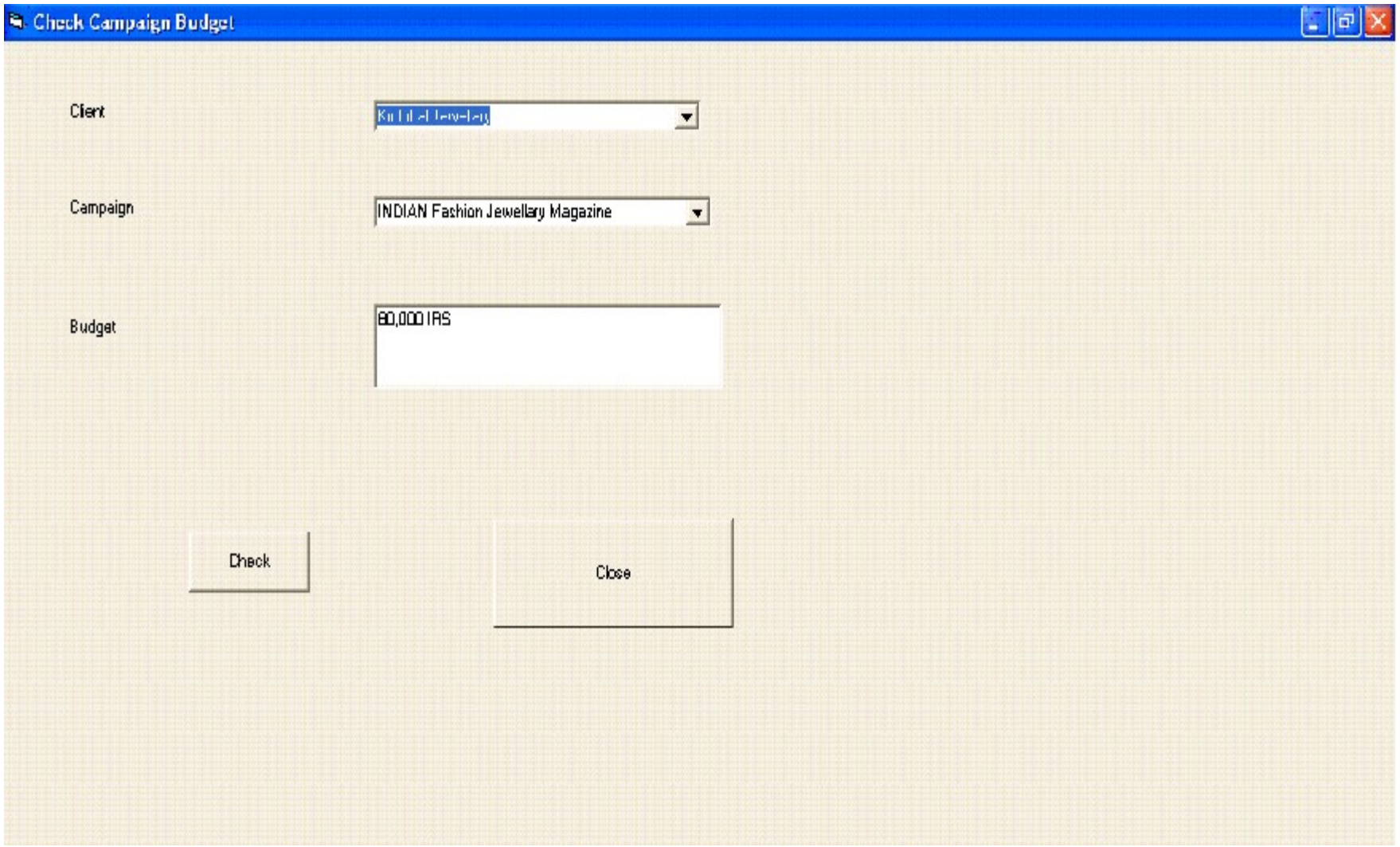
USE CASE diagram for <<include>> to find a campaign



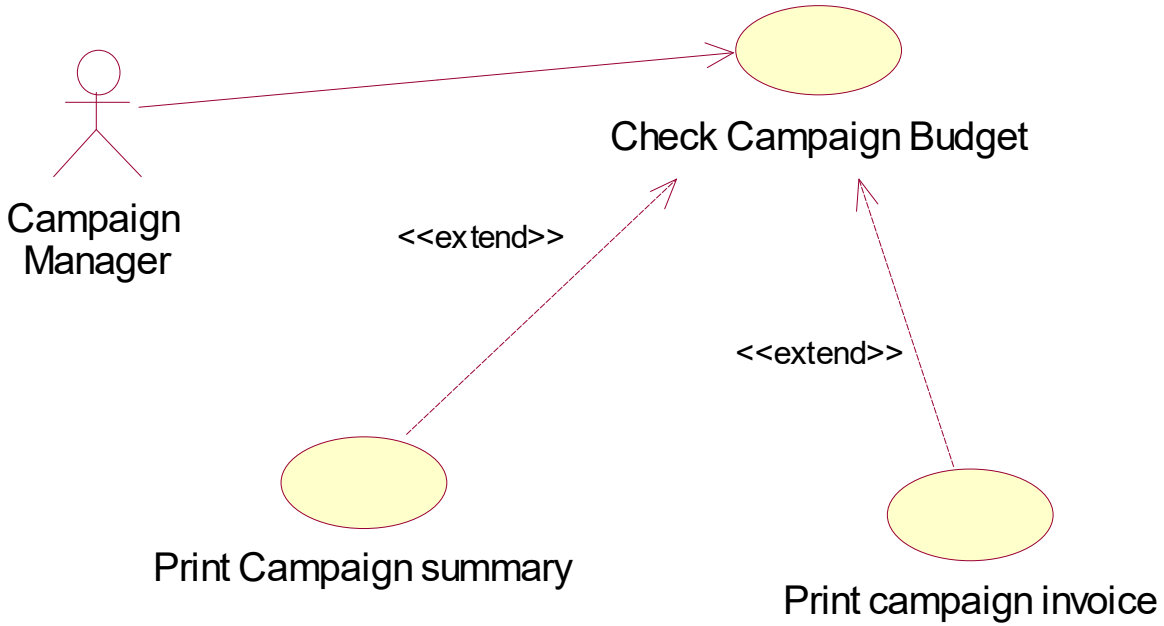
Prototype interface for the FIND campaign Use case



Prototype Use case for the Check Campaign Budget



Modified Use case diagram with Extensions



Activities of Requirements Modelling :

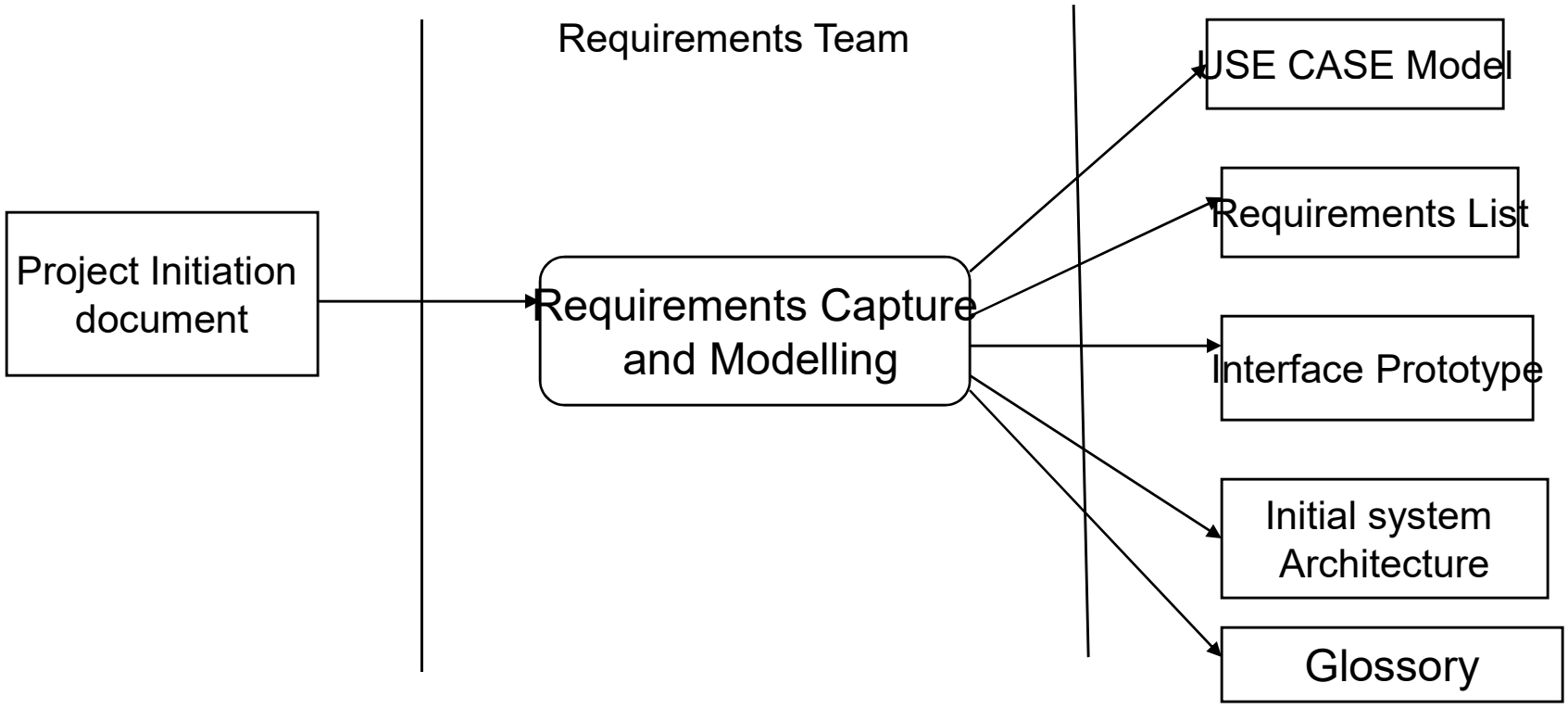
In a project which adopts an iterative life cycle, these activities may take place over a series of iterations.

In the first iteration, the emphasis will be on requirements capture and modelling Means Identifying some Use cases.

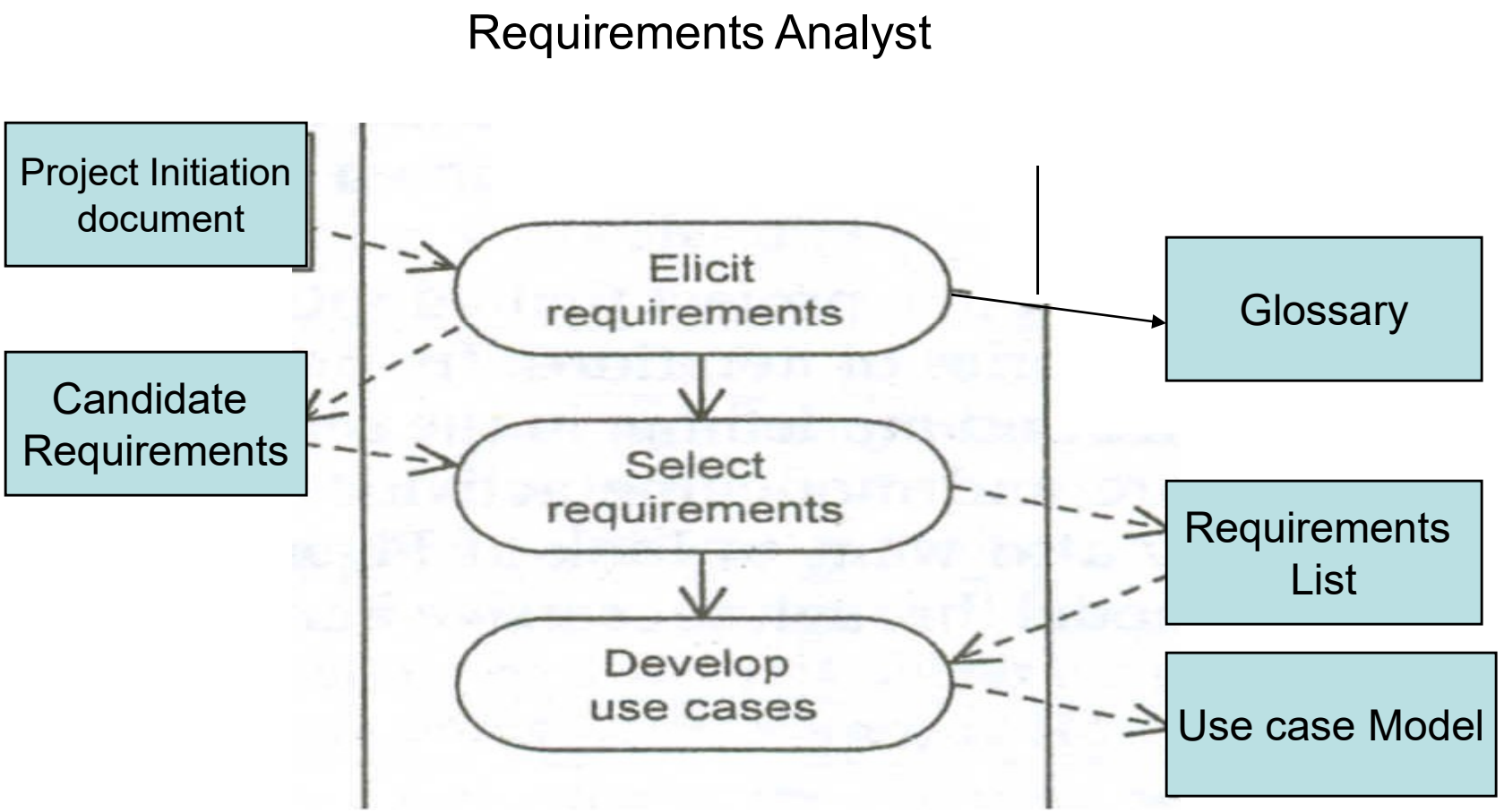
In the second, it will shift to analysis, but some requirements capture and modelling activities may still take place – Means Adding Additional Use cases and Developing Prototypes.

In the third Iteration , listing structured use cases and Descriptions.

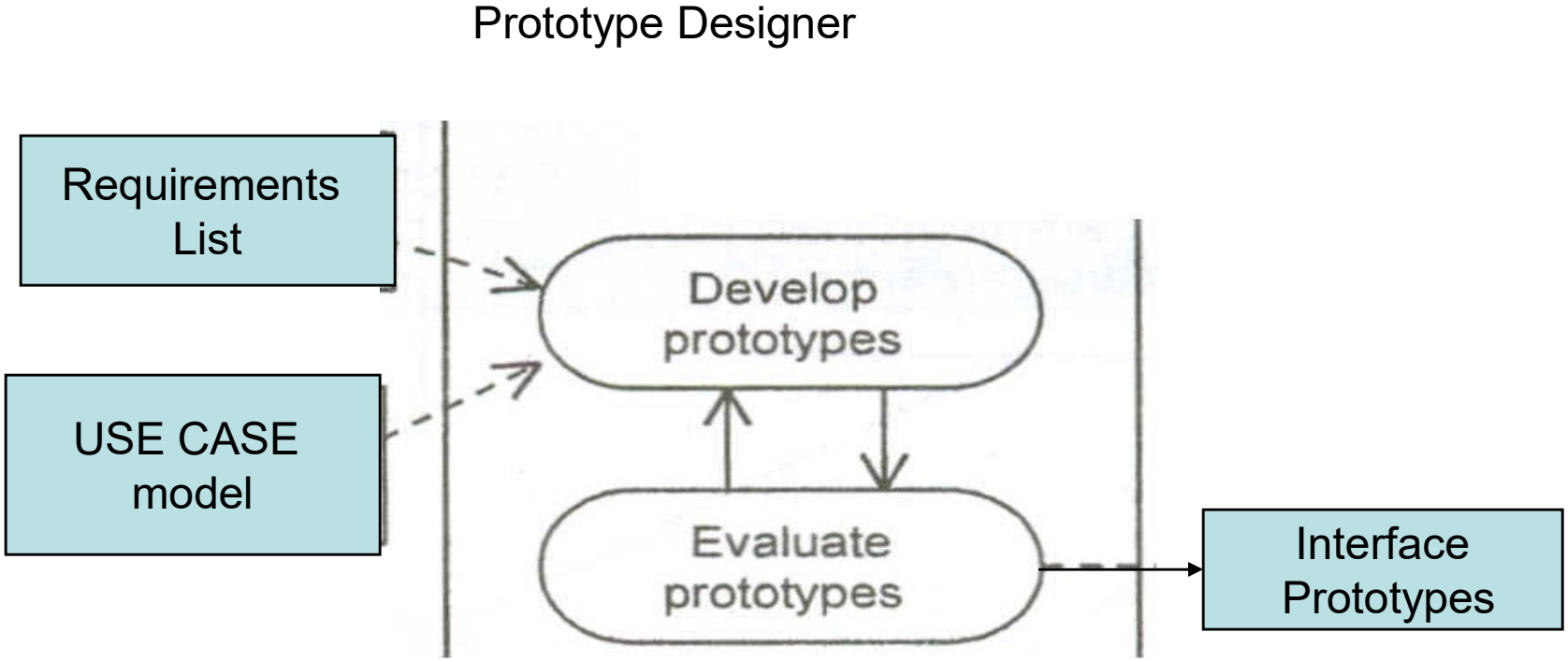
Activity diagram for Requirements capture and Modelling



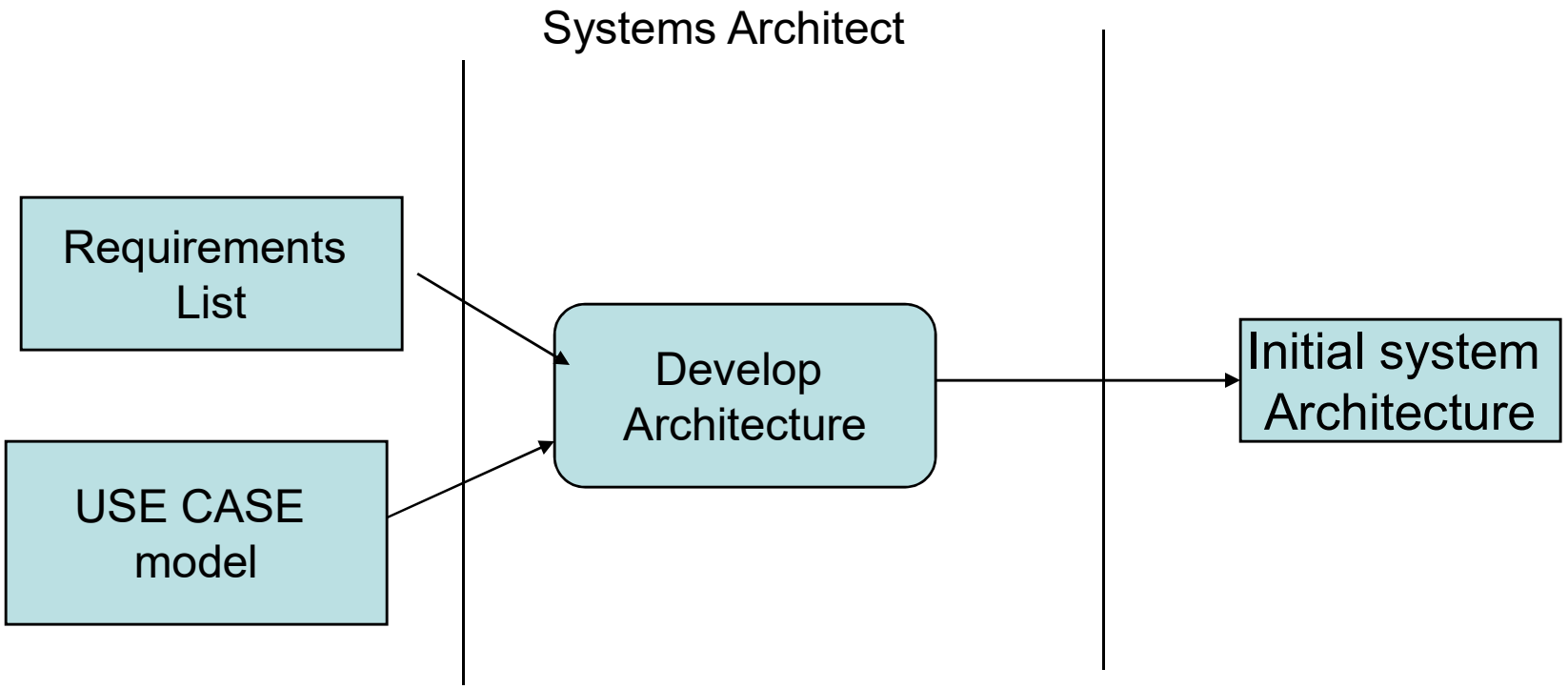
Activity diagram to show the activities involved in Capturing requirements



Activity diagram to show the activities involved in developing prototypes



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



7. REQUIREMENTS ANALYSIS

What must a Requirements Model DO ?

The most important factor for the success of an IS project is whether the software product satisfies its users' requirements. Models constructed from an analysis perspective focuses on determining these requirements. This means Requirement Model includes gathering and documenting facts and requests.

The use case model gives a perspective on many user requirements and models them in terms of what the software system can do for the user. Before the design of software which satisfies user requirements, we must analyze both the logical structure of the problem situation, and also the ways that its logical elements interact with each other. We must also need to verify the way in which different, possibly conflicting, requirements affect each other. Then we must communicate this under standing clearly and unambiguously to those who will design and build the software.

We do all of this by building further models, and these must meet several objectives.

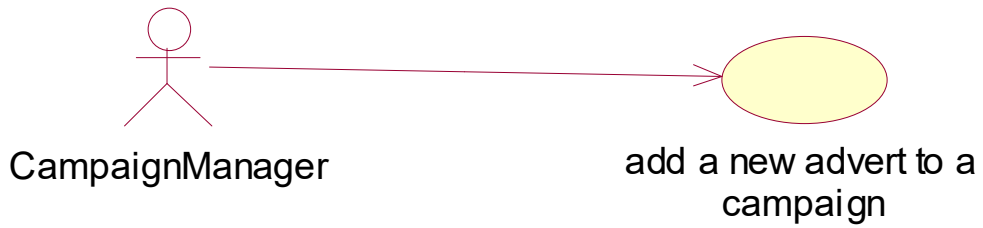
- They must contain an overall description of what the software should do.
- They must represent any people, physical things and concepts that are important to the analyst's understanding of what is going on in the application domain.
- They must show connections and interactions among these people, things and concepts.
- They must show the business situation in enough detail to evaluate possible designs.
- Ideally, they should also be organized in such a way that they will be useful later for designing the software.

Use Case Realization :

From use case to Sequence, collaboration , Object and class diagrams

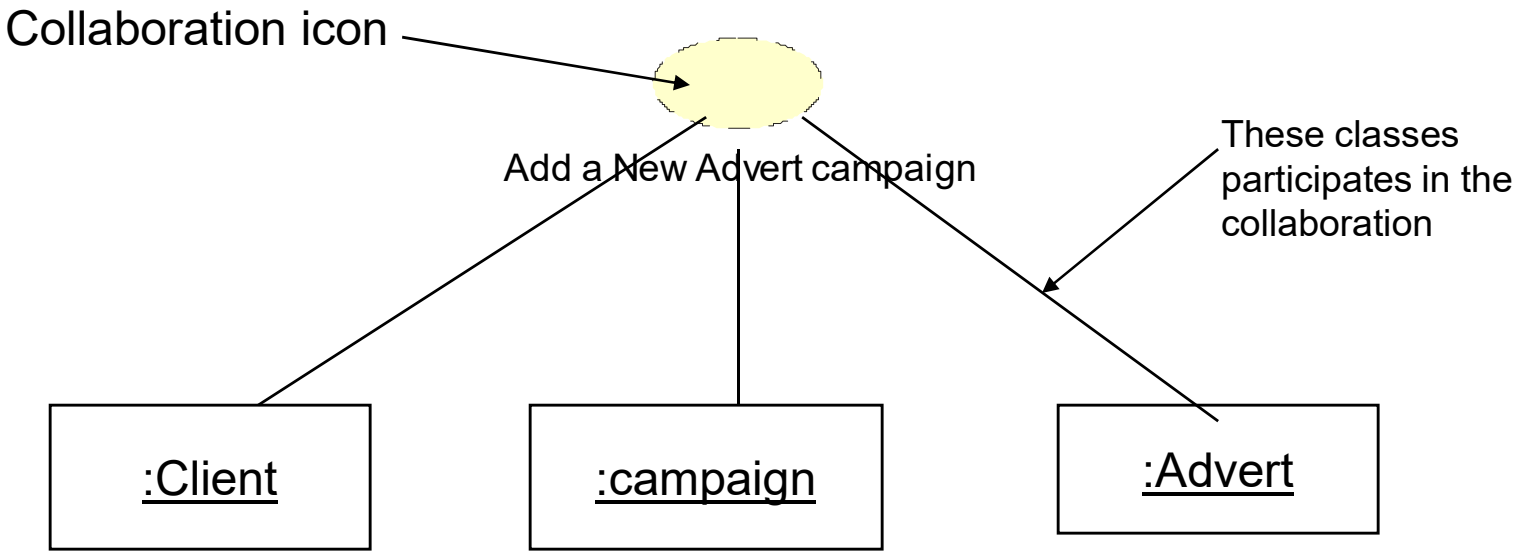
To move from an initial use case to the implementation of software that entirely satisfies the requirements identified by the use case involves at least one iteration through all of the development activities, from requirements modelling to implementation.

In relation to a single use case, this activity is known as *use case realization. Conisider* the use case Add a new advert to a campaign, which is shown below.



Use case realization is nothing but an instance of a use case which involves the identification of a possible set of classes, together with an understanding of how those classes might interact to deliver the functionality of the use case. The set of classes is known as a *collaboration.*

The simplest representation of a collaboration as a set of classes is shown below



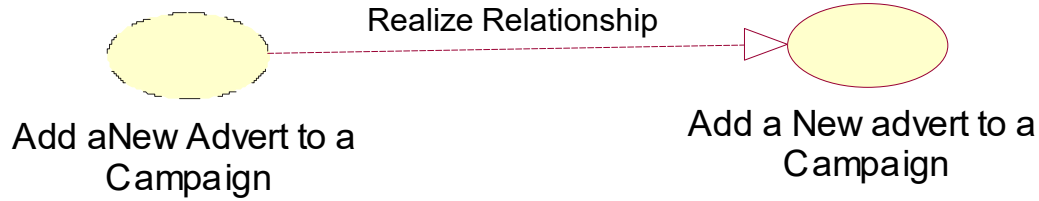
Here these three classes participate in the collaboration in other words, they can interact, when implemented as software, in such a way as to achieve the result described by the use case.

The another view of the following at the collaboration 'from the outside‘, which is used to show the relationship between a collaboration and the use case that it realizes.

Here we have similarity between the two icons: a collaboration appears to differ from a use case in this view only in being drawn with a dashed line instead of a solid one. However, the collaboration is not the same thing as the use case. Instead, it has a relationship with the use case that is shown here by the *dependency* arrow. Here, this means that the definitions of these classes must maintain a reference to the collaboration, which in turn must maintain a reference to the use case.

A collaboration realizes a specific use case (or)

USE CASE realization for ADD A NEW ADVERT CAMPAIGN



Means this type of representation doesn't includes , how they interact nor how they relate to other parts of the model, but this is just one view of a collaboration.

In detail this can be realized by using the following UML diagrams.

1. Sequence diagram
2. Collaboration diagram
3. Object diagram
4. Class diagram

USE CASE REALIZATION IN TERMS OF SEQUENCE DIAGRAM

Collaboration diagrams and sequence diagrams are called interaction diagrams.

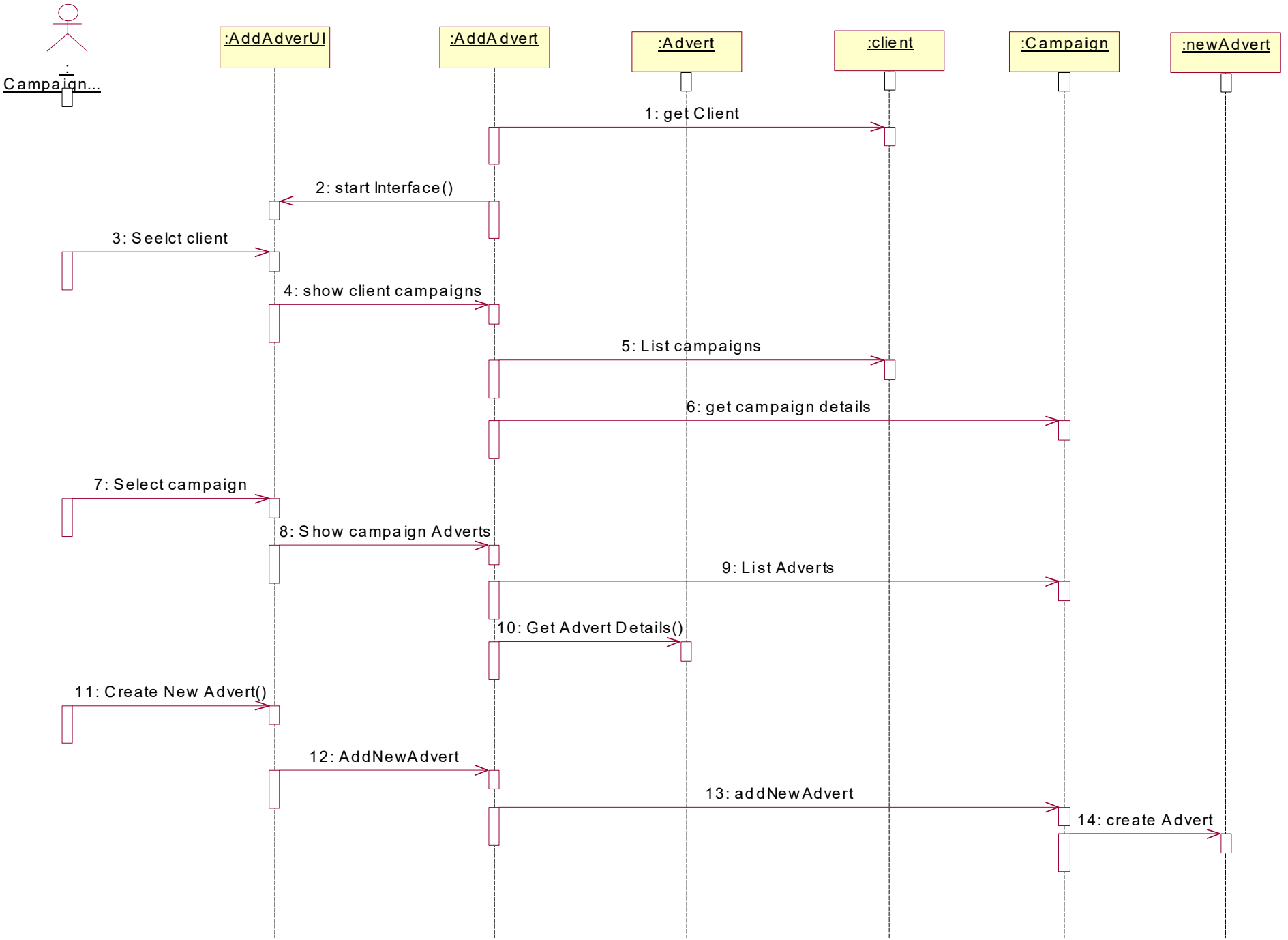
A sequence diagram is a graphical view of a scenario that shows object interaction in a time-based sequence - what happens first, what happens next. Sequence diagrams establish the roles of objects and provides essential information to determine class responsibilities and interfaces. This type of diagram is best used during early analysis phases in design because they are simple and easy to comprehend.

Sequence diagrams are normally associated with use cases. Sequence diagrams are closely related to collaboration diagrams and both are alternate representations of an interaction.

There are two main differences between sequence and collaboration diagrams: sequence diagrams show time-based object interaction while collaboration diagrams show how objects associate with each other.

A sequence diagram has two dimensions: typically, vertical placement represents time and horizontal placement represents different objects.

The following Sequence diagram for a ADD A NEW ADVERT TO A CAMPAIGN


:
Campaign...
:AddAdverUI
:AddAdvert
:Advert
:client
:Campaign
:newAdvert
1: get Client
2: start Interface()
3: Seelct client
4: show client campaigns
5: List campaigns
6: get campaign details
7: Select campaign
8: Show campaign Adverts
9: List Adverts
10: Get Advert Details()
11: Create New Advert()
12: AddNewAdvert
13: addNewAdvert
14: create Advert

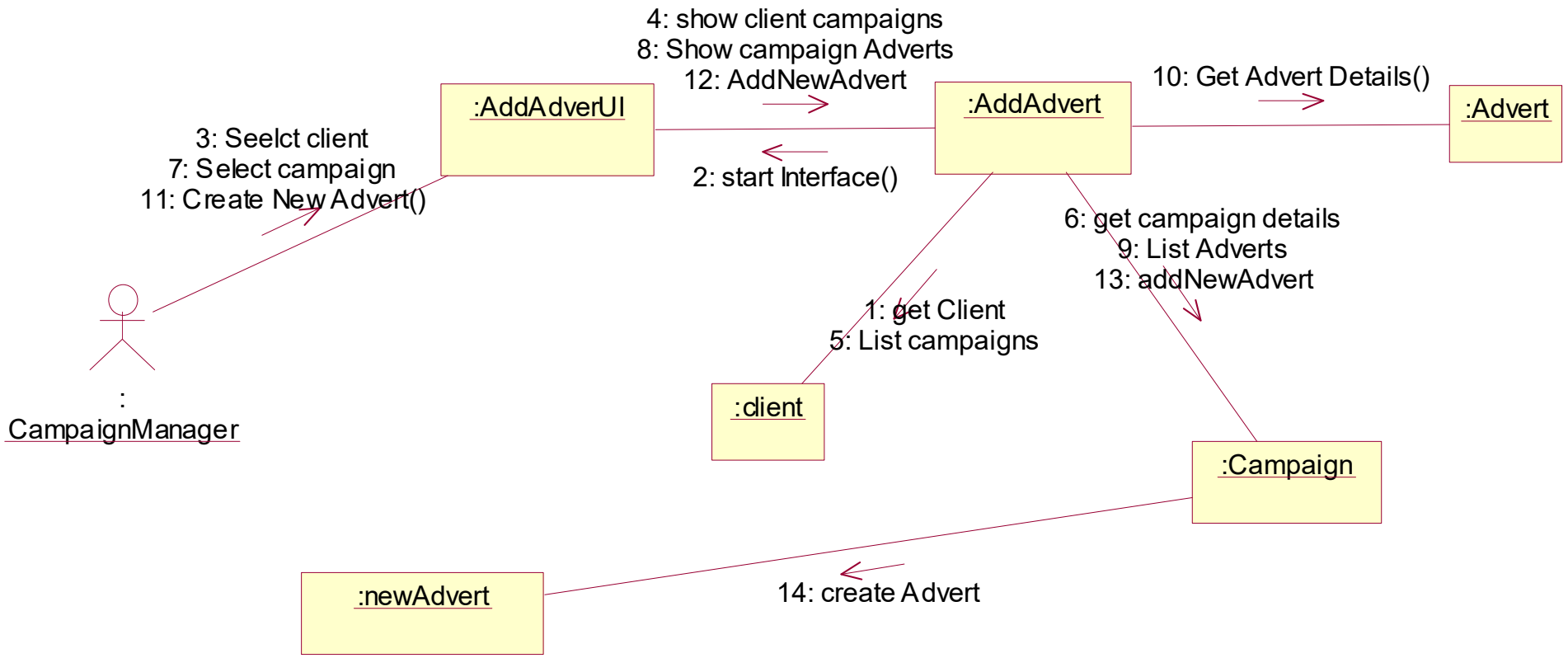
USE CASE REALIZATION IN TERMS OF COLLABORATION DIAGRAM

Collaborations can also be represented in various ways that reveal their internal details. The *collaboration diagram* is probably the most useful among all UML diagrams for use case realization.

Collaboration diagrams and sequence diagrams are called interaction diagrams. A collaboration diagram shows that the order of messages that implement an operation or a transaction. Collaboration diagrams show objects, their links, and their messages. They can also contain simple class instances and class utility instances. Each collaboration diagram provides a view of the interactions or structural relationships that occur between objects and object like entities in the current model.

The following collaboration diagram shows some of the structure that exists between the objects that take part in the collaboration.

Collaboration diagram for a ADD A NEW ADVERT TO A CAMPAIGN



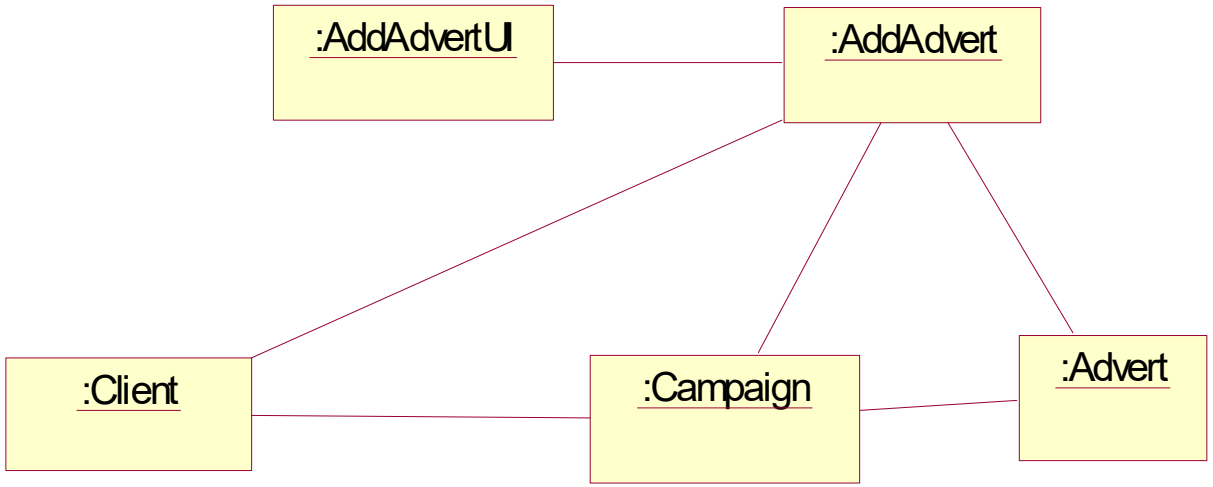
USE CASE REALIZATION IN TERMS OF OBJECT DIAGRAM

An object diagram shows the existence of objects and their relationships in the logical design of a system. An object diagram may represent all or part of the object structure of a system, and primarily focuses on the semantics of mechanisms in the logical design. A single object diagram represents transitory event or configuration of objects.

After realization of USE CASE as a collaboration diagram, it can be represented as an *object* (or *instance)* diagram. The corresponding object diagram is shown below for the same USE CASE.

There exists strong structural and notational similarity B/W Object and collaboration diagram . This is because both show object instances and links, although only the collaboration diagram shows messages between the objects.

Object diagram for a ADD A NEW ADVERT TO A CAMPAIGN

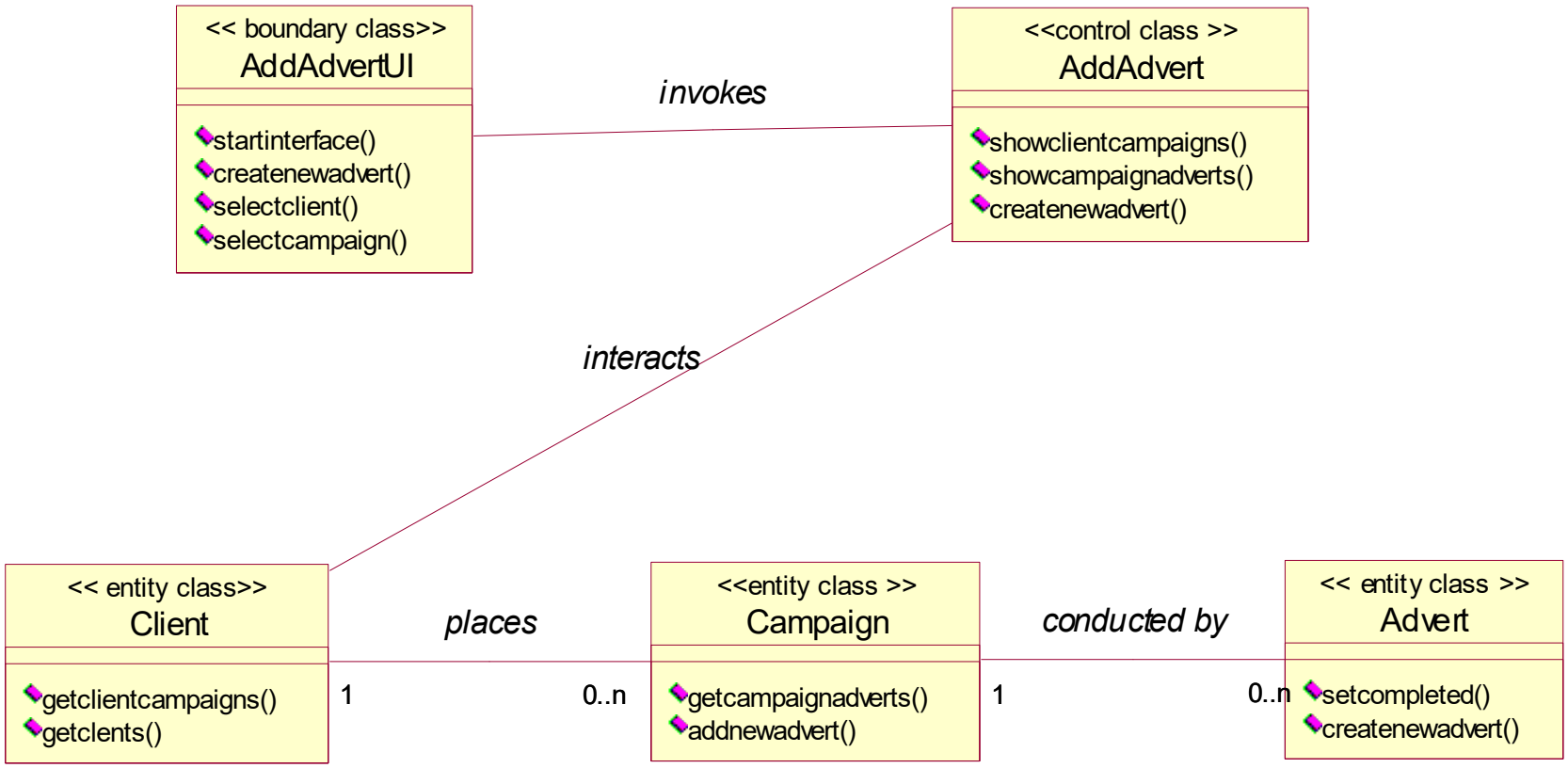


USE CASE REALIZATION IN TERMS OF CLASS DIAGRAM

Class diagrams contain icons representing classes, interfaces, and their relationships. You can create one or more class diagrams to represent the classes at the top level of the current model; such class diagrams are themselves contained by the top level of the current model. You can also create one or more class diagrams to represent classes contained by each package in your model; such class diagrams are themselves contained by the package enclosing the classes they represent; the icons representing logical packages and classes in class diagrams.

Finally ,After use case realization in the form of a collaboration diagram, it can be represented as a class diagram as shown below.

CLASS diagram for a ADD A NEW ADVERT TO A CAMPAIGN



The distinction among these different representations ,

The collaboration icon is in itself simply a high-level abstraction that can stand for any of the other forms. The diagrams (use case realization, collaboration , object, class) shows some of the intermediate forms that realize a use case during the progressive and iterative development of the resulting software.

Each form in this series is, one step closer to a design model, and thus ultimately to executable code. Each also serves a particular modelling perspective.

The main difference B/W collaboration and class diagram is,

a collaboration diagram highlights the interaction among a group of collaborating objects, but it ignores internal details of classes and some aspects of the structure. It is also not easy to read the sequence of messages on a collaboration diagram.

where as

a class diagram ignores the interaction altogether, but shows the structure in more detail and can show a lot of the internal features of the classes.
Collaborations can also be expressed in ways that do not concern us so much from a requirements analysis perspective .

Analysis class stereotypes

Analysis class stereotypes , represents three particular kinds of class that will be encountered again and again when carrying out requirements modelling.

UML DEFINITION :

Stereotype :

• A new type of modeling element that extends the semantics of the metamodel.

• Stereotypes must be based on certain existing types or classes in the metamodel.

• Stereotypes may extend the semantics but not the structure of preexisting classes.

• Certain stereotypes are defined in the UML, others may be user defined .

UML is designed to be capable of extension, developers can add new stereotypes depends on need. But this is only done when it is absolutely necessary.

Three analysis class stereotypes to the UML are :

Boundary,

Control and

Entity classes.

1. Boundary classes :

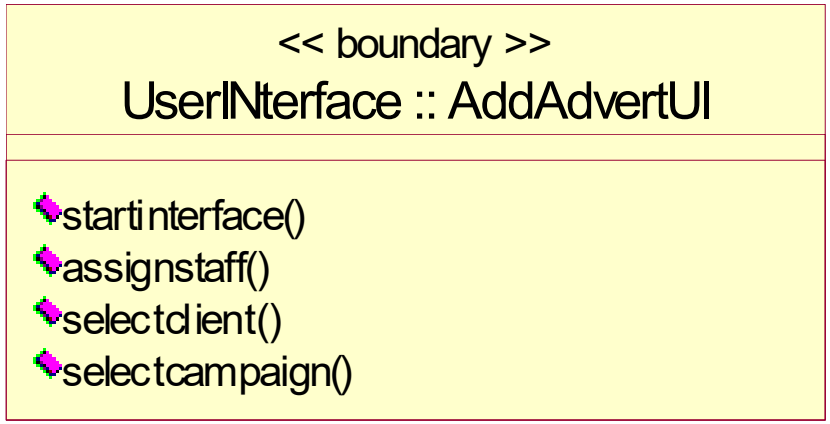
Boundary classes , it is a 'model interaction between the system and its actors' . Since they are part of the requirements model, boundary classes are relatively abstract. They do not directly represent all the different sorts of interface that will be used in the implementation language. The design model may well do this later, but from an analysis perspective we are interested only in identifying the main logical interfaces with users and other systems.

This may include interfaces with other software and also with physical devices such as printers, motors and sensors. Stereotyping these as boundary classes emphasizes that their main task is to manage the transfer of information across system boundaries. It also helps to partition the system, so that any changes to the interface or communication aspects of the system can be isolated from those parts of the system that provide the information storage.

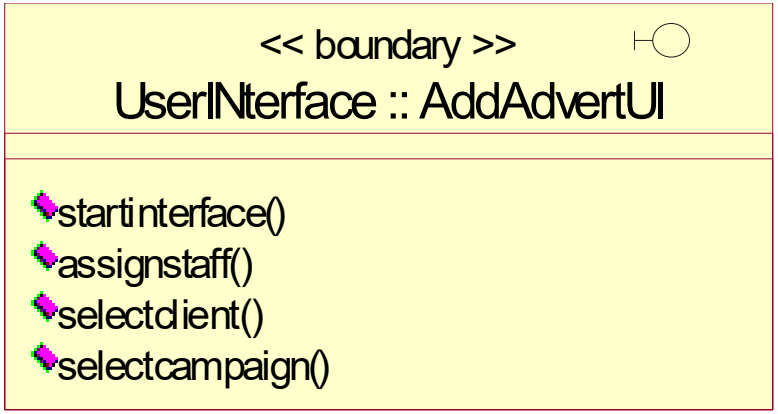
The class User Interface:: AddAdvertUI is a typical boundary class. This style of writing the name shows that the class is AddAdvertUI and it belongs to the User Interface package When we write the package name in this way before the class name, it means that this class is imported from a different package from the one with which we are currently working. In this case, the current package is the Agate application package, which contains the application requirements model, and thus consists only of domain objects and classes.

Alternative notations for Boundary class stereotype can be represented as shown below

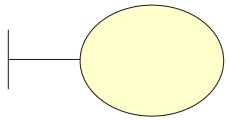
a) With stereotype



b) With stereotype and symbol



c) symbol



UserInterface :: AddAdvertUI

2. Entity classes

The second analysis class stereotype is the entity class,which are given in the class diagram of ADD A NEW ADVERT TO A CAMPAIGN by the three classes Client, Campaign and Advert .

Entity classes are used to model 'information and associated behaviour of some phenomenon or concept such as an individual, a real-life object, or a real-life event' . As a general rule, entity classes represent something within the application domain, but external to the software system, about which the system must store some information.

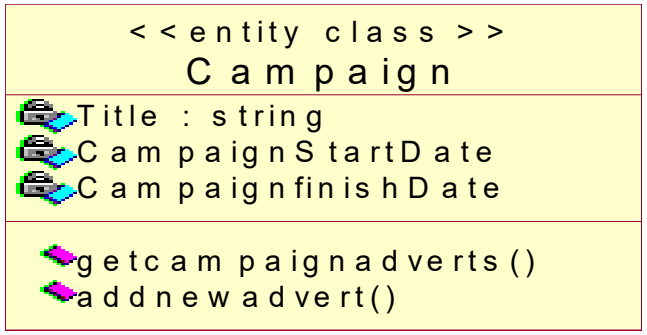
Instances of an entity class will often require persistent storage of information about the things that they represent. This can sometimes help to decide whether an entity class is the appropriate modelling construct.

For example, an actor is often not represented as an entity class. This is in spite of the fact that all actors are within the application domain, external to the software system and important to its operation. But most systems have no need to store information about their users nor to model their behaviour. While there are some obvious exceptions to this (consider a system that monitors user access for security purposes), these are typically separate, specialist applications in their own right. In such a context, an actor would be modelled appropriately as an entity class, since the essential requirements for such a system would include storing information about users, monitoring their access to computer systems and tracking their actions while logged on to a network. But it is more commonly the case that the software we develop does not need to know anything about the people that use it, and so actors are not normally modelled as classes.

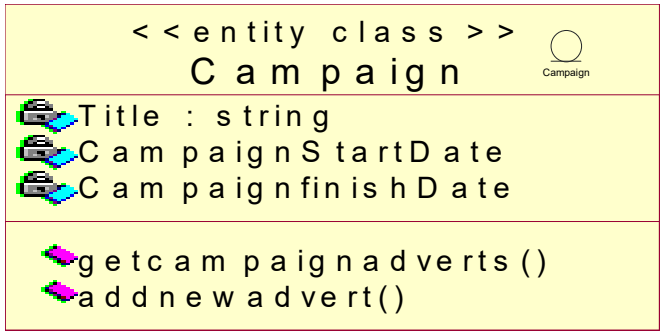
The following are representations for Entity classes.

The following are representations for Entity classes.

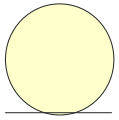
a) With stereotype



b) With stereotype and symbol



c. symbol



Campaign

3. Control classes

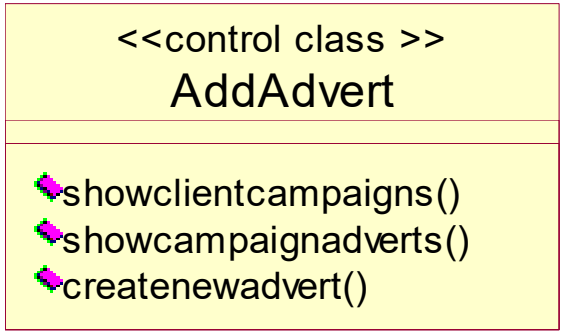
The third of the analysis class stereotypes is the control class, given by y the class Control: :AddAdvert in a ADD A NEW ADVERT TO CAMPAIGN.

Control classes 'represent coordination, sequencing, transactions and control of other objects' .In the USDP, as in the earlier methodology Objectory, it is generally recommended that there should be a control class for each use case .

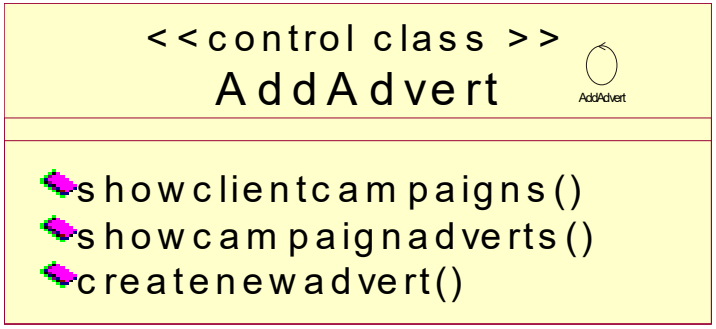
In a sense, then, the control class represents the calculation and scheduling aspects of the logic of the use case at any rate, those parts that are not specific to the behaviour of a particular entity class, and that *are* specific to the use case. Meanwhile the boundary class represents interaction with the user and the entity classes represent the behaviour of things in the application domain and storage of information that is directly associated with those things.

The following are the notations can be used to represent Control class

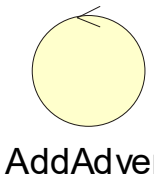
a) With stereotype



b) With stereotype and symbol



c. symbol



CLASS DIAGRAM

The following are the important Aspects to be considered in the CLASS diagram :

1 Relative stability of classes and instances

Entity classes often represent the more permanent aspects of an application domain, while boundary and control classes represent relatively stable aspects of the way that the software is intended to operate.

For example, as long as Agate continues to operate in the advertising business, its business activities are likely to involve campaigns, budgets and adverts, creative staff and campaign managers. And as long as the user requirements for the system do not change, the same boundary and control classes can model the operation of the software and its interaction with users. Thus the description of each class is also relatively stable, and will probably not change frequently.

By contrast, object instances often change frequently, reflecting the need for the system to maintain an up-to-date picture of a dynamic business environment. Instances are subject to three main types of change during system execution.

First, they are created. For example, when Agate undertakes a new campaign, details are stored in a new Campaign object. When a new member of staff is recruited, a corresponding Staff Member object is created.

Second, they can be destroyed. For example, after a campaign is completed and all invoices are paid, eventually there comes a time when it is no longer of interest to the company. All information relating to the campaign is then deleted by destroying the relevant Campaign instance.

Instances of boundary and control classes are particularly volatile - that is, they have short lifetimes and are subject to frequent creation and destruction. For example, a control object and one or more boundary objects are typically instantiated at the start of each execution of a use case, and then destroyed again as soon as the execution is completed.

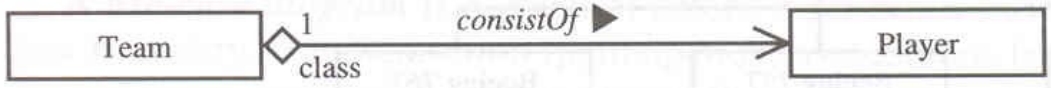
Finally, objects can be updated, which means a change to the recorded values of one or more characteristics. This is typically done to keep each object in step with the thing that it represents.

Relationships among Classes :

1) Dependency Relationship
2) Association Relationship – a) Role Names
   b) Relation names
   c) Multiplicity
   d) Aggregation –

Aggregation is a form of association. A hollow diamond is attached to the end of the path to indicate aggregation. However, the diamond may not be attached to both ends of a line, it to be shown towards whole.

FIGURE 5–9
Association path.

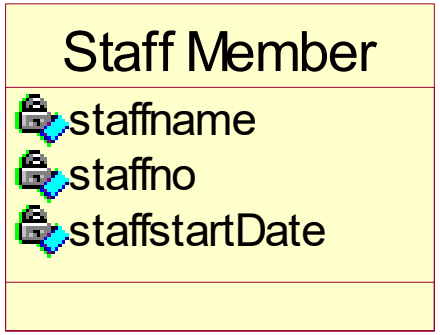


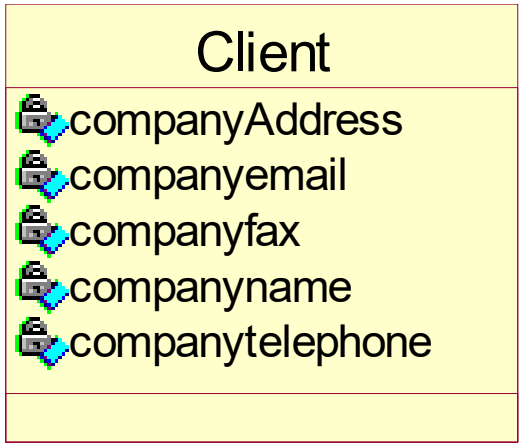
3) Generalization Relationship – (Feature of Inheritance)

2 Attributes

Attributes are part of the essential description of a class. They belong to the class, unlike objects, which instantiate the class. Attributes are the common structure of what a member of the class can 'know'. Each object will have its own, possibly unique, value for each attribute.

The following fig shows some possible attributes of Client and Staff Member in the Agate case study.





Here a class is partly defined by its attribute structure, instances are described by the values of their attributes. Some attribute values change during the life of an object.

3 Attributes and state

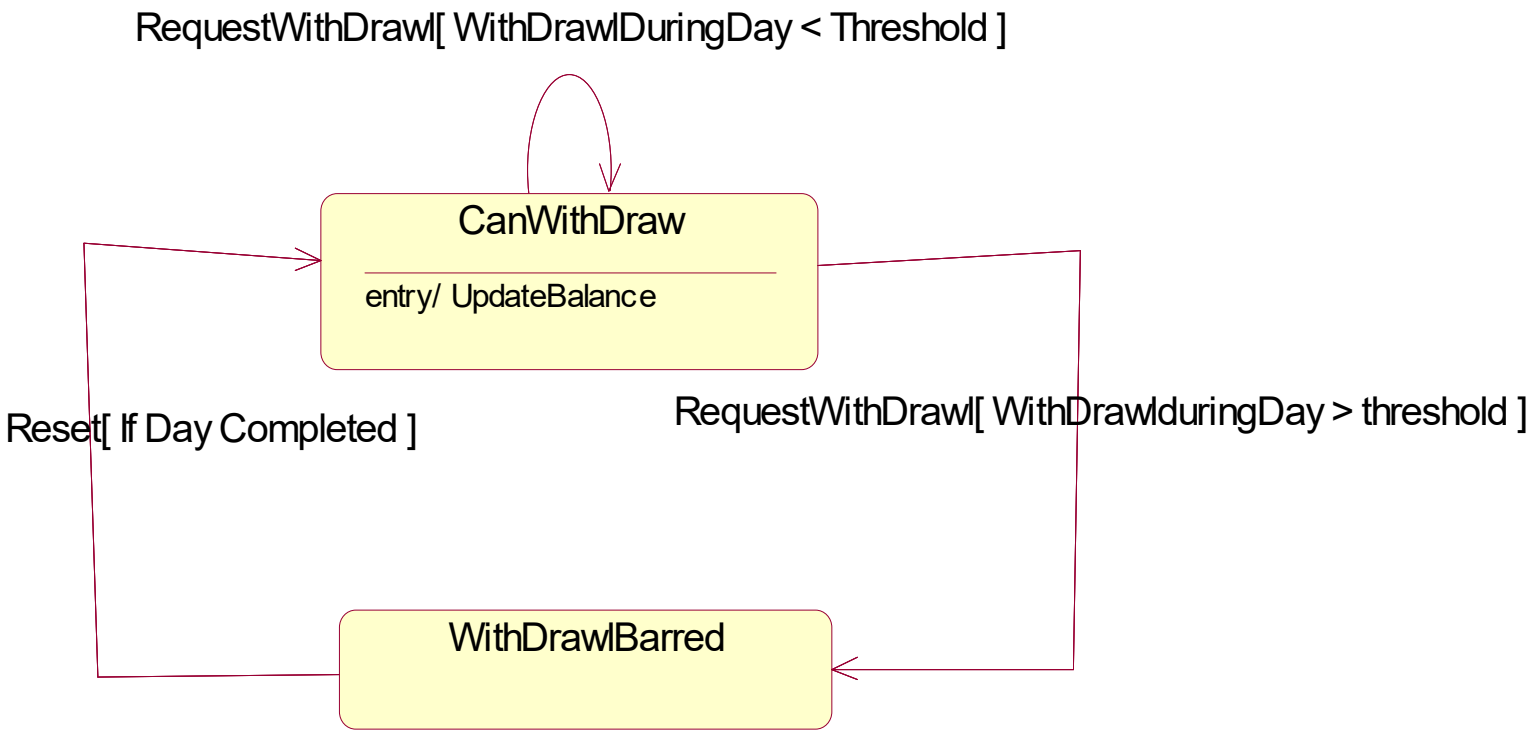
The current state of an object is partly described by the instance values of its attributes. When an attribute value changes, the object itself may change state. Some state changes are not significant, which simply means they do not affect the overall behaviour of the object , and hence of the system as a whole. Others have important implications for object and system behaviour.

An example is that is when your cash card will not allow further cash withdrawals because you have reached your daily limit. We can understand this in terms of object states, as follows.

Within the bank's computer system, an object yourAccount has an attribute withdrawnDuringDay that stores your total withdrawals of the day, and states CanWithdraw and WithdrawalBarred every day, the value of withdrawnDuringDay is reset to zero every day.

The object is in its CanWithdraw state. Each time you withdraw cash, withdrawnDuringDay is updated to take account of the transaction. Once the value of withdrawnDuringday reaches a critical threshold set by the bank, yourAccount changes state to withdrawalBarred.

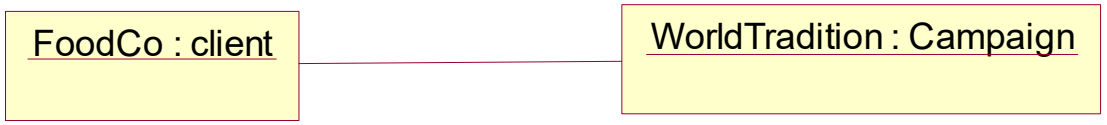
The following figure shows this as a UML state transition diagram.



4 link and association

The concepts of link and association, like those of object and class, are very closely related. A link is a logical connection between two or more objects.

In requirements analysis, this expresses a logical relationship in the application domain. An example for Agate is the connection between FoodCo and the 'World Tradition' TV campaign, described by: 'FoodCo is the client for the World Tradition campaign.'



Linked instances may be from different classes or from the same class. An example of the latter might be the link supervises between a manager and another staff member who are both instances of Staff

Just as a link connects two instances, an association connects two classes, and just as a class describes a set of similar instances, an association describes a set of similar links , here link and Association are same .The similarity is that those links that exist are all between objects of the associated classes. An association is a connection between two classes that represents the possibility of their instances participating in a link.

5 Significance of associations

While a link between two objects represents a real-world connection, an association between two classes represents the possibility of links.

For example, In Agate, a member of staff is assigned to each client as a staff contact. This can be mirrored by links between each : Client and a corresponding : Staff Member, but a link is modelled only if it supports a specific requirement. We know from the Agate use cases that campaign managers need to be able to assign and change a client contact. Therefore the requirements model must permit this link, otherwise it will not be possible to design software that meets these needs.

This is shown below along with association Names

Association between client and staff member

6 Associations and state

object's state is defined by its current set of links to other objects. Whenever a link to another object is created or destroyed, the object changes state. As with state changes that depend on attribute updates, some of these are significant and others are less so.

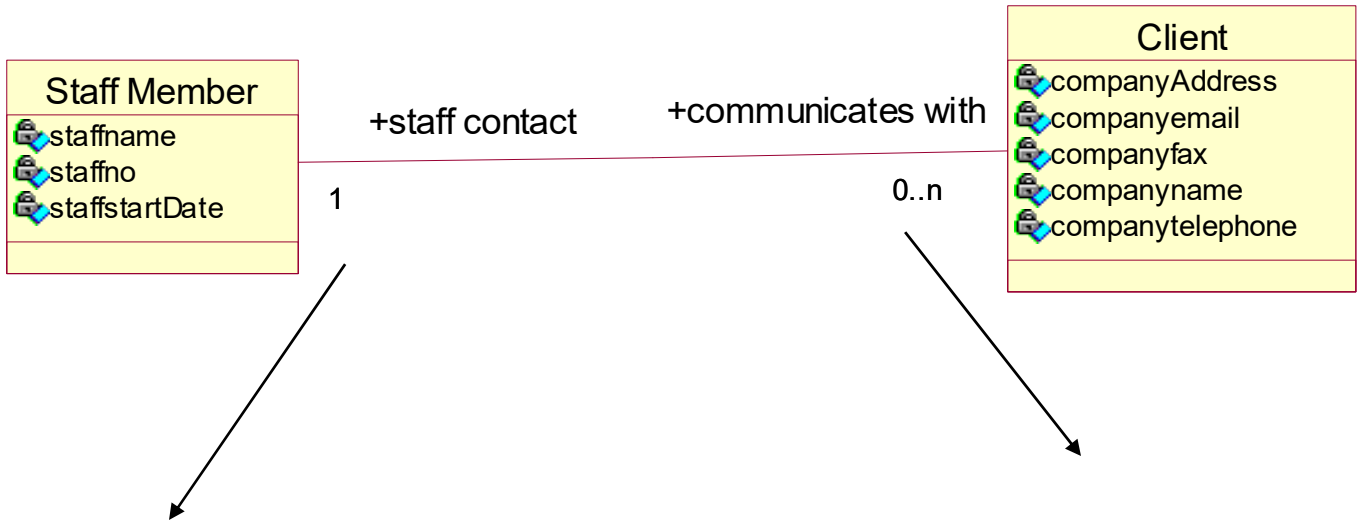
An example of link creation is when a campaign manager assigns a member of staff to be staff contact for a new client. If, instead, an existing staff contact is changed, this involves destroying one link and creating another in its place.

7 Multiplicity

Associations represent the possible existence of links between objects. It is often important to define a limit on the number of links with objects of another specific class in which a single object can participate. Multiplicity (the range of allowed *cardinalities)* is the term used to describe constraints on the number of participating objects.

A familiar example is the number of people allowed to be the designated as account holder for a bank account. A sole account has *one and only one* account-holder, a joint account may have *exactly two* account holders and a business partnership account may have *two or more* account holders. It is important to model these constraints correctly, as they may determine which operations will be permitted by the software. A badly specified system might incorrectly allow an unauthorized second person to draw money from a sole account. Alternatively it might prevent a customer from being able to draw money from a joint account.

The following figure shows communication of staff contact with client with multiplicity



Exactly one member is assigned with each client For communication purpose

Zero or more no of clients are allocated to each staff member

8 Operations

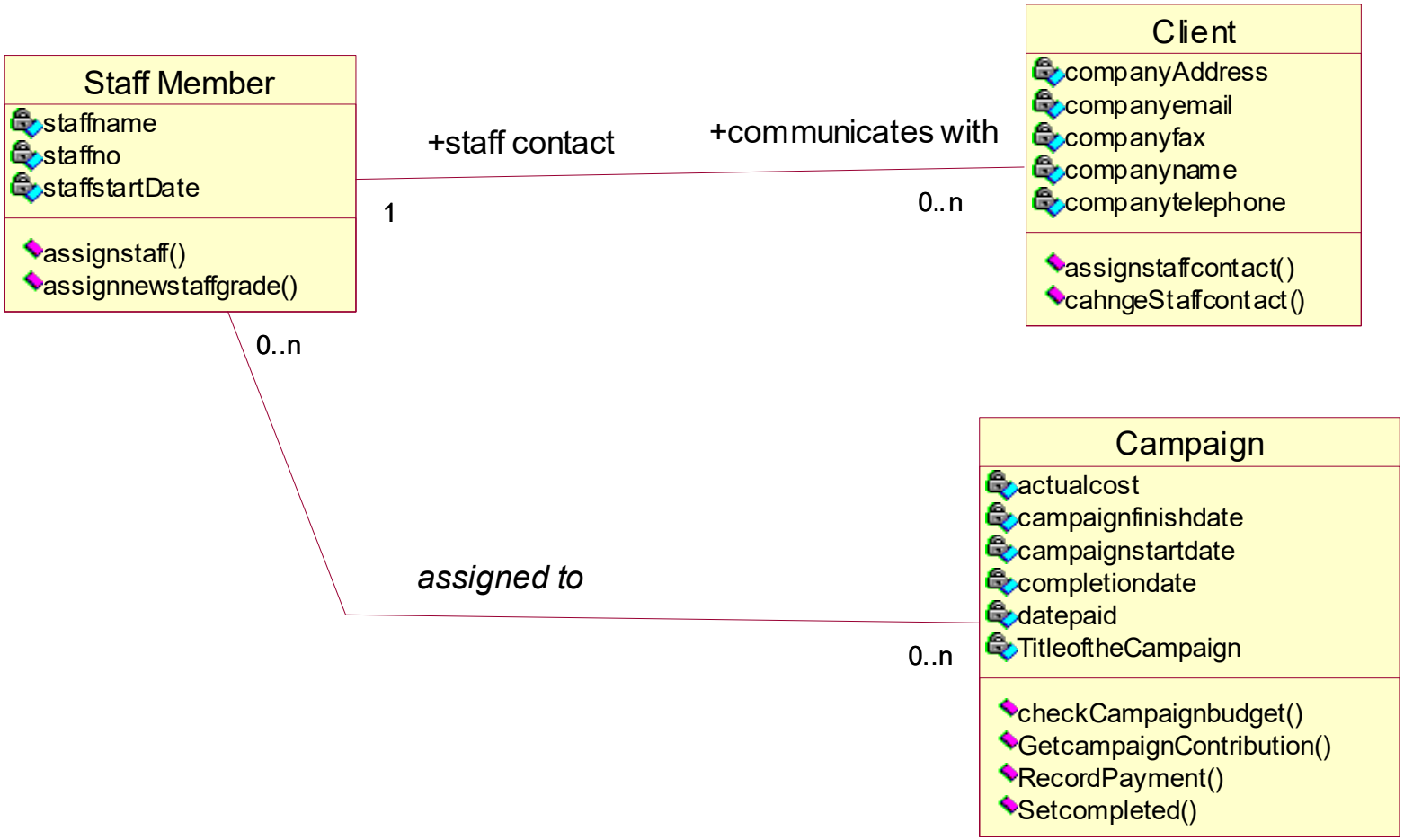
Operations are the elements of common behaviour shared by all instances of a class. They are actions that can be carried out by or carried on, an object. The classes modelled during requirements analysis represent real-world things and concepts, so their operations can be said to represent aspects of the behaviour of the same things and concepts.

Another way of defining operations, operations are services that objects may be asked to perform by other objects.

For example, in the Agate case study, Staff Member has an operation that calculates the amount of bonus pay due for a staff member. And, since staff bonus is partly based on the profit of each campaign a member of staff has worked on, Campaign has an operation to calculate the profit for each campaign.

An operation is a specification for some aspect of the behaviour of a class. Here Operations are eventually implemented by *methods,* and what a method actually does on any given occasion may be constrained by the value of object attributes and links when the method is invoked.

The Following figure shows Classes with Operations



Here operation parameters are placed in brackets after its name. Empty brackets indicate either that operation has no parameters, or they have not yet been defined

9 Operations and state

Here there is a relationship between an object's operations and its states. An object can only change its state through the execution of an operation. In fact, we are saying no more here than that attributes cannot store or update their own values, and links cannot make or break themselves.

But this gives an important point about the way that object encapsulation assists modularity in a system. The services that an object provides can only be accessed through the object's interface, which consists entirely of operation signatures defined at the class level.

In order to get an object to do anything at all change its data, create or destroy links, even respond to simple queries, another object must send a message that includes a valid call on an operation.

Drawing a class diagram

Consider practical aspects of drawing a class diagram, which involves providing where to look for the necessary information, and also a recommended sequence for carrying out the various tasks.

In practice, one project differs greatly from another, and some steps may be omitted altogether or carried out at a completely different stage of the life cycle. Experienced analysts will always use their own decisions on how to proceed in a given situation.

Importance of Identifying classes :

The class diagram is fundamental to object-oriented analysis. Through successive iterations, it provides both a high level basis for systems architecture, and a low-level basis for the allocation of data and behaviour to individual classes and object instances, and ultimately for the design of the program code that implements the system. So,it is important to identify classes correctly. However, given the iterative nature of the object-oriented approach, it is not essential to get this right on the first attempt itself.

APPROACHES FOR IDENTIFYING CLASSES :

We have four alternative approaches for identifying classes:

1. The noun phrase approach;
2. The common class patterns approach;
3. The use- case driven TO sequence/collaboration modeling approach;
4. Class Responsibility collaboration cards (CRC) approach.

The first two approaches have been included to increase analysts understanding of the subject; the unified approach uses the use-case driven approach for identifying classes and understanding the behavior of objects. However, these can be combined to identify classes for a given problem.

Another approach that can be used for identifying classes is Classes, Responsibilities, and Collaborators (CRC)

1. NOUN PHRASE APPROACH .

In this method, analyst read through the requirements or use cases looking for noun phrases. Nouns in the textual description are considered to be classes and verbs to be methods of the classes All plurals are changed to singular, the nouns are listed, and the list divided into three categories relevant classes, fuzzy classes (the "fuzzy area," classes we are not sure about), and irrelevant classes as shown below.

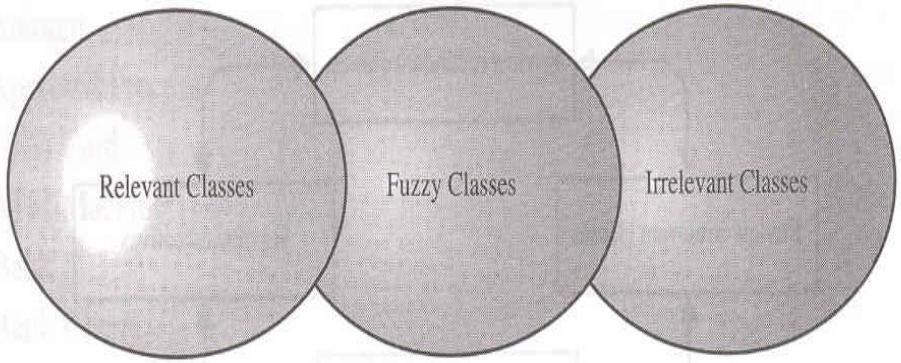


FIGURE 7-2
Using the noun phrase strategy, candidate classes can be divided into three categories: Relevant Classes, Fuzzy Area or Fuzzy Classes (those classes that we are not sure about), and Irrelevant Classes.

It is safe to scrap the irrelevant classes, which either have no purpose or will be unnecessary. Candidate classes then are selected from the other two categories. Here identifying classes and developing a UML class diagram just like other activities is an iterative process. Depending on whether such object modeling is for the analysis or design phase of development, some classes may need to be added or removed from the model .Analyst must be able to formulate a statement of purpose for each candidate class; if not, simply eliminate it.

1 Identifying Tentative Classes :

The following are guidelines for selecting classes in an application:

. Look for nouns and noun phrases in the use cases.
. Some classes are implicit or taken from general knowledge.
. All classes must make sense in the application domain; avoid computer implementation classes-defer them to the design stage.
. Carefully choose and define class names.

Identifying classes is an incremental and iterative process. This incremental and iterative nature is evident in the development of such diverse software technologies as graphical user interfaces, database standards, and even fourth-generation languages.

2 Selecting Classes from the Relevant and Fuzzy Categories :

The following guidelines help in selecting candidate classes from the relevant and fuzzy categories of classes in the problem domain.

a)Redundant classes. Do not keep two classes that express the same information. If more than one word is being used to describe the same idea, select the one that is the most meaningful in the context of the system. This is part of building a common vocabulary for the system as a whole. Choose your vocabulary carefully; use the word that is being used by the user of the system.

Eg: Registrar, University I/C

b) Adjectives classes. "Be wary of the use of adjectives. Adjectives can be used in many ways. An adjective can suggest a different kind of object, different use of the same object, or it could be utterly irrelevant. Does the object represented by the noun behave differently when the adjective is applied to it? If the use of the adjective signals that the behavior of the object is different, then make a new class".

For example : Single account holders behave differently than Joint account holders, so the two should be classified as different classes.

c)Attribute classes : Tentative objects that are used only as values should be defined or restated as attributes and not as a class. For example, Client Status and Details of Client are not classes but attributes of the Client class.

d) Irrelevant classes : Each class must have a purpose and every class should be clearly defined and necessary. You must formulate a statement of purpose for each candidate class. If you cannot come up with a statement of purpose, simply eliminate the candidate class.

As this is an incremental process. Some classes will be missing, others will be eliminated or refined later. Unless you are starting with a lot of domain knowledge, you probably are missing more classes than you will eliminate. Although some classes ultimately may become superclasses, at this stage simply identify them as individual, specific classes. Your design will go through many stages on its way to completion, and you will have adequate opportunity to revise it.

This refining cycle through the development process until you are satisfied with the results. Remember that this process (of eliminating redundant classes, classes containing adjectives, possible attributes, and irrelevant classes) is not sequential. You can move back and forth among these steps as often analysts likes.

Eg: Initial List of Noun Phrases for ATM BANK: Candidate Classes

The initial study of the use cases of the bank system produces the following noun phrases (candidate classes-maybe).

Account
Account Balance
Amount
Approval Process
ATM Card
ATM Machine
Bank
Bank Client ,
Card Cash ,
Check ,Checking ,
Checking Account,
Client ,
Client's Account ,
Currency ,

Dollar ,
envelope ,
Four Digits,
Fund ,
Invalid PIN,
Message ,
Money ,
Password ,
PIN ,
PIN Code ,
Record ,
Savings,
Savings Account ,
step,
Transaction
Transaction History

from the list bolded irreverent nouns can be eliminated

Reviewing the Redundant Classes and Building a Common Vocabulary :

We need to review the candidate list to see which classes are redundant. If different words are being used to describe the same idea, we must select the one that is the most meaningful in the context of the system and eliminate the others.

The following are the different class names that are being used to refer to the same concept:

Client, BankClient = BankClient (the term chosen)
Account, Client's Account = Account
PIN, PIN Code = PIN
Checking, Checking Account = Checking Account
Savings, Savings Account = Savings Account
Fund, Money = Fund
ATM Card, Card = ATM Card

Here is the revised list of candidate classes:

Account
Account Balance
Amount
Approval Process
ATM Card Bank
ATM machine
BankClient
Cash
Check
Checking Account
Currency
Dollar
Fund
Invalid PIN
Message
Password
PIN
Record
Savings Account
System
Transaction
Transaction History

In this list we don't have any classes with adjective nature, so no need to apply further elimination.

Reviewing the Possible Attributes

The next review focuses on identifying the noun phrases that are attributes, not classes. The noun phrases used only as values should be restated as attributes. This process also will help us identify the attributes of the classes in the system.

Amount: A value, not a class.
Account Balance: An attribute of the Account class.
Invalid PIN: It is only a value, not a class.
Password: An attribute, possibly of the BankClient class.
Transaction History: An attribute, possibly of the Transaction class.
PIN: An attribute, possibly of the BankClient class.

The list of classes after eliminating attributes are :

Account
Approval Process
ATM Card
ATM machine
Bank
BankClient
Cash
Check
Checking Account
Currency
Dollar
Fund
Message
Record
Savings account
System
Transaction

Reviewing the Class Purpose :

Identifying the classes that play a role in achieving system goals and requirements is a major activity of object-oriented analysis. Each class must have a purpose. Every class should be clearly defined and necessary in the context of achieving the system's goals. If you cannot formulate a statement of purpose for a class, simply eliminate it. The classes that add no purpose to the system have been deleted from the list.

The final candidate classes are these:

ATM Machine class: Provides an interface to the ViaNet bank.

ATMCard class: Provides a client with a key to an account.

BankClient class: A client is an individual that has a checking account and, possibly, a savings account.

Bank class: Bank clients belong to the Bank. It is a repository of accounts and processes the accounts' transactions.

Account class: An Account class is a formal (or abstract) class, it defines the common behaviors that can be inherited by more specific classes such as CheckingAccount and SavingsAccount.

CheckingAccount class: It models a client's checking account and provides more specialized withdrawal service.

SavingsAccount class: It models a client's savings account.
Transaction class: Keeps track of transaction, time, date, type, amount, and balance.

The major problem with the noun phrase approach is that it depends on the completeness and correctness of the available document, which is rare in real life. On the other hand, large volumes of text on system documentation might lead to too many candidate classes. Even so, the noun phrase exercise can be very educational and useful if combined with other approaches, especially with use cases as we did here.

2) COMMON CLASS PATTERNS APPROACH

The second method for identifying classes is using common class patterns, which is based on a knowledge base of the common classes.

The following patterns are used for finding the candidate class and object:

a) Concept class :

A concept is a particular idea or understanding that we have of our world. The concept class encompasses principles that are not tangible but used to organize or keep track of business activities or communications.
Example. Performance is an example of concept class object.

b) Events class

Events classes are points in time that must be recorded. Things happen, usually to something else at a given date and time or as a step in an ordered sequence. Associated with things remembered are attributes (after all, the things to remember are objects) such as who, what, when, where, how, or why.
Example. Landing, interrupt, request, and order are possible events.

c) Organization class

An organization class is a collection of people, resources, facilities, or groups to which the users belong; their capabilities have a defined mission, whose existence is largely independent of the individuals.

Example. An accounting department might be considered a potential class.

d) People class (also known as person, roles, and roles played class)

The people class represents the different roles users play in interacting with the application.

Example. Employee, client, teacher, and manager are examples of people.

e) Places class
Places are physical locations that the system must keep information about.

Example. Buildings, stores, sites, and offices are examples of places.

3) **USE-CASE DRIVEN APPROACH:**

IDENTIFYING CLASSES AND THEIR BEHAVIORS THROUGH SEQUENCE/COLLABORATION MODELING

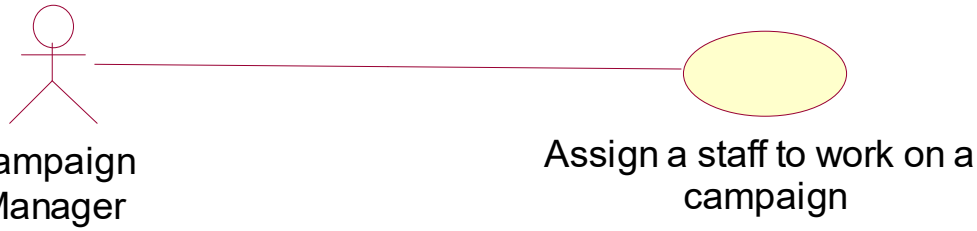
One of the first steps in creating a class diagram is to derive from a use case, via a collaboration (or collaboration diagram), those classes that participate in realizing the use case. Through further analysis, a class diagram is developed for each use case and the various use case class diagrams are then usually assembled into a larger analysis class diagram. This can be drawn first for a single subsystem or increment, but class diagrams can be drawn at any scale that is appro-priate, from a single use case instance to a large, complex system.

Identifying the objects involved in a collaboration can be difficult at first, and takes some practice before the analyst can feel really comfortable with the process. Here a collaboration (i.e. the set of classes that it comprises) can be identified directly for a use case, and that, once the classes are known, the next step is to consider the interaction among the classes and so build a collaboration diagram.

Consider the following use case Assign a staff to work on Campaign , to identify a set of classes

Consider the following use case Assign a staff to work on Campaign , to identify a set of classes.

Use case description: Assign staff to work on a campaign



Actor Action

1. None
3.The actor (a campaign manager) selects the client name .
5.Selects the relevant campaign .
7.Highlights the staff members to be assigned to this campaign

System Response

2. Displays list of client names.
4. Lists the titles of all campaigns for that client .
6.Displays a list of all staff members not already allocated to this campaign.
8.Presents a message confirming that staff have been allocated

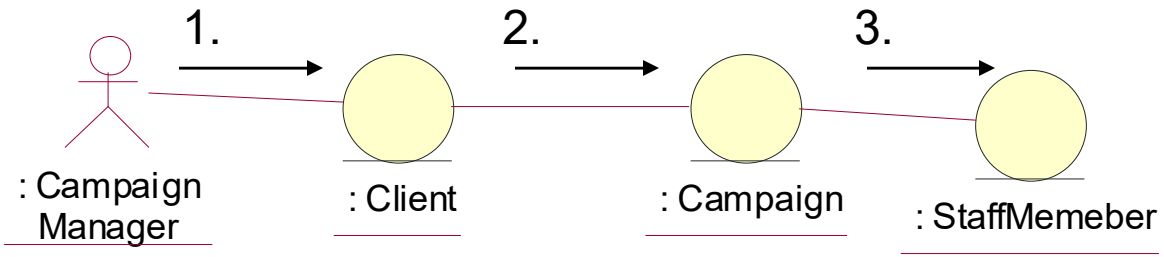
Here objective is to find a set of classes that can interact to realize the use case. **In** this case, we know from the use case diagram that the campaign manager is the actor for this use case. The use case description tells us that the manager selects a client name, the system responds by displaying all campaigns for that client, the manager selects a campaign, the system displays all staff not yet allocated to it, the manager selects staff to assign to the campaign and the system confirms the result. The objective of the use case is to allow the manager to assign staff to a campaign.

First begin by picking out from the description all the important things or concepts in the application domain. Our first list might include: campaign manager, client name, campaigns, client, staff. But we are only interested in those about which the system must store some information or knowledge in order to achieve its objectives. The campaign manager will be modelled as an actor.

For the purposes of this particular use case, it is unlikely the system will need to encapsulate any further knowledge about the actor. Here we can eliminate client name, since this is just part of the description of a client.

This leaves Client, Campaign and Staff Member in the collaboration.

Initial Collaboration Diagram for the Above USE CASE



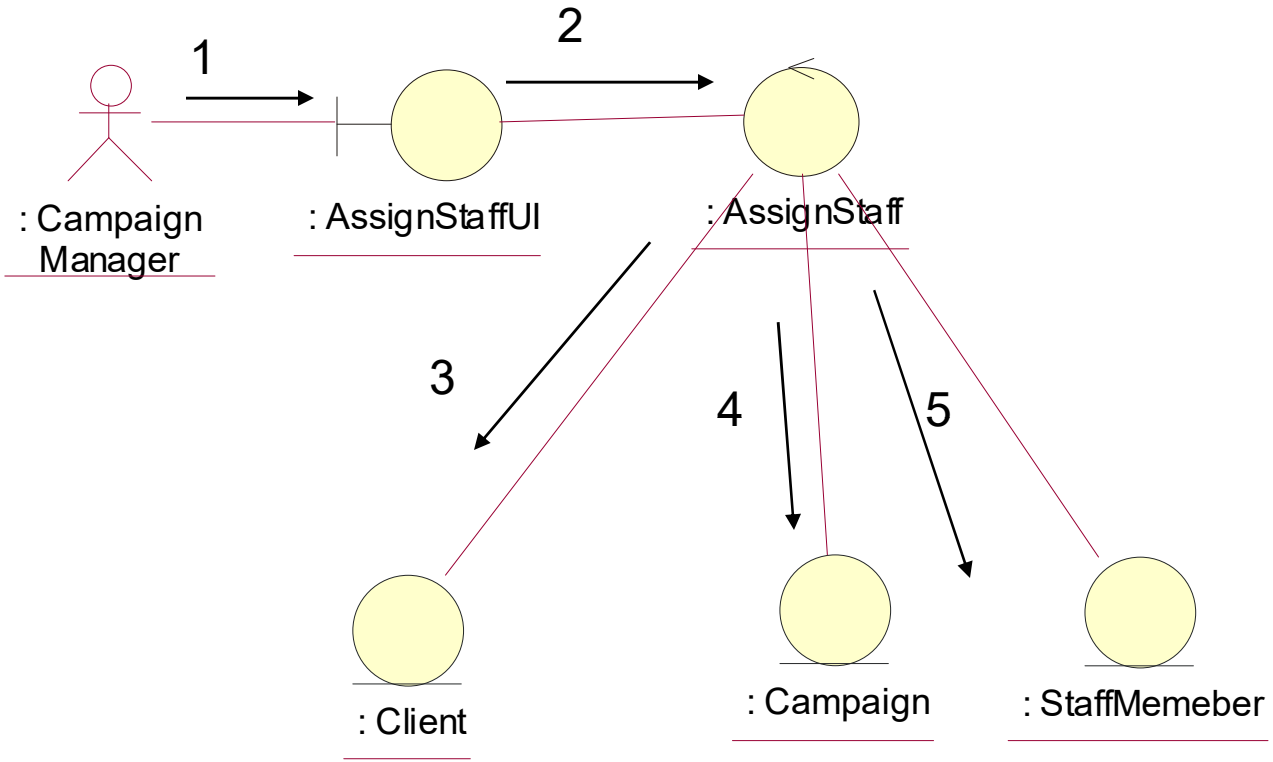
A collaboration is between individual object instances, not between classes. This is shown in the diagram by the convention of writing a colon before the class name, which indicates that this is an anonymous instance of the class, rather than the class itself. Messages between classes are shown by arrows, and their sequence is indicated by the number alongside. **In** this example, these are not yet labelled, although some those that can be most easily related to the use case description will probably soon be given names that correspond to responsibilities of the class to which the message is addressed.

In the collaboration diagram does not yet show any boundary or control objects, and these must be added. It is also based on certain assumptions about how the interaction between objects would take place, and we must make these assumptions explicit and question them.

Although the messages are not yet labelled, the numbers alongside the message arrows indicates their sequence. The diagram implies a linear flow of messages, along the following lines.

An initial message is directed to a Client, which is assumed to know its Campaigns and returns the list .Each Campaign is also assumed to know which Staff Members are currently assigned to it, and which are not. The Client object asks the selected Campaign object to return a list of unassigned Staff Members, and the Client passes this on to the interface. The Client object then asks the Campaign object to tell the selected Staff Member object to assign itself to the Campaign, by creating a link between them.

The following figure shows the collaboration diagram after this refinement adding a boundary object and control object.



A boundary object will be responsible for the capture of input from the user and display of results. All messages are now routed centrally through the control object. This means that no entity class needs to know anything about any other entity class unless this is directly relevant to its own responsibilities.

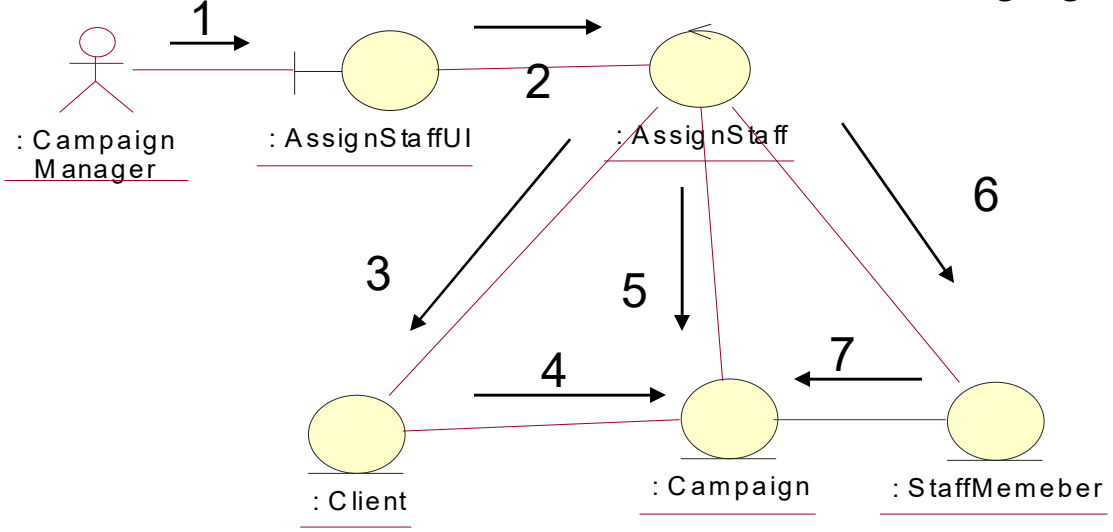
For example, a Client object is now assumed to know (and to tell the control object) which are its Campaigns, but it is no longer responsible for knowing which Staff Members are assigned to its Campaigns.

The above figure addresses one major issue, that, It seems reasonable to assume that a Client is responsible for knowing its own Campaigns. But the collaboration diagram now shows no communication between Clients and Campaigns, so it is not clear how this knowledge will be maintained. A similar consideration applies to the question of how a Campaign object maintains its knowledge of the Staff Members that have been assigned to work on it.

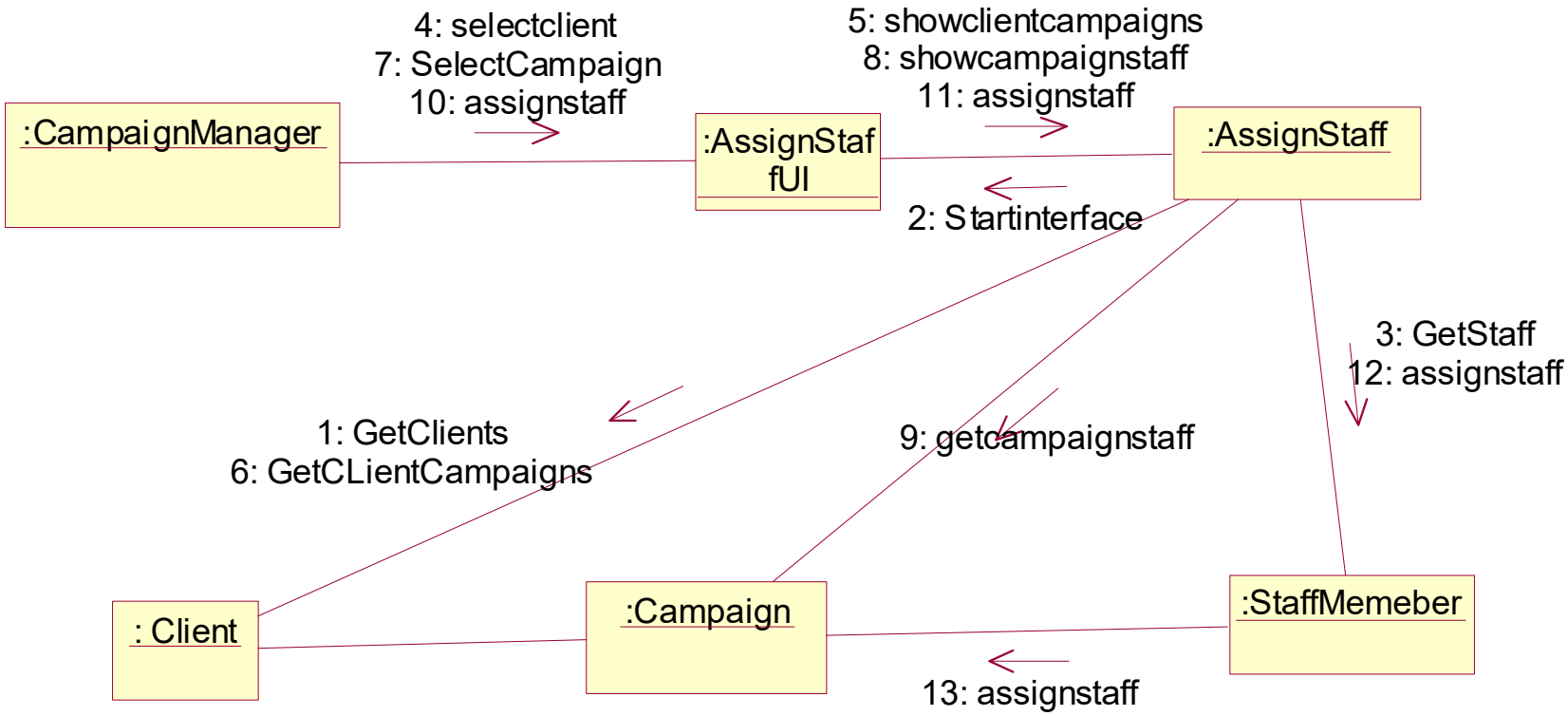
For this we can have another way that the interaction might work, and we could perhaps expand this as follows. The control object obtains a list of Clients and asks the boundary object to display them. It then asks a Client for a list of its Campaigns.

The Client may obtain some information directly from the Campaign objects themselves .The control object then directly asks the selected Campaign for information regarding which staff are currently assigned to it, and next asks unassigned Staff Members for their details, passing each bundle of information to the boundary object for display. The control object then instructs the selected Staff Member to assign itself, which it does by sending a message to the Campaign.

All these modifications are shown in the following figure.

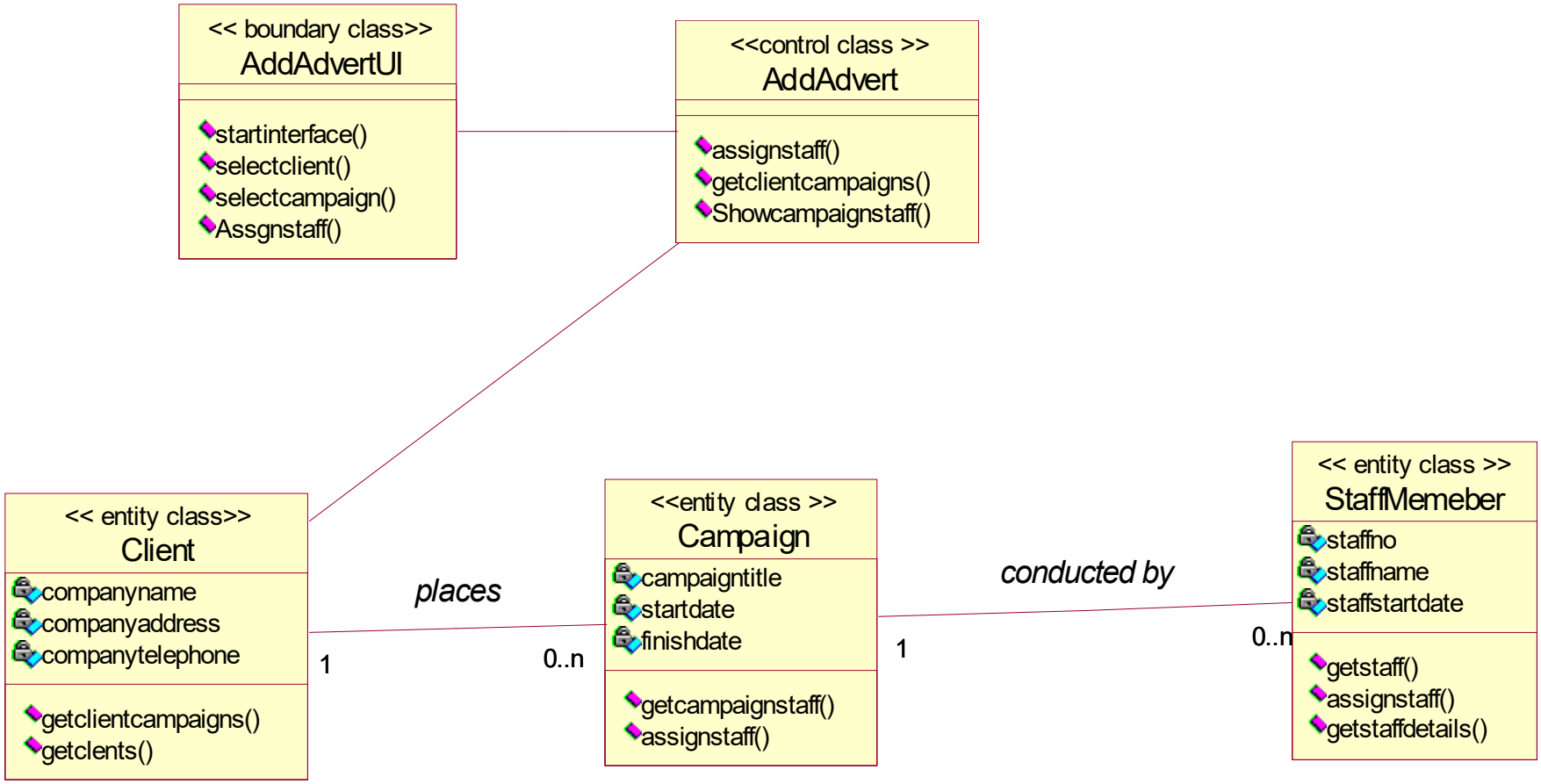


This diagram also uses the icon symbols for objects, but it could equally well be represented using rectangular object symbols and relevent messages to the objects as shown below.



From collaboration diagram to class diagram

The next step in the development of a requirements model is usually to produce a class diagram that corresponds to each of the collaboration diagrams. The class diagram that corresponds to the use case Assign staff to work on a campaign is shown below.



Collaboration diagrams are obtained by result of reasonably careful analysis, the transition is not usually too difficult.

The similarities & differences B/W Collaboration and class diagrams are :

First, consider the similarities :

Both show class or object symbols joined by connecting lines. **In** general, a class diagram has more or less the same structure as the corresponding collaboration diagram. **In** particular, both should show classes or objects of the same types. Any of the three analysis stereotype notations for a class can be used on either diagram, and stereotype labels can also be omitted from individual classes, or from an entire diagram.

Next, the differences are:

1.The difference is that an actor is almost always shown on a collaboration diagram, but not usually shown on a class diagram. This is because the collaboration diagram represents a particular interaction and the actor is an important part of this interaction. However, a class diagram shows the more enduring structure of associations among the classes, and frequently supports a number of different interactions that may represent several different use cases.

2. A collaboration diagram usually contains only object instances, while a.class diagram usually contains only classes.

3.The connections between the object symbols on a collaboration diagram symbolize links between objects, while on a class diagram the corresponding connections stand for associations between classes.

4. A collaboration diagram shows the dynamic interaction of a group of objects and thus every link needed for message passing is shown. The labelled arrows alongside the links represent messages between objects. On a class diagram, the associations themselves are usually labelled, but messages are not shown.

5. Finally, any of the three stereotype symbols can be used on either diagram, there are also differences in this notation.

When the rectangular box variant of the notation is used in a collaboration diagram it represents object instances rather than classes, is normally undivided and contains only the class name .On a class diagram, the symbol is usually divided into three compartments that contain in turn the class name, its attributes and its operations.

4) Class Responsibility collaboration Cards (CRC Cards)

At the starting , for the identification of classes we need to concentrate completely on uses cases. A further examination of the use cases also helps in identifying operations and the messages that classes need to exchange. However, it is easy to think first in terms of the overall responsibilities of a class rather than its individual operations.

A *responsibility* is a high level description of something a class can do. lt reflects the knowledge or information that is available to that class, either stored within its own attributes or requested via collaboration with other classes, and also the services that it can offer to other objects. A responsibility may correspond to one or more operations. lt is difficult to determine the appropriate responsibilities for each class as there may be many alternatives that all appear to be equally justified.

Class Responsibility Collaboration (CRC) cards provide an effective technique for exploring the possible ways of allocating responsibilities to classes and the collaborations that are necessary to fulfill the responsibilities .

CRC cards can be used at several different stages of a project for different purposes.

1.They can be used early in a project to help the production of an initial class diagram .

2.To develop a shared understanding of user requirements among the members of the team.

3. CRCs are helpful in modelling object interaction.

The format of a typical CRC card is shown below

Class Name:	
Responsibilities	Collaborations
Responsibilities of a class are listed in this section	*Collaborations with other classes are listed here, together with a brief description of the purpose of the collaboration*

CRC cards are an aid to a group role-playing activity . Index cards are used in preference to pieces of paper due to their robustness and to the limitations that their size (approx. 15cm x 8cm) imposes on the number of responsibilities and collaborations that can be effectively allocated to each class.

A class name is entered at the top of each card and responsibilities and collaborations are listed underneath as they become apparent. For the sake of clarity, each collaboration is normally listed next to the corresponding responsibility.

From a UML perspective, use of CRC cards is in analysing the object interaction that is triggered by a particular use case scenario. The process of using CRC cards is usually structured as follows.

1.Conduct a session to identify which objects are involved in the use case.
2. Allocate each object to a team member who will play the role of that object.
3. Act out the use case.

This involves a series of negotiations among the objects to explore how responsibility can be allocated and to identify how the objects can collaborate with each other.

4.Identify and record any missing or redundant objects.

Before beginning a CRC session it is important that all team members are briefed on the organization of the session and a CRC session should be preceded by a separate exercise that identifies all the classes for that part of the application to be analysed.

The team members to whom these classes are allocated can then prepare for the role playing exercise by considering in advance a first-cut allocation of responsibilities and identification of collaborations. Here , it is important to ensure that the environment in which the sessions take place is free from interruptions and free for the flow of ideas among team members.

During a CRC card session, there must be an explicit strategy that helps to achieve an appropriate distribution of responsibilities among the classes. One simple but effective approach is to apply the rule that each object should be as lazy as possible, refusing to take on any additional responsibility unless instructed to do so by its fellow objects.

During a session conducted according to this rule, each role player identifies the object that they feel is the most appropriate to take on each responsibility, and attempts to persuade that object to accept the responsibility. For each responsibility that must be allocated, one object is eventually persuaded by the weight of rational argument to accept it. This process can help to highlight missing objects that are not explicitly referred to by the use case description. When responsibilities can be allocated in several different ways it is useful to role-play each allocation separately to determine which is the most appropriate. The aim normally is to minimize the number of messages that must be passed and their complexity, while also producing class definitions that are cohesive and well focused.

Consider CRC exercise for the use case Add a new advert to a campaign. The use description is repeated below for ease of reference.

This use case involves instances of Client, Campaign and Advert, each role played by a team member. The resulting CRC cards are shown in the following figure.

Class Name : Client	
Responsibilities	Collaborations
Provide client information *Provide campaign details*	*Campaign provides campaign details.*

CRC card for Client class in ADD A NEW ADVERT CAMPAIGN

Class Name : campaign	
Responsibilities	Collaborations
Provide Campaign information *Provide list of adverts* Add a new advert:	*Advert provides advert details* Advert constructs new object

CRC card for Campaign class in ADD A NEW ADVERT CAMPAIGN

CRC card for Advert class in ADD A NEW ADVERT CAMPAIGN

Class Name : *Advert*	
Responsibilities	Collaborations
Provide advert details *Construct adverts.*	

Other approaches to finding objects and classes

As use cases are the best place to look for entity objects, and the best way to find them is through thinking about interactions between them that support the use case.

The following are the other approaches used to identify objects and classes :

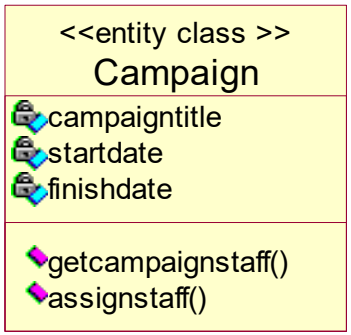
One approach is to first develop a *domain model* - *an* analysis class model that is independent of any particular use cases. For example, the domain model is a significant feature of the ICONIX method . In the approach ,the development of a domain model is considered primarily as a step that follows the development of class diagrams for each use case .

Next, we assemble the various class diagrams that result from use case realization into a single analysis class diagram. This may consist of a single package of entity classes, with boundary and control classes typically located in separate packages. With large systems, the domain model alone may comprise several distinct packages, each representing a different functional subsystem of the overall system.

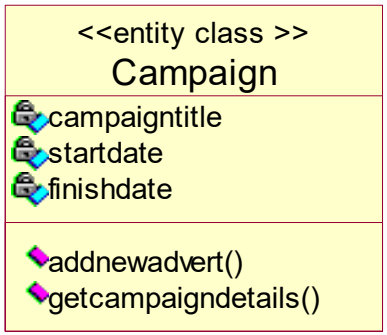
There exists usually a technical difficulty in this step. All we really have to do is to place the various entity classes into a single class diagram. Where we find that we have defined the same class in different ways to meet the needs of different use cases, we simply assemble all of the operations and attributes into a single class definition.

For example, consider the Campaign class as seen in relation to Add a new advert to a campaign and Assign staff to work on a campaign. Different use cases have suggested different operations. Putting these together results in a class that is capable of meeting the needs of both use cases. When we consider other use cases too, a more complete picture of the class emerges. These stages are represented below.

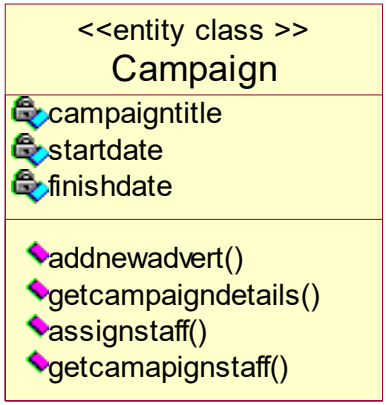
1. Campaign Class for Assign a staff to work on a campaign



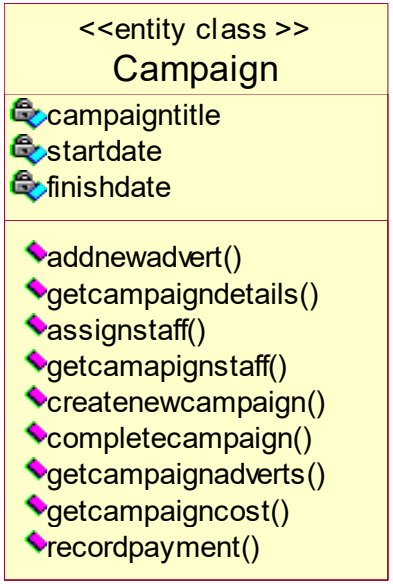
2. Campaign class for Add a new advert to a campaign



3. Campaign class that meets the conditions for both the use cases



3. A Campaign class that meets the requirements of all the use cases in AGATE Ltd



Sometimes a domain model will already exist, and in other cases it will make sense to produce a domain model before producing any use cases. The key to success is iterative refinement of the models, however they are produced in the first place.

Second approach is reviewing any background documentation which is gathered during the fact-finding stage. This type of attempt at class modelling, can discover more classes, due to your clearer understanding of the problem.

Ideally, user representatives will be closely involved in discussing and developing the class diagram. Nowadays users often work alongside professional analysts as part of the project team. Most projects are a learning experience for everyone involved, so it is not unusual for users' understanding of their own business activity to grow and develop, and it is likely that users will identify a number of additional classes that were not apparent at first.

Approach may be what ever for the identification of classes, it helps to have a general idea of what you are looking for. Some pointers have been developed over the years that help to discriminate between likely classes and unlikely ones.

The following categories are kinds of things and concepts that are more likely than others to need representation as an entity object or class. These textual descriptions and definitions will be an important supplement to the diagrams. Check the requirement list carefully as it grows to identify classes and objects potentially..

Category	Examples
People	Mr Bean(a campaign manager), Dilip (a copywriter)
Organizations	Jones & Co (a forklift truck distributor), the Soong Motor Company, Agate's Creative Department
Structures	Team, project, campaign, assembly
Physical things	Fork-lift truck, electric drill, tube of toothpaste
Abstractions of people	Employee, supervisor, customer,
Abstractions of physical things	vehicle, hand tool, retail goods
Conceptual things	Campaign, employee, rule, team, project

Next, these are some guidelines to help you to prune out unsuitable candidate classes. For each item on the list apply the following questions for the clarification.

1. *Is it beyond the scope* of *the system?*
2. *Does it refer to the system as a whole?*
3. *Does it duplicate another class?*
4. *Is it too vague?*
5. *Is it too specific?*
6. *Is it too tied up with physical inputs and outputs?*
7. *Is it really an attribute?*
8. *Is it really an operation?*
9. *Is it really an association?*

Adding and locating attributes :

Many attributes will already appear in the use case descriptions. Others will become apparent as analyst thinks about the model in more detail. The simple rule is that attributes should be placed in the class they describe.

This usually presents few problems.

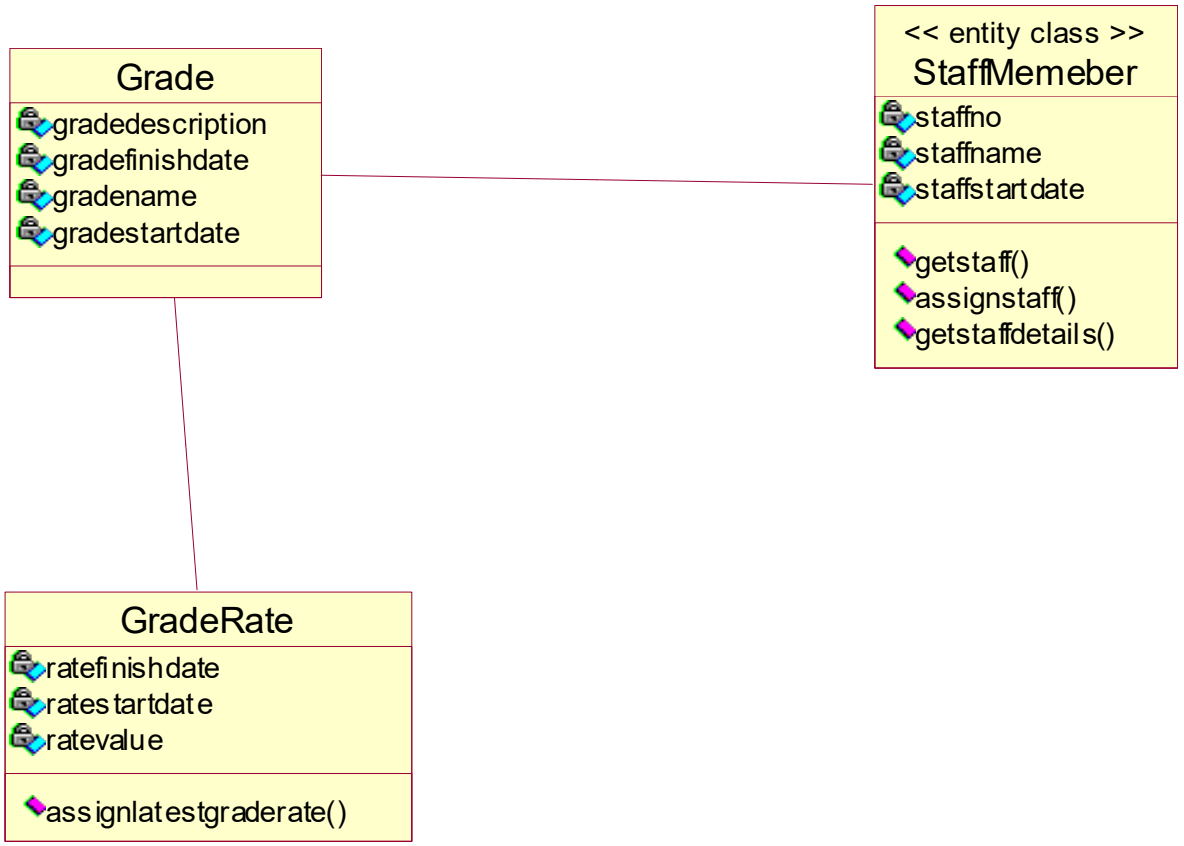
For example, the attributes staff No, staff Name, and staffStartDate all clearly describe a member of staff, so should be placed in the Staff class.

Sometimes it is more difficult to identify the correct class for an attribute. The attribute may not properly belong to any of the classes you have already identified.

Consider the following observations are madein a preliminary analysis which yields the following list of classes and attributes:

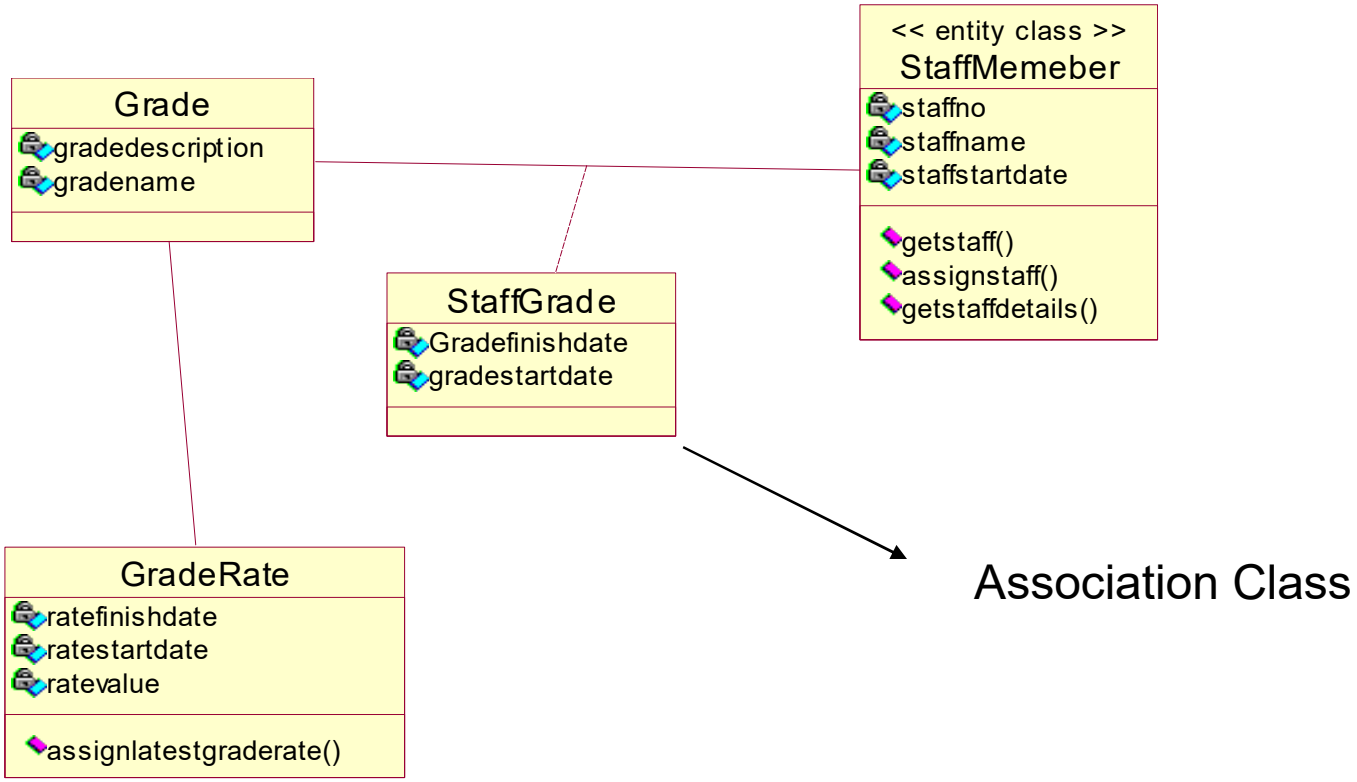
classes: Staff Member, Grade, Rate;
attributes: gradeStartDate, gradeFinishDate, rateStartDate, rateFinishDate, rateValue.

An initial class diagram may looks as shown below which is incomplete



Here , One problem is where to put the attributes gradeStartDate and gradeFinishDate. These could be placed in Grade, but this would commit it to recording multiple start and finish dates. There may be also many members of staff associated with a grade. The computer system must be able to identify the member of staff to which each date applies, so the structure of dates that might need to be stored could grow quite complex.

For this, create an additional class (called an association class) specifically to provide these attributes with a home. This is shown as below.



Adding associations :

Identify the associations by considering logical relationships among the classes in the model. Associations may be found in use case descriptions, and other text descriptions of the application domain, as stative verbs (which express a permanent or enduring relationship), or as actions that need to be remembered by the system.

'Customers *are responsible for* the conduct of their account' is an example of the first, while 'purchasers *place* orders' is an example of the second.

But this is not a very reliable way of finding associations. Some will not be mentioned at all, while others may be too easily confused with classes, attributes or operations. A full understanding of the associations in a class model can only be reached later by analysing the interaction between different classes.

Finding operations :

Operations are really a more detailed breakdown of the high-level system responsibilities already modelled as use cases. An operation can be thought of as a small contribution of one class to achieving the larger task represented by a whole use case. They are sometimes found as action verbs in use case descriptions, but this picture is likely to be fairly incomplete until the interaction between classes has been under stood in more depth. For this purpose UML interaction diagrams very helpful.

Preliminary allocation of operations :

Before attempting to allocate operations to specific classes, it is needed to remember each entity class is only a representation of something in the application domain. As an analyst, to build a logical model that helps to understand the domain, not necessarily a replica that is perfect in every detail.

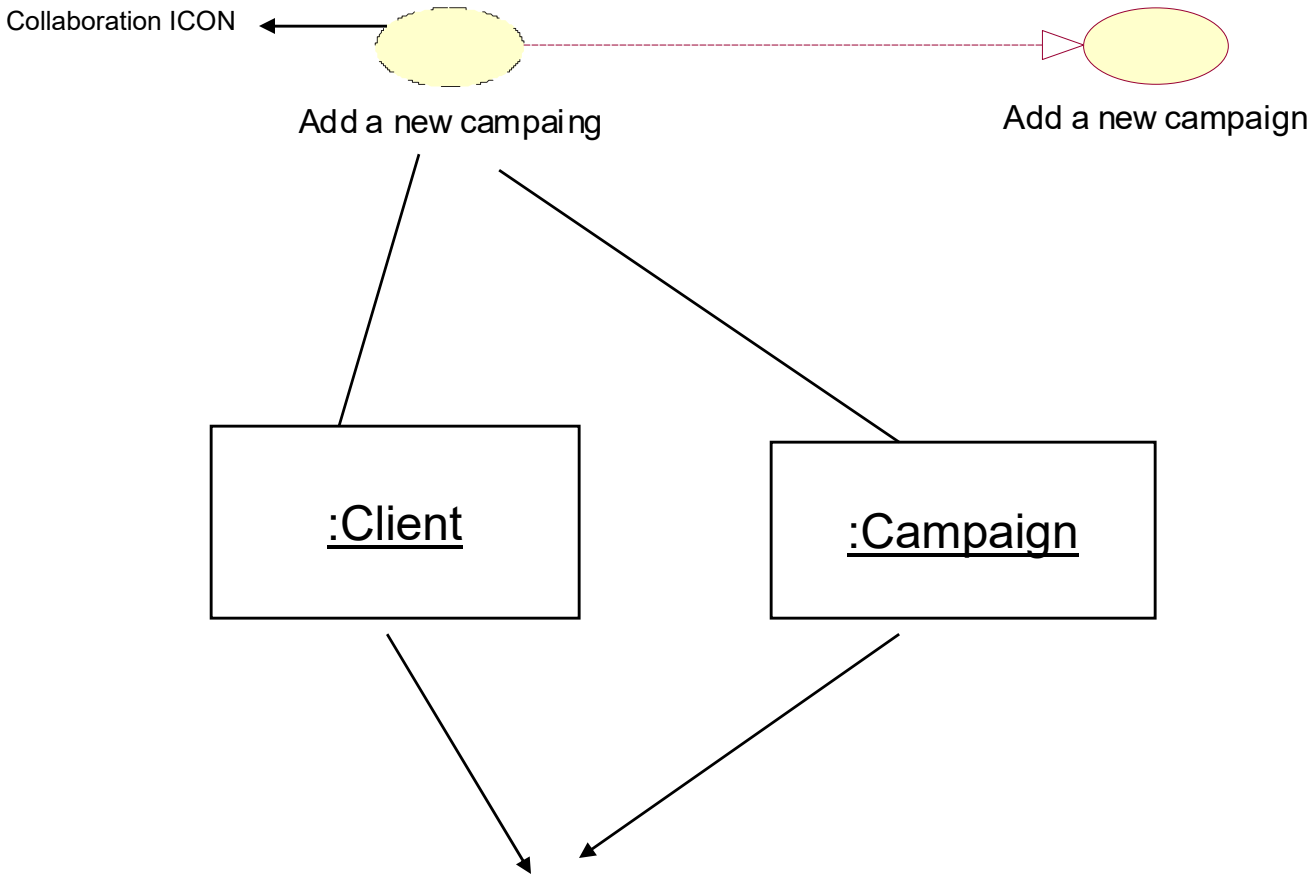
The following two guidelines helps in deciding which class to locate each operation in, but there is not a single answer- its only a satisfactory assumption.

1. Imagine each class as an independent actor, responsible for doing or knowing
certain things. For example, we might ask 'What does a staff member need to know or need to be able to do in this system?'
2. Locate each operation in the same class as the data it needs to update or access.

As a general comment on this stage, the important thing is not to expect to get things right at the first attempt. Analyst will always need to revise his assumptions and models as his understanding grows.

AGATE Ltd. – CASE STUDY

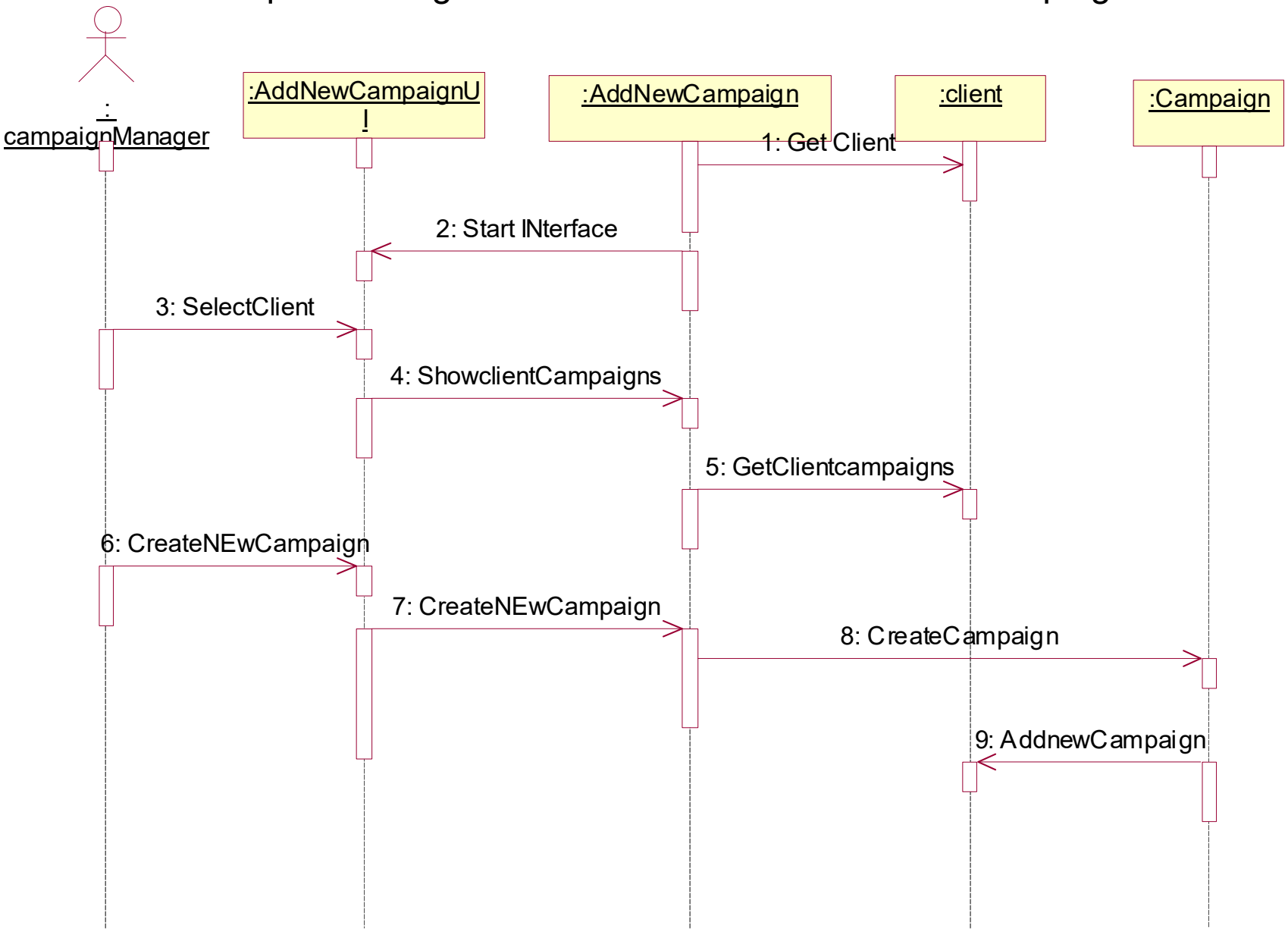
1. USE CASE Realization for a ADD NEW CAMPAIGN use case



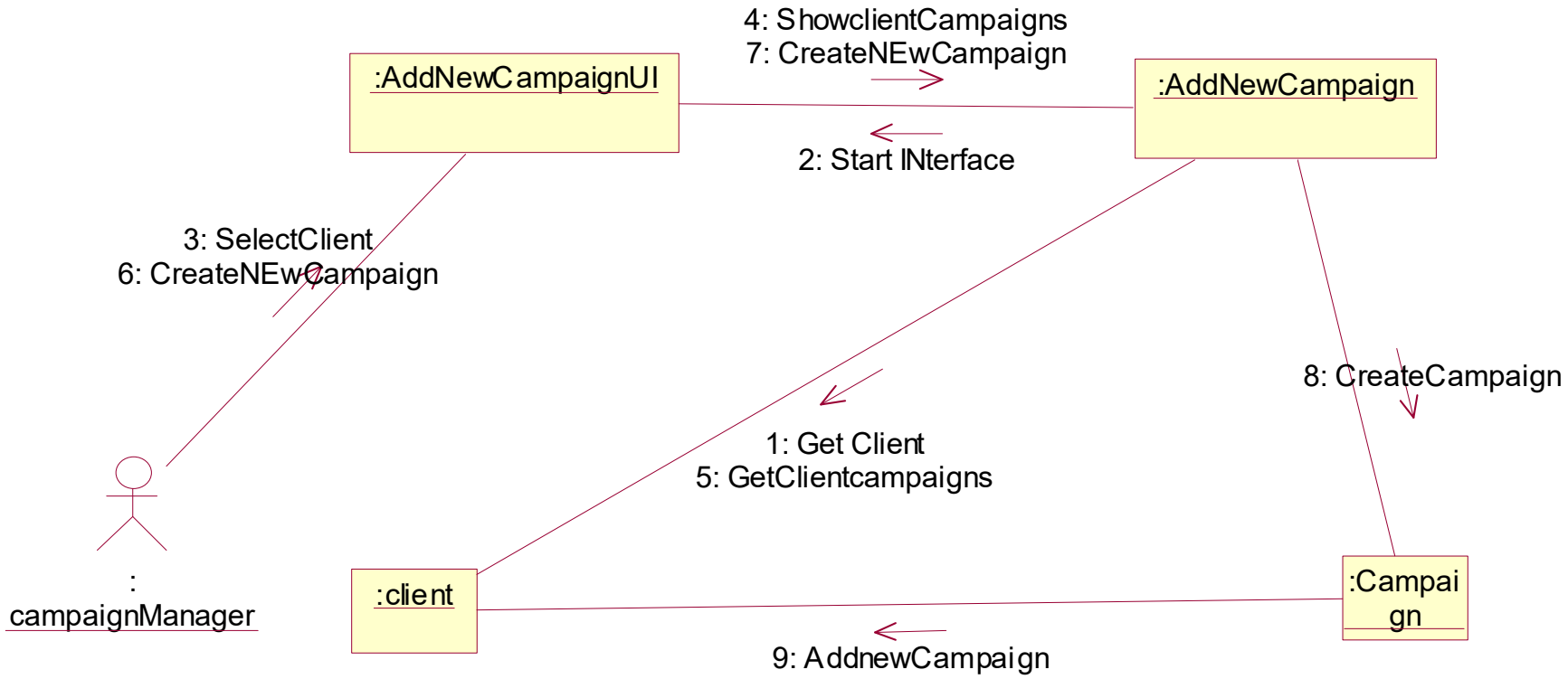
These are the Essential classes needed to implement this use case

Next figures shows realization of this use case

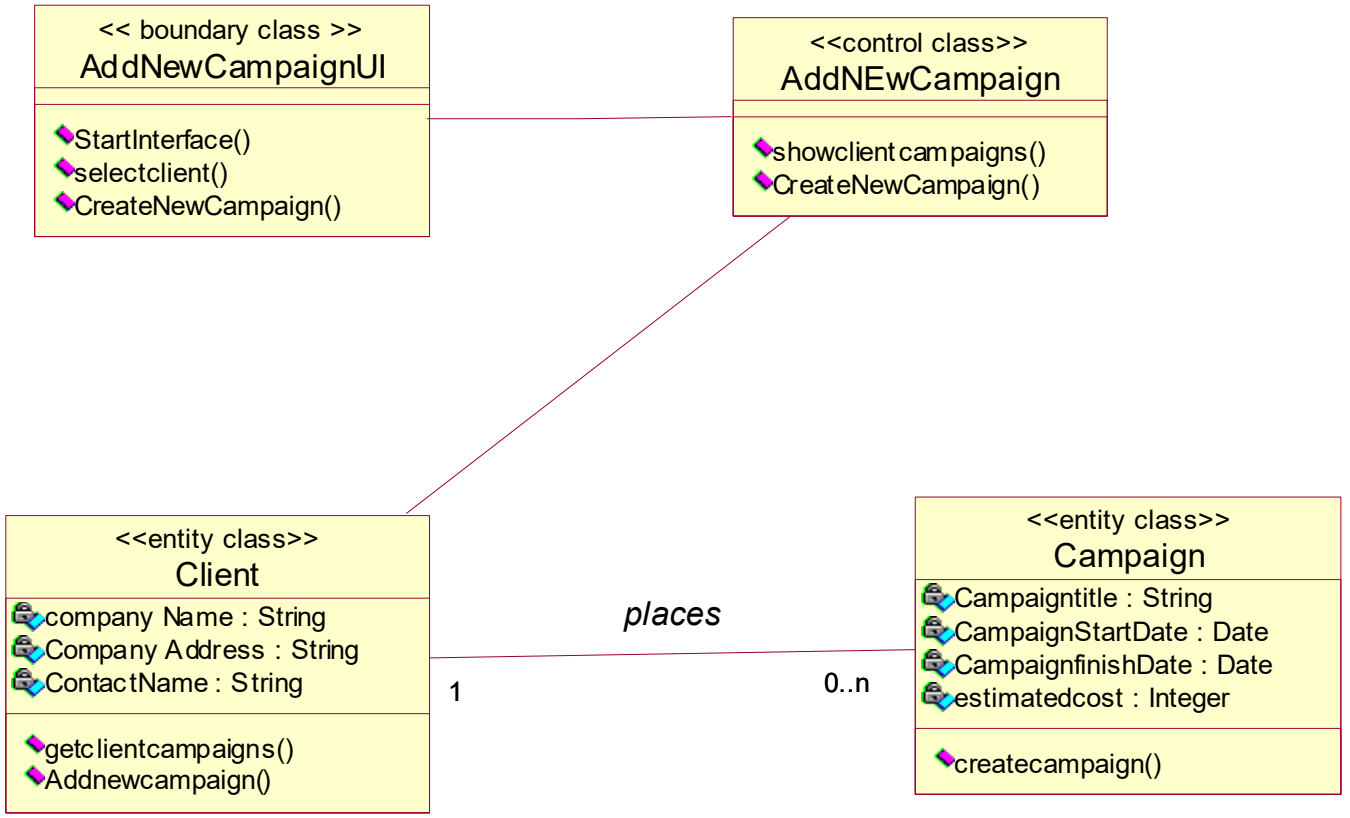
Sequence diagram for the use case Add a new campaign



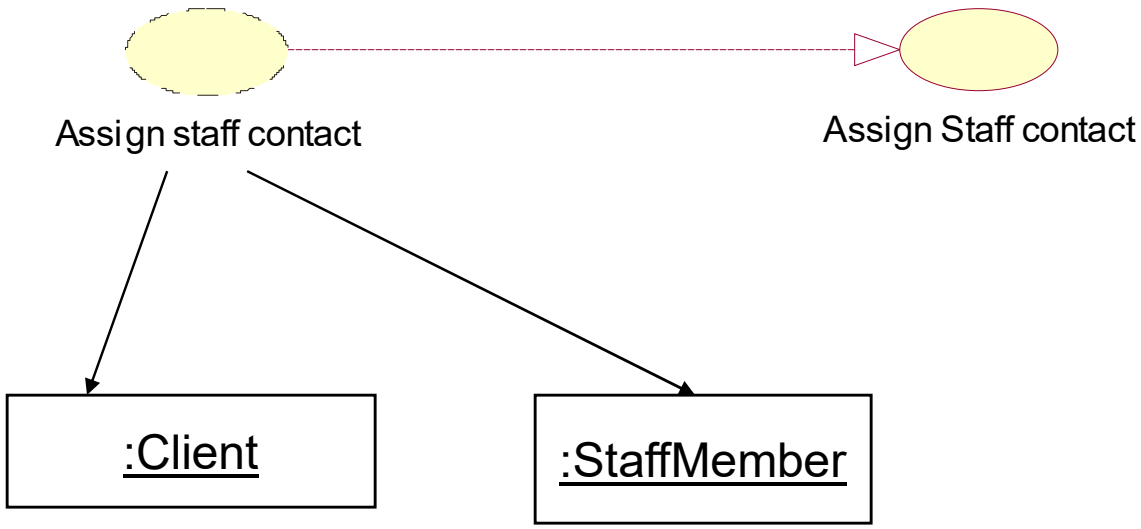
Collaboration diagram for the use case Add a new campaign



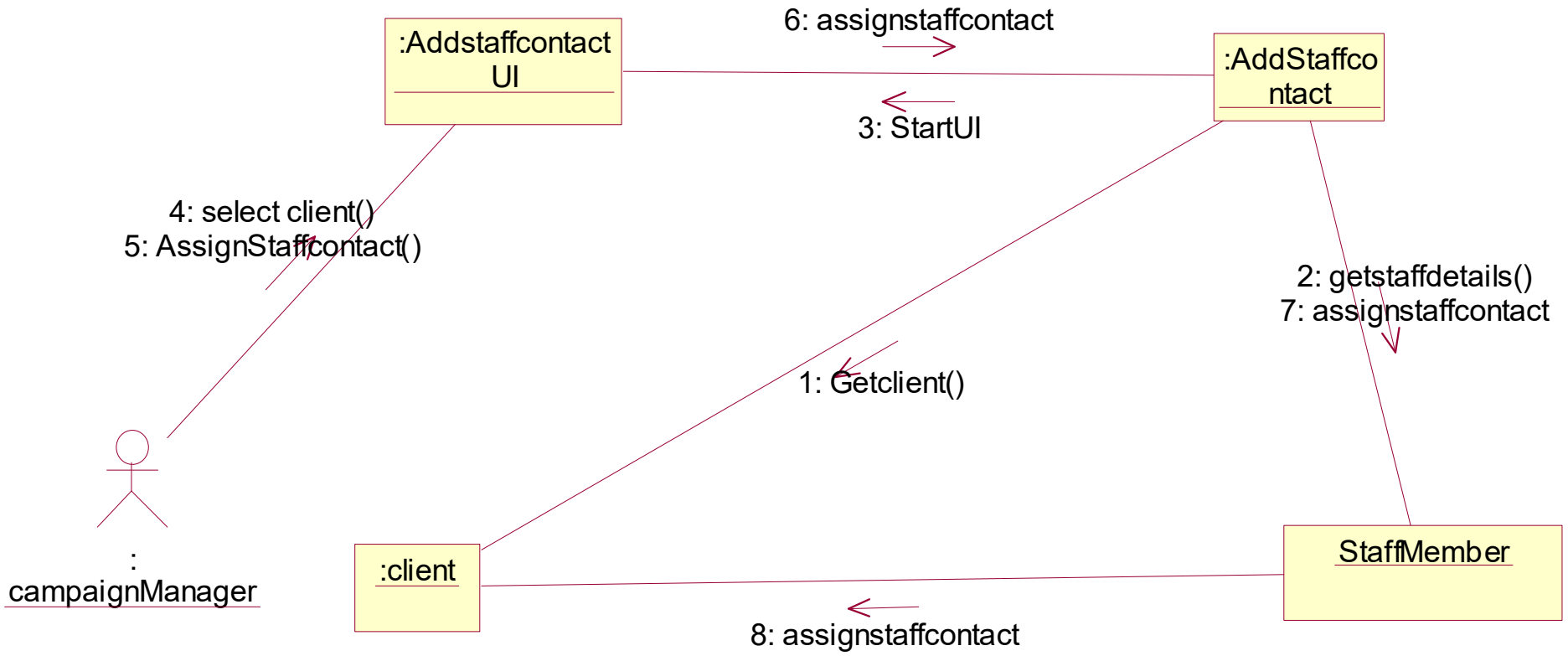
Class diagram for the use case Add a new campaign



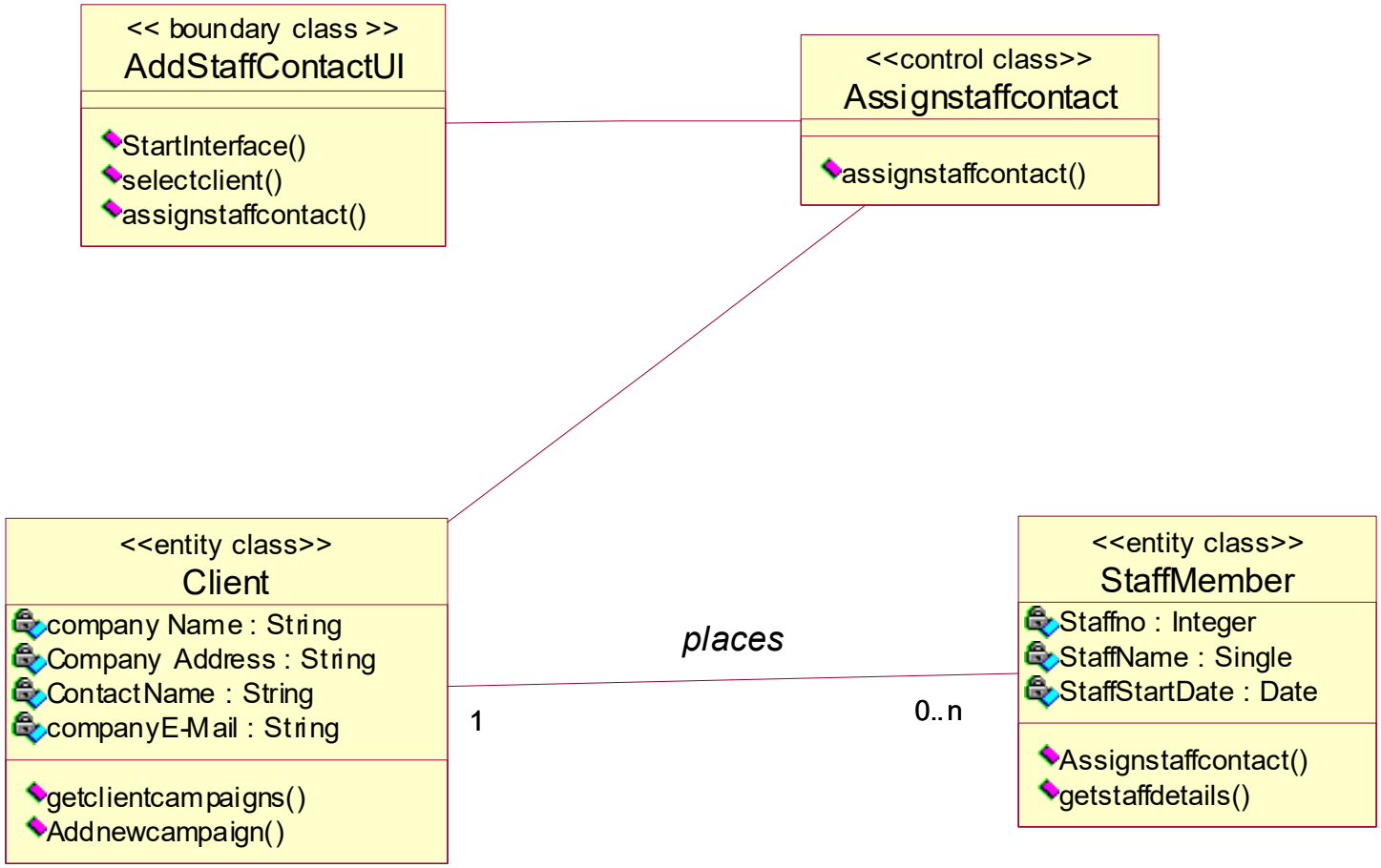
2. USE CASE Realization for a Assign staff contact use case



Collaboration diagram for the use case Assign staff contact :



Class diagram for the use case Assign staff contact



Exercises :

Give the realizations for the following use cases in Agate Ltd.

1) Check Campaign Budget
2) To record completion of a campaign

Combine Class diagram for four use cases Add a New Campaign, Assign a Staff contact, Check Campaign budget, Record completion of a campaign