8. REFINING THE REQUIREMENTS MODEL

Once the requirements have been identified and the various use case class diagrams have been assembled into a single analysis class diagram which shows all relevant classes and associations , the next step in the life cycle of a development of a system is to refine this requirements model.

This will be undertaken with a particular view to creating the conditions for developing components that can be re-used either within the project or on future projects. In this process we need to know

- What do you mean by component ?
- Role of Abstraction in an OO approach for an identifying and building re-usable components .
- A composition which is a special type of abstraction.
- S/W development patterns.

Component Based Development :

Nowadays in an industry the use of standard components, and even standard designs, is not common.

To take a familiar example, an architect may design many different houses, each unique in its own particular ways. But it would be very unusual to design every component from scratch. The architect typically chooses components from a catalogue, each house being assembled from a standard set of bricks, roofing timbers, tiles, doors, window frames, electrical circuits,…etc.

Thus, while one house may be completely unlike another in overall appearance, floor plan and number of rooms, the differences between them lie in the way standard components have been assembled.

Software components: the problems :

The information systems development organizations believe that the analysis of requirements should begin from scratch on every new project. **In** one sense, this is necessary, since at first we know nothing about what the requirements for a new system are. It is also advantageous if it encourages analysts to take account of the unique characteristics of the proposed system and its environment.

But starting from a position of knowing nothing can also have significant disadvantages. Effort may be wasted on finding new solutions to old problems that have already been adequately solved by others in the past. While it is important that real differences between two projects should not become blurred, this should not prevent the team from capitalizing on successful past work, provided that it is relevant to the current problem.

In the majority of cases, those analysts, designers and programmers who seem unnecessarily to reinvent the wheel do not do so deliberately. Good professionals have always tried to learn as much as possible from experience, both their own and that of their colleagues. Programmers have built up extensive libraries that range from personal collections of useful subroutines, to commercially distributed products that contain large numbers of industry-standard components.

One example of this is the use of code libraries such as .DLL (Dynamic Link Library) files in Microsoft Windows. So why do some software developers carry on reinventing so many wheels?

One reason for this is the **NIH** ('Not Invented Here') syndrome, which mainly seems to afflict programmers.

Another reason is the functionally-based decomposition of the modelling techniques in structured analysis, which primarily affects analysts as well as designers also.

The NIH syndrome

In spite of all the library resources available today, some professional programmers still fall prey to the NIH syndrome. This is the attitude of one who thinks: 'I don't trust other people's widgets-even those that appear to work, suit my purpose and are affordable-I want to invent my own widgets anyway.'

This is understandable in someone who enjoys a technical challenge, or has reasons not to trust the work of others, but it usually does not make good commercial sense.

One remedy can be found in object orientation, partly because of the different attitude to program development it engenders, but also partly because object orientation actually makes it easier to use library components.

Model organization :

Analysts suffer from the NIH syndrome too, but the biggest obstacle to reuse of successful analysis work has been the way that structured models are organized.

It is difficult enough to create a model that is useful at other stages of one single project. It is even harder to create a structured model that is useful on a completely different project. This is partly because structured models are partitioned according to functions, which are a particularly volatile aspect of a business domain.

There is also little in structured analysis models such as data flow diagrams that explicitly encourages encapsulation. This is an area where object-orientation can make a distinctive contribution to the reuse of requirements analysis.

How object-orientation contributes to reuse :

Object-oriented software development tackles the problem of achieving reuse in ways that resemble the practice of other industries that use standard components.

The aim is to develop components that are easy to use in systems for which they were not specifically developed. Ideally, software analysts should be in the position described for an architect, free to think about how their client intends to use the system, without needing to worrying about how individual components are built.

One of the keys is the encapsulation of internal details of components, so that other components requesting their service need not know how the request will be met. This allows different parts of the software to be effectively isolated in operation and greatly reduces the problems in getting different sub systems to interact with each other, even when the sub-systems have been developed at different times or in different languages, and even when they execute on different hardware platforms. Sub-systems that have been constructed in this way are said to be *decoupled* from each other.

The effect can be scaled up to the level of complex sub systems, by applying the same principle of encapsulation to larger groups of objects. Any part of a software system or, by extension, a model of one- can be considered for reuse in other contexts, provided certain criteria are met.

- A component should meet a clear- cut but general need (in other words, it delivers a coherent service) .

- A component should have a simple, well-defined external interface.

In theory, reusable components can be designed within any development approach, but object-orientation is particularly suited to this task. Well chosen objects meet both of the criteria above, since an object requesting a service need only know the message protocol, and the identity of an object that can provide it!.

Another important aspect of object-orientation is the way that models, and hence also code, are organized.

Requirements reuse

When we write of a 'reusable requirement', we really mean a reusable *model* of a requirement. This is one of the least developed areas of software reuse. In any case, only parts of any model are likely to be reusable. But the key to reuse in requirements modelling is that models are organized so that they abstract out (hide) those features of a requirement that are not necessary for a valid comparison with a similar requirement on another project.

Second, the whole point of reuse is to save work, so it should also not be necessary to develop a full model of the second require ment in order to make the comparison.

Finally, any relevant differences between the two requirements being compared should be clearly visible and it should not be necessary to develop a full model of the second requirement in order to see these either.

Generalization

As a simple example of generalization that enables reuse of a component, The design of a program, which makes it suitable for use in many different situations, rather than being restricted to one narrow set of circumstances, is an example of abstraction by generalization.

The Object-oriented developers have one significant advantage that, Using inheritance, a 'software architect' has a way of spawning new products from old ones with minimal effort.

But inheritance provides a way of designing and building the larger part of a new software component in advance, leaving only the specialized details to be completed at a later stage. This is because, in a class hierarchy, those characteristics that are shared by subclasses are maintained at the superclass level, and are instantly available to any subclass when required.

Composition

Composition is a type of abstraction that encapsulates groups of classes that collectively have the capacity to be a reusable sub-assembly. Unlike generalization, the relationship is that between a whole and its parts.

The essential idea is that a complex whole is made of simpler components. These, while less complex than the whole, may themselves be made of still less complex sub-assemblies, elementary components or a mixture of the two.

A simple example of the usefulness of the idea can be seen in a house-builder fitting a window frame to a new house. Like many other house components, window frames are delivered to site as ready-assembled units. All internal details of the sub-assembly are 'hidden' from both architect and builder, in the sense that they do not need to think about them.

Composition, also known as the a-part-of, is a form of aggregation with strong ownership to represent the component of a complex object. Composition also is referred to as a part-whole relationship.

Composition in UML

Composition (or *composite aggregation)* is based on the rather less precise concept of *aggregation,* which is a feature of many object-oriented programming languages.

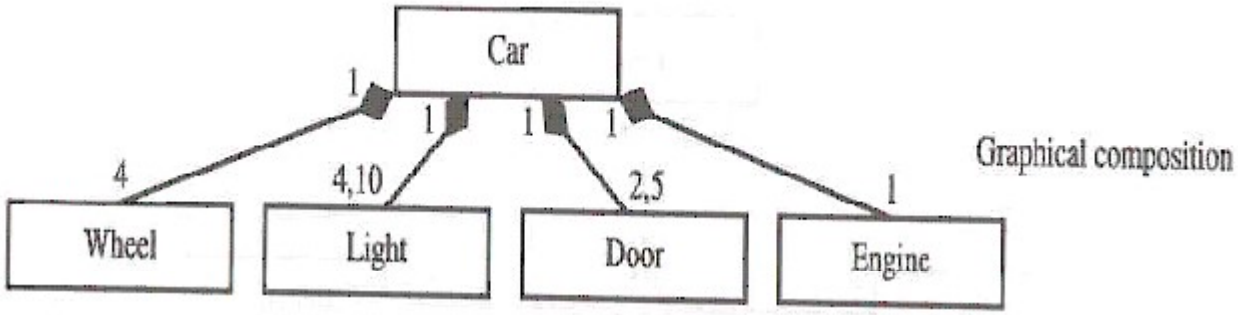
The UML Specification is deliberately rather vague about aggregation, but says the following about the relationship between aggregation and composition.

Composite aggregation is a 'strong form of aggregation which requires that a part instance be included in at most one composite at a time and that the composite object has sole responsibility for the disposition of its parts. This means that the composite object is responsible for the creation and destruction of its parts.'

While composition and aggregation may be identified during requirements analysis, their main application is during design and implementation activities, where they can be used to encapsulate a group of objects as a potentially re-usable sub-assembly.

This is more than just a matter of labelling the group of objects with a single name. The fact that they are encapsulated is much more important. The external interface for a composition is actually the interface of the single object at the 'whole' end of the association. Details of the internal structure of the composition-that is, what other objects and associations it contains-remain hidden from the client.

General example for composition :

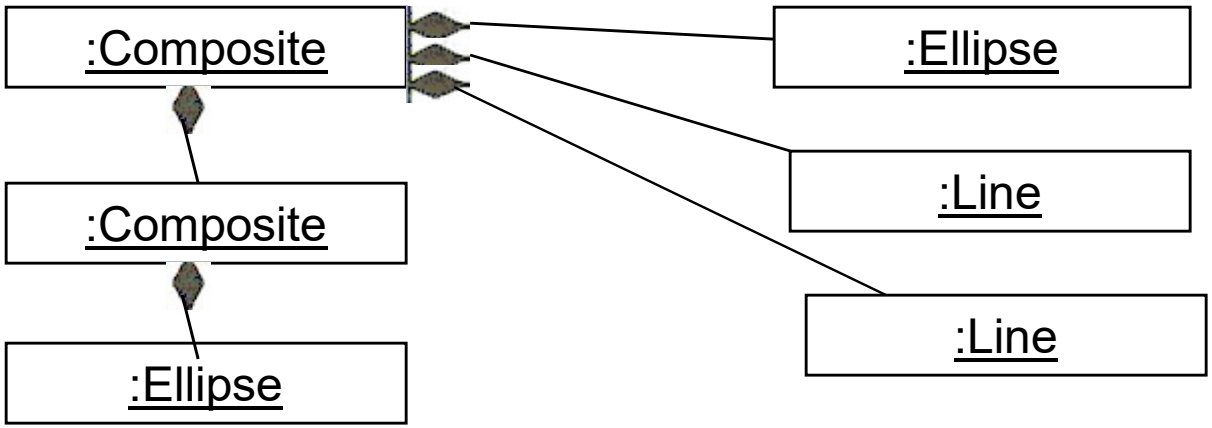


On a practical level, composition is familiar to users of most common computer drawing packages.

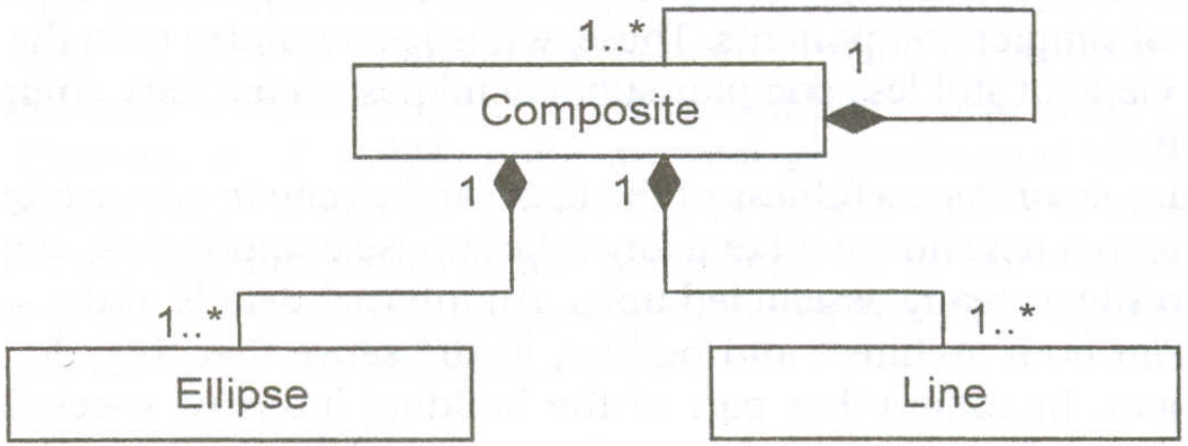
For example, many of the drawings were prepared or edited using a widely used drawing package.

Within this application, several drawing objects can be selected and grouped. They then behave exactly like a single object, and can be sized, rotated, copied, moved or deleted with a single command. The following Figures shows this type of composition as both objects and classes.

Some objects in a computer drawing package.



Composition used in class diagram to represent composite objects.



This example is nested one - the composition itself contains a further composition. In just the same way that a composite drawing object can only be handled as a single drawing component, the 'part' objects in a composition structure can usually not be directly accessed, and the whole presents a single interface to other parts of the system.

This notation is similar to a simple association, but with a diamond at the 'whole' end. The diamond is filled with solid colour to indicate composition, and left unfilled for aggregation.

Adding further Structure to the Class Diagram :

consider how to add structure to the class diagram that will help with reuse at later stages of development. In this First, we need to concentrate on generalization, since it is the more useful of the two concepts for this purpose. Then we need to consider how to model a structure that combines generalization and aggregation.

Modelling Generalization :

The following table shows an interview carried out by an analyst in the Agate case study. Analysts main objective was to understand more about different types of staff. These facts highlights some useful information that must be modelled appropriately:

- There are two types of staff,
- Bonuses are calculated differently and
- Different data should be recorded for each type of staff.

Analyst's note of the differences between Agate staff types :

Brief interview with Finance Director and Analyst :

Purpose - clarification of staff types.

Asked about staff types

- only two types seem relevant to system -creative staff (C) and admin staff (A)

How do they differ?
- main difference is bonus payment ...

1. (C) bonus calculated on basis of campaign profits (only those campaigns they worked on)

2. (A) paid rate based on average of all campaign profits

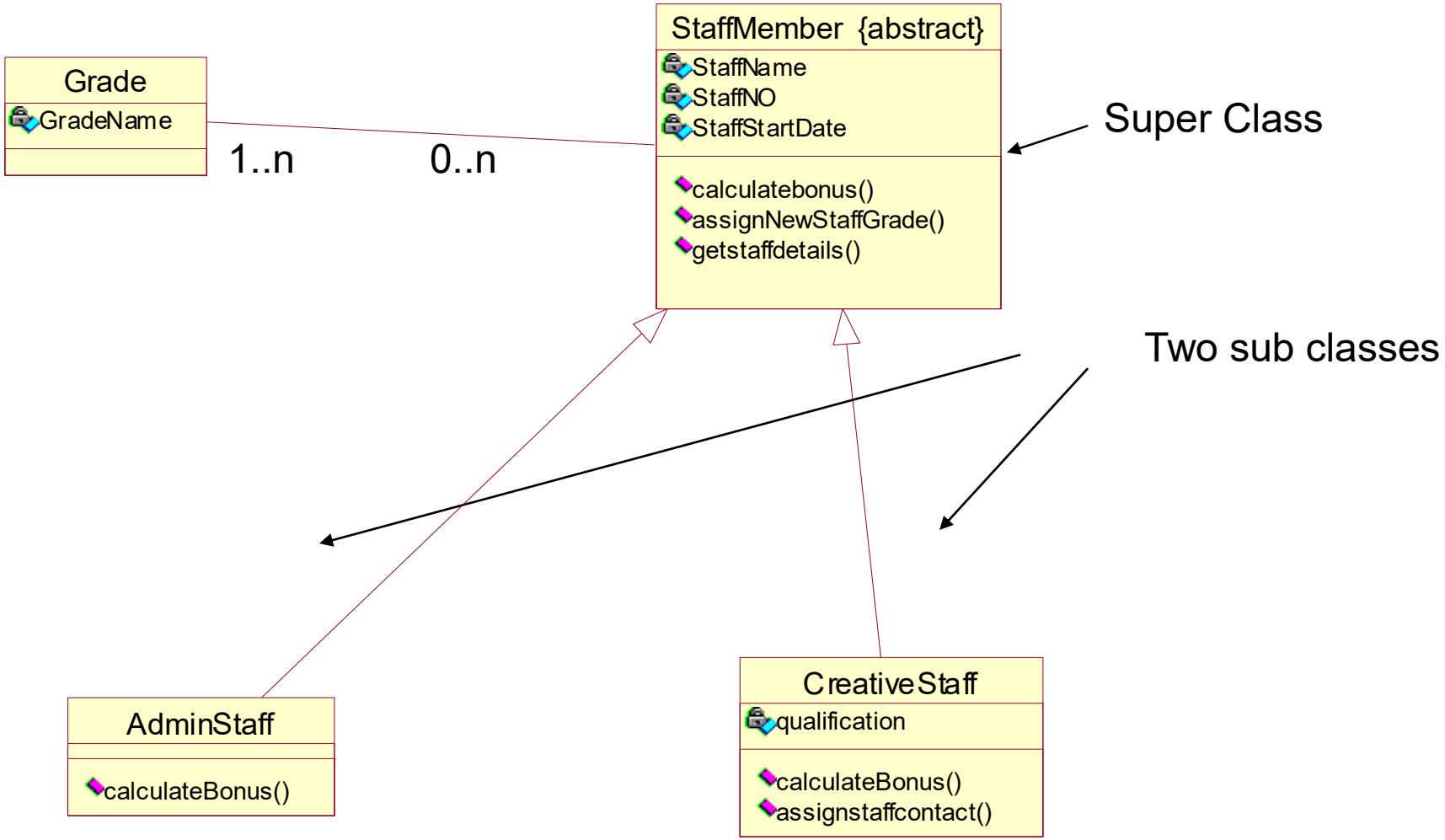
Any other diffs? Finance director says –

- C qualifications need to be recorded
- C can be assigned as contact for a client

- A are not assigned to specific campaigns

No other significant differences.

The following figure shows the concerned Class diagram for previous note :



This class diagram includes a generalization association between Staff Member, AdminStaff and CreativeStaff. Of these, Staff Member is the super-class, while AdminStaff and CreativeStaff are subclasses.

The generalization symbol states that all characteristics of the superclass Staff Member (its attributes, operations and associations) are automatically inherited by AdminStaff and CreativeStaff. There is no need to repeat superclass characteristics within a subclass definition.

From a subclass perspective, inherited features are actually features that belong to the subclasses but have been removed to a higher level of abstraction. Generalization saves the analyst from the need to show these characteristics explicitly for each subclass to which they apply. Common attributes, operations and associations thus may be shared by several subclasses, but need be shown only once, in the superclass. This aids the general consistency of the model, and can also considerably reduce its complexity.

Redefined operations :

Here we have an operation calculateBonus () , represented even in sub classes also means redefined .
Because, while both AdminStaff and CreativeStaff require an operation calculateBonus (), it works differently in each case. Since the precise logic for the calculation differs between these two groups of staff, the two operations will need to be treated separately later when each algorithm is designed, and also when programme code is written to implement the algorithm. This difference in logic justifies the separate appearance of a superficially similar operation in both subclasses.

This sort of fine distinction is not always recognized during analysis.
then, what is the reason the operation calculateBonus () is included in the superclass Staff Member?

The answer is that it is an attempt at 'future-proofing'. One of the consequences of identifying a superclass is that it may later acquire other subclasses, that are as yet unknown. In this case the analyst has recognized or assumed that objects belonging to *all* subclasses of Staff Member are likely to need an operation of some kind to calculate bonus. For this reason, at least a 'skeleton' of the operation is included in the superclass.

Abstract and concrete classes

Here the {abstract} annotation at the starting of Staff Member class name in Figure, means that a class is an abstraction of its members. However, Staff Member is abstract in the still more compelling sense that it has no instances. This is shown by the {abstract} *property* (an alternative notation for this is to write the class name in italics).

The {abstract} property can only be applied to a superclass in a generalization hierarchy. All other classes have at least one instance, and are said to be *concrete* or *instantiated.*

Applying the {abstract} property to Staff Member means that no staff exist at Agate who are 'general' members of staff, and not members of a particular sub-group. All staff members encountered so far are defined as either AdminStaff or CreativeStaff. Should we later discover another group of staff that is distinct in behaviour, data or associations, and if we need to model this new group, it should be represented in the diagram by a new subclass. The whole reason for the existence of a superclass is that it sits at a higher level of abstraction than its subclasses. This generality allows it to be adapted for use in other systems.

The usefulness of *generalization*

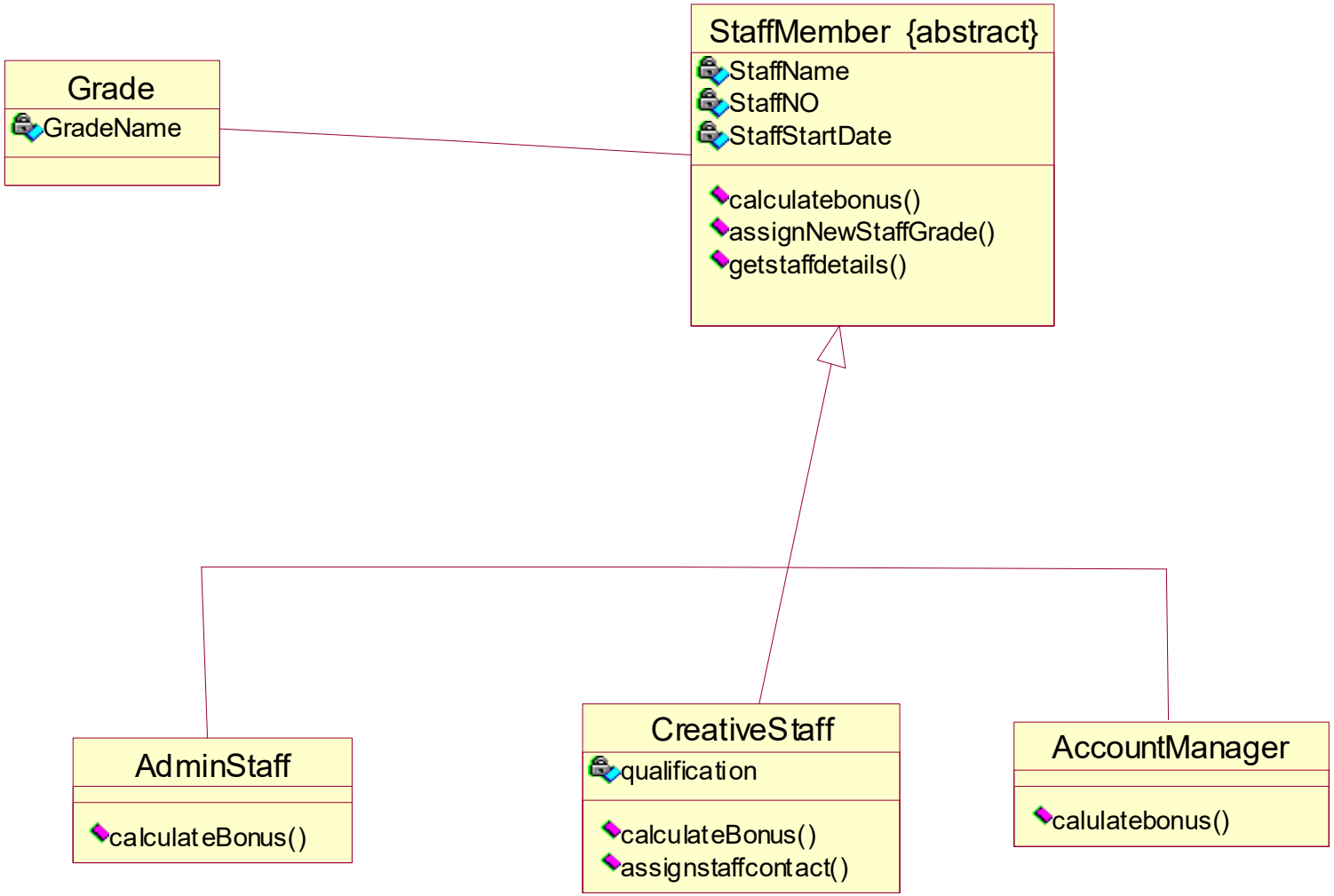
consider the contribution of generalization hierarchies to reuse, means This provides to add a new sub class with the already available generalization.

Imagine that the Agate system is completed and in regular use. Some time after installation, the Directors decide that they want to reorganize the company, and one of the results is that Account Managers are to be paid bonuses related to campaign profits.

Their bonus is to be calculated in a different way from both administrative and other creative staff, and is to include an element from campaigns that they supervise, and an element from the general profitability of the company. This is shown as below .

Here the three subclasses are organized into a tree structure with a single triangle joining this to the super class. The UML Specification calls this representation as shared target styles and in the previous figure UML calls that representation as the separate target style. Both are acceptable in UML.

New sub class easy to add with generalization



Adding a new subclass requires relatively little change to the existing class model, essentially just adding a new subclass AccountManager to the staff hierarchy. In practice, some judgement would be needed as to whether this is better modelled as a subclass of Staff Member, or of CreativeStaff, and this would be based on assumptions about difficulty of implementation and likely future benefits. In either case, the impact is minimal.

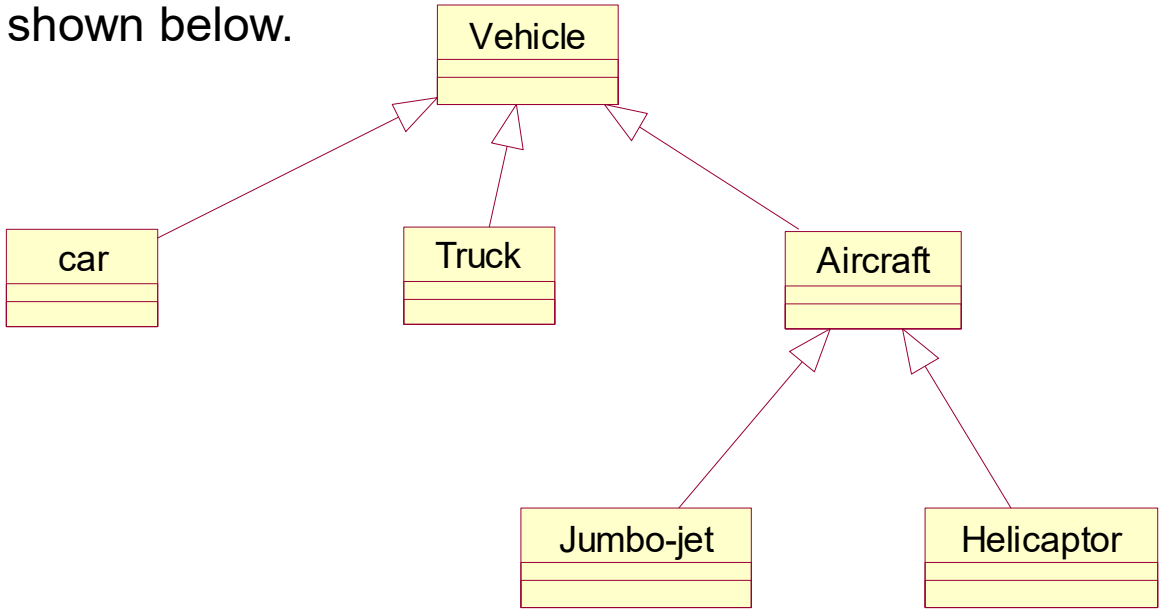
The reuse lies in the fact that the existing abstract class Staff Member has been used as a basis for AccountManager. The latter it can inherit all attributes and operations that are not part of its own .

Identifying generalization (a *top-down approach) :*

It is relatively easy to discover generalization where this exists between classes that have already been identified. The rule to be used here for the identification is straightforward , If an association can be described by the expression is *a kind of,* then it can usually be modelled as generalization.

Sometimes this is so obvious, for example, 'administrative staff are a kind of staff'. More often, it is not quite so obvious, but still straightforward.

For example, 'a helicopter is a type of aircraft and so is a jumbo-jet' and 'a truck is a type of vehicle and so is a car ' imply generalizations with similar structures, as shown below.



It is not uncommon to find multiple levels of generalization. This simply means that a superclass in one relationship may be a subclass in another. As in figure Aircraft is both a superclass of Helicopter , jumbo-jet and a subclass of Vehicle. In practice, more than about four or five levels of generalization in a class model is too many ,but this is primarily for design reasons.

Adding generalization (a *bottom-up approach)*

An alternative approach to adding generalization is to look for similarities among the classes in your model, and consider whether the model can be simplified by introducing super classes that abstract the similarities.

This needs to be done with some care. The purpose of doing this is quite explicitly to increase the level of abstraction of the model.

Multiple inheritance

It is quite possible, and often appropriate, for a class to be simultaneously the subclass of more than one superclass.

When not to use generalization

Generalization can be over used, for this consider the implicit judgements made in deciding to model the generalization structure developed in previous figures.

First, as a rule we only model a class as a superclass of another if we are confident that what we know about the former (that is its attributes, operations and associations) applies *completely* to the latter.

In this example, the analyst had to be reasonably sure that everything he knew about Staff Member applied also to both AdminStaff and CreativeStaff. This is part of the UML definition of generalization: a subclass is 'fully consistent with' its superclass. Even when this is true, if the differences between two potential subclasses are too great, the forced creation of a superclass can give rise to confusion rather than clarity.

Second, we should not anticipate subclasses that are not justified by currently known requirements.

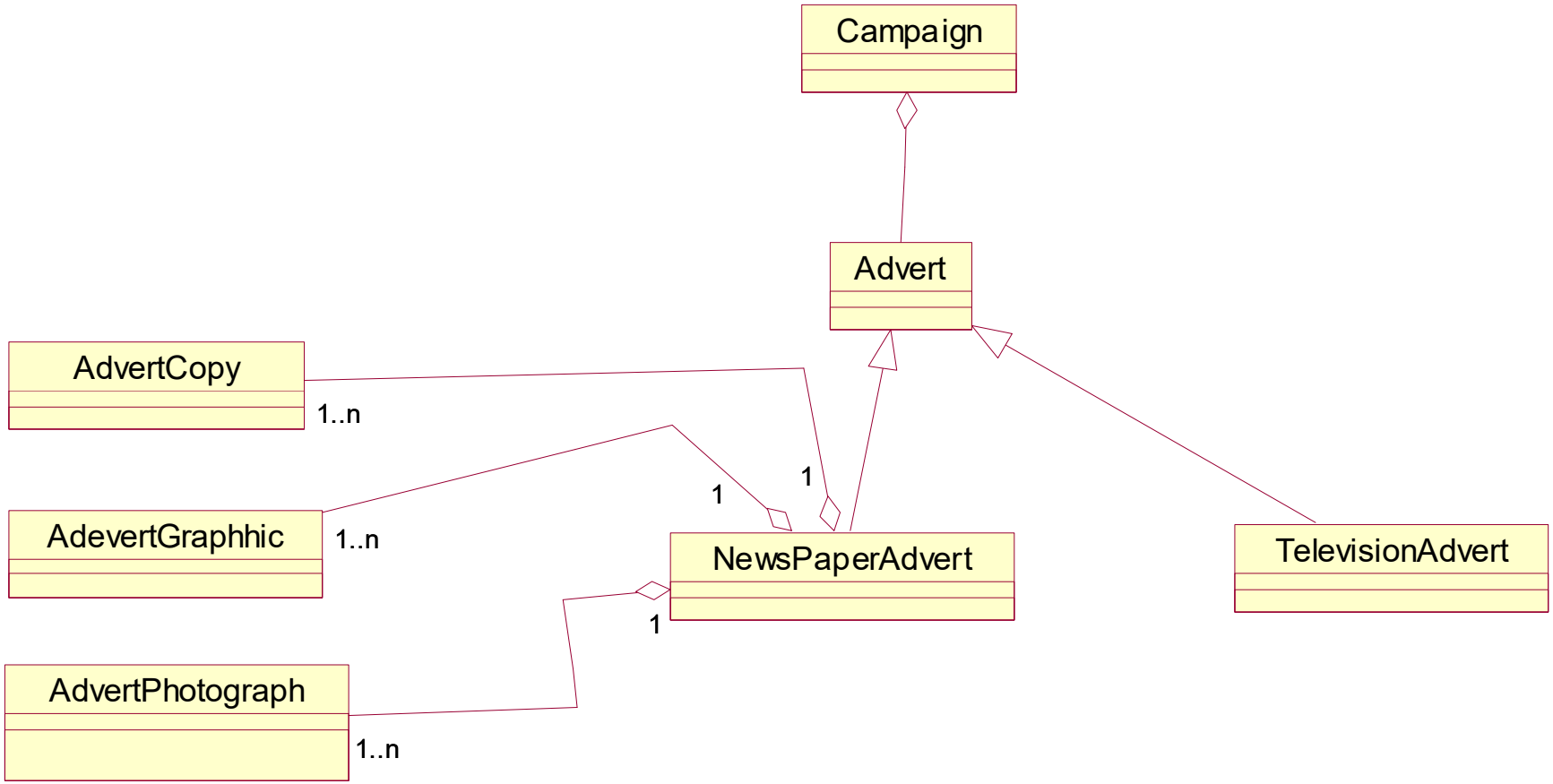
For example, at Agate AdminStaff and CreativeStaff are distinct classes based on differences in their attributes and operations. We also know about other kinds of staff in the organization, e.g. the directors. But we have no information that suggests a need to model directors as part of the system, and thus we should not create another subclass of Staff Member called DirectorStaff. Even if it were to turn out that we do need to model Directors, there is no reason yet to suppose they must be a distinct class. They might turn out to be members of an existing class. This, too, is part of the UML standard definition: a subclass must 'add additional information‘.

On the one hand, generalization is modelled in order to permit future subclassing in situations that the analyst inevitably cannot fully anticipate. Indeed the ability to take advantage of this is one of the main benefits of constructing a generalization hierarchy.

Yet, on the other hand, if generalization is overdone, it just adds needlessly to the complexity of the model for no return.

Combining generalization with composition or aggregation

Consider the following figure to represent the combination of both Generalization and aggregation.



In an Campaign , 'adverts can be one of several types‘, Like a newspaper advert *is a kind of* advert and televisionAdvert *is a kind of* advert . This suggests that advert could be a superclass, with newspaper advert, etc. as its subclasses.

This identification is made based on the following :

First, is *everything* that is true of Advert also true of NewspaperAdvert or not ? The answer appears to be yes. All actual adverts must be one of the specific types, with no such thing in reality as a 'general' advert. Advert is thus a sound generalization of common features shared by its specialized subclasses.

Second, does NewspaperAdvert include some details (attributes, operations or associations) that are *not* derived from Advert? Here also answer appears to be yes. A newspaper advert consists of a particular set of parts. The precise composition of each type of advert is different, and so this structure of associations could not be defined at the superclass level .

Finally, there is no reason to suppose that a newspaper advert has any other potential ancestor besides advert, so we do not need to consider this rule at higher levels of recursion.

Next, we can see possible composition in the association between Campaign and Advert, and in turn between Advert and its associated parts. A campaign includes one or more adverts. A newspaper advert includes written copy, graphics and photographs.

In order to identify the type of a composition consider the following :

First, can an advert belong to more than one campaign? This is not stated in the case study, but it seems unlikely that an advert can simultaneously be part of more than one campaign.

secondly, can copy, graphics or photographs belong to more than one newspaper advert? This seems unlikely, but should really be clarified.

Finally, has each of these components a coincident lifetime with the advert? Probably some do, and some do not.

It is hard to imagine that advertising copy would be used again on another advert, but photographs and graphics seem less certain. This is another point to be clarified, but in the meantime composition does not seem justified in any of these cases, and aggregation has therefore been used.

Analysis packages and dependencies :

The analyst is responsible to create a model that will be robust in the face of changing requirements based on his decisions and skill. To some extent this depends on defining analysis packages that are relatively independent of each other while still internally highly cohesive .

The UML package is a tool for managing the complexity of a model, and they are also a useful way of identifying subsystems that can stand alone as components. Packages are the means by which a developer can 'factor out' classes or structures that have potential use in a wider context than this project alone. But when a model is partitioned into packages it is very important to keep track of the dependencies between different classes and packages.

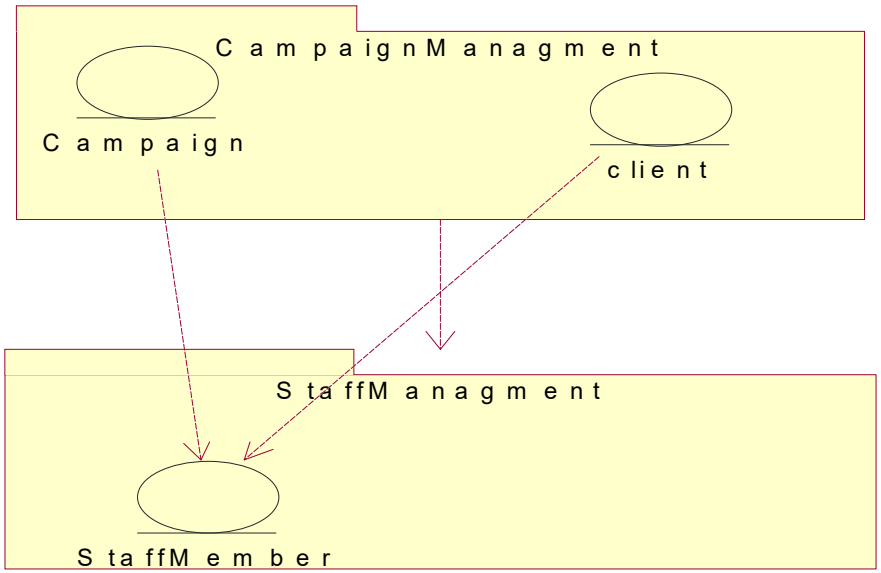
In the Agate case study we have mainly two related but distinct application areas:
1.campaign management
2.staff management.

If this model proves to be only one part of a larger domain, it will probably make sense to model these as two separate analysis packages: Campaign Management and Staff Management. If this is done, it is quite likely that some entity objects will prove to be common to both packages.

Alternative dependencies among packages and among objects within packages.

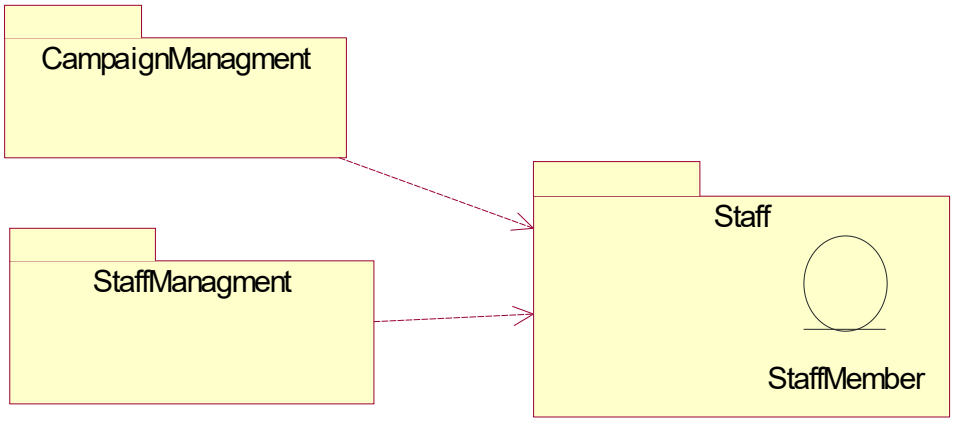
1. Based on preliminary analysis , the StaffMernber entity class plays a role in both packages, So this class can be placed within the Staff Management package. And here, we need to model a dependency from Campaign Management to Staff Management, since the Client and Campaign classes need an association with StaffMernber

as shown below.

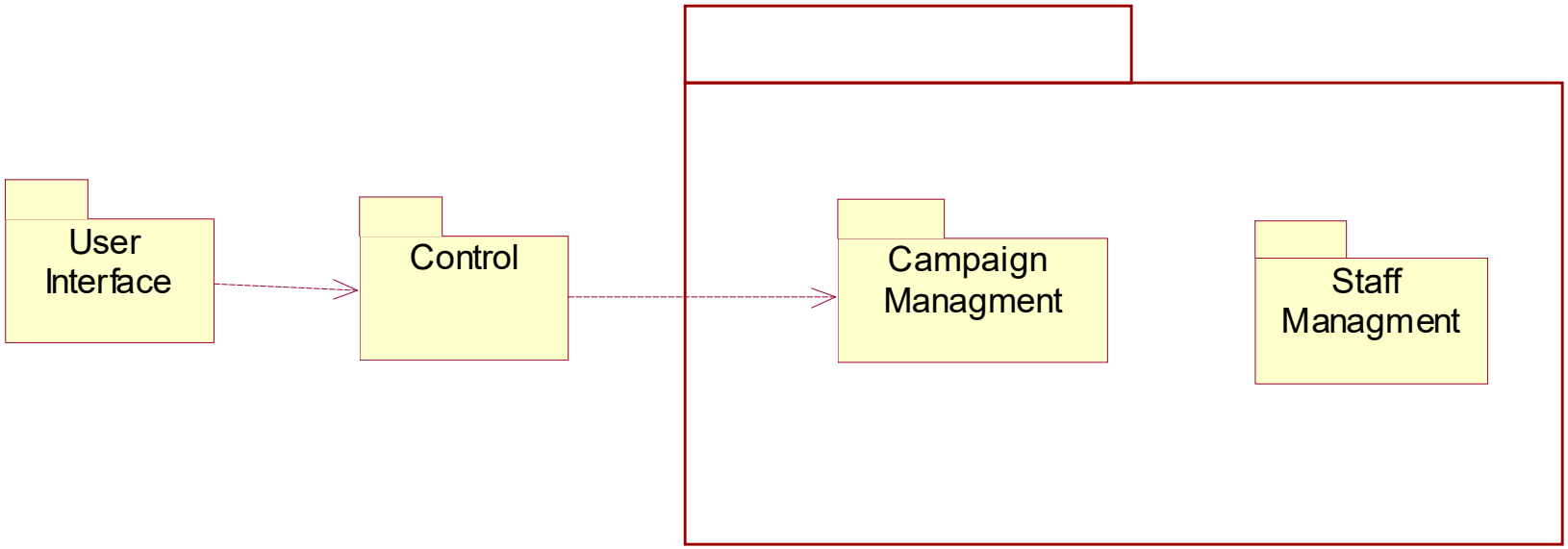


2. We could remove StaffMernber to a separate package. This would be justified if it appears to have more widespread use.

   For example, there may also be wages, personnel, welfare and pension applications that need this class. In this case, we need to model dependencies from all the corresponding packages to the package that contains the StaffMernber class as shown below.



3. The following figure shows by placing all boundary objects into a User Interface package and all control objects into a Control package. Objects in these specialized packages will certainly have dependencies on objects in other packages .



SOFTWARE DEVELOPMENT PATTERNS

What is a pattern?

Generally, a pattern refers to a kind of design that is used to reproduce images in a repetitive manner in a daily life. In software development, the term is related to this idea, but has a much more specific meaning.

Pattern : Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of a solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

Christopher Alexander, who first used the above term pattern to describe solutions to recurring problems in architecture. Alexander identified many related patterns for the development of effective and harmonious architectural forms in buildings. Alexander's patterns address many archi-tectural issues-for example the best place to site a door in a room, or how to organize and structure a waiting area in a building so that waiting can become a positive experi-ence. Alexander argued that his patterns became a design language within which solutions to recurring architectural problems could be developed and described.

A pattern provides a solution that may be applied in different ways depending upon the specific problem to which it is being applied.

Another One definition of a pattern that is appropriate to software systems development is this:

A pattern is the abstraction from a concrete form which keeps recurring in specific non arbitrary contexts.

Each pattern is a three-part rule, which expresses a relation between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain software configuration which allows these forces to resolve themselves.

This definition focuses on three elements-a *context* that can be understood as a set of circumstances or preconditions, *forces* that are issues that have to be addressed and a software configuration that addresses and resolves the forces.

In this definition the term 'software configuration' might suggest that patterns are limited to software design and construction. **In** fact, patterns are applied much more widely in systems development. The Analysis class stereotypes are patterns widely applied during requirements analysis and systems design.

Coad make the distinction between a *strategy* which they describe as a plan of action intended to achieve some defined purpose and a *pattern* which they describe as a template that embodies an example worth emulating.

This view of a pattern is slightly different from the views described earlier as it does not emphasize contextual aspects to the same extent. An example of a Coad strategy is 'Organize and Prioritize Features' and relates to the need to prioritize requirements .

A simple example of an analysis pattern from is the Transaction-Transaction Line Item pattern as shown below.

The following figure shows the pattern as it might be applied to a sales order processing system. Here the Transaction suggests a SalesOrder class and the Transaction Line Item suggests a SalesOrderLine class.

Very similar structures are used in a wide variety of other circumstances also. A trainee software developer has to learn this structure, or to reinvent it. The latter is much less efficient. The act of describing it as a pattern highlights it as a useful piece of development expertise and makes it readily available for the a trainee software developer .

Coplien identifies the critical aspects of a pattern as follows.

- It solves a problem.
- It is a proven concept.
- The solution is not obvious.
- It describes a relationship.
- The pattern has a significant human component.

In the same way that a pattern captures and documents proven good practice, *anti-patterns* capture practice that is demonstrably bad. It is sensible to do this. We should ensure not only that a software system embodies good practice but also that it avoids known pitfalls. Antipatterns are a way of documenting attempted solutions to recurring problems that proved unsuccessful.

For example, Mushroom Management is an example of an anti pattern in the domain of software development organizations. It describes a situation where there is an explicit policy to isolate systems developers from users in an attempt to limit requirements go with flow or float.

In such an organization, requirements are passed through an intermediary such as the project manager or a requirements analyst. The negative consequence of this pattern of development organization is that inevitable inadequacies in the analysis documentation are not resolved.

Furthermore design decisions are made without user involvement, and the delivered system may not address users' requirements. The reworked solution is to use a form of spiral process development model . Other reworked solutions include the involvement of domain experts in the development team, as recommended by the Dynamics Systems Development Method (DSDM).

Patterns have been applied to many different aspects of software development.

1. Software patterns are applied in order to describe aspects of interface design in Smalltalk environments.

2. A set of patterns specifically for use in C++ programming (patterns that are related to constructs in a specific programming language are now known as *idioms).*

3. Analysis and Design Patterns have been applied to software development approaches other than object-orientated ones.

A series of analysis patterns are used for data modelling and 'Design Patterns are Elements of Reusable Object-Oriented Software'.

Architectural patterns – Responsibilities

1. Addresses some of the issues concerning the structural organization of software systems.

2. *Architectural patterns also* describes the structure and relationship of major components of a software system.

3. These patterns identifies subsystems, their responsib-ilities and their interrelationships.

Design patterns – Responsibilities

1. These patterns identify the interrelationships among a group of software components describing their responsibilites, collaborations and structural relationships.
2. Idioms describe how to implement particular aspects of a software system in a given programming language.

Analysis patterns : Resposibilities

These are defined as describing groups of concepts that represent common constructions in domain modelling. These patterns may be applic-able in one domain or in many domains.

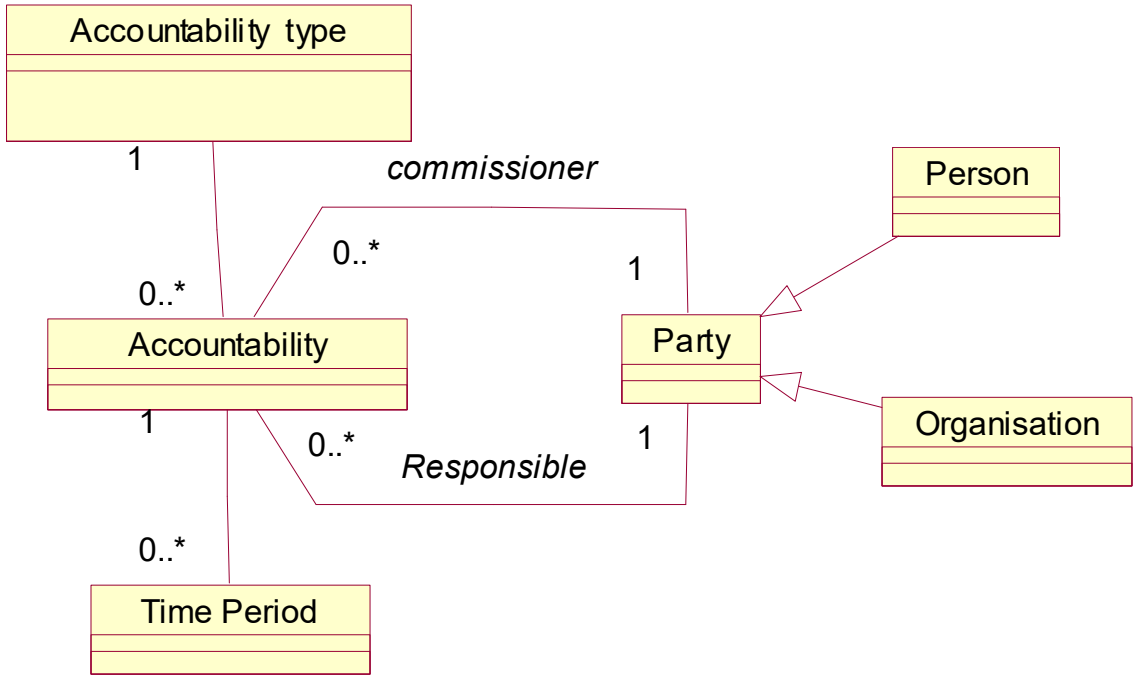
The use of analysis patterns is an advanced approach that is principally of use to experienced analysts,. They are closely related to design patterns also.

An analysis pattern is essentially a structure of classes and associations that is found to occur over and over again in many different modelling situations.

Each pattern can be used to communicate a general understanding about how to model a particular set of requirements, and therefore the model need not be invented from scratch every time a similar situation occurs. Since a pattern may consist of whole structures of classes, the abstraction takes places at a higher level than is normally possible using generalization alone.

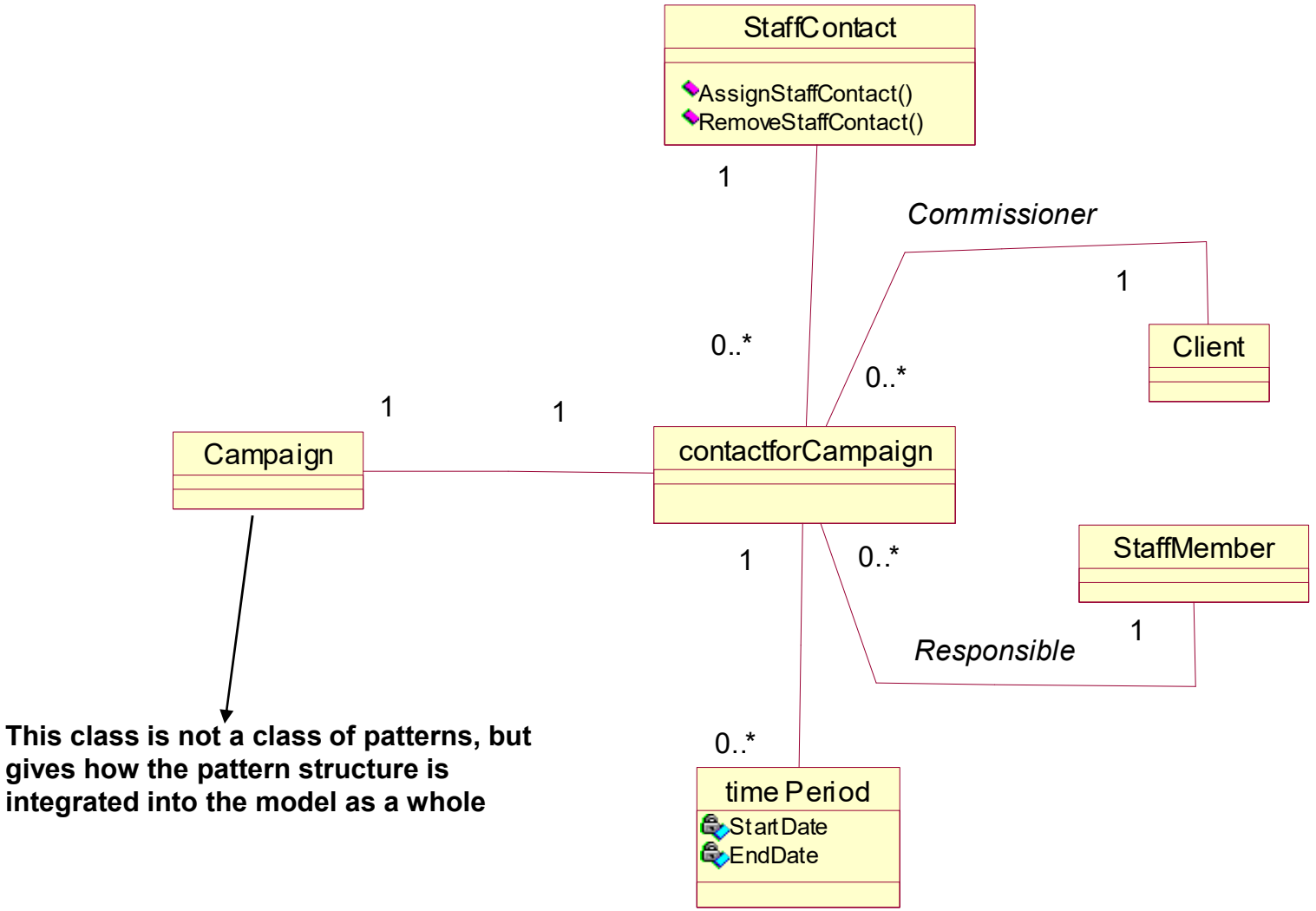
A particular pattern may belong to more than one category. For example, the MVC architecture can be viewed as an architectural pattern when it is applied to subsystems, as a design pattern when it is concerned with smaller com-ponents or individual classes

The following figure shows Accountability pattern as an illustration of an analysis pattern .Here we will consider only the class structure, An accountability structure may be of many kinds, such as management or contract supervision. The generalization of Person and Organization as Party allows the pattern to represent relationships between individuals, organizations, or a mixture of the two.



The Accountability analysis pattern (adapted from Fowler, 1997).

The following figure shows Accountability analysis pattern, In Agate “ Staff contact “ relationship.



This pattern could apply to several different relation-ships: that between a manager and a member of staff they supervise, that between a client and a client contact, or that between a client and a campaign manager. Since the details of the relationship itself have been abstracted out as Accountability Type, this one class structure is sufficiently general to be adapted to any of these relationships, given an appropriate set of attributes, operations and associations to other classes specific to the application model. The generalization of Person and Organization as Party similarly allows the pattern to represent relationships between individuals, organizations, or a mixture of the two.

9.Object Interaction

Communication and collaboration between the objects is fundamental concept in OO. This involves communication to request information , to share information and to request to help from each other. Among the set of autonomous objects in an interaction , each is responsible for a small part of systems overall behavior. These objects produce the required behavior through collaboration , by exchanging messages that request information, that give information or that ask another object to perform some task.

Object Interaction and Collaboration :

When an object sends a message to another object, an operation is invoked in the receiving object.

For example, in the Agate case study there is a requirement to be able to determine the current cost of the advertisements for an advertising campaign. This responsibility is assigned to the Campaign class. For a particular campaign this might be achieved if the Campaign object sends a message to each of its Advert objects asking them for their current cost.

In a programming language, sending the message getCost () to an Advert object, might use the following syntax.

```
advertCost = anAdvert.getCost()
```

The cost of each advert returned by the operation getCost () is totalled up in the attribute actualCost in the sending object, Campaign, so in order to calculate the sum of the costs for all adverts in a campaign the above statement must be executed repeatedly. For this purpose we are using message passing mechanism for object interaction. This Message passing can be represented on an object diagram, as shown below.

It can be difficult to determine what messages should be sent by each object. In this case, the getCost () operation should be located in the Advert class. This operation requires data that is stored in the advertCost attribute, and this has been placed in Advert. We can also see that an operation that calculates the cost of a Campaign must be able to find out the cost of each Advert involved.

But this is a simple collaboration and *the allocation* of *these operations is* largely dictated by the presence of particular attributes in the classes. More complex requirements may *involve* the performance of complex tasks, such that an object receiving one message must itself send messages that initiate further collaboration with other objects.

the objective of OOAD to distribute system functionality appropriately among its classes. This does not mean that all classes have exactly equal levels of responsibility but rather that each class should have appropriate responsibilities. Where responsibilities are evenly distributed, each class tends not to be complex and easy to develop , test and maintain.

An appropriate distribution of responsibility among classes has the important side effect of producing a system that is more resilient to changes in its requirements. When the users' requirements for a system change it is reasonable to expect that the application will need some modification, but ideally the change in the application should be of no greater magnitude than the change in the requirements.

An application that is resilient in this sense costs less to maintain and to extend than one that is not. The following figure illustrates this resilient concept.

Resilience of design

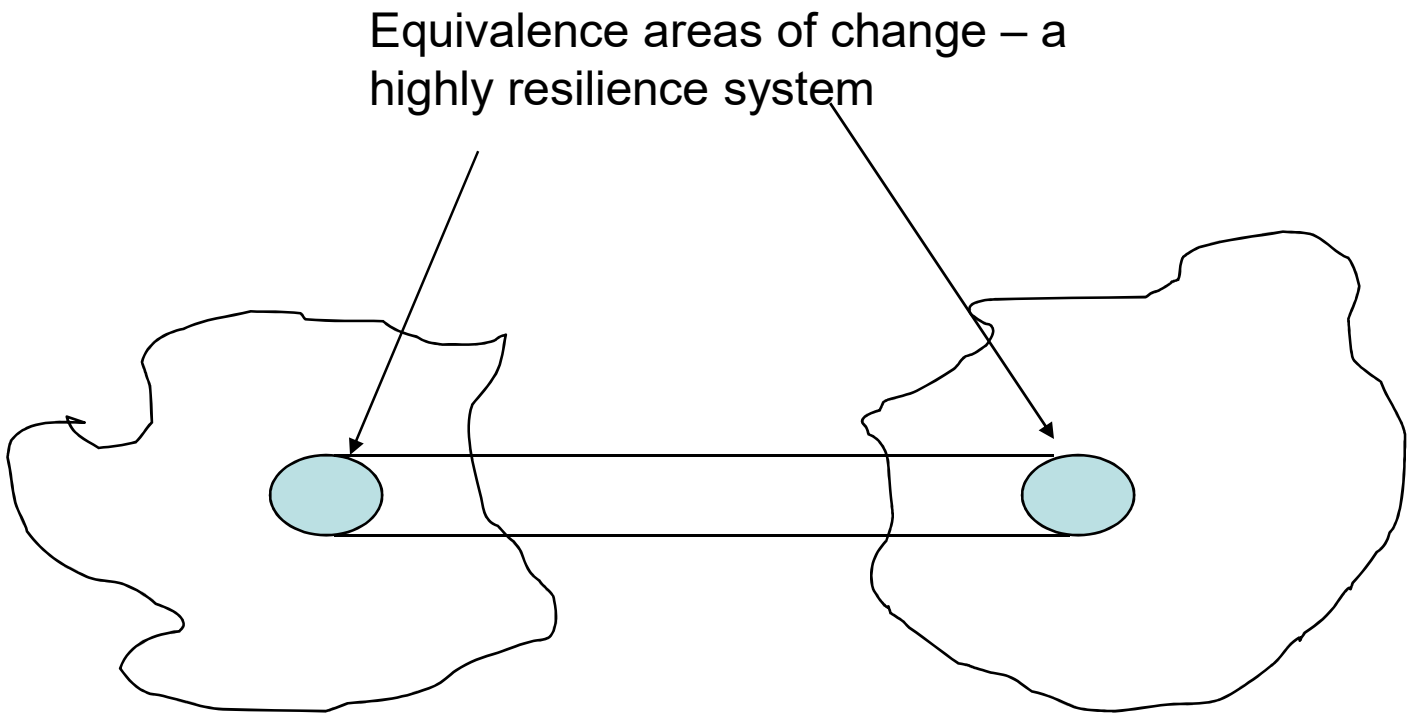
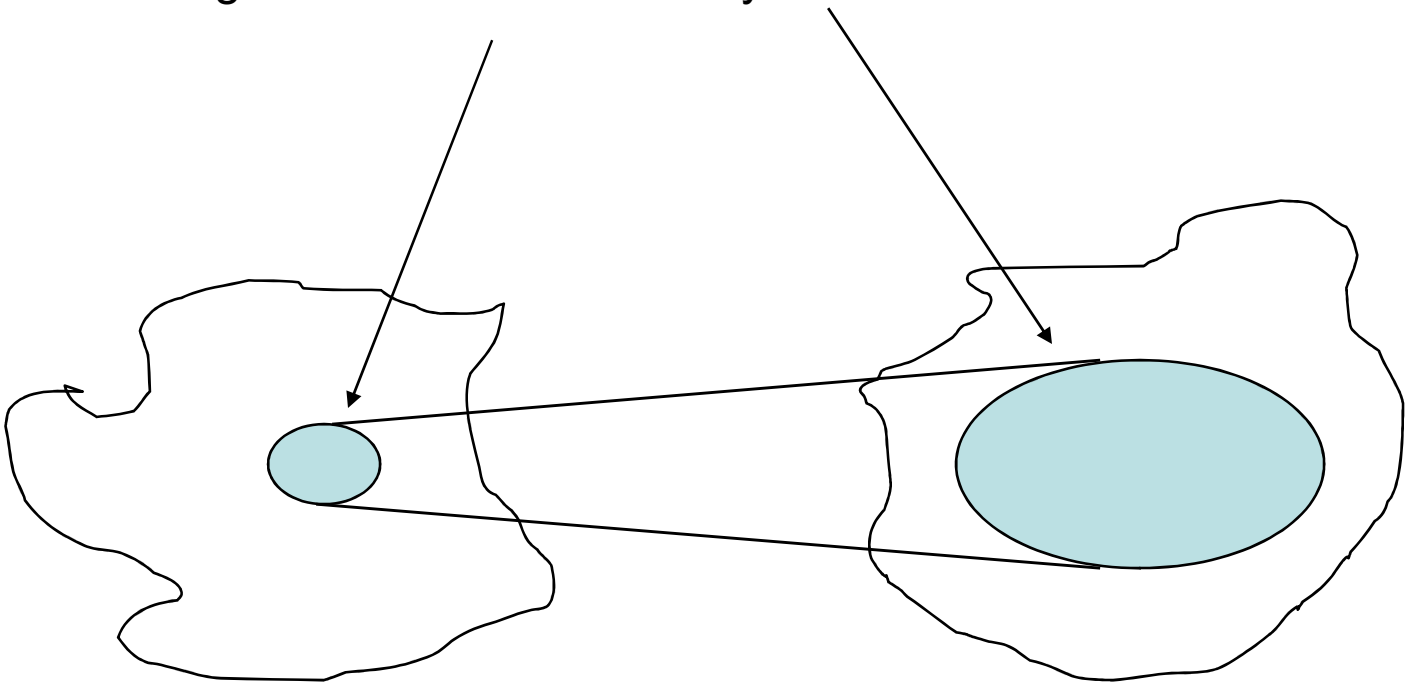


Figure (a)

A small change in requirements causes a much greater change in s/w not a resilient system



Real world requirements

Application that caters for these requirements

Figure (b)

The purpose of a object interaction is to determine the most appropriate scheme of messaging between objects in order to support a particular user requirement.

For this interaction , user requirements are first documented by use cases , each use case can be seen as a dialogue between an actor and the system that results in objects performing tasks so that the system can respond in the way that is required by the actor.

For this reason many interaction diagrams explicitly include objects to represent the user interface (boundary objects) and to manage the object communication (control objects). When such objects are not shown explicitly it can be assumed in most cases that they will need to be identified at a later stage. The identification and specification of boundary objects is in part an analysis activity and in part of a design activity also. During analysis our concern is to identify the nature of a dialogue in terms of the user's need for information and his or her access to the system's functionality.

UML defines object interaction within the context of a collaboration and defines a collaboration as follows.

‘ The structure of Instances playing roles in a behavior and their relationships is called a *Collaboration .'*

The behaviour mentioned above can be that of an operation or a use case . A particular object instance may play different roles in different contexts or collaborations and may play more than one role in a given collaboration.

Interaction among the objects will be provided an interaction diagrams called

1. Interaction Sequence diagrams
2. Interaction Collaboration Diagrams

Interaction Sequence Diagrams :

An interaction sequence diagram (or simply a *sequence diagram)* is one of the two kinds of UML interaction diagram. The other one is the collaboration diagram,.

An Interaction is defined in the context of Collaboration. It specifies the communication patterns between the roles in the Collaboration. More precisely, it contains a set of partially ordered *Messages,* each one specifying communication; what Signal to be sent or what Operation to be invoked, as well as the roles to be played, by the sender and receiver respectively.

A sequence diagram shows an interaction between objects arranged in a time sequence. Sequence diagrams can be drawn at different levels of detail and to meet different purposes at several stages in the development life cycle. The commonest application of a sequence diagram is to represent the detailed object interaction that occurs for one use case or for one operation. When a sequence diagram is used to model the dynamic behaviour of a use case it can be seen as a detailed specification of the use case.

Sequence diagrams drawn during analysis differ from those drawn during design in two major respects.

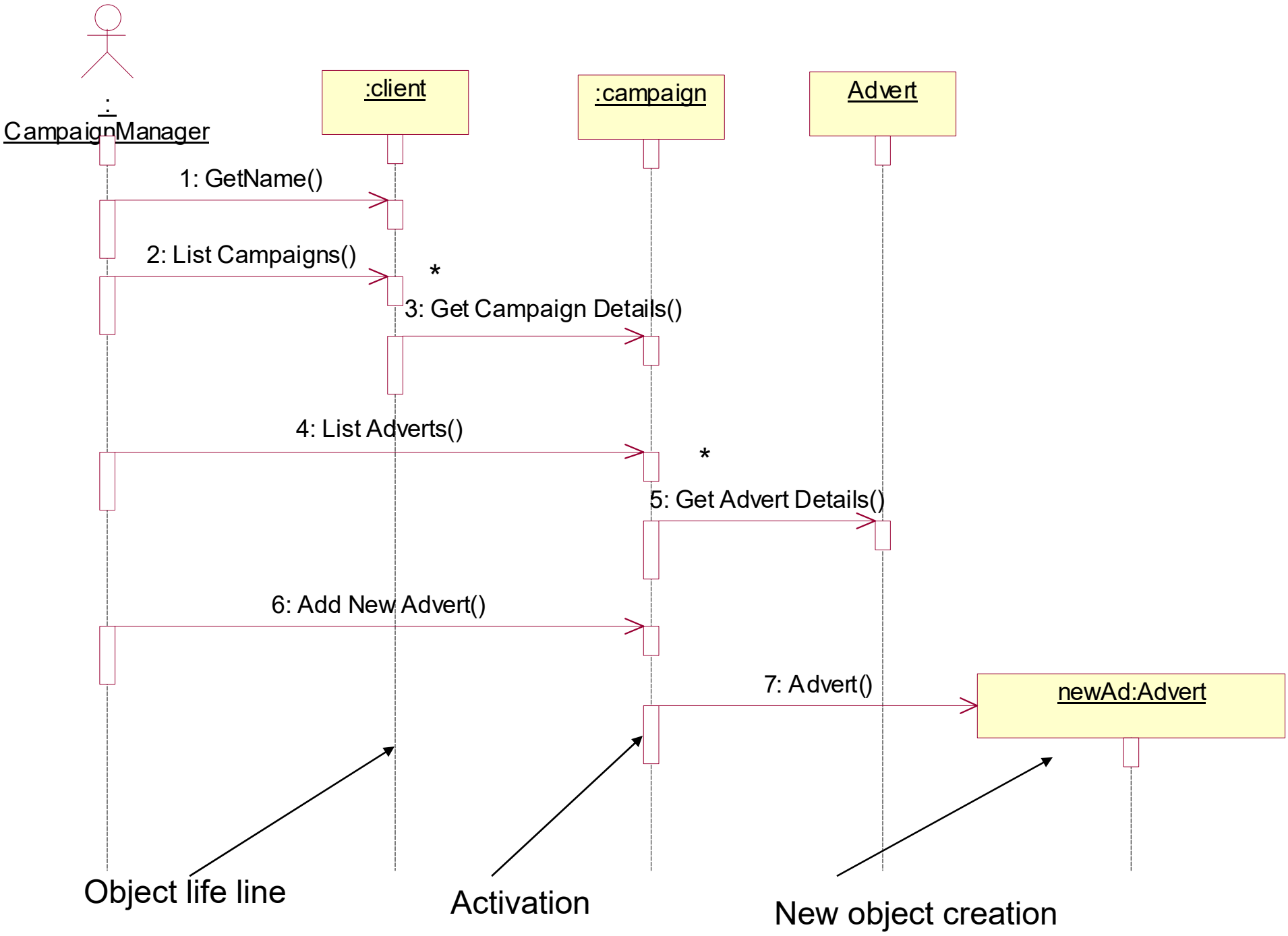
1. Analysis sequence diagrams normally do not include design objects .
2. They usually do not specify message signatures in any detail.

Basic concepts and notation :

In Sequence diagram the vertical dimension represents time and all objects involved in the interaction are spread horizontally across the diagram.

Time normally proceeds down the page. However, a sequence diagram may be drawn with a horizontal time axis if required, and in this case, time proceeds from left to right across the page. Each object is represented by a vertical dashed line, called a *lifeline,* with an object symbol at the top. A message is shown by a solid horizontal arrow from one lifeline to another and is labelled with the message name. Each message name may optionally be preceded by a sequence number that represents the sequence in which the messages are sent, but this is not usually necessary on a sequence diagram since the message sequence is already conveyed by their relative positions along time axis.

The following figure shows a sequence diagram for the use case Add a new advert to a campaign.


:
CampaignManager
:client
:campaign
Advert
1: GetName()
2: List Campaigns()
*
3: Get Campaign Details()
4: List Adverts()
*
5: Get Advert Details()
6: Add New Advert()
7: Advert()
newAd:Advert
Object life line
Activation
New object creation

UML uses the general term *stimulus* to describe an interaction between two objects that conveys information with an expectation of some action. Formally then, a message specifies the sender and receiver objects and the action of a stimulus. A message may correspond to calling an operation or raising a *signal.* In UML a signal is an asynchronous communication that may have parameters. An event is the specification of an occurrence of significance and may for instance be the receipt of a message or a signal by an object.

When a message is sent to an object, it invokes an operation of that object. Once a message is received, the operation that has been invoked begins to execute. The period of time during which an operation executes is known as an *activation,* and is shown on the sequence diagram by a rectangular block laid along the lifeline. The activation period of an operation includes any delay while the operation waits for a response from another operation that it has itself invoked as part of its execution. The message name is usually the same as the particular operation that is being invoked.

The above Figure shows a sequence diagram for to **add new advet to campaign** without boundary or control objects.

The getName () message is first message received by the Client and is intended to correspond to the Campaign Manager requesting the name of the selected Client. The Client object then receives a listCampaigns () message and a second period of operation activation begins. This is shown by the tall thin rectangle that begins at the message arrowhead. The Client object now sends a message getCampaign Details () to each Campaign object in turn in order to build up a list of campaigns. This repeated action is called iteration and is indicated by an asterisk (*) before the message. The conditions for continuing or ceasing an iteration may be shown beside the message name.

This example of a continuation condition is written as follows.

[For all client's campaigns] *getCampaignDetails()

The Campaign Manager next sends a message to a particular Campaign object asking it to list its advertisements. The Campaign object delegates responsibility for getting the advertisement title to each Advert object although the Campaign object retains responsibility for the list as a whole.

When an advertisement is added to a campaign an Advert object is created. This is shown by the Advert () message arrow drawn with its arrowhead pointing directly to the object symbol at the top of the lifeline. Where an object already exists prior to the interaction the first message to that object points to the lifeline below the object symbol. For example, this is the case for the Campaign object, which must exist before it can receive an addNewAdvert () message.

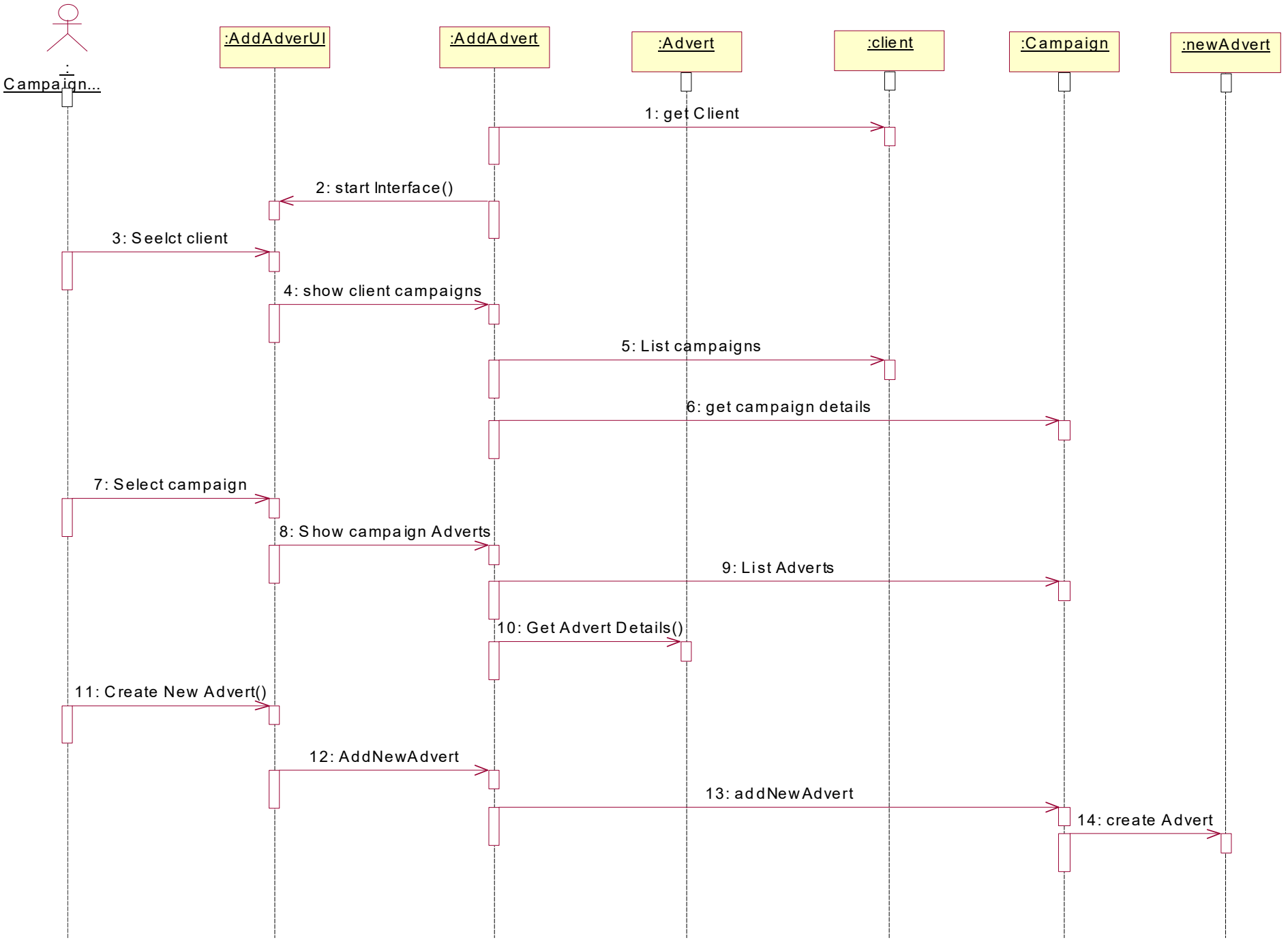
Most use cases imply at least one boundary object that manages the dialogue between the actor and the system and one control object which controls the overall activity flow, these specifications will be provided in design phase diagrams.

According to USDP boundary and control classes are identified in Analysis phase, but generally these to be identified during design stage only.

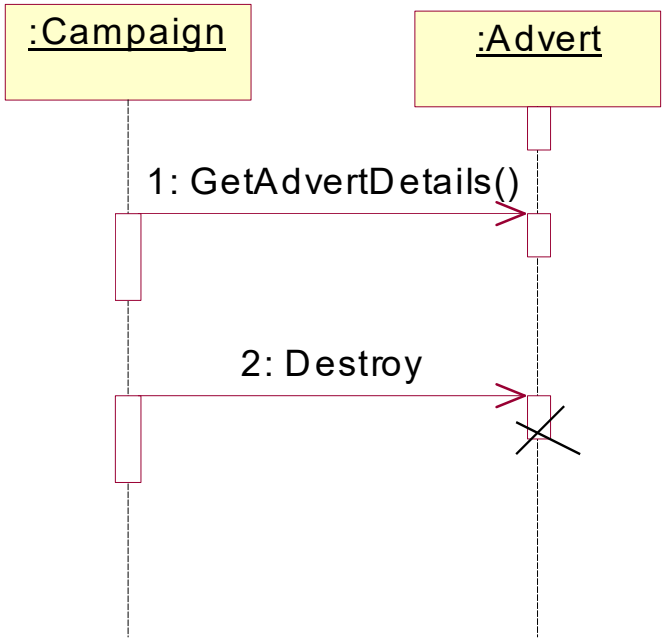
The following figure shows an alternative sequence diagram for the use case Add a New Advert to a Campaign with boundary and control objects. Essentially this is in the style of the Unified Software Development Process.

The boundary object, representing the user interface, is : AddAdvertUI. The control object is : AddAdvert and this manages the overall object communication.

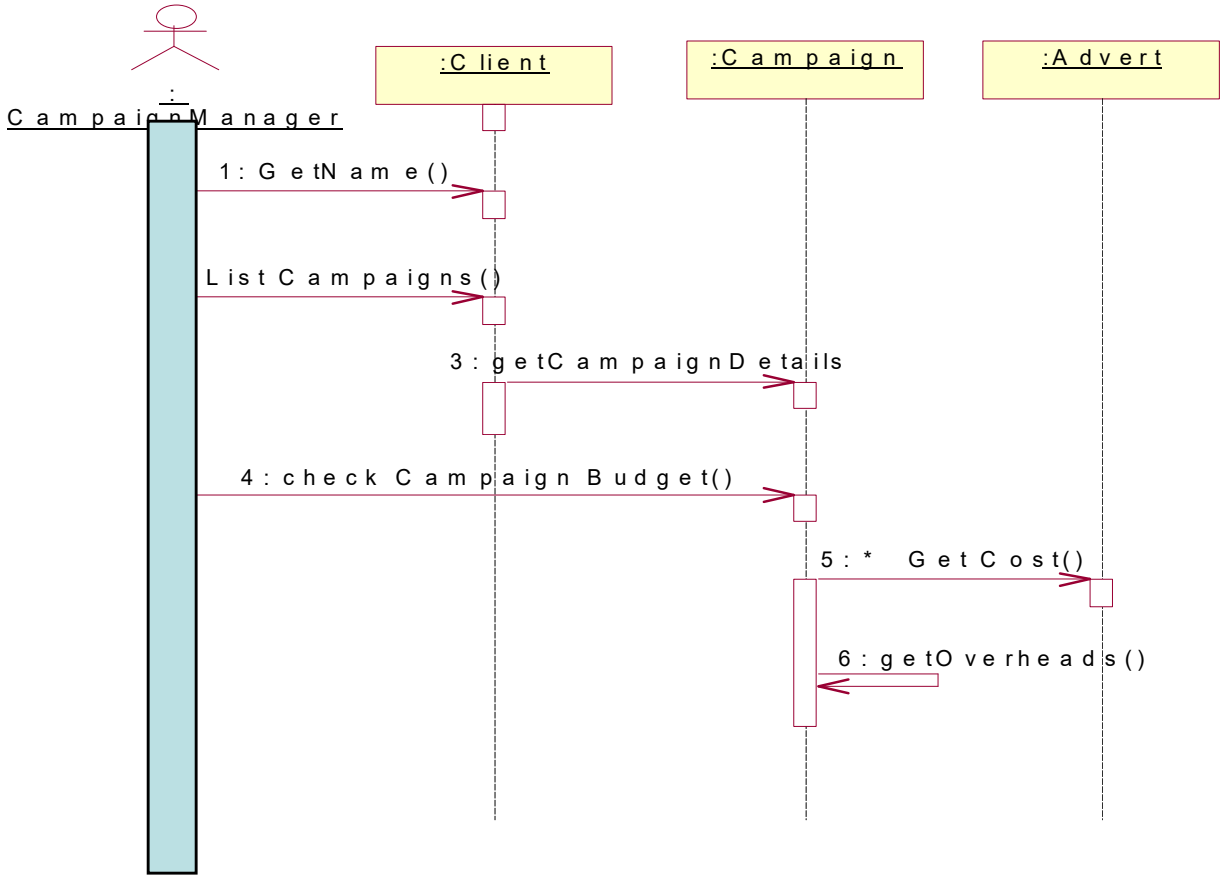
An object can send a message to itself. This is known as a *reflexive message* and is shown by a message arrow that starts and finishes at the same object lifeline.


:
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

Objects may be created or destroyed at different stages during an interaction. On a sequence diagram the destruction of an object is indicated by a large X on the lifeline at the point in the interaction when the object is destroyed. An object may either be destroyed when it receives a message or it may self-destructive at the end of an activation if this is required by the particular operation that is being executed. This is shown as follows.



The campaign budget may be checked to ensure that it has not been exceeded. The current campaign cost is determined by the total cost of all the adverts and the campaign overheads. The corresponding sequence diagram is shown below, which includes a reflexive message getOverheads () sent from a Campaign object to itself.



In this case the reflexive message invokes a different operation from the operation that sent the message and a new activation symbol is stacked on the original activation.

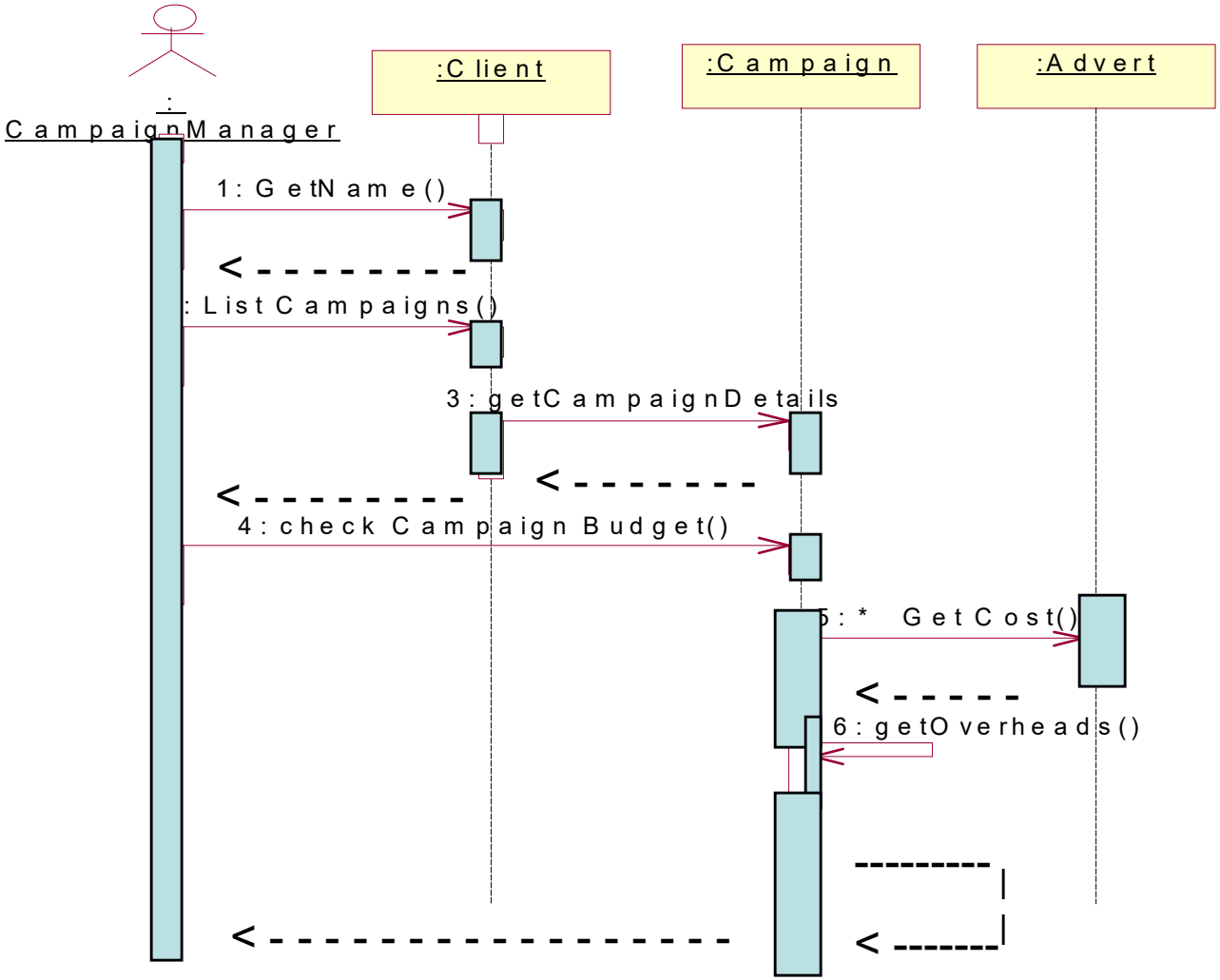
Focus Of control :

The *focus of control* indicates times during an activation when processing is taking place within that object. Parts of an activation that are not within the focus of control represent periods when, for example, an operation is waiting for a return from another object. The focus of control may be shown by shading those parts of the activation rectangle that correspond to active processing by an operation.

In the next figure , the Check campaign budget use case is redrawn with foci of control shaded. The focus of control for the checkCampaignBudget () operation is initially with the Campaign object, but is then transferred to the Advert object and the activation rectangle in the Campaign object is now unshaded while the Advert object has the focus of control. The checkCampaignBudget () activation is also unshaded while the getOverheads ( ) operation is activated by the reflexive message getOverheads ( ) .

A *return* is a return of control to the object that originated the message that began the activation. This is not a new message, but is only the conclusion of the invocation of an operation. Returns are shown with a dashed arrow, but it is optional to show them at all since it can be assumed that control is returned to the originating object at the end of the activation in a destination object.

The following figure some variations in the sequence diagram :



A *return-value* is the value that an operation returns to the object that invoked it. These are rarely shown on an analysis sequence diagram, in Figure the operation invoked by the message getName ( ) would have return-value of clientName and no parameters. In order to show the return-value the message could be shown as

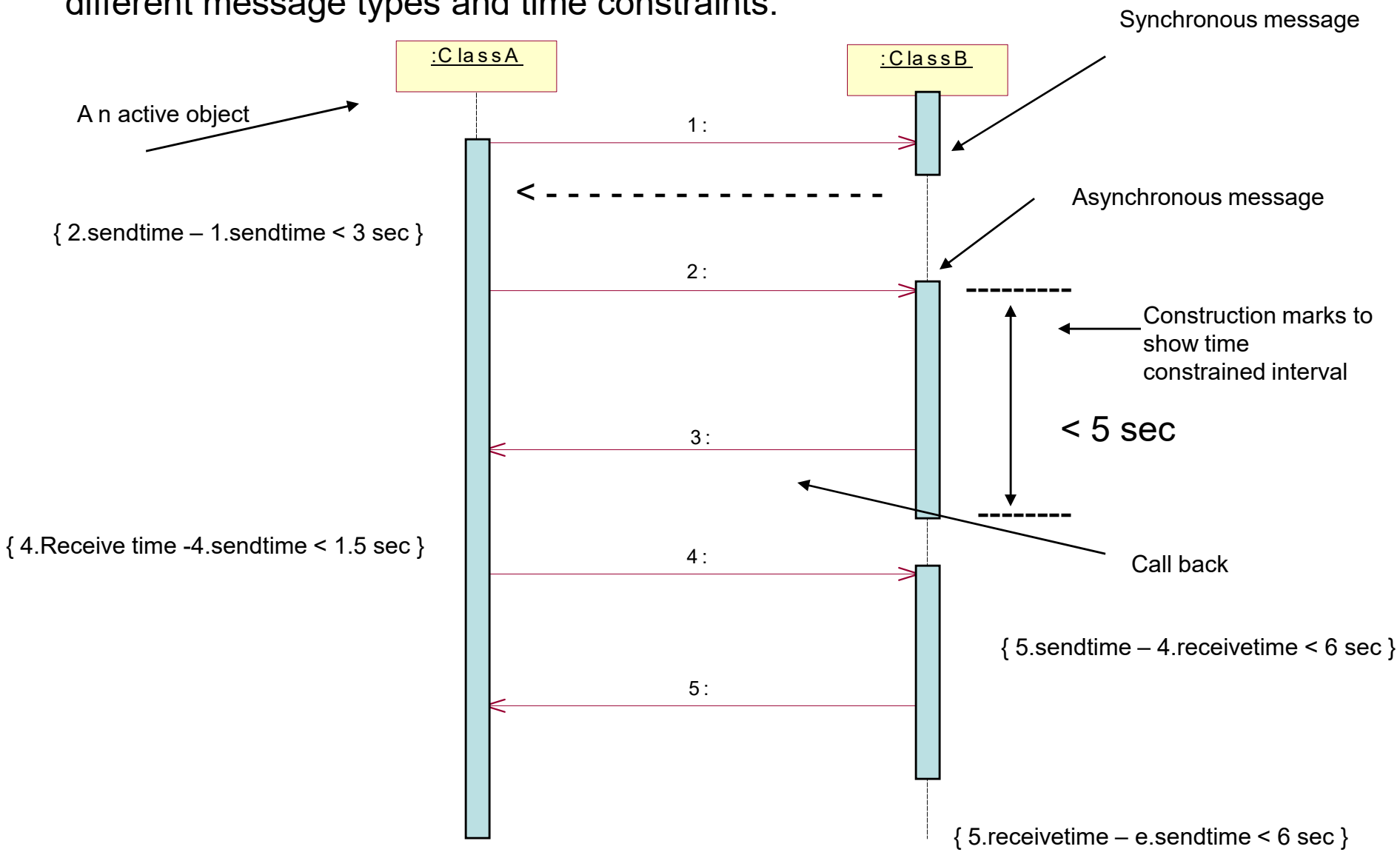
clientName := getName()

where clientName is a variable of type Name.

A *synchronous message* or *procedural call* is shown with a full arrowhead , and is one that causes the invoking operation to suspend execution until the focus of control has been returned to it. This is essentially a nested flow of control where the complete nested sequence of operations is completed before the calling operation resumes execution. This may be because the invoking operation requires data to be returned from the destination object before it can proceed. The se are shown In Figure with procedural calls and explicit returns. Procedural calls are appropriate for the interaction since each operation that invokes another does so in order to obtain data and cannot continue until that data is supplied.

Synchronous and asynchronous messages with sequence diagram

The following figure shows a sequence diagram for ClassA and ClassB with different message types and time constraints.



An *asynchronous message,* drawn with an open arrowhead as in above Figure , does not cause the invoking operation to halt execution while it awaits a return. When an asynchronous message is sent operations in both objects may carry out processing at the same time.

Asynchronous messages are frequently used in real-time systems where operations in different objects must execute concurrently, either for reasons of efficiency, or because the system simulates real-world activities that also take place concurrently. It may be necessary for an operation that has been invoked asynchronously to notify the object that invoked it when it has terminated. This is done by explicitly sending a message (known as a *callback)* to the originating object.

Time constraints :

A sequence diagram can be labelled to document time constraints in various ways. Labels may be included with, descriptions of actions or any time constraints that apply to the execution of operations.

In the above figure each of the messages is simply named with 1,2.., and so on. Time expressions may be associated with the name of the message so that time constraints can be specified for the execution of an operation or the transmission of a message.

The standard functions sendTime (the time at which a message is sent by an instance) and receiveTime (the time at which an instance receives a message) give times when applied to message names. Thus 1. sendtime gives the time that the message 1 is sent.

Construction marks may also be used to show a time interval with a constraint. This is shown in above Figure to show the interval between the receipt of message 2 and sending message 3. Time constraints are frequently used in modelling real-time systems where the application must respond within a certain time, typically for reasons of safety or efficiency. For most other information systems time constraints are not significant and only the sequence of the messages matters.

A message arrow here indicates that the time taken to send a message is not significant in comparison to the time taken for operation execution. There is consequently no need to model another activity during the period while a message is in transit. In some applications the length of time taken to send a message is itself significant.

For example, in distributed systems messages are sent over a network from an object on one computer to another object on a different computer. If the transit time for a message is significant the message arrow is slanted downwards so that the arrowhead (the arrival of the message) is below (later than) the tail (the origination of the message).

Branching :

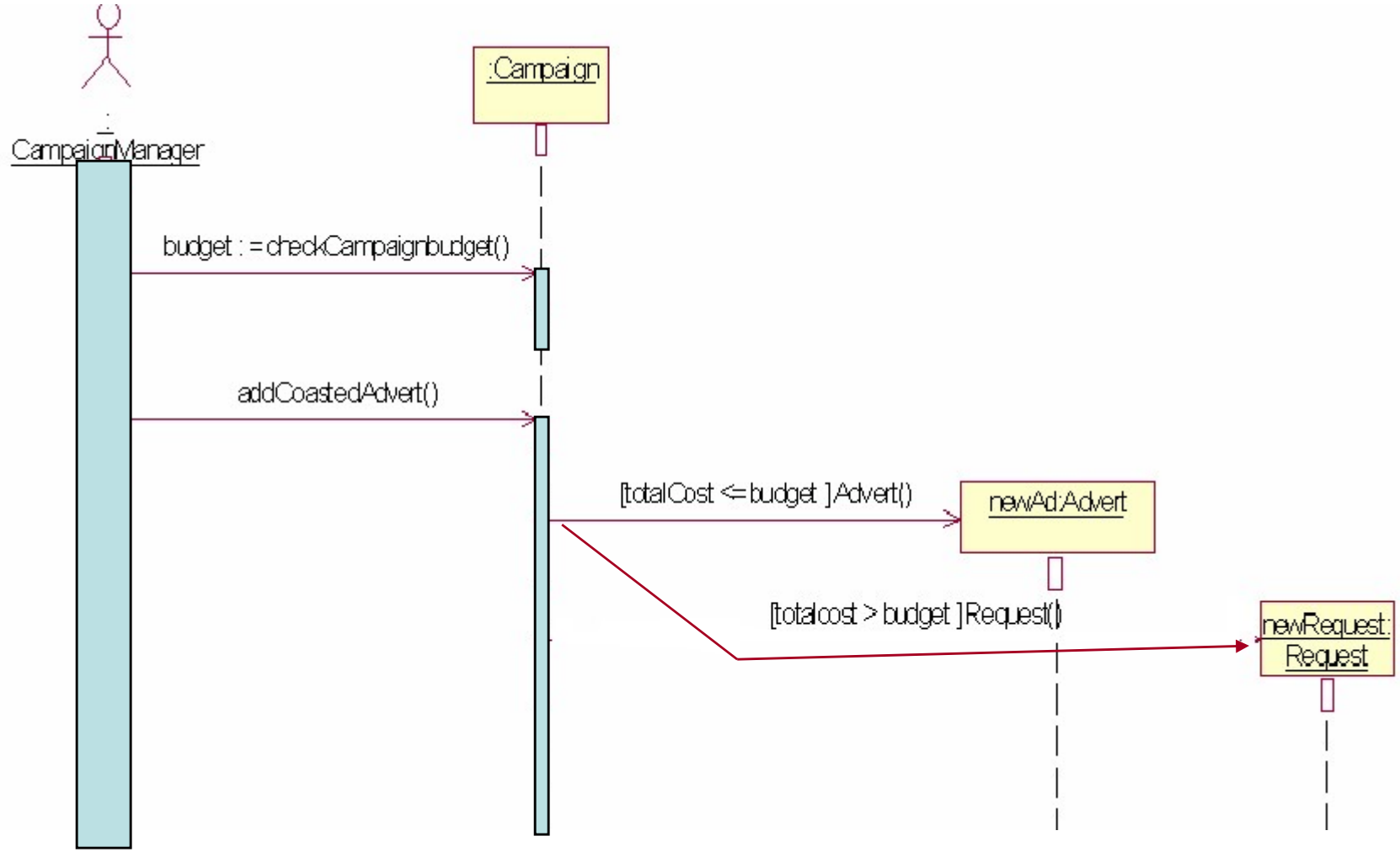
The interactions so far have only one execution path, although some have iterations during their execution. Some interactions have two or more alternative execution pathways. Each reflects a branch in the possible sequence of events for the use case it represents.

The notation for branching is illustrated in the following Figure . This shows a sequence diagram for the use case Add a new advert to a campaign if within budget .

The use case description is as follows.

A new advertisement is added to a campaign by the campaign manager only if the campaign budget is not exceeded by adding the new advert. If adding the advertisement would cause the budget to be exceeded then a campaign budget extension request is generated. This will be recorded for later reference. The budget extension request is printed and sent to the client at the end of the day.

Sequence diagram for " Add a new Advert to a campaign if within budget " showing branching



Here messages 3 and 4 will be originated from the same Campaign object.

The first part of this sequence diagram is identical to that for Check campaign budget but only the checkCarnpaignBudget () message has been shown. Comparing with previous figure where all the messages that result from the execution of the operation checkCampaignBudget () are shown here.

The branching is seen where two messages Advert () and Request () both start from the same point on the Campaign lifeline. Each branch is followed only if the branch condition is true; this is shown in square brackets before the message label. The fact that the Advert () message is above the Request () message does not imply a time sequence since the two branches are actually alternative execution pathways. Only one branch is followed during anyone execution of the use case.

The branching notation can be used at a generic level to create a sequence diagram that represents all possible sequences of interaction for a use case. Such a generic diagram will typically show communication between anonymous objects rather than particular instances.

In general looping and branching constructs correspond respectively to iteration and decision points in the use case. When drawn at an instance level a sequence diagram shows a specific interaction between particular objects. The two kinds of sequence diagram (generic and instance level) are equivalent to one another if the interactions implied by the use case contain no looping or branching constructs.

Managing sequence diagrams

On occasions it is necessary to link two or more sequence diagrams together. It may be that a single sequence diagram is too complex and unwieldy to represent an interaction in an easily assimilable fashion.

Sometimes the interaction involves too many lifelines to place on a single diagram or perhaps there is a subsequence that is common to several interactions. Another possibility is that part of the interaction involves complex messaging between members of a group of objects and that this part of the interaction is best shown separately.

One approach for this problem is to split a complex diagram into two or more smaller diagrams with the connections between the diagrams indicated by message arrows that are left in mid-air and do not end at a lifeline.

For example the sequence diagram in “ Add a new Advert to a campaign “ is redrawn as two sequence diagrams in Figure 1 and Figure 2 each of which shows fewer lifelines that the original. This approach relies on clear annotation in each diagram to show how it is related to other diagrams.

Figure 1 : first part of interaction for use case **Add a new Advert to a Campaign**

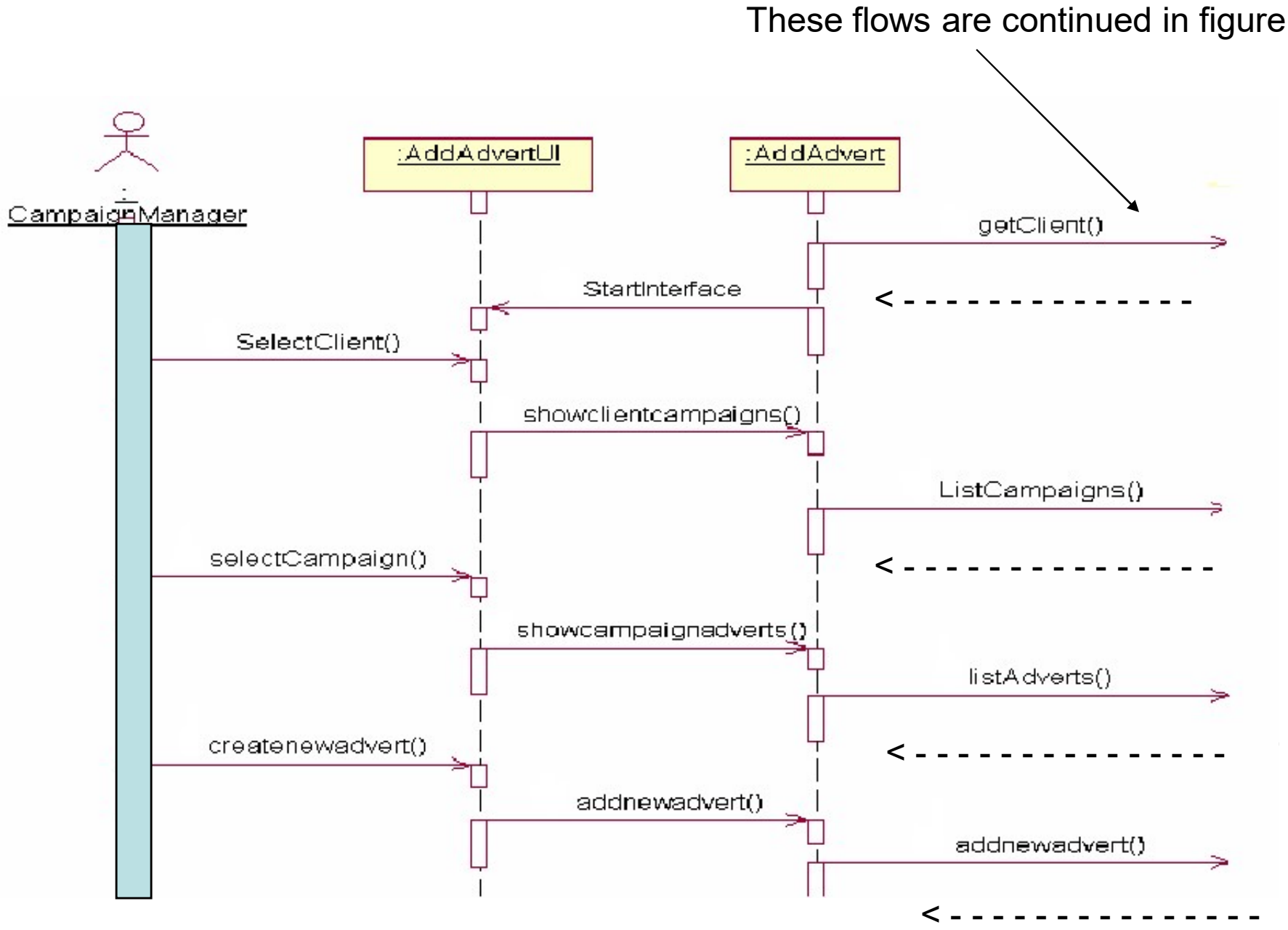
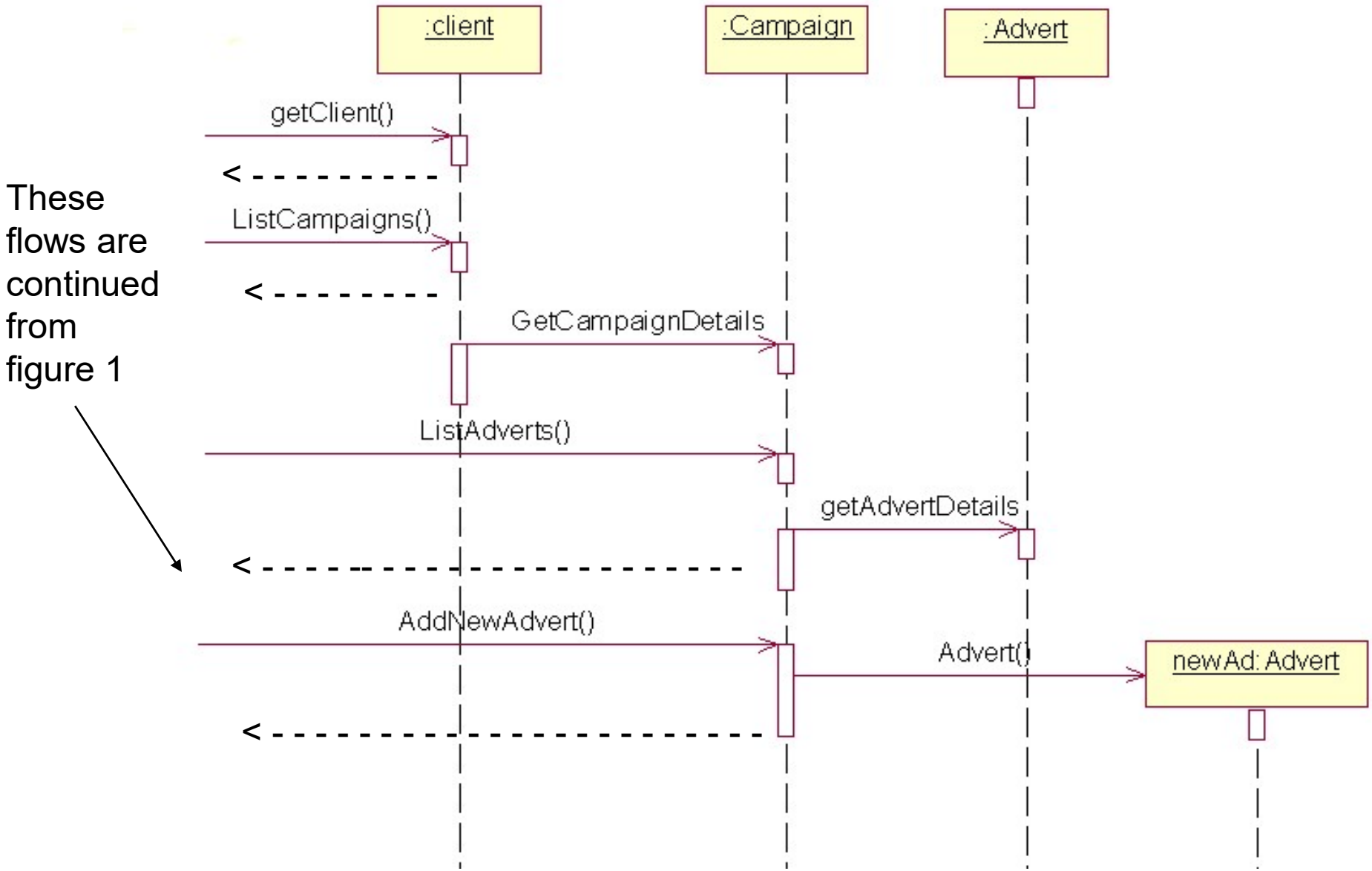
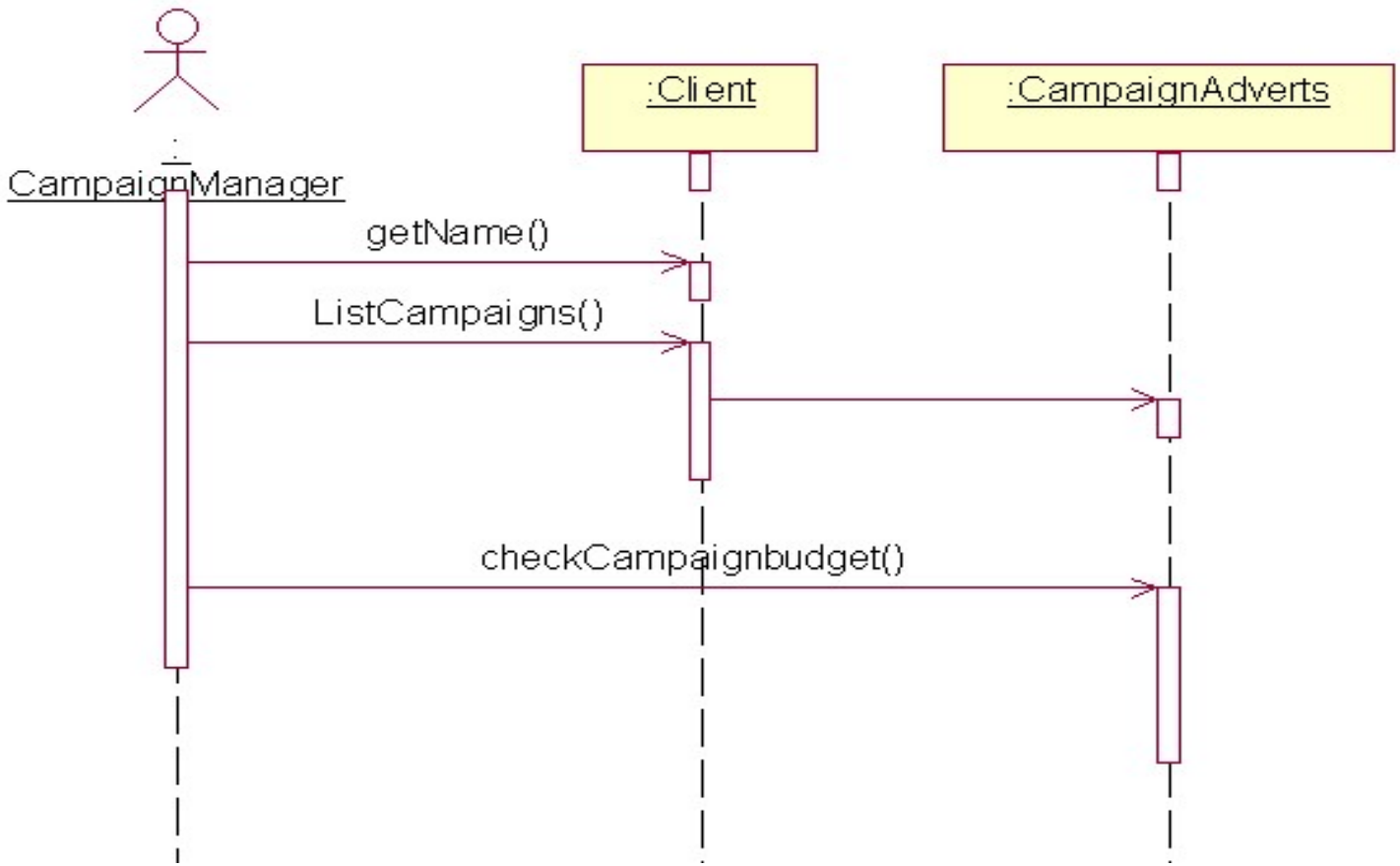


Figure 2 : Second part of interaction for use case **Add a new Advert to a Campaign**



Another approach is to represent a group of objects by a single lifeline. This is shown in the following Figure , where the lifeline for the object group CampaignAdverts represents the Campaign and Advert objects.

This approach is useful when drawing an interaction at a high level and not showing the detailed interaction within a group of objects. Here in this notation, it is implicit that received messages are dealt with by interaction between members of the group or by a single object in the group.



Interaction Collaboration Diagrams

Collaboration diagrams are the second kind of interaction diagram in the UML diagrams. They are used to represent the collaboration that realizes a use case.

Basic concepts and notation

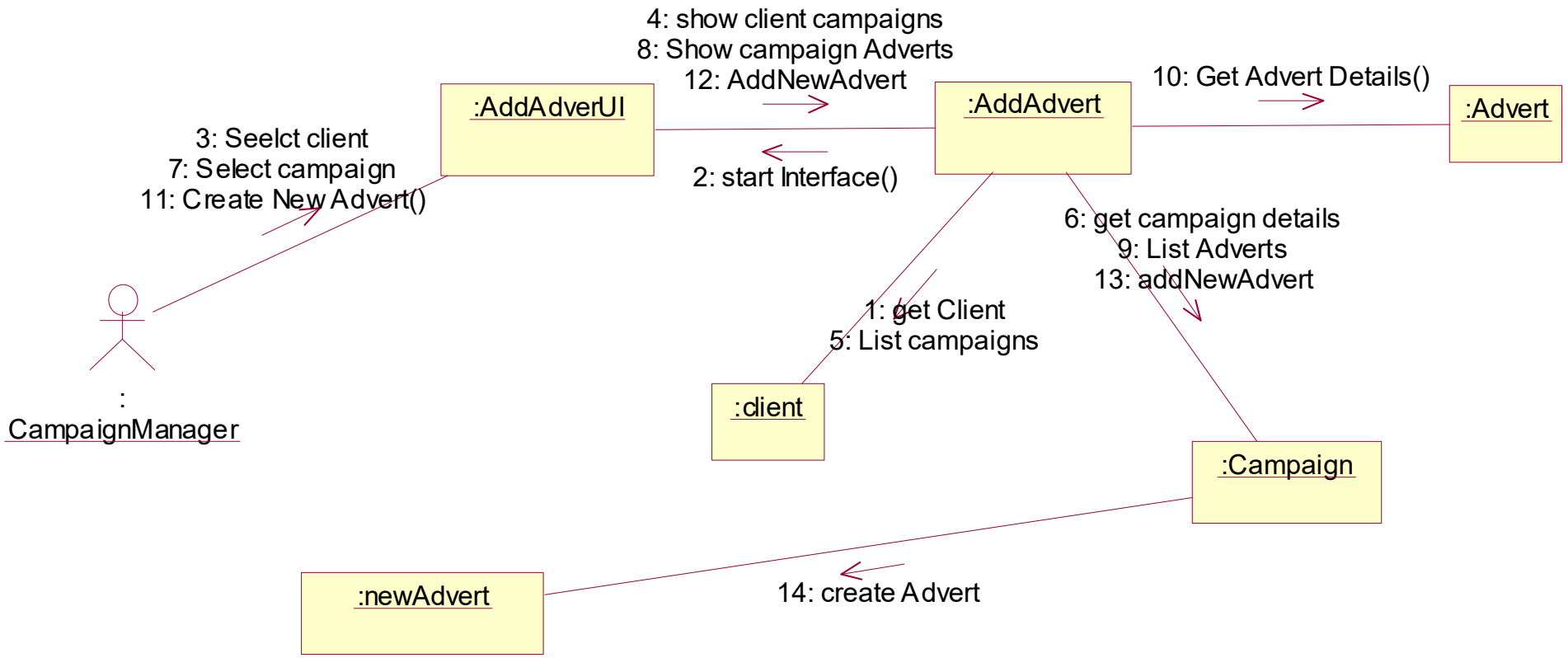
Collaboration diagrams have many similarities to sequence diagrams.
1.They express the same information in a different format, and, like sequence diagrams, they can be drawn at various levels of detail and during different stages in the system development process.
2. Due to their similar content, collaboration diagrams can be used for the auto-matic generation of sequence diagrams and vice versa.

The most significant difference between the two types of interaction diagram is that a collaboration diagram explicitly shows the links between the objects that participate in a collaboration, as in sequence diagrams , there is no explicit time dimension.

In a collaboration diagram the interaction is drawn on a fragment of a class or object diagram. 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 figure shows collaboration diagram for a use case **Add a New Advert to a Campaign**

Collaboration diagram for a ADD A NEW ADVERT TO A CAMPAIGN



Here collaboration diagram contains , a boundary object : AddAdvertUI and the control object : AddAdvert .This level of detail is sufficient to capture the nature of a collaboration. Since the diagram has no time dimension the order in which messages are sent is represented by sequence numbers.

In this diagram the sequence numbers are written in a nested style to indicate the nesting of control within the interaction that is being modelled. Thus the operation showCampaign Adverts () passes control to the operation listAdverts (), which has one deeper level of nesting. A similar style of numbering is used to indicate branching constructs.

Message labels in collaboration diagrams :

Messages on a collaboration diagram are represented by a set of symbols that are the same as those used in a sequence diagram, but with some additional elements to show sequencing and recurrence as these cannot be inferred from the structure of the diagram. Each message label includes the message signature and also a sequence number that reflects call nesting, iteration, branching, concurrency and synchron-ization within the interaction.

The formal message label syntax is as follows:

[predecessor] [guard-condition] sequence-expression [return-value ':='] message-name' (' [argument-list] ')'

A *predecessor* is a list of sequence numbers of the messages that must occur before the current message can be enabled. This permits the detailed specification of branching pathways. The message with the immediately preceding sequence number is assumed to be the predecessor by default, so if an interaction has no alternative pathways the predecessor list may be omitted without any ambiguity. The syntax for a predecessor is as follows:

sequence-number { ',' sequence-number} *'/'*

The *'/'* at the end of this expression indicates the end of the list and is only included when an explicit predecessor is shown.

Guard conditions are written in Object Constraint Language (OCL) ,and are only shown where the enabling of a message is subject to the defined condition. A guard condition may be used to represent the synchronization of different threads of control.

A *sequence-expression* is a list of integers separated by dots ('.') optionally followed by a *name* (a single letter), optionally followed by a *recurrence* term and terminated by a colon. A sequence-expression has the following syntax:

integer { '.' integer } [narne] [recurrence] ':'

In this expression *integer* represents the sequential order of the message. This may be nested within a loop or a branch construct, so that, for example, message 5.1 occurs after message 5.2 and both are contained within the activation of message 5.

The *name* of a sequence-expression is used to differentiate two concurrent messages since these are given the same sequence number. For example, messages 3.2.1a and 3.2.1b are concurrent within the activation of message 3.2.

Recurrence reflects either iterative or conditional execution and its syntax is as follows:

Branching: '['condition-clause‘] ,

Iteration: ‘ * ‘ ‘ [‘ iteration-clause ‘] '

10. Specifying Operations

Operation specifications play a similar role in the project repository to that of other entries, such as attribute specifications. They support a graphical model by adding precision so that users can confirm the correctness of the model, and designers can use them as a basis for software development. But they are potentially the most complex of all entries in the repository, since they explain the detailed behaviour of the system.

Here we need to consider

- The need for specifying operations .
- The 'contract' as a kind of black box specification - If the behaviour of an operation is simple, a contract that describes only its external interface may be all that is required, and if its behaviour is not yet understood in any detail, a black box specification may be all that is possible.
- Operation's logic or internal behaviour - we have two types.

1. ‘Algorithmic' or 'procedural'
2. ‘Non-algorithmic' or 'declarative'.

A non-algorithmic approach is generally preferred in object-oriented development, but in some situations only an algorithmic approach is sufficiently expressive.

➢ UML do not requires any specific techniques for specifying operations, but activity diagrams can be used to express the logic of an operation in a graphical form . The UML has also a formal language known as the Object Constraint Language (OCL), which is intended mainly for specifying general constraints on a model.

1. The Role of Operation specifications –

Each operation specification is a small but necessary step on a path that begins with a user's idea of a business activity, and leads ultimately to a software system made up of collaborating objects with attributes and methods.

From an analysis perspective, an operation specification is created at a point when the analyst's understanding of some aspect of an application domain can be fed back to users, ensuring that the proposals meet users' needs.

From a design perspective, an operation specification is a framework for a more detailed design specification that later guides a programmer to a method that is an appropriate implementation of the operation in code.

An operation specification can also be used to verify that the method does indeed meet its specification, which in turn describes what the users intended, thus checking that the requirements have been implemented.

New programmers often do not appreciate the need to design, still less specify, an operation before beginning to write it in program code. This is partly because beginners are given simple tasks.

In Object-oriented programming it is important to describe the logical operation of the planned software as early as possible. Modelling object interaction is part of this description process, as it helps to determine the distribution of behaviour among the various classes. A detailed description of individual operations must also now be provided.

There are differences of opinion on how much specification should be done.

1. According to Rumbaugh , only operations that are 'computationally interesting' or 'non-trivial' need to be specified. 'Trivial' operations (e.g. those that create or destroy an instance, and those that get or set the value of an attribute) need not be specified at all. Further, operation specifications are kept simple in form, and consist only of the operation signature and a description of its 'transformation' (i.e. its logic).
2. According to Allen and Frost , recommend the specification of all operations, although the level of detail may vary according to the anticipated needs of the designer.

Among these two the latter approach is the better one, because if the problems that can arise later in a project are not documented, then it is better to go for second approach. It is important to keep at least to a minimal documentation standard, even for operations that are very simple.

Each operation has a number of characteristics, which should be specified at the analysis stage. Users must confirm the logic, or rules, of the behaviour. The designer and the programmer responsible for the class will be the main users of the specification, as they need to know what an operation is intended to do: does it perform a calculation, or transform data, or answer a query? Designers and programmers of other parts of the system also need to know about its effects on other classes. For example, if it provides a service to other classes, they need to know its signature. If it calls operations in other classes or updates the values of their attributes, this may establish dependencies that guide how these classes should be packaged during design or implementation .

Defining operations should neither be begun too early, nor left too late, this task should be left until the class diagram has stabilized. In a project where the development activity has been broken down at an early stage to correspond to separate sub-systems, this may refer only to that part of the class diagram which relates to a particular sub-system. But for any given part of the model, it is important to create all operation specifications before moving into the object design activity.

2. Contracts :

The term 'contract' is a deliberate echo of legal or commercial contracts between people or organizations. Signing a contract involves making a commitment to deliver a defined service to an agreed quality standard.

For example, a small ground-care company has a contract to cut the grass on the lawn in front of the Agate’s building. The contract speifies how often the grass must be cut , the maximum height of the grass immediately after it is cut and how much Agate will pay for the service). The contract does *not* spell out how the work will be done .

In the language of system theory, a contract is an interface between two systems. In this example, Agate is a business system and the ground-care company is a system for ‘cutting Agate's grass. The contract defines inputs and outputs, and treats the grass-mowing system to some extent as a black box, with its irrelevant details hidden. Which details are deemed irrelevant is always a matter of choice, and any contract can specify that some details of the implementation should be visible to other systems. For example, Agate's directors might not wish to permit the ground-care contractor to use toxic pesticides or weedkillers. This can be included as a constraint in the contract.

Analogy between commercial contracts and service relationships between objects is given by Meyer. This word in object-oriented development stresses on the encapsulation of classes and sub-systems in a model.
One of Meyer's principal arguments for using the analogy of a contract is that design-by-contract helps to achieve a software design that is correct in terms of its require-ments. Specification by contract means that operations are defined primarily in terms of the services they deliver, and the 'payment' they receive (usually just the operation signature).

Finally , these various aspects related to Contracts can be summarized as follows.

- The intent or purpose of the operation.
- The operation signature including the return type.
- An appropriate description of the logic.
- Other operations called, whether in the same object or in other objects.
- Events transmitted to other objects.
- Attributes set during the operation's execution.
- The response to exceptions (e.g. what should happen if a parameter is invalid).
- Any non-functional requirements that apply.

3. Describing Operational logic :

The following are the reasons for the classification of operations based on various ways of describing their logic.

First, operations that have side-effects. Possible side-effects include the creation or destruction of object instances, setting or returning attribute values, forming or breaking links with other objects, carrying out calculations, sending messages or events to other objects or any combination of these. A complex operation may do several of these things, and, where the task is at all complex, an operation may also require the collaboration of several other objects. It is partly for this reason that we identify the pattern of object collaboration before specifying operations in detail.

Second, operations that do not have side-effects. These are pure queries; they request data but do not change anything within the system.

Like classes, operations may also have the property of being either {abstract} or {concrete} .Abstract operations have a form that consists at least of a signature, sometimes a full specification, but they will not be given an implementation (i.e. they will not have a method). ‘Typically, abstract operations are located in the abstract superclasses of an inheritance hierarchy. They are always overridden by concrete methods in concrete subclasses.

A specification may be restricted to defining only external and visible effects of an operation, and for this we have two approaches for specifying operational logic.

1. Non- algorithmic approach

2.Algorithmic approach.

A specification may also define internal details, but this is effectively a design activity.

1 Non-algorithmic approaches

A non-algorithmic approach concentrates on describing the logic of an operation as a black box. In an object-oriented system this is generally preferred for two reasons.

First, classes are usually well-encapsulated, and thus only the designers and programmers responsible for a particular class need concern themselves with internal implementation details. Collaboration between different parts of the system is based on public interfaces between. Classes and sub-systems implemented as operation signatures (or message protocols). As long as the signatures are not changed, a change in the implementation of a class, including the way its operations work, has no effect on other parts of the system•

Second, the relatively even distribution of effort among the classes of an object-oriented system generally results in operations that are small and single-minded. Since the processing carried out by anyone operation is simple, it does not require a complex specification.

Even in non object-oriented approaches, a declarative approach has recognized as particularly useful where, for example, a structured decision is made, and the conditions that determine the outcome are readily identified, but the actual sequence of steps in reaching the decision is unimportant.

For situations like this, structured methods uses non-algorithmic techniques called decision tables and pre- and post-condition pairs .

Decision tables

A decision table is a matrix that shows the *conditions* under which a decision is made, the *actions* that may result and how the two are related. They cater best for situations where there are multiple outcomes, or actions, each depending on a particular combination of input conditions.

One common form shows conditions in the form of questions that can be answered with a simple yes or no. Actions are listed, and check-marks are used to show how they correspond to the conditions.

The following is an example of a possible application in the Agate case study. The following Figure shows a corresponding decision table.

A decision table with two conditions and three actions, yielding three distinct rules.

Conditions and Actions	**Rule1**	**Rule2**	**Rule3**
conditions			
Is budget likely to be overspent?	N	Y	Y
Is overspend likely to exceed 2%	-	N	Y
Actions			
No Action	X		
Send Letter		X	X
Set up meeting			X

When a campaign budget is overspent, this normally requires prior approval from the c1ient otherwise Agate is unlikely to be able to recover the excess costs. A set of rules has been established to guide Campaign Managers when they identify a possible problem. If the budget is expected to be exceeded by up to 2%, a letter is sent notifying the client of this. If the budget is expected to be exceeded by more than 2%, a letter is sent and the staff contact also telephones the client to invite a representative to a budget review meeting. If the campaign is not thought likely to exceed its budget, no action is taken

The vertical columns with Y, N and X entries are known as *rules.* Each rule is read vertically downwards, and the arrangement of Ys and Ns indicates which conditions are true for that rule. An X indicates that an action should occur when the corresponding condition is true (i.e. has a Y answer). We can paraphrase the table into text as follows.

Rule 1. If the budget is not overspent (clearly in this case the scale of overspend is irrelevant, indicated by a dash against this condition), no action is required.

Rule 2. If the budget is overspent and the overspend is not likely to exceed 2%, a letter should be sent.

Rule 3. If the budget is overspent and the overspend is likely to exceed 2%, a letter should be sent and a meeting set up.

A single rule may have multiple outcomes that overlap with the outcomes of other rules. Decision tables are very useful for situations that require a non-algorithmic specification of logic, reflecting a range of alternative behaviours. But this is relatively unusual in an object-oriented system, where thorough analysis of object collaboration tends to minimize the complexity of single operations.

Pre- and post-conditions

It suggests, this technique concentrates on providing answers to the following questions .

- What conditions must be satisfied before an operation can take place?
- What are the conditions that can apply (i.e. what states may the system be in) after an operation is completed?

Let us consider an example from Agate. The operation Advert. getCost() If it contains the following signature.

Advert.getCost() : Money

This operation has no pre-condition. (Here the object sending the message must know the identity of the object that contains the operation, but this is not in itself a pre-condition for the operation to execute correctly when invoked).

The post-conditions should express the valid results of the operation upon completion. In this case, a money value is returned .

Pre-condition: none
Post-condition: a valid money value is returned

Consider the another example, for a use case Assign staff to work on a campaign. This involves calling the operation Campaign.assignStaff() for each member of staff assigned. Let us assume that the signature of this operation is as follows:

Campaign.assignStaff(creativeStaff)

This example has one pre-condition: a calling message must supply a valid creative Staff object. There is one post-condition: a link must be created between the two objects.

Pre-condition: creativeStaffObject is valid

Post-condition: a link is created between campaignObject and creativeStaffObject

For many operations in an object-oriented model, such a specification would sufficiently detailed. It must meet the following two tests .

- A user should be able to check that it correctly expresses the business logic .
- A class designer should be able to produce a detailed design of the operation for a programmer to code.

2. Algorithmic approach:

An *algorithm* describes the internal logic of a process or decision by breaking it down into small steps .The level of detail to which this is done varies greatly, depending on the information available at the time and on the reason for defining it.

An algorithm also specifies the sequence in which the steps are performed. In the field of computing and information systems, algorithms are used either as a *description* of the way in which a programmable task is currently carried out , or as a *prescription* for a program to automate the task. This dual meaning reflects the differing perspectives of analysis (understanding a problem and determining what must be done to achieve a solution) and design (the creative act of imagining a system to implement a solution).

An algorithmic technique is almost always used during method design, because a designer is concerned with the efficient implementation of requirements, and must therefore select the best algorithm available for the purpose. But algorithms can also be used with an analysis intention.

A major difference here is that the analyst no need to worry about efficiency, since the algorithm need only illustrate accurately the results of the operation.

Control structures in algorithms *:*

Algorithms are generally organized procedurally, means they use the fundamental programming control structures of sequence, selection and iteration.

We can illustrate this in the Agate case study by considering the operation that calculates the total cost of a campaign. This operation is invoked during the use case Check campaign budget.

Use case description : The campaign budget may be checked to ensure that it has not been exceeded. The current campaign cost is determined by the total cost of all the adverts and the campaign overhead costs.

If there simple formula for this calculation, based on summing the individual total costs of each advert, and adding the campaign overhead costs. For further simplicity, assume that the overhead cost part of the calculation simply involves multiplying the total of all other costs by an overhead rate. To convey an understanding of the calculation, we can begin by representing it as a mathematical formula.

totaLcampaign_cost = (sum of all advert_costs) * overhead_rate

This does not explicitly identify all the steps, but a sequence can be deduced. In fact, several possible sequences can be deduced, but any sequence that always produces a correct result will do.

One possible sequence, at a very coarse level of detail would include the following steps:

1. add up all the individual advert costs;
2. multiply the total by the overhead rate;
3. the resulting sum is the total campaign cost.

For such a simple calculation , the formula itself serves better as a specification, but if they complex, Then we can use Structured English approach.

Structured English **:**

This is a ‘language' of written English that is about halfway between everyday non-technical language and a formal programming language. When it is necessary to specify an operation procedurally, this is the most useful and versatile technique.

Advantages of Structured English :

- It provides readability and understandability same as of everyday English. It also allows the construction of a formal logical structure that is easy to translate into program code.
- Structured English is very easy to write iteratively, at successively greater levels of detail, and it is easily dividing into components that can be reassembled in different structures without a lot of reworking. The logical structure is made explicit through the use of keywords and indentation, while the vocabulary is kept as close as possible to everyday usage in the business context. Above all, expressions and keywords that are specific to a particular programming language are avoided. The result ideally is something that a non-technical user is able to understand, alter or approve, as necessary, while it should also be useful to the designer. This means it must be capable of further development into a detailed program design without undue difficulty.
  The main principles of Structured English are as follows. A specification is made up of a number of simple sentences, each consisting of a simple imperative statement or equation. Statements may only be combined in restricted ways that correspond to the sequence, selection and iteration control structures of structured programming. The very simplest specifications contain only sequences, and differ little from everyday English except in that they use a more restricted vocabulary and style .

Here are some statements that illustrate a typical style of Structured English:

```
get client contact name

sale cost = item cost * (1 - discount rate)

calculate total bonus

description = new description
```

Selection structures show alternative courses of action, the choice between them depending on conditions that prevail at the time the selection is made. For example, an *if-then-else construct,* which has only two possible outcomes, is shown in the following fragment:

```
if client contact is 'Sushila'  then
        set discount rate to 5%
else
        set discount rate to 2%
end if ;
```

If the two alternatives are not really different actions, but are rather a choice between doing something and not doing it, the 'else' branch can be omitted. The following fragment shows this simpler form:

```
if      client contact is 'Sushila'    then      set discount rate to 5%        end if ;
```

Multiple outcomes are handled either by a *case* construct or by a *nested-if.* The following fragment illustrates the case structure:

```
 begin case
case client contact is 'Sushila'
set discount rate to 5%
case client contact is' 'Wu'
set discount rate to 10%
case client contact is 'Luis'
set discount rate to 15%
otherwise set discount rate to 2%
end case ;
```

The following fragment illustrates the nested-if structure:

```
if client contact is 'Sushila' set discount rate to 5%
else if client contact is 'Wu' set discount rate to 10%
else if client contact is 'Luis' set discount rate to 15%
 else set discount rate to 2%    end if ;  end if ;  end if ;
```

The third type of control structure is iteration. This is used when a statement, or group of statements, needs to be repeated. Typically this is a way of applying a single operation to a set of objects. Logically, once something has begun to occur repeatedly, there must be a condition for stopping the repetition. There are two main forms of control of iteration. These differ in whether the condition for ending the repetition is tested before or after the first loop. The next two examples show typical applications of each kind of structure. In the first, the test is applied before the loop is entered, so that if the list is empty no bonus is calculated.

do while there are more staff in the list
calculate staff bonus
store bonus amount
end do ;

The repeat – until construct can also be used in structured english as shown below

repeat
allocate member of staff to campaign
increment count of allocated staff
until count of allocated staff = 10

A Structured English specification can be made as complex as it needs to be, and it can also be written in an iterative, top-down manner.

For example, an initial version of an algorithm is defined at a high level of granularity. Then, provided the overall structure , Then, more detail is easily added progressively. In refining the level of detail, structures can be nested within each other to any degree of complexity. One of the guideline is that a Structured English specification should not be longer than one page of typed A4 size paper, or one screen if it is likely to be read within a CASE tool environment-although in practice the acceptable length of a section of text depends on the context.

Pseudo-code

Pseudo-code differs from Structured English in that it is closer to the vocabulary and syntax of a specific programming language. There are thus many different languages of pseudo-code, each corresponding to a particular programming language. They differ from each other in vocabulary, in syntax and in style.

Structured English avoids language specificity primarily to avoid reaching conclusions about design questions too early. But the final implementation language has been decided early in the project. This can be misleading, as it may be desirable 'at a later stage to redevelop the system in a different programming language. If the operations have been specified in a language-specific pseudo-code, it would then be necessary to rewrite them.

However language-specific it may be, pseudo-code remains only a skeleton of a program, intended only to illustrate its logical structure without including full design and implementation detail. In other words, it is not so much a fully developed program as an outline that can later be developed into program code.

The following pseudo-code for Check campaign budget can be compared with the Structured English version above.

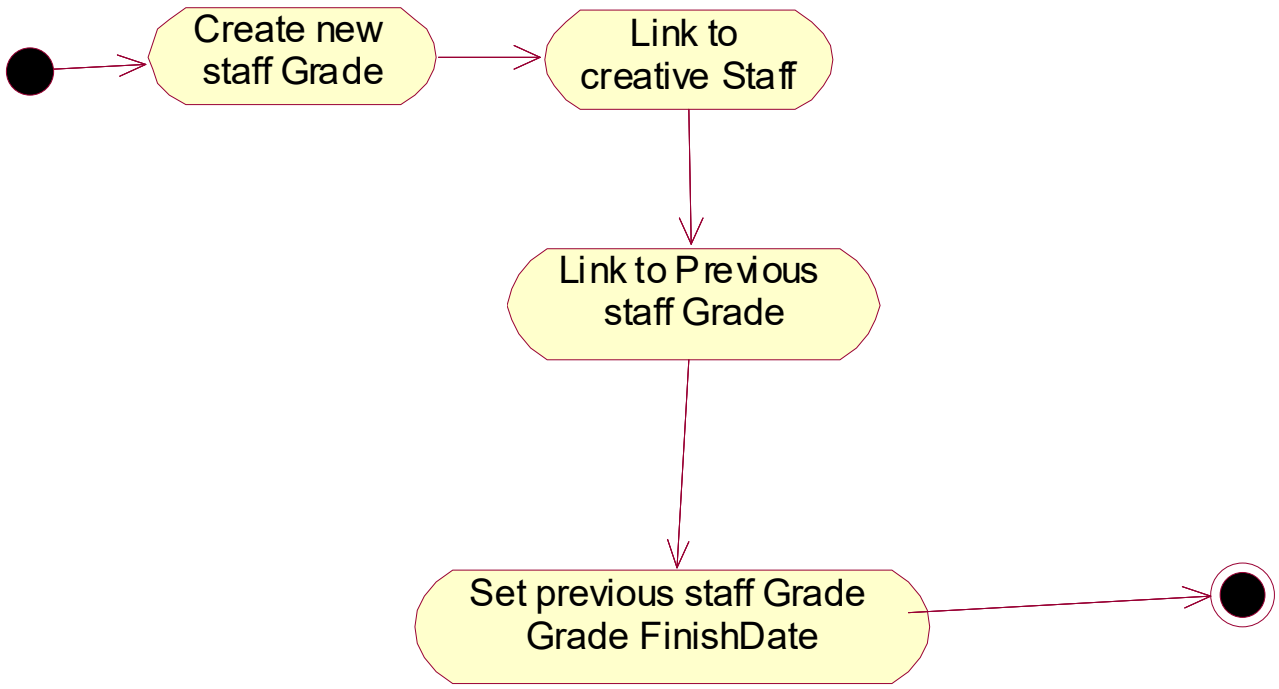
```
{
{ while more adverts:
next advert;
get advertcost;
cumcost = cumcost + advertcost;
endwhile;
}
{ campaigncost = cumcost X ohrate;
get campaignbudget;
case campaigncost >= campaignbudget:
 return warningflag;
endcase
}
}
```

Activity diagrams :

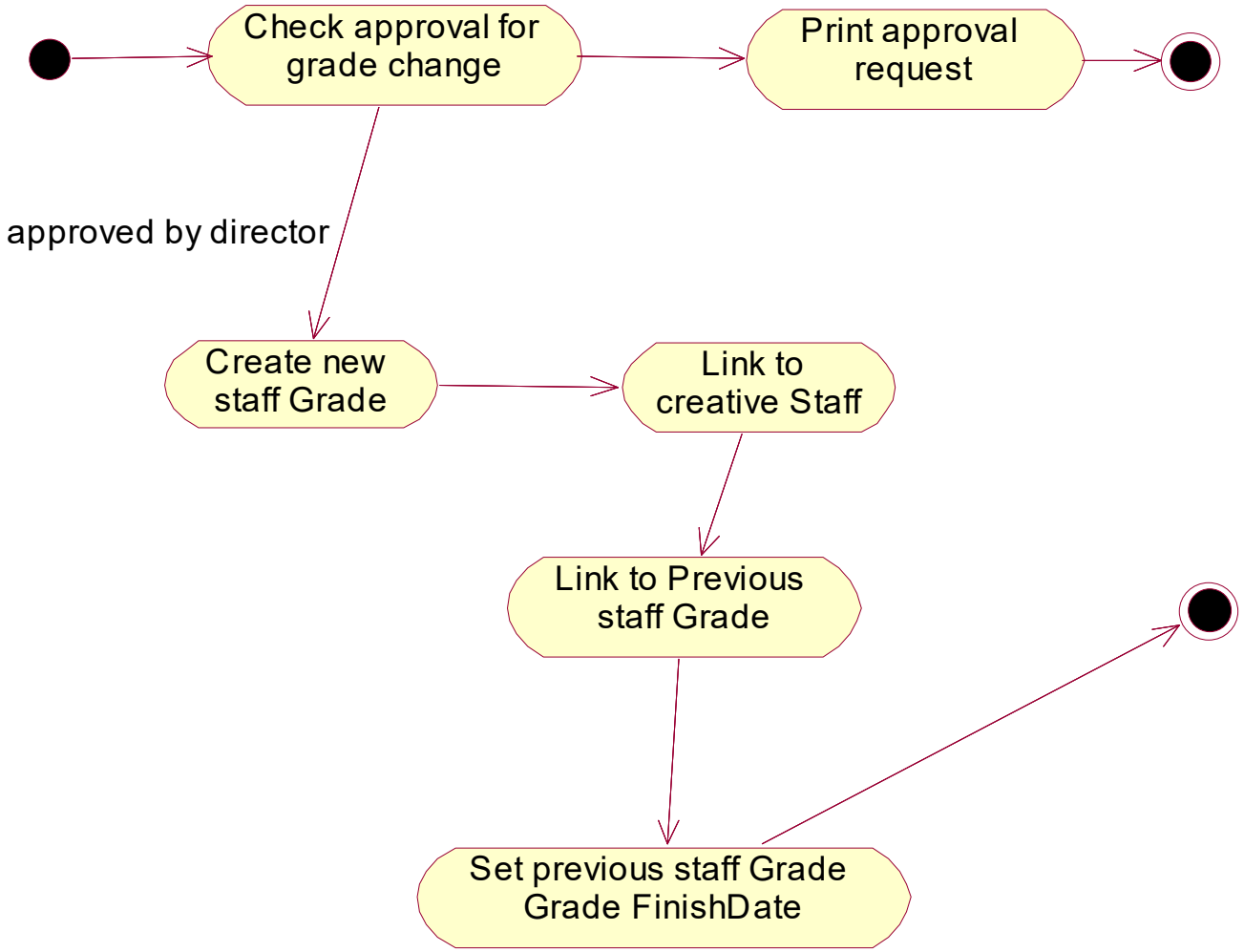
Activity diagrams can also be used to specify the logic of procedurally complex operations. If these activity diagrams are used for specifying the logic of operation, activity **states** in the diagram usually represent steps in the logic of the operation. This can be done at any level of abstraction, so that, if appropriate, an initial high level view of the operation can later be decomposed to a lower level of detail.

Activity diagrams are inherently very flexible in their use, and therefore a little care should be exercised when they are employed in operation specification. A diagram may be drawn to represent a single operation on an object, but it may just as easily be drawn to represent a collaboration between several objects

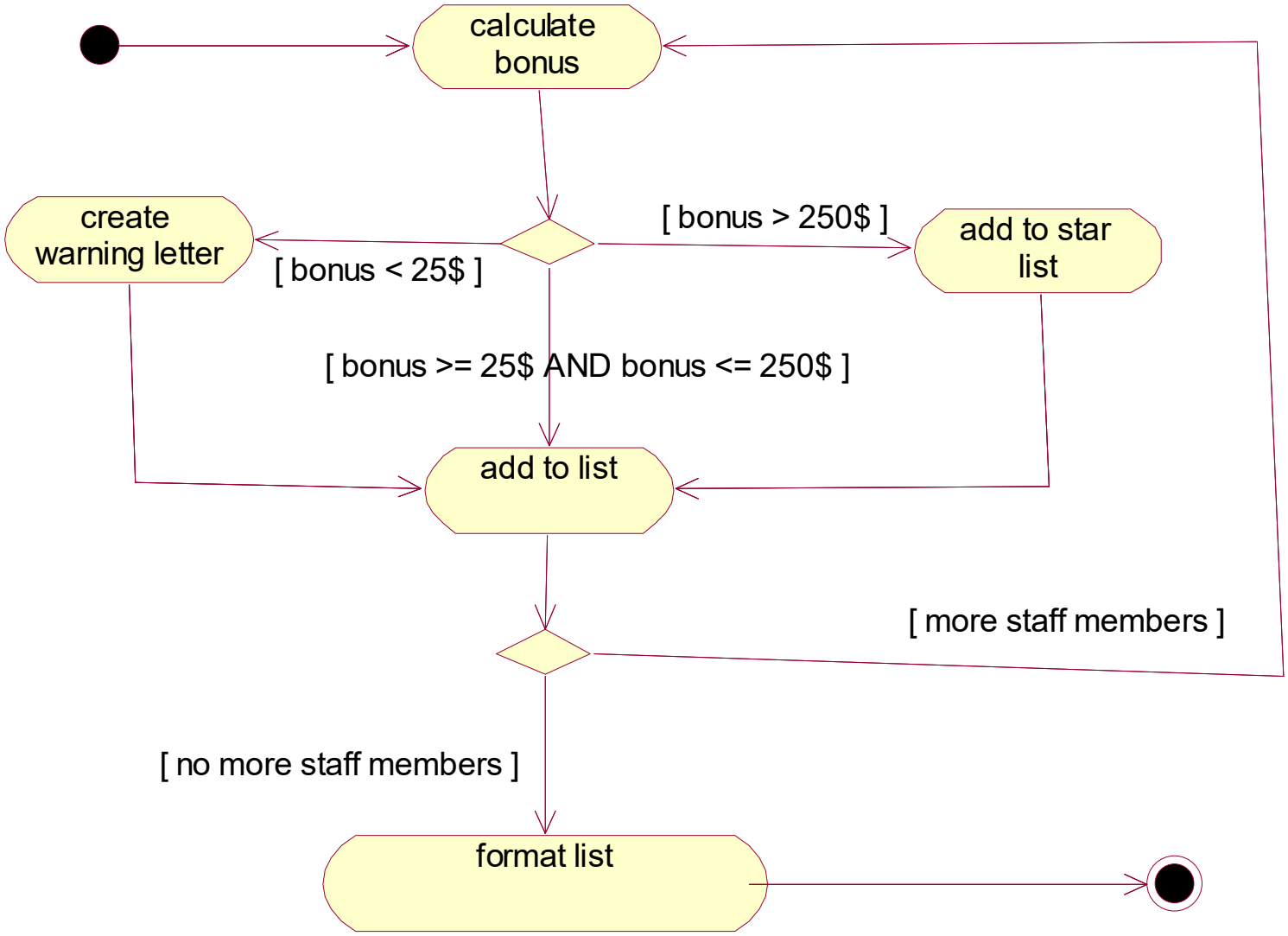
The following figure shows activity diagram for the operation CreativeStaff.changeGrade(). This example contains a single selection.



There is no complex behaviour to be shown at this level of abstraction. However, if we consider the operation logic at a more detailed level some selection logic may become apparent. For this purpose the following figure shows more complex activity diagram for CreativeStaff. changeGrade () with an initial selection to check that approval has been given .



The following figure shows an Activity diagram for prepareBonusList () showing selection and iteration structures.



4 . Object constraint Language (OCL)

In drawing any class diagram, much of the time and effort is spent in working out what constraints to be applied.

For example, the multiplicity of an association represents a constraint on how many objects of one class can be linked to any object of the other class. This particular example can be expressed in the graphical language of the class diagram, but this is not equally so for all constraints. Among those for which it is not true are many of the constraints within operation specifications.

For example, many pre- and post-conditions in a contract are constraints on the behaviour of objects that are party to the contract. Sometimes the definition of such constraints can be done in an informal manner ,but where greater precision is required, OCL provides a formal language.

OCL expressions are constructed from a collection of pre-defined elements and *types,* and the language has a precise grammar that enables the construction of unambiguous statements about the properties of model components and their relationships to each other.

Most OCL statements consist of the following structural elements:

- A *context* that defines a domain within which the expression is valid. This is often an instance of a specific type, for example an object in a class diagram. A link may also be the context for an OCL expression.
- A *property* of that instance which is the context for the expression. Properties may include attributes, association-ends and query operations.
- An OCL *operation* that is applied to the property,includes the arithmetical operators *, +, - and */,* set operators such as size, isEmpty , select and type operators such as oclIsTypeOf.

OCL statements can also include OCL *keywords* that include the logical operators such as **and, or, implies, if, then, else** and **not** and the set operator **in**, printed in bold to distinguish them from other OCL terms and operations. Together with the non-keyword operations mentioned above, these can be used to define complex pre-and post-conditions for an operation.

The following table gives some examples of expressions in OCL, mainly adapted from the OCL Specification, which is part of the UML Specification .All have an object of some class as their context. The figure shows examples of OCL syntax and an interpretation of the meaning of each.

OCL Expression	Interpretation
Person self.gender	In the context of a specific person, the value of the property 'gender' of that person-i.e. a person's gender.
Person self.savings >= 500	The property 'savings' of the person under consideration must be greater than or equal to 500.
Person self.husband->notEmpty **implies** self.husband.gender = male	If the set 'husband' associated with a person is not. empty, then the value of the property 'gender' of the husband must be male. The boldface denotes an OCl keyword, but has no semantic import in itself.
Company self.CEO->size <= 1	The size of the set of the property 'CEO' of a company must be less than or equal to 1. That is, a company cannot have more than 1 Chief Executive Officer.
Company self.employee->select (age < 60)	The set of employees of a company whose age is less than 60.

OCL can specify many constraints that cannot be expressed directly in diagrammatic notation, and is thus useful as a precise language for pre- and post-conditions. The general syntax for operation specification is as follows:

Type::operation(parameter1:type,parameter2:type) : return type

pre: parameter1 operation

parameter2 operation

post: result = ...

Here the contextual type is the Type (normally a class) that owns the operation as a feature.

The **pre**: expressions are functions of operation parameters, while

post: expressions are functions of self, of operation parameters, or of both.

OCL expressions can be written with an explicit Context declaration.

The following example is used to explain this usage, together with an inv: label to denote an invariant

Context Person **inv**:
self.age >= 0

Here the invariant is a person's age must always be greater than or equal to zero- arguably, this should not need specification, but poorly specified computer systems often get the really obvious things wrong. The context of an OCL expression associated with a diagram (such as a class or collaboration diagram) is often obvious; when this is the case, the declaration can be omitted.

One of the useful feature of OCL is its ability to define two values for a single property using the postfix **@pre**. As you might expect, this refers to the previous value of a property, and it can only be used in post-condition clauses. A typical use is to constrain the relationship between the values of an attribute before and after an operation has taken place.

Consider the following decision table , defines different actions depending on changes in the estimated cost of a campaign in comparison with its budget. If the new estimated cost is greater than the old estimated cost, but exceeds the budget by no more than 2%, the value of this attribute is set to true, flagging a need to generate a warning letter to the client.

Conditions and Actions	**Rule1**	**Rule2**	**Rule3**
conditions			
Is budget likely to be overspent?	N	Y	Y
Is overspend likely to exceed 2%	-	N	Y
Actions			
No Action	X		
Send Letter		X	X
Set up meeting			X

We can model this in a very simple way by adding an attribute Campaign. clientLetterRequired.

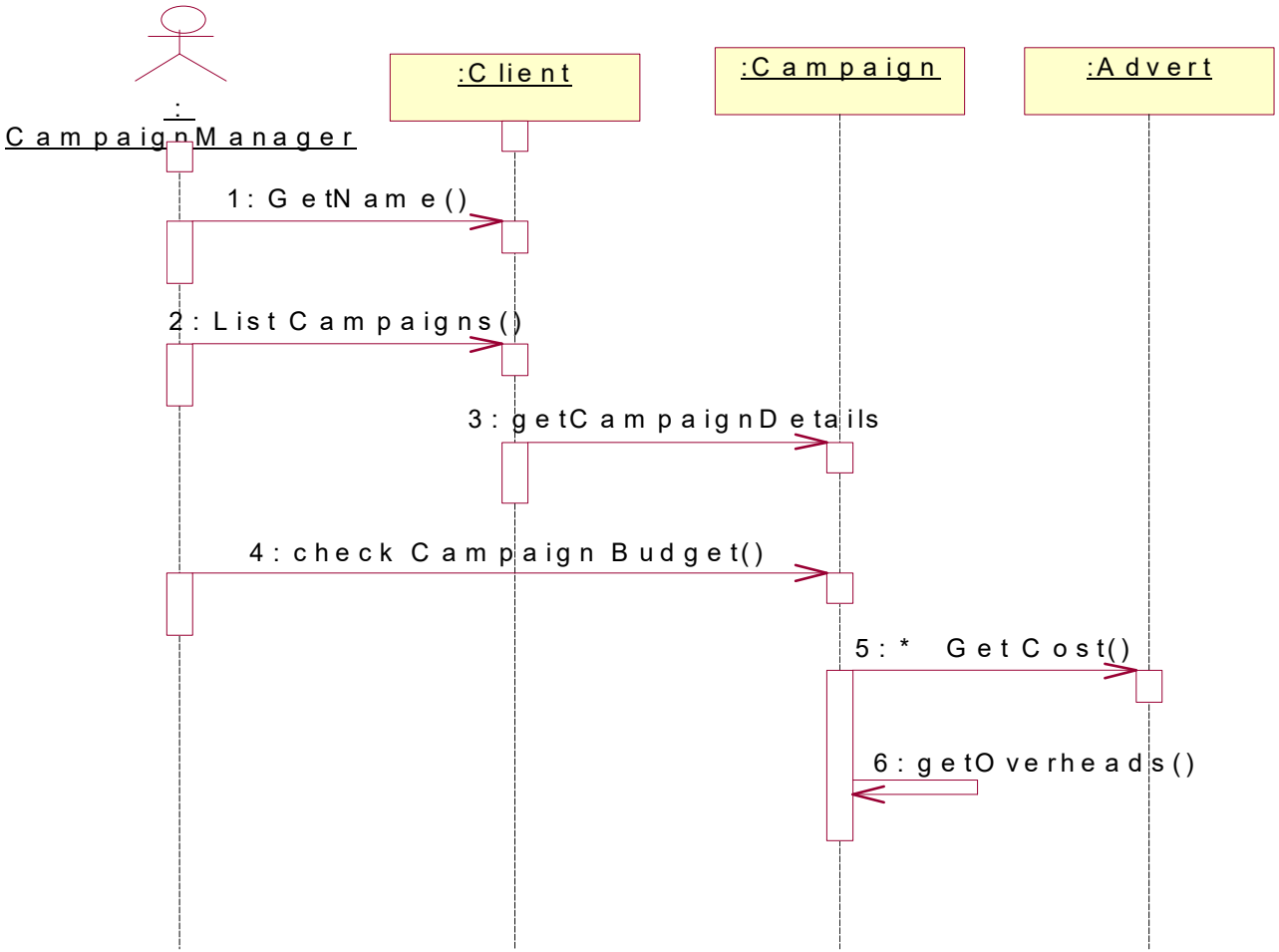
We can write part of the logic in OCL as follows:

Context Campaign **inv:**
post: if estimatedCost > estimatedCost@pre and estimatedCost > budget **and** estimatedCost <= budget * 1.02 **then**
self.clientLetterRequired : Boolean = 'true'
endif

Operation specifications frequently include invariants. When an invariant is associated with an operation specification it describes a condition that always remains true for an object, and which must therefore not be altered by an operation side-effect. Formal definition of invariants is valuable because they provide rigorous tests for execution of the software.

5. Creating an Operation specification

The following figure shows the sequence diagram for the use case **Check campaign budget** .In this example the message checkCampaignBudget invokes the operation Campaign. checkCampaignBudget ().



A specification for **Campaign. checkCampaignBudget ()** is given below.

Operation specification: checkCampaignBudget
Operation intent: return difference between campaign budget and actual costs.

The invocation appears not to require anyparameters, but does have a return type that we can expect it to contain a numerical value. Let us assume that there is a Money type available. The signature is shown below, followed by the pre- and post-conditions.

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

logic description (pre- and post-conditions):
pre:self ->exists
post:campaignBudget = self.estimatedCost
committedExpenditure = self.adverts.estimatedCost->sum

From the sequence diagram it gives, this operation calls two other operations and these must be listed. In a full specification.

Two Other operations called are :

Advert.getCost and self.getOverheads

Events transmitted to other objects: none

The only messages are those required to *call the operations just mentioned, whose return values are required by this operation. An 'event' is a message that starts another distinct thread* of *processing .*

Attributes set: none

This is a query operation whose only purpose is to *return data already stored within the system.*

Response to exceptions: none defined

Here we could define how the operation should respond to *error conditions, e.g. what kind* of *error message will be returned if a calling message uses an invalid signature.*

Non-functional requirements: none defined

11.Specifying Control

The various types of UML notations enable us to model the static structure of an application (class diagrams) and the way in which objects interact (sequence and collaboration diagrams). Another important aspect of an application that must be modelled is the way that its response to events can vary depending upon the passage of time and the events that have occurred already.

For an application such as a real-time system it is easy to understand that the response of the system to an event depends upon its state. For example, an aircraft flight control system should respond differently to events (for example, engine failure) when the aircraft is in flight and when the aircraft is taxiing along a runway. A general example is that of a vending machine, which does not normally dispense goods until an appropriate amount of money has been inserted.

This variation in behaviour is determined by the state of the machine-which depends on whether or not sufficient money has been inserted to pay for the item selected. In reality, of course, the situation is more compli-cated than this. For example, even when the correct amount of money has been inserted; the machine cannot dispense an item that is not in stock.

Objects can have similar variations in their behaviour dependent upon their state. So, It is important to model state dependent variations in behaviour since they represent constraints on the way that a system should behave.

1. State Chart diagram :

The statechart is a versatile technique, and can be used within an object-oriented approach for other purposes than the modelling of object life cycles. A statechart diagram shows the **states** of a single object, the events or messages that cause a **transition** from one state to another , and the **actions** that result from a state change. As in Activity diagram , statechart diagram also contains special symbols for start state and stop state.

States and Events :

All objects will have a state in a system. The current state of an object is a result of the events that have occurred to the object, and is determined by the current value of the object's attributes and the links that it has with other objects. Some attributes and links of an object are significant for the determination of its state while others are not.

For example, in the Agate case study staffName and staff No attributes of a Staff Member object have no impact upon its state, whereas the date that a staff member started his or her employment at Agate determines when the probationary period of employment ends . The Staff Member object is in the Probationary state for the first six months of employment. While in this state, a staff member has different employment rights and is not eligible for redundancy pay in the event that they are dismissed by the company.

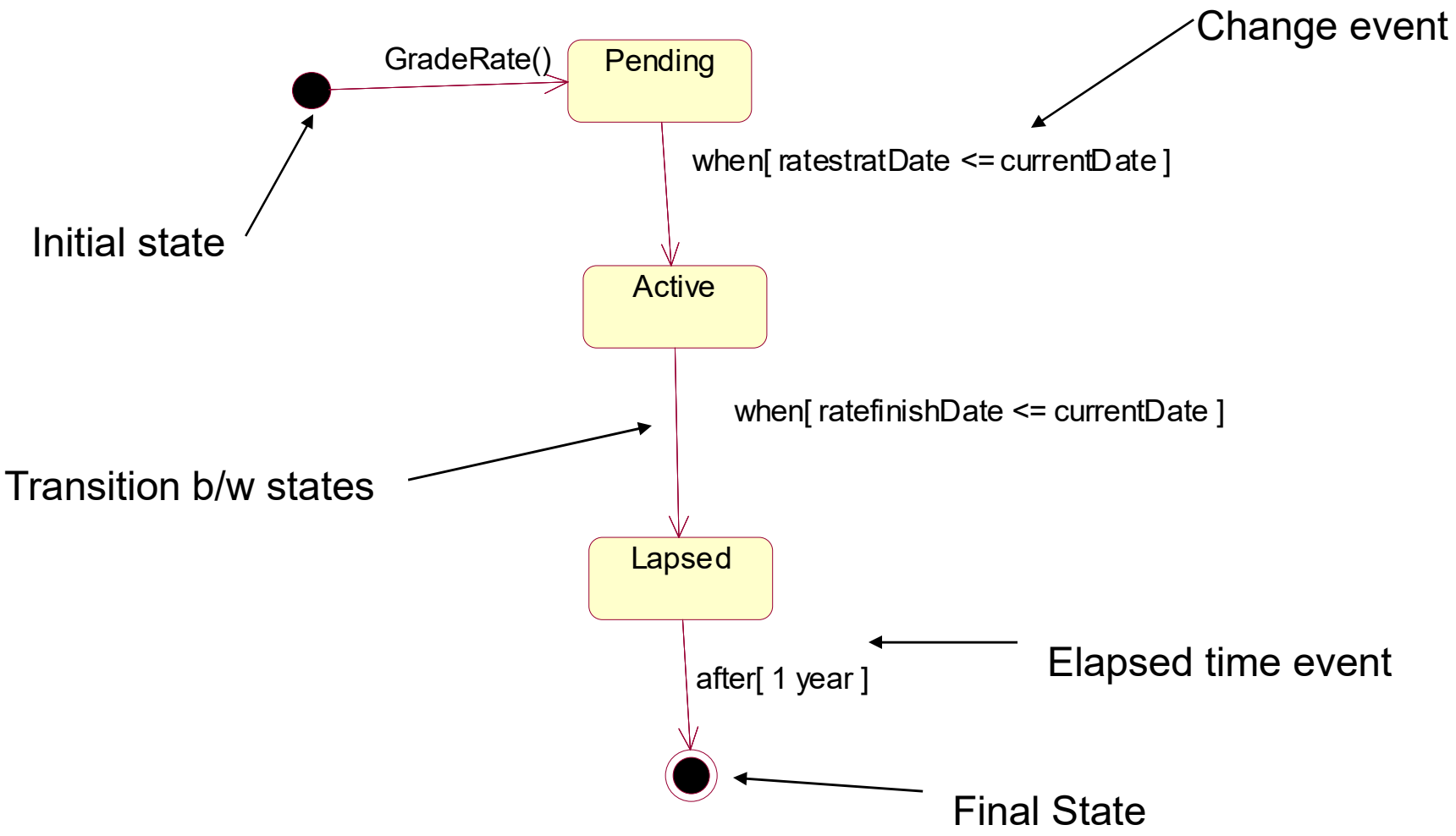
The UML specification defines a state as follows:

A state is a condition during the life of an object or an interaction during which it satisfies some condition, performs some action or waits for some event .... Conceptually, an object remains in a state for an interval of time. The possible states that an object can occupy are limited by its class. Objects of some classes have only one possible state.

For example, in the Agate case study a Grade object either exists or it does not. If it exists it is available to be used, and if it does not exist it is not available. Objects of this class have only one state, which we might name Available. Objects of other classes have more than one possible state.

For example, an object of the class GradeRate may be in one of several states. It may be Pending, if the current date is earlier than its start date, Active, if the current date is equal to or later than the start date but earlier than the finish date , or Lapsed, if the current date is later than the finish date for the grade. If the current date is at least a year later than the finish date then the object is removed from the system. The current state of a GradeRate object can be determined by examining the values of its two date attributes (alternatively, the GradeRate class might have a single attribute2 (an enumerated type-that has an integer value for each possible state) with values that indicate the current state of an object).

It is important that movement from one state to another for a GradeRate object is dependent upon events that occur with the passage of time. The following figure shows a state chart for GradeRate.



Movement from one state to another is called a *transition,* and is triggered by an *event.* When its triggering event occurs a transition is said to *fire.* A transition is shown as a solid arrow from the source state to the target state.

An event is an occurrence of a stimulus that can trigger a state change and that is relevant to the object or to an application.

For example, the cancellation of an advert at Agate is an event that will change the state of the Advert object being cancelled. Just as a set of objects is defined by the class of which they are all instances, events are defined by an *event type* of which each event is an instance. Here this cancellation is defined by the event type cancellationOfAdvert (). An event can have parameters and a return value, and in an object-oriented system it is implemented by a message.

Events can be grouped into several general types. A *change event* occurs when a condition becomes true. This is usually described as a Boolean expression, which means that it can take only one of two values: true or false. Change events are annotated by the keyword *when* followed by the Boolean expression in parenthesis. This form of conditional event is different from a guard condition that is only evaluated at the moment that its associated event fires.

A *call event* occurs when an object receives a call for one of its operations either from another object or from itself. Call events correspond to the receipt of a call message and are annotated by the signature of the operation as the trigger for the transition.

A *signal event* occurs when an object receives a signal. As with call events the event is annotated with the signature of the operation invoked. There is no syntactic difference between call events and signal events. It is assumed that a naming convention is used to distinguish between them.

The basic syntax for a call or signal event is: event-name' (' parameter-list 'j'

where the parameter-list contains parameters of the form:

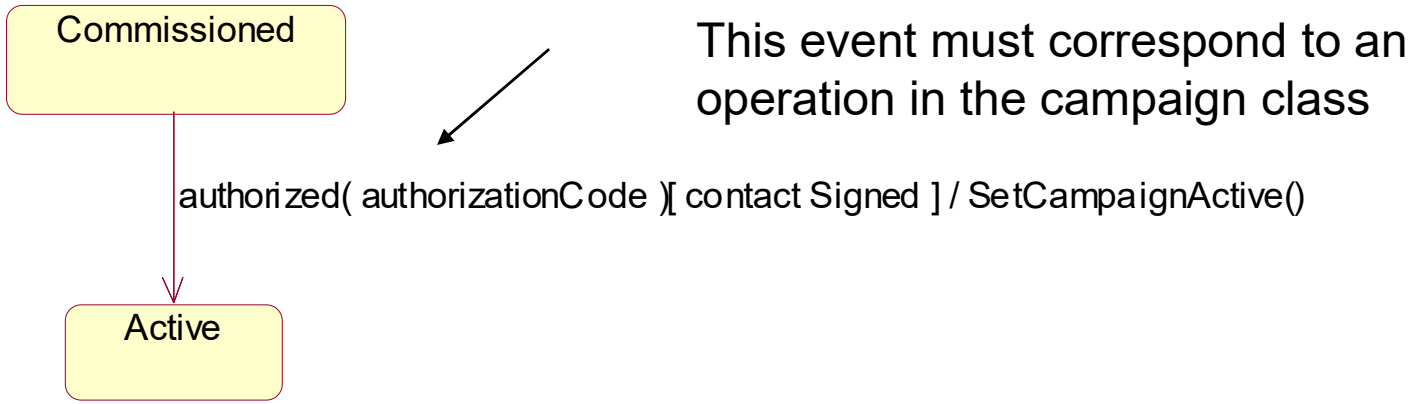
parameter-name':' type-expression
separated by commas. Characters in single quotes, such as '(', are *literals* that appear as part of the event.

An *elapsed-time event* is caused by the passage of a designated period of time after a specified event .Elapsed-time events are shown by time expressions as triggers for the transitions. The time expression is placed in parentheses and should evaluate to a period of time. It is preceded by the keyword *after* and if no starting time is indicated it reflects the passage of time since the most recent entry to the current state.

Basic Notation

The *initial state* of a life cycle is indicated by a small solid filled circle. The initial state is a notational convenience, and an object cannot remain in its initial state but must immediately move into another named state. In previous Figure the GradeRate object enters the Pending state immediately on its creation. A transition from the initial state can optionally be labelled with the event that creates the object. The Final state means end point of a life cycle is shown by a bull's-eye symbol. This too is a notational convenience, and an object cannot leave its final state once it has been entered. All other states are shown as a rectangle with rounded corners and should be labeled with a meaningful name. In this example all transitions except the transition from the initial state are triggered by change events. The state chart for a GradeRate object is very simple, since it enters each state only once. Some classes have much more complex life cycles.

The Figure shows the basic notation for a state chart with two states for the class Campaign and one transition between them. A transition should be annotated with a *transition string* to indicate the event that triggers it.



For call and signal events the format of the transition string is as follows:

event-signature ' [‘ guard-condition ‘] ' ‘ / ' action-expression

The event signature takes the following form:

event-name ' (‘ parameter-list ‘) '

A *guard condition* is a Boolean expression that is evaluated at the time the event fires. The transition only takes place if the condition is true. A guard condition is a function that may involve parameters of the triggering event and also attributes and links of object that owns the state chart. A guard condition is contained in square brackets , [, ... '] '. In Figure the guard condition is a test on the contractSigned attribute in the Campaign classand since the attribute is Boolean it could be written follows: [contractSigned]

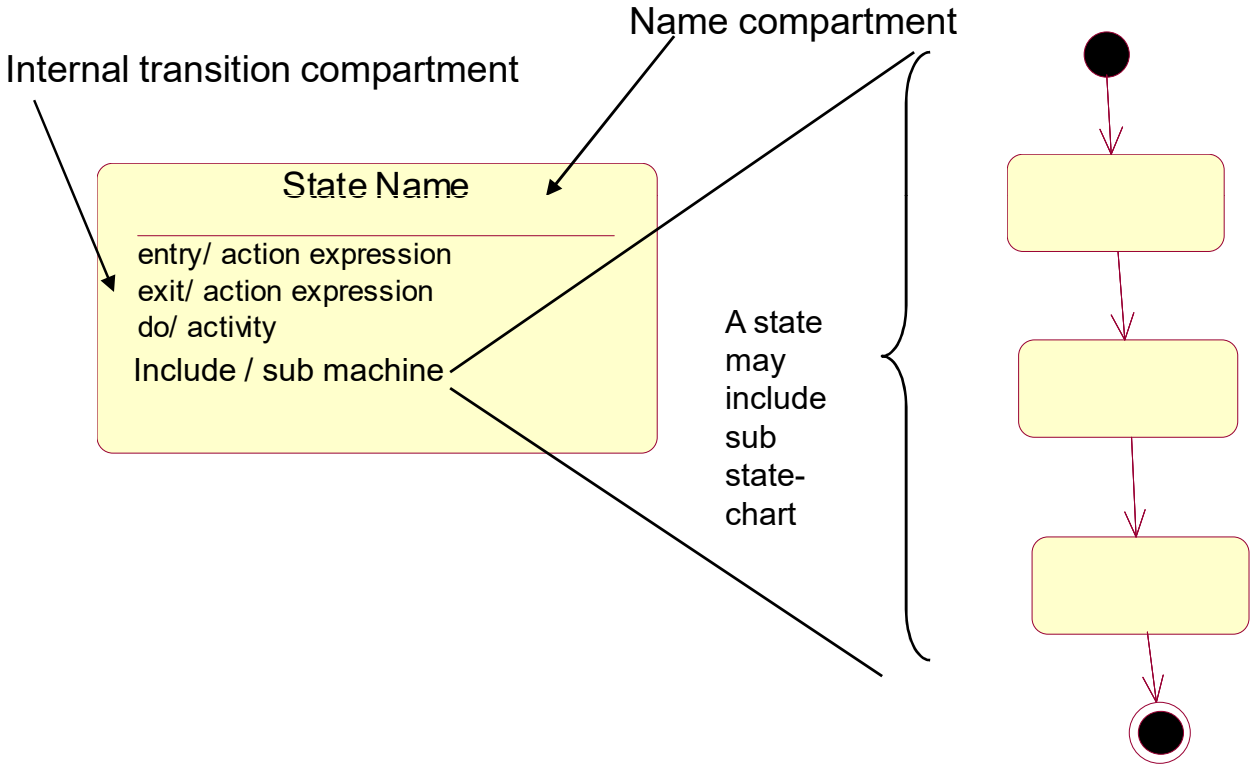
This expression evaluates to true only if contractSigned is true. A guard condition can also be used to test concurrent states of the current object or the state of some other reachable object.

An *action-expression* is executed when an event triggers the transition to fire. Like a guard condition, it may involve parameters of the triggering event and may also involve operations, attributes and links of the owning object.

In the above Figure the action-expression begins with the '/' delimiter character and is the execution of the Campaign object's operation setCampaignActive () . An action-expression may comprise a sequence of actions and include actions that may generate events such as sending signals or invoking operations. Each action in an action string is separated from its preceding action with a semicolon.

The action-expressions that are associated with a transition, can also be useful to model internal actions or activities associated with a state. These actions may be triggered by events that do not change the state, or by an event that causes the state to be entered or by an event that results in exiting the state.

The following Figure the **state symbol** is shown with two compartments, a name compart-ment and an internal transitions compartment.



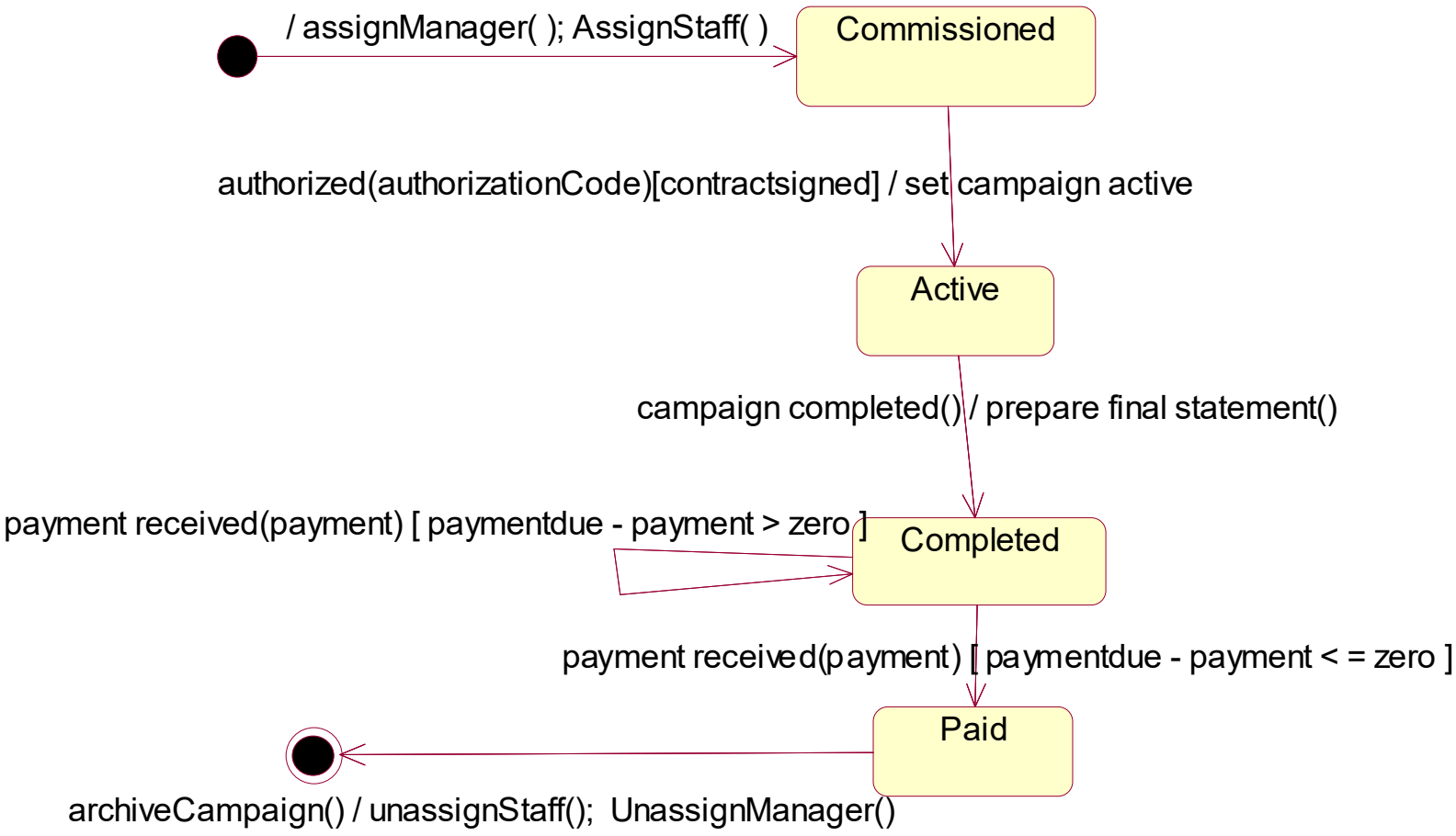
Two kinds of internal event have a special notation. These are the *entry event* and the *exit event,* respectively indicated by the keywords *entry* and *exit.* These cannot have guard conditions as they are invoked implicitly on entry to the state and exit from the state respectively. Entry or exit *action-expressions* may also involve parameters of incoming transitions and attributes and links of the owning object. It is important to emphasize that any transition into a state causes the entry event to fire and all transitions out of a state cause the exit event to fire.

Activities are preceded by the keyword do and have the following syntax:

'do' 'I' activity-name' (' parameter-list ')'

It is also possible to show that a state contains substates by using the keyword *include'* followed by the name of the contained sub-statechart or submachine. Complex states may be represented by a statechart nested within the state. The nesting of one state-chart within another allows the representation of highly complex behaviour. When an activity in a state ends the state is considered completed and the object makes a transition triggered by the completion of this activity. Alternatively an activity may persist as long as the object remains in the state, in which case it does not trigger a transition from the state. The activity will only end when some other specified event triggers a transition from the state.

The following Figure shows a statechart for the class Campaign.



The transition from the initial state to the Commissioned state has been labelled only with an action-expression that comprises the operations assignManager () and assignStaff (). Execution of these operations ensures that when a campaign is created a manager and member(s) of staff are assigned to it. The operations are triggered by the event that creates a Campaign object. The transition from the Completed state to the Paid state has a guard condition that only allows the transition to fire if total amount due (paymentDue) for the Campaign has been completely paid.

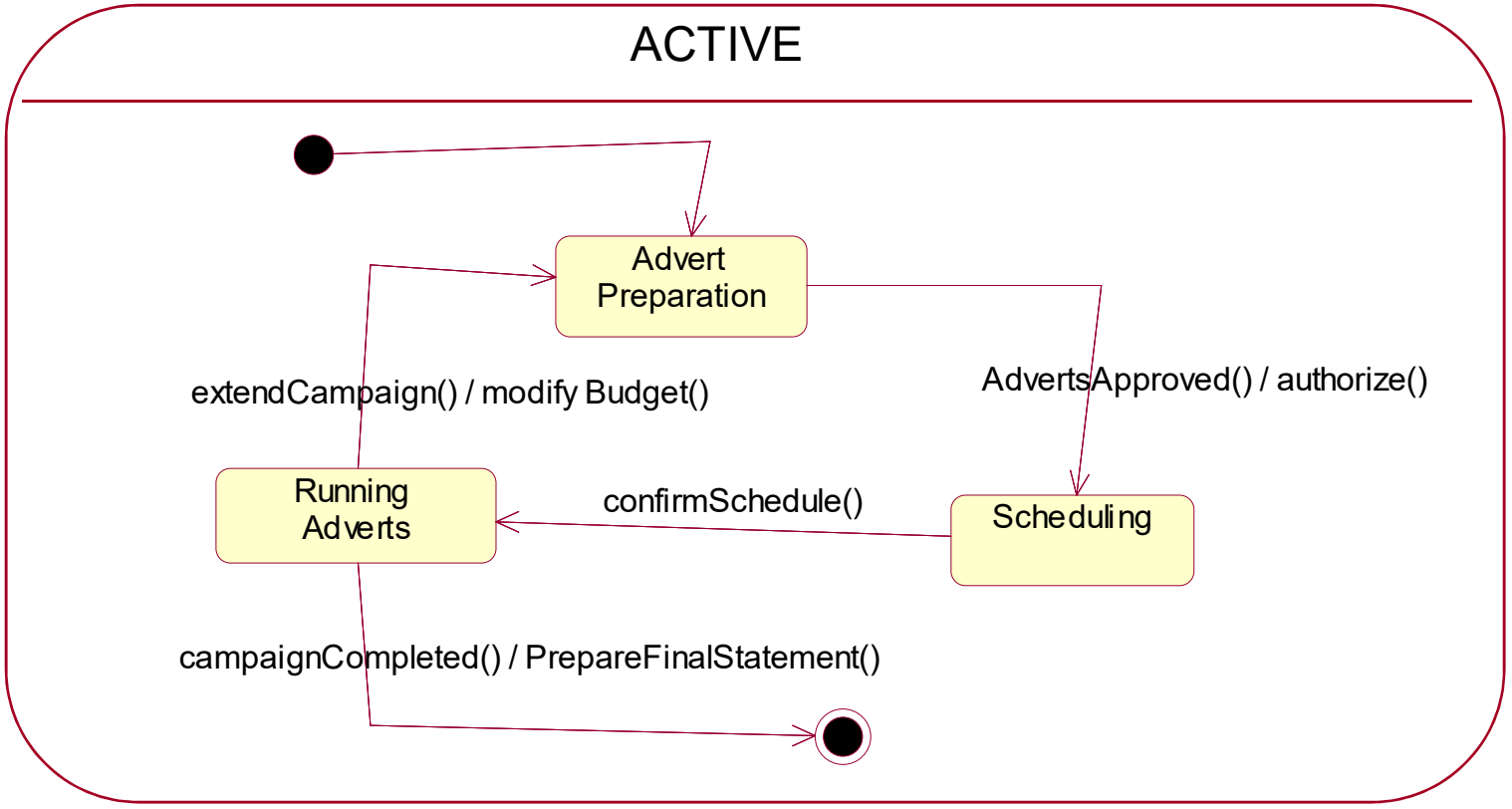
The recursive transition from the Completed state models any payment event that does not reduce the amount due to zero or beyond. Only one of the two transitions from the Completed state can be triggered by the paymentReceived event since the guard conditions are mutually exclusive. It would be bad practice to construct a statechart where one event can trigger two different transitions from the same state. A life cycle is only unambiguous when all the transitions from each state are mutually exclusive.

Further Notation :

The state chart notation can be used to describe highly complex time-dependent behaviour. Hierarchies of states can be nested and concurrent behaviour can also be represented.

1.Nested States :

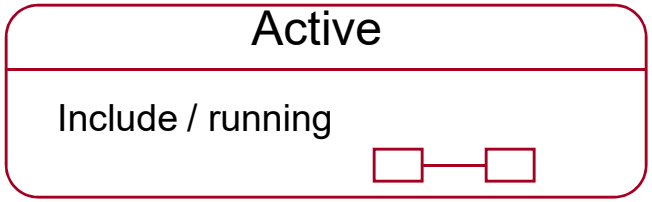
When the state behaviour for an object or an interaction is complex it may be necessary to represent it at different levels of detail and to reflect any hierarchy of states that is present in the application. For example, in the statechart for Campaign the state Active encompasses several *substates.* These are shown below where the Active state is seen to comprise three disjoint substates: Advert Preparation, Scheduling and Running Adverts.



This diagram now shows a single state which contains within it a nested state diagram. In the nested statechart . within the Active state, there is an initial state symbol with a transition to the first substate that a Campaign object enters when it becomes active. The transition from the initial pseudo state symbol to the first substate (Advert Preparation) should not be labelled with an event but it may be labelled with an action. It is implicitly fired by any transition to the Active state. A final pseudostate symbol may also be shown on a nested state diagram. A transition to the final pseudostate symbol represents the completion of the activity in the enclosing state (i.e. Active) and a transition out of this state triggered by the completion event. This transition may be unlabelled (as long as this does not cause any ambiguity) since the event that triggers it is implied by the completion event.

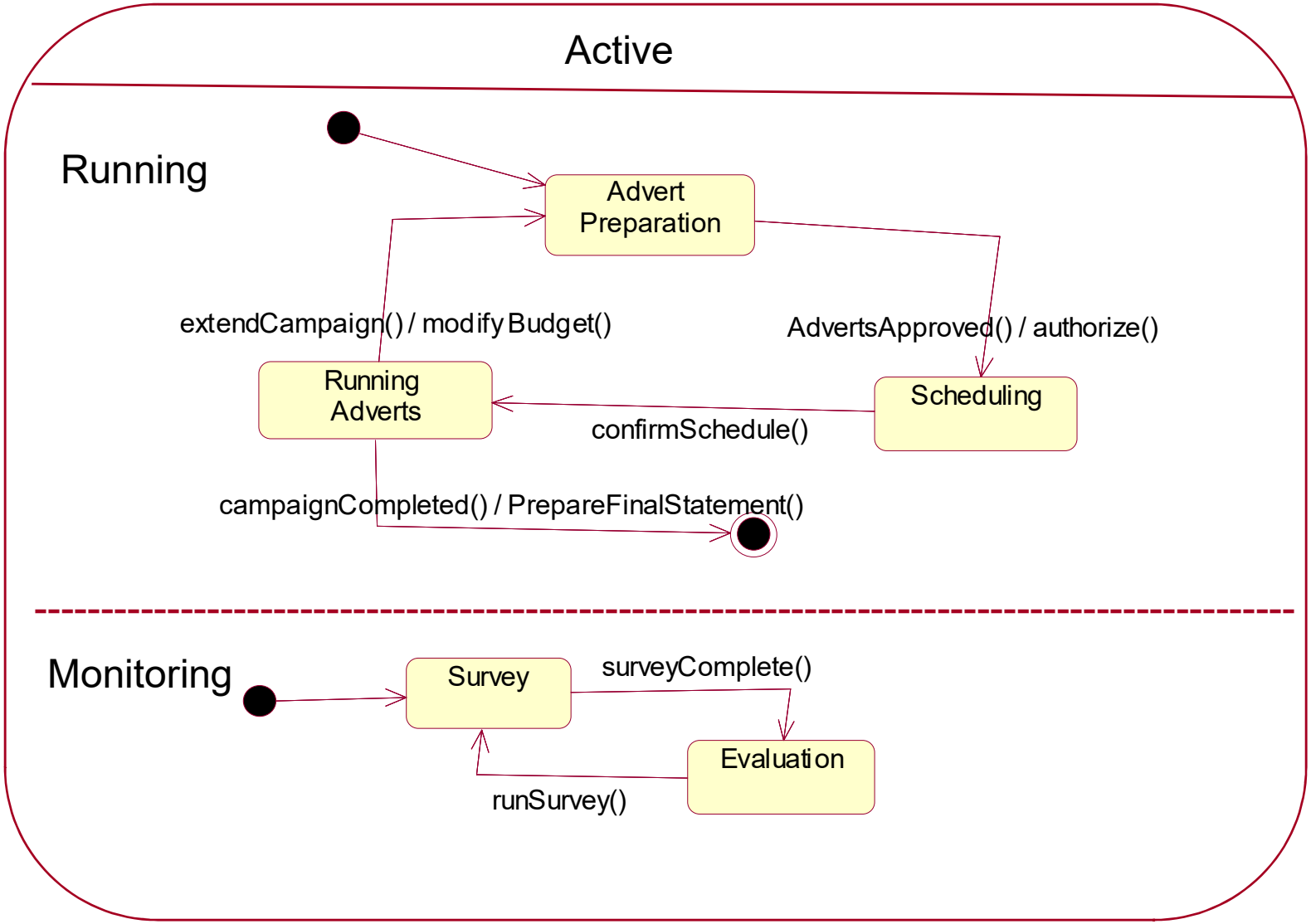
When a campaign enters the Active state in, it first enters the Advert Preparation substate, then if the adverts are approved it enters the Scheduling substate and finally enters the Running Adverts substate when the schedule is approved. If the campaign is deemed completed the object leaves the Running Adverts substate and also leaves the Active enclosing state, moving now to the Completed state . If the campaign is extended while in the Running Adverts substate the Advert Preparation substate is re-entered . A high level state chart for the class Campaign can be drawn to include within the main diagram the detail that is shown in the nested statechart for the Active state if so desired.

If the detail of the sub machine is not required on the higher level statechart or is just too much to show on one diagram the higher level statechart can be annotated with the hidden decomposition indicator icon (two small state symbols linked together) as shown below. The submachine Running is referenced using the include statement.



2.Concurrent states :

Objects can have concurrent states. This means that the behaviour of the object can best be explained by regarding it as a product of two distinct sets of substates, each state of which can be entered and exited independently of substates in the other set. The following figure illustrates this concurrent states form.

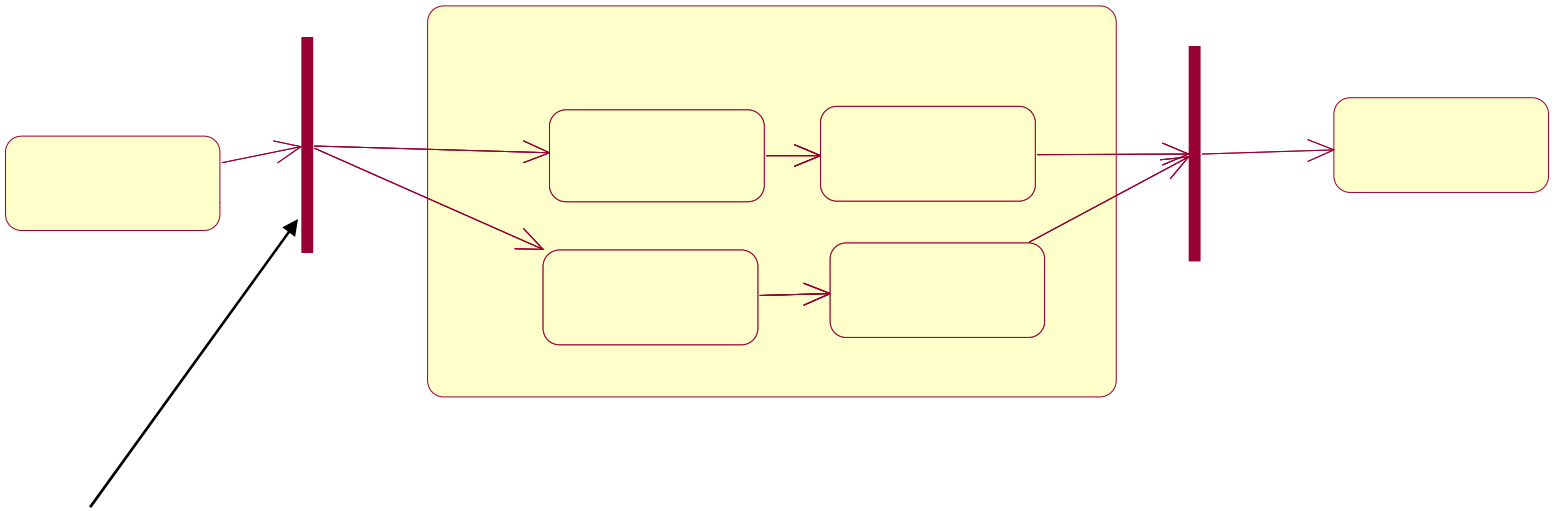


The Active state with concurrent substates.

Suppose in Agate Ltd. , a campaign is surveyed and evaluated while it is also active. A campaign may occupy either the Survey substate or the Evaluation substate when it is in the Active state. Transitions between these two states are not affected by the campaign's current state in relation to the preparing and running of adverts. We model this by splitting the Active state into two concurrent nested statecharts, Running and Monitoring, each in a separate sub-region of the Active statechart. This is shown by dividing the state icon with a dashed line. These concurrent substates for the Active state of the Campaign class are shown in above figure.

A transition to a complex state such as this one is equivalent to a simultaneous transition to the initial states of each concurrent statechart. An initial state must be specified in both nested statecharts in order to avoid ambiguity about which substate should first be entered in each concurrent region. A transition to the Active means that the Campaign object simultaneously enters the **Advert Preparation** and **Survey states**. A transition may now occur within either concurrent region without having any effect on the state in the other concurrent region. However, a transition of the **Active** state applies to all its substates. We can say that the sub states inherit the campaignCompleted () transition from Active state since it applies implicitly to them all. This equivalent to saying that an event that triggers a transition out of the Active state triggers a transition out of any substates that are currently occupied. The nested statechart Monitoring does not have a final state and when the Active state is exited one of the two states **Survey** or **Evaluation** will be occupied.

The following figure shows the use of synchronization bars to show explicitly how an event triggering a transition to a state with nested concurrent states causes specific concurrent substates to be entered and also shows that the super-state is not exited until both concurrent nested state charts are exited.



Synchronization Bar

2.Approaches to prepare STATECHART

State charts can be prepared from various perspectives. The statechart for a class can be seen as a description of the ways that use cases can affect objects of that class. Use cases give rise to interaction diagrams (sequence diagrams or collaboration diagrams) and these can be used as a starting point for the preparation of a statechart.

Interaction diagrams show the messages that an object receives during the execution of a use case. The receipt of a message by an object does not necessarily correspond to an event that causes a state change.

For example, simple 'get' messages like getTitle() ,query messages like listAdverts() are not events in this sense. This is because they do not change the values of any of the object's attributes, nor do they alter any of its links with other objects. Some messages change attribute values without changing the state of an object. For example, a message receivePayment () to a Campaign object will only cause a change of state to Paid if it represents payment at least of the full amount due.

StateCharts can be prepared with the help of Interaction diagrams by using the following two approaches.

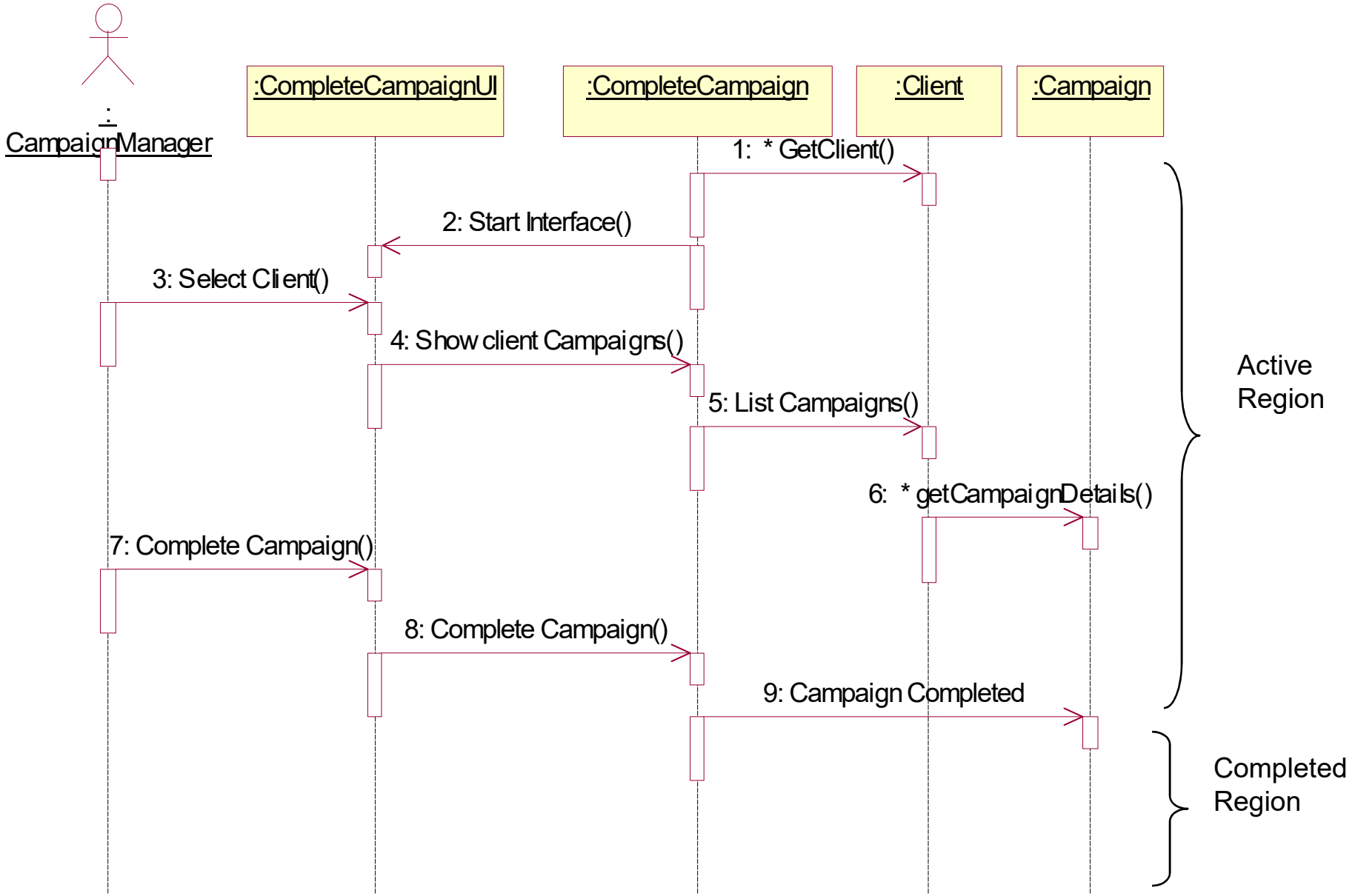
1. A Behavioural approach.
2. A Life Cycle Approach

1. A behavioural approach

The following figure shows a sequence diagram for the use case Record completion of a campaign. The receipt of the message campaignCompleted () by a Campaign object is an event from the perspective of the Campaign object. In this example this event is a call event and causes the campaignCompleted () operation to invoked triggering a transition from the Active state to the Completed state. Incoming messages to an object generally correspond to an event and trigger a state change.

Allen and Frost describes these interaction diagrams can be used to develop a statechart as a behavioural approach.

Sequence diagram for use case Record completion of a campaign



The preparation of a statechart from a set of interaction diagrams using this behavioural approach has the following sequence of steps.

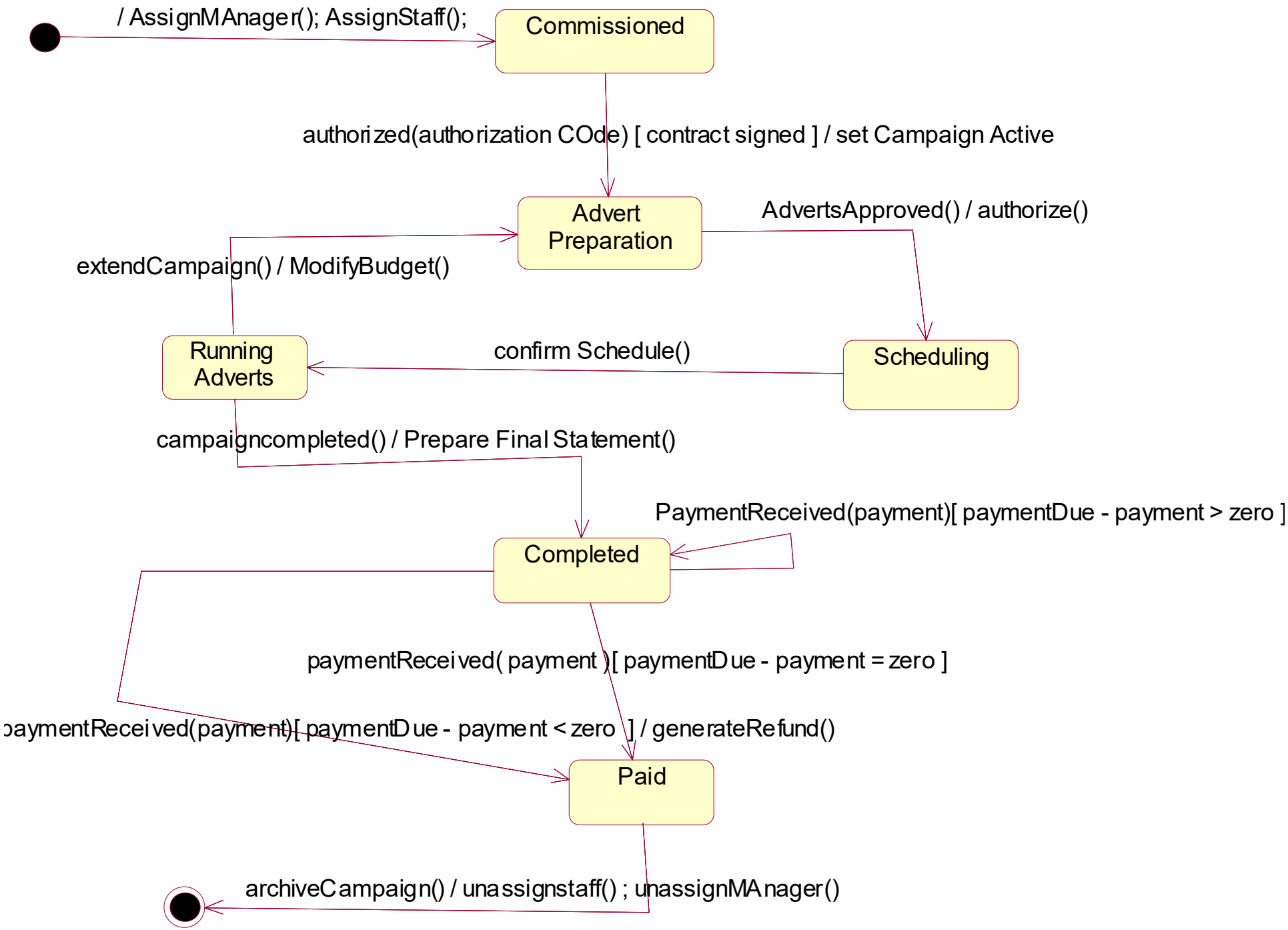
1. Examine all interaction diagrams that involve each class that has heavy messaging.
2. Identify the incoming messages on each interaction diagram that may correspond to events. Also identify the possible resulting states.
3. Document these events and states on a statechart.
4. Elaborate the statechart as necessary to cater for additional interactions as these become evident, and add any exceptions.
5. Develop any nested statecharts if already identified.
6. Review the statechart to ensure consistency with use cases. In particular, check that any constraints that are implied by the state chart are appropriate.
7. Iterate steps 4, 5 and 6 until the statechart captures the necessary level of detail.
8. Check the consistency of the state chart with the class diagram, with interaction diagrams and with any other statecharts.

The above sequence diagram can be annotated to indicate the state **change** that is triggered by the event campaignCompleted (). **In** order to identify all incoming messages that may trigger a state change for an object, all interaction diagrams that affect the object should be examined .Analysis of the interaction diagrams produces a first-cut list of all events (caused by incoming messages) that trigger state changes, and also a first-cut list of states that the object may enter as a result of these events.

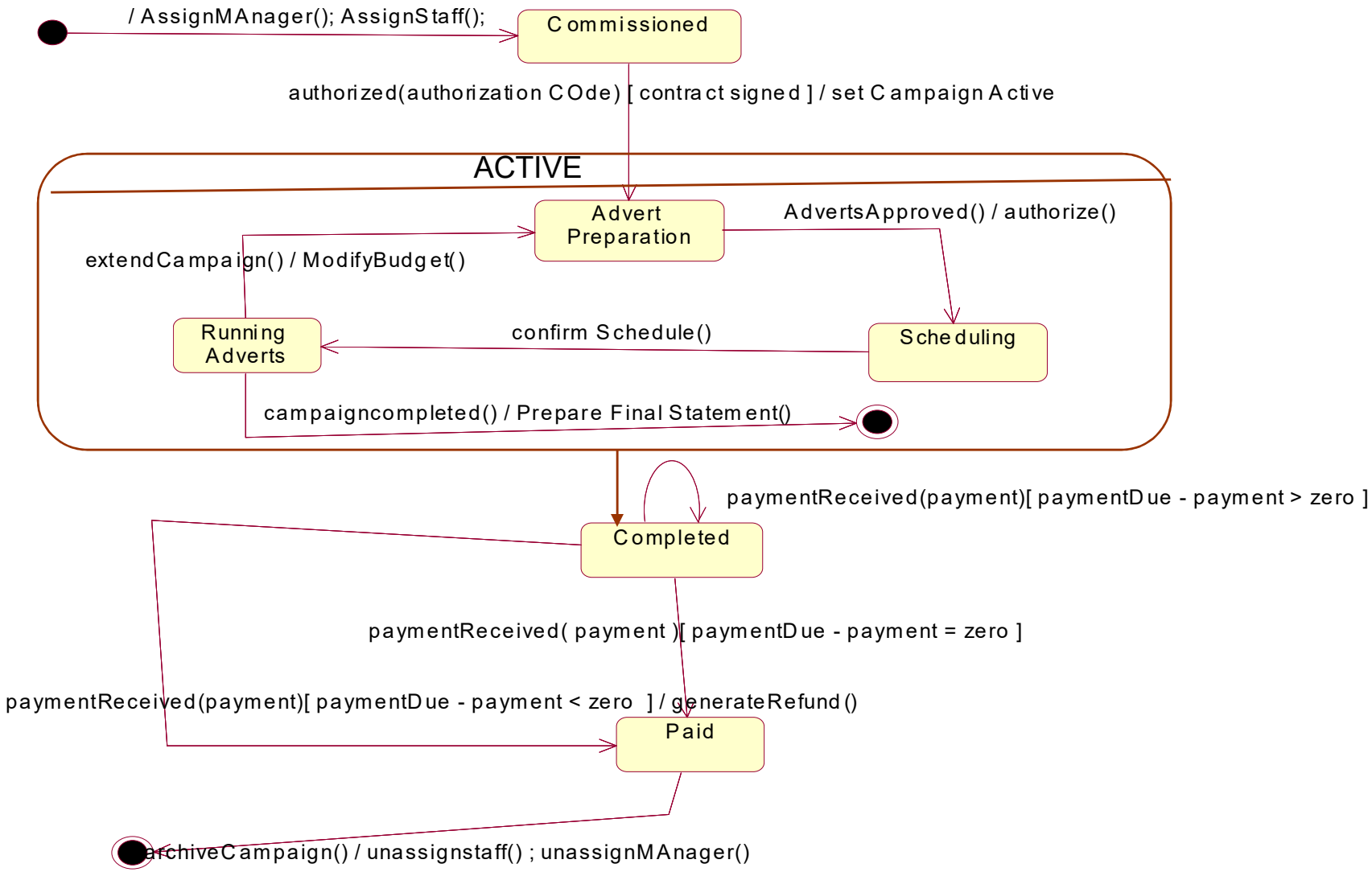
The next step is to prepare a draft state chart for the class. The following Figure shows **the** level of detail that might be shown in a first-cut statechart for the Campaign class. **This** would need to be expanded in order to reflect any events that have not been identified from the interaction diagrams, and also to include any exceptions. Complex nested states can be refined at this stage.

The following figure shows Initial statechart for the Campaign class -

using a behavioural approach.


/ AssignMAnager(); AssignStaff();
Commissioned
authorized(authorization COde) [contract signed] / set Campaign Active
Advert Preparation
AdvertsApproved() / authorize()
extendCampaign() / ModifyBudget()
Running Adverts
confirm Schedule()
Scheduling
campaigncompleted() / Prepare Final Statement()
PaymentReceived(payment)[paymentDue - payment > zero]
Completed
paymentReceived(payment)[paymentDue - payment = zero]
paymentReceived(payment)[paymentDue - payment < zero] / generateRefund()
Paid
archiveCampaign() / unassignstaff() ; unassignMAnager()

The following figure shows a revised state chart for the previous Figure results in the addition of the Active state to encompass the states Advert Preparation, Scheduling and Running Adverts



2. A life cycle approach

An alternative approach to the preparation of statecharts is based on the consideration of life cycles for objects of each class. This approach does not use interaction diagrams as an initial source of possible events and states. Instead, they are identified directly from use cases and from any other requirements documentation that happens to be available. First, the main system events are listed (at Agate *'A client commissions a new campaign'* might be one of the first to consider). Each event is then examined in order to determine which objects are likely to have a state dependent response to it.

The steps involved in the life cycle approach to state modelling are as follows:

1. Identify major system events.
2. Identify each class that is likely to have a state dependent response to these events.
3. For each of these classes produce a first-cut statechart by considering the typical life cycle of an instance of the class.
4. Examine the state chart and elaborate to encompass more detailed event behaviour.
5. Enhance the statechart to include alternative scenarios.

6. Review the statechart to ensure that is consistent with the use cases. In particular, check that the constraints that the state chart implies are appropriate.
7. Iterate through steps 4, 5 and 6 until the statechart captures the necessary level of detail.
8. Ensure consistency with class diagram and interaction diagrams and other statecharts.

Consistency Checking :

Statecharts must also be consistent with other models also. Here Consistency checks are an important task in the preparation of a complete set of models. This process highlights omissions and errors, and encourages the clarification of any ambiguity or incompleteness in the requirements.

1. Every event should appear as an incoming message for the appropriate object on an interaction diagram.

2. Every action should correspond to the execution of an operation on the appro· · priate class, and perhaps also to the despatch of a message to another object.

3. Every event should correspond to an operation on the appropriate class (but note that not all operations correspond to events).

4. Every outgoing message sent from a statechart must correspond to an operation on - another class.

12. Moving into Design

Here we need to discuss about

1.Differences between analysis and Design activities –

Analysis – It is often said to be about the What? Of a system

Design – It is described as being about the How?.

2. Difference between Logical (implementation Independent) Vs Physical Design (implementation dependent).

3. Two levels of designing –

System design – addresses architectural aspects that affects the overall system.

Detailed Design – addresses the design of classes and the detailed working of system.

4. Qualities and Objectives of Analysis and Design

5. Measurable objectives in design and planning for design

1. How is design different from Analysis?

Design has been described by Rumbaugh as 'how the system will be constructed without actually building it'. The models that are produced by design activities show how the various parts of the system will work together; the models produced by analysis activities show what is in the system and how those parts are related to one another.

The word *analysis* comes from a Greek word meaning to break down into component parts. When we analyse an organization and its need for a new system, the analysis activity is characterized as asking *what* happens in the current system and *what* is required in the new system. It is a process of seeking to understand the organization, investigating its requirements and modelling them. The result of this analysis activity is a specification of what the proposed system will do based on the requirements.

Design is about producing a solution that meets the requirements that have been analysed. The design activity is concerned with specifying *how* the new system will meet the requirements. There may be many possible design solutions, but the intention is to produce the best possible solution in the circumstances. Those circumstances may reflect constraints such as limits on how much can be spent on the new system or the need for the new system to work with an existing system.

Jacobson defines design as part of the construction process (together with implementation). The systems designer has his or her attention focused on the implementation of the new system, while the systems analyst is focused on the way the business is organized and a possible better organization.

consider a simple example, in the Agate case study. Analysis identifies the fact that each Campaign has a title attribute, and this fact is documented in the class model. Design determines how this will be entered into the system, displayed on screen and stored in some kind of database together with all the other attributes of Campaign and other classes.

Design can be seen either as a stage in the systems development life cycle or as an activity that takes place within the development of a system.

In projects that follow the waterfall life cycle model , the Analysis stage will be complete before the Design stage begins. However, in projects that follow an iterative life cycle, design is not such a clear-cut stage, but is rather an activity that will be carried out on the evolving model of the system.

In the Unified Process , design is organized as a workflow - a series of activities with inputs and outputs-that is independent of the project phase. In the Rational Unified Process , analysis and design are combined into a single workflow-the analysis activities produce an overview model, if it is required, but the emphasis is on design-and the workflow is similarly independent of the' project phase.

A project consists of major phases (Inception, Elabor-ation, Construction and Transition); each phase requires one or more iterations, and within the iterations, the amount of effort dedicated to the activities in each workflow gradually increases and then declines as the project progresses.

a) Design in the traditional life cycle

In large-scale projects that follow a traditional systems development life cycle there are a number of advantages to making a clear break between analysis and design. These are concerned with:

1. project management,
2. staff skills and experience,
3. client decisions, and
4. choice of development environment.

1. ***Project management -*** The project manager will have an overall budget in terms of money and staff time within which the system must be developed. Some proportion of those resources will have been allocated to analysis and some to design. In order to manage and control the project effectively, the project manager will want to have a clear idea of how much time is spent on each of these activities. If the two activities are allowed to merge, then the management of the project becomes more difficult, If all the time is being spent on analysis, then the project will fall behind schedule, whereas if all the time is being spent on design, then it is likely that the requirements have not been properly understood .

2. *Staff skills and experience –*

Analysis and design may be carried out by staff with different skills and experience. Staff with job titles such as business analyst and systems analyst will have the skills and expertise to carry out the an alysis, while systems architects and systems designers will have an understanding of the technology available to deliver the solution and will carry out the design.

3. *Client decisions*.

The clients will want to know what they are paying for. The end of analysis is often a decision point in a project. The clients will be provided with a specification of the system that can be traced back to their requirements and they will have to agree to this or *sign it off* before work progresses to design. In some projects, the client may be presented with a number of alternative specifications that differ in scope (what parts of the system will be computerized). In this case, the client must choose which of these alternative systems to take forward into design.

4. *Choice of development environment –*

In many projects the hardware and software that will be used to develop and deliver the finished system will not be known at the time of the analysis stage. The reasons for delaying the choice of hardware and software until the requirements for the system have been determined by the analysis, especially in the rapidly changing world of information technology in which some new technology always seems to be due for release. Because the choice of hardware, development language and database will affect the design, it may be necessary to make a break between analysis and design so that decisions about the development environment can be made.

b) Design in the iterative life cycle

There are also advantages to be gained from using an iterative life cycle such as the Unified Software Development Process.

These are concerned with:

1. risk mitigation,
2. change management,
3. team learning and
4. improved quality.

1.Risk mitigation –

An iterative process enables the identification of potential risks and problems earlier in the life of a project. The early emphasis on architecture and the fact that construction, test and deployment activities are begun early on, make it possible to identify technological problems and take action to reduce them. Integration of sub-systems is begun earlier and is less likely to throw up unpleasant surprises at the last minute.

2. Change management –

Users' requirements do change during the course of a project, often because of the time that projects take, and often because until they see some results they may not be sure what they want. In a waterfall life cycle, changing requirements are a problem, in an iterative life cycle there is an expectation that some requirements activities will still be going on late in the project, and it is easier to cope with changes. It is also possible to revise decisions about technology during the project, as the hardware and software' available to do the job will almost certainly change during the project.

3. Team learning –

Members of the team, including those concerned with testing and deploying, are involved in the project from the start, and it is easier for them to learn' about and understand the requirements and the solution from early on. They are not then suddenly presented with a new and unfamiliar system. It is also possible to identify training needs and provide the training while people are still working on an aspect of the system.

4. Improved quality –

Testing of deliverables begins early and continues throughout the project. This helps to prevent the situation where all testing is done in a final 'big bang' and there is little time to resolve the bugs that are found.

The use of object-oriented techniques helps to take advantage of an iterative life cycle. Before object-oriented approaches were developed, structured analysis and design was the dominant approach to analysis and design. In structured approaches, a clear distinction between analysis and design is made in terms of the types of diagram that are used. During analysis data flow diagrams are used to model requirements, whereas structure charts or structure diagrams are used to model the design of 'the' system and the programs in it.

One of the main advantage for the use of object-oriented approaches is that the same model (the class diagram or object model) is used right through the life of the project.

Analysis identifies classes, those classes are refined in design, and the eventual programs will be written in terms of classes. While this so-called *seamlessness* of object-oriented methods may seem like an argument for weakening the distinction between analysis and design, when we move into design different information is added to the class diagram, and other different diagrams are used to support the class diagram.

2. Logical design Vs Physical design

In the life of a system development project a decision must be made about the hardware and software that are to be used to develop and deliver the system-the hardware and software platform. In some projects this is known right from the start. Many companies have an existing investment in hardware and software, and any new project must use existing system software (such as programming languages and database management systems) and will be expected to run on the same hardware. This is more often the case in large companies with mainframe computers. In such companies the choice of configuration has been limited in the past to the use of terminals connected to the mainframe.

However, client-server architectures and open system standards, that allow for different hardware and software to operate together, have meant that even for such companies, the choice of platform is more open. For many new projects the choice of platform is relatively unconstrained, and so at some point in the life of the project a decision must be made about the platform to be used.

Some aspects of the design of systems are dependent on the choice of platform. These will affect the system architecture, the design of objects and the interfaces with various components of the system.

Examples include the following.

- The decision to create a distributed system with elements of the system running on different machines will require the use of some *middleware* such as is provided by CORBA to allow objects to communicate with one another across the network. This will affect the design of objects.

- The decision to write programs in Java and to use a relational database that supports ODBC (Object Data Base Connectivity) will require the use of JDBC (Java Data Base Connectivity) and the creation of classes to map between the objects and the relational database.

- The choice of Java as a software development language will mean that the developer has the choice of using the standard Java AWT (Abstract Windowing Toolkit), the Java Swing classes or proprietary interface classes for designing the interface.

- Java does not support multiple inheritance; other object-oriented languages such as C++ do. If the system being developed appears to require multiple inheritance then in Java this will have to be implemented using Java's interface mechanism.

- If the system has to interface with special hardware, for example bar-code scanners, then it may be necessary to design the interface so that it can be written in C as a *native method* and encapsulated in a Java class, as Java cannot directly access low-level features of hardware.

- The interaction between objects to provide the functionality of particular use cases can be designed using interaction diagrams or collaboration diagrams.

- The layout of data entry screens can be designed in terms of the fields that will be required to provide the data for the objects that are to be created or updated, and the order in which they will appear on the screen can be determined.

- The nature of commands and data to be sent to and received from special hard-ware or other systems can be determined without needing to design the exact format of messages.

Because of this, design is sometimes divided into two stages.

The first is *implemen-tation-independent* or *logical* design and the second is *implementation-dependent* or *physical* design.

Logical design is concerned with those aspects of the system that can be designed without knowledge of the implementation platform; physical design deals with those aspects of the system that are dependent on the implementation platform that will be used.

Having an implementation-independent design may be useful for if system to be re-implemented with little change to the overall design but on a different platform.

In many projects, design begins after hardware and software decisions have been made. However, if this is not the case, then the project manager must ensure that the plan of work for the project takes account of this and that logical design activities are tackled first. In an iterative project life cycle, logical design may take place in the early design iterations, or if the system is partitioned into sub-systems, the logical design of each sub-system will take place before its physical design.

3. System Design and Detailed Design

Design of systems takes place at two levels: 1.system design and

2. detailed design.

System design is concerned with the overall architecture of the system and the setting of standards, for example for the design of the human-computer interface.

Detailed design is concerned with designing individual components to fit this architecture and to conform to the standards. In an object-oriented system, the detailed design is mainly concerned with the design of objects.

1 System design -

During system design the designers make decisions that will affect the system as a whole. The most important aspect of this is the overall architecture of the system. Many modern systems use a client-server architecture in which the work of the system is divided between the clients and a server . This arises questions about how processes and objects will be distributed on different machines, and it is the role of the system designer or system architect to decide on this.

The design will have to be broken down into sub-systems and these sub-systems may be allocated to different processors. This introduces a requirement for communication between processors, and the systems designer will need to determine the mechanisms used to provide for this communication. Distributing systems over multiple processors also makes it possible for different sub-systems to be active simultaneously or concurrently.

Many organizations have existing standards for their systems. These may involve interface design issues such as screen layouts, report layouts or how on-line help is provided. Decisions about the standards to be applied across the whole system are part of system design, whereas the design of individual screens and documents is part of detailed design.

When a new system is introduced into an organization, it will have an impact on people and their existing working practices. Job design is often included in system design and addresses concerns about how people's work will change, how their interest and motivation can be maintained, and what training they will require in order to carry out their new jobs. How people use particular use cases will be included in the detailed design of the human-computer interface.

2 Detailed Design –

This addresses the design of classes and the detailed working of system. These activities we need to consider under traditional and OO approaches.

a) Traditional detailed design -

Here, detailed design was seen as consisting of four main activities:

- Designing inputs - Designing inputs meant designing the layout of menus and data entry screens;

- Designing outputs - Designing outputs concerned with the layout of enquiry screens, reports and printed documents;

- Designing processes - Designing processes dealt with the choice of algorithms and ensuring that processes correctly reflected the decisions that the software needed to make;

- Designing files - Designing files dealt with the structure of files and records, the file organization and the access methods used to update and retrieve data from the files.

The development of structured design methods made two major changes to the way in which design was carried out.

First, the work of Jackson provided a method for designers to design programs by using a technique to match the structure of the inputs and outputs with the structure of data to be read from or written to files.

Second , A criteria which is to be considered in breaking systems and programs down into modules to ensure that they are easy to develop and maintain. These criteria concern two issues: *cohesion* and *coupling.*

Criteria to maximize desirable types of cohesion have as their aim the production of modules-sections of program code in whatever language is used-that carry out a clearly defined process or a group of processes that are functionally related to one another. This means that all the elements of the module contribute to the performance of a single function. Poor cohesion is found when processes are grouped together in modules for other reasons.

Examples of poor types of cohesion include:

- When processes are grouped together for no obvious reason *(coincidental cohesion)*

- Because they handle logically similar processes such as inputs *(logical cohesion)*

- Because they happen at the same time-for example when the system initializes-*(temporal cohesion)*

- because the outputs of one process are used as inputs by the next *(sequential cohesion).*

By aiming to produce modules that are functionally cohesive, the designer should produce modules that are straightforward to develop, easy to maintain and have the maximum potential to be reused in different parts of the system. This will be assisted if coupling between modules is also reduced to the minimum.

Criteria to minimize the coupling between modules have as their aim the production of modules that are independent of one another and that can be amended without resulting in knock-on effects to other parts of the system. Good coupling is achieved if a module can perform its function using only the data that is passed to it by another module and using the minimum necessary amount of data.

Poor coupling is found in the following circumstances:

- Modules that rely on data in global variables or data in common blocks (used in languages such as COBOL and FORTRAN) that other modules may change.

- Modules that are designed to need large amounts of data to be passed to them as parameters .

- Modules that are not cohesive because they perform several functions and therefore require control information as well as data to be passed as parameters so that a decision can be made within the module about which function is required.

Modules with low coupling and high cohesion are the aim of structured design and programming techniques.

b) Object-Oriented Detailed Design

Traditionally, detailed design has been about designing inputs, outputs, processes and file or database structures; these same aspects of the system also have to be designed in an object-oriented system, but they will be organized in terms of classes.

During the analysis phase of a project, concepts in the business will have been identified and elaborated in terms of classes, and use cases will have been identified and described. The classes that have been included in the class diagram will reflect the business requirements but they will only include a very simple view of the classes to handle the interface with the user, the interface with other systems, the storage of data and the overall co-ordination of the other classes into programs. These classes will be added in design with greater or lesser degrees of detail depending on the hardware and software platform that is being used for the new system.

Different authors describe these additional aspects of the system in different ways.

According to Coad and Yourdon - calls the business classes the problem domain component and develop three further components in the design phase:

- Human interface component,
- Data management component and
- Task management component.

Coad propose a slightly different set of components:

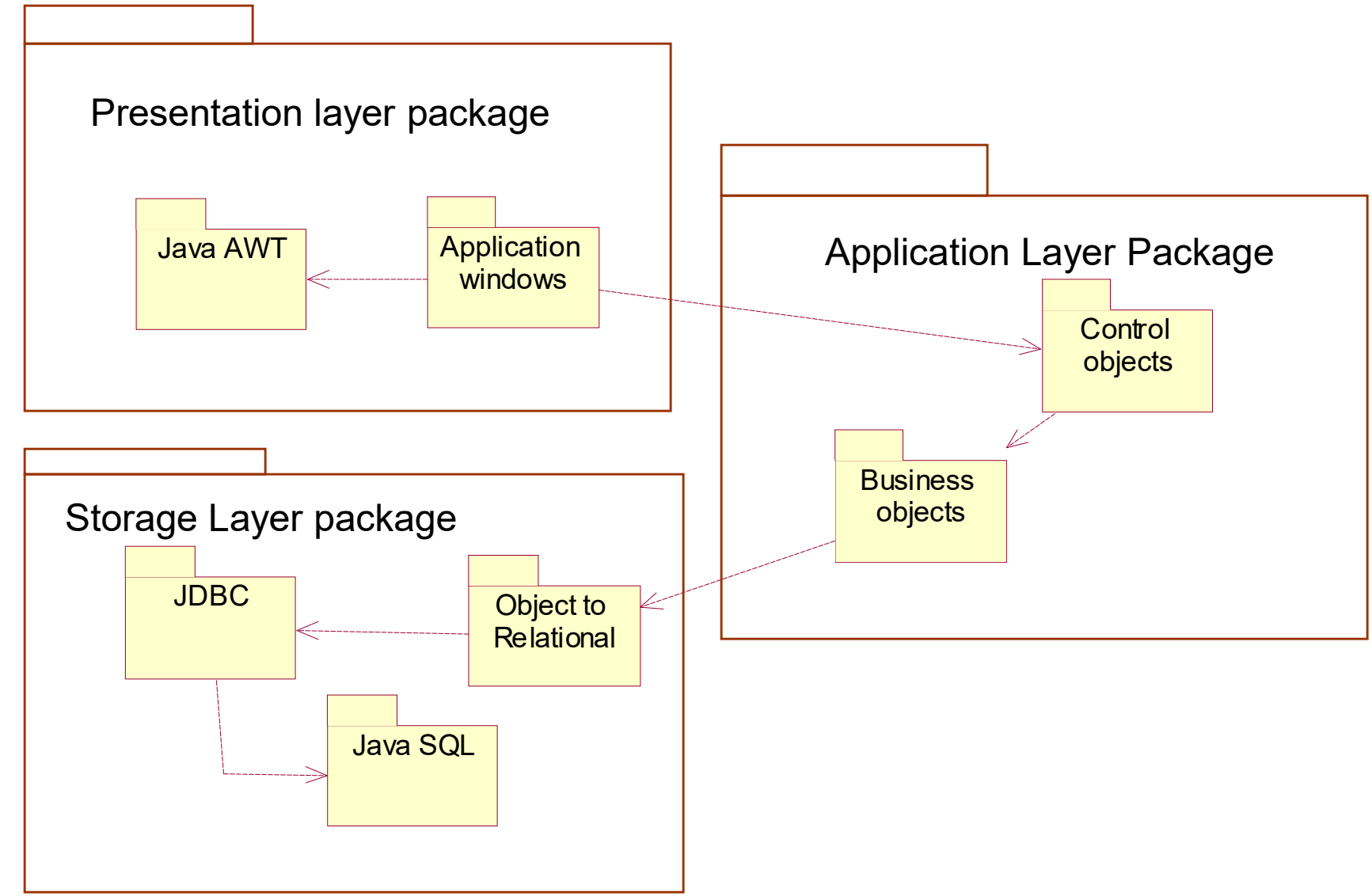
- human interface component,
- data management component and
- system interaction component

In their designs, windows play the co-ordinating role played by the classes in the task management component.

Larman proposes an architecture based on three layers:

- presentation layer (windows and reports),
- application logic layer and
- storage layer.

However, his application logic layer includes both domain concepts-equivalent to the problem domain component-and services, which may include interfaces to the data-base and to other systems. The following figure shows UML packages representing layers in the three-tier architecture.



UML packages representing layers in the three-tier architecture

Important Aspects to be considered in an Object-Oriented Detailed Design :

Certain aspects of the detailed design require special attention in the development of object-oriented systems. These include reuse and assignment of responsibilities to classes.

One of the arguments for the use of object-oriented languages is that they promote reuse through encapsulation of functionality and data together in classes and through the use of inheritance. This is not just a programming issue, but one that also affects recognition of the need to reuse analysis results in object-oriented systems development.

Design reuse already takes place at two levels: first through the use of design patterns and second by recognizing during design that business classes that have been identified during analysis may be provided by reusing classes that have already been designed within the organization, or even bought in from outside vendors.

There is a move in the software industry towards the use of *components* that provide this kind of functionality and that can be bought from vendors. CASE tool suppliers such as SELECT Software Tools have added component management software to their range of products.

The assignment of responsibilities to classes is an issue that is related to reuse. In an object-oriented system , it is important to assign responsibility for operations to the right classes, and there is often a choice.

4. Qualities and Objectives of Analysis and Design

Reusability is one of the feature which designers of object-oriented systems are trying to achieve in their design. Reusability is not the only objective of design. There are a number of other criteria for a good design. Perhaps the most obvious measure of design quality is whether the finished application is of high quality. This assumes that the analysis that preceded the design work was itself of high quality.

What makes for good analysis?

The cost of flxing faults in a system 'increases as the system progresses through the system's development life cycle. If an error occurs in the analysis of a system, it is cheaper to fix it during the analysis phase than it is later when that error may have propagated through numerous aspects of the design and implementation. It is most expensive to fix it after the system has been deployed and the error may be reflected in many different parts of the system. The quality of the design is, therefore, dependent to a large extent on the quality of the analysis.

Some methodologies have explicit quality criteria that can be applied to the products of every stage of the life cycle, but these quality criteria typically check syntactic aspects of the products, that is whether the notation is correct in diagrams, rather than semantic aspects, that is whether the diagrams correctly represent the organization's requirements.

To provide foundation for design phase, analysis should meet the following four criteria:

- correct scope,
- completeness,
- correct content and
- consistency.

Correct scope. The scope of a system determines what is included in that system and what is excluded. It is important, first that the required scope of the system is clearly understood, documented and agreed with the clients, and second that every-thing that is in the analysis models *does* fall within the scope of the system.

Coad included a ***not this time*** component with their other four components (problem domain, human interface, data management and system interaction). The not this time component is used to document classes and business services that emerge during the analysis but are not part of the requirements this time. This is a useful way of forcing consideration of the scope of *the* system.

Clearly determining the scope of a system before design starts is a strong argument for making a distinction and a break between the analysis and design phases of a project.

Completeness. Just as there is a requirement that everything that is in the analysis models is within the scope of the system, so everything that is within the scope of the system should be documented in the analysis models. Everything that is known about the system from the requirements capture should be documented and included in appropriate diagrams. Often the completeness of the analysis is dependent on the skills and experience of the analyst. Knowing what questions to ask in order to list out requirements comes with time and experience. However, analysis patterns and strategies, can help the less experienced analyst to identify likely issues.

Non-functional requirements should be documented even though they may not affect the analysis models directly. Rumbaugh suggests that some of the requirements found during analysis are not analysis requirements but design requirements. These should be documented, but the development team may only have to consider them once the design phase has begun.

Correct content*.* The analysis documentation should be correct and accurate in what it describes. This applies to textual information, diagrams and also to quantitative features of the non-functional requirements. Examples include correct descriptions of attributes and any operations that are known at this stage, correct representation of associations between classes, particularly the multiplicity of associations, and accurate information about volumes of data.

Consistency. Where the analysis documentation includes different models that refer to the same things (use cases, classes, attributes or operations) the same name should be used consistently for the same thing. Errors of consistency can result in errors being made by designers, for example, creating two attributes with different names that are used in different parts of the system but should be the same attribute. If the designers spot the inconsistency, they may try to resolve it themselves, but may get it wrong because the information they have about the system is all dependent on what they have received in the specification of requirements from the analysts.

Errors of scope or completeness will typically be reflected in the finished product not doing what the users require; the product will either include features that are not required or lack features that are. Errors of correctness and consistency will typically be reflected in the finished product performing incorrectly. Errors of completeness and consistency will most often result in difficulties for the designers; in the face of incomplete or inconsistent specifications, they will have to try to decide what is required or refer back to the analysts.

What makes for good design?

The quality of the design will clearly be reflected in the quality of the finished system that is delivered to the clients. Moreover, in the same way as the quality of analysis affects the work of designers, the quality of the design has an impact on the work of the programmers who will write the program code in order to implement the system based on the design.

Some of the criteria given below for a good design will bring benefits to the developers, while some will provide benefits for the eventual users of the system.

Objectives of Design

The designers of a system seek to achieve many objectives that have been identified as the characteristics of a good design since the early days of information systems development.

Characteristics of Good Design are efficiency, flexibility, generality, maintainability and reliability and Other characteristics of a good design includes , it should be functional, portable, secure and economical in the context of object-oriented systems, reusability is a priority objective.

Functional. When we use a computer system, we expect it to perform correctly and completely those functions that it is claimed to perform; when an information system is developed for an organization, the staff of that organization will expect it to meet their documented requirements fully and according to specification.

So, the staff of Agate will expect their system to provide them with the functionality required to document advertising campaigns, record notes about campaigns and store information about the advertisements to be used in those campaigns. If it does not perform these functions, it is not fully functional.

Efficient*.* It is not enough that a system performs the required functionality; it should also do so efficiently, in terms both of time and resources. Those resources can include disk storage, processor time and network capacity. This is why design is not just about producing any solution, but about producing the best solution.

Economical , Linked to efficiency is the idea that a design should be economical. This applies not only to the fixed costs of the hardware and software that will be required to run it, but also to the running costs of the system. The cost of memory and disk storage is very low compared to 20 years ago, and most small businesses using Microsoft Windows probably now require more disk space for their programs than they do for their data.

Reliable. The system must be reliable in two ways: first, it should not be prone to either hardware or software failure; second it should reliably maintain the integrity of the data in the system. Hardware reliability can be paid for : some manufacturers provide systems with redundant components that run in parallel or that step in when an equivalent component fails; RAID (redundant arrays of inexpensive disks) technology can provide users with disk storage that is capable of recovering from failure of one drive in an array. The designers must design software reliability into the system. In physical design, detailed knowledge of the development environment is likely to help ensure reliability .

Secure. Systems should be designed to be secure against malicious attack by out-siders and against unauthorized use by insiders. System design should include considerations of how people are authorized to use the system and policies on passwords.

Flexible. It is the ability to adapt changing business requirements as time passes.

Generality. It Describes the extent to which a is general purpose.Means it includes the feature of portability.

Manageable. A good design should allow the project manager to estimate the amount of work involved in implementing the various sub-systems.If implementation of these sub-systems is completed then forwarded for testing will not have any affects on other parts of the system.

Maintainable. Maintenance activities include fixing bugs, modifying reports and screen layouts, enhancing programs to deal with new business requirements, migrating systems to new hardware and fixing the new bugs that are introduced by all of the above. A well-designed and documented system is easier to maintain than one that is poorly designed and documented. If maintenance is easy then it is with less cost.

Usable. Usability includes a range of aspects including the idea, mentioned above, that a system should be both satisfying and productive.

Reusable. Reusability is the important feature of object-oriented development. Many of the features of object-oriented systems are geared to improve the possibility of reuse. Reuse affects the designer in three ways.

First, he or she will consider how economies can be made by designing reuse into the system through the use of inheritance;

second, he or she will look for opportunities to use design patterns, which provide templates for the design of reusable elements;

third, he or she will seek to reuse existing classes either directly or by sub-classing them.

Constraints on Design

Constraitns arise from the context of the project as well as from the users' requirements. The clients' budget for the project, the timescale within which they expect the system to be delivered, the skills of staff working on the project, the need to integrate the new system with existing hardware or systems, and standards set as part of the overall systems design process can all constrain what can be achieved. Resolving conflicts between requirements and constraints results in the need for compromises in design.

The following is one of the example which illustrates how these can occur.

If the users of Agate's new system require the ability to change fonts in the notes that they write about campaigns and adverts, then they will want to be able to edit notes with the same kind of functionality that would be found in a word processor. As pointed out above, this will seriously impact the storage requirements for notes. It will also have an effect on network traffic, as larger volumes of data will need to be transferred across the network when users browse through the notes. The designers will have to consider the impact of this requirement. It may be that the users will have to accept reduced functionality or the management of Agate will have to recognize that their system will have higher costs for storage than first envisaged.

Compromise solutions may involve only transferring the text of a note (without the overhead of all the formatting information) when users are browsing a note and transferring the full file only when it needs to be viewed or edited. However, this will increase the processing load on the server. Another compromise solution might be to use a different file format such as RIF (rich text format) rather than the word-processor format.

Measurable objectives in Design

Some objectives are specific to a particular project, and it is important to be able to assess whether these objectives have been achieved or not. One way of doing this is to ensure that these objectives are expressed in measurable terms so that they can be tested by simulation during the design phase, in prototypes that are built for this purpose or in the final system.
Measurable objectives often represent the requirements that we referred to as non-functional requirements. They also reflect the fact that information systems are not built for their own sake, but are developed to meet the business needs of some organization. The system should contribute to the strategic aims of the business, and so should help to achieve aims such as:

- provide better response to customers, or
- increase market share.

Planning for Design

In a waterfall life cycle project, the transition from the analysis phase to the design phase also gives the project manager the opportunity to plan for the activities that must be undertaken during design.

1. If the hardware and software platform has not been decided upon, then design will begin as logical design, but the project manager must plan for the time when the platform is known and physical design can begin.

2. The system architecture must be agreed and system standards must be set that will affect the design of individual sub-systems.

3. If the designers are not familiar with all aspects of the platform, time must be allowed for training, or additional staff must be brought in, perhaps contractors with expertise in particular aspects of the hardware or software.

4. Design objectives must be set and procedures put in place for testing those that can be tested by simulation during the design process.

5. Procedures must also be put in place for resolving conflicts between constraints and requirements and for documenting and agreeing any trade-offs that are made.