13.SYSTEM DESIGN

1. The major elements of system Design

The system design activity specifies the context within which detailed design will occur. A major part of system design is defining the *system architecture.* The meaning and scope of the term architecture for computerized information systems is much debated but it is generally accepted that it is an important feature of the delivered system.

The architecture of a system is concerned with its overall structure, the relationships among its major components and their interactions. If the system being considered contains human, software and hardware elements then its architecture includes how these elements are structured and how they interact.

On the other hand, if the system being considered comprises software and hardware, then its architecture only concerns these elements. It is important to consider the structure of the software elements of the system and this is termed the *software architecture.* The hardware architecture of a system describes computers and peripherals required for the system and how software is allocated to them.

The architecture of the information system is first considered early in the project during the requirements capture and analysis activities. This first view of the system architecture is driven significantly by the use cases and then informs the continuing requirements capture and analysis activities. This forms a useful basis from which to develop the design architecture.

The detailed software architecture of a computerized information system develops as the design process continues into object design but it is important to identify an overall system architecture within which the detail can be refined. High-level architectural decisions that are made during system design deter-mine how successfully the system will meet its non-functional objectives (e.g. perfor-mance, extensibility) and thus its long-term utility for the client.

Reuse is one of the much wanted benefits of object-orientation and poor software architecture usually reduces both the reusability of the components produced and the opportunity to reuse existing components.

During system design we need to consider the following activities.

- ❖ Sub-systems and major components are identified.
- ❖ Any inherent concurrency is identified.
- ❖ Sub-systems are allocated to processors.
- ❖ A data management strategy is selected.
- ❖ A strategy and standards for human-computer interaction are chosen.
- ❖ Code development standards are specified.
- ❖ The control aspects of the application are planned.
- ❖ Test plans are produced.
- ❖ Priorities are set for design trade-offs.
- ❖ Implementation requirements are identified (for example, data conversion).

2. Software Architecture

Software architecture, like system architecture has no generally agreed definition and can be interpreted differently depending upon the context. According to Booch , the architecture of a system is its class structure. He suggests that part of the architecture is the way in which classes are grouped together. Rumbaugh use the term system architecture to describe the overall organization of a system into sub-systems.

A software architecture is a description of the sub-systems and components of a software system and the relationships between them. Subsystems and components are typically specified in different views to show the relevant functional and non-functional properties of a software system. Software architecture is the result of the software design activity.

According to Soni , identified the following four different aspects of software architecture as shown below.

Type of architecture	Examples of elements	Examples of relationships
Conceptual	Components	Connectors
Module	Sub-systems, modules	Exports, imports
Code	Files, Directories, libraries	Includes, contains
Execution	Tasks, threads, object interactions	Uses, calls

The ***conceptual architecture*** is concerned with the structure of the static class model and the connections between the components of the model. The ***module architecture*** describes the way the system is divided into sub-systems or modules and how they communicate by exporting and importing data. The ***code architecture*** defines how the program code is organized into files and directories and grouped into libraries. The ***execution architecture*** focuses on the dynamic aspects of the system and the communication between components as tasks and operations execute.

There are alternative ways of giving different aspects of architecture. For example, a logical architecture might comprise the class model, while a physical architecture is concerned with mapping the software onto the hardware components. However, in all cases these different aspects combine to define a software architecture for the system. There are various architectural styles, each of which has characteristics that make it more or less suitable for certain types of application.

Sub-systems

A sub-system typically groups together elements of the system that share some common properties. An object-oriented sub-system encapsulates a coherent set of responsibilities in order to ensure that it has integrity and can be maintained. For example, the elements of one sub-system deals about human-computer interface, the elements of another may deals about data management and the elements of a third may all focus on a particular functional requirement.

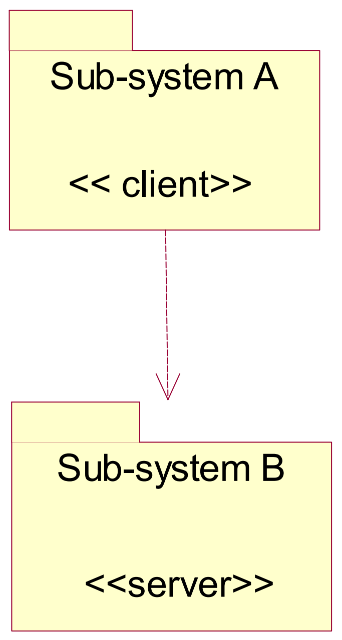
The division of an information system into sub-systems has the following advantages.

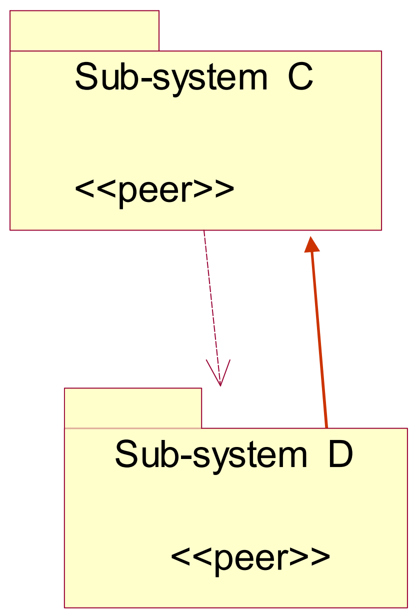
- It produces smaller units of development.
- It helps to maximize reuse at the component level.
- It helps the developers to cope with complexity.
- It improves maintainability.
- It aids portability.

Each sub-system should have a clearly specified boundary and fully defined interfaces with other sub-systems. A specification for the interface of a sub-system defines the precise nature of the sub-system's interaction with the rest of the system but does not describe its internal structure .
A sub-system can be designed and constructed independently of other sub-systems, simplifying the development process. Sub-systems may correspond to increments of development that can be delivered individually as part of an incremental life cycle also.

Dividing a system into sub-systems is an effective strategy for handling complexity. Sometimes it is only feasible to model a large complex system piece by piece, with the sub-division forced on the developers by the nature of the application. Splitting a system into sub-systems can also aid reuse as each sub-system may correspond to a component that is suitable for reuse in other applications.

Each sub-system provides services for other sub-systems, and there are two different styles of communication that make this possible. These are known as *client-server* and *peer-to-peer* communication and are shown below.





The server sub-system does not depend on the client sub-system and is not affected by changes to the client's interface.

Each peer sub-system depends on the other and each is affected by changes in the other's interface

Client-server communication requires the client to know the interface of the server sub-system, but the communication is only in one direction. The client sub-system requests services from the server sub-system and not vice versa. Peer-to-peer commun-ication requires each sub-system to know the interface of the other, thus coupling them more tightly. The communication is two way since either peer sub-system may request services from the other.

In general client-server communication is simpler to implement and to maintain, as the sub-systems are less tightly coupled than they are when peer-to-peer communi-cation is used. In Figure the sub-systems are represented using packages that have been stereotyped to indicate their role. Component and deployment diagrams can also be used to model the implementation of sub-systems .

Layering and partitioning

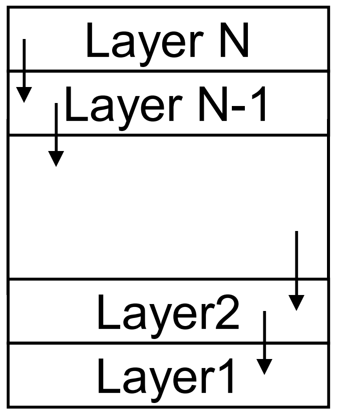
There are two general approaches to the division of a software system into sub-systems. These are known as

1. *Layering - so* called because the different sub-systems usually represent different levels of abstraction
2. *Partitioning* - which usually means that each sub-system focuses on a different aspect of the functionality of the system as a whole. In practice both approaches are often used together on one system, so that some of its sub-systems are divided by layering, while others are divided by partitioning.

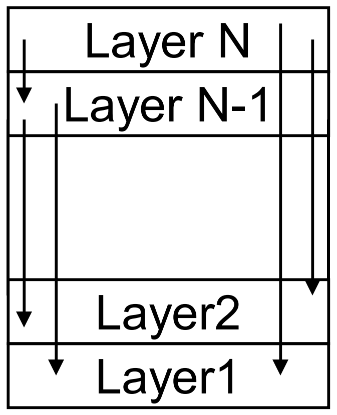
1. Layered sub-systems

Layered architectures are used to specify high-level structures for a system. A schematic of the general structure is shown below.

Schematic of a layered architecture



Closed architecture - messages may only be sent to the adjacent lower layer.



Open architecture - messages can be sent to any lower layer

Each layer corresponds to one or more sub-systems, which may be differentiated from each other by differing levels of abstraction or by a different focus of their functionality.

In this architecture , the top layer uses services provided by the layer immediately below it. This in turn may require the services of the next layer down. Layered architectures can be either open or closed, and each has its particular advantages.

In a closed layered architecture a certain layer (say layer N) can only use the services of the layer immediately below it (layer N - 1). In an open layered architecture layer N may directly use the services of any of the layers that lie below it.

A closed architecture minimizes dependencies between the layers and reduces the impact of a change to the interface of anyone layer. An open layered architecture produces more compact code since the services of all lower level layers can be accessed directly by any layer above them without the need for extra program code to pass messages through each intervening layer. However this breaks the encapsulation of the layers, increases the dependencies between layers and increases the difficulty caused when a layer needs to be changed. \

Networking protocols provide some of the best known examples of layered architectures. Eg: The OSI (Open Systems Interconnection) 7 Layer Model was defined by the International Standard-ization Organization (ISO) as a standard architectural model for network protocols

Buschmann suggests that a series of issues need to be addressed when applying a layered architecture for an application. These includes:

- Maintaining the stability of the interfaces of each layer;
- The construction of other systems using some of the lower layers;
- Variations in the appropriate level of granularity for sub-systems;
- The further sub-division of complex layers;
- Performance reductions due to a closed layered architecture.

The following steps are adapted from Buschmann , and provide an outline process for the development of a layered architecture for an application. These steps offers guidelines on the issues that need to be addressed during the development of a layered architecture.

1. Define the criteria by which the application will be grouped into layers - A commonly used criteria is level of abstraction from the hardware. The lowest layer provides primitive services for direct access to the hardware while the layers above provide more complex services that are based upon these primitives. Higher layers in the architecture carry out tasks that are more complex and correspond to concepts that occur in the application domain .

2. Determine the number of layers - Too many layers will introduce unnecessary overheads while too few will result in a poor structure.

3. Name the layers and assign functionality to them - The top layer should be concerned with the main system functions as perceived by the user. The layers below should provide services and infrastructure that enable the delivery of the functional requirements.

4. Specify the services for each layer. In general it is better in the lower layers to have a small number of low-level services that are used by a larger number of services in higher layers.

5. Refine the layering by iterating through steps 1 to 4.

6. Specify interfaces for each layer.

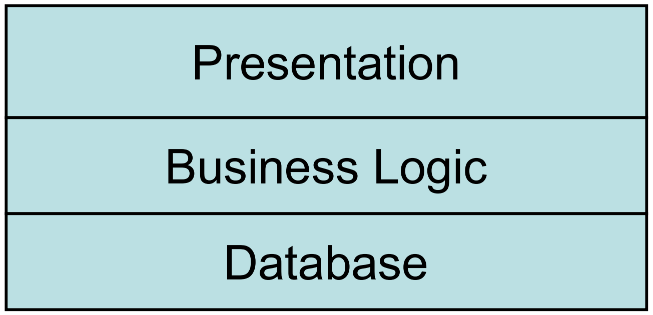
7. Specify the structure of each layer. This may involve partitioning within the layer

8. Specify the communication between adjacent layers.

9. Reduce the coupling between adjacent layers - This effectively means that each layer should be strongly encapsulated. Where a client-server communication protocol will be used, each layer should have knowledge only of the layer immediately below it.

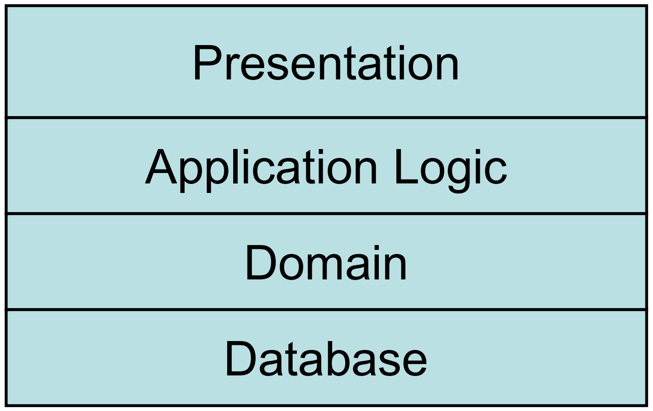
One of the simplest application architectures has only two layers- the application layer and a database layer. Tight coupling between the user interface and the data representation would make it more difficult to modify either independently, so a middle layer is often introduced in order to separate the conceptual structure of the problem domain.

This gives the architecture shown in the following Figure , which is commonly used for business-oriented information systems. It is same as the three-tier architecture (user interface, business objects and database tiers).



Three layer architecture

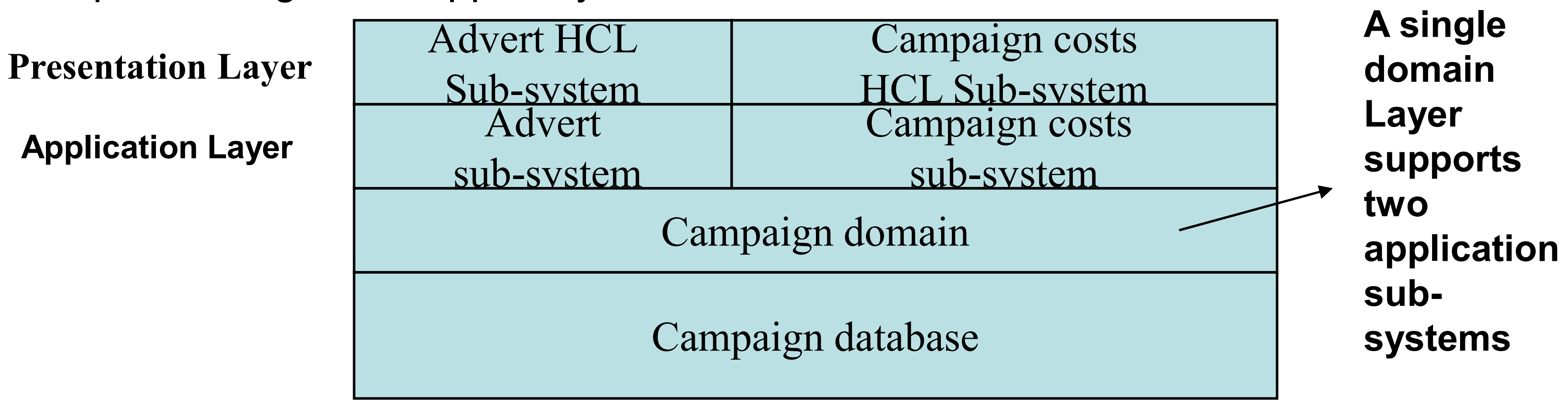
The following figure, four layer architecture separates the business logic layer into application logic and domain layers.



The approach which is used during the analysis activity of use case realization results in the identification of boundary, control and entity classes and it is easy to map the boundary classes onto a presentation layer, the control classes onto an application logic layer and the entity classes on a domain layer. Thus from an early stage in the development of an information system some of element of layering is being introduced into the software architecture. However, it is important to appreciate that as we move through design, the allocation of responsibility amongst these types of class may be adjusted to accommodate non-functional requirements . A good design solution is one that balances competing requirements effectively.

2. Partitioned sub-systems

In the design phase , some layers within a layered architecture may have to be decomposed because of their complexity. The following figure shows a four layer architecture for part of Agate's campaign management system that also has some partitioning in the upper layers.


Presentation Layer
Application Layer
Advert HCL Sub-system
Campaign costs HCL Sub-system
Advert sub-system
Campaign costs sub-system
Campaign domain
Campaign database
A single domain Layer supports two application sub-systems

In this, the application layer corresponds to the analysis class model for a single application, and is partitioned into a series of sub-systems. These sub-systems are loosely coupled and each should deliver a single service or coherent group of services.

The Campaign Database layer provides access to a database that contains all the details of the campaigns, their adverts and the campaign teams.

The Campaign Domain layer uses the lower layer to retrieve and store data in the database and provides common domain functionality for the layers above. For example, the Advert sub-system might support individual advert costing while the Campaign Costs sub-system uses some of the same common domain functionality when costing a complete campaign. Each application sub-system has its own presentation layer to cater for the differing interface needs of different user roles.

A system may be split into sub-systems during analysis because of the system's size and complexity. However, the analysis sub-systems should be reviewed during design for coherence and compatibility with the overall system architecture. The sub-systems that result from partitioning should have clearly defined boundaries and well specified interfaces, thus providing high levels of encapsulation so that the implementation of an individual sub-system may be varied without causing dependent changes in the other sub-systems.

Model-View-Controller Architecture

Many interactive systems uses the Model-View-Controller (MVC) architecture. This structure is capable of supporting user requirements that are presented through differing interface styles, and it aids maintainability and portability.

The MVC architecture separates an application into three major types of component:

1.Models that comprise the main functionality,

2. Views that present the user interface,

3. Controllers that manage the updates to views.

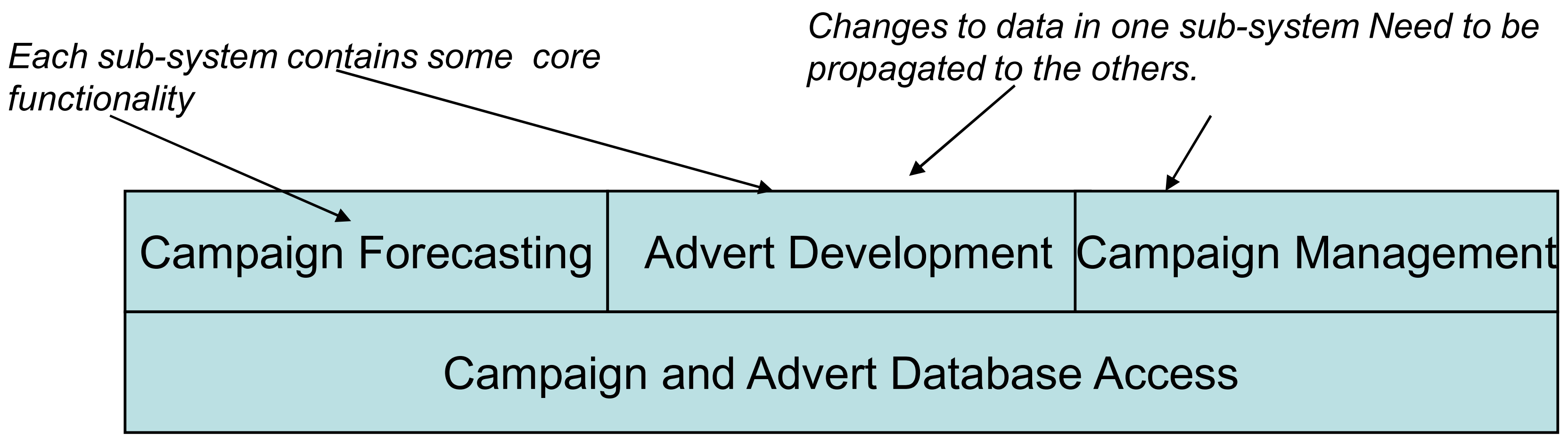
It is common for the view of an information system that is required for each user to differ according to their role. This means that the data and functionality available to any user should be customized to his or her needs. The needs of different types of users can also change at varying rates. For these reasons provide access to each user separately, only to the relevant part of the functionality of the system as a whole.

For example, in the Agate case study many users need access to information about campaigns, but their perspectives vary.

- The campaign manager needs to know about the current progress of a campaign and concerned with the current state of each advertisement and how this impacts on the campaign as a whole-is it prepared and ready to run, or is it still in the preparation stage? If an advert is behind schedule does this affect other aspects of the campaign?
- The creative artist also needs access to adverts but he is likely to need access to the contents of the advert as well as some scheduling information.
- A director may wish to know about the state of all live campaigns and their projected income over the next six months.

This gives at least three different perspectives on campaigns and adverts, each of which might use different styles of display. The director may require charts and graphs that summarize the current position at quite a high level. The campaign manager may require lower level summaries that are both textual and graphical in form. The graphic designer may require detailed textual displays of notes with a capability to display graphical images of an adverts content. Ideally, if any information about a campaign or an advert is updated in one view then the changes should also be reflected immediately in all other views.

The following figure shows a possible architecture with multiple interfaces for the same core functionality, but some problems remain.

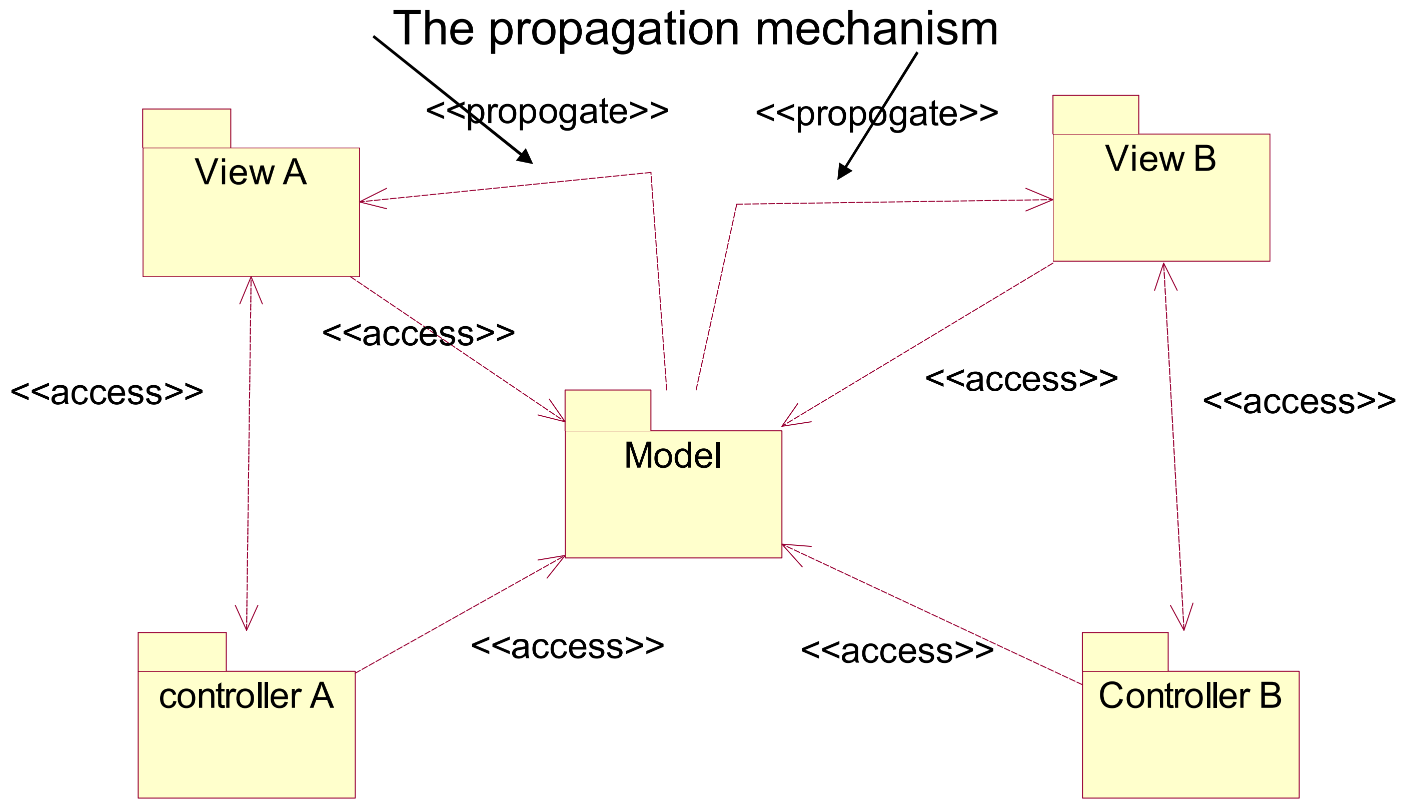

Each sub-system contains some core functionality
Changes to data in one sub-system Need to be propagated to the others.
Campaign Forecasting
Advert Development
Campaign Management
Campaign and Advert Database Access

The following are some of the difficulties that need to be resolved for this type of application.

- The same information should be capable of presentation in different formats in different windows.
- Changes made within one view should be reflected immediately in the other views.
- Changes in the user interface should be easy to make.
- Core functionality should be independent of the interface to enable multiple interface styles to co-exist.

The MVC architecture solves this type of problems through its separation of core functionality (model) from the interface and through its incorporation of a mechanism for propagating updates to other views. The interface itself is split into two elements: the output presentation (view) and the input controller (controller).

The following figure shows the basic general structure of the Model – View Controller.



The responsibilities of the components of an MVC architecture are …

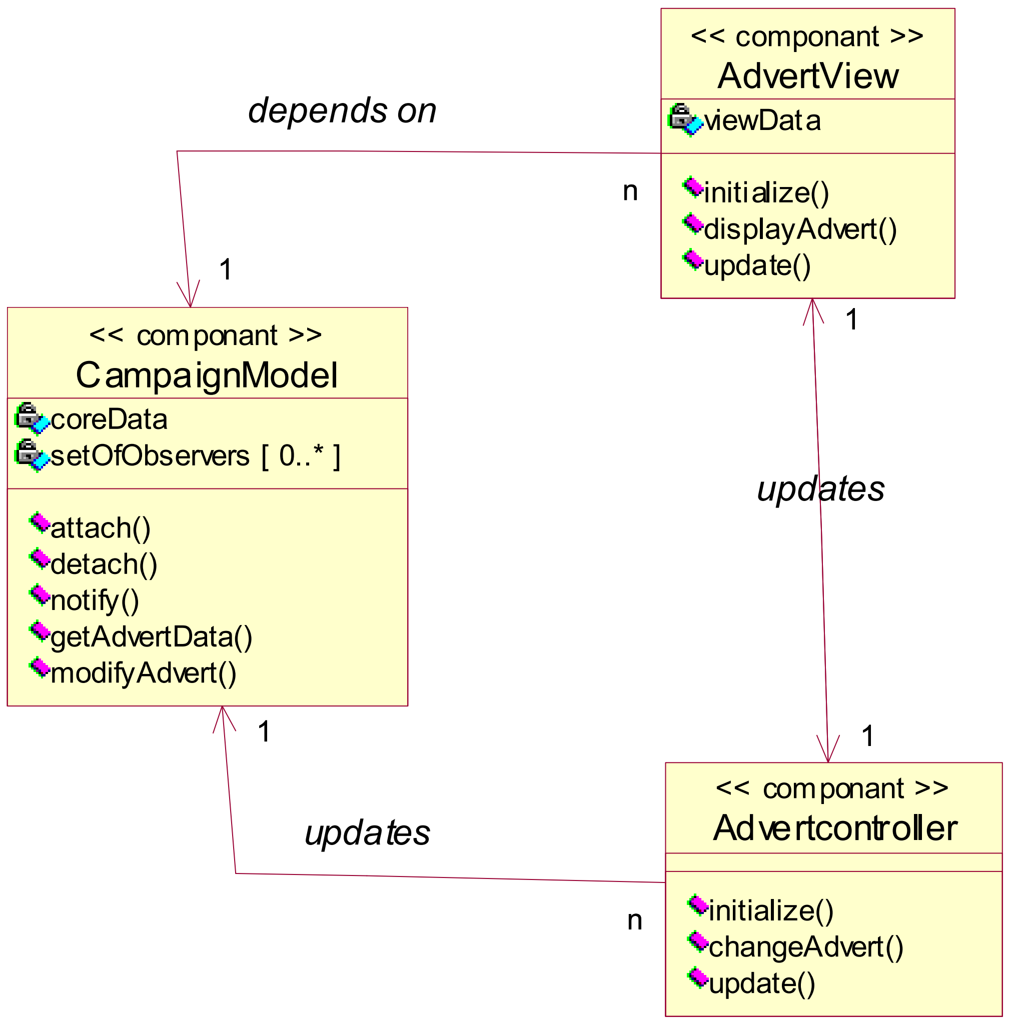
Model - The model provides the central functionality of the application and is aware of each of its dependent view and controller components .

View - Each view corresponds to a particular style and format of presentation of information to the user. The view retrieves data from the model and updates its presentations when data has been changed in one of the other views. The view creates its associated controller.

Controller - The controller accepts user input in the form of events that trigger the execution of operations within the model. These may cause changes to the information and in turn trigger updates in all the views ensuring that they are all up to date.

Propagation Mechanism - This enables the model to inform each view that the model data has changed and as a result the view must update itself. It is also often called the dependency mechanism.

The following figure represents the capabilities offered by the different MVC components as they might be applied to part of the campaign management system at Agate.

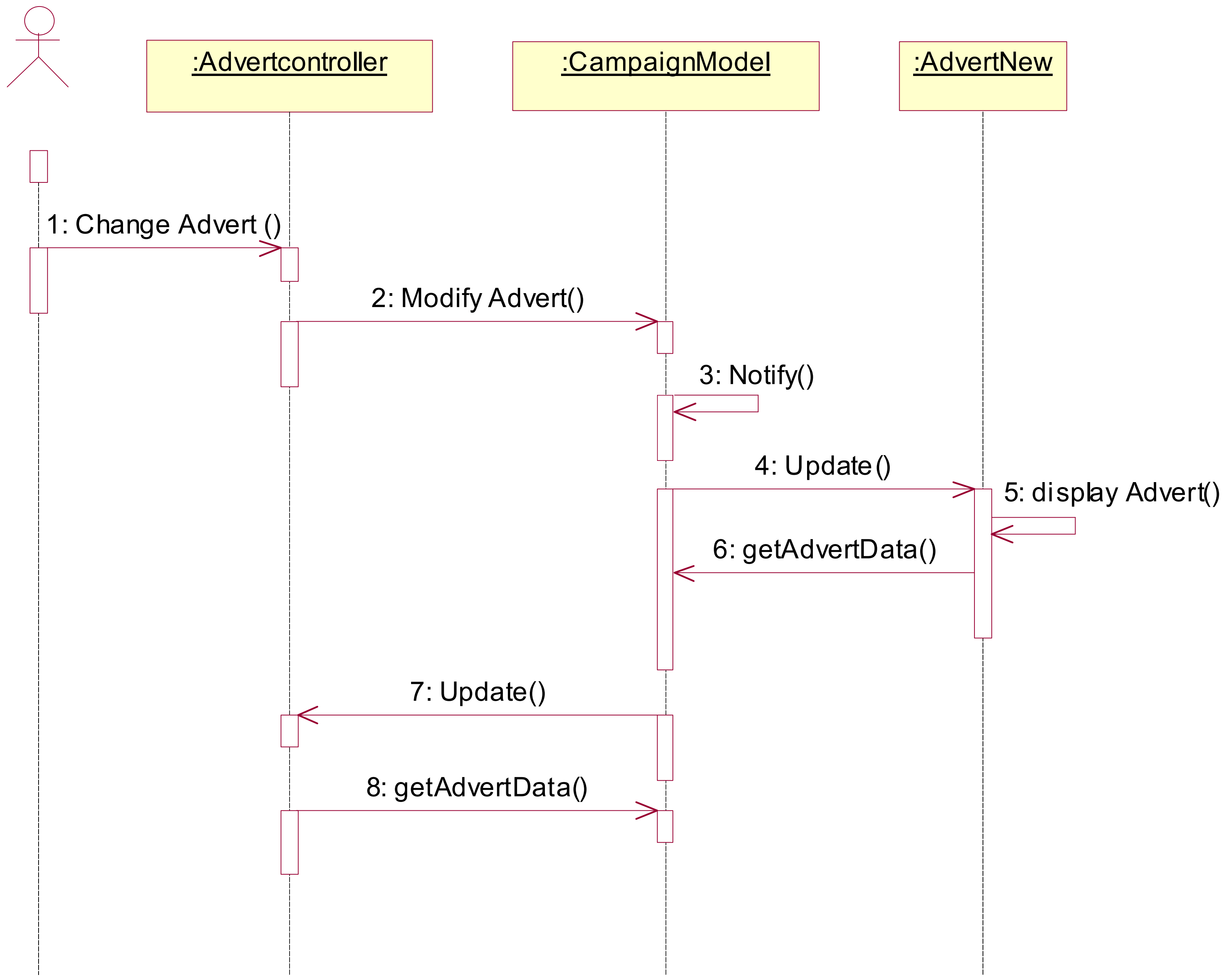


The operation update () in the AdvertView and AdvertController components triggers these components to request data from the CampaignModel component. This model component has no knowledge of the way that each view and controller component will use its services. It need only know that all view and controller components must be informed whenever there is a change of state.

The attach () and detach () services in the CampaignModel component enable views and controllers to be added to the setOfObservers. This contains a list of all components that must be informed of any change to the model core data. Generally there would be separate views, each with its own controller, to support the requirements of the campaign manager and the director.

The following interaction sequence diagram gives the communication that is involved in the operation of an MVC architecture.

MVC component interaction



An AdvertController component receives the interface event changeAdvert ( ). In response to this event the controller invokes the modifyAdvert () operation in the CampaignModel object. The execution of this operation causes a change to the model.

For example, if the target completion date for an advertisement is altered. This change of state must now be propagated to all controllers and views that are currently registered with the model as active. For this, the modifyAdvert () operation invokes the notify () operation in the model that sends an update() message to the view. The view responds to the update () message by executing the displayAdvert () operation which requests the appropriate data from the model via the getAdvertData () operation. The model also sends an update () message to the AdvertController, which then requests the data it needs from the model.

One of the important aspects of the MVC architecture is that each model knows only which views and controllers are registered with it, but not what they do. The notify () operation causes an update message to all the views and controllers.

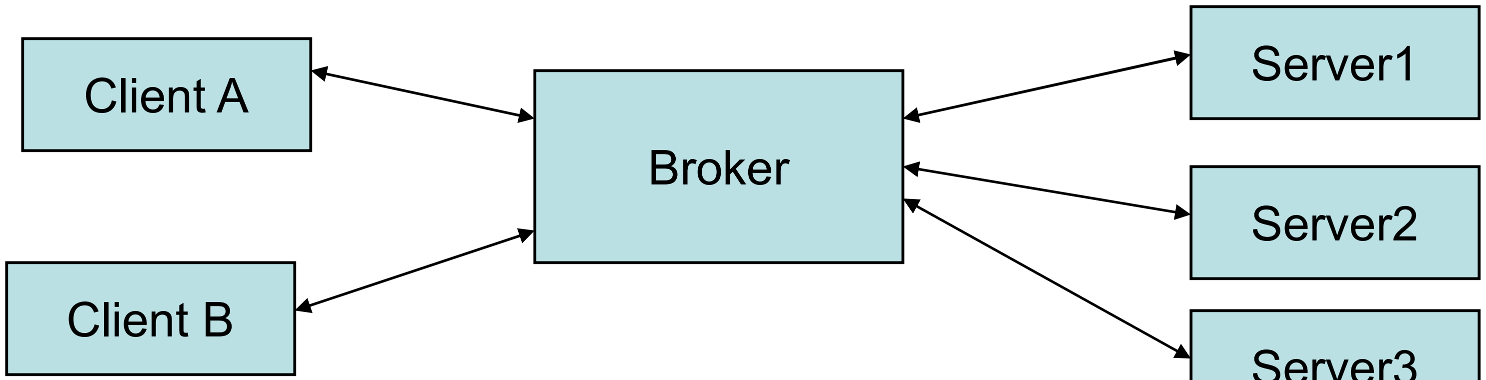
The change propagation mechanism can be structured so that further views and controllers can be added without causing a change to the model. Other kinds of communication may take place between the MVC components during the operation of the application. The controller may receive events from the interface that require a change in the way that some data is presented to the user but do not cause a change of state. The controller's response to such an event would be to send an appropriate message to the view. There would be no need for any communication with the model.

3. Architectures for distributed systems

Distributed information systems are becoming more common as communications technology improves and becomes more reliable. An information system may be distributed over computers at the same location or different locations. Since Agate has offices around the world, it may need information systems that use data that is distributed among different locations. If Agate grows, it may also open new offices and require new features from its information systems.

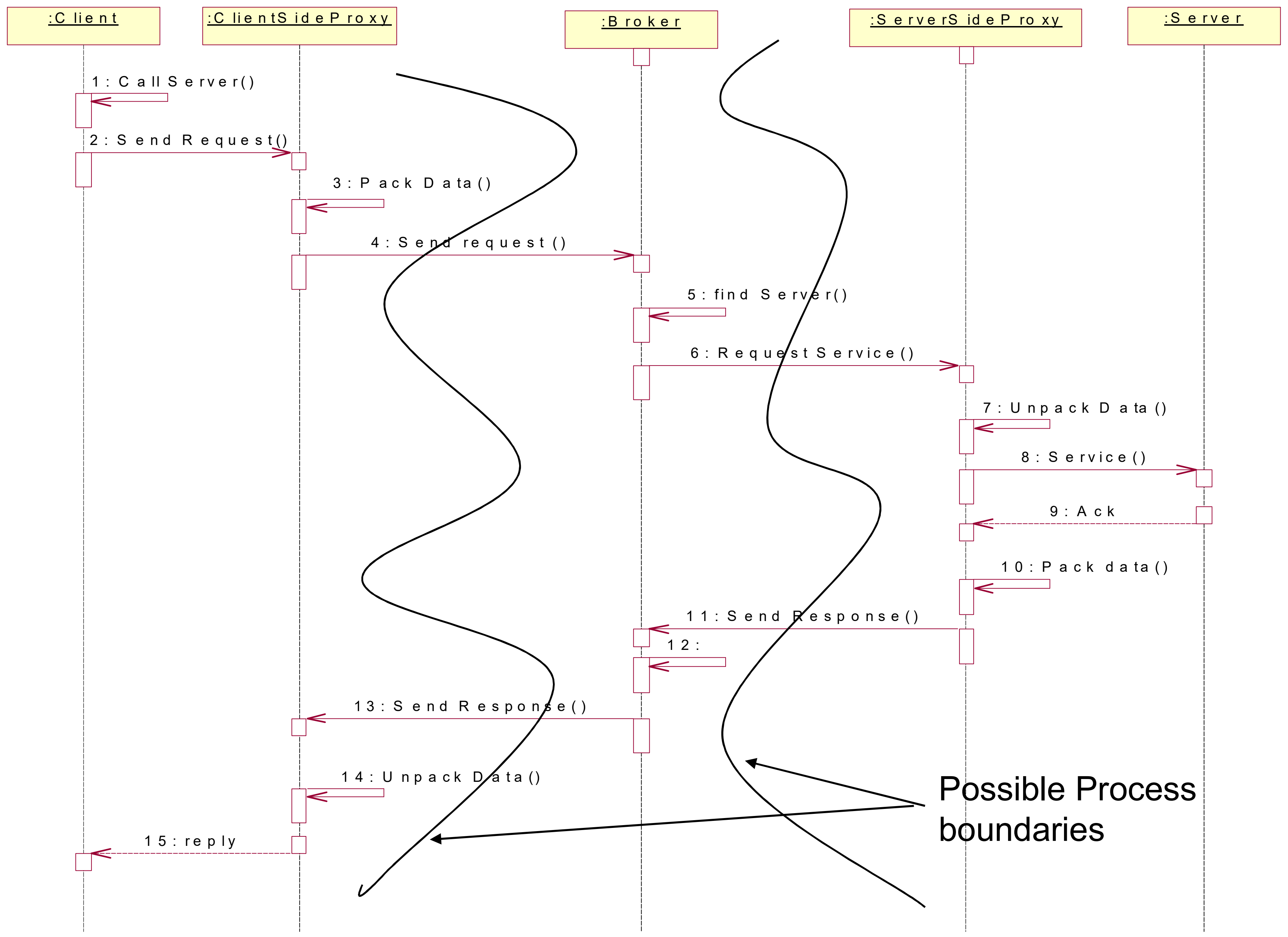
An architecture that is suitable for distributed information systems needs also to be flexible so that it can cope with change. A distributed information system may be supported by software products such as distributed database management systems or object request brokers .

A simplified version of the broker architecture is shown in the following figure.



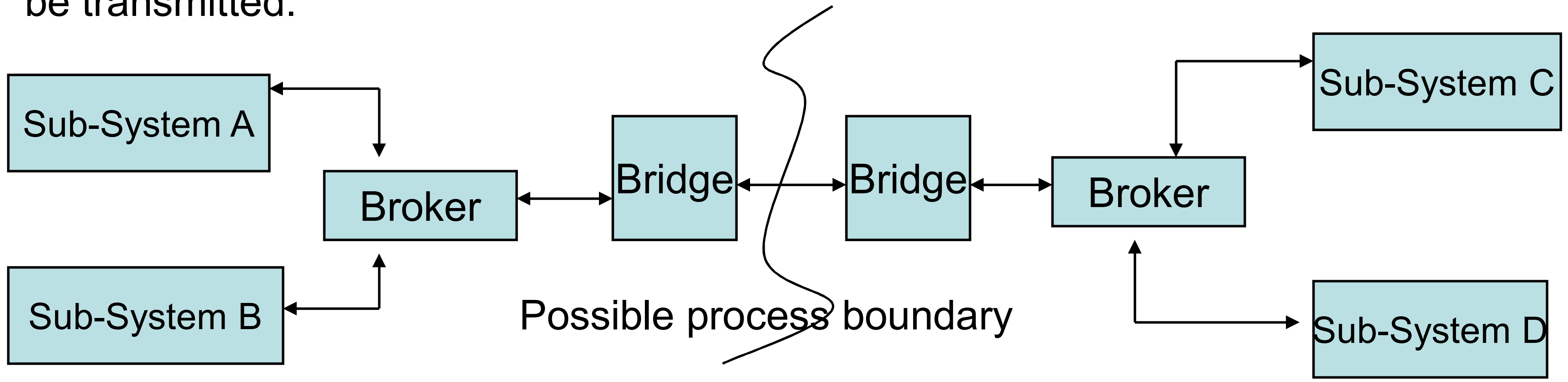
A broker component increases the flexibility of the system by decoupling the client and server components. Each client sends its requests to the broker rather than communicating directly with the server component. The broker then forwards the service request to an appropriate server. A broker may offer the services of many servers and part of its task is to identify the relevant server to which a service request should be forwarded. The advantage offered by a broker architecture is that a client need not know where the server is located, and it may therefore be stored on either a local or a remote computer. Only the broker needs to know the location of the servers that it handles.

The following figure shows a sequence diagram for client-server communication using the broker architecture. Here the server sub-system is on a local computer. In addition to the broker itself, two additional *proxy* components have been introduced to insulate the client and server from direct access with the broker. On the client side a ClientSide Proxy receives the initial request from the client and packs the data in a format suitable for transmission. The request is then forwarded to the Broker which finds an appropriate server and invokes the required service via the ServerSideProxy.


:Client
:ClientSideProxy
:Broker
:ServerSideProxy
:Server
1 : Call Server()
2 : Send Request()
3 : Pack Data ()
4 : Send request ()
5 : find Server()
6 : Request Service ()
7 : Unpack Data ()
8 : Service ()
9 : Ack
10 : Pack data ()
11 : Send Response ()
12 :
13 : Send Response ()
14 : Unpack Data ()
15 : reply
Possible Process boundaries

The ServerSideProxy then unpacks the data and issues the service request sending the service () message to the Server object. The service () operation then executes and on completion control returns to the ServerSideProxy. The response is then sent to the Broker which forwards it to the originating ClientSideProxy. Note that these are both new messages and not returns. The reason for this is that a broker does not wait for each response before handling another request. Once its sendRequest - activation has been completed the broker will in all probability deal with many other requests, and thus requires a new message from the ServerSideProxy object that causes it to enter a new activation. Unlike the broker, the ClientSideProxy has remained active; this then unpacks the message and the response becomes available to the client as control returns.

The following figure shows a schematic broker architecture that uses *bridge* components to communicate between two remote processors. Each bridge converts service requests into a network specific protocol so that the message can be transmitted.



4. CONCUREENCY

In most systems there are many objects that do not need to operate concurrently but some may need to be active simultaneously. Object-oriented modelling captures any inherent concurrency in the application through the development of interaction diagrams and statecharts and the examination of use cases also helps with the identification of concurrency.

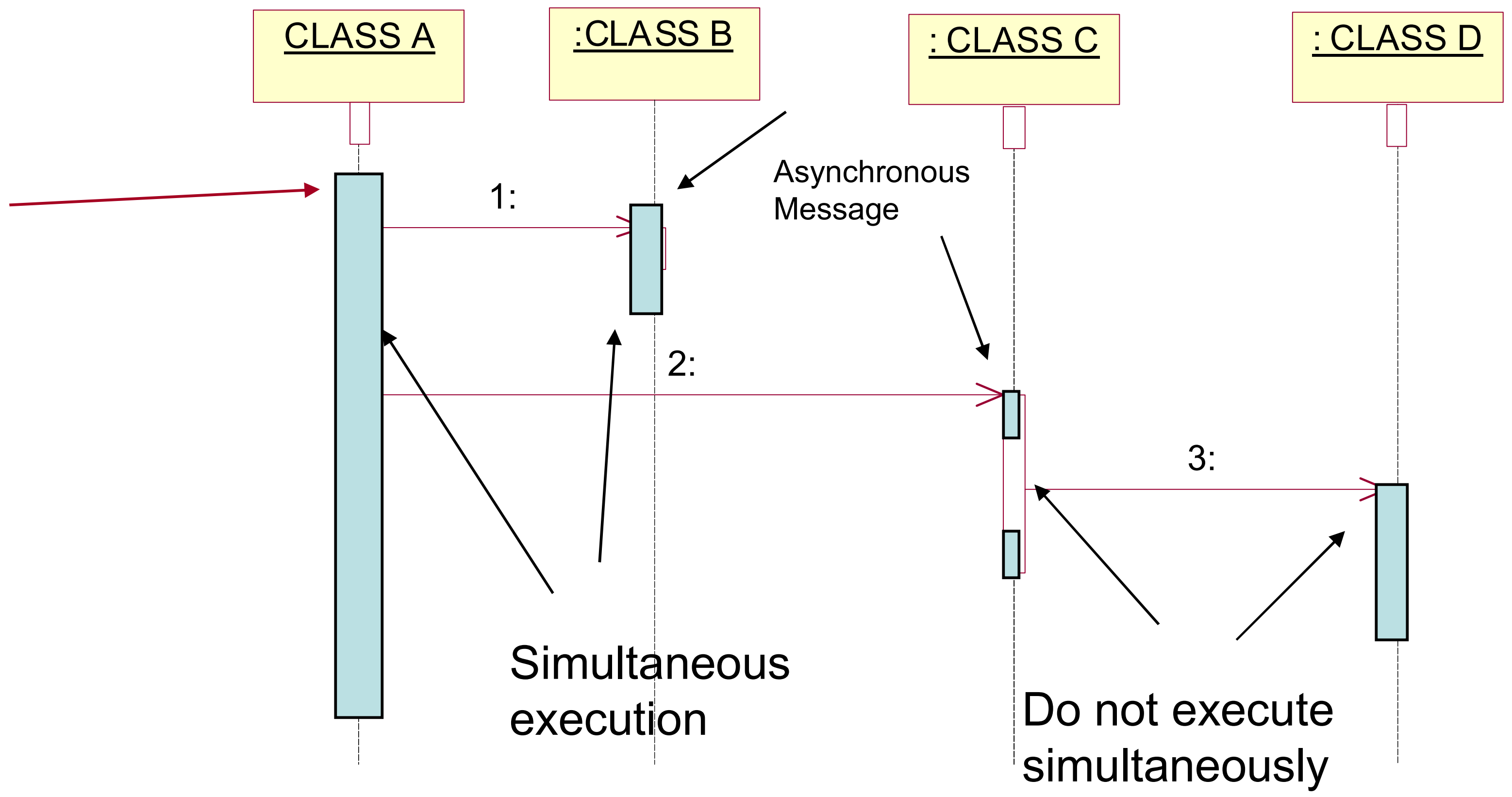
These models can be used to identify circumstances where concurrent processing is necessary.

1. A use case may indicate a requirement that the system should be able to respond simultaneously to different events, each of which triggers a different thread of control.

2. If a statechart reveals that a class has complex nested states which themselves have concurrent substates, then the design must be able to handle this concurrency. The statechart for the class Campaign has nested concurrent states within the Active state and there may be the possibility of concurrent activity.

In cases where an object is required to exhibit concurrent behaviour it is sometimes necessary to split the object into separate objects in order to avoid the need for concurrent activity within anyone object.

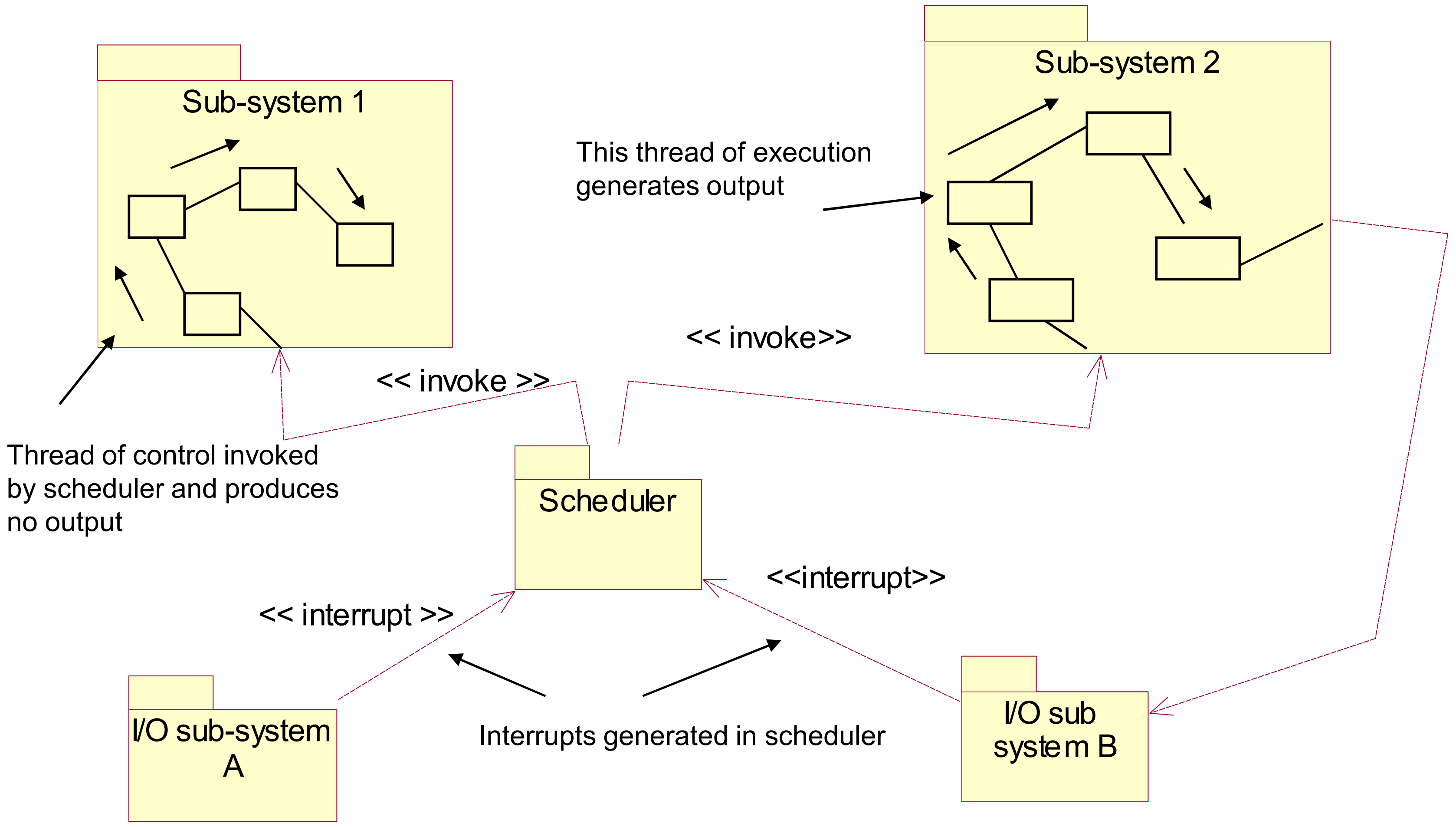
Concurrent processing may also be indicated if interaction diagrams reveal that a single thread of control requires that operations in two different objects should execute simultaneously, perhaps because of asynchronous invocation. This essentially means that one thread of control is split into two or more active threads. An example of this is shown below.



Concurrency activity in an interaction diagram

Different objects that are not active at the same time can be implemented on the same logical processor .Objects that must operate concurrently must be implemented on different logical processors.

The following figure shows a possible relationship between a scheduler and the other parts of a system. Events that are detected by the I/O (input/output) sub-systems generate interrupts in the scheduler. The scheduler then invokes the appropriate thread of control. Further interrupts may invoke other threads of control and the scheduler allocates a share of physical processor time to each thread .


Sub-system 1
Sub-system 2
This thread of execution generates output
<< invoke>>
<< invoke >>
Thread of control invoked by scheduler and produces no output
Scheduler
<<interrupt>>
<< interrupt >>
I/O sub-system A
Interrupts generated in scheduler
I/O sub system B

Another way of implementing concurrency is to use a multi-threaded programming language. These permit the direct implementation of concurrency within a single processor task. Finally, a multi-processor environment allows each concurrent task to be implemented on a separate processor. Most concurrent activity in a business information system can be supported by a multi-user environment. These are designed to allow many users to perform tasks simul-taneously.

5 Processor Allocation

In the case of single-user system it is almost always appropriate for the complete system to operate on a single computer. The software for a multi-user information system may be installed on many computers that all use a common file-server. More complex applications sometimes require the use of more than one type of computer, where each provides a specialized kind of processing capability for a specific sub-system.

An information system may also be partitioned over several processors either because sub-systems must operate concurrently, or because some parts of the application need to operate in different locations. Information systems that use the Internet or company intranets for their communications are being built more frequently. Such distributed information systems operate on diverse computers and operating systems.

The allocation of a system to multiple processors on different platforms involves the following steps.

- The application should be divided into sub-systems.
- Processing requirements for each sub-system should be estimated.
- Access criteria and location requirements should be determined.
- Concurrency requirements for the sub-systems should be identified.
- Each sub-system should be allocated to an operating platform-either general purpose (PC or workstation) or specialized (embedded micro-controller or specialist server).
- Communication requirements between sub-systems should be determined.
- The communications infrastructure should be specified.

The estimation of processing requirements requires careful consideration of such factors as event response times, the data throughput that is needed, the nature of the I/O that is required and any special algorithmic requirements.

6. Data Management Issues

Suitable data management approaches for an information system can vary from simple file storage and retrieval to sophisticated database management systems of various types. **In** some applications where data has to be accessed very rapidly, the data may be kept in main memory while the system executes. However, most data management is concerned with storing data, often large volumes, so that it may be accessed at a later stage either by the same system or by another.

Database management systems (DBMS) provide the following facilities that are useful in many applications.

- different views of the data by different users,
- control of multi-user access,
- distribution of the data over different platforms,
- security,
- enforcement of integrity constraints,
- access to data by various applications,
- data recovery,
- portability across platforms,
- data access via query languages and
- query optimization.

DBMS exhibits a significant performance overhead and the standard data access mechanisms may be inappropriate for specialized systems.

Relational DBMS is likely to be appropriate if there are large volumes of data with varying (perhaps ad hoc) access requirements.

Object-oriented DBMS is more likely to be suitable if specific transactions require fast access or if there is a need to store complex data structures and there is not a need to support a wide range of transaction types.

A third type of DBMS is emerging-the object-relational DBMS-that is similar to an object-oriented DBMS in its support for complex data structures, but that also provides effective querying facilities. In some systems there may be different data management requirements for different sub-systems and it may be best then to use a mix of DBMS types.

7. Development standards

All information systems development projects should operate with clearly defined guidelines within which all members of the development team work. Many organizations will have corporate style guides that govern the production of software development artefacts, including the delivered system.

In some organizations these corporate guides may be adapted for particular development projects. From the design perspective it is important to specify guidelines for the development of I/O sub-systems and their interfaces and standards for the development of code.

1. **HCI guidelines -** Standards for the human-computer interface are an important aspect of the design activity, since it is with the interface that users actually interact.

2 Input/output device guidelines

Where an application interacts with mechanical or electronic devices such as temperature and pressure sensors or actuators that control heaters or motors, it is equally important to develop guidelines.

The objective is to use a standard form of interface with the devices so that hardware may be changed or updated without occasioning any changes to the core system functionality. Sensor and controller devices usually have fully specified communications protocols and standardization is probably best achieved by encapsulating all direct access with each device in a single I/O device object. An I/O device class can be sub classed so that for each particular device involved with the application there is a class that deals with its communications protocol. Since all the I/O device classes would then be subclasses of one inheritance hierarchy they can easily be constructed to provide consistent interfaces to the rest of the application. This feature can be achieved by using Polymorphism concept of OO.

3 Construction guidelines

Construction guidelines will normally include advice on the naming of classes, operations and attributes, and where this is the case these guidelines are also applicable during the analysis activity. Wherever possible, consistent naming conventions should be enforced throughout the project since this makes it easier to trace an analysis class directly through to its implementation.

8. Prioritizing Design Trade-Offs

Design frequently involves choosing the most appropriate compromise. The designer is often faced with design objectives that are mutually incompatible and he or she must then decide which objective is the more important. The requirements model may indicate the relative priorities of the different objectives but, if it does not, then prepare general guidelines used for this purpose. These guidelines must be agreed with clients since they determine the nature of the system and what functionality will be delivered.

Guidelines for design trade-offs ensure consistency between the decisions that are made at different stages of development. They also ensure consistency between different sub-systems . However, no guidelines can legislate for every case. Design experience and further discussions with the client will remain necessary to resolve those situations that cannot be anticipated .

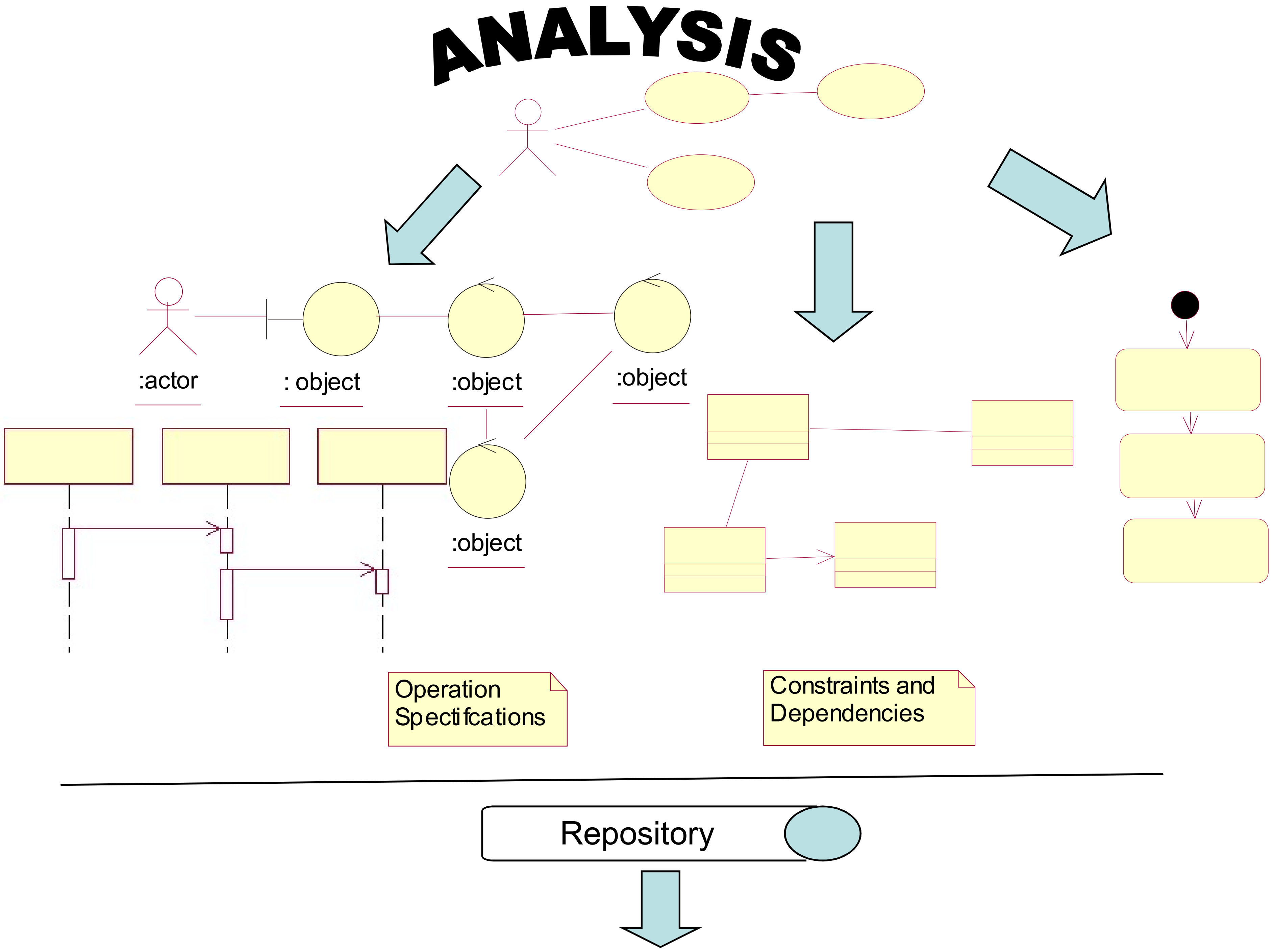
14. Object Design

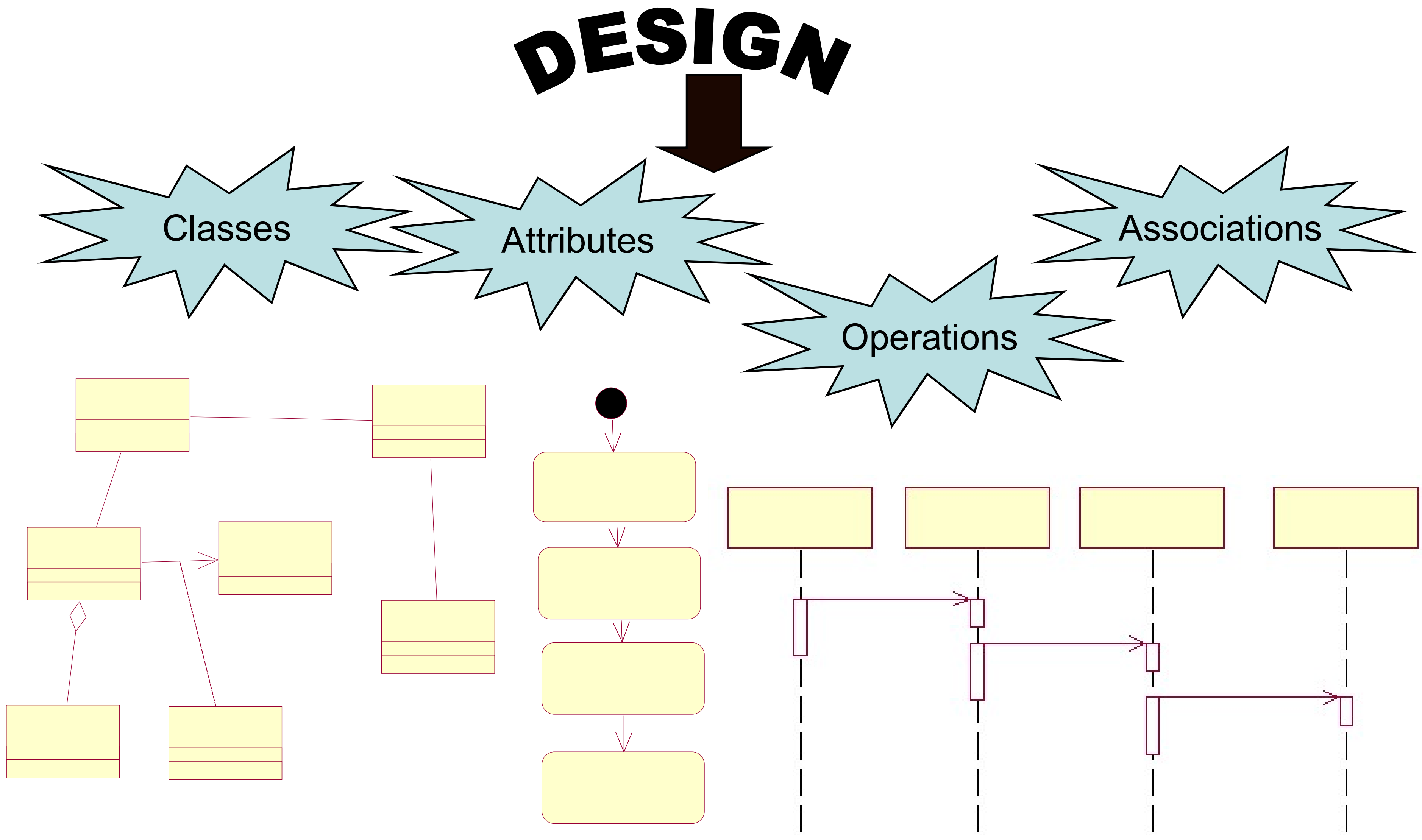
1. Object Design Process

Object design is concerned with the detailed design of the objects and their interactions. It is completed within the overall architecture defined during system design and according to agreed design guidelines and protocols. Object design is particularly concerned with the specification of the attribute types, how operations function and how objects are linked to other objects.

The following figure shows various resources of information that guides the object design process .During the design process the analysis models undergo some degree of transformation. There is a commonly accepted view that changes made to analysis artifacts to produce the design model should be kept to a minimum, as the analysis model is a coherent and consistent description of the requirements .

Object design produces a detailed specification of the classes ,attributes and operation signatures. In this an important aspect of every class is what attributes and operations are generally accessible during the interaction among the objects.


ANALYSIS
:actor
: object
:object
:object
:object
Operation Specifcations
Constraints and Dependencies
Repository


DESIGN
Classes
Attributes
Operations
Associations

CLASS SPECIFICATION - Attributes and operation signatures

Attributes

During analysis Stage we need to consider in detail the data types of the attributes also. For example, an attribute temperature might be a floating-point data type if it holds the temperature in Centigrade or it might be an enumerated data type if it holds one of the values, 'hot' or 'cold'. The attribute has a different meaning and would be manipulated differently for each of these data types and it is important to determine during analysis which meaning is appropriate.

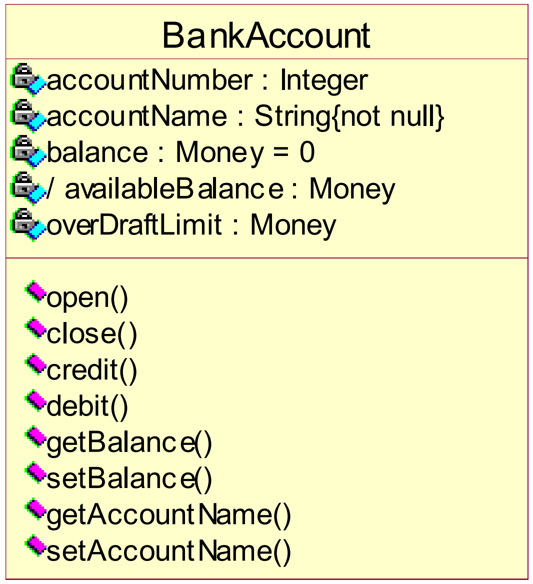
Common primitive data types include Boolean (true or false), Character (any alphanumeric or special character), Integer (whole numbers) and Floating-Point (decimal numbers). In most object-oriented languages more complex data types, such as Money, String, Date, or Name can be constructed from the primitive data types or may be available in standard libraries. An attribute's data type is declared in UML using the following syntax:

name ':' type-expression '=' initial-value '{'property-string'}'

The name is the attribute name, the type-expression is its data type, the initial-value is the value the attribute is set to when the object is first created and the property-string describes a property of the attribute, such as constant or fixed. The characters in single quotes are literals.

The following is a class BankAccount which is shown along with attribute data types declared. The attribute balance in a BankAccount class might be declared with an initial value of zero using the syntax:

balance:Money = 0.00



The attribute accountName might be declared with the property string indicating that it must have a value and may not be null using the syntax:

accountName : String {not null}

Attribute declarations can also include arrays also. For example, an Employee class might include an attribute to hold a list of qualifications that would be declared using the syntax:

qualification[O .. 10]: String

Operations

Each operation also has to be specified in terms of the parameters that it passes and returns. The syntax used for an operation is:

operation name' ('parameter-list ') " : " return-type-expression

An operation's *signature* is determined by the operation's name, the number and type of its parameters and the type of the return value if any. In the BankAccount class we have a credit () operation that passes the amount being credited to the receiving object and has a Boolean return value. The operation would be defined using the syntax:

```
credit(amount : Money): Boolean
```

A credit() message sent to a BankAccount object could have the format:

```
creditOK = accObject.credit(500.00)
```

where creditOK holds the Boolean return value that is available to the sending object when the credit () operation has completed executing. This Boolean value may be tested to determine whether the credit () operation performed successfully.

As UML is a modelling language, It does not determine what operations should be shown in a class diagram instead, It provides the notation to use and suggestions on presentation, but does not tell the analyst or designer what to include and what not to include.

Specific guidelines about operations in object-oriented analysis and design includes in the following way.

According to Coad and Yourdon , an object provides services to other objects in a system. These services are like responsibilities at the system level, and these can also be called asl the operations of classes. Also we have some services which are implicit services , like creating instances of objects, to modify attributes of instances, to select instances based on some kind of key or identifier, and to delete instances.

These operations need not be shown on diagrams explicitely, as they clutter up the diagrams and make them difficult to read. They also point out that sometimes it is important to be able to see these services. However, this is an issue about the functionality offered by CASE tools rather than methodologies. Ideally, it should be possible to switch off the display of any operation that the analyst or designer does not wish to have displayed in the operations compartment in a class.

An alternative approach would be to follow the way that some CORBA tools work. The IDL2JAVA tool will generate two Java operations for every attribute: one to set the values of the attribute and one to get the value of the attribute.

The following example shows the operations generated to set and get the value of the attribute smallUnit in a Currency class.

```
short smallUnit;
…..
public void smallUnit(short smallUnit)
{
// implement attribute writer ...

smallUnit = smallUnit;
}
public short smaIIUnit()
{
// implement attribute reader ...
return smallUnit;
}
```

The framework for this fragment of code was generated automatically from the following IDL interface definition.

```
interface Money {
attribute long largeUnit;       attribute short smallUnit;        attribute string format;
Money Money( in long large, in short small);      string toString();
} ;
```

So, if CASE tool generates set and get operations for every attribute automatically, then no need to include them in the class diagram.

One commonly held approach is normally not to show primary operations on analysis class diagrams as it can be assumed that such functionality is available. During analysis issues such as the visibility of operations or the precise data types of attributes may not have been finally decided. However, when completing a design class diagram it may be important to indicate that certain primary operations have public or protected visibility and as such these may justifiably be shown on the diagram. Those that are private may be omitted as they do not constitute part of the class public interface.

Exceptionally primary operations may usefully be included on analysis class diagrams either if they reflect particular functionality that has to be publicly visible or if it is important to indicate that more than one constructor, for example, is required. A class may need more than one constructor if objects could be instantiated in one of several initial states that require different input parameters. Each constructor would have a different signature.

Object visibility

The concept of encapsulation is one of the fundamental principles of object-orientation. During analysis various assumptions have been made regarding the encapsulation boundary for an object and the way that objects interact with each other.

For example, it is assumed that the attributes of an object cannot be accessed directly by other objects but only via 'get' and 'set' operations (primary operations) that are assumed to be available for each attribute. Moving to design involves making decisions regarding which operations (and possibly attributes) are publicly accessible. **In** other words we must define the encapsulation boundary.

A class bankAccount has the attribute balance, which we might assume during analysis, can be accessed directly by the simple primary operations getBalance () and setBalance ( ). However, the balance should be updated through the operations credit () and debit () that contain special processing to check whether these transactions should be permitted and to ensure that the transactions are logged in an audit trail. **In** these circumstances, it is important that changes to the value of the balance attribute can only occur through the debit() and credit() operations. The operation setBalance () should not be publicly available for use by other classes. And here the attribute availableBalance is a derivable attribute indicated in UML by the symbol ' / '. A derivable attribute is one whose value can be calculated or determined from the value of other attributes.

Meyer introduced the term 'secret' to describe those features that are not available in the public interface. Programming languages designate the non-public parts of a class, which may include attributes and operations, in various ways. The four commonly accepted termss used to describe *visibility* are listed in the following figure.

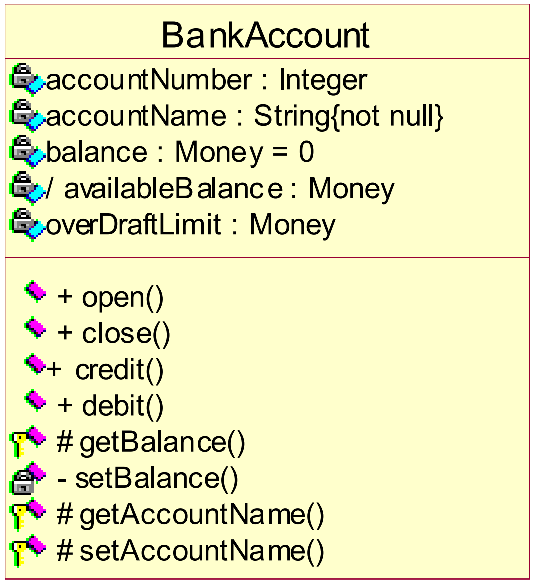
Visibility may also be shown as a property string.

Balance : Money {visibility = private}

Visibility symbol	Visibility	Meaning
+	Public	The feature (an operation or an attribute) is directly accessible by an instance of any class.
-	Private	The feature may only be used by an instance of the class that includes it.
#	protected	The feature may be used either by instances of the class that includes it or of a subclass or descendant of that class.
~	Package	The feature is directly accessible only by instances of a class in the same package.

To enforce encapsulation the attributes of a class are normally designated private. The operation setBalance() is also designated private to ensure that objects from other classes cannot access it directly , and shown in the following figure.

In the figure the operation getBalance () is assigned protected visibility so that subclasses of BankAccount can examine the value of the balance attribute. For example, the debit () operation might be redefined polymorphically in a Junior BankAccount subclass. The redefined operation would use getBalance () to access the balance and check that a debit would not result in a negative balance.



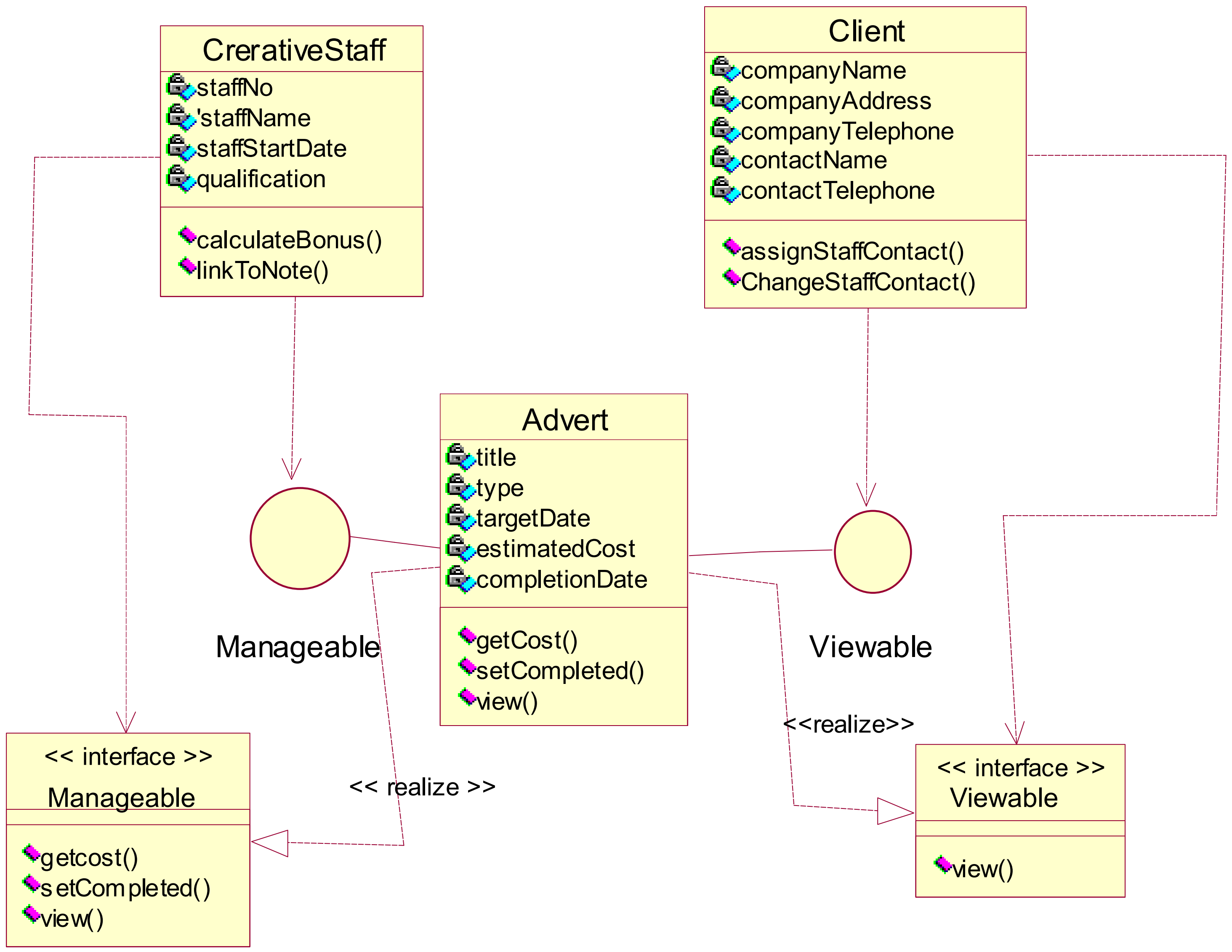
2. Interfaces

Generally a class may present more than one external interface to other classes or the same interface may be required from more than one class. An interface in UML is a group of externally visible (i.e. public) operations. The interface contains no internal structure, it has no attributes, no associations and the implementation of the operations is not defined. Formally, an interface is equivalent to an abstract class that has no attributes, no associations and only abstract operations.

The following figure shows two alternative notations for an interface. The simpler of the two UML interface notations is a circle. This is attached by a solid line to the classes that support the interface. For example, in Figure the Advert class supports two interfaces, Manageable and Viewable, that is, it provides all of the operations specified by the interface . The circle notation does not include a list of the operations provided by the interface type, though they should be listed in the repository. The dashed arrow from the CreativeStaff class to the Manageable interface circle icon indicates that it uses or needs, at most, the operations provided by the interface.

The alternative notation uses a stereotyped class icon. As an interface only specifies the operations and has no internal structure, the attributes compartment is omitted. This notation lists the operations on the diagram. The *realize* relationship, represented by the dashed line with a triangular arrowhead, indicates that the client class (e.g. Advert) supports at least the operations listed in the interface .Again the dashed arrow from CreativeStaff means that the class needs or uses no more than the operations listed in the interface.

Interfaces for the Advert Class



3. Criteria for Good Design

1 Coupling and cohesion – These factors coupling and cohesion are inportant factors for good design.

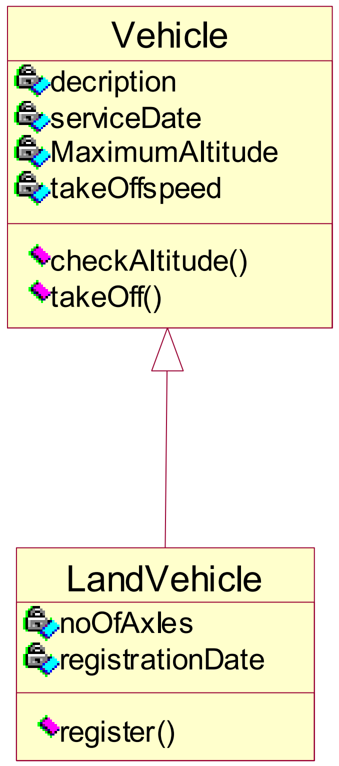
Coupling describes the degree of interconnectedness between design components and is reflected by the number of links an object has and by the degree of interaction the object has with other objects.

Cohesion is a measure of the degree to which an element contributes to a single purpose. The concepts of coupling and cohesion are not mutually exclusive but actually support each other. This criteria can be used within object-orientation as described below.

Interaction Coupling is a measure of the number of message types an object sends to other objects and the number of parameters passed with these message types. Interaction coupling should be kept to a minimum to reduce the possibility of changes rippling through the interfaces and to make reuse easier. When an object is reused in another application it will still need to send these messages and hence needs objects in the new application that provide these services. This complicates the reuse process as it requires groups of classes to be reused rather than individual classes.

Inheritance Coupling describes the degree to which a subclass actually needs the features it inherits from its base class.

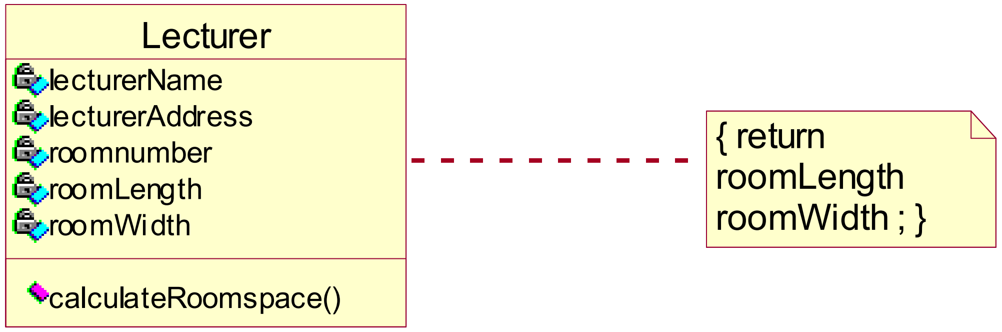
Inheritance coupling.



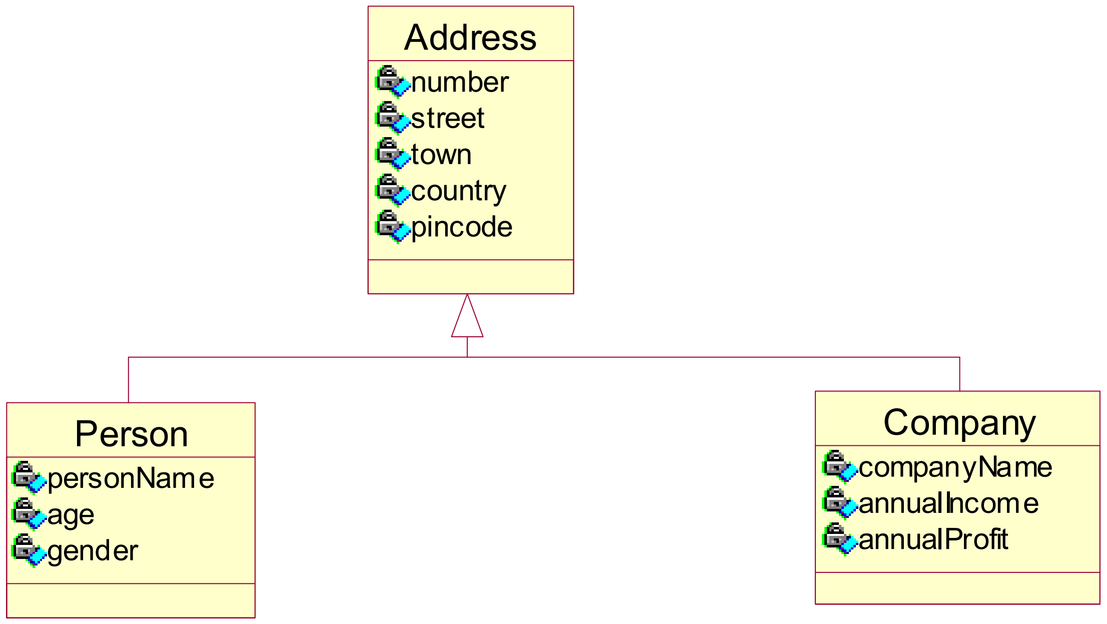
For example, in the above figure, the inheritance hierarchy exhibits low inheritance coupling and is poorly designed. The subclass LandVehicle needs neither the attributes maximumAltitude and takeOff Speed nor the operations checkAltitude () and takeOff (). They have been inherited unnecessarily.

In this example it shows the base class, Vehicle, would be better named FlyingVehicle and the inheritance relationship is somewhat suspect. A land vehicle is not a kind of flying vehicle .However, many systems developers view designs with a small degree of unnecessary inheritance as being acceptable if the hierarchy is providing valuable reuse and is meaningful. However, a subclass with unnecessary attributes or operations is more complex than it needs to be and objects of the subclass may take more memory than they actually need. The real problems may come when the system needs maintenance. The system's maintainer may not realize that some of the inherited attributes and operations are unused and may modify the system incorrectly as a result. Alternatively the system's maintainer may use these unneeded features to provide a fix for a new user requirement, making the system even more difficult to maintain in the future. For these reasons, unnecessary inheritance should be kept as low as possible.

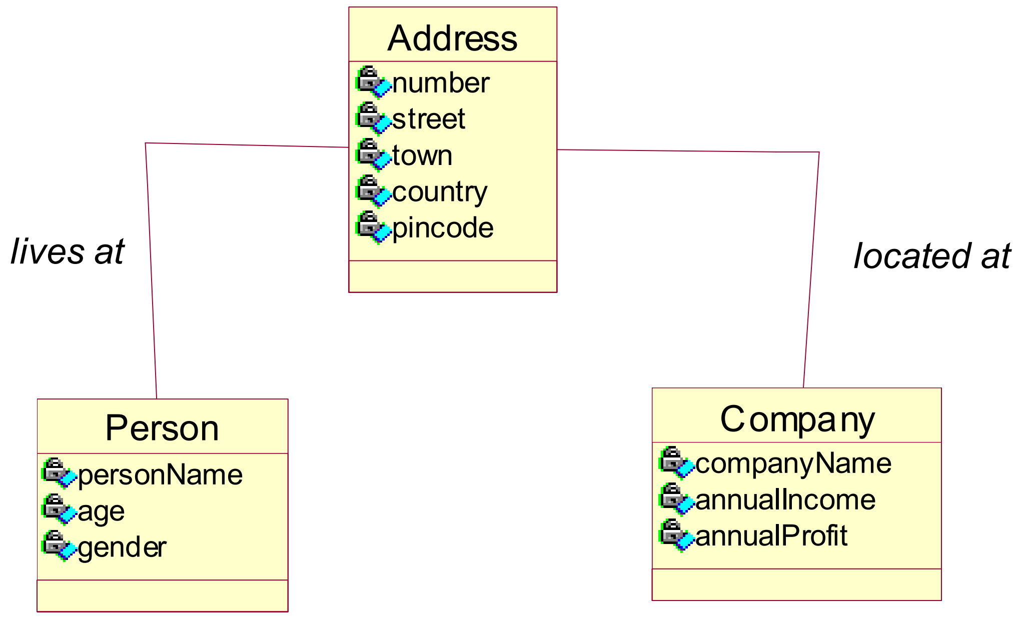
Operation Cohesion measures the degree to which an operation focuses on a single functional requirement. Good design produces highly cohesive operations, each of which deals with a single functional requirement. For example in the following figure , the operation calculateRoomSpace () is highly cohesive.



Class Cohesion reflects the degree to which a class is focused on a single requirement. The class Lecturer in the previous figure exhibits low levels of cohesion as it has three attributes (roomNumber, roomLength and roomWidth and one operation calculate RoomSpace ()) that would be more appropriate in a class Room. The class Lecturer should only have attributes that describe a Lecturer object (e.g. lecturerName and lecturerAddress) and operations that use them. ***Specialization Cohesion*** addresses the semantic cohesion of inheritance hierarchies. For example in the following figure all the attributes and operations of the Address base class are used by the derived classes - this hierarchy has high inheritance coupling. However, it is neither true that a person is a kind of address nor that a company is a kind of address. The example is only using inheritance as a syntactic structure for sharing attributes and operations. This structure has low specialization cohesion and is poor design. It does not reflect meaningful inheritance in the problem domain.



A better design is shown in the following figure , in which a common class Address is being used by both the Person and Company classes.

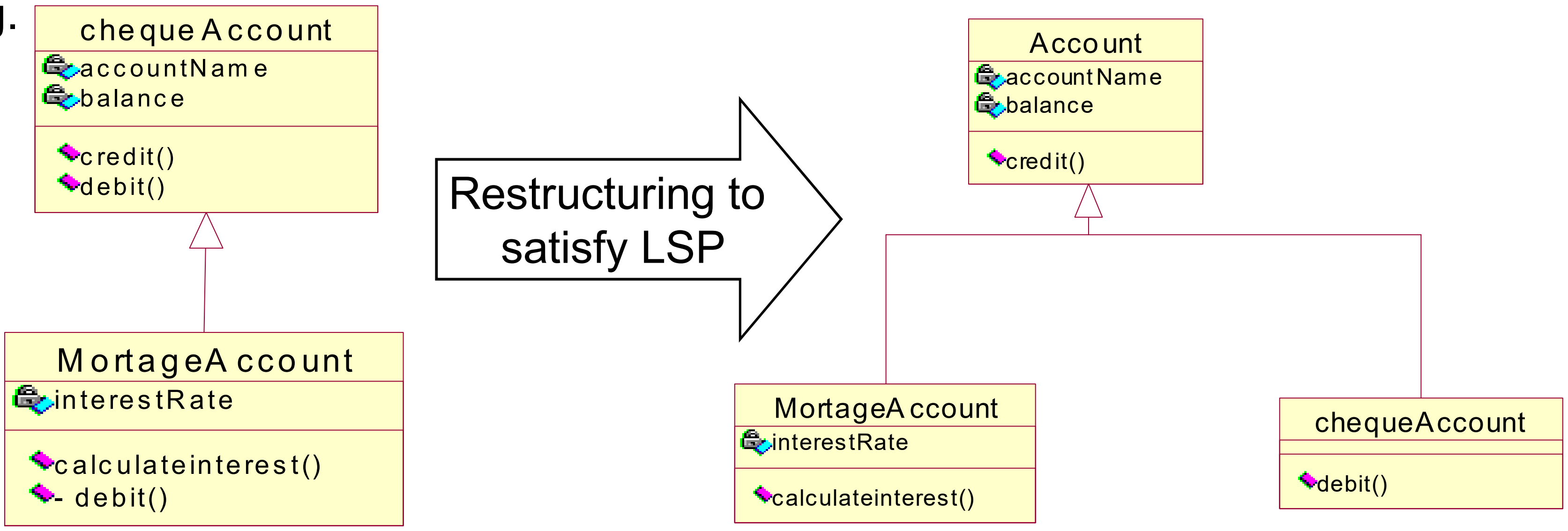


All the above are considered as design criteria , which may be applied at the same time to good effect.

2 Liskov Substitution Principle

The ***Liskov Substitution Principle (LSP)*** is another design criteria which is applicable to inheritance hierarchies. LSP states that, in object interactions, it should be possible to treat a derived object as if it were a base object. If the principle is not applied then it may be possible to violate the integrity of the derived object.

In the following figure objects of the class MortgageAccount cannot be treated as if they are objects of the class ChequeAccount because MortgageAccount objects do not have a debit operation whereas ChequeAccount objects do. The debit operation is declared private in MortgageAccount and hence cannot be used by any other object. It also shows an alternative structure that satisfies LSP. Interestingly, this inheritance hierarchy has maximal inheritance coupling, and enforcing the LSP normally produces structures with high inheritance coupling.



3 Further design guidelines

The following are the further guidelines which are to be considered for good design.

Design Clarity. A design should be made as easy to understand as possible. This reinforces the need to use design standards or protocols that have been specified.

Don't Over-Design. Developers are on occasions tempted to produce designs that may not only satisfy current requirements but may also be capable of supporting a wide range of future requirements. Designing flexibility into a system has a cost, the system may take longer to design and construct but this may be offset in the future by easier and less expensive modification. However, it is not feasible to design for every eventuality. Systems that are over-designed in first instance are more difficult to extend if the modifications are not sympathetic to the existing structure.

Control Inheritance Hierarchies. Inheritance hierarchies should be neither too deep nor too shallow. If a hierarchy is too deep it is difficult for the developer to understand easily what features are inherited. There is a tendency for developers new to 00 to produce over-specialized hierarchies, thus adding complexity rather than reducing it.

Keep Messages and Operations Simple. In general it is better to limit the number of parameters passed in a message to no more than three .Ideally an operation should be capable of specification in no more than one page.

Design Volatility. A good design will be stable in response to changes in requirements. It is reasonable to expect some change in the design if the requirements are changed. However, any change in the design should be commensurate with the change in requirements. Enforcing encapsulation is a key factor in producing stable systems.

Evaluate by Scenario. An effective way of testing the suitability of a design is to role play it against the use cases using CRC cards.

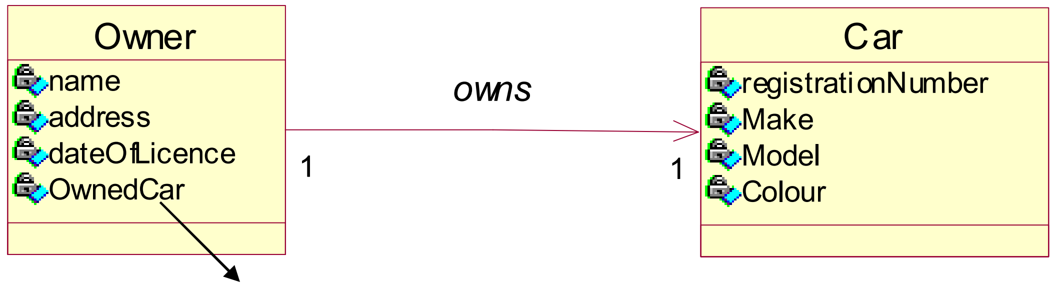
Design by Delegation. A complex object should be decomposed (if possible) into component objects forming a composition or aggregation. Behaviour can then be delegated to the component objects producing a group of objects that are easier to construct and maintain. This approach also improves reusability.

Keep Classes Separate. In general, it is better not to place one class inside another. The internal class is encapsulated by the other class and cannot be accessed independently. This reduces the flexibility of the system.

4. Designing Associations

An association between two classes indicates the possibility that links will exist between instances of the classes. The links provide the connections necessary for message passing to occur. When deciding how to implement an association it is important to analyze the message passing between the objects tied by the link.

1 One-to-one Associations In the following figure objects of the class Owner need to send messages to objects of the class Car but not vice versa. This particular association may be implemented by placing an attribute to hold the object identifier for the Car class in the Owner class. Thus Owner objects have the Car object identifier and hence can send messages to the linked Car object. Here the **owns** association is an example of a one-way association: the arrow-head on the association line shows the direction along which it may be navigated. So before an association can be designed it is important to decide in which direction or directions messages may be sent.



CarobjectId is placed in the owner class

In general an association between two classes A and B should be considered with the questions:

1.Do objects of class A have to send messages to objects of class B?

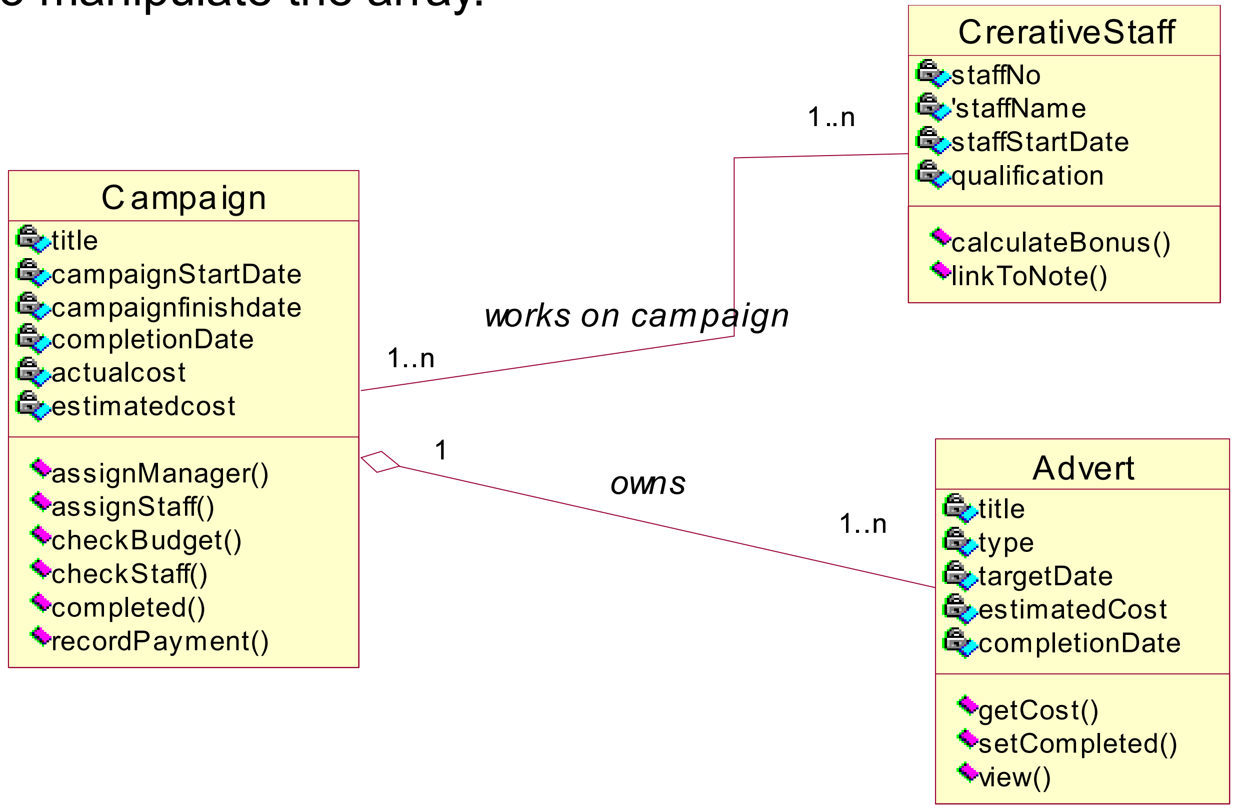
2. Does an A object have to provide some other object with B object identifiers?

If either of these questions is answered 'yes' then A objects need B object identifiers. However if A objects get the required B object identifiers as parameters in incoming messages, A objects need not remember the B object identifiers. Essentially, if an object needs to send a message to a destination object it must have the destination object's identifier either passed as a parameter in an incoming message just when it is required, or the destination object's identifier must be stored in the sending object.

An association that has to support message passing in both directions is a two-way association. A two-way association is indicated with arrowheads at both ends or with solid line , and it is important to minimize the coupling between objects. Minimizing the number of two-way associations keeps the coupling between objects as low as possible.

2 One-to-many Associations

In the following figure, objects of the class Campaign need to send messages to objects of the class Advert but not vice versa. If the association between the classes was one-to-one, the association could be implemented by placing an attribute to hold the object identifier for the Advert class in the Campaign class. However, the association is in fact one-to-many and many Advert object identifiers need to be tied to a single Campaign object. The object identifiers could be held as a simple one-dimensional array in the Campaign object but program code would have to be written to manipulate the array.



3 Many-to-many Associations

The Association of the type many-to-many association workOnCampaign is shown between Creative Staff and Campaign in previous figure.Assuming this is a two-way association, each Campaign object will need a collection of CreativeStaff object identifiers and each Creative Staff object will need a collection of Campaign object identifiers.

5. Integrity Constraints

Systems analysis identifies a series of integrity constraints that have to be enforced to ensure that the application holds data that is mutually consistent and manipulates it correctly. These integrity constraints come in various forms and includes:

Referential Integrity constraints ensures that an object identifier in an object is actually referring to an object that exists.

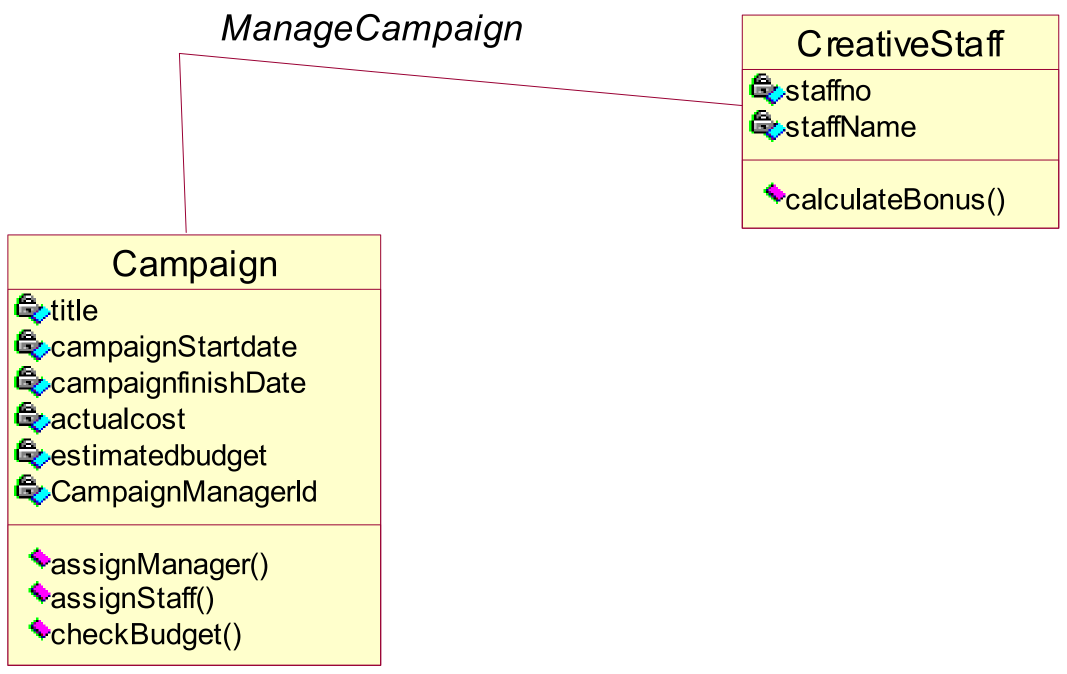
Dependency Constraints ensures that attribute dependencies, where one attribute may be calculated from other attributes, are maintained consistently.

Domain Integrity constraint ensures that attributes only hold permissible values.

1 Referential integrity Constraints

The concept of referential integrity as applied to a relational database management system can be applied when considering references between objects.

In the following figure the association manageCampaign between CreativeStaff and Campaign is two-way, and an object identifier called campaignManagerId that refers to the particular CreativeStaff object that represents the campaign manager is needed in Campaign. To maintain referential integrity the system must ensure that the attribute campaign ManagerId either is null or contains the object identifier of a CreativeStaff object that exists.



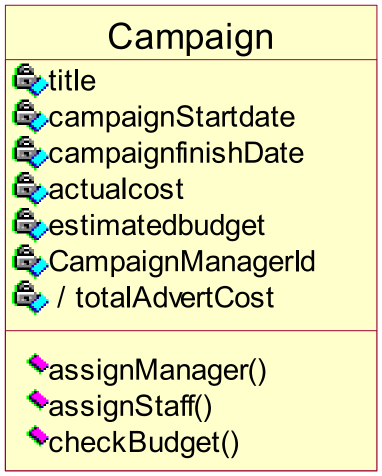
In this particular case the association states that a Campaign must have a CreativeStaff instance as its manager, and it is not correct to have a Campaign with a null campaignManagerId attribute. In order to enforce this constraint, the constructor for Campaign needs as one of its parameters the object identifier of the CreativeStaff object that represents the campaign manager, so that the campaignManagerId attribute can be instantiated with a valid object identifier.

Problems in maintaining the referential integrity of a Campaign may occur during its lifetime. For instance, the campaign manager, NameX, may leave the company to move to another job and NameX CreativeStaff object will then be deleted. Referential integrity is maintained by ensuring that the deletion of a CreativeStaff object that is a campaign manager always involves allocating a new campaign manager. The task of invoking the operation assignManager() is included in the Creative Staff destructor, and it will request the object identifier of the new campaign manager. Similarly, any attempt to remove the current campaign manager from a Campaign must always involve allocating the replacement.

The multiplicity of exactly one represents a strong integrity constraint for the system. In the example just discussed, it seems to be appropriate that a campaign should always have a manager, even when it has just been created. However, great care should be taken when assigning a multiplicity of exactly one (or in general a minimum of one) to an association, as the consequences in the implemented system can be quite dramatic.

2 Dependency constraints

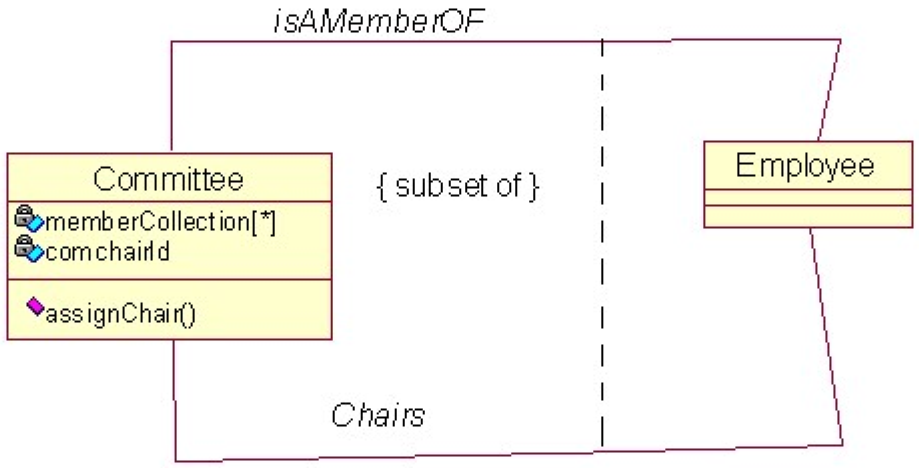
Attributes are dependent upon each other in various ways. These dependencies may have been identified during analysis and must now be dealt with during design. A common form of dependency occurs when the value of one attribute may be calcuated from other attributes. For instance, a requirement to display the total advertising cost may be satisfied either by storing the value in the attribute totalAdvertCost in the Campaign class or by calculating the value every time it is required. The attribute totalAdvertCost is a derived attribute and its value is calculated by summing the individual advert costs. Placing the derived attribute in the class reduces the processing required to display the total advertising cost as it does not require calculation. On the other hand, whenever the cost of an advert changes, or an advert is either added to or removed from the campaign, then the attribute totalAdvertCost has to be adjusted so that it remains consistent with the attributes upon which it depends. The UML symbol ('/') used to indicate that a modelling element (attribute or association) is derived as shown below.



In order to maintain the consistency between the attributes, any operation that changes the value of an Advert's cost must trigger an appropriate change in the value of totalAdvertCost by sending the message adjustCost () to the Campaign object. The operation adjustCost () is an example of a *synchronizing operation.* The operations that have to maintain the consistency are setAdvertCost () and the Advert destructor. When a new advert is created the constructor would use setAdvertCost () to set the advert cost. This would invoke adjustCost() and hence ensure that the totalAdvertCost is adjusted. So any change to an Advert's cost takes more processing if the derived attribute totalAdvertCost is used. Thus one part of the system executes more quickly while another part executes more slowly. Generally it is easier to construct systems without derived attributes, as this indicates the need for complex synchronizing operations. Derived attributes should only be introduced if performance constraints cannot be satisfied without them. If performance is an issue then one of the skills needed in design is how to optimize the critical parts of the system without making the other parts of the system inoperable.

Another form of dependency occurs where the value of one attribute is constrained by the values of other attributes.

Dependency constraints can also exist between or among associations. In the following example, the chairs association is a subset of the isAMemberOf association. This constraint is stating that the chair of a committee must be a member of the committee, and it can be enforced by placing a check in the assignChair () operation in Committee to confirm that the Employee object identifier passed as a parameter is already in the collection class of committee members. More complex constraints may also exist that require several associations. Derived associations may also be introduced to improve performance if absolutely necessary and, as in the case of derived attributes, synchronizing operations are needed to ensure that the derived links are consistent with the links on which they depend.



3 Domain integrity

Domain integrity is concerned with ensuring that the values an attribute takes are from the appropriate underlying domain. For instance, the attributes from the Cost domain might reasonably be non-negative decimal values with two decimal places. These constraints may be viewed as an extended form of those implied by data types. The necessary integrity checking code is normally placed in the 'set' operations or in any interactive interface that permits the entry of values.

6. Designing Operations

The design of operations involves determining the best algorithm to perform the required function. In the simplest case, primary operations require little design apart from the inclusion of code to enforce integrity checks. For more complex operations, algorithm design can be an involved process. Various factors constrain algorithm design including:

- the cost of implementation,
- performance constraints,
- requirements for accuracy and
- the capabilities of the implementation platform.

The following factors should be considered when choosing among alternative algorithm designs.

- ***Computational complexity.*** This is concerned with the performance characteristics of the algorithm as it operates on increasing numbers of input values. For example, the bubble sort algorithm has an execution time that is proportional to N X N where N is the number of items being sorted.
- ***Ease of implementation and understandability.*** It is generally better to sacrifice some performance to simplify implementation.
- ***Flexibility.*** Most software systems are subject to change and an algorithm should be designed with this in mind .
- ***Fine-tuning the object model.*** Some adjustment to the object model may simplify the algorithm and should be considered.

Designing the main operations in a class is likely to highlight the need for lower-level private operations to decompose complex operations. This process is much the same as traditional program design. Techniques such as step-wise refinement or structure charts may well be used to good effect.

UML offers activity diagrams as a technique both to document and to design operations. In circumstances where high levels of formality are required in operation design.

Responsibilities identified during analysis may map onto one or more operations. The new operations that are identified need to be assigned to classes. In general, if an operation operates on some attribute value then it should be placed in the same class as the attribute. On occasions a particular operation may modify attributes in more than one class and could sensibly be placed in one of several classes. In choosing where to locate the operation, one view is that minimizing the amount of object interaction should be a major criterion, while another significant criterion is simplicity.

7. Normalization

One form of dependency can be given among class attributes as functional dependency. For two attributes A and B, A is functionally dependent on B if for every value of B there is precisely one value of A associated with it at any given time.

This is shown notationally as: $\mathbf{B} \longrightarrow \mathbf{A}$

Normalization may be useful when using a relational database management system as part of the implementation platform or as a guide to decomposing a large, complex object. Most object-oriented approaches to software development do not view normalization as essential, and structures that are not normalized are considered acceptable. In general however, object-oriented approaches, if applied with suitable quality constraints, will produce structures that are largely redundancy free.

15. DESIGN PATTERNS

Successful software development relies on the knowledge and expertise of the developer, among others also. These are built and refined during development stage. A system analyst applies his solutions to development problems, monitors their success or failure and produces more effective solutions . In the software development there may be possibility same type problem may to re-occur, in this case instead of going for new solutions we can go for utilizing an existing solutions which are already implemented, that is through an existing patterns.

Patterns provide a means for capturing knowledge about problems and successful solutions in software development.

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.

1. Software development patterns

A. Frameworks Vs Patterns

Frameworks are partially completed software systems that may be targeted at a specified type of application, for example sales order processing. An application system tailored to a particular organization may be developed from the framework by completing the unfinished elements and adding application specific elements. This may involve the specialization of classes and the implementation of some operations. Essentially the framework is a reusable mini-architecture that provides a structure and behaviour common to all applications of this type.

The major differences between patterns and frameworks can be summarized as :

1. Patterns are more abstract and general than frameworks. A pattern is a description of the way that a type of problem can be solved, but the pattern is not itself a solution.

2. Unlike a framework, a pattern cannot be directly implemented in a particular software environment. A successful implementation is only an example of a design pattern.

3. Patterns are more primitive than frameworks. A framework can employ several patterns but a pattern cannot incorporate a framework.

B. Pattern catalogues and languages

Patterns are grouped into catalogues and languages. A *pattern catalogue* is a group of patterns that are related to some extent and may be used together or independently of each other. The patterns in a *pattern language* are more closely related, and work together to solve problems in a specific domain.

For example, Cunningham documented the 'Check Pattern Language of Information Integrity', which consists of eleven patterns that address issues of data validation. All were developed from his experiance of developing interactive financial systems in Smalltalk.

One of these patterns, Echo, describes how data input should be echoed back to the user after it has been modified and validated by the information system Typically users enter small batches of values and then look at the screen to check that they have been correctly entered. The sequence in which a user can enter data into fields may not be fixed and so validation feedback be given one field at a time. For example, a user enters a value as 5.236. This might be echoed back by the system as 5.24 (correctly rounded to two decimal places). The user receives direct visual feedback that the value has been accepted and how it has been modified.

C. Software development principles and patterns

Patterns are intended to represent good design practice and hence are based upon software development principles, many of which have been identified since the early days of software development and applied within other development approaches than object-oriented ones. The following are the key principles that underlie patterns:

- abstraction,
- encapsulation,
- information hiding,
- modularization,
- separation of concerns,
- coupling and cohesion,
- sufficiency, completeness and primitiveness,
- separation of policy and implementation,
- separation of interface and implementation,
- single point of reference and
- divide and conquer (this means breaking a complex problem into smaller, more manageable ones).

D. Patterns and non-functional requirements

Patterns can address the issues that are raised by non-functional requirements also. The following are the important non-functional properties of a software architecture:

- changeability,
- interoperability,
- efficiency,
- reliability,
- testability and
- reusability.

These properties may be required of a complete system or a part of a system. For example, a particular set of functional requirements may be seen as volatile and subject to change. It is important to develop a structure for these requirements that can cope with change. Another requirement may be that a particular aspect of an application must be highly reliable. Again this requirement must be met by the design.

2. Documenting Patterns – Pattern templates

Patterns can be documented using one of several alternative templates. The *pattern template* determines the style and structure of the pattern description, and these vary in the emphasis they place on different aspects of patterns. The differences between pattern templates may provide variations in the problem domain but there is no agreement as to the most appropriate template even within a particular problem domain. Generally a pattern description should include the following elements .

Name. A pattern should be given a meaningful name that reflects the knowledge embodied by the pattern. This may be a single word or a short phrase. These names become the vocabulary for discussing conceptual constructs in the domain of expertise.

Problem. This is a description of the problem that the pattern addresses means the intent of the pattern. It should identify and describe the objectives to be achieved, within a specified context and constraining forces. For example, one problem might be concerned with producing a flexible design, another with the validation of data.

The problem can frequently be written as a question, for example 'How can a class be constructed that has only one instance and can be accessed globally within the application?' This question expresses the problem addressed by the Singleton pattern .

Context. The context of the pattern represents the circumstances or preconditions under which it can occur. The context should provide sufficient detail to allow the applicability of the pattern to be determined.

Forces. The forces embodied in a pattern are the constraints or issues that must be addressed by the solution. These forces may interact with and conflict with each other, and possibly also with the objectives described in the problem. They reflect the details of the pattern.

Solution. The solution is a description of the static and dynamic relationships among the components of the pattern. The structure, the participants and their collaborations are all described. A solution should resolve all the forces in the given context. A solution that does not resolve all the forces fails.

The following are the other features which are to be mentioned in pattern templates:

- An example of the use of a pattern that serves as a guide to its application;
- The context that results from the use of the pattern;
- The rationale that justifies the chosen solution;
- Related patterns;
- Known uses of the pattern that validate it (some authors suggest that until the problem and its solution have been used successfully at least three times-the *rule of three-they* should not be considered as a pattern);
- A list of aliases for the pattern;

3. Types of Design Patterns

Patterns are classified according to their scope and purpose into the following three main categories.

1. *Creational patterns*
2. *Structural patterns*
3. B*ehavioural patterns.*

The scope of a pattern may be primarily at either the class level or at the object level. Patterns that are principally concerned with objects describe relationships that may change at run-time and hence are more dynamic. Patterns that relate primarily to classes tend to be static and identify relationships between classes and their subclasses that are defined at compile-time.

The patterns are based on principles of good design that include the maximizing of encapsulation and the substitution of composition for inheritance wherever possible. Using composition as a design issue produces composite objects whose component parts can be changed, perhaps dynamically under program control hence resulting in a highly flexible system. Generally patterns will frequently use both inheritance and composition to achieve the desired result.

Changeability involves several different aspects - main-tainability, extensibility, restructuring and portability.

Maintainability is concerned with the ease with which errors in the information system can be corrected.

Extensibility addresses the inclusion of new features and the replacement of existing components with new improved versions. It also involves the removal of unwanted features.

Restructuring focuses on the reorganization of software components and their relationships to provide increased flexibility.

Portability deals with modifying the system so that it may execute in different operating environments, such as different operating systems or different hardware.

1. Creational patterns

A creational design pattern is concerned with the construction of object instances. In general, creational patterns separate the operation of an application from how its objects are created. This decoupling of object creation from the operation of the application gives the designer considerable flexibility in configuring all aspects of object creation. This configuration may be dynamic or static .

For example, when dynamic configuration is appropriate, an object-oriented system may use composition to make a complex object by aggregating simpler component objects. Depending upon circumstances different components may be used to construct the composite and, irrespective of its components, the composite will fulfill the same purpose in the application.

Creating composite objects is not simply a matter of creating a single entity but also involves creating all the component objects. The separation of the creation of a composite object from its use within the application provides design flexibility. By changing the method of construction of a composite object, alternative implementations may be introduced without affecting the current use.

Eg : Singleton Pattern

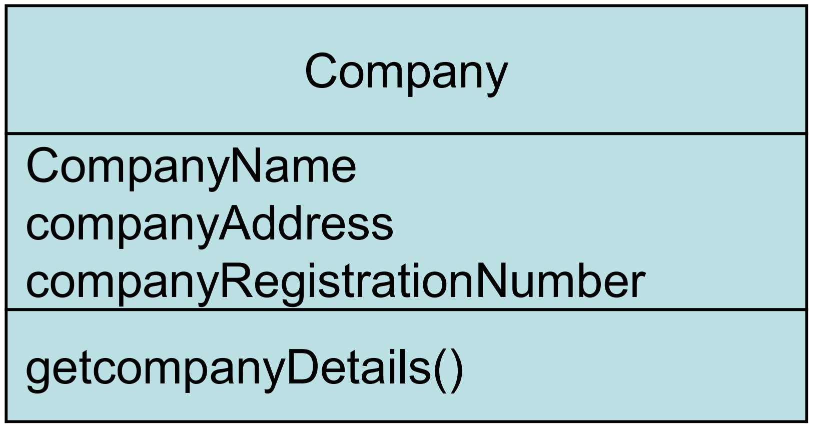
Singleton Pattern

Singleton pattern is one, which can be used to ensure that only one instance of a class is created. In order to understand the use of the pattern we need to consider the circumstances under which a single instance may be required.

The Agate campaign management system needs to hold information regarding the company. For example, its name, its head office address and the company registration details need to be stored so that they can be displayed in all application interfaces and printed on reports. This information should be held in only one place within the application but will be used by many different objects.

One design approach would be to create a global data area that can be accessed by all objects, but this violates the principle of encapsulation. Any change to the structure of the elements of global data would require a change to all objects that access them.

The creation of a Company class overcomes this problem by encapsulating the company attributes as shown below. These are then accessible to other objects through the operations of the Company object. But there is still a problem with this proposal. An object that wants to use the Company object needs to know the Company object's identifier so that it can send messages to it. This suggests that the Company object identifier should be globally accessible-but again this is undesirable since it violates encapsulation.



Some object-oriented programming languages provide a mechanism that enables certain types of operations to be accessed without reference to a specified object. These are called *class operations* or *static operations.* This offers a solution to the problem of providing global access without the need to globally define the object identifier.

For example, a static operation getCompanyInstance() can be defined in such a way that it will provide any client object with the identifier for the Company instance. This operation can be invoked by referencing the class name as shown below.

```
Company.getCompanyInstance()
```

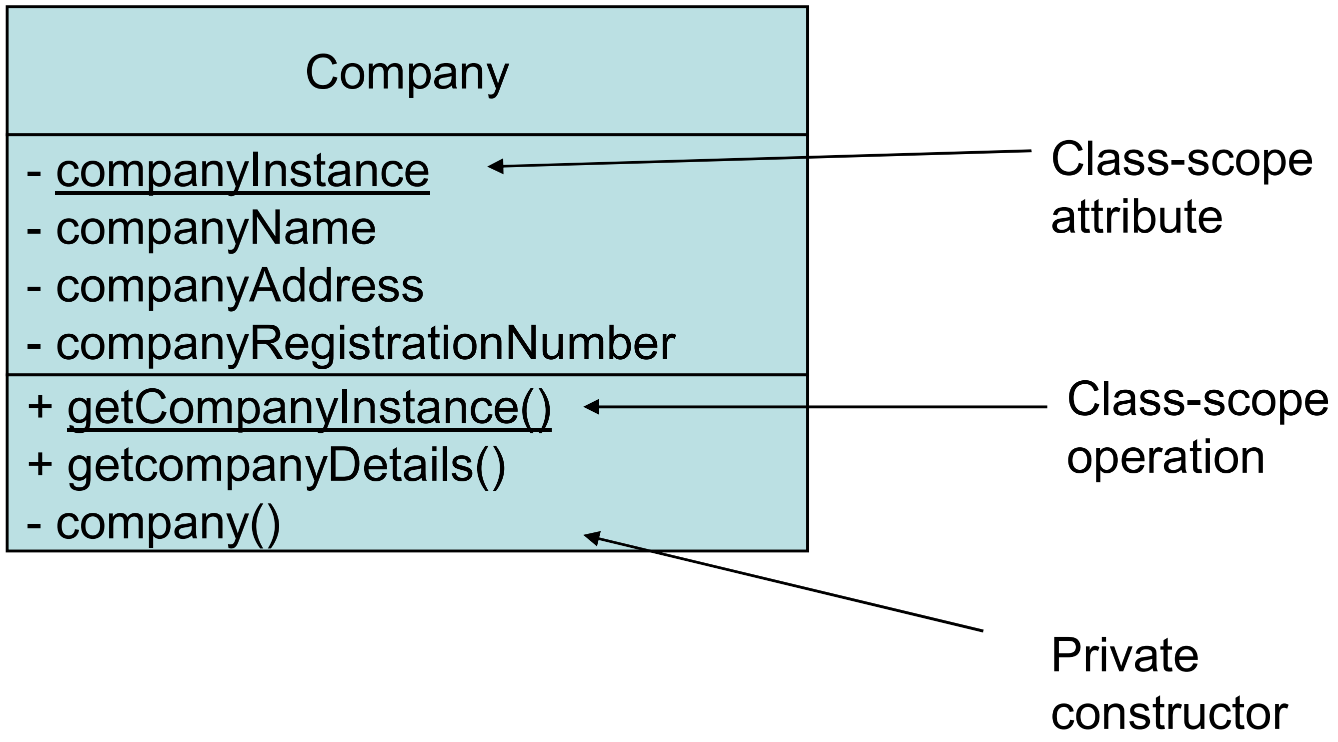
When a client object needs to access the Company object it can send this message to the Company class and receive the object identifier in reply. The client object can now send a getCompanyDetails () message to the Company object.

The design problem can be solved in the following way. It is important that there should only be one instance of this object. To ensure system integrity the application should be constructed so that it is impossible to create more than one. This aspect of the problem can be solved by giving the Company class sole responsibility for creating a Company object. This is achieved by making the class constructor private so that it is not accessible by another object. The next issue that needs to be addressed is the choice of an event that causes the creation of the company object. Perhaps the simplest approach is to create the Company object at the moment when it is first needed. When the Company class first receives the message getCompanyInstance() this can invoke the Company class constructor. Once the Company object has been created, the object identifier is stored in the class attribute companyInstance so that it can be passed to any future client objects.

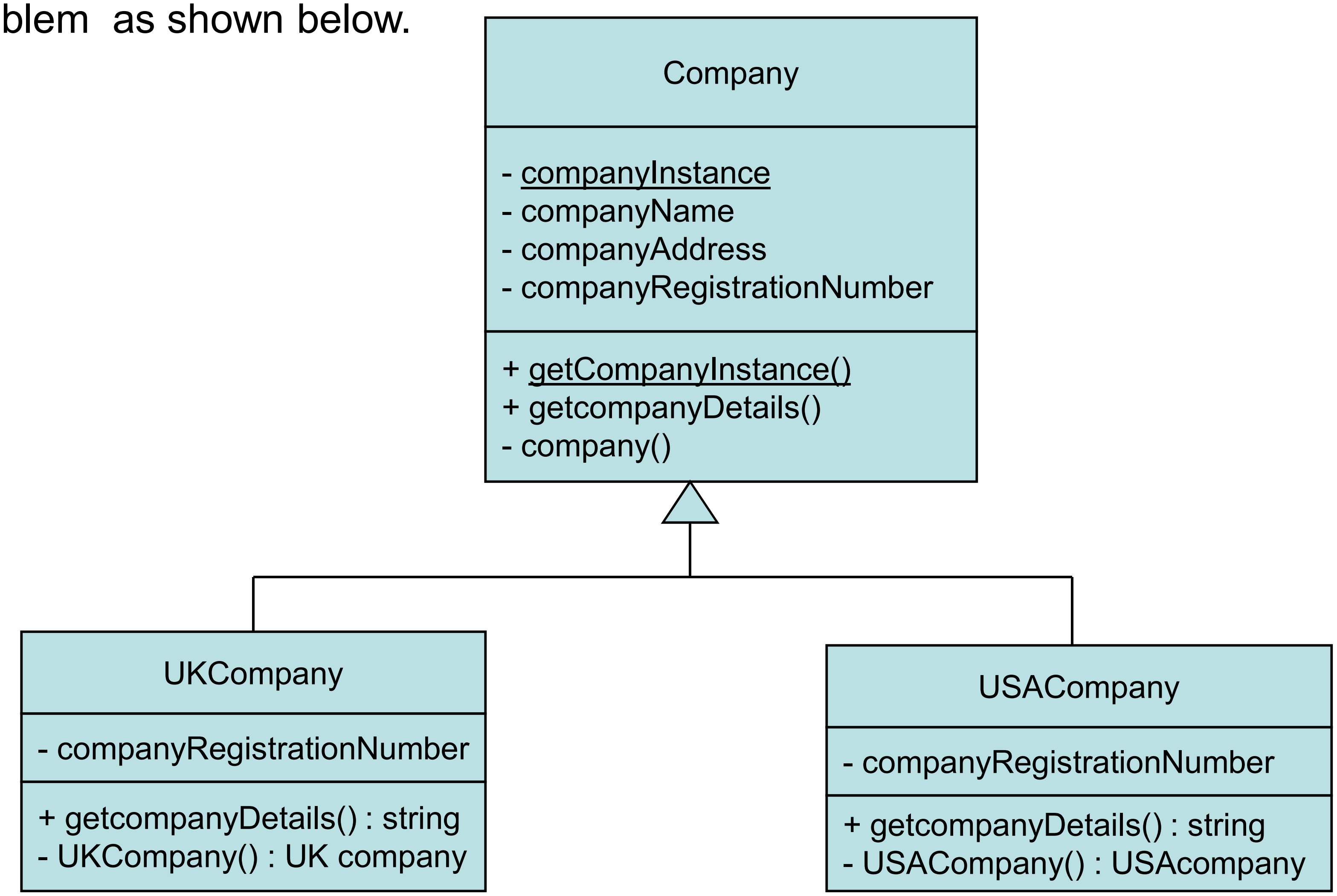
The following Company class provides a single global point of access via the class operation getCompanyInstance() and that also ensures that only one instance is created.

A simple version of the logic for the getCompanyInstance() operation is shown below.

```
            If (companyInstance == null)
{
companyInstance = new Company()
}   return companyInstance
```



The design may need to accommodate further requirements also. Since Agate operates as a separate company in each country , variations in company law from country to country may need different company registration details to be recorded for each country. This suggests a requirement for different types of Company class each with its own variation of the registration details. The creation of a separate subclass for each style of company registration details is a solution to this aspect ofthe problem as shown below.


Company
- companyInstance
- companyName
- companyAddress
- companyRegistrationNumber
+ getCompanyInstance()
+ getcompanyDetails()
- company()
UKCompany
- companyRegistrationNumber
+ getcompanyDetails() : string
- UKCompany() : UK company
USACompany
- companyRegistrationNumber
+ getcompanyDetails() : string
- USACompany() : USAcompany

The singleton pattern is described below in more general language.

Name. Singleton.

Problem. How can a class be constructed that should have only one instance and that can be accessed globally within the application?

Context. In some applications it is important that a class have exactly one instance. A sales order processing application may be dealing with sales for one company. It is necessary to have a Company object that holds details of the company's name, address, taxation reference number and so on. Clearly there should only be one such object. Alternative forms of a singleton object may be required depending upon different initial circumstances.

Forces. One approach to making an object globally accessible is to make it a global variable but in general this is not a good design solution as it violates encapsulation. Another approach is not to create an object instance at all but to use class operations and attributes . However, this limits the extensibility of the model since polymorphic redefinition of class operations is not possible in all development environments .

Solution. Create a class with a class operation getInstance( ) , which, when the class is first accessed, creates the relevant object instance and returns the object identity to the client. On subsequent accesses of the getInstance() operation no additional instance is created but the object identity of the existing object is returned.

The Advantages and Disadvantages of singleton patterns.

+ It provides controlled access to the sole object instance as the Singleton class encapsulates the instance.

+ The namespace is not unnecessarily extended with global variables.

+ The Singleton class may be subclassed. At system start-up user-selected options may determine which of the subclasses is instantiated when the Singleton class is first accessed.

+ A variation of this pattern can be used to create a specified number of instances if required.

- Using the pattern introduces some additional message passing. To access the singleton instance the class scope method has to be accessed first rather than accessing the instance directly.

- The pattern limits the flexibility of the application. If requirements change and as a result the singleton class may have many instances then accommodating this new requirement necessitates significant modification to the system.

- The singleton pattern is quite well known and developers are tempted to use it in circumstances that are inappropriate. Patterns must be used with care.

2. Structural patterns

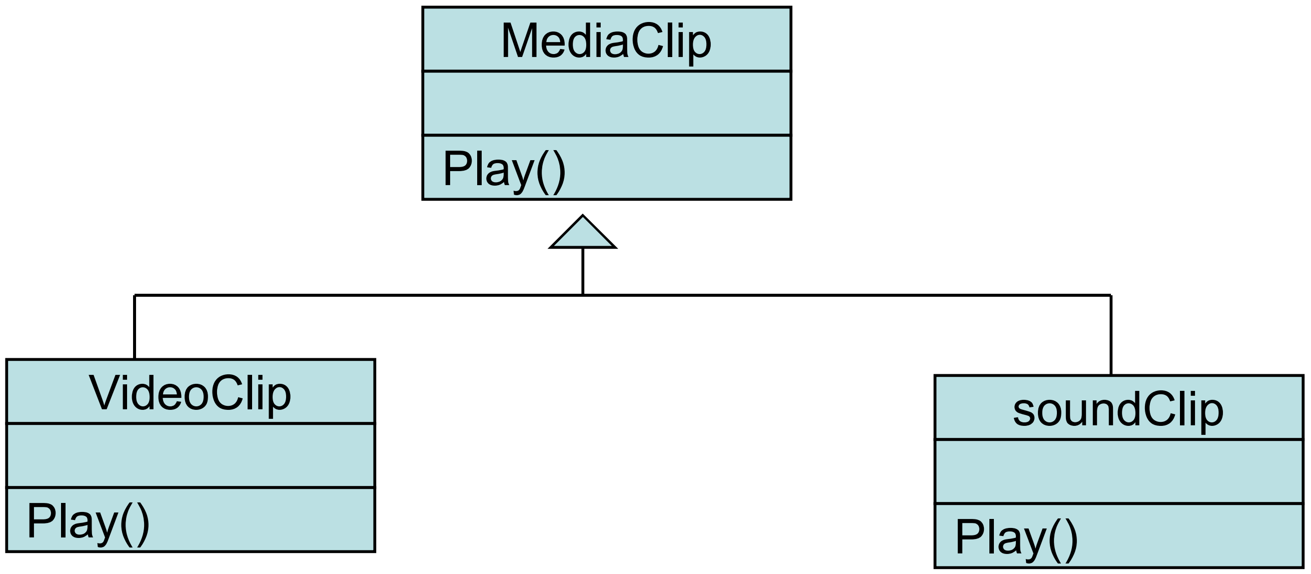
Structural patterns address issues concerned with the way in which classes and objects are organized. Structural patterns offer effective ways of using object-oriented constructs such as inheritance, aggregation and composition to satisfy particular requirements. If there , a requirement for a particular aspect of the application to be extensible. In order to achieve this, the application should be designed with constructs that minimize the sideeffects of future change. Alternatively it may be necessary to provide the same interface for a series of objects of different classes also.

Eg : Composite Pattern

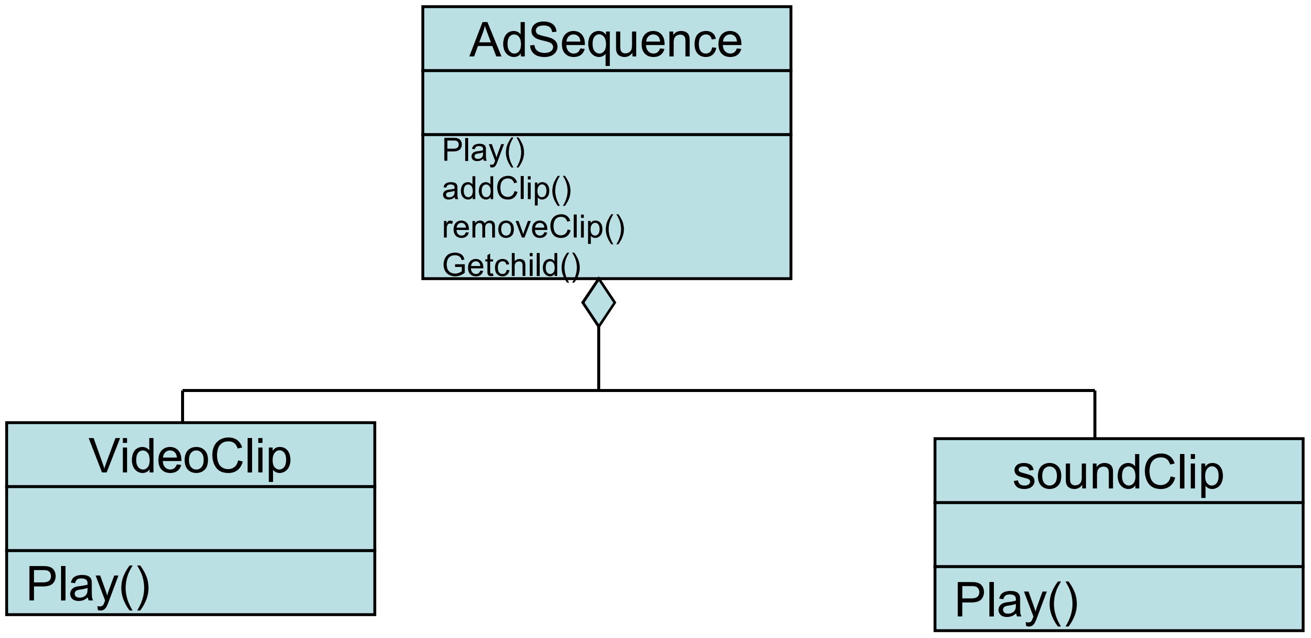
Composite Pattern

To apply Composite structural pattern in a design for the Agate case study , consider if further work is required to design a multimedia application that can store and play components of an advert.

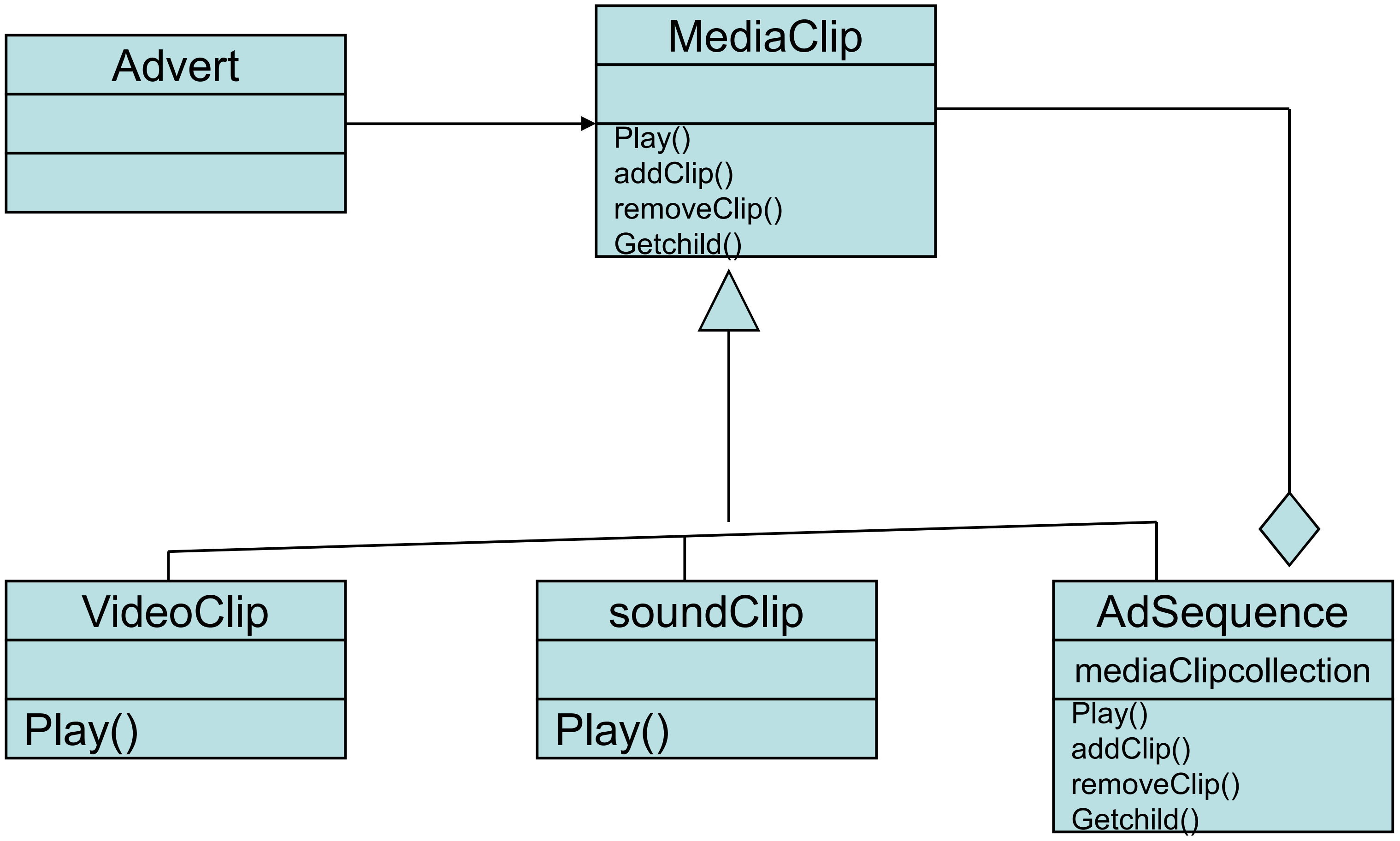
Here an advert is made up of sound clips and video clips each of which may be played individually or as part of an advert. The classes SoundClip and VideoClip have attributes and operations in common and it is appropriate that these classes are subclassed from MediaClip as shown below as MediaClip Inheritance hierachy.



But not all advert clips are primitive that is, made up of only a single MediaClip. Some consist of one or more sequences of clips, such that each sequence is in turn an aggregation of SoundClip and video Clip objects as shown below.



These two orthogonal hierarchies can be integrated by treating AdSequence both as a subclass of MediaClip and also as an aggregation of MediaClip objects as shown below. All the subclasses have the polymorphically redefined operation play( ) . For the subclasses VideoClip and SoundClip this operation actually plays the object. But for an AdSequence object, an invocation of the play() operation results in it sending a play() message to each of its components in turn.



This structure of the Composite pattern is described below.

Name. Composite.
Problem. There is a requirement to represent whole-part hierarchies so that both whole and part objects offer the same interface to client objects.
Context. In an application both composite and component objects exist that are required to offer the same behaviour. Client objects should be able to treat composite or component objects in the same way. A commonly used example for the composite pattern is a graphical drawing package. Using this software package a user can create atomic objects like circle or square and can also group a series of atomic objects or composite objects together to make a new composite object. It should be possible to move or copy this composite object in exactly the same way as it is possible to move or copy an individual square or a circle.
Forces. The requirement that the objects, whether composite or component, offer the same interface suggests that they belong to the same inheritance hierarchy. This enables operations to be inherited and to be polymorphically redefined with the same signature. The need to represent whole-part hierarchies indicates the need for an aggregation structure.
Solution. The solution resolves the issues by combining inheritance and aggregation hierarchies. Both subclasses, Leaf and Composite, have a polymorphically redefined operation anOperation (). The Composi te subclass also has additional operations to manage the aggregation hierarchy so that components may be added or removed.

3. Behavioural Patterns

Behavioural patterns addresses the problems that arise when assigning responsibilities to classes and when designing algorithms. Behavioural patterns specifies static relationships between objects and classes and how the objects of one class communicates with another. Behavioural patterns may use inheritance structures to spread behaviour across the subclasses or they may use aggregation and composition to build complex behaviour from simpler components.

Eg: The State pattern, uses both of these techniques.

State Pattern

Consider Agate case study to determine whether it has features that may satisfy the application of the state pattern or not. Identify whether there are any objects with significant state dependent behaviour or not. Among those Campaign objects will have behaviour that varies according to state , may be in any one of four main states, as shown below.

1.Commissioned State.

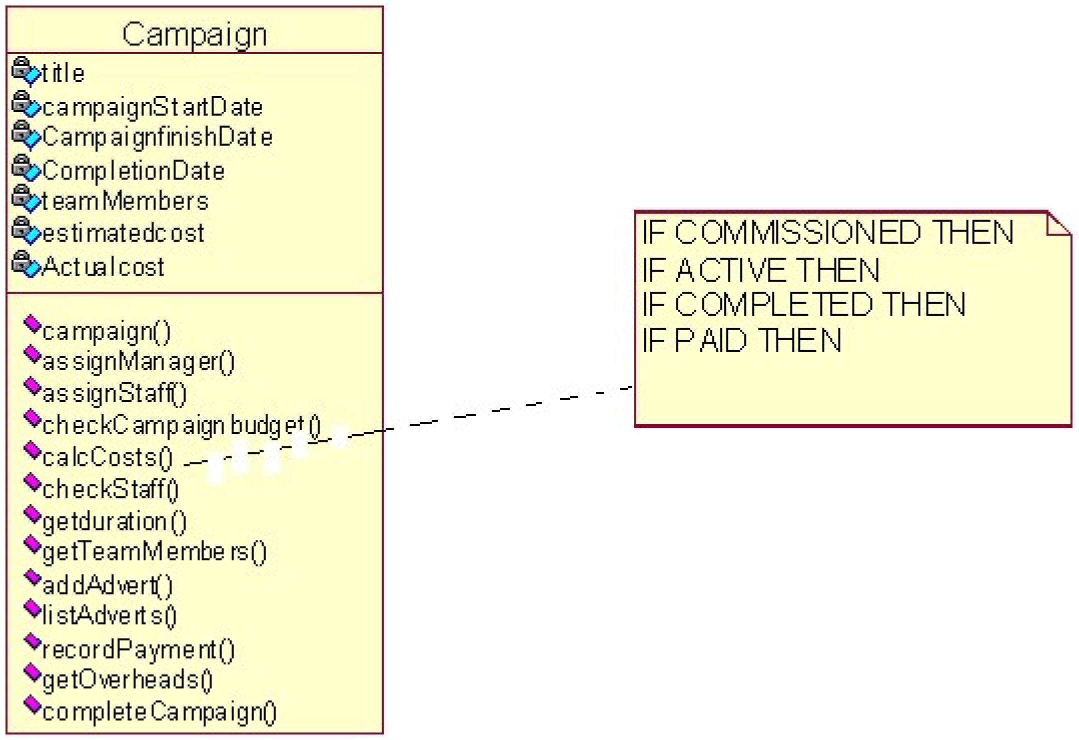
2.Active State.

3.Completed State

4.Paid State.

Clearly a Campaign object's state changes dynamically as the campaign progresses, thus necessitating changes in the behaviour ofthe object.

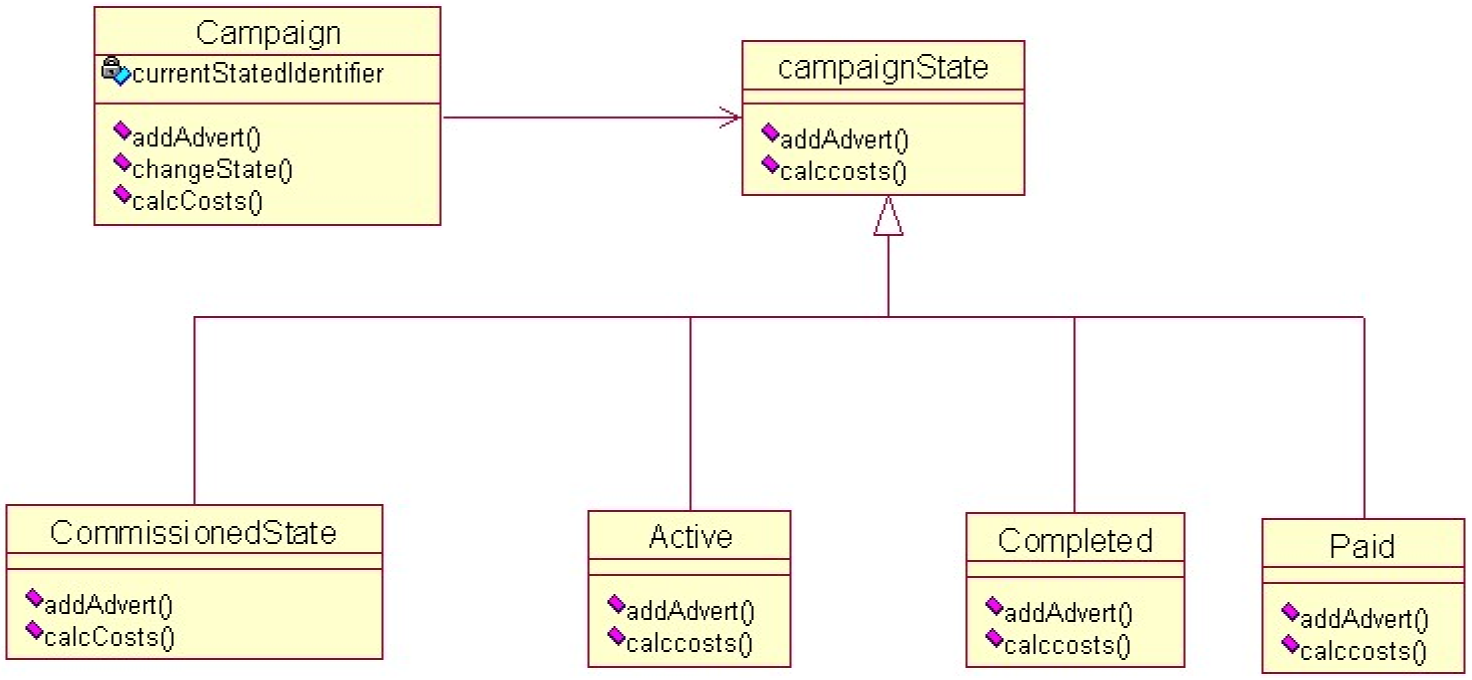
For example, when a campaign is planned a Campaign object is created in the Commissioned state. It remains in this state until a campaign budget has been agreed and only then does it become possible to run advertisements, although some preparatory work may be done for the campaign in the meantime. Once a Campaign object enters the Active state all advert preparation and any other work that is done is subject to an agreed billing schedule. Several operations, for example addAdvert ( ) and calcCosts (), will behave differently depending upon the state of the Campaign object. The following figure shows Campaign class with different operations including all states.



However, this would be a complex class which is further complicated by state dependent operations such as calcCosts (), which would need to be specified with a series of case or if-then-else statements to test the state of the object.

It would be simpler to subdivide the operations that have state dependent behaviour, which in this case would result in four separate calcCosts() operations, one for each state.

Another possibility is to create additional classes, one for each state so that each holds a state specific version of the operations, and this is how the State pattern works. A class diagram fragment illustrating this application of the State pattern is shown in the following figure.



The State pattern is described more generally below.

Name. State pattern.

Problem. An object exhibits different behaviour when its internal state changes making the object appear to change class at run-time.

Context. **In** some applications an object may have complex behaviour that is dependent upon its state. **In** other words the response to a particular message varies according to the object's state. One example is the calcCosts () operation in the Campaign class.

Forces. The object has complex behaviour that should be factored into less complex elements. One or more operations have behaviour that varies according to the state of the object. Typically the operation would have large, multi-part conditional statements depending on the state. One approach is to have separate public operations for each state but client objects would need to know the state of the object so that they could invoke the appropriate operation. For example four operations calcCosts Commissioned(), calcCostsActive(), calcCostsCompleted() and calcCostsPaid() would be required for the Campaign object. The client object would need to know the state of the Campaign object in order to invoke the relevant calcCosts( ) operation. This would result in undesirably tight coupling between the client object and the Campaign object. An alternative approach is to have a single public calcCosts () operation that invokes the relevant private operation (calcCosts Commissioned () would be private). However, the inclusion of a separate private operation for each state may result in a large complex object that is difficult to construct, test and maintain.

Solution: The state pattern separates the state dependent behaviour from the original object and allocates this behaviour to a series of other objects, one for each state. These state objects then have sole responsibility for that state's behaviour.
Advantages and Disadvantages of State pattern includes :

+ State behaviour is localized and the behaviour for different states is separated. This eases any enhancement of the state behaviour, in particular the addition of extra states.

+ State transitions are made explicit. The state object that is currently active indicates the current state of the Context object.

+ Where a state object has no attributes relevant to a specific Context object it may be shared among the Context objects.

- If the State objects cannot be shared among the Context objects each Context object will have to have its own State object thus increasing the number of objects and the storage requirements for the system.

- State objects may have to be created and deleted as the Context object changes state, thus introducing a processing overhead.

- Use of the State pattern introduces at least one extra message, the message from the Context class to the State class, thus adding a further processing overhead.

4. How to Use Design Patterns

The use of a pattern requires careful analysis of the problem that is to be addressed and the context in which it occurs. Before contemplating the application of patterns within a software development environment it is important to ensure that all members of the team receive appropriate training.

When a developer identifies a part of the application that may be subject to high coupling, a complex class or any other undesirable feature, there may be a pattern that addresses the difficulty.

The following are the issues to be considered before applying a pattern to resolve the problem.

- Is there a pattern that addresses a similar problem?
- Does the pattern trigger an alternative solution that may be more acceptable?
- Is there a simpler solution? Patterns should not be used just for the sake of it.
- Is the context of the pattern consistent with that of the problem?
- Are the consequences of using the pattern acceptable?
- Are constraints imposed by the software environment that would conflict with the use of the pattern?

Gamma et al. suggested seven-part procedure that should be followed after an appropriate pattern has been selected in order to apply it successfully.

1. Read the pattern to get a complete overview.
2. Study the Structure, Participants and Collaborations of the pattern in detail.
3. Examine the Sample Code to see an example of the pattern in use.
4. Choose names for the pattern's participants that is classes that are meaningful to the application.

5) Define the classes.
6) Choose application specific names for the operations.
7) Implement operations that perform the responsibilities and collaborations in the pattern.

A pattern should not be viewed as a prescriptive solution but rather as guidance on how to find a suitable solution. It is likely that a pattern will be used differently in each particular set of circumstances. At a simple level the classes involved will have attributes and operations that are determined by application requirements. Often a pattern is modified to accommodate contextual differences. Alternatively a pattern may suggest some other solution also to the developer.

It is critical to the use of patterns that pattern catalogues and languages should be made readily available to the developer. Many patterns are documented in hypertext on the Internet or on company Intranets. CASE tool support for patterns is developing and is provided by some vendors.

5.Benefits and Dangers of Using Patterns

One of the most important benefits of object-orientation is reuse. Reuse at the object and class level has proved more tangible than was initially expected. Patterns provide a mechanism for the reuse of generic solutions for object-oriented and other approaches. They provides a strong reuse culture. Within the design context, patterns suggest reusable elements of design and, most significantly, reusable elements of demonstrably successful designs. This reuse permits the transfer of expertise to less experienced developers so that a pattern can be applied again and again

Another benefit gained from patterns is that they offer a vocabulary for discussing the problem domain means whether it be analysis, design or some other aspect of information systems development at a higher level of abstraction than the class and object making it easier to consider micro-architectural issues. Pattern catalogues and pattern languages offer a rich source of experience that can be explored and provide patterns that can be used together to generate effective systems.

Some people believe that the use of patterns can limit creativity. Since a pattern provides a standard solution the developer may be tempted not to spend time on considering alternatives. The use of patterns in an uncontrolled manner may lead to over-design. Developers may be tempted to use many patterns irrespective of their benefits, thus rendering the software system more difficult to develop, maintain and enhance.

When a pattern is used in an inappropriate context the side effects may also occur with the system. For example, the use of the State pattern may significantly increase the number of objects in the application with a consequent reduction in performance.

Developers need to spend time understanding the relevant pattern catalogues, they need to be provided with easy access to the relevant catalogues and they need to be trained in the use of patterns. Another aspect of the introduction of patterns is the necessary cultural change. Patterns can only be used effectively in the context of an organizational culture of reuse.

These dangers emphasize that the use of patterns in software development requires care and planning. In this respect patterns are no different from any other form of problem solving: they must be used with intelligence. It is also important to appreciate that patterns only address some of the issues that occur during systems development.

16. Human-Computer Interaction

1. What is the user interface?

Users of an information system need to interact with it in some way.

Whether they are users of tele-sales system entering orders made over the telephone by customers, or members of the public using a touch screen system to find tourist information, they will need to carry out the following secondary tasks:

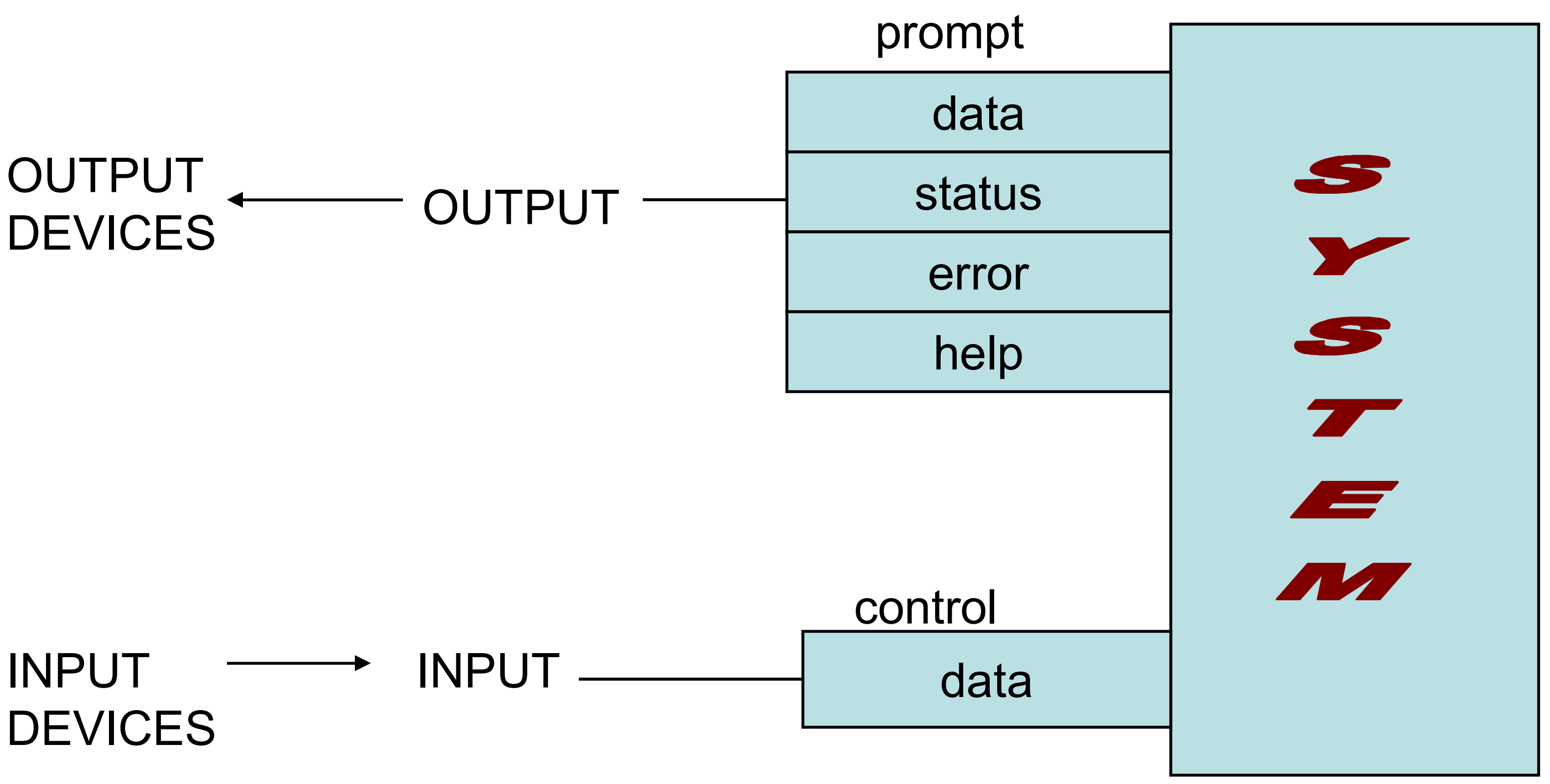
- read and interpret information that instructs them how to use the system;
- issue commands to the system to indicate what they want to do;
- enter words and numbers into the system to provide it with data to work with;
- read and interpret the results that are produced by the system either on screen or as a printed report;
- respond to and correct errors ;

In the above examples , the primary tasks are to take a customer order and to find tourist information. If the system has been designed well, the secondary, system-related tasks will be easy to carry out; if it has not been designed well, the secondary tasks will intrude into the process and will make it more difficult for the users to achieve their primary tasks.

The dialogue metaphor

In the design of many computer systems, interaction between the user and the system takes the form of a ***dialogue**.* The idea that the user is carrying on a dialogue with the system is a ***metaphor**.* There is no real dialogue in the sense of a conversation between people going on between the user and the computer, but as in dialogues between people, messages are passed from one participant to the other.

The following figure shows the human-computer dialogue in schematic form.



the following table describes what is meant by each of the types of message that can be found in this dialogue.

Output	prompt	request for user input
	data	data from application following user request
	status	acknowledgement that something has happened
	error	processing cannot continue
	help	additional information to user
Input	control	user directs which way dialogue will proceed
	data	data supplied by user

Types of Messages in Human-computer dialogue

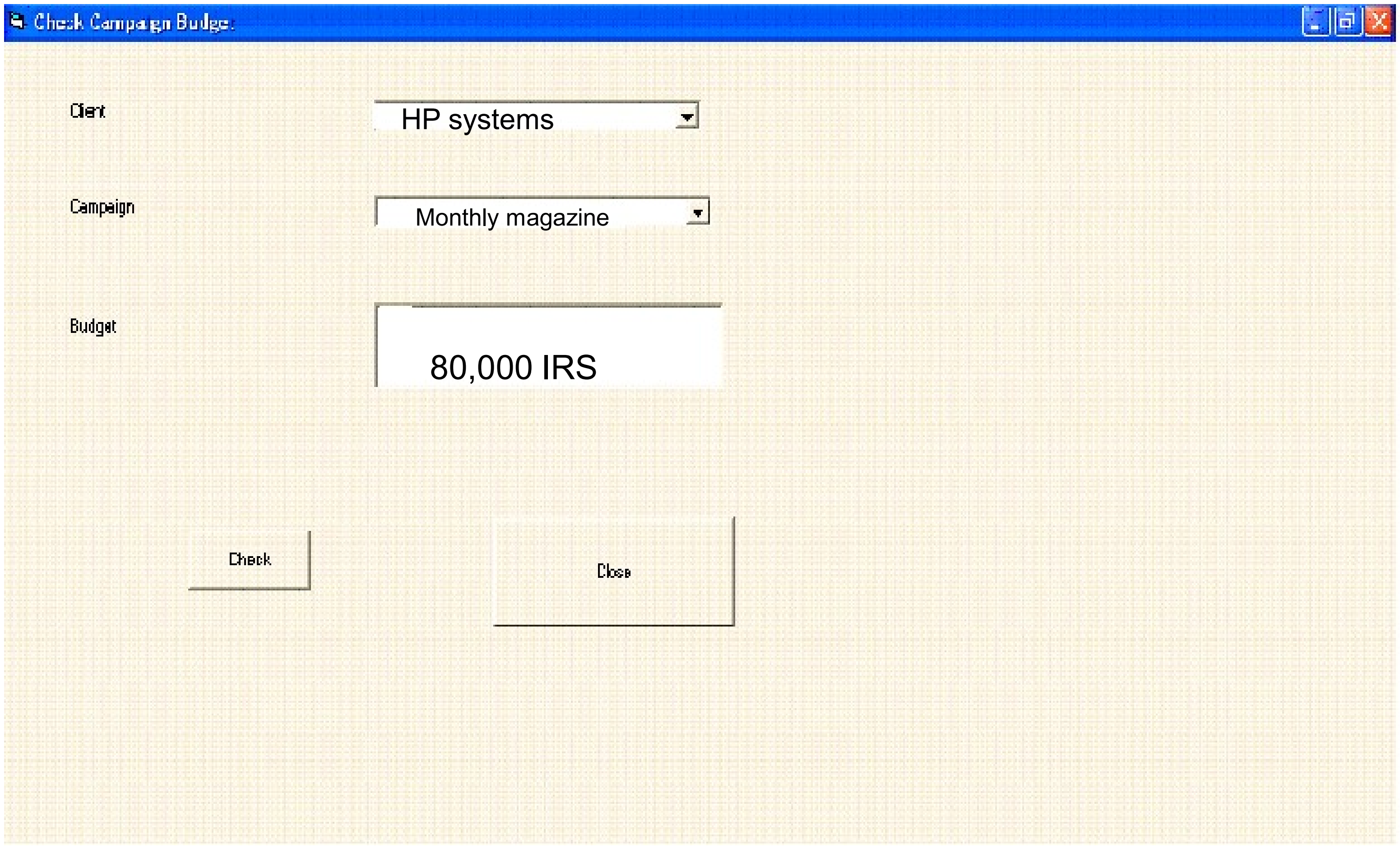
The direct manipulation metaphor

The other metaphor for the design of the user interface is the ***direct manipulation* metaphor**. Many people are now familiar with this through the use of GUIs. When you use a software package with this kind of interface you are given the impression that you are manipulating objects on the screen through the use of the mouse. This metaphor is reflected in the concrete nature of the terms that are used. The user of the system can:

- drag and drop an icon,
- shrink or expand a window,
- push a button and
- pull down a menu.

Such interfaces are *event-driven.* Graphical objects are displayed on the screen and the window management part of the operating system responds to events, Most such events are the results of the user's actions. The user can click on a button, type a character, press a function key, click on a menu item or hold down a mouse button and move the mouse. The design of user interfaces to support this kind of interaction is more complicated than for text-based interfaces using the dialogue metaphor.

The following figure shows the interface of a VB program to implement the use case Check campaign budget for the Agate case study.



HP systems

Monthly magazine

80,000 IRS

In this use case, the user first selects the name of a client from a list box labelled Client. Having selected the client, a list of all active campaigns for that client is placed in the list box labelled Campaign. At this point, no campaign is selected, and the user can click on the arrow at the end of the list box to view the list and select a campaign. When a campaign has been selected, the user can click on the button labelled Check. The program then totals up the cost of adverts in that campaign, subtracts it from the budget and displays the balance as a money value .In this interface design, there is no point in the user selecting a campaign until a client has been selected or clicking the Check button until a client and a campaign have been selected. The designer may choose to disable the Campaign list box until the client has been selected, and disable the button until both client and campaign have been selected. Having checked one campaign, the user may choose a different client, in which case the contents of the Campaign list box have to be changed and the button disabled again until a different campaign has been selected.

Windows like the one in the example above are usually called *dialogue boxes* in GUI environments. In terms of the metaphors, they combine elements of a dialogue with the user with direct manipulation of buttons and lists.

Characteristics of good dialogues

Regardless of whether a system is being developed for a text-based environment or for a GUI environment, there are a number of important general characteristics of good dialogue design. These includes the following :

- consistency,
- appropriate user support,
- adequate feedback from the system and
- minimal user input.

Consistency. A consistent user interface design helps users to learn an application and to apply what they know across different parts of that application. This applies to commands, the format for the entry of data such as dates, the layout of screens and the way that information is coded by the use of colour or highlighting.

Appropriate user support. When the user does not know what action to take or has made an error, it is important that the system provides appropriate support at the interface. This support can be informative and prevent errors by providing help messages, or it can assist the user in diagnosing what has gone wrong and in recovering from their error. Help messages should be context-sensitive. This means that the help system should be able to detect where the user has got to in a dialogue and provide relevant information.

In a GUI environment, this means being able to detect which component of the interface is active and providing help that is appropriate to that part of the interface. The help provided may be general, explaining the overall function of a particular screen or window, or it may be specific, explaining the purpose of a particular field or graphical component and listing the options available to the user. It may be necessary to provide a link between different levels of help so that the user can move between them to find the information they require. Help information may be displayed in separate screens or windows, or it may be displayed simultaneously in a status line also or using *tooltips* as the user moves through the dialogue or positions the cursor over an item. Many web page designers provide help about elements of their pages by dis-playing messages in the status line at the bottom of the browser window or by displaying a tooltip-style message in a box as the cursor moves over an item on the page.

This way of displaying Warning messages can prevent the user from making serious errors by providing a warning or caution message before the system carries out a command from the user that is likely to result in an irreversible action. Warning messages should allow the user to cancel the action that is about to take place.

Adequate feedback from the system. Users expect the system to respond when they make some action. If they press a key during data entry, they expect to see the character appear on the screen ; if they click on something with the mouse, they expect that item to be highlighted and some action from the system. Users who are uncertain whether the system has noticed their action keep on pressing keys or clicking with the mouse, with the result that these further key presses and clicks are taken by the system to be the response to a later part of the dialogue, with unpredictable results. It is important that users know where they are in a dialogue or direct manipulation interface: in a text-based interface there should be a visible cursor in the current active field; in a GUI environment the active object in the interface should be highlighted.

Minimal user input. Users resent making what they see as unnecessary keypresses and mouse clicks. Reducing unnecessary input also reduces the risk of errors and speeds data entry. The interface should be designed to minimize the amount of input from the user. The user can be helped in this way by:

- using codes and abbreviations,
- selecting from a list rather than having to enter a value,
- editing incorrect values or commands rather than having to type them in again,
- not having to enter or re-enter information that can be derived automatically
- Using default values

Style guides

Guidelines for user interface design are usually referred to as *style guides,* and large organizations with many different information systems produce their own style guides for the design of systems to ensure that all their applications, whether they are produced in-house or by outside software companies, conform to a standard set of rules that will enable users quickly to become familiar with a new application. The use of style guides and the characteristics of a good dialogue relate to dialogue and interface design in general.

For example Microsoft produces a book of guidelines called ***The Windows Interface Guidelines for Software Design*** that lays down the standards to which developers must follow if they are to be granted Windows certification.

2. Approaches to USERINTERFACE Design

There are many different ways of designing and implementing the elements of the user interface that support the interaction with users by using formal and informal approaches.

The following are the factors to be considered while designing User Interface .

- the nature of the task that the user is carrying out,
- the type of user,
- the amount of training that the user will have undertaken,
- the frequency of use
- the hardware and software architecture of the system.

These factors may be very different from system to systems which are listed in the following table. The following are the factors for the tele-sales system and a WAP tourist information system. Systems that are used by members of the public are very different from information systems used by staff.

	Tele-Sales System	WAPTourist Information System
The nature of the task that the user is carrying out	Routine task; closed solution; limited options.	Open-ended task; may be looking for information that is not available.
The type of user	Clerical user of the system; no discretion about use	Could be anyone; discretion about use of system; novice in relation to this system.
The amount of training that the user will have undertaken	Training provided as part of job.	No training provided.
The frequency of use	Very frequent; taking an order every few minutes.	Very occasional; may never use it again.
The hardware and software architecture of the system	Mini-computer, dumb terminals with text screens, keyboard data entry. All software runs on the mini-computer. Structured programs with subroutines for data access and screen-painting.	Mobile telephone screen with keypad and scroll buttons to move through menus. WAP browser runs on mobile telephone, WAP gateway connects to server, which generates WML for WAP browsers and HTML for other browsers using XML and stylesheets.

The following are the three types of more formal and methodical approaches to the analysis of usability requirements to be considered for design of HCI User Interfaces.

1. Structured approaches
2. Ethnographic approaches
3. Scenario-based approaches.

These approaches are very different from one another. However, they all carry out three main steps in **HCI** design:

1. requirements gathering,
2. design of the interface and
3. interface evaluation.

1. Structured approaches

Structured approaches to user interface design have been developed in response to the growth in the use of structured approaches to systems analysis and design. Structured analysis and design methodologies have a number of characteristics. They are based on a model of the systems development life cycle, which is broken down into stages, each of which is further broken down into steps that are broken down into tasks. Specific analysis and design techniques are used, and the methodology specifies which techniques should be used in which step. Each step is described in terms of its inputs , the techniques applied and the deliverables that are produced as outputs (diagrams and documentation).

These approaches are more structured than the simple waterfall model of the life cycle, as they provide for activities being carried out in parallel where possible rather than being dependent on the completion of the previous step or stage.

Structured approaches uses data flow diagrams to model processes in the system and take a view of the system that involves decomposing it in a top-down way. Structure charts or structure diagrams are used to design the programs that will implement the system.

Benefits of Structured approaches :

- They make management of projects easier. The breakdown of the project into stages and steps makes planning and estimating easier, and thus assists management control of the project.

- They provide for standards in diagrams and documentation that improves understanding between the project staff in different roles means between analyst, designer and pro-grammer.

- They improve the quality of delivered systems. Because the specification of the system is comprehensive, it is more likely to lead to a system that functions correctly.

Structured approaches make use of diagrams to show the structure of tasks and the allocation of tasks between users and the system. They also make extensive use of checklists in order to categorize the users, the tasks and the task environments. Evaluation is typically carried out by assessing the performance of the users against measurable usability criteria.

The following are the two examples of structured approaches.
1. STUDIO (STructured User-interface Design for Interface Optimization)
2. The RESPECT User Requirements Framework

STUDIO is divided into Stages, and each Stage is broken down into Steps. The activities undertaken in each of the Stages are shown below :

Stage	**Summary of activities**
Project Proposal and Planning	Decide whether user interface design expenditure can be justified. Produce quality plan.
User Requirements Analysis	Similar to systems analysis, with focus on gathering information relating to user interface design rather than general functionality .
Task Synthesis	Synthesize results of requirements analysis to produce initial user interface design. Produce user support documentation.
Usability Engineering	Prototyping combined with impact analysis to provide an approach to iterative development that is easy to manage.
User Interface Development	Handover of the user interface specification to developers to ensure that usability requirements are understood.

STUDIO uses a number of techniques such as :

- task hierarchy diagrams,
- knowledge representation grammars,
- task allocation charts, and
- state charts.

2. Ethnographic approaches

The term ethnography is applied to a range of techniques used in sociology and anthropology and reflects a particular philosophy about how scientific enquiry should be carried out.

In HCI this means that the professional charged with carrying out the user interface design spends time with the users immersed in their everyday working life. Only by spending time in this way can the real requirements of the users be understood and documented.

Ethnographic methods also concentrates on how different users interpret their experience of using systems subjectively, and it is this subjective interpretation that the HCI professional must understand rather than assuming that the system can be assessed objectively.

Ethnographic approaches use a range of techniques to capture data: interviews, discussions, prototyping sessions and videos of users at work or using new systems. These data are analysed from different perspectives to gain insights into the behaviour of the users.

3. Scenario-based approaches

Scenario-based design has been developed by John Carroll and others . It is less formal than the structured approaches but more clearly defined than most ethnographic approaches. Scenarios are step-by-step descriptions of a user's actions that can be used as a tool in requirements gathering, interface design and evaluation. Use cases are similar to scenarios.

Among these three approaches, scenario-based design fits best with use case modelling. Scenarios can be textual narrative describing a user's actions or they can be in the form of storyboards (a series of pictures that depict those actions), video mock-ups or even prototypes.

Scenarios can be used for requirements gathering to document the actions that a user carries out in their current system. They can also be used to document ideas about how the user would see themselves using the new system. This is called envisioning the design. Alternative scenarios describing different approaches to the design can be compared by the designers and the users.

Scenarios provide a means of communication that can be used by professionals and end-users to communicate about the design of the users' interaction with the system. They are simple enough that users can produce them without the need for the kind of training that they would need to understand class diagrams, for example. Scenarios can be used with use cases. The use cases can provide a description of the typical interaction;

scenarios can be used to document different versions of the use case, for example, to document what happens when a user is adding a new note but is not authorized to work on the project they try to add it to. Use cases are concerned with the functionality offered by the system, while scenarios focus on the interaction between the user and the system.

Scenario based design can result in large volumes of textual information that must be organized and managed so that it is easily accessible. There is a document management task to be undertaken that requires a rigorous approach to control different versions of scenarios and to cross-reference them to claims and feedback from users.