19. IMPLEMENTATION

Implementation might be considered outside the scope of analysis and design. However, in projects that use rapid application development techniques, the distinction between different roles tends to break down . Analysts may have a role during implementation in dealing with system testing, data and user training also.

1. **Software Implementation -**

Software tools :

The implementation of a system will require a range of tools. Ensuring that these are available in compatible versions and with sufficient licences for the number of developers who will be using them is part of the project management role. Many of these tools have been designed and developed to make the work of the system developer easier. The following are different types of tools to be considered for Implementation.

CASE tools

Computer-aided software engineering tools allow the analysts and designers to produce the diagrams that make up their models of the system. The repository for the project should be maintained using the CASE tool to link the textual and structured descriptions of every class, attribute, operation, state and so on to its diagrammatic representation.

To ensure that the implementation accurately reflects the design diagrams, it may be possible to generate code in one or more programming languages from the documentation in the CASE tool. CASE tools exist that generate code for languages such as Visual Basic, C++ and Java. Some support the generation of SQL statements to create relational database tables to implement data storage, and the generation of the CORBA IDL for a distributed system. Some CASE tools provide support for reverse engineering from existing code to design models. This implementation is combined with code generation, so it is known as round-trip engineering.

Compilers, interpreters and run-time support

Irrespective of the language used in the implementation , some kind of compiler or interpret required to translate the source code into executable code.

- c++ must be compiled into object code that can be run on the target machine.
- Java is compiled into an intermediate bytecode format and requires a run-time program to enable it to execute. Fo this run-time program is provided in the web browser, otherwise it is provided by the program called simply java or java.exe.
- C# can be compiled into bytecode in Intermediate Language (MSIL) format for .NET applications.

Visual editors

Graphical user interfaces can be extremely difficult to program manually. Since the advent of Visual Basic, visual development environments have been produced a wide range of languages. These enable the programmer to develop a user interface by dragging and dropping visual components onto forms and setting the parameters that control their appearance in a properties window.

Integrated development environment

Large projects involve many files containing source code and other information as the resource files for prompts in languages. Keeping track of all these files and the dependencies between them, and recompiling all those that have changed as a project is being built is a task best performed by software designed for that purpose.

Integrated development environments (IDEs) incorporate a multi-window editor, mechanisms for managing the files that make up a project, links to the compiler so that code can be compiled from within the IDE and debugger to help the programmer step through the code to find errors. An IDE may also include a visual editor to help build the user interface and a version control system to keep track of different versions of the software.

Configuration management

Configuration management tools keep track of the dependencies between components and the versions of source code and resource files that are used to produce a particular release of a software package. Each time a file is to be changed, it must be check of a repository. When it has been changed it is checked in again as a new version. The tool keeps track of the versions and the changes from one version to the next. software release is built, the tool keeps track of the versions of all the files that were used in the build. To ensure that an identical version can be rebuilt, other tools such as compilers and linkers should also be under version control.

Class browsers

In an object-oriented system, a browser provides a visual way of navigating the class hierarchy of the application and the supporting classes to find their attributes and operations.

Component managers

Component managers provide the user with the ability to search for suitable components, to browse them and to maintain different versions of components for re-usability purpose.

DBMS

A large-scale database management system will consist of a considerable amount of software. If it supports a client-server mode of operation, there will be separate client and server components . To use ODBC or JDBC will require ODBC software installed on the client. For any database, special class libraries or Java packages may be required on the client either during compilation or at run-time or both.

CORBA

An ORB is required in order to use CORBA. It will include the IDL compiler that takes interface definitions in IDL and produces the interface, stub and skeleton files necessary to use CORBA.

Testing tools

Automated testing tools are available for some environments. What is more likely is that programmers will develop their own tools to provide harnesses within which to test classes and sub-systems.

Conversion tools

In most cases data for the new system will have to be transferred from an system. Whereas once the existing system was usually a manual system, most now a days replace an existing computerized system, and data will have to be extracted from files or a database in the existing system and reformatted so that it can be set up the database for the new system. Packages like Data Junction provide automated mated tools to extract data from a wide range of systems and format it for system.

Documentation generators

In the same way that code can be generated from the diagrams and documents in a CASE tool, it may be possible to generate technical and user documentation. In Windows there are packages such as Documentation Studio that can be produce files in Windows Help format. Java includes a program called javadoc that processes Java source files and builds **HTML** documentation in the style of the API documentation from special comments with embedded tags in the source code.

Coding and documentation standards

On any project in which people collaborate to develop software, standards for the naming of classes, attributes, operations and other elements system are essential. Naming standards should have been agreed before the analysis began as shown below.

- Classes are named with an initial capital letter. Words are concatenated together when the class name is longer than one word. Capital letters within the name show where these words have been joined together. Eg : SalesOrderProxy.

- Attributes are named with an initial lower case letter. The same approach is taken as for classes by concatenating words together. Eg: customerOrderRef.

- Operations are named in the same way as attributes. Eg: getOrderTotal()

Consistent naming standards also make it easier to trace requirements from an through design to implementation. This is particularly important for class, attri and operation names.

The following are the reasons for documenting code.

- Think of the next person. Someone else may be maintaining the code you written.
- Your code can be an educational tool. Good code can help others, but without comments complicated code can be difficult to understand.
- No language is self-documenting. However good your naming conventions, you can always provide extra help to someone reading your code.
- You can automate its production means by using javadoc program generates HTML code from your comments.

2. Component diagrams

Two types implementation diagrams in UML Terminology are

1. Component Diagrams
2. Deployment diagrams

In a large project there will be many files that make up the system. These files will have dependencies on one another. The nature of these dependencies will depend on the language or languages used for the development and may exist at compile-time, at link-time or at run-time. There are also dependencies between source code files and the executable files or bytecode files that are derived from them by compilation. Component diagrams are one of the two types of implementation diagram in UML. Component diagrams show these dependencies between software components in the system. Stereotypes can be used to show dependencies that are specific to particular languages also.

A component diagram shows the allocation of classes and objects to components in the physical design of a system. A component diagram may represent all or part of the component architecture of a system along with dependency relationships.

The dependency relationship indicates that one entity in a component diagram uses the services or facilities of another.

- Dependencies in the component diagram represent compilation dependencies.
- The dependency relationship may also be used to show calling dependencies among components, using dependency arrows from components to interfaces on other components.

Different authors use component diagrams in different ways.

Fowler suggests that components correspond exactly to packages of the system.

Muller suggests that packages contain components as well as other packages.

Strictly in UML packages should be used for model management: for organizing models into convenient parts that contain types of diagram or sub-systems.

Here We have the following distinction between them.

- Components in a component diagram should be the physical components of a system.

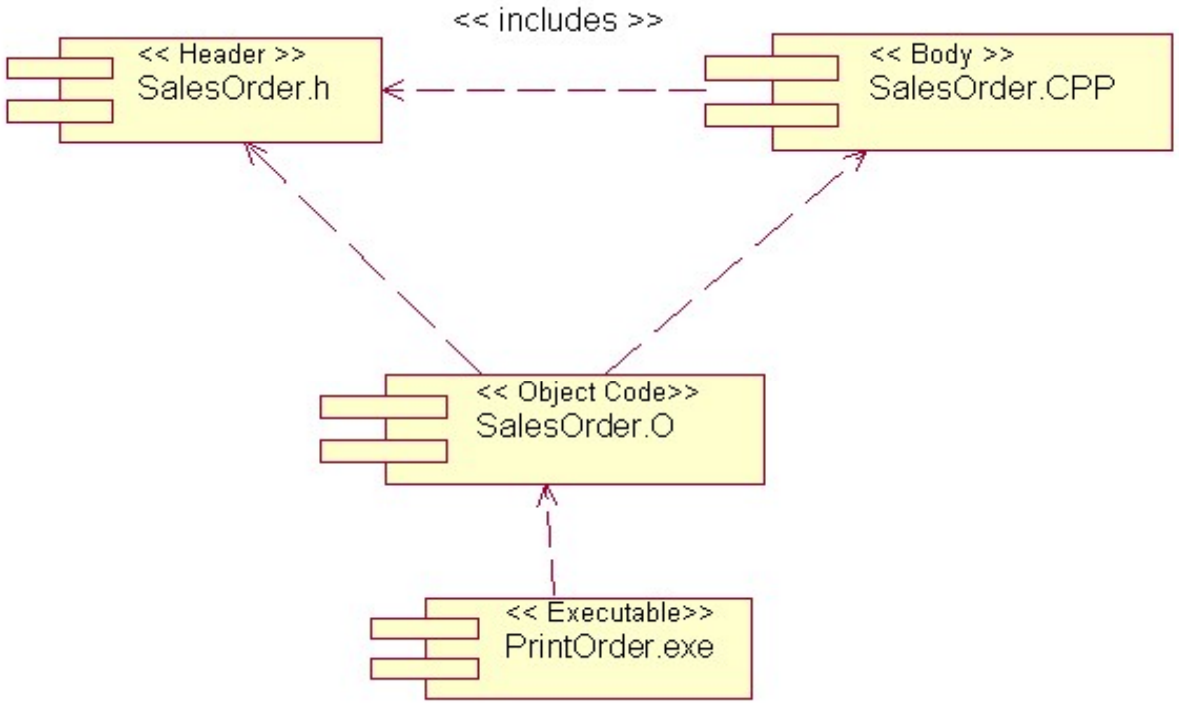
- During analysis and the early stages of design, package diagrams can be used to show the logical grouping of class diagrams or of models that use other kinds of diagrams into packages relating to sub-systems.

- During implementation, package diagrams can be used to show the grouping of physical components into sub-systems .

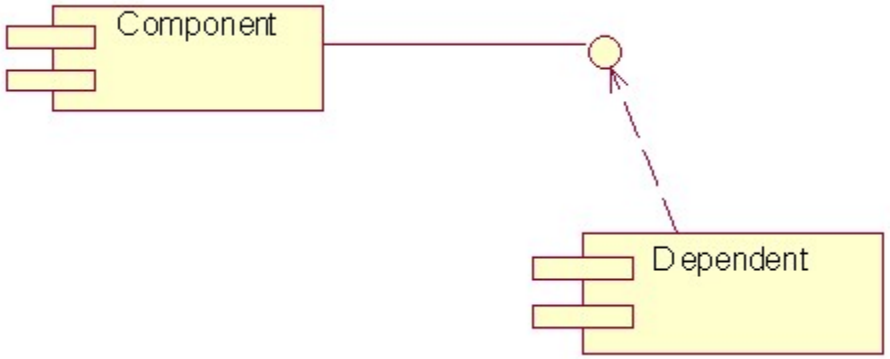
- component diagrams can also be combined with deployment diagrams to show the physical location of components of the system. The classes in one logical package may be distributed across physical locations in a physical system, and the component diagram and deployment diagram can be used to show this.

  If component diagrams are used , it is better to keep separate sets of diagrams to show compile-time and run-time dependencies. However, this is likely to result in a large number of diagrams. Component diagrams show the components as types. If you wish to show instances of components you can use a deployment diagram.

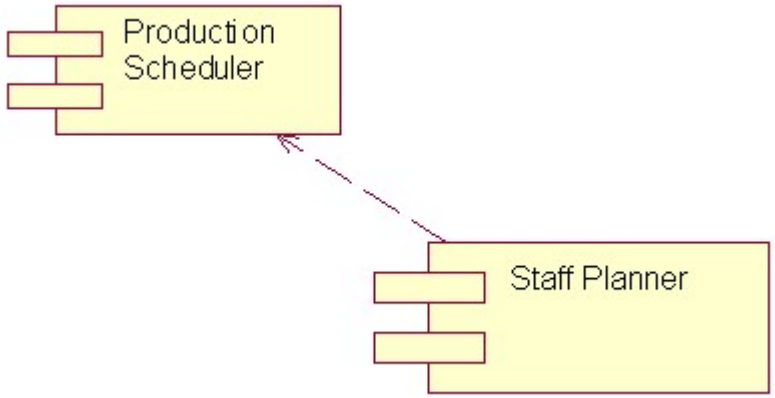
The following figure shows a component diagram that represents the dependency of a C++ source code file on the associated header file, the dependency of the object file on both and the dependency of an executable on the object file. Stereotypes can be used to show the types of different components.



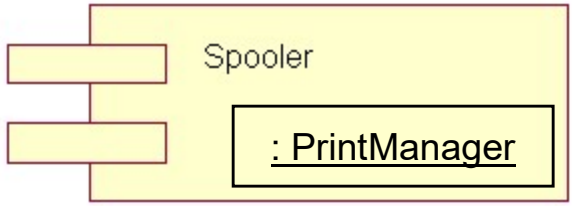
An alternative representation of part of the diagram above is to use the UML interface notation to show the specification of a class (the header file in C++) as an interface and the body as the component, which is shown below.



This notation can be used in Java to show the dependency of classes on the interfaces of other classes. This is particularly appropriate for distributed systems using CORBA in which applications running on a client are dependent on the interfaces of classes that are actually implemented on other machines on the network. Component diagrams need not be used at this low level, but can be used to show dependencies between large-scale components within a system as shown below.



In Java, component diagrams can be used to show the dependency of classes on packages that contain the classes that they import also. This is particularly important in a language such as Java where the availability of packages of classes at run-time is critical to the running of a program. Active objects, typically processes running on a separate thread, can be shown in a component diagram. An example of this is shown below.



3. Deployment diagrams :

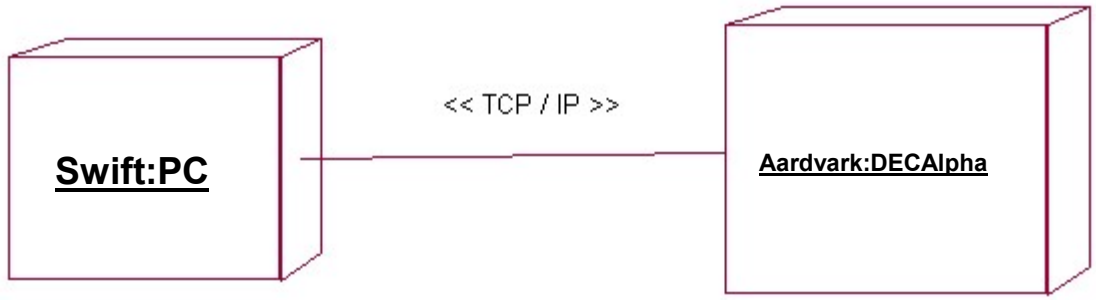
Two types implementation diagrams in UML Terminology are

1. Component Diagrams
2. Deployment diagrams

The second type of implementation diagram provided by UML is the deployment diagram. Deployment diagrams are used to show the configuration of run-time processing elements and the software components and processes that are located on them.

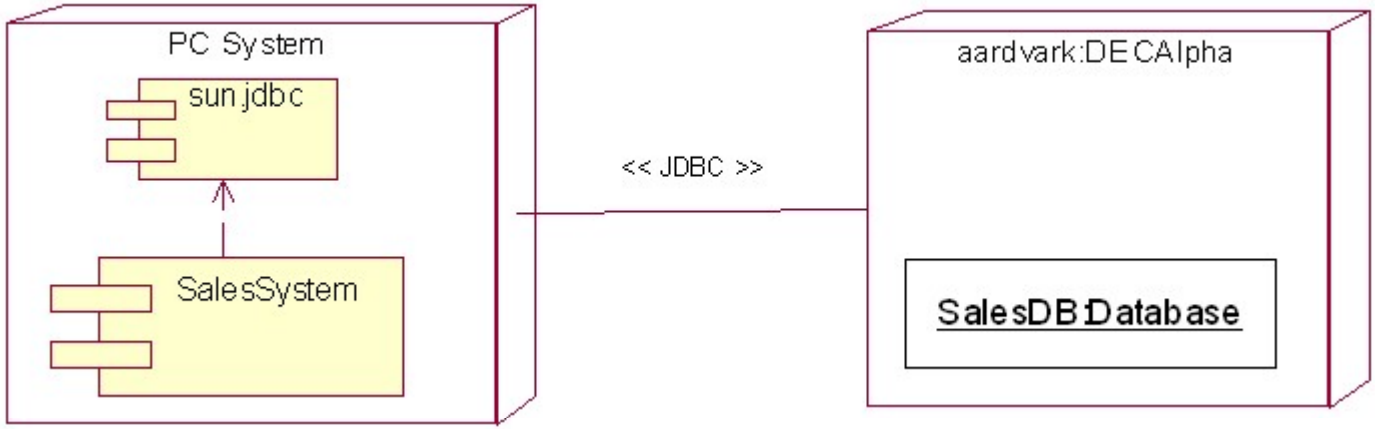
Deployment diagrams are made up of nodes and communication associations. Nodes are typically used to show computers and the communication associations show the network and protocols that are used to communicate between nodes. Nodes can be used to show other processing resources such as people or mechanical resources.

Nodes are drawn as 3D views of cubes or rectangular prisms, and the following figure shows a simplest deployment diagram where the nodes connected by communication associations .



Deployment diagrams can show either types of machine or particular instances as in the above figure **swift** is the name of a PC.

Deployment diagrams can be shown with components and active objects within the nodes to indicate their location in the run-time environment also. The following figure shows the location of the Sales database on the server and some components on client PCs.



Deployment diagrams show the physical architecture of the system. If all the components of a system in deployment diagrams are represented within the component then they are likely to become very large or difficult to read. They can serve the purpose of communicating information about the location of key components to other members of the team or to users. Generally , most of the computer professionals will have to draw an informal diagram like this at some time in their working lives to show where different parts of a system are to be located.

If purpose of component and deployment diagrams is to illustrate principles about the way that the new system will be structured, then they are fine as diagramming technique. However, if the aim of drawing implementation diagrams is provide a complete specification of the dependencies between components at compile time and run-time and the location of all software components in the implementation system.

Having implemented a system in Java that uses class files, classes from a visual editor, JDBC drivers, CORBA and ObjectStore Persistent Storage Engine on PCs, and a Oracle on a workstation, we know that keeping track of all these dependencies and documenting which components have to be on which machines is not a trivial task. For most systems, this information may be easier to maintain in a tabular fomat and a spreadsheet may be the best way of doing this.

Component diagrams can be replaced by a table that shows a list of all the software components down the rows and the same list across the top of the columns. It may be best to keep up to three tables for compile-time, link-time and run-time dependencies. For each case where a component is dependent on another, place a mark where the row of the dependent component intersects with the column of the component on which it is dependent.

A simple example of this way of representation in the form of tabular format separately for each is shown below.

Campaign Database – Compile Time dependencies				
	JDBC Sun.jdbc.*	Campaign. java	Campaign. broker.java	Campaign.proxy. java
JDBC Sun.jdbc.*				
Campaign. java				
Campaign. broker.java	✓	✓		
Campaign. proxy.java		✓	✓	

In the same way, deployment diagrams can be replaced by a table that lists components down the rows and either types of machines or particular instances across the top of the columns. A mark is entered in the row-column intersection for every component that has to be on a particular machine or type of machine. If the exact location of components in a directory structure is important, then that location can be entered into the table.

This is shown in the following figure. Later this will form the basis of the information required for installing software onto users machines for testing and evntual deployment.

Campaign Databse – Run – time locations		
	Client Pc	Database Server
JDBC.sun.jdbc.*	C:\jdbc	
Campaign.class	C: \Agate\campaign	
CampaignBroker.class	C: \Agate\campaign	
CampaignProxy.class	C: \Agate\campaign	
SQL * Net		✓
OCI Listener		✓

4. SOFTWARE TESTING

Who carries out the testing?

One view of testing is that it is too important to be left to the programmers who have developed the software for the system. Here important fact is testing is carried out by someone whose assessment of the software will be objective and impartial. It is often difficult for programmers to see the faults in the program code that they have written. An alternative is provided by Extreme Programming (XP).

XP is an approach to rapid application development in which programmers are expected to write test harnesses for their programs before they write any code. Every piece of code can then be tested against its expected behaviour, and if a change is made can easily be retested.

The analysts who carried out the initial requirements analysis will be involved in testing the system as it is developed. The analysts will have an understanding of the business requirements for the system and will be able to measure the performance of the system against functional and non-functional requirements.

The systems analysts will use their knowledge of the system to draw up a test .This will specify what is to be tested, how it is to be tested, the criteria by which it is possible to decide whether a particular test has been passed or failed, and the order in which tests are to take place. Based on their knowledge of the requirements, the analysts will also draw up sets of test data values that are to be used.

The other key players in the process of testing new software are the eventual users of the system or their representatives. Users may be involved in testing the system against its specification, and will almost certainly take part in final user acceptance tests before the system is signed off and accepted by the clients. If a use-case-driven approach to testing is used, the use cases are used to provide scenarios to form the basis of test scripts. Testers should also watch out for unexpected results.

What is tested?

In testing any component of the system, find out whether its requirements have been met or not. One kind of testing needs to answer the following questions.

- ✓ Does it do what it's meant to do?
- ✓ Does it do it as fast as it's meant to do it?

This type of testing for functional specifications of the system known as ***black box*** testing because the software is treated as a black box. Test put into it and it produces some output, but the testing does not investigate how processing is carried out. Black box testing tests the quality of performance of software. It is also necessary to check how well the software has been designed internally.

The second type of testing known as *white box* testing in which we need answer for the following question.

Is it not just a solution to the problem, but a *good* solution?

because it tests the internal workings of the software whether the software works as specified. White box testing tests the quality construction of the software. In a project where reusable components are used, may not be possible to apply white box testing to these components, as they may provided as compiled object code.

Ideally, testers will use both white box and black box testing methods together to ensure:

- completeness (black box and white box),
- correctness (black box and white box),
- reliability (white box), and
- maintainability (white box).

The aim of any kind of testing is always to try to get the software to fail -find errors- rather than to confirm that the software is correct. For this reason the test data should be designed to test the software at its limits, not merely to show that it copes acceptably with routine data.

Testing can take place at as many as five levels:

1. Unit Testing
2. Integration testing
3. Sub-system testing
4. System testing
5. Acceptance Testing

   In an object-oriented system, the units are likely to be individual classes. Testing of classes should include an initial *desk check,* in which the tester manually walks the source code of the class before compilation. The class should then be compiled, and the compilation should be clean with no errors or warnings. To test the running of a class the tester will require some kind of test program that will create one or more instances of a class, populate them with data and invoke both instance methods and class methods. If pre-conditions and post-conditions have been specified for operations, then the methods that have been implemented will be tested to ensure that they comply with the pre-conditions and that the post-conditions are met when they have completed. State chart diagrams can be used to check that classes are conforming to the behaviour in the specification or not.

   Unit testing merges into integration testing when groups of classes are tested together. The obvious test unit for this purpose is the use case only. The interaction between classes can be tested against the specification of the sequence diagrams and collaboration diagrams.

Testing is generally place at three levels.

LEVEL - I

- Tests individual modules (e.g. classes).
- Then tests whole programs (e.g. use cases).
- Then tests whole suites of programs (e.g. Agate system).

LEVEL - II

- Also known as Alpha testing or verification.
- Executes programs in a simulated environment.
- Particularly tests for all the inputs

    - negative values when positive ones are expected (and vice versa),
    - out of range or close to range limits, or
    - invalid combinations.

LEVEL – III

- Also known as Beta testing or validation .
- Tests programs in live user environment:

    - for response and execution times,
    - with large volumes of data,
    - for recovery from error or failure.

A final stage of testing is ***user acceptance*** testing during which the system is evaluated by the users against the original requirements before the client signs the project off.

Test documentation

Thorough testing requires careful documentation of what is planned and what is achieved. This includes the expected outcomes for each test, the actual outcomes, and for any test that is failed, details of the retesting.

the following figure shows part of a test plan for the Agate case study. It shows details of each test and its expected outcomes. The results of the actual tests will be documented in a separate, but similar format, with columns to show the actual result of each instance of each test and the date when each test was passed, and to document problems. Many organizations have standard forms for these documents or may use spreadsheets or databases to keep this information.

Test no.	Test description	Test data	Expected result
234	Create a new campaign		Campaign Estimated Cost is set to null
235	Add Advert 1 to Campaign.	Advert Estimated Cost = $500.00	Campaign Estimated Cost is set to $500.00

5. Data Conversion

Data from existing systems will have to be entered into a new system when it is introduced. The organization may have a mixture of existing manual and computerized systems that will be replaced by the new system. The data from these systems must be collated and converted into the necessary format for the new system. The timing of this will depend on the implementation strategy that is used, but it is likely to be a costly task, involving the use of staff time, the employment of temporary staff or the use of software to convert data from existing computer systems. These costs should have been identified in any cost benefit analysis that was carried out at the inception of the project.

If data is being collected from existing manual systems, it may be necessary to gather it from different sources. Data may be stored in different files, on index cards, in published documents, such as catalogues, or in other paper-based systems. If this data is going to be entered manually into the new system, by users keying it in, then the designers should draw up paper forms that can be used to collate the information so that it is all in one place when it is keyed in. Some data will only ever be entered when the system is started up, for example codes that are used in the system and will not be altered. Special data entry windows will be required for this kind of one-off activity.

Data from existing computer systems will have to be extracted from existing files and databases and reformatted to be usable with the new system. This provides an opportunity to clean up the data: removing out-of-date records and arranging orderly the values that are stored. The work of converting the data may be done by using special programs written by the developers of the system,

The tasks involved in data conversion are given as follows.

- Creating and validating the new files, tables or database.
- Checking for and correcting any format errors.
- Preparing the existing data for conversion:
  - verifying the existing data for correctness,
  - collating data in special forms for input,
  - obtaining specially written programs to convert and enter the data.
- Importing or inputting the data.
- Verifying the data after it has been imported or input.

All the converted data may have to be ready for entry into the new system to meet a tight deadline, or it may be possible to enter it over a period of time.

6. User Documentation and training

User manuals :

While preparing the technical documentation for the system, analysts will be involved in producing manuals for end-users. The technical documentation will be required by the system manager and other staff responsible for running the system, and by staff who have to maintain the system.

Ordinary users of the system, who will be using it to carry out their daily work tasks, require a different kind of documentation. Users will require two kinds of manuals.

During training they will need training materials that are organized around the tasks that they have to carry out with the new system. These may be in the form of self-study tutorials that users can work through independently of any formal training that is provided.

The users will also need a reference manual that they can refer to while they are using the system. The reference manual should be a complete description of the system in non-technical language. Many software companies assigns technical authors to write manuals in language that users can understand.

The manual should be organized in a easy way for usage. This involves the author understanding how the user will carry out their tasks and the kind of problem that they will face. The reference manual may be replicated in the on-line help so that the users can refer to it while they are using the system. However, it should also be available as a paper manual that the users can refer to if there is a problem with the system,

User training

Temporary staff and existing staff will have to be trained in the tasks that they will carry out on the new system. Analysts are also to be involved in the design of the training program, the development of training materials, the planning of the training sessions and the delivery of the training itself.

Training programs should be designed with clear learning objectives for the trainees. They will be using the system, and it is important that the training is practical and geared to the tasks that they will be performing. If it is too theoretical or technical, they will not find it useful.

Training should be delivered 'just in time' -when the users need it-as they will forget much of what they are told within a short space of time, so training delivered a few weeks before it is required is likely to be wasted. On-line computer-based training using video and audio materials that users can refer to when they need it is likely to be of most use. If formal training sessions are used, then trainees should be given learning tasks to take away and carry out in their workplace. This implies that they will be allocated adequate time for training. Staff will not get the best out of the system and are likely to become frustrated if they do not understand how to work the system. It is often worth following up after users have started using a new system to check that they are using it correctly.

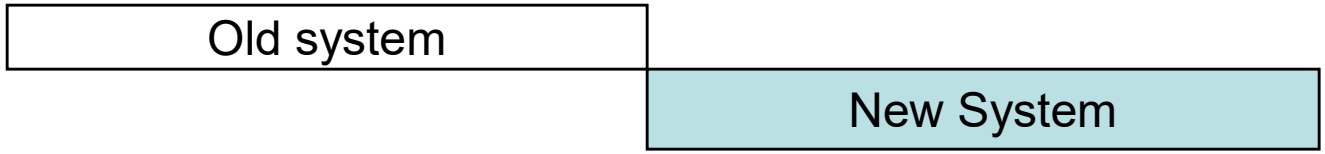
7. Implementation Strategies :

There are four main strategies for switching over to the new system:

1. Direct changeover;
2. Parallel running;
3. Phased changeover;
4. Pilot project.

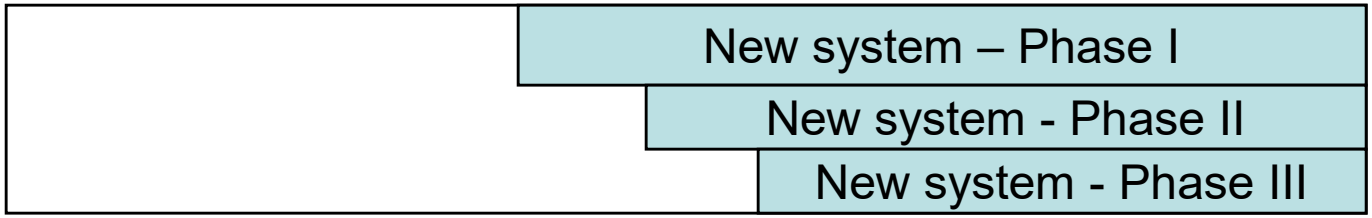
The following figure shows three of these changeover strategies in diagram form. Each of them has its advantages and disadvantages.

Direct changeover :



Parallel running :

Phased changeover :



Direct changeover means that on an agreed date users stop using the old system and start using the new system. Direct changeover is usually timed to happen over a week-end to allow some time for data conversion and implementation of the new system. Direct changeover is suitable for small-scale systems and other systems where there is a low risk of failure such as the implementation of established package software. The advantages and disadvantages of this approach are:

+ the new system will bring immediate business benefits to the organization, so should start paying for itself straightaway;

+ it forces users to start working with the new system, so they will not be able to undermine it by using the old system;

+ it is simple to plan;

- there is no fallback if problems occur with the new system;

- contingency plans are required to cope with unexpected problems;

- the plan must work without difficulties for it to be a success.

Parallel running allows the existing system to continue to run alongside the new system. Parallel running should be used in situations where there is a high level of risk associated with the project and the system is central to the business operations of the organization.

The advantages and disadvantages of this approach are:

+ there is a fallback if there are problems with the new system;

+ the outputs of the old and new systems can be compared-so testing can continue;

- there is a high cost as the client must pay for two systems during the overlap period, and this includes the staffing necessary to maintain information in the old system as well as the new;

-there is a cost associated with comparing the outputs of the two systems;

- users may not be committed to the new system as it is easier to stick with the familiar system.

In a ***phased changeover****,* the system is introduced in stages. The nature of the stages depends on the sub-systems within the software, but introduction into one department at a time may be appropriate. Phased changeover is suitable for large systems in which the sub-systems are not heavily dependent on one another.

The advantages and disadvantages are:

+ attention can be paid to each individual sub-system as it is introduced;

+ if the right sub-systems can be chosen for the first stages then a fast return on invest-ment can be obtained from those sub-systems;

+ thorough testing of each stage can be carried out as it is introduced;

- disaffection and rumour can spread through the organization ahead of the imple-mentation if there are problems with the early phases;

-there can be a long wait before the business benefits of later stages are achieved.

8. Review and Maintenance

The work of analysts, designers and programmers will continue after a system has been implemented also. There is a continuing requirement for staff to work on the new system.

First, it is important that the organization reviews both the 'finished' product and the process which was undertaken to achieve it. This is just for to check that the product meets requirements. However, there is a growing recognition of the need for organizations to learn from experience and to record and manage the organizational knowledge which results from this learning. If there were any problems during the lifetime of the project, then these should be reviewed and conclusions drawn about how they might be avoided in the future. The amount of time spent on different tasks during the project can be used as the basis for metrics to estimate the amount of time that will be required for similar tasks in future projects.

Second, it is unlikely that the system will be working perfectly according to the users' requirements, and further work will have to be done.

Third, in an object-oriented project, the design should be reviewed to identify candidate components for future reuse.

The review process and evaluation report :

The review process will normally be carried out by the systems analysts who have been involved in the project from the start and outside consultants in the process also. They will normally be supported by representatives of users and user management. The various stakeholders who have invested time, money and commitment in the project will all have an interest in the content of the evaluation report. The report can be very detailed or can provide an overview evaluation-like everything else in the project, there will be a cost associated with producing it.

The report's authors should consider the following areas.

Cost benefit analysis. The evaluation should refer back to criteria that were set for the project at its inception. It may not be possible to determine whether all the benefits projected in the cost benefit analysis have been achieved, but most of the costs of development, installation, data conversion and training will have been incurred and can be compared with the projections.

Functional requirements. It is important to check that the functional requirements of the system have been met. Clearly, this is something that should have been taking place throughout the lifetime of the project, but a summary can now be produced. Any actions that were taken to reduce the functional requirements, perhaps to keep the project within budget or on schedule, should be documented for future action under the heading of maintenance. If large areas of functionality were removed to bring the project in on schedule or within budget, a new project should be considered.

Non-functional requirements. The system should be reviewed to ensure that it meets the targets for non-functional requirements that were documented during the requirements analysis stage.

User satisfaction. Both quantitative and qualitative evaluations of the users' satis-faction with the new system can be undertaken, using questionnaires or interviews or both.

Problems and issues. *T*his is an important part of the evaluation process. Problems that occurred during the project should be recorded. These problems may have been technical or political, and it is important to handle the political issues with tact.

Positive experiences. It is all too easy to focus on the negative aspects of a completed project. It is worth recording what parts of the project went well and to give credit to those responsible.

Quantitative data for future planning*.* The evaluation report provides a place in which to record information about the amount of time spent on different tasks in the project, and this information can be used as the basis for drawing up future project plans. The quantitative data should be viewed in the light of the problems and issues that arose during the project, as the amount of time spent on a difficult task that was being tackled for the first time will not necessarily be an accurate predictor of how much time will be required for the same task in the future.

Candidate components for reuse. If these have not already been identified during the project itself, then they should be identified at this stage. There will be different issues to be addressed, depending on whether the project has been carried out by in-house development staff or external consultants.

For in-house projects, the reuse of software components should be viewed as a process of recouping some of the investment made and being able to apply those reusable elements of the system in future projects.

For projects undertaken by external consultants, it may highlight legal issues about who owns the finished software that should have been addressed in the contract at the start of the project.

Future developments. Any requirements for enhancements to the system or for bugs to be fixed should be documented. If possible, a cost should be associated with each item. Technical innovations that are likely to become mature technologies in the near future and that could be incorporated into the system in an upgrade should also be identified.

Actions. The report should include a summary list of any actions that need to be undertaken as a result of carrying out the review process, with an indication of who is responsible for carrying out each such action and proposed timescales.

Maintenance Activities :

Very few systems are completely finished at the time that they are delivered and implemented, and there is a continuing role for staff in ensuring that the system meets the requirements. Next maintenance of the system includes providing initial and on-going training, particularly for new staff ; improving documentation; solving simple problems; implementing simple reports that can be achieved using SQL or OQL without the need for changes to the system software and documenting bugs that are reported; and recording requests for enhancements that will be dealt with by maintenance staff.

Maintenance involves more significant changes to a system once it is up and running. The following are the reasons for the maintenance :

- There will almost certainly be bugs in the software that will require fixing. The use of object-oriented encapsulation should mean that it is easier to fix bugs without creating knock-on problems in the rest of the system. It is sometimes suggested that – bug fixing involves spending as much time fixing bugs that were introduced by the previous round of maintenance as it does in fixing bugs in the original system.
- In an iterative life cycle, parts of the system may be in use while further development undertaken. Subsequent iterations may involve maintaining what has already developed

- users request enhancements to systems virtually from day one after implementation. some of these will be relatively simple, such as additional reports, and may be dealt with by support staff, while others will involve significant changes to the software and will require the involvement of a maintenance team.
- In some cases, changes in the way that the business operates or in its environment, for example new legal representative for the system, will result in the need for changes to the system.
- Similarly, changes in the technology that is available to implement a system may result in the need for changes in that system.
- Disasters such as fires that result in catastrophic system failure or loss of data may result in the need for maintenance, staff to be involved in restoring the system from data back-ups. Procedures for handling disastrous system failure should be put in place before disasters take place.

In each of these cases, it is necessary to document the changes that are required. In the same way as it is necessary to have a system in place during a project for handling users' requests for changes to the requirements (a change control system), it is necessary - to have a system for documenting requests for changes and the response of the maintenance team.

This should include the following elements.

Bug reporting database. Bugs should be reported and stored in a database. The screen forms should encourage users to describe the bug in as much detail as possible. In particular, it is necessary to document the circumstances in which the bug occurs so that the maintenance team can try to replicate it in order to work out the cause.

Requests for enhancements. These should describe the new requirement in similar amount of detail. Users should rate enhancements on a scale of priorities that the maintenance team can decide how important they are.

Feedback to users. There should be a mechanism for the maintenance team to feed back to users on bug reports and requests for enhancements. Assuming that bugs the agreed functionality of the system, users will expect them to be fixed as part of original contract or under an agreed maintenance contract. The maintenance team should provide an indication of how soon each bug will be fixed.

Depending on the contractual situation, enhancements may be carried out under a maintenance contract or they may be subject to some kind of costing procedure. Significant enhancements may cost large amounts of money to implement. They will require the same kind of assessment as the original requirements. They should not be left to maintenance programmers to implement as they see fit, but should involve analysts and designers to ensure that the changes fit into the existing system and do not have repercussions on performance or result in changes to sub-systems that affect others. This process itself may incur significant costs just in order to work out how much an enhancement will cost to implement.

Implementation plans. The maintenance team will decide how best to implement changes to the system, and this should be carried out in a planned way. For example, significant additions to a class that affect what persistent data is stored in the database will require changes to the database structure, and may also require all existing instances of that class to be processed in order to put a value into the new attribute. This will probably have to take place when the system is not being used, for example over a weekend.

Technical and user documentation. Changes to a system must be documented in exactly the same way as the original system. Diagrams and repository entries must be updated to reflect the changes to the system. If this is not done, then there will be a growing divergence between the system and its technical documentation; this will make all future changes more difficult, as the documentation that maintenance analysts consult will not describe the actual system. Clearly, user documentation, training and help manuals as well as on-line help must all be updated.

In large organizations with many systems, staff in the information systems department may spend more time on maintenance of existing systems than they do on development of new systems.

There is a growing movement for organizations to ***out source*** their maintenance. This means handing over the responsibility for maintenance of a system to an external software development company under a contractual agreement that may also involve the provision of support. Some companies now specialize entirely in maintaining other people's software.

20. Reusable Components

In the system life cycle, implementation is followed by maintenance, when the new system has any remaining bugs removed and enhancements made to it. Using object-oriented technology does not make the problems of removing bugs and enhancing systems go away, but it does add a possible further stage to the life cycle - the reuse stage, here we need to consider how object-oriented software can be reused.

1. Why Re-Use?

Reusability is one of the reasons for adopting object-oriented development techniques and programming languages, for this inheritance and composition are two techniques that facilitate the develop reusable components, finally reusability as one characteristics of a good object-oriented design.

Reusable software has been one objectives of developers . Using top-down functional decomposition of designs in languages such as Fortran or C, the development of reusable libraries of functions has made it possible for programmers to save time and effort by reusing others work. The growth of Visual Basic as a programming language was aided by the ability of controls that could be bought off the shelf and incorporated into applications to provide functionality that would be difficult for the less experienced programmer to develop reusable code.

The arguments for reuse are partly economic and partly concerned with quality.

- If some of the requirements of a project can be met by models or software components that have been developed on a previous project or are bought in from an outside supplier, then the time and money spent producing those models or code is saved. Although the saving will be partly offset by the cost of managing a catalogue of reusable models or code or of paying to buy them from elsewhere.

- If a developer can reuse a design or a component that has been tested and proved work in another application, then there is a saving in the time spent to test quality assure the component.

  Developers of object-oriented systems are often end-users of reusable components, when they use packages, libraries, classes, or controls in their chosen development environment. However, object-oriented systems have not achieved the level of reuse that was expected of them in terms of generating reusable components that can applied again within the same organization. There are a number of reasons for this . some are technical and some are concerned with organizational culture.

1. ***Inappropriate choice of projects for reuse.*** Not all organizations or projects within those organizations are necessarily suitable to take advantage of or act as to sources of reusable components.

2. ***Planning for reuse too late.*** If reuse is appropriate, it is something that needs to be planned for even before a project starts, not an after project starts. By the time a project has been completed, it is likely that anything that might have been reusable will have been designed in such a way that it cannot easily be extracted from the rest of the system. To achieve reuse, the organization needs to be structured to support it, with the people and tools in place to make it possible.

3. ***level of coupling between different classes in an object-oriented design.*** *Many* people have thought of classes as the unit of reuse in object-oriented developments. However, when we come to design classes for different systems, it may be possible to identify similar classes that could be developed in a way that makes them of use in more than one system, but more often than not, the implementations of these classes will include attributes and associations that tie them into other classes in the particular application of which they are a part.

4. ***lack of standards for reusable components.*** This has changed recently with developments in the technology of repositories in which to store components and the introduction of standards such as the Object Management Group's CORBA version 2.0 and the W3C's SOAP (Simple Object Access Protocol).

1.Choice of Project

Not all projects are necessarily suitable for the development of reusable components. The two main factors that influence this are the nature of the business within which the software development is taking place and the maturity of the organization's object-oriented development.

Jacobson et al. identified the following four kinds of software business, which are suitable candidates for developing reusable components.

• Organizations where creating an RSEB (Re-use Driven Software engineering Business) improves the business processes within the organization: large organizations with a considerable information systems infrastructure and a portfolio of projects to support business activities.

• Organizations producing hardware products that contain embedded software

• Consultancy companies and software houses that develop software for external clients that have outsourced their information systems development.

• Developers of software products, such as Microsoft, where reusable components can be applied across a large product range and where end-users can also benefit from the interoperability of software through mechanisms such as DCOM (Distributed Component Object Model).

If, an outside consultancy company is doing the development for both Agate and FoodCo, then there are a number of areas where reuse may be applicable. A software company looking at the two systems would identify areas such as managing information about staff that are common to both.

2 Organizational structure

Jacobson et al. based on experience at Hewlett-Packard, describe organizations as typically going through six stages of development of a reuse culture. At each stage some benefit is to be gained, but it is unlikely that an organization can leap from a situation in which there is no reuse of design models or software taking place to one in which there is a complete organizational culture of reuse and the structures and the tools are in place to support the consistent reuse of components in a way that brings the kind of business benefits from re-usability.

The six stages are as follows :

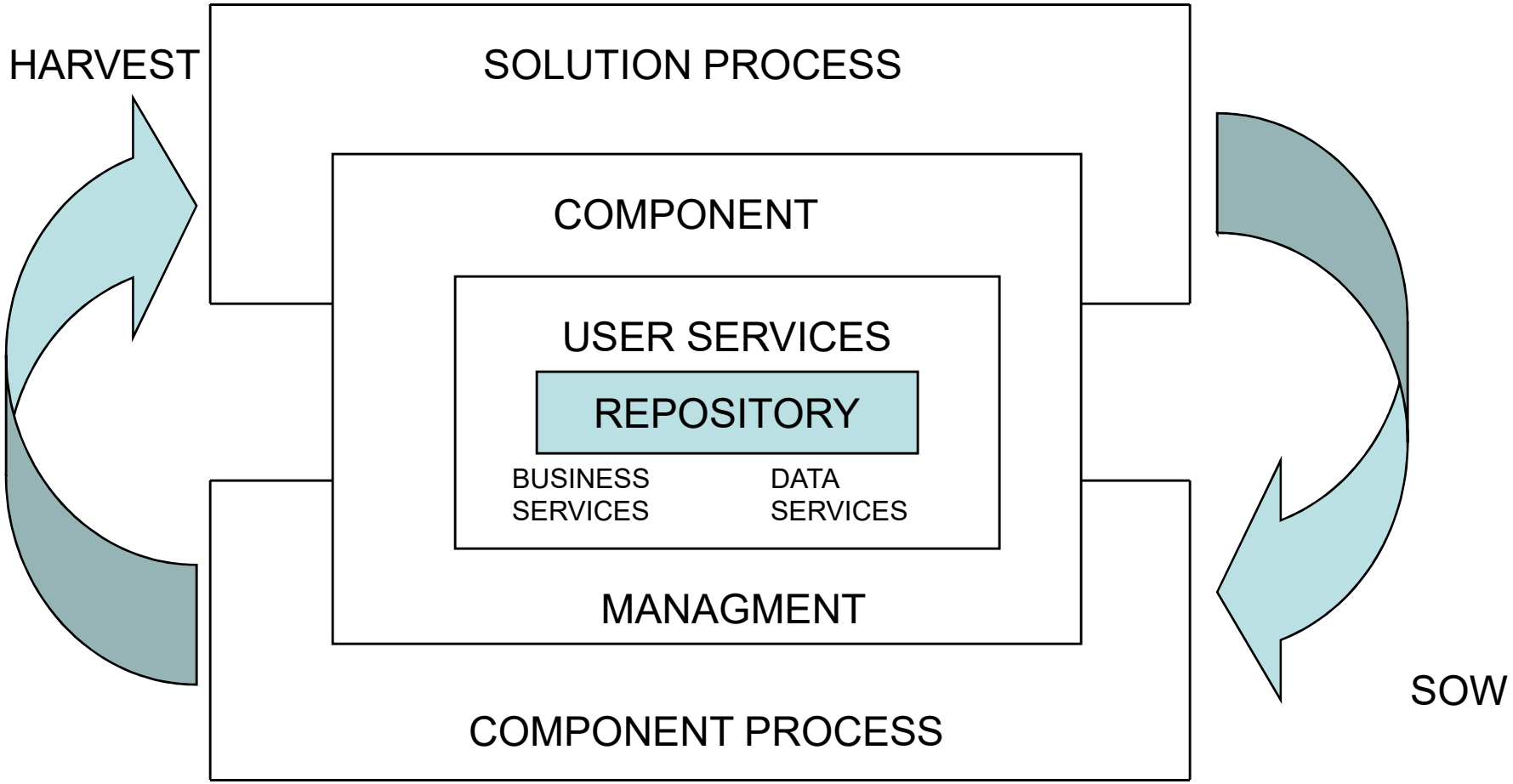
1. **None.** No code reuse takes place; everything is developed from scratch.
2. **Informal code reuse.** Developers trust each other enough to begin to reuse each other's code in order to save time on development.
3. **Black-box code reuse.** Particular pieces of code are engineered for reuse, and all developers are encouraged or required to use them to ensure a consistent approach and reduce maintenance costs.
4. **Managed work product reuse.** An organizational structure is developed to manage reusable code, to maintain versions, to document functionality and to train developers.
5. **Architected reuse.** In order to ensure that components work together, a common architecture is designed and applied to all development processes.

6. Domain-specific reuse-driven organization. The organization's software development is geared to the production of reusable components for the business domain and the culture and structure of the organization supports this approach.

To gain the benefits of an RSEB requires an incremental process of change within the organization, involving technical experts to argue the technical case and develop the software architecture, management experts who believe in the business benefit and will provide the support and investment to allow the change to take place, and the development of support structures within the organization. Among these, the first is the most critical: to achieve effective reuse, the elements of the software architecture must be common across different systems.

One of the most significant requirements for support structures is that if developers are to use reusable components in their code they need some way of finding out what components are available and what their specifications are. This requires software tools to manage a repository of components and staff to maintain the components in the repository and to document them.

Allen and Frost (1998) place a repository at the centre of their model of the development process for reusable components. The following figure shows this with the two complementary processes: sowing reusable components during development and harvesting reusable components for reuse in other projects.



The SELECT Perspective service-based process

To develop reusable components while achieving the development of a system to meet users' needs, the Perspective approach breaks the development process into two parts:

1.The solution process focuses on specific business needs and delivering services to meet the users' requirements. Its products have immediately definable business value. During the solution process, developers will draw on the component process in their search for reusable components that can be applied to the project.
2.The component process focuses on developing reusable components in packages that group together families of classes to deliver generic business services. During the component process, the developers produce components that can be reused in the solution process. The component process also searches out opportunities to reuse services from existing legacy systems and legacy databases and from other packages of components.

Allen and Frost used, the analogy of sowing and harvesting reusable services: the component process sows reuse and the solution process harvests services.

Software support is needed for effective component reuse to take place. This support takes the form of repository-based component management software. Components are placed in the repository as a means of publishing them and making them available to other users. The repository is made up of catalogues and the catalogues contain details of components, their specifications and their interfaces. Component management software tools provide the functionality for adding components to the repository and for browsing and searching for components. Component management software may be integrated with CASE tools to allow the storage of analysis and design models as well as source code and executables.

3. Appropriate unit of reuse

If we consider the case studies, there is a need for a Client or customer class both in the Agate system and in the FoodCo system. During analysis, these two classes may look very similar, but as we move into design, the associations between these classes and others in their system will be resolved into specific attributes.

The Agate Client class will have attributes to link it to Campaigns while the FoodCo Customer class will be linked to SalesOrders. If the development of both systems is being carried out by the same software company, then it requires a novel style of project management and organization to recognize this commonality in two different projects. If the commonality is recognized, then there is no guarantee of successful reuse unless a suitable architecture is developed that will support the reuse of the common elements of the Client class and allow it to be tailored to the requirements of the individual systems.

For example, either a Client class can be obtained from else where and subclassed differently for each project, or a Client class can be written that is domain-neutral and then subclassed for each different project. This inheritance-based approach also helps to solve a problem that is common with software that is tailored to the needs of different customers of the software house: it clearly separates those parts of the that are common to all users from those that have been tailored to specific needs. This helps with the installation of upgrades and prevents the changes made for customer being implemented for all customers.

However, even if we can reuse the Client class in both applications by extending functionality through inheritance, there are going to be other aspects of the Client class that we may or may not want to take through into another system. These include control classes and the business logic associated with the management of clients related boundary classes and the mechanisms that manage the persistent storage instances of Client in some kind of database. So we cannot reuse the class on their own decisions. Because class is the wrong level of granularity at which to apply for reuse. And reuse should take place at the level of components rather than classes.

According to Allen and Fost , A component is an executable unit of code that provides physical black-box encapsulation of related services. Its services can only be accessed through a consistent, published interface that includes an interaction standard. A component must be capable of being connected other components (through a communications interface) to form a larger group.

According to Jacobson et al. define a component as follows.

A component is a type, class or any other work product that has been specifically engineered to be reusable. This definition is more useful, as it does not limit the developer to only considering executable code for reuse. The intermediate products of the development life cycle - use case model, analysis model, design model, test model-can all be considered as candidates for reuse. There are two outcomes from this view .

First, we may choose to reuse as components sub-systems that provide more functionality than just a single class.

Second, we may choose to reuse intermediate products.

Jacobson et al. Suggests the following are different mechanisms for reusing components.

- Inheritance , composition and Aggregation .
- the «include» relationship between use cases
- extensions and extension points in use cases and classes,
- parameterization, including the use of template classes,
- building applications by configuring optional components into systems and
- generation of code from models and templates.

The last two are considerable in development processes rather than specific design structures, and make reuse easier to achieve.

4.Component standards

In we want to consider black-box reuse, the potential for reuse depends on the software mechanisms for reusable components. If we want to consider white-box reuse, then the potential depends on the mechanisms for exchanging software models. In the second approach, UML is a candidate for exchangeable, reusable software models, especially if CASE tool vendors implement the XMI (XML Metadata Interchange). In the first case, we are dependent on the developers of programming languages and software development infrastructure to deliver appropriate tools to the development community to enable them to develop reusable components.

A number of programming languages and development environments provide mechanisms by which developers can package software into components. The following figure lists some of these.

A sample of languages and development environments with mechanisms for reuse

Language or development environment	Mechanism for component reuse
Microsoft Visual Basic	.vbx files-Visual Basic Extensions .ocx files
Microsoft Windows	.ole files - Object Linking and Embedding DDE - Dynamic Data Exchange .dll files - Dynamic Link Libraries COM-Common Object Model DCOM-Distributed Common Object Model
Java	.jar files - Java Archive packages JavaBeans
CORBA	.idl files - Interface Definition Language IOP - inter-ORB Protocol

The above table shows that the search for ways of promoting reuse through some kind of modular architecture is not new in the software development industry. Reuse has been an objective that has driven the design of programming languages and has informed the development of programming styles. However, the potential for developing reusable components has been increased recently by three factors.

- The development of CORBA as a standard for interoperability of components written in different languages and running on different platforms.

- The promotion ofJava as an object-oriented language with relatively straightforward mechanisms for producing software in packages to deliver different services.

- The growth of the Internet and the WWW, which has made it possible for people to make their software components easily available to a wide marketplace of potential reusers.

. NET, which is emerging at the time of writing, together with C#, the new language from Microsoft, may provide an alternative to Java. In particular, it provides a mechanism based on the SOAP Contract Language (SCL) to discover the services offered by Web Services, which describe themselves using XML.

.NET also defines extensions to Microsoft's Portable Executable (PE) format so that metadata is stored with the byte code in Microsoft Intermediate Language (MSIL) executables, allowing them to provide information about the services they offer in response to requests in the correct format. Microsoft will provide

2. Planning a strategy for Reuse

In some organizations, reuse may just be about making use of reusable components from elsewhere, In others, reuse will be about the kind of organizational change .

The following are two approaches to the introduction of a reuse strategy .

1 The SELECT Perspective

Allen and Frost describes the SELECT Perspective approach to the developers of reusable components. At the level of practical techniques, this includes guidelines for the modelling of business-oriented components and for wrapping legacy software in component wrappers. They distinguish between reuse at the level of component packages, which consist of executable components grouped together, and service packages, which are abstractions of components that group together business services.

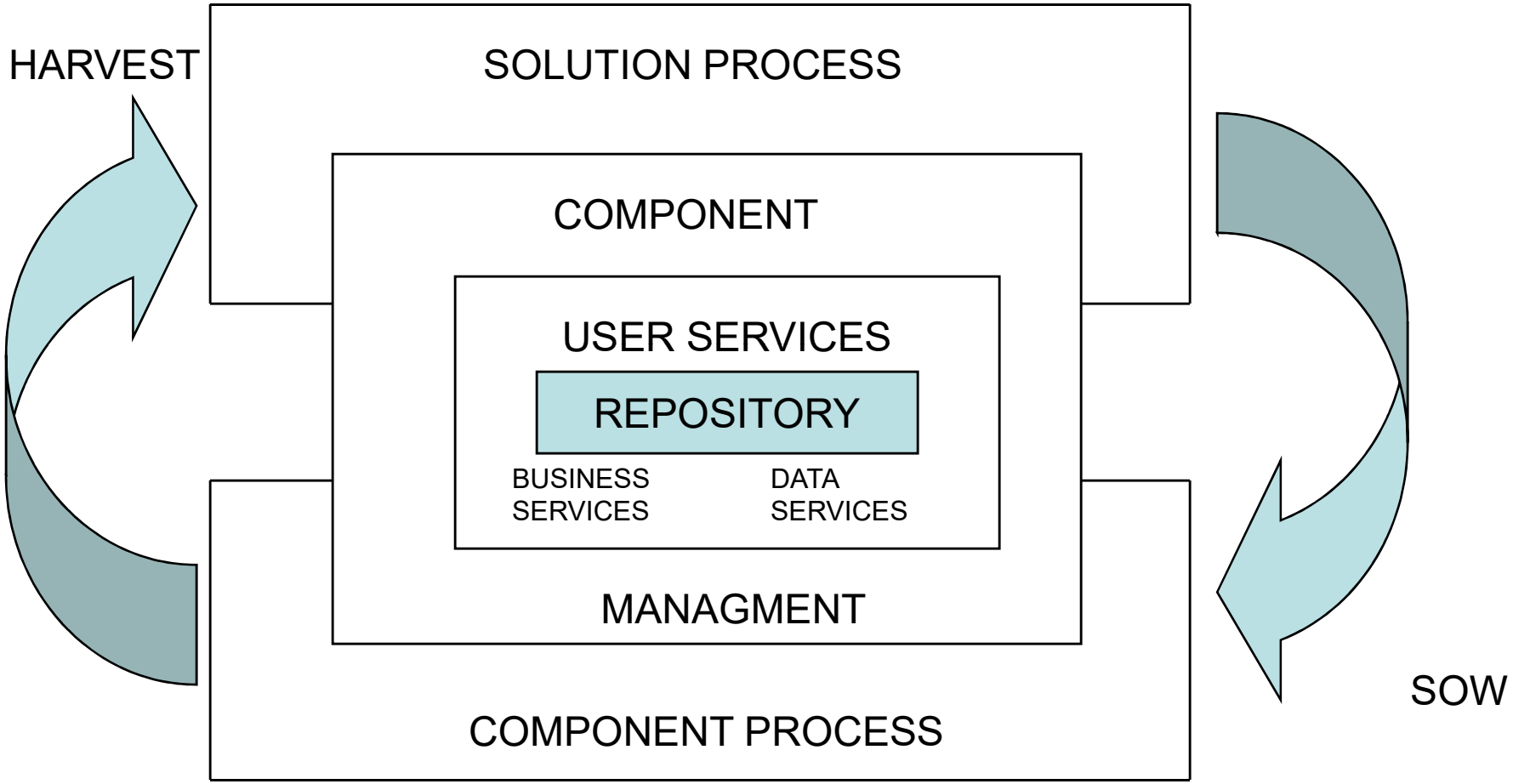
The focus of this approach is to identify the services that belong together and classes that implement them. Service classes in a single package should have a high level of internal interdependency and minimal coupling to classes in other packages .

In order to develop reusable components while achieving the development of a system to meet users' needs, the Perspective approach breaks the development process into two parts: 1. The solution process

2. The component process.

These two parts run in parallel and feed off each other.

Allen and Frost placed a repository at the centre of their model of the development process for reusable components. The following figure shows this with the two complementary processes: sowing reusable components during development and harvesting reusable components for reuse in other projects.



The SELECT Perspective service-based process

- The solution process focuses on specific business needs and delivering services to meet the users' requirements. Its products have immediately definable business value. During the solution process, developers will draw on the component process in their search for reusable components that can be applied to the project.

- The component process focuses on developing reusable components in packages that group together families of classes to deliver generic business services. During the component process, the developers produce components that can be reused in the solution process. The component process also searches out opportunities to reuse services from existing legacy systems and legacy databases and from other packages of components.

Allen and Frost used, the analogy of sowing and harvesting reusable services: the component process sows reuse and the solution process harvests services.

Software support is needed for effective component reuse to take place. This support takes the form of repository-based component management software. Components are placed in the repository as a means of publishing them and making them available to other users. The repository is made up of catalogues and the catalogues contain details of components, their specifications and their interfaces. Component management software tools provide the functionality for adding components to the repository and for browsing and searching for components. Component management software may be integrated with CASE tools to allow the storage of analysis and design models as well as source code and executables.

2 . Reuse-driven Software Engineering Business

Jacobson et al. describe an approach to developing reusable software components with RSEB, which is based on practical experience within Ericsson and Hewlett-Packard.

Unlike Allen and Frost, who consider components as executables or as packages of executables designed to deliver a particular service, Jacobson et al. consider reuse in terms of any of the work products of systems development. This means that models that are produced before the finished program code are candidates for reuse, and that artefacts other than classes, for example use cases, can be reused. However, the key point of this approach is that the design of systems to make use of reusable components requires an architectural process right from the start. And that means changing the way the business operates.

Jacobson et al. explains an approach to business process reengineering that is based on OOSE and Objectory. The task of developing a reuse business is a reengineering task that can be modelled using object-oriented business engineering, and that leads to the development of systems to support the RSEB. And they suggests that the end result is a business consisting of the following competence units:

- ✓ Requirements Capture Unit,
- ✓ Design Unit, Testing Unit,
- ✓ Component Engineering Unit,
- ✓ Architecture Unit,
- ✓ Component Support Unit

These competence units are groupings of staff with particular skill sets and the business and documents for which they are responsible.

The emphasis in RSEB is to design an architecture for systems that support reuse from the start. This is done through three engineering processes:

1. Application Family Engineering
2. Component System Engineering
3. Application System Engineering.

Application Family Engineering (AFE) is an architectural process that captures the requirements for a family of systems and turns them into a layered architecture , consisting of an application system and a supporting component system.

Component System Engineering (CSE) is the process of focusing on the requirements for the component system and developing the use cases, analysis models and design for reusable components to support application development.

Application System Engineering (ASE) is the process of developing the requirements for applications and developing the use cases, analysis models and design produce application software that makes use of the reusable component systems developed by CSE.

The life cycle for this kind of project is an iterative one. The engineering processes run concurrently, with the emphasis changing as the project progresses.

3. Commercially available Componentware

Most commercially available components took the form of utilities or graphical user interface components. The best example of this is the wide variety of controls that are available for use with Microsoft Visual Basic. Originally these were supplied as add-ins in the form of .vbx files which could be included in the Visual Basic toolbar in the same way as the built-in controls, or in the form of OLE (Object Linking and Embedding) objects which allowed the functionality of other software packages such as word-processors to be embedded in applications. With introduction of ActiveX, based on the Microsoft COM architecture, these add-ins are now available as .ocx files.

If catalogue of a good software supplier who sells development tools is considered, then you will find pages of ActiveX controls that be used in your applications and that provide the developer with the possibility of building a wide range of different functions into their software without having reinvent the wheel.

Examples include:

1. serial communications ,
2. computer-aided design,
3. project management including Gantt charts,
4. spreadsheets,
5. scientific charts ,
6. barcode reading and printing.

The use of standardized mechanisms to access the functionality of these controls has meant that other software companies can also write interfaces to them.

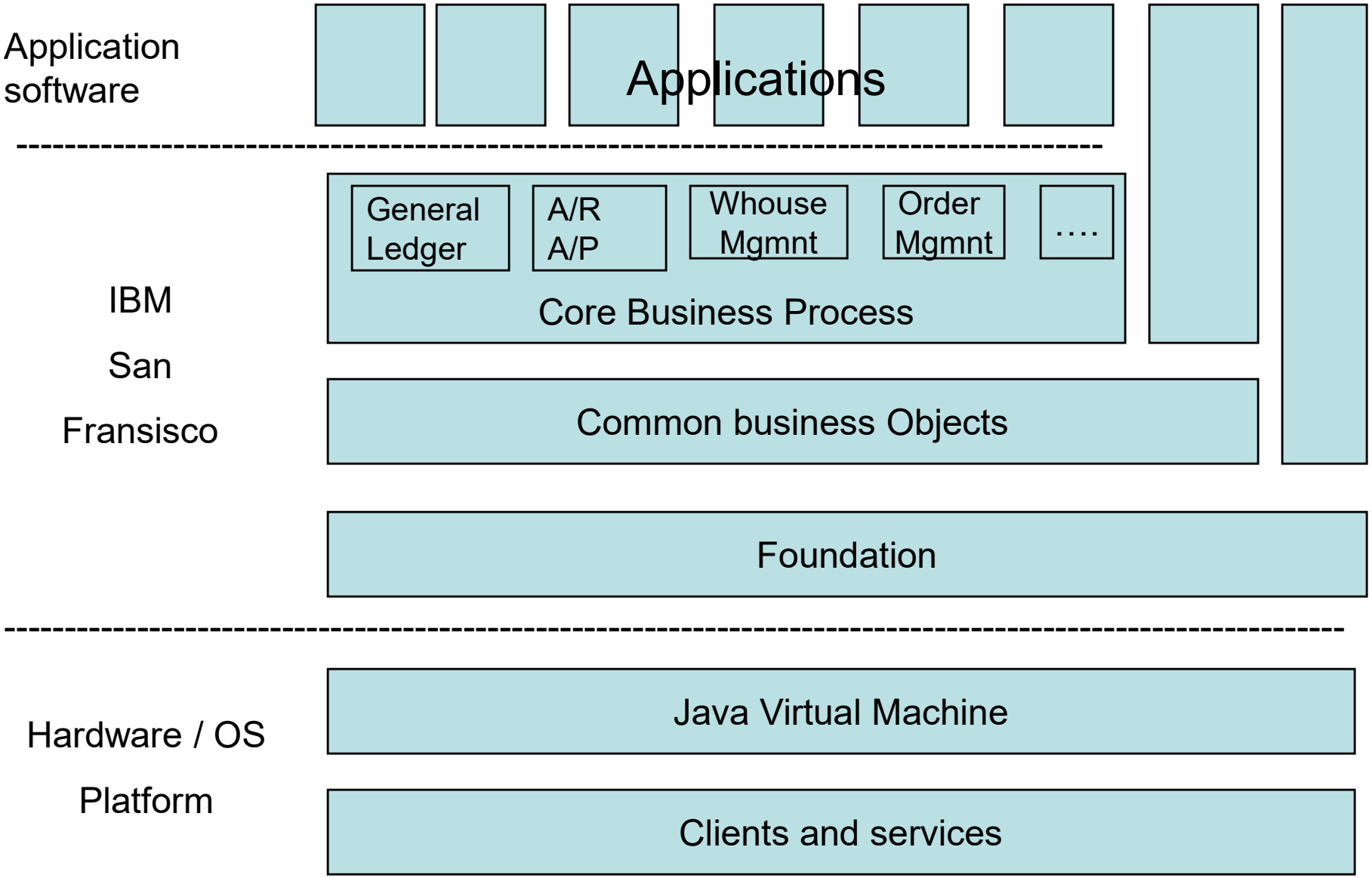
For applications written in Java, however, there is another mechanism that can be used; the JavaBean. JavaBeans are components written in Java that typically combine a graphical element with the functionality to support it. Beans support the Component-Container model in the Java Abstract Windowing Toolkit , which means that Beans that encapsulate access to particular domain objects can be added into applications in the same way as other visual components can be added to the applications. Most of these add-in controls provide generic capabilities rather than reusable components for particular types of business operations.

The San Francisco project provides distributed, server-based components for different types of business processes. San Francisco uses a layered architecture as shown in the following figure.

The Foundation layer provides a programming model to support distributed transactions, and uses a set of distributed object services and utilities written entirely in Java. It also provides an optional GUI framework written using JavaBeans. The common Business Objects layer implements general purpose business objects together with the facts and rules required for any business application. This includes business objects such as company, business partner, address and calendar.

The following are Four components are provided in the Core Business Processes layer.

1.General Ledger.
2.Accounts receivable and accounts payable.
3. Warehouse Management.
4. Order Managment



Layered architecture of the San Francisco project

These have been built using design patterns, many of which were discovered as the project developed, and provide support for E-Commerce applications

4. Façade Pattern :

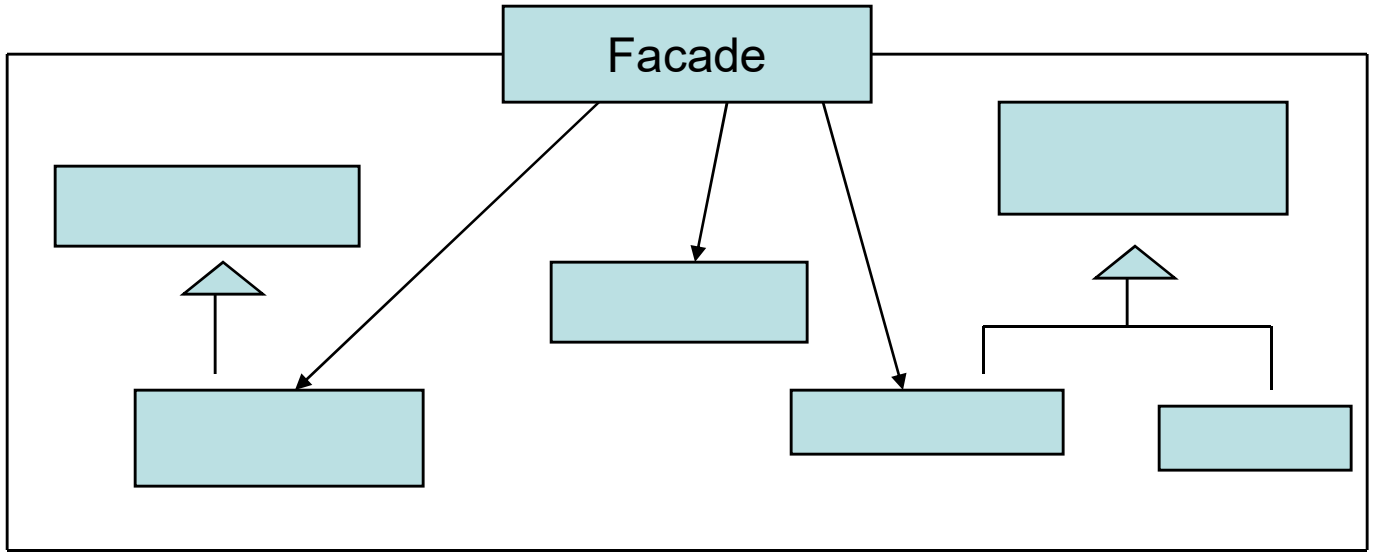
If you want to provide a unified interface to a set of interfaces in a sub-system of a system then use, Facade pattern which defines a higher-level interface that makes the sub-system easier to use in a system.

Applicability :

Use the Facade pattern when

- you want to provide a simple interface to a complex sub-system.
- there are many dependencies between clients and the implementation classes of an abstraction. Introduce a facade to decouple the sub-system from clients and other sub-systems, thereby promoting sub-system independence and portability.
- you want to layer your sub-systems .

The structure of the Facade pattern is shown below



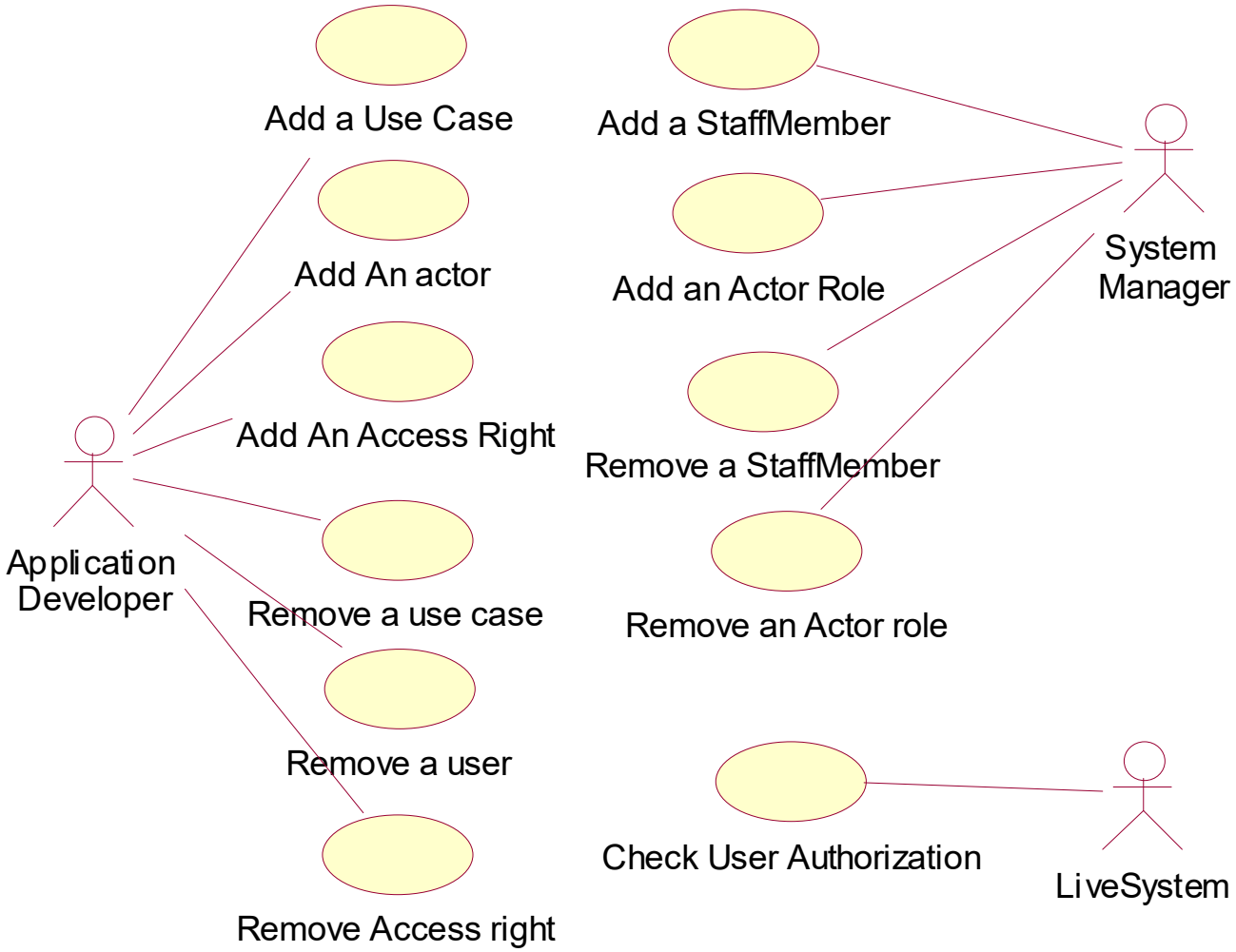
5. Case Study example

A common feature of many applications is the need to control the access of members of staff to the different programs that make up a system. A non-functional requirement for Agate Ltd. is to restrict access of staff to the use cases that they are permitted to use. This requirement can be summarized as follows .

Each program in the system will be represented by a use case, in the use case diagram. One or more actors will be associated with each use case, and each actor may be associated with more than one use case. A member of staff will fill the role of one or more actors, and each actor will be represented by one or more members of staff. Each actor will be given access rights to use specific use cases. A member of staff may only use those use cases (programs) for which one of the roles they fill has been given an access right.

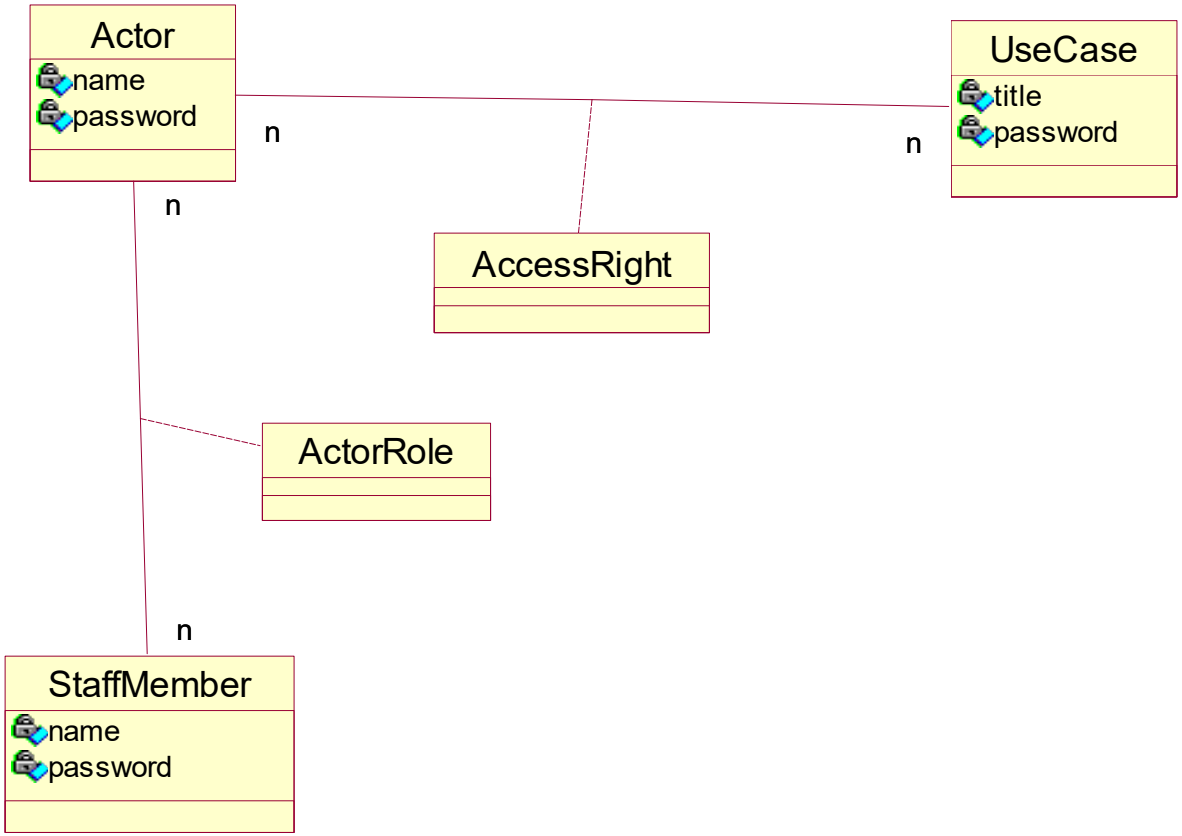
This non-functional requirement in the context of the main systems can be viewed as the basis for functional requirements in a security sub-system. This sub-system is a potential candidate for the development of a reusable component.

the following figure shows the use cases for this security sub-system.

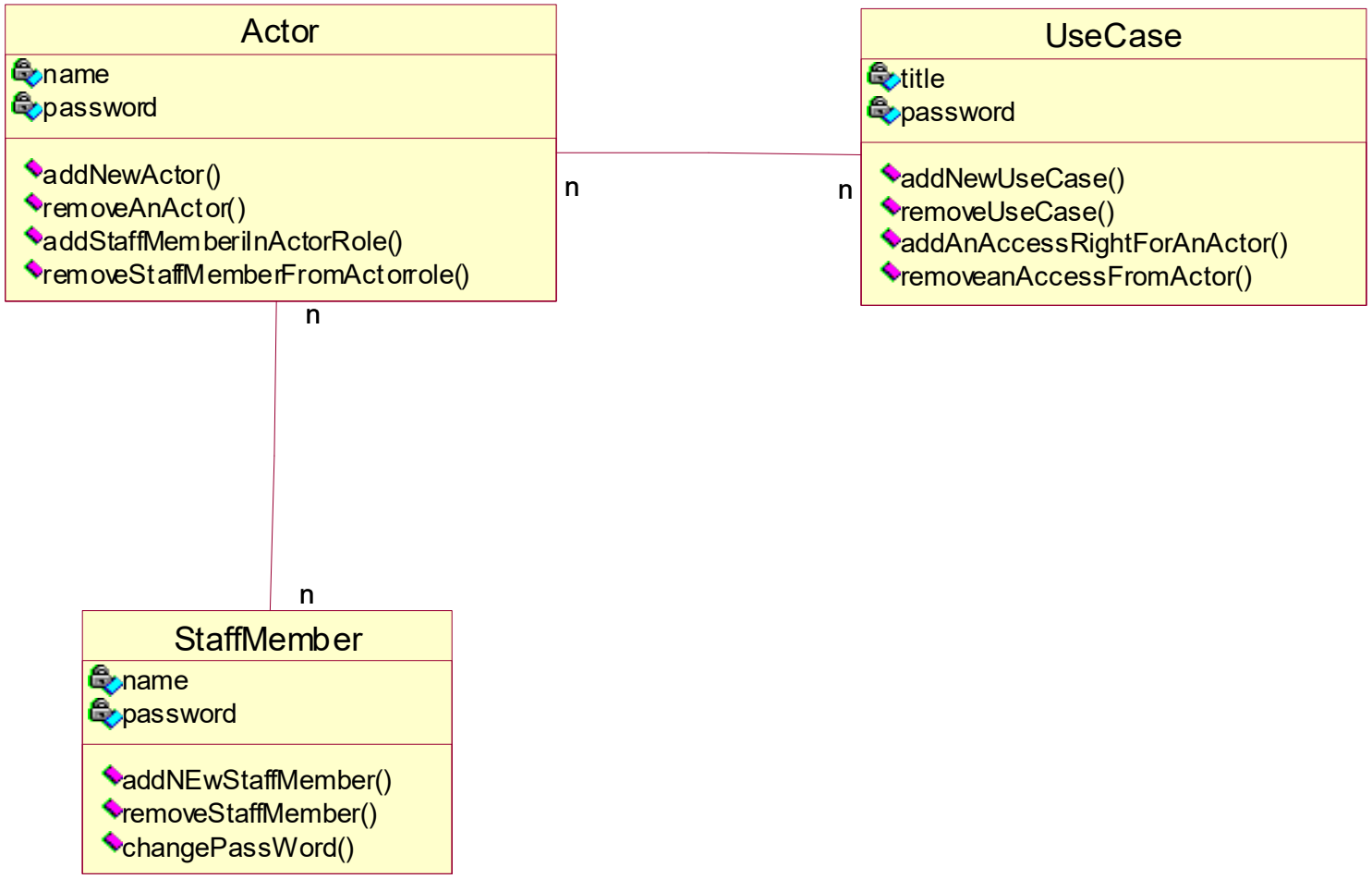


Use case diagram for security Management system

It can be modelled in a class diagram in the same way as the business classes that meet the functional requirements of the system. The following figure shows the initial domain class diagram for this requirement. Two association classes ActorRole and AccessRight have been included in the class diagram, as it was initially considering that there might be some data associated with the creation of links between instances, for example the data that an access right was given, or the type of an access right (Read/Write/Update).



However, further discussion with users and analysis of the requirements indicates that this is likely to make the sub-system more complicated than it needs to be, so they have been removed from figure as shown below , which gives the analysis class diagram.



There are many design alternatives for this part of the system. The particular design alternative that we choose will affect the detailed design of this sub-system.

If we are the software company developing software for Agate , then some of the alternatives are as follows .

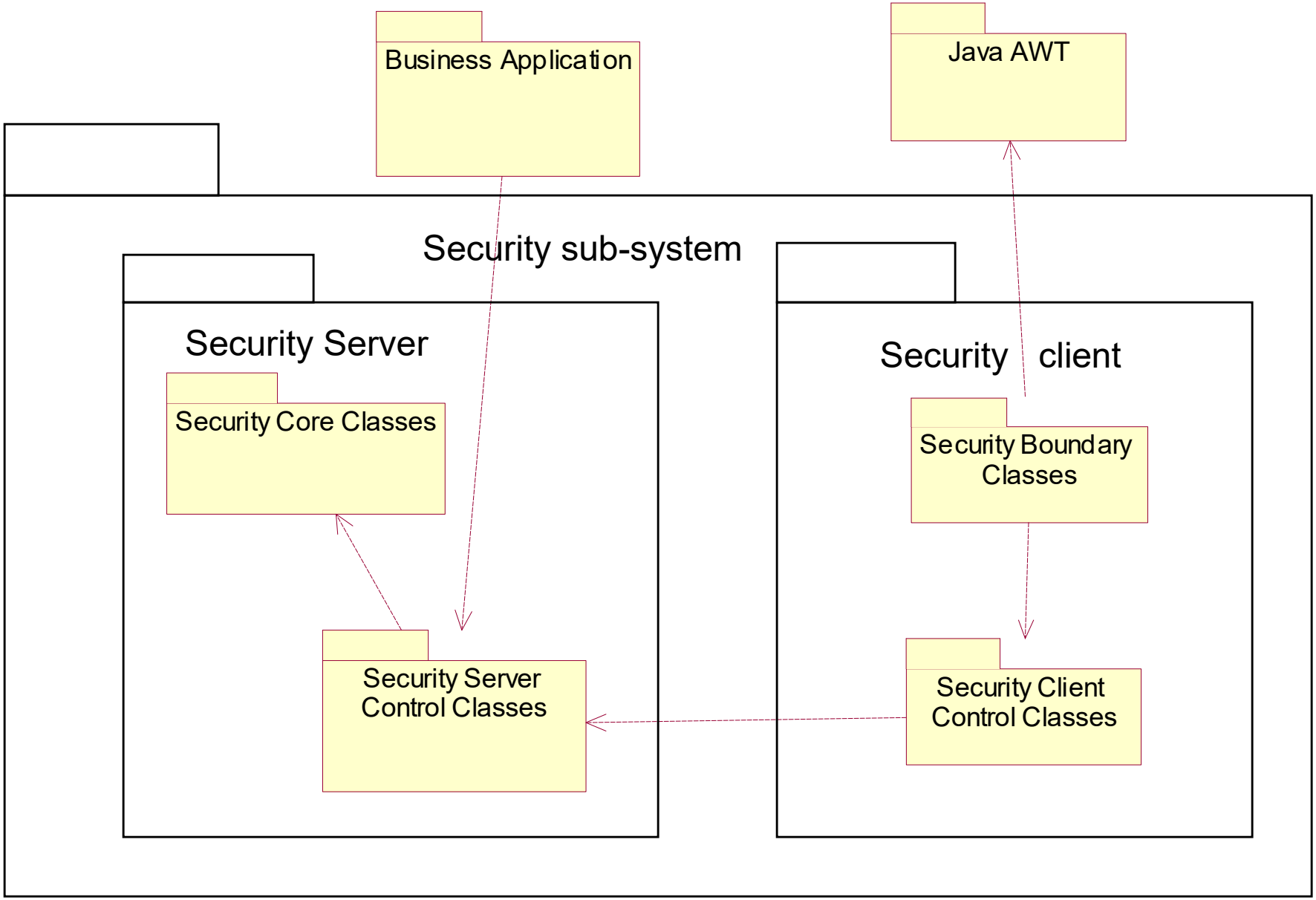
- Do we design this sub-system with a set of boundary and control classes to support the use cases which are in the security sub-system.
- Can we reuse the existing Staff Member class in the business domains of Agate Ltd.? We do not really want to have to set up data about staff members in two places.
- What happens if we do reuse the Staff Member class in the business domain and then want to use this security sub-system to support a system that does not have Staff Member as an entity class?
- If this security sub-system is to be implemented for all the application software we develop, then we are going to have to make some classes in the software (means the control classes) aware of the interface to this sub-system. How do we extend these control classes: rewrite them, extend them or subclass them?
- How do we provide persistent data storage for this sub-system?
- What parts of this sub-system are we going to make visible to other applications?

We might choose to design the system so that when a user starts running the application, they are prompted for their user ID and a password. Alternatively, if they are required to log into a network anyway, the software could obtain the user's ID from the network operating system. Each time a user starts a new use case in the main application, the application will need to check with the security classes whether that user is authorized to use that particular use case.

The security requirement is not part of the business requirements of the domain applications, and we want to reuse the security software in other applications, so it makes sense to separate these classes from the rest of the software and put them in a package of their own. The security classes will require their own boundary classes, to allow the actors to carry out the use cases. These will run on client computers and will be in a separate package within the overall security package. They will have dependencies on other packages that provide these services. We have created two packages for control classes, one for classes that will run on the clients and control the boundary classes, and one for control classes that will run on the server. These control classes will have a dependency on the core security classes.

The following figure shows these package dependencies. Here also we have shown a package to represent a business application that will be using the services of the security package to authenticate users. It is arguable whether this should have a dependency on the server control classes or on some kind of client package that hides the implementation. Whatever approach we take, we want to provide a clean interface to the functionality of the security sub-system for developers to use.

Package diagram showing security classes and dependencies



It should be possible to design and implement a separate security client, which uses the interface to the security server control classes. Also, programmers should have a straightforward application programming interface (API) to the authentication service - the use case Check user authorization.

One way of doing this would be to replace the control classes in the Security Server Control Classes package with a single control class. This will make it easier for developers to reuse the package, and application programmers only need to know the API of this one class. However, that would lead to a single class with no attributes and a large number of operation implementations.

An alternative approach is to leave the control classes as they are and to create a Facade class that has the operations from the control classes within the sub-system but does not contain their implementation.

This second approach is based on a design pattern, called the Facade pattern. The Facade pattern is a combination of Four structural pattern, and is described in the following terms.

Intent :

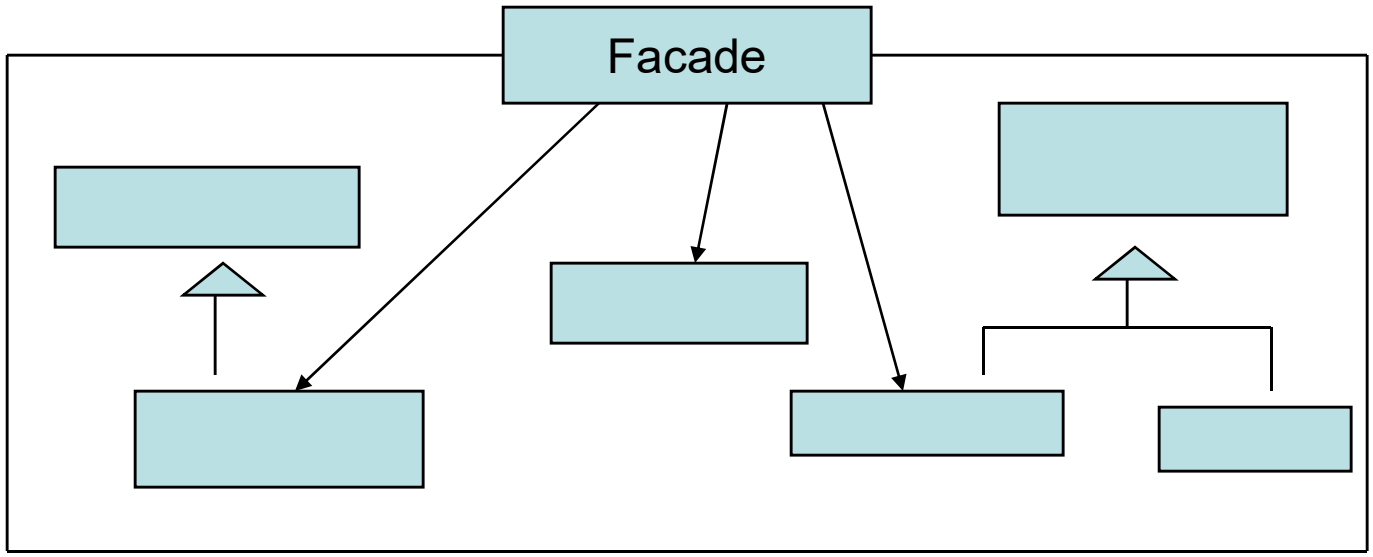
Provide a unified interface to a set of interfaces in a sub-system. Facade defines a higher-level interface that makes the sub-system easier to use.

Applicability

Use the Facade pattern when

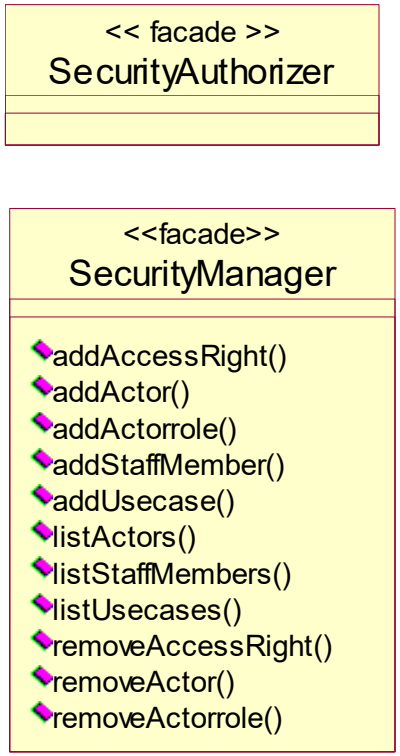
- you want to provide a simple interface to a complex sub-system.
- there are many dependencies between clients and the implementation classes of an abstraction. Introduce a facade to decouple the sub-system from clients and other sub-systems, thereby promoting sub-system independence and portability.
- you want to layer your sub-systems .

The structure of the Facade pattern is shown below

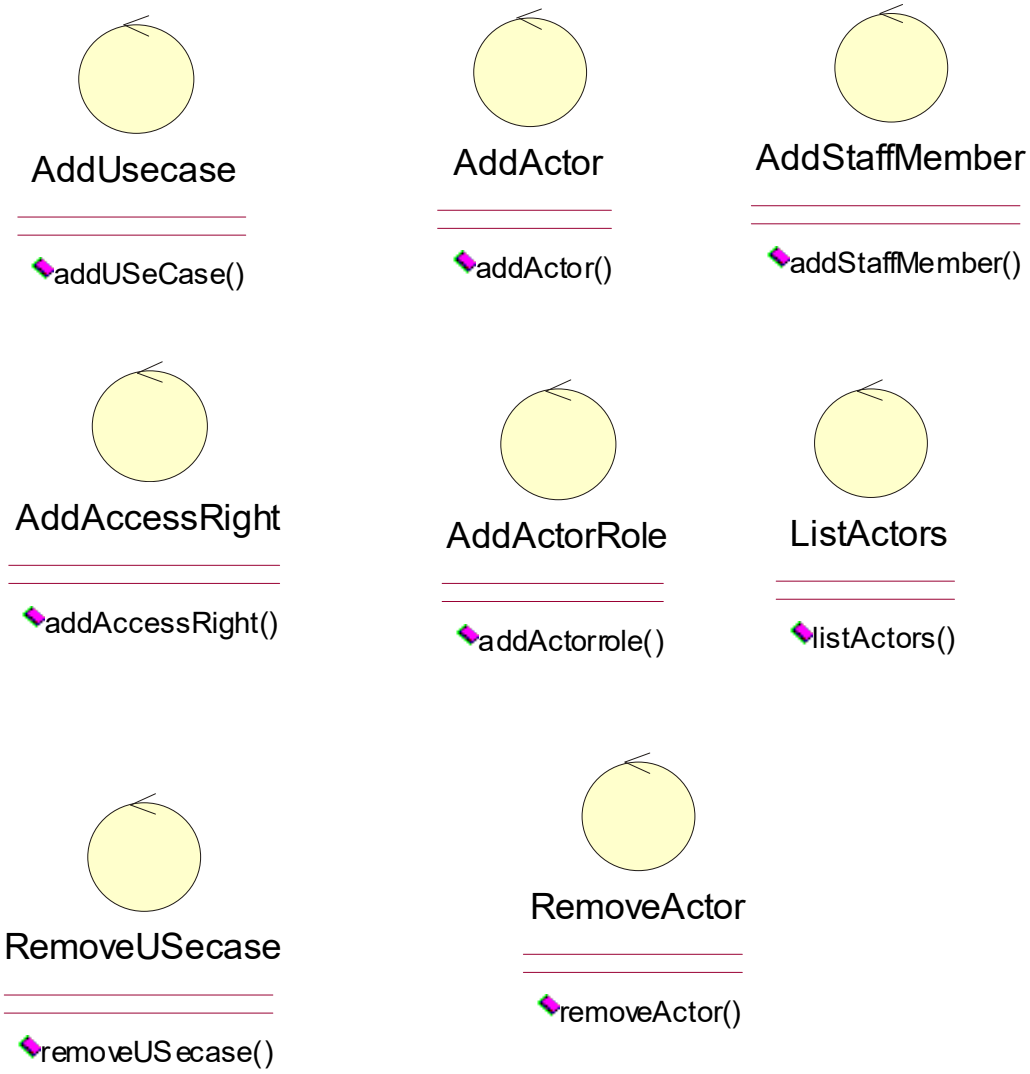


We could use this structure to add a single class, called SecurityManager which provides the API to the functionality in the security package. Or we could add two separate Facade classes, one for the management of the security sub-system (adding staff members etc.), and one for the authentication service used by business applications.

This is shown below in the form of class diagram. Here we added other operations that will be required in order to support the use cases for maintaining the information in the sub-system, for example to list all the actors for a particular use case.



The control classes in the Security Server Control Classes package can probably be designed to be Singletons . The following figure shows the some of control classes in this package.

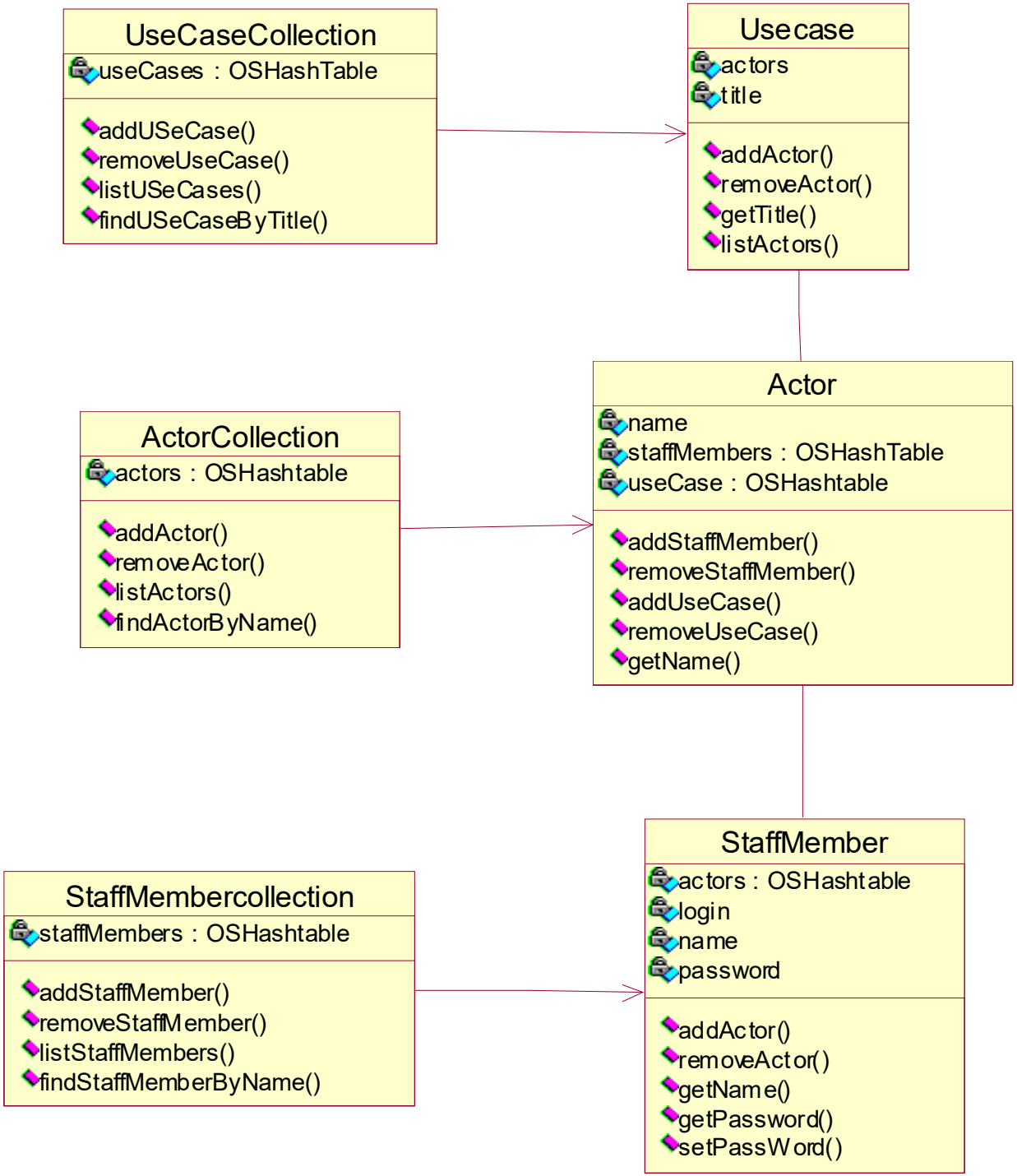


In order to make the security package as reusable as possible, it either needs to make use of whatever data storage mechanisms are used in the application with which it is supplied, or it needs to have its own mechanism for persistent storage. The simplest approach is to provide the security package with its own persistence mechanism. We can use an object-oriented database management system such as Object Store PSE Pro for Java to provide a persistence mechanism without having to worry about brokers and proxies.

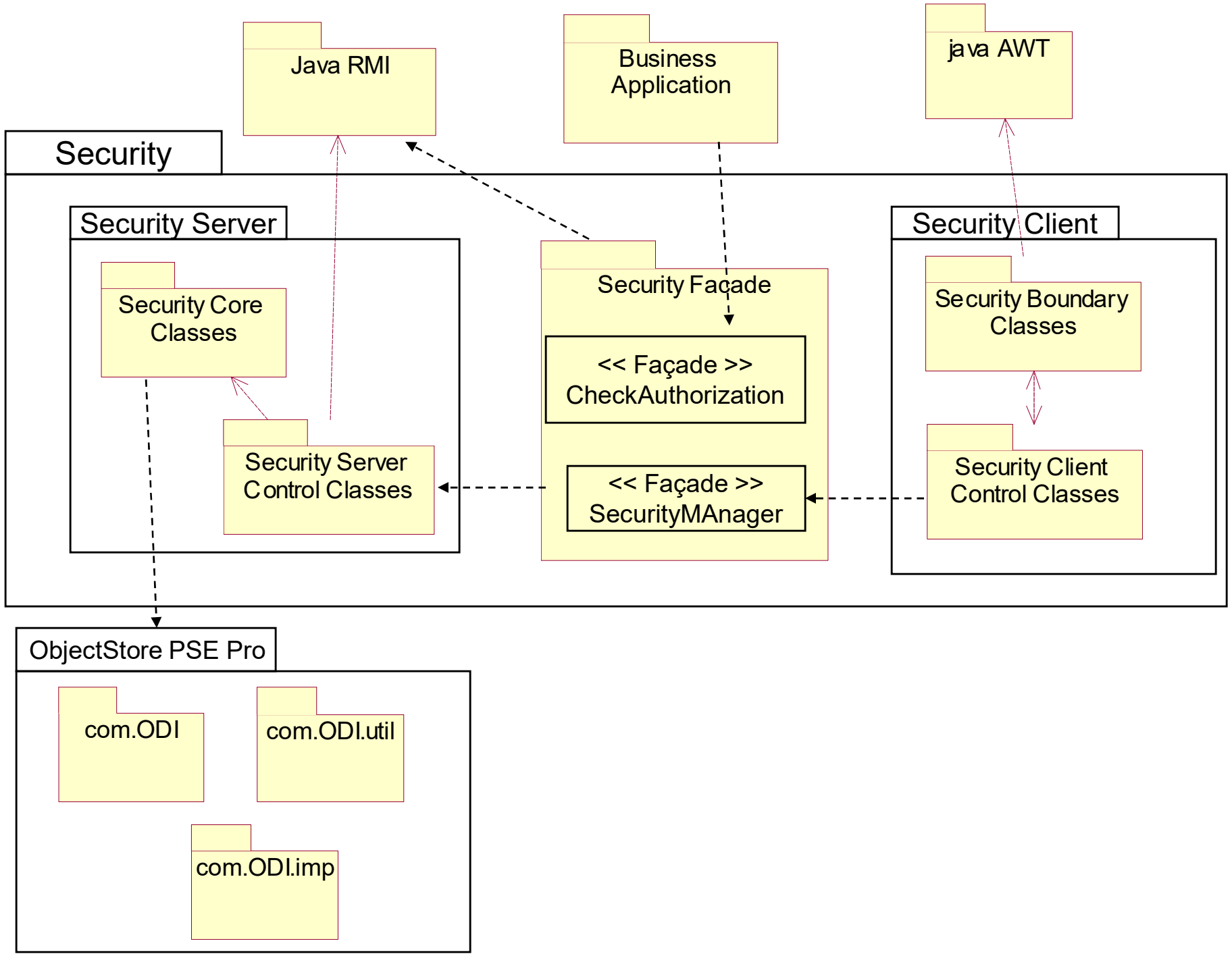
The following design class diagram , we represented collection classes to provide entry points from the control classes to the lists of Actors, UseCases and Staff Members. . We also have added hashtables as collection classes to implement the associations between the classes.

Adding a link between a **UseCase** and an **Actor** means adding the **Actor** to the **UseCase. actors** hashtable and adding the **UseCase** to the **Actor. useCases** hashtable. Mainly here, the collection classes support operations that have been added to the facade class and the control classes.

Design class diagram showing collection classes.


UseCaseCollection
useCases : OSHashTable
addUSeCase()
removeUseCase()
listUSeCases()
findUSeCaseByTitle()
Usecase
actors
title
addActor()
removeActor()
getTitle()
listActors()
Actor
name
staffMembers : OSHashTable
useCase : OSHashtable
addStaffMember()
removeStaffMember()
addUseCase()
removeUseCase()
getName()
ActorCollection
actors : OSHashtable
addActor()
removeActor()
listActors()
findActorByName()
StaffMember
actors : OSHashtable
login
name
password
addActor()
removeActor()
getName()
getPassword()
setPassWord()
StaffMembercollection
staffMembers : OSHashtable
addStaffMember()
removeStaffMember()
listStaffMembers()
findStaffMemberByName()

If we use Java Remote Method Invocation (RMI) to allow client packages to cannect to the Security package, then we also require the dependency on the Java RMI package, which is shown in the following figure. Here also we have , the facade packages in a separate **Security Facade** package. All the operations that update the database will require a reference to an ObjectStore database and must take place in the context of a transaction. The dependencies on the ObjectStore Database and Transaction classes are also reflected in the dependencies .


Java RMI
Business Application
java AWT
Security
Security Server
Security Core Classes
Security Server Control Classes
Security Facade
<< Façade >> CheckAuthorization
<< Façade >> SecurityMAnager
Security Client
Security Boundary Classes
Security Client Control Classes
ObjectStore PSE Pro
com.ODI
com.ODI.util
com.ODI.imp

21. Managing Object-Oriented Projects

Information systems development is a complex activity that requires careful management. The UML techniques highlights the need to plan and manage the whole process. There are inter-dependencies between the artifacts of software development, and their production has to be co-ordinated if the process is to be efficient.

A large software development project may involve many developers, some with specialized skills.

1. The specialist in requirements capture is required early in the project.
2. The expert in ODBMS implementation is needed during design and construction
3. The installation and support teams become involved to some extent during design and construction and more fully when the information system is complete.

The different activities may require different resources whose availability has to be planned. Here, The timing of the installation may be critical for the success of the project. The management process is further complicated by the fact that the sequence of some activities may be significant. For example, testing of a system can only begin when at least some elements have been constructed, though of course test scripts and test harnesses may be prepared early in the project.

1. Resource Allocation and Planning

Given the complexity of project management there is a need for tools and techniques to support the process. Yourdon identifies three particular areas of the management of software development where modelling techniques can play a useful role:

- in the estimation of money, time and people required;
- in assisting the revision of these estimates as a project continues;
- in helping to track and manage the tasks and activities carried out by a team of software developers.

Many tools have been developed to support the management of any type of project, not just those that are focused on software development.

1.1 Critical Path Analysis

The technique known as Critical Path Analysis (CPA) was developed for use on major weapons development projects for the US Navy. Originally known as Project (or Program) Evaluation and Review Technique (PERT), it is also called Network Analysis and it has been widely used on many different types of project.

For the purposes of carrying out a critical path analysis, a project is viewed as a set of activities or tasks, each of which has an expected duration. Completion of an activity corresponds to a milestone or event for the project. Each milestone also represents the start of activities that are directly dependent on the completion of the predecessor or predecessors.

CPA is based on an analysis of sequential dependencies among the activities, and uses the expected duration for each task to derive an estimate of the overall duration of the project. Particularly, it identifies any inter-task dependencies that are critical to the project duration-collectively these are known as the *critical path.* The preparation of a CPA chart involves the following steps.

A. List All project activities and Milestones

A sample list for the development of the Agate Campaign Management system is shown in the following figure. Each activity is labelled with a letter and has a short description. The third column in the table contains a milestone number that represents the completion of that activity.

Activity	Description	Milestone	Preceding activities	Expected duration	Staffing
A	Interview users	2	-	5	2
B	Prepare use cases	3	-	2	See A
C	Review use cases	4	A,B	2	3
D	Draft screen layouts	5	C	2	2
E	Review screens	6	D	2	2
F	Identify classes	7	C	2	3
G	CRC analysis	8	F	4	3
H	Prepare draft class diagram	9	F	5	3
I	Review class diagram	10	G,H	4	4

B. Determine the dependencies among the activities

Some activities cannot start until another has been completed. The preceding activities are listed in column 4 of the previous figure. For example, the activity Review use cases must be completed before the activity Identify classes can begin.

C. Estimate the duration of each activity

There are several different approaches for estimating the duration of each activity, due to the uncertainty involved in estimating task duration. One that is used widely is given by the following formula:

$$ED = \frac{MOT + (4 \times MLT) + MPT}{6}$$

where *ED* is the expected duration of a task, *MOT* is the most optimistic time, *MLT* is the most likely time and *MPT* is the most pessimistic time for its completion. The *ED* is thus a weighted average of the three estimates.

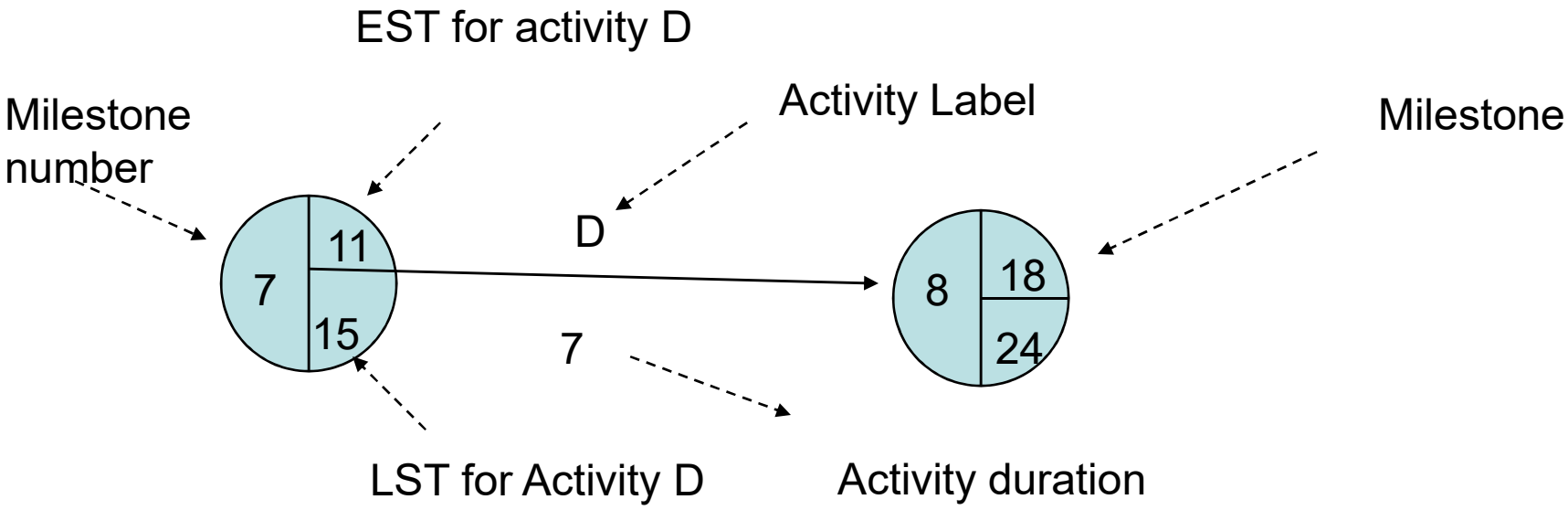
Each *MOT* assumes that a task will not be delayed, even by likely events such as employee absence. The *MPT* assumes that most things that can go wrong will go wrong, and that completion of the activity will be delayed to the maximum extent. Equipment will arrive late, technical problems will occur and some staff will be ill. The *EDs* are entered in the fifth column of the previous table. Note that in this example the staff requirements for activities A and B have been treated as one since the two activities are highly interdependent.

D. Draw the CPA chart

Two types of notations can be used for CPA charts –

1. Activity on the node diagrams representation
2. Activity on the arrow diagrams representation.

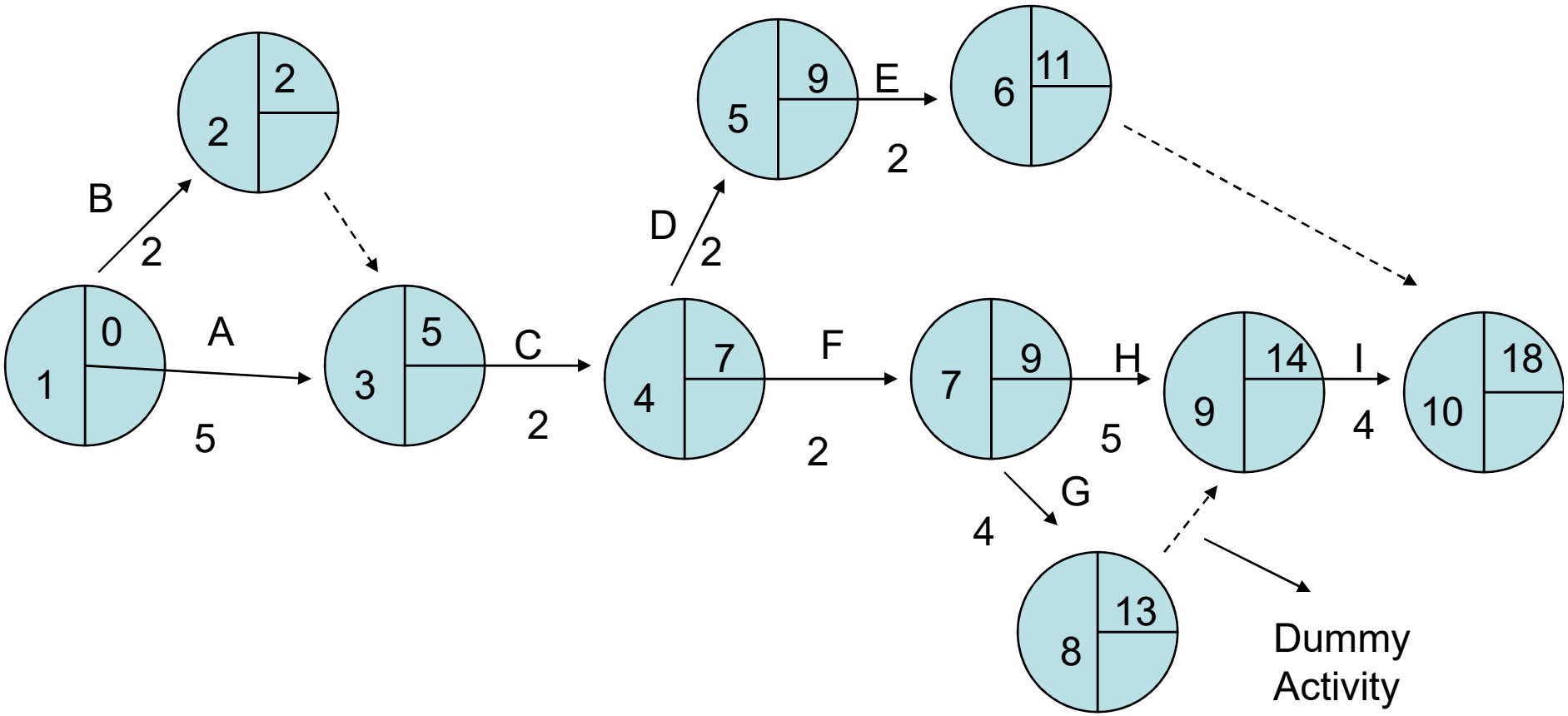
Both the types gives the same information, but they look very different from each other. Consider the representing CPA chart by ‘Activity on the arrow' notation. In this style each milestone is represented by a circle divided into three compartments. One compartment is labelled with the milestone number, and the other two will hold the earliest start time (EST) and the latest start time (LST) for all activities that begin at that milestone. The following figure represents Activity on the arrow diagrams notation.



CPA notation

The first draft of a CPA chart shows dependencies between activities and their expected durations, since this information is known or can be estimated before the diagram is drawn.

The following figure is the partially completed CPA chart which represents graphically the activity precedence's corresponding to the Agates' previous table.



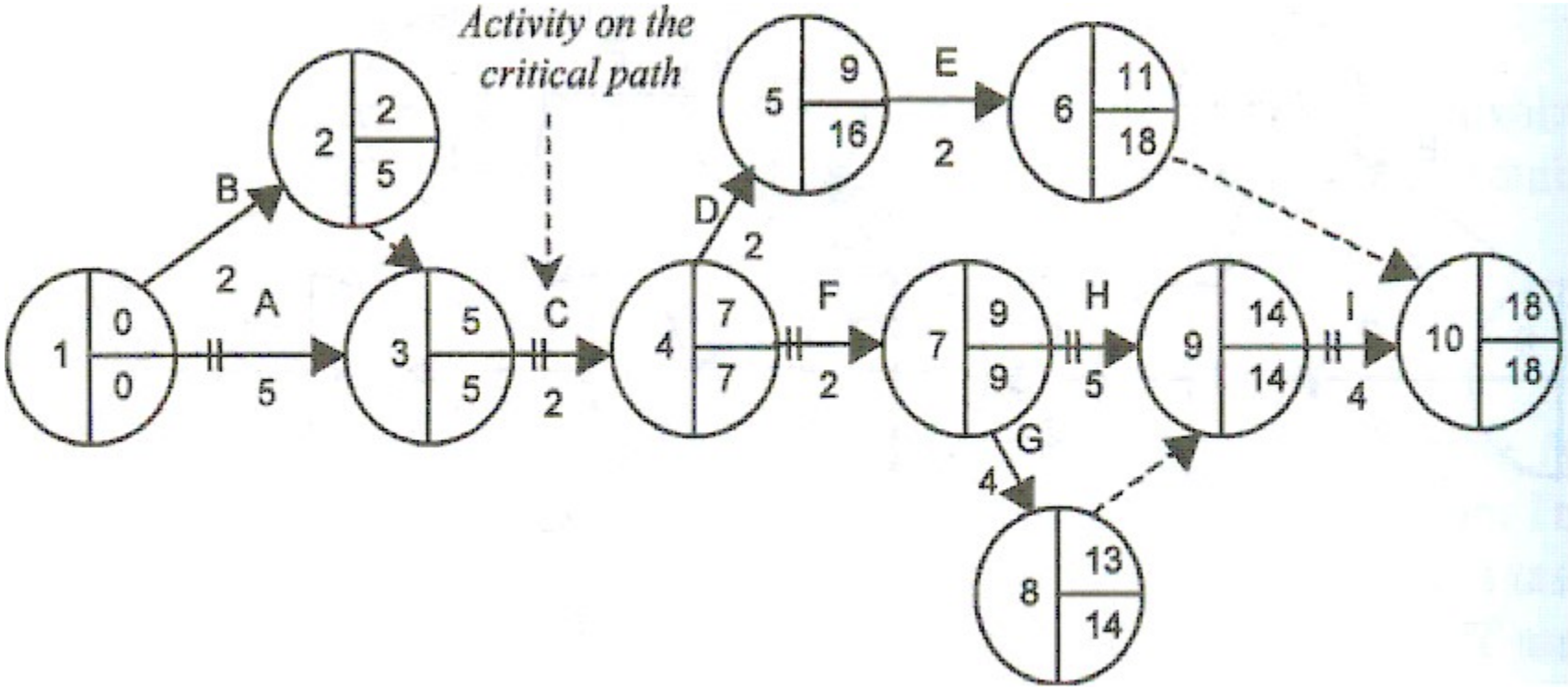
Here, ***dummy activity*** between milestones 8 and 9 (there are others between 2 and 3 and between 6 and 10). This is due to , activities H and G both depend on milestone 9 (the completion of activity F), and are also both predecessors to milestone 10 (where activity I begins). Since H and G may not necessarily finish at the same time, an extra milestone is needed for one of these events. In effect, milestone 8 represents the completion of G, regardless of whether or not H has finished. A dummy activity (with ED of 0) then needs to be introduced in order to connect milestone 8 to milestone 9, thus preserving the sequence of dependencies.

The next step is to enter an earliest start time for each milestone. This is done by working through the diagram from the very first milestone to the very last (sometimes called a *forward pass).* It is a convention that the EST for the first milestone is set to 0. The EST for most other milestones is calculated simply by adding the EST of the immediately preceding activity to its duration. For example, activity C (the immediate predecessor for D) has an EST of 5 and an ED of 2, therefore the EST for activity D is 7. Where an activity has two or more predecessors, its EST is determined by the predecessor that has the latest completion time. For example, activity I is dependent upon the completion of both G and **H,** so its EST is set to the later of the two calculations. **In** general the EST for any milestone is set to the earliest time that *all* predecessor activities can be completed.

The next step requires completing the latest start time for each milestone. The latest start time is entered by working back from the last milestone (sometimes this is called a *backward pass).* The LST for the last milestone is set equal to its EST. Each preceding LST is then calculated as follows. The LST for most milestones equals the LST for its successor minus the ED of the intervening activity. For example, the LST for milestone 9 is 18 - 4 = 14 (the LST for milestone 10 minus the ED for activity I). Milestone 7 presents more of a problem as this has two successor activities, G and H. **In** such cases, two calculations are performed (or more, if there are more than two successors) and the earlier of the two answers is taken. For example, if the LST for milestone 7 were determined purely by activity G, this would give 14 - 4 = 10 (the LST for milestone 8 is 14). This would mean that activity G could afford to begin as late as time 10 without delaying any other activities. But a similar calculation for activity H gives 14 - 5 = 9. This means that if H begins later than time 9, its completion will be delayed beyond time 14, which in turn would delay milestones 9 and 10 and thus also activity I and the project completion. The LST for milestone 7 is therefore set to 9. **In** general, the LST for a milestone is set to the latest time that allows *every* activity that begins at that milestone to be completed by the LST for its succeeding milestone.

Identify critical path

Once all LSTs have been entered onto the diagram, the *slack time* (or *float)* for each activity can be calculated. This is the difference between an activity's EST and its LST, and it represents the time by which that particular activity can be delayed without affecting the overall duration of the project. The path through all milestones that have a slack time of 0 is called the *critical path.* This is indicated by a double bar across activity arrows that connect the milestones. These milestones, and their intervening activities, are critical to the completion of the project on time. Milestones that have an EST that is different from their LST are not critical, in the sense that they have some scheduling flexibility. The completed diagram is shown below with critical path for the Agate Ltd.



A CPA chart is an effective tool for identifying those activities whose completion is critical to the completion of a project on time. If any activity that is on the critical path falls behind schedule then the project as a whole is behind schedule. However, while critical activities naturally receive the closest scrutiny, all project activities should be monitored. Delay even in a non-critical activity can, if it is sufficiently severe, alter the critical path.

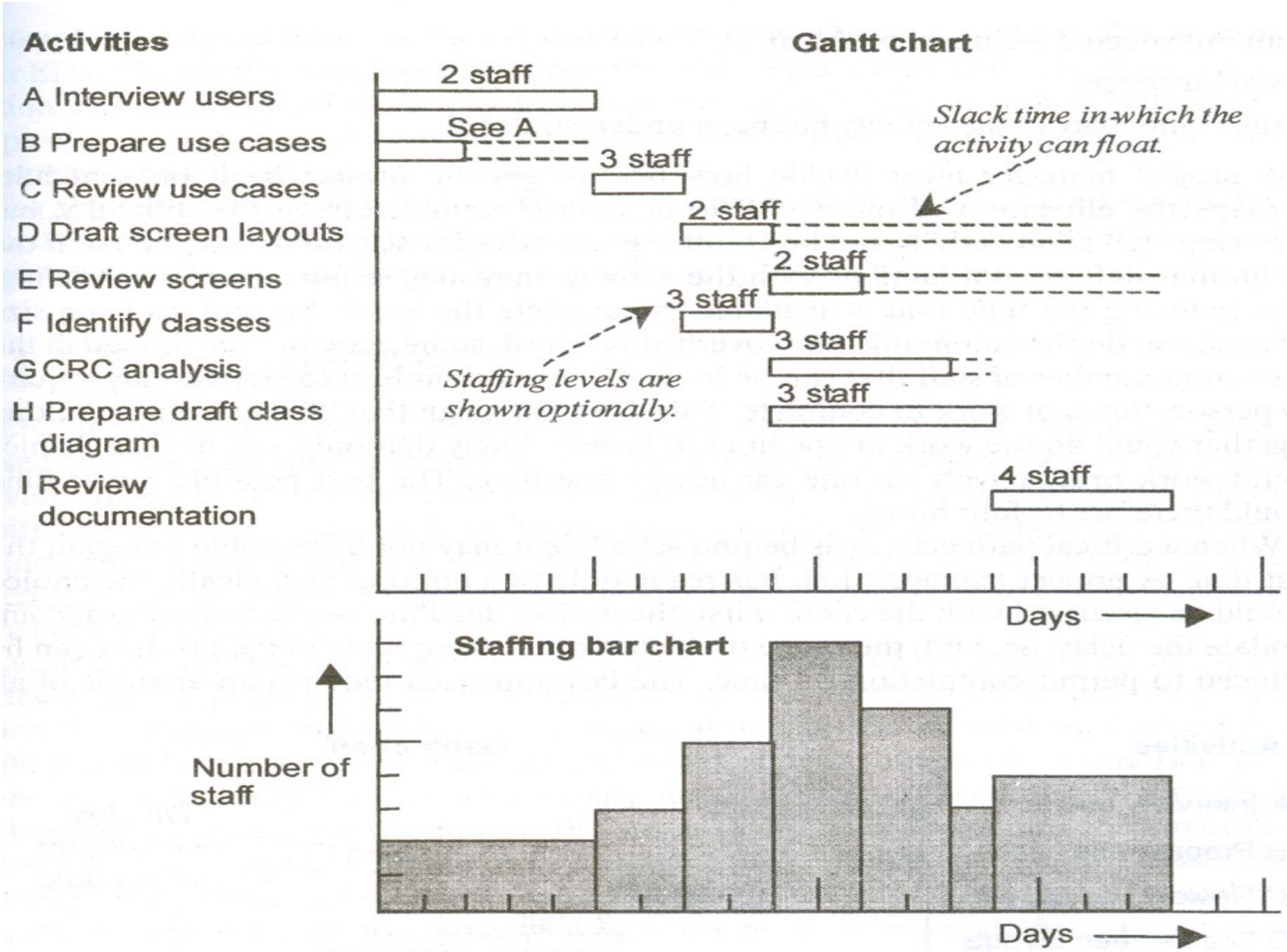
And here, PERT is more elaborate than CPA, using statistical measures in addition to critical path analysis, but the terms are generally used synonymously. CPA charts are sometimes known as activity diagrams but this name introduces confusion with UML activity diagrams.

2. Gantt charts

The Gantt chart is a simple time-charting technique that uses horizontal bars to represent project activities. The horizontal axis represents time and is often labelled with dates or week numbers so that the completion of each activity can be monitored easily. Activities are listed vertically on the left of the chart, and the length of the bar for each activity corresponds to its ED.

A Gantt chart shows the overlap of activities clearly and this provides an effective way of considering alternative resource allocations. The Gantt chart can be drawn with either dashed lines or dashed boxes that show the slack time for non-critical activities.

A Gantt chart can also be used to convert into a stacked bar graph that can be used to show the way that the total resource allocation for a project changes over time. The following Figure shows a Gantt chart and a staffing bar chart for the Agate advertising sub-system project.



igure 21.5 Gantt Chart with staffing bar chart.

The staffing chart is derived as follows. The final column of project activity table gives a staff allocation for each activity. The Gantt chart is read vertically for each successive time interval to calculate the total number of staff required for all project activities combined. The result is shown as a vertical bar that indicates the total staffing required at that time.

Activities that are not on the critical path can have their start time adjusted. For example, activity E cannot begin until time 9 at the earliest, but it could start as late as time 16 without affecting project completion. A project manager can adjust the resource profile to accommodate staff availability, a process known as resource *smoothing.* For example, activities E, G and H can occur concurrently. H is on the critical path and cannot be moved, but the slack time for E allows it to begin at time 16 instead of time 9. The manager can minimize the overall resource requirement by rescheduling activity E to begin at time 14. This should be done with care, however, as it may move an activity onto the critical path.

The following figure shows the smoothed resource profile.

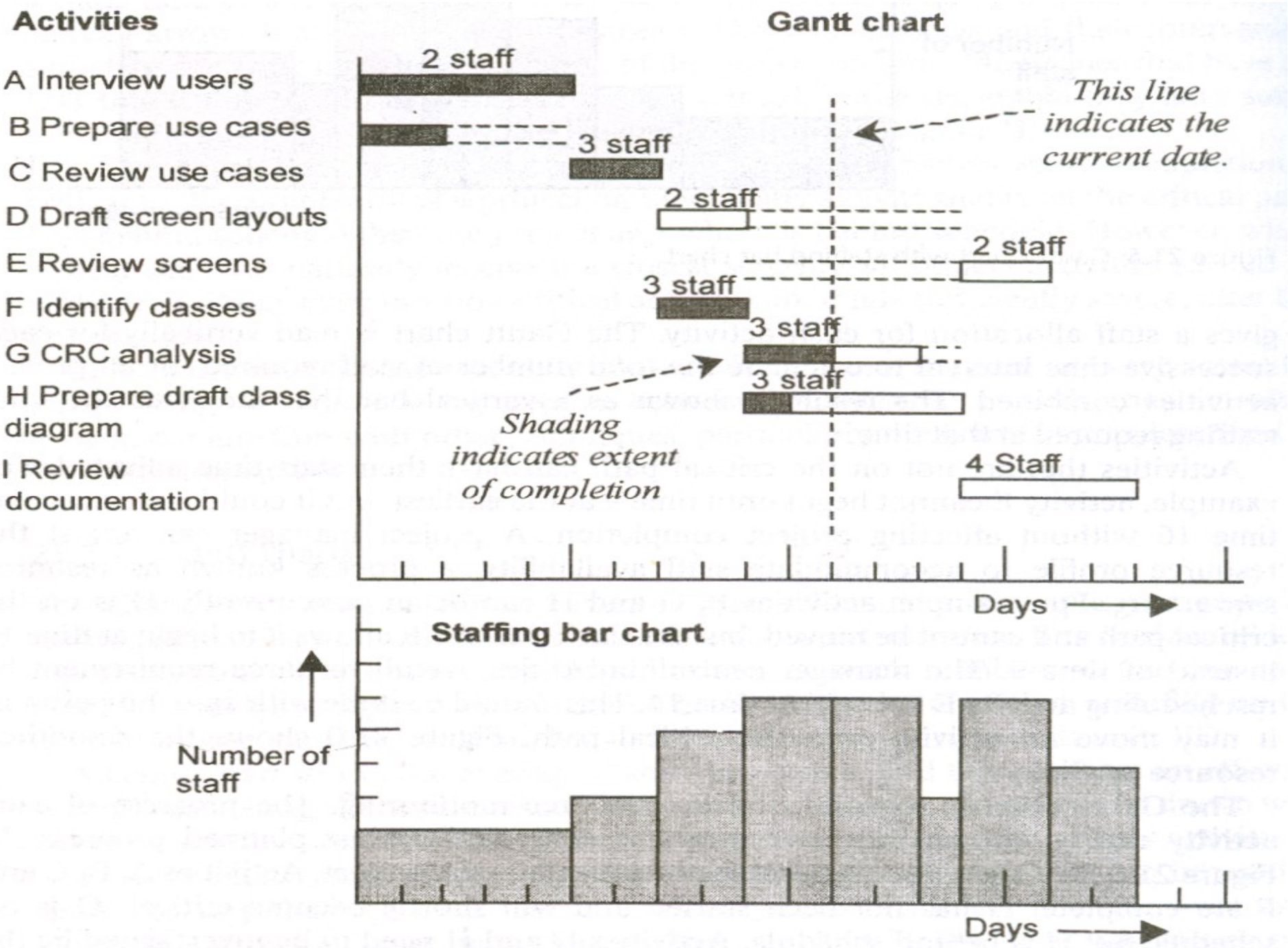


Figure 21.6 Gantt chart showing activity completion and smoothed staff profile.

The Gantt chart is a useful tool for resource monitoring. The progress of each activity can be shown independently and compared against planned progress. In Previous figure the Gantt chart reflects the current state of a project. Activities A, B, C and F are complete. D has not been started and will shortly become critical. G is on schedule but H is behind schedule. Activities D and H need to be investigated by the project manager. Possible reasons for the delay include:

- an unexpected technical problem;
- staff absence;
- the complexity of the activity has been underestimated

When a critical path activity is behind schedule it may not be possible to regain the lost time. A project manager then has really only two options, and ideally the choice should be discussed with the client.

1. The project deadline can be moved to accommodate the delay.
2. The scope of the project or the quality of the product can be reduced to permit completion on time.

The last approach requires an analysis of all uncompleted critical path activities in order to identify what can be omitted to reduce their EDs. The resultant changes may alter the critical path, and activities whose completion was not critical before may now become critical. Any attempt to reduce the scope of a project requires user involvement to ensure that only non-critical features are omitted, particularly during the first increment.

2. Managing Iteration

•Prototyping is an iterative development activity.
•We need a criteria to control the number of iterations.
•At the end of an iteration the product, say a prototype, is evaluated against pre- defined objectives.
•But it is difficult to determine whether the objectives of the iterative activity have been achieved or not.
• Let us suppose that the interface for the Agate Campaign Management system is to be developed by prototyping, with the explicit objective of producing an interface with which the campaign staff are happy.
• Although this objective may be worth-while it is of little use for the management of the activity.
• Imagine that the users are never completely happy at the end of any iteration.
• They will continue to suggest further improvements without end.
• As the process continues, the nature of the modifications that are suggested at each iteration will change.
• Over time the improvements will become cosmetic and ultimately peripheral to the utility of the system.
• It would be sensible to end the iterative process before this point is reached.
A more suitable objective for the exercise might be phrased as follows:

Continue the iterations until fewer than five cosmetic changes are requested on a single iteration.

• It is still not clear how many iterations will be needed to satisfy this criterion.

• If the project has unlimited time and an unlimited budget this may not be a problem but this is unlikely to be the case.

• Additional criteria can be added to tighten up the objectives, such as the following.

The prototyping phase must be completed before the end of mentioned month say October and must not exceed 50 developer-hours.

3. Dynamic Systems Development Method (DSDM) :

The Dynamic Systems Development Method (DSDM) is a management and control framework for rapid application development (RAD). The distinction between RAD and prototyping is sometimes unclear. A RAD approach aims to build a working system rapidly while a prototyping approach also builds rapidly, but usually only produces a partially complete system, typically to confirm some aspect of the requirement. Because both approaches aim to build software quickly, similar development environments are used and one approach to prototyping continues the development of a prototype incrementally until it becomes a working system. In effect this is a RAD development approach.

The traditional waterfall approach to systems development has deficiencies, particularly the time taken to deliver a working system and the inflexibility of the approach to requirements change. Iterative approaches to development can also be problematic , As mentioned in the above management of iterations is sometimes difficult to cease the iterations when they become unproductive.

Later RAD became more popular and was viewed as a way of matching systems development to the fast changing needs of business. However, there were until recently no commonly accepted structures for either the use or the management of RAD.

Later in 1994 the DSDM was formed to produce an industry standard definition of the RAD process and DSDM was subsequently defined. The DSDM framework defines structure and controls to be used in a RAD project but does not specify a development methodology. DSDM may be used with either an object-oriented or a structured methodology.

DSDM takes a fundamentally different perspective on project control. Rather than viewing requirements as fixed and attempting to match resources to the project, DSDM fixes resources for the project, fixes the time available and then sets out to deliver only what can be achieved within these constraints.

DSDM is based upon the following nine principles.

1. Active user involvement is imperative. Many other approaches effectively restrict user involvement to requirements acquisition at the beginning of the project and acceptance testing at the end of the project. In DSDM users are members of the project team and include one known as an 'Ambassador' user.
2. DSDM teams are empowered to make decisions. A team can make decisions that refine the requirements and possibly even change them without the direct involve-ment of higher management.

3. The focus is on frequent product delivery. A team is geared to delivering products in an agreed time period and it selects the most appropriate approach to achieve this. The time periods are known as timeboxes and are normally kept short (2 to 6 weeks). This helps team members to decide in advance what is feasible. Products can include analysis and design artefacts as well as working systems.
4. The essential criterion for acceptance of a deliverable is fitness for business purpose. DSDM is geared to delivering the essential functionality at the specified time.
5. Iterative and incremental development is necessary to converge on an accurate business solution. Incremental development allows user feedback to inform the development of later increments. The delivery of partial solutions is considered acceptable if they satisfy an immediate and urgent user need. These solutions can be refined and further developed later.
6. All changes during development are reversible. If the iterative development follows an inappropriate development path then it is necessary to return to the last point in the development cycle that was considered appropriate. Changes are limited within a particular increment.
7. Requirements are initially agreed at a high level. Once requirements are fixed at a high level they provide the objectives for prototyping. The requirements can then be investigated in detail by the DSD M teams to determine the best way to achieve them. Normally the scope of the high level requirements is not changed significantly.

8. Testing is integrated throughout the life cycle. Since a partially complete system may be delivered it must be tested during development, rather than after completion. Each software component is tested by the developers for technical compliance and by user team members for functional appropriateness.

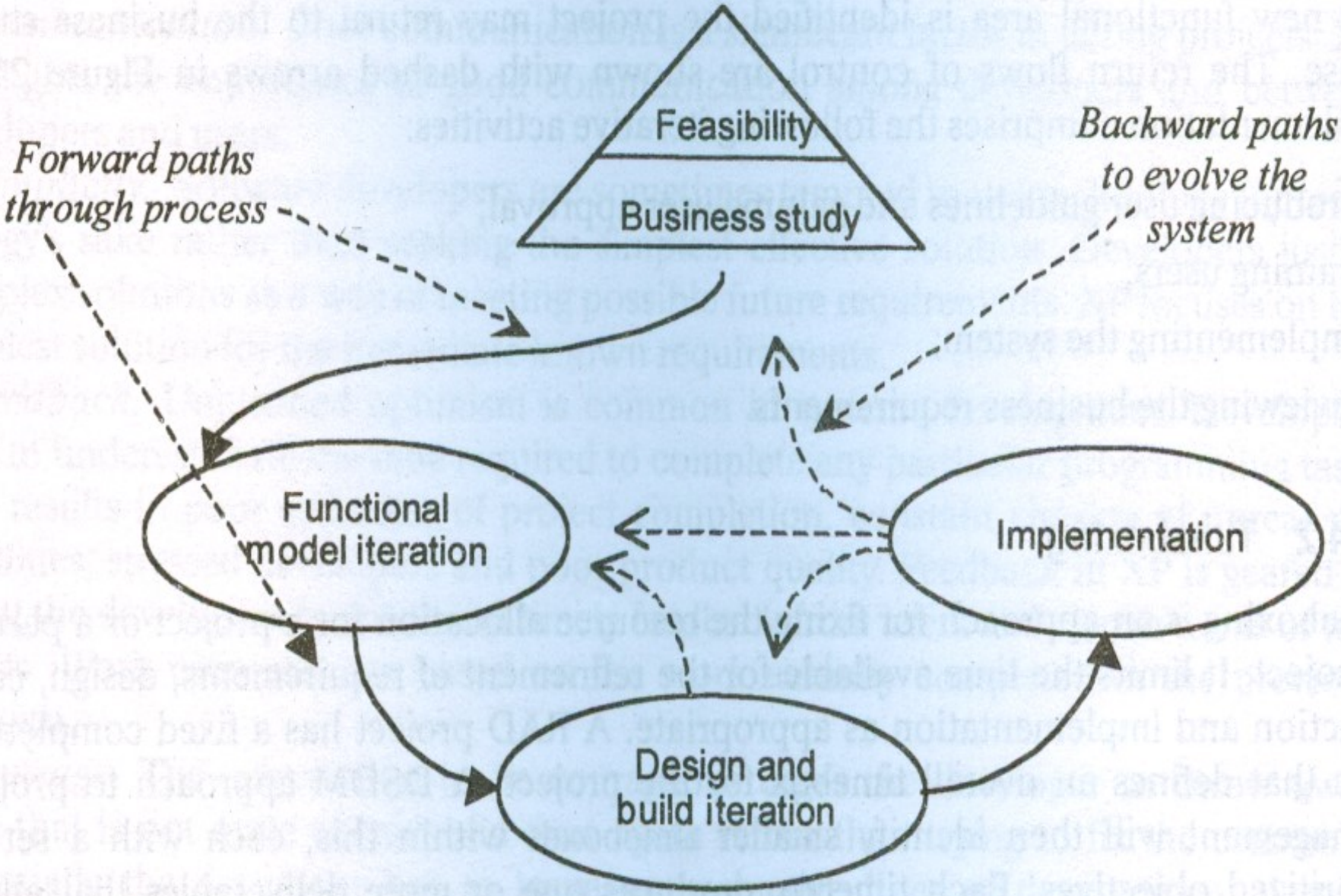
9. A collaborative and co-operative approach between all stakeholders is essential. The emphasis here is on the inclusion of all stakeholders in a collaborative development process. Stakeholders not only include team members, but others such as resource managers and the quality assurance team.

The DSDM life cycle

The DSDM life cycle has the following phases:

- ✓ feasibility study,
- ✓ business study,
- ✓ functional model iteration,
- ✓ design and build iteration,
- ✓ implementation.

The relationships between the phases are shown graphically in the following figure, here last three are actually iterative processes.



Simplified model DSDM Life Cycle

1. **The *feasibility study* phase** determines whether the project is suitable for a DSDM approach or not. It typically lasts only weeks, whereas the feasibility stage can last months on a traditionally run project. The study should also answer the following questions :

- Is the computerized information system technically possible?
- Will the benefit of the system be outweighed by its costs?
- Will the information system operate acceptably within the organization?

2. **The *business study* phase** identifies the overall scope of the project and results in agreed high level functional and non-functional requirements. Maintainability objectives are set at this stage and these determine the quality control activities for the remainder of the project. There are three levels of maintainability:

- maintainable from initial operation;
- not necessarily maintainable when first installed but this can be addressed later;
- short life-span system that will not be subject to maintenance.

3. **The *functional model iteration* phase** is concerned with the development of proto-types to elicit detailed requirements. The intention of DSDM is to develop prototypes that can ultimately be delivered as operational systems, so these must be built to be sufficiently robust for operational use and also to satisfy any non-functional require-ments such as performance. When completed the functional model comprises high level analysis models and documentation together with prototypes that are concerned with detailed functionality and usability.

During the functional model iteration the following activities are undertaken:

- the functional prototype is identified;
- a schedule is agreed;
- the functional prototype is created;
- the functional prototype is reviewed.

4. The *design and build iteration* phase is concerned with developing the prototypes to the point where they can be used operationally. The distinction between the functional model iteration and the design and build iteration is not clear-cut and both phases can run concurrently. The activities for the design and build iteration phase are very similar to those described above for the functional model iteration phase.

5. The *implementation* phase deals with the installation of the latest increment including user training. At this point it is important to review the extent to which the requirements have been met. If they have been fully satisfied the project is complete. If some non-functional requirements have yet to be addressed the project may return to the design and build iteration phase. If some element of functionality was omitted due to time constraints the project may return to the functional model iteration phase. If a new functional area is identified the project may return to the business study phase. The return flows of control are shown with dashed arrows in figure . Implementation comprises the following iterative activities:

1. producing user guidelines and gaining user approval; 2. training users;
3. implementing the system; 4. reviewing the business requirements

4. Timeboxing

Timeboxing is an approach for fixing the resource allocation for a project or a part of a project. It limits the time available for the refinement of requirements, design, con-struction and implementation as appropriate.

A RAD project has a fixed completion date that defines an overall timebox for the project. A DSDM approach to project management will then identify smaller timeboxes within this, each with a set of prioritized objectives. Each timebox produces one or' more deliverables that allow progress and quality to be assessed. Within a timebox the team have three major concerns. They must first carry out any investigation needed to determine the direction that should be taken for that part of the project. They must then develop and refine the specified deliverables. Finally they must consolidate their work prior to the final deadline.

It is sometimes difficult to prioritize the requirements that will be actioned during a timebox. One way of doing this is to apply the set of rules that are known as the *MoSCoWrules* (for Must ... Should ... Could ... Want).

Must have requirements are crucial. If these are omitted the system will not operate. In DSDM the set of *Must have* requirements are known as the minimum usable subset.

Should have requirements are important but if necessary the system can operate usefully without them.

Could have requirements are less important and provide less benefit to the user. *Want to have but will not have this time around* requirements can reasonably be left for development in a later increment.

All of these requirements are important for the final system but not to the same extent. If the full set cannot be addressed within a timebox, the MoSCoW categorization can be used to focus the requirements in an appropriate way.

5. Extreme Programming

Extreme Programming (XP) is a novel combination of elements of best practice in systems development. It incor-porates a highly iterative approach to development. It has become well known in a relatively short period of time for its use of ***pair programming*** though it encompasses various other important ideas. Pair programming involves writing the program code in pairs and not individually.

Beck identified the following four underlying principles of XP as communication, simplicity, feedback and courage.

Communication. Poor communication is a significant factor in failing projects, XP highlights the importance of good communication among developers and between developers and users.

Simplicity. Software developers are sometimes tempted to use technology for tech-nology's sake rather than seeking the simplest effective solution. Developers justify complex solutions as a way of meeting possible future requirements. XP focuses on the simplest solution for the immediate known requirements.

Feedback. Unjustified optimism is common in systems development. Developers tend to underestimate the time required to complete any particular programming task. This results in poor estimates of project completion, constant chasing of unrealistic deadlines, stressed developers and poor product quality. Feedback in XP is geared to giving the developers frequent and timely feedback from users and also in terms of test results. Work estimates are based on the work actually completed in the previous iteration.

Courage. The exhortation to be courageous urges the developer to throwaway code that is not quite correct and start again rather than trying to fix the unfixable. Essentially the developer has to leave unproductive lines of development despite personal investment in the ideas.

Requirements capture in XP is based on *user stories* that describe the requirements. These are written by the user and form the basis of project planning and the development of test harnesses. User stories are very similar to use cases though some proponents of XP suggest that there are key differences in granularity. A typical user story is about three sentences long and does not include any detail of technology. When the developers are ready to start writing the system they get detailed descriptions of requirements face to face with the customer.

The following are the main Activities in XP

- *The planning game* involves quickly defining the scope of the next release from user priorities and technical estimates. The plan is updated regularly as the iteration progresses.

- The information system should be delivered in *small releases* that incrementally build up functionality through rapid iteration.

- A unifying *metaphor* or high level shared story focuses the development.
- The system should be based on as *simple design.*
- *Programmers prepare unit tests in advance of software construction and customers define acceptance tests.*
- The programe code should be restructured to remove duplication, simplify the code and improve flexibility-this is known as *refactoring.*
- Pair programming means that code is written by two programmers using one workstation.
- The code is owned collectively and anyone can change any code.
- The system is integrated and built frequently each day. This gives the opportunity for regular testing and feedback.
- Normally staff should work no more than forty hours a week.
- A user should be a full-time member of the team.
- All programmers should write code according to agreed standards that emphasize

good communication through the code.

The XP approach is best suited to projects with a relatively small number of programmers - say no more than ten. **In** XP it is critical to maintain clear communicative code and to have rapid feedback. If a project precludes either of these then XP is not the most appropriate approach. One key feature of XP is that the code itself is the design documentation. This runs counter to some aspects of the approach suggested in this book. We have suggested that requirements are effectively analyzed and suitable designs produced though the use of visual models using UML.

6. Software Metrics

Planning and managing a software development project requires the estimation of the resources required for each of its constituent activities. A resource estimate for an activity can be based upon subjective perceptions of the activity or it can be based upon measurements of size and complexity, either of the activity itself or of the artefact that is produced.

A *software metric* is a measure of some aspect of software development, either at project level - usually its cost or its duration - or at the level of the application - typically its size or its complexity.

Software metrics can be divided broadly into two categories:

1. ***process metrics*** that measure some aspect of the development process
2. ***product metrics*** that measure some aspect of the software product.

Examples of process metrics are

1. the project cost to date
2. the amount of time spent so far on the project.

Product metrics relate to the information system that is under development. One of the simplest product metrics is the number of classes in an analysis class diagram.

Software metrics can also be categorized as ***result metrics*** or ***predictor metrics***, which are used respectively to measure outcomes and to quantify estimates. The current cost of a project is a result metric .

A measure of class size (a crude measure might be a simple count of attributes and operations) would be a predictor metric, so called because it can be used as a basis for predicting the time that it will take to produce program code for that class. Result metrics are also known as control metrics since they are used to determine how management control should be exercised.

The term 'predictor metric' is generally applied only to a measure of some aspect of a software product that is used to predict some other aspect of the product or of the project progress. Predictor metrics are not used individually for estimation. The results obtained from their application to a project may indicate, for example, that the system will be difficult to maintain or that it may offer very low levels of reuse. Since neither outcome is desirable, managers may attempt to change the design for the system or the process for its development in order to improve the system.

The validity of predictor metrics is based upon three assumptions :

- there is some aspect of a software product that can be accurately measured;
- there is a relationship between the measurable aspect and some other relevant characteristics of the product;
- this relationship has been validated and can be expressed in a model or a formula.

The last of these assumptions suggests that a significant volume of historical data must be collected so that an appropriate statistical analysis can validate the relationship. In practice this is only feasible if the data collection is automated.

A number of metrics have been identified for use with structured analysis and design approaches. For example, De Marco developed a complexity metric known as the **Bang Metric** for use on structured analysis projects. Other metrics that focused on the degree of coupling and cohesion between program modules have also been suggested for use with a structured design approach.

A number of authors have identified metrics for object-oriented systems development , which includes the following a list of desirable features as part of a general description of a useful metric given by De Champeaux :

- either it is elementary and focuses on a single well-defined aspect, or it is an aggregation of elementary metrics;
- it is suitable for automated evaluation;
- gathering the metric data is not too costly;
- the metric can be measured numerically and arithmetic operations are meaningful

De Champeaux listed a series of quality metrics, for example, a dependency metric that provides a measure of the stability of a system. When applied to a package or a sub-system this measures the degree of inter-package or inter-sub-system coupling. It is calculated by the following formula:

I = (CE) / (Ca + Ce)

where **I** is the instability of the system, *Ca* is the level of afferent coupling (the number of classes outside the package that depend on classes within the package), and *Ce* is the level of efferent coupling (the number of classes outside the package upon which classes within the package depend).

When **I** is zero the package is maximally stable and has no dependencies on classes in other packages. When *I* is 1 the package is maximally unstable and has dependencies only on classes outside itself.

7. Process Patterns

Coplien (1995) has defined a pattern language that is focused on the development process. The pattern language comprises both organization and process patterns. As with design patterns , Process patterns capture elements of experience as problem - solution pairs. The process patterns address issues such as team selection,
organizational size,
team structure,
the roles of Team members and so on.

Conway's Law discusses how the architecture of the system comes to reflect the organizational structure or vice versa.

Mercenary Analyst is concerned with producing project documentation successfully. This pattern suggests that it is frequently more effective to hire a technical writer who can focus solely on the documentation.

8. Legacy systems

Legacy systems means any computerized information system that has been in use for some time, that was built with older technologies may be using a different development approach at different times and, most importantly, that continues to deliver benefit to the organization. Most computerized information systems interact with other computerized information systems. They may share data, the output from one may be an input to another and so on. Any new information system is likely to need to interact with older legacy systems that have not been built using the same technologies. Redeveloping legacy systems so that they interact appropriately with new systems is likely to be prohibitively expensive and probably involves too much risk. These legacy systems may be critical to the operation of the organization. The problem is one of integrating new object-oriented systems with non-object-oriented systems.

One strategy that enables the interoperation of old and new is the use of an *object wrapper* . An object wrapper functions as an interface that surrounds a non-object-oriented system so that it presents an interface suitable for use with new object-oriented systems. Essentially the old system appears to be object-oriented. The form of the wrapper depends on the nature of the legacy system. Where the old system uses text or form-based screen interfaces the wrapper may involve program code that reads data from the screen and writes data to the screen (sometimes known as screen scrapers) using some form of virtual terminal.

When an organization embarks upon object-oriented software development there may be an intention to migrate all existing software to the new technologies. The cost and risk involved may constrain this but where it is feasible to migrate systems it is important to manage the process carefully. Wrappers may be used initially to provide an object-oriented interface to the system. Then, once the system has been wrapped, it can be redeveloped without affecting its interface to other systems. A further variation on this approach is to use the Facade pattern to wrap sub-systems so that they can be migrated incrementally.

22.System Development Methodologies

1. Method and Methodology

The techniques of system development must be organized into an appropriate developmental life cycle if they are to work together. For example, once an analyst has constructed collaboration diagrams for the main use cases, should the next steps be to convert these into sequence diagrams and write operation specifications, or should he or she now concentrate on preparing a class diagram and developing inheritance and composition structures. All of these tasks to be completed at this point,using UML methods by the Analyst.

The ***method*** of a project is the term given to the particular way that the tasks in that project are organized. Sometimes this is called the ***process* of software development**, although process can also have a more all-embracing meaning, that includes what the tasks are, how they are carried out and how they are organized.

The words 'method' and 'methodology' are used interchangeably by many authors, but their meanings actually differ in a significant way. In order to plan and organize for the next project, project managers must be able to think at a still higher level. It is at this level that the term 'methodology' applies.

A method is a step-by-step description of the steps involved in doing a job. Since no two projects are exactly alike, any method is specific to one project. A methodology is a set of general principles that guide a practitioner or manager to the choice of the particular method suited to a specific task or project. Or, to put it in familiar object-oriented terms, a methodology is a type while a method is its instantiation on one project.

The following figure summarizes the different levels of abstraction involved.

Increase level of abstraction	**Example of application**	**Typical product**
Task	Developing a first-cut class diagram for FoodCo.	A specific version of the FoodCo class diagram.
Technique	Description of how to carry out a technique, e.g. UML class modelling.	Any UML class diagram.
Method	Specific techniques used on a particular project (e.g. FoodCo uses cases, class model, collaboration diagrams, etc.) that lead to a specific product.	FoodCo's product costing system.
Methodology	General selection and sequence of techniques capable of producing a range of software products.	A range of object-oriented business applications.

Methodology

A methodology in the domain of Information System must cover a number of aspects of the project, which varies from one to another. A methodology can be described as a collection of many components . Typically, each methodology has procedures, techniques, tools and documentation aids that are intended to help the system developer in his or her efforts to develop an information system. There is usually also some kind of underlying philosophy that captures a particular view of the meaning and purpose of information systems development.

Checkland , gave a more general definition that captures well the notion of methodology as a guide to method. In his view, a methodology is a set of principles that in any particular situation has to be reduced to a method uniquely suited to that particular situation.

To give some examples of these aspects:

- The UML class diagram is a technique, and so is operation specification.
- Rational Rose is a tool.
- The activity represented by 'find classes by inspecting the use case descriptions' is an aspect of process. So is the advice that an analyst is usually the best person to write test plans.
- The advice that 'operation specifications should not be written until the class model is stable' is an aspect of structure, as it identifies a constraint on the sequence in which two steps should be performed.
- Analysis and design can be viewed as distinct stages.

- The statement 'object-oriented development promotes the construction of software which is robust and resilient to change' is an element of a systems development philosophy.

A package that contains enough information about each of these aspects of the overall development process can also be named a methodology. Many attempts have been made to capture the essence of methodology for software development, and the resulting methodologies are almost as varied as are the projects themselves.

Logical views of a system

At a very abstract level , three complementary views of a real-world system must be understood in order to model it adequately for the purpose of conducting software development. These are:

- **Data view**, that describes the real-world system in terms of attributes and associations that must be stored within the software;

- **Process view**, that describes the operations that are(or need to be) carried out on that data;

- **Temporal view**, that captures the time sequence and time constraints on individual processes, and also the possible sequences of events that may impinge on the system.

2. Why Use a Methodology

Over many decades, IS methodologies have been developed and introduced specifically to overcome those problems of software development projects that were perceived to be important at the time. However, to date, no methodology has been completely successful in fulfilling its objectives, partly because computing is a highly dynamic field, and the nature of both projects and their problems is constantly changing. In a changing world, it is unlikely that yesterday's solution will ever completely solve today's problems.

Advantages of Methodologies :

- The use of a methodology helps to produce a better quality product, in terms of documentation standards, acceptability to the user, maintainability and consistency of software.
- A methodology can help to ensure that user requirements are met completely.
- Use of a methodology helps the project manager, by giving better control of project execution and a reduction in overall development costs.
- Methodologies promote communication between project participants, by defining essential participants and interactions, and by giving a structure to the whole process .
- Through the standardization of process and documentation, a methodology can even encourage the transmission of know-how throughout an organization.

3. Different Types Of Software Development Methodologies

3.1 Structured Methodologies :

Early structured methodologies, such as those authored by DeMarco ,Gane and Sarson , were introduced to overcome some of the problems encountered with the Traditional Life Cycle. Their objective was to produce a 'structured' specification of the proposed software, with all functions, data storage, and interfaces between sub-systems clearly defined. To achieve this, it is necessary to define the techniques to be used and the deliverables at the end of each stage, as well as the stages of the life cycle. This all provides a more solid basis for project management, since the development process is more visible, and progress can be measured in terms of deliverables.

A project manager using a structured methodology might be able to apply tests like this: ‘ If the data flow model and entity-relationship diagram have been approved, then analysis phase is completed ‘ .

However, structured methodologies did not overcome all of the difficulties in systems development. For example, a manager still sometimes had no sound way of knowing whether a team was being optimistic in its progress reports or not.

And most structured methodologies focus either on functional or data aspects of the system being modelled. For example, Yourdon's structured methodology is primarily process-oriented, while Finkelstein structured methodology , is primarily data-oriented. Whichever view holds away, there is a danger of neglecting important aspects of the other. In this way it is difficult in co-ordinating the models of the two views.

Some structured methodologies attempted to cater for all views in a balanced way. One leading example was SSADM (Structured Systems Analysis and Design M*ethod*). SSADM has separate but carefully correlated models of processes, data and temporal sequence. While SSADM has undoubtedly been very successful, any methodology that attempts this degree of co-ordination inevitably tends to grow large and unwieldy, partly due to the need for continual checking and cross-referencing between the separate models.

SSADM continues to evolve to this day, and a number of changes have been made in the most recent version, SSADM4+ , to make the methodology more compatible with an object-oriented development approach. However, the fundamental models created in SSADM4+ are still organized around a logical separation of process and data. At heart, the methodology still appears structured, rather than object-oriented.

3. 2 Object-oriented development Methodologies

The main difficulty of structured methodologies in general is that most remain tied to a waterfall life cycle. This runs directly counter to the natural, iterative style of development for object-oriented software and makes it very difficult for a structured analysis and design approach to lead to an object-oriented implementation. In object-oriented development, necessary methodology is iterative cycle of development which is desirable.

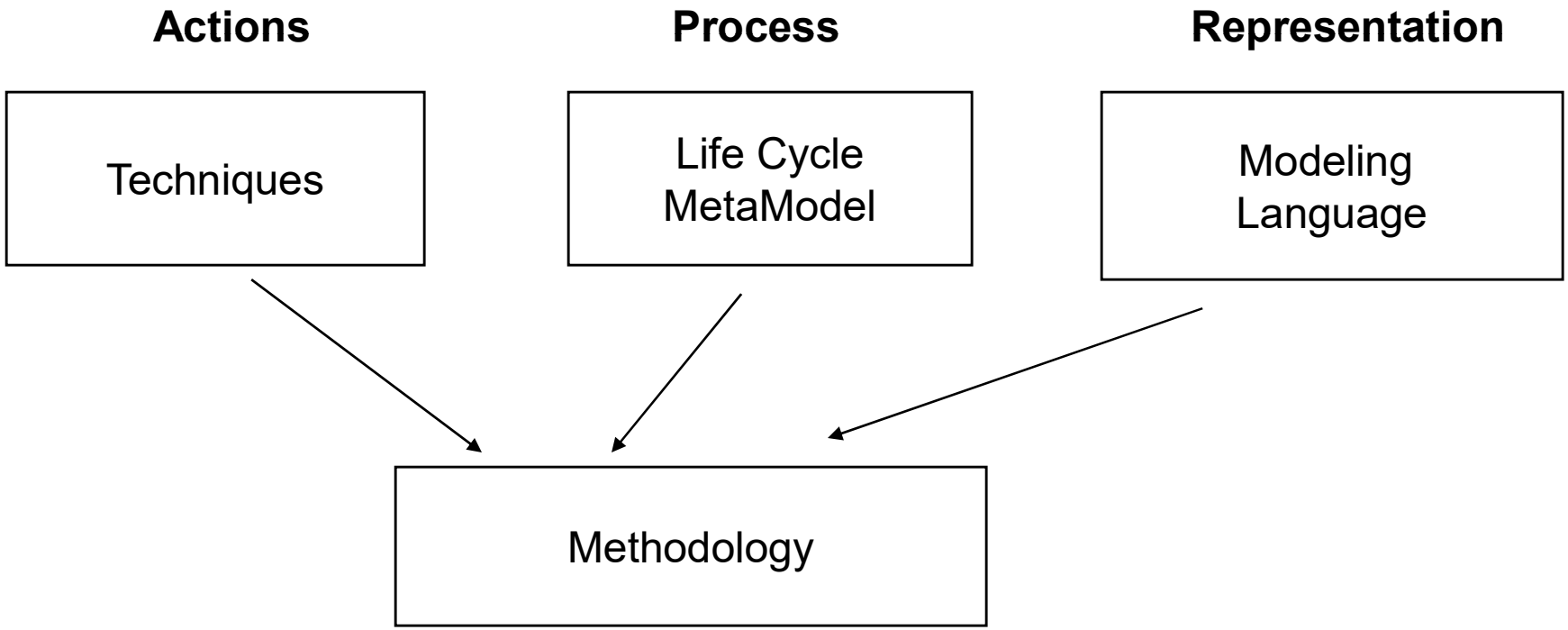
Different types of OO Methodologies are :

1. OPEN methodology (Object–Oriented Process , Environment and Notation).

2. USDP Methodology which combines features of Objectory, Object Modelling Technique (OMT) and the Booch method , which were earlier three leading object-oriented methodologies .

1. OPEN Methodology :

OPEN is a methodology which contains different types of elements as shown below.



As a package, OPEN consists of a number of components as explained below.

Activities. An activity is a collection of tasks seen from a project manager's perspective. Activities are similar to stages and phases in other methodologies. Activities have both pre- and post-conditions, and are carried out within timeboxes. Some examples of activities are: *project initiation, requirements engineering, analysis and model refinement, project planning* and *build.*

Tasks. Within each activity, one or more tasks are carried out. These primarily represent the developer's view of the project, although management tasks are also included in a very comprehensive list ,Some examples of tasks are: (draw) *rich pictures* , *analyze user require-ments, choose project team, design UI* and *perform class testing.*

Techniques. Each technique describes how to carry out one or more tasks. The list of techniques is also comprehensive, and includes many that are not original to OPEN. For example, the *rich pictures* task involves use of the *rich picture* technique, and the CRC technique can also be used . Other examples of techniques are: *class internal design* and *object life cycle histories.*

Deliverables. Deliverables are the post-conditions for activities, and often also the pre-conditions for other activities. Some examples of deliverables are: *user requirements statement* , *requirements specification ,*and many diagrams such as *inheritance diagrams* and *deployment diagrams.*

Notation. For object modelling, OPEN is not tied to any particular notation. UML may be used but the original specification of the methodology used its own notation, called COMN (Common Object Modelling Notation).

2. USDP Methodology (Unified Software Development Process)

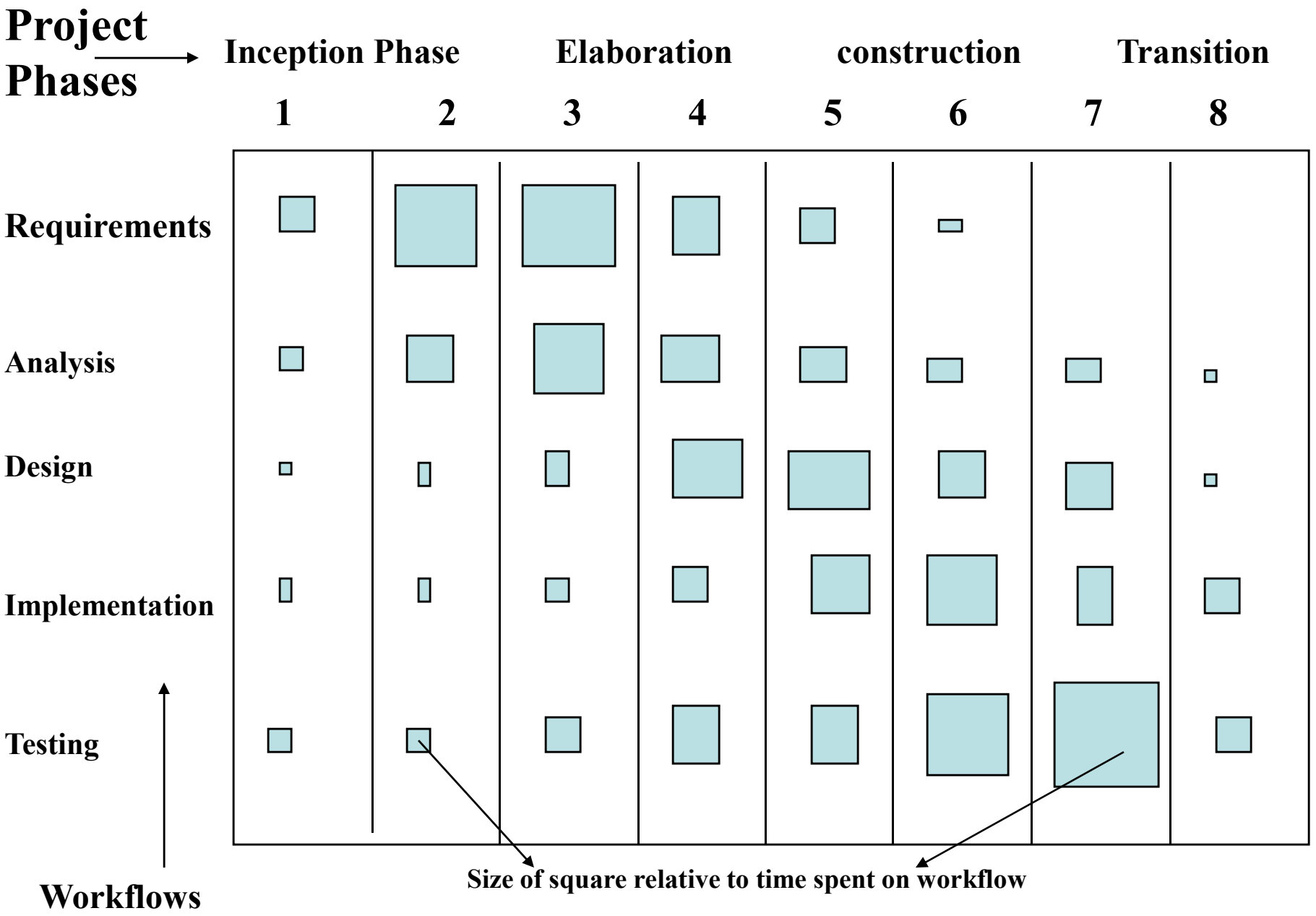
In the USDP, the resulting software architecture is an essential theme in modelling from the earliest stages of a project. This is reflected in the stereotyping of the classes that contribute to realizing a use case as boundary, control and entity classes.

The following **figure** shows *Phases and workflows* involved in USDP. This figure illustrates the relationship between the phases, iterations and workflows of the USDP. Here workflows are requirements, analysis, design , Implementation and Testing.

While an activity is something that has particular meaning for the developers who carry it out, a *phase* is considered primarily from the perspective of the project manager. He or she must necessarily think in terms of milestones that mark the progress of the project along its way to completion.

Phases are sequential. A project passes through each phase in turn and then (usually) moves on to the next. The end of a phase is a decision point for the project management. When each phase is complete, those in charge must decide whether to begin the next phase or to halt development at that point. The focus of the manager's attention shifts as the project progresses from one phase to the next.

Phases and Workflows in USDP



Within each phase, the workflows are essentially the same. All four phases include the full range of workflows from requirements to testing, but the emphasis that is given to each workflow changes between the phases. In the earlier phases, the emphasis lies more on the capture, modelling and analysis of requirements, while in the later phases the emphasis moves towards implementation and testing.

Inception phase : During the inception phase, the essential decision is that of assessing the potential risks of the project in comparison with its potential benefits. This judgement of project viability during the inception phase resembles the feasibility stage of a waterfall life cycle. The decision will probably be based partly on a similar financial assessment .

One principal difference at this early stage is that the viability of a USDP project is much more likely to be judged partly also on the delivery of a small subset of the requirements as working software. During the inception phase, the main activities are thus requirements capture and analysis, followed by a small amount of design, implementation and testing.

Another major difference is that, even at this early stage, there is the likelihood of iteration. That this is even possible, is due to the fact that the development approach is object-oriented.

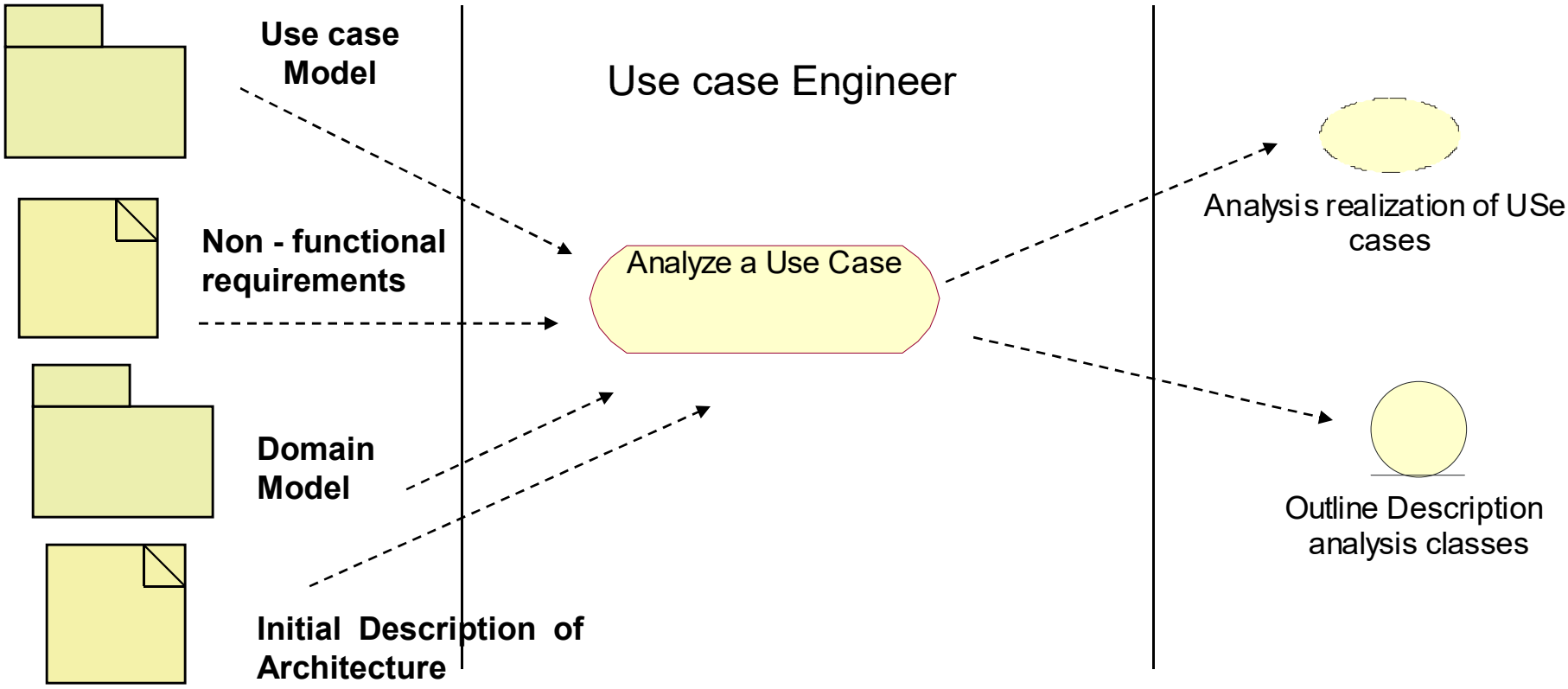
Elaboration phase : During the elaboration phase, attention shifts to the reduction of cost uncertainties. This is done principally by producing a design for a suitable system that demonstrates how it can be built within an acceptable timescale and budget. As the emphasis shifts towards design, the proportion of time spent on design activities increases signifi-cantly. There is a further small increase in the time spent on implementation and testing, but this is still small in relation to the analysis and design activity.

Construction phase : The construction phase concentrates on building, through a series of iterations, a system that is capable of satisfactory operation within its target environment. Implementation and testing rapidly become core activities in this phase, with a move further away from design and towards testing as each iteration gives way to the next.

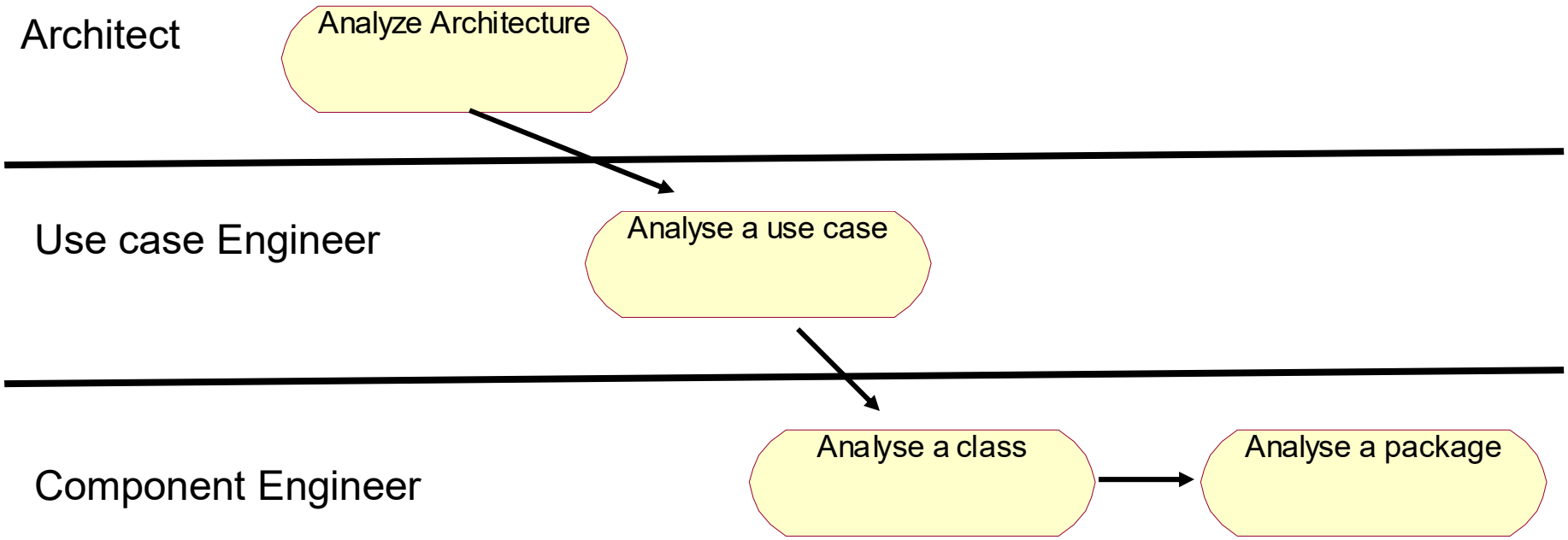
Transition phase : The transition phase concentrates on achieving the intended full capability of the system. This deals with any defects or problems that have emerged late in the project. It could also include system conversion, if an older system is being replaced

Workers and activities. The USDP differentiates between the real people who are involved with any project, such as users, analysts, managers and customers, and the more abstract *worker* means , someone who plays a specified part in carrying out an activity. Some examples of workers are: use-case specifier, system architect, component engineer and integration tester.

Most USDP activities can be partially defined in terms of the workers who carry them out, and the artefacts that either serve as inputs or are produced as outputs. These things are shown in the following figure, for the activity ***Analyze a use case.***



And a workflow can be seen as a flow of activities. Since each activity can be related to a worker who will carry it out, we can identify which workers will need to participate in the project. The following figure shows the Analysis workflow broken down into its constituent activities.



4. Participative Design Approaches

Participatory Design (PD) is the name given to a collection of approaches to information systems development that share a guiding mechanisms more than they share any particular tools or techniques.

Sometimes known as co-operative design, PD should perhaps be seen as a movement rather than as a methodology. The common guide is based on the assumption that active involvement of users in the design and development activity is critical to the success of an information system. This is because a successful design for an information system is said to rely just as much on knowledge and understanding of the work that is to be supported as it does on knowledge of the possibilities and limitations of the available technology.

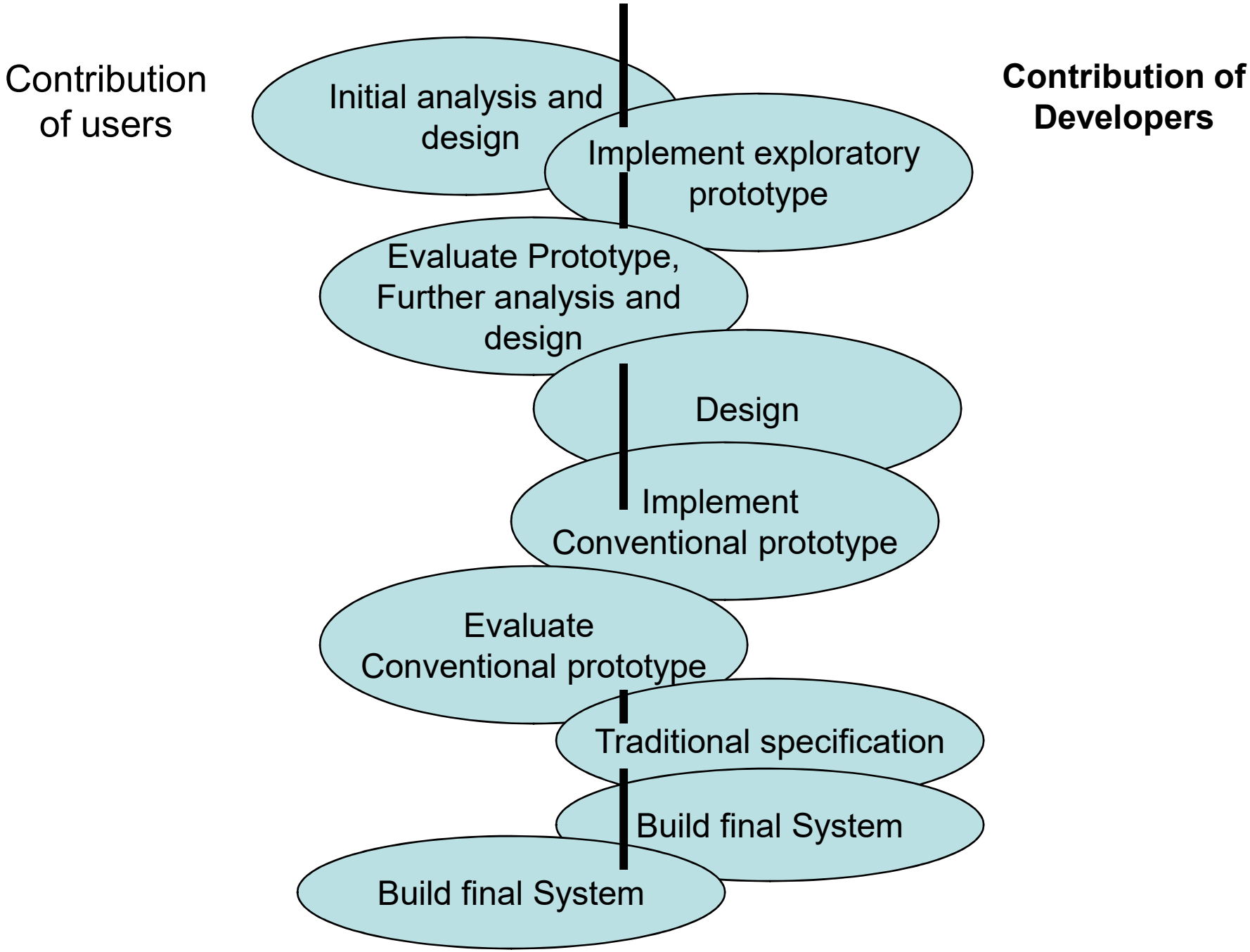
- Some approaches to PD, particularly that of Kyng and his collaborators emphasize the use of *use scenarios.* In many respects these resemble the use cases of Objectory and USDP, although there are differences in granularity
- PD approaches usually emphasize prototypes and storyboards, often low-technology mock-ups drawn on paper or cardboard, for requirements capture.
- The typical PD life cycle is experimental and iterative in nature, in recognition of the fact that design is always to some extent a learning experience for all participants.

- The active participation of users in design and development is fundamental to all PD approaches, usually throughout the project life cycle. An iterative approach to object-oriented development naturally encourages the involvement of users at many stages of the life cycle. The DSDM as a project management methodology that is thoroughly compatible with object-oriented development, and DSDM, like PD, insists on the active involvement of users in the project team.

Many styles of participative design have been proposed and practised, Here we are giving one type of Co-operative Design approach developed by Morten Kyng at..al

The following diagram shows lifecycle for Co-operative Design .

The diagram illustrates a possible sequence of activities within a project and also shows whether the degree of responsibility for each activity that lies with users or with developers. Most activities are shared to some extent between the two, but we can see that some rely chiefly on the contributions of developers (for example, building the prototypes) while others rely primarily on the contribution of users (for example, evaluating the prototypes).


Contribution of users
Initial analysis and design
Implement exploratory prototype
Contribution of Developers
Evaluate Prototype, Further analysis and design
Design
Implement Conventional prototype
Evaluate Conventional prototype
Traditional specification
Build final System
Build final System

The Figure lifecycle for Co-operative Design looks like a waterfall life cycle model, a more careful reading reveals that in fact the underlying life cycle is iterative. Two cycles of prototyping are shown followed by a final system build, but this can just as easily be interpreted as three cycles of iterative development.

Part of the background to PD is the belief among its proponents that the models built during the analysis and design of a proposed system fall into two general categories.

The first category includes representations of the work that is done by people who will become users of the system. In the Agate case study, the use case Assign staff to work on a campaign falls into this category.

The second category includes representations of the system that is being designed. In the Agate case study, the sequence diagram for the same use case falls into this category.

Only a user can really tell whether a given design fully takes into account all requirements together with the constraints and limitations that are imposed by work practices, working environment, technology and so on. Thus the activities of system design also require active user participation.

In practice, since it is important for both users and developers that their mutual knowledge and understanding should grow as the project progresses, it makes sense for both groups to share responsibility for the whole project from inception to completion.

5. Hard Vs Soft Methodologies

Distinction between hard and soft methodologies which arised principally from the broad systems movement, and summarized in the following figure .

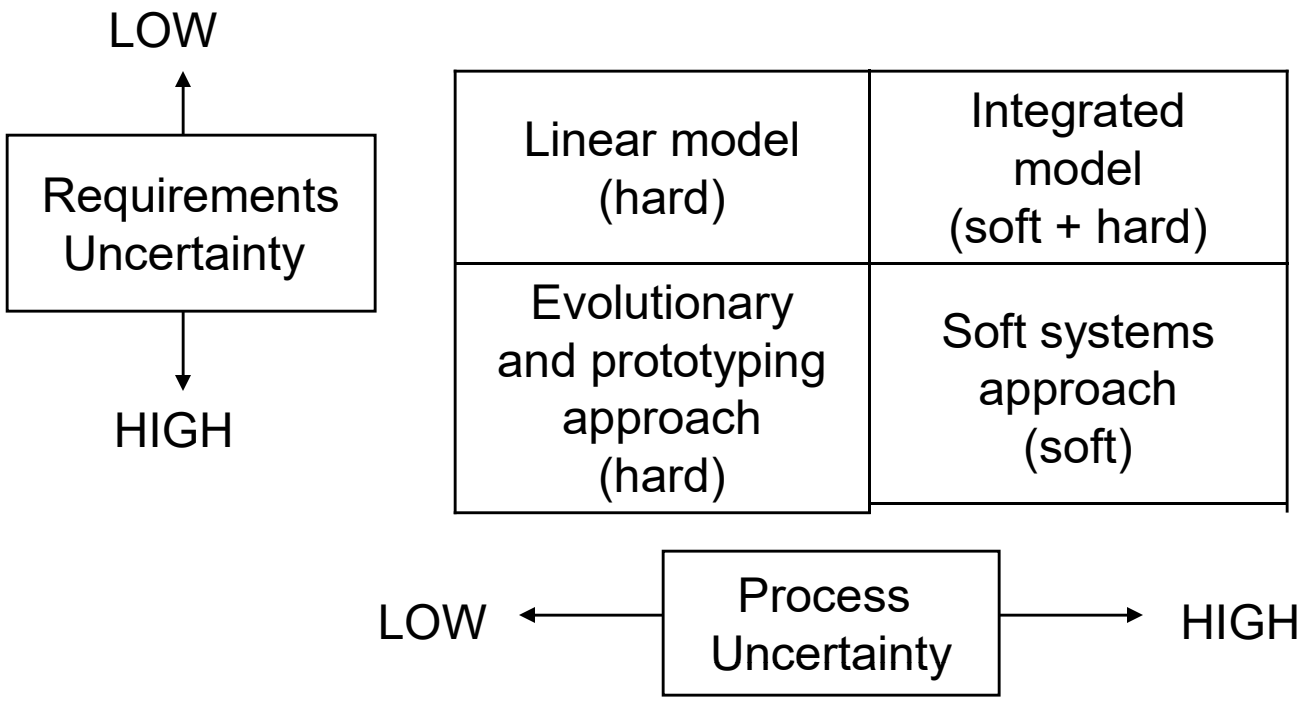
Hard systems view	Soft systems view
The activity of IS development is all about building a technical system that is made only of software and hardware.	An IS also comprises the social context in which the technical system (software and hardware) will be used.
Human factors are chiefly important from the perspective of the software's usability and acceptability. Politics and group behaviour are only an issue for project managers.	A new IS impacts on interpersonal communication, social organization, working practices and much more, so human and social factors are paramount.
Organizations exist only to meet rational objectives, through the application of rational principles of business management. It is possible to be both rational and objective about the requirements for a new system.	Organizations are made up of individuals with distinct views and motivations, so any picture of requirements is subjective. It is not always possible even to reach a consensus. In practice, this means that the powerful decide, not the wise.
When requirements are uncertain or unclear, it is up to management to decide. Setting objectives is a principal role of management, and others should follow their lead.	If management has not accommodated the full range of views in the organization, encouraging managers to decide on the requirements may be completely counter-productive.

Here, 'hard' is usually taken to mean objective, quantifiable or based on rational scientific and engineering principles , where as 'soft' involves people issues and is ambiguous and subjective. Both the structured and OO Methodologies all derive mainly from the hard tradition, although some influence of a soft approach can be discerned in Participative Design and also in the use case technique, since this aims at eliciting the practical, context-based requirements of individual users.

On the whole, those methodologies that might be characterized as principally soft in their orientation tend to focus more on making sure that the 'right' system is developed, than on how to actually develop the system. Their intellectual antecedents are diverse. However, in spite of their very different origins, both provide ways of exploring and agreeing the characteristics of the organization as a system, before any attempt is made to define a specific information system that will help users meet their goals.

In certain situations, hard and soft methodologies can complement each other, and can be used together to help overcome some of the persistant difficulties in systems development. Flynn proposed a 'contingency framework' which is shown in the following figure, aims at helping to select an appropriate methodology for a specific organizational context.

For example, a new system intended primarily to automate an existing manual system may have relatively low requirements uncertainty. 'Process uncertainty' refers to the degree of doubt about the best way to build the proposed system. A project intended to introduce Electronic Commerce to an organization with no experience of it might fall in this category.



'Requirements uncertainty' is the extent to which requirements are unknown or subject to debate or disagreement, and also whether they are expected to change during development. For example, a new system intended primarily to automate an existing manual system may have relatively low requirements uncertainty.

The term 'Linear Model' refers to a sequential life cycle model like the waterfall model. A project is rated along both dimensions, and this helps to indicate an appropriate development approach. For example, where both the requirements and process are clear from the outset, a linear model of development is recommended., which in practice might either mean using a traditional structured methodology, or procuring a ready-made solution. At the other extreme, a soft approach is recommended, so that the character of the problem is clarified before any further action is taken.