UML Process

Sharam Hekmat
PragSoft Corporation
www.pragsoft.com

Contents

1. Introduction

1.1 Purpose

UMLProcess is a defined process for developing software systems using object technology. The purpose of this document is to define the UMLProcess at a level that is suitable for practitioners who have had no prior exposure to a similar process.

1.2 Scope

This document is intended to be a *concise* guide to the processes it covers, rather than giving a detailed description of each process. By focusing on the key concepts (and deferring the practical details to workshops and mentoring sessions), we can maximise the usefulness of the handbook as a learning tool.

1.3 Software Tool

If you plan to implement the UMLProcess in your organisation, we recommend that you use a UML modelling tool to formalise your modelling activities. PragSoft provides two very popular tools for this purpose:

- UMLStudio allows you to create UML models, generate code from them, and reverse engineering UML models from code.
- UMLServer allows you to deploy UMLStudio in a collaborative environment.

Both tools can be downloaded from *www.pragsoft.com*.

1.4 Glossary of Terms

BPR	Business Process Re-engineering
CBD	Component Based Development
DSDM	Dynamic Software Development Method
GUI	Graphical User Interface
POC	Proof Of Concept
RAD	Rapid Application Development
SC	Small Change
UML	Unified Modelling Language

2. Reference Models

The processes that underpin the development of modern information systems are varied and complex. This section provides a number of reference models to help manage this complexity, covering three broad areas of:

- Process
- Lifecycle
- Architecture

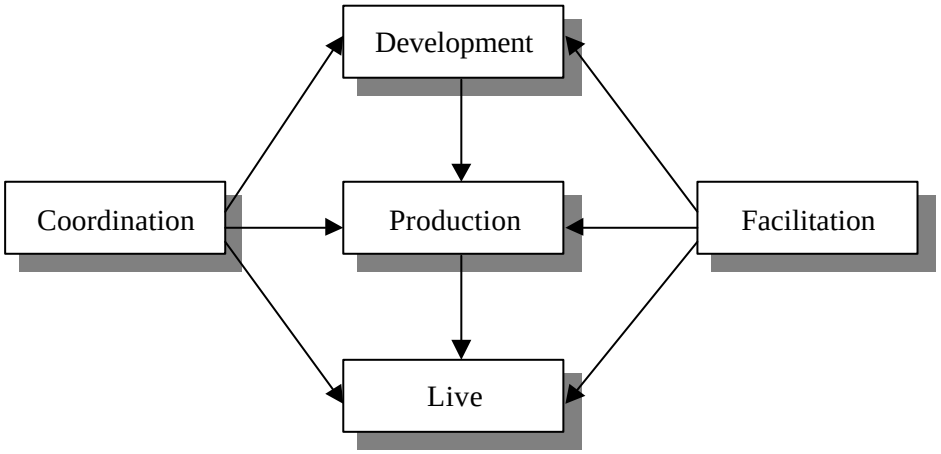
The reference models provide a common understanding, so that when we later talk, for example, of production, business objects, or Gate 2, the intent is clear.

2.1 Process Reference Models

A process is a well-defined collection of activities, each undertaken by possibly a different participant, which takes one or more inputs and produces one or more outputs. Every manufacturing or service industry uses a set of inter-related processes for its operation. The quality of the *design* of these processes and the quality of their *implementation* determines the overall quality of the organisation. In other words, to improve an organisation, one needs to improve its underlying processes.

2.1.1 Process Domains Reference Model

At the highest level, the processes that underpin the development and operation of information solutions can be divided into 5 domains, as illustrated below.



The **development** domain is concerned with processes that directly contribute to the development of the solution. These include:

- Business modelling
- Application modelling
- Architectural design

- Detailed design
- Coding and testing
- System testing
- Problem reporting and fixing

The **production** domain is concerned with processes that directly affect the evolution of the system after it is fully developed. These include:

- Detailed design
- Coding and testing
- System testing
- Acceptance testing
- Problem reporting and fixing

The **live** domain is concerned with processes that directly affect the operation of the solution in a live environment. These include:

- Release management
- Performance monitoring
- Help desk
- Problem reporting and fixing

The **coordination** domain is concerned with processes that regulate and manage the successive progression of the solution through its various stages. These include:

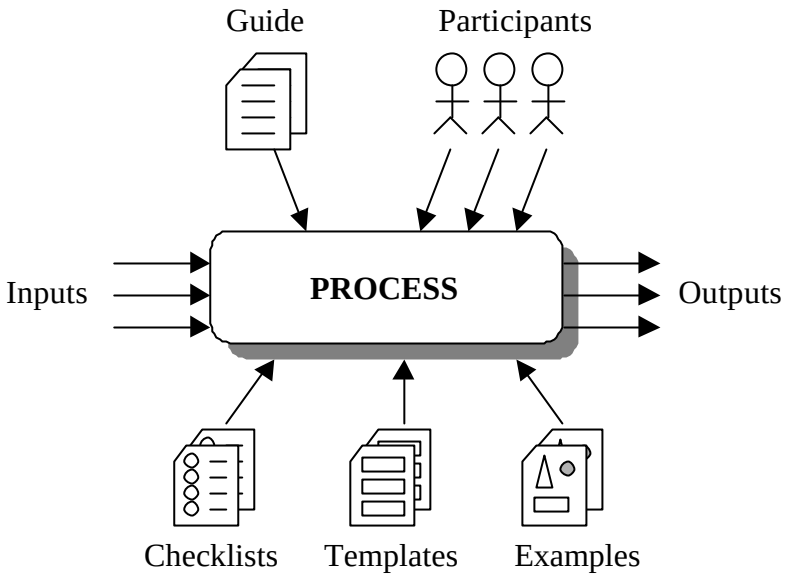
- Project management
- Quality management
- Change management

The **facilitation** domain is concerned with processes that indirectly contribute to development, production, and live processes by way of providing guidance and/or administrative assistance. These include:

- Configuration management
- Training and mentoring
- Quality reviews
- Metrics collection and reporting

2.1.2 Process Reference Model

Each process is described by a set of process **elements**, as illustrated below.

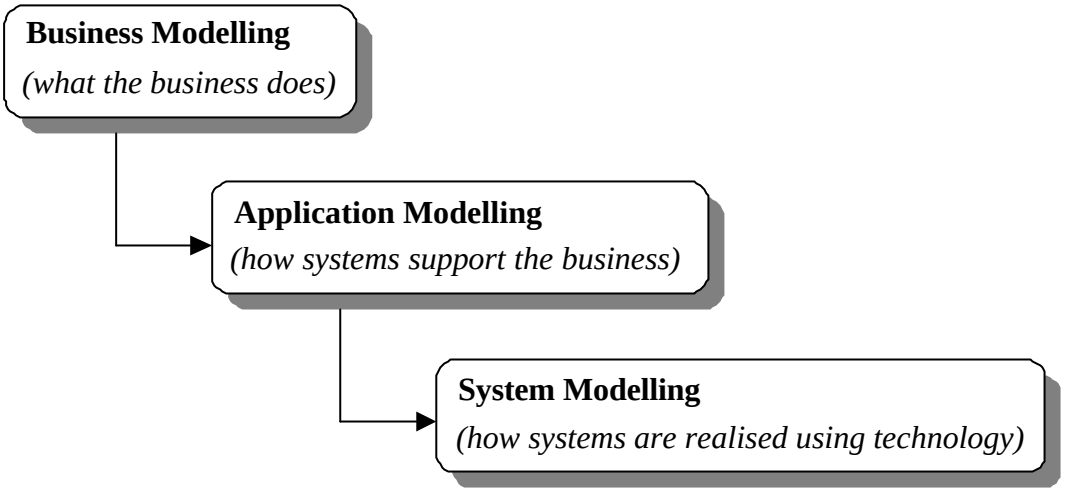


The **guide** describes the process, its **inputs**, constituent parts, **outputs**, and how each **participant** contributes to it. The **checklists** provide a means of verifying that the process parts have been completed to satisfaction and meet the necessary criteria. The **templates** provide a standard format and structure for the deliverables (outputs) produced by the process. The **examples** serve as a learning aid and illustrate to the process participants sample deliverables produced by the real-life application of the process.

2.1.3 Modelling Reference Model

A different way of looking at development processes is to view them as an iteration of modelling exercises. Each modelling exercise takes one or more earlier models and produces a new, more enriched model by making additional design decisions. This results in a progression from abstract (requirements) to detailed (working solution). This approach, combined with object-oriented modelling, has the distinct advantage of producing representations that can be verified through logical reasoning, testing, or even simulation. For example, a business process map can be tested by mentally passing imaginary cases through it that exercise its different logical paths to see if it copes with the possibilities and produces the required output.

There are three broad types of modelling, as illustrated below.



Business modelling is concerned with *what the business does*. This is before using information systems to automate aspects of the business. This may appear as redundant if the business already has systems in place. But this is exactly the point. Technologists often forget that information systems are not an end to themselves, but a means for serving the business (i.e., to support the business processes they are aimed at). If it is not clear what the business does, then it will be equally unclear how systems may be able to support it.

The business model is described in purely business terms. One of its key objectives is to establish a common, unambiguous understanding between the business users and the technologists who will ultimately build appropriate system solutions for it. The importance of this baseline cannot be overstated. Its quality and completeness will, more than any other model, influence the success of the final solution.

Business modelling produces the following artefacts:

- End-to-end Business Processes
- Business Process Maps
- Activity Maps
- Action Narratives
- Use-cases

Application modelling is concerned with *how systems support the business*. Having established a business model that describes what the business does, we are then in a position to come up with an application solution that addresses the business needs. This is essentially an **external** view of the solution and shows how the users interact with the application, its look and feel, and the business abstractions (objects) that are represented by the application. Application modelling is where functional requirements are addressed.

The application model does not assume any specific implementation technology and is primarily described in non-technological terms. It should, therefore, be reasonably understandable by the business users.

Application modelling produces the following artefacts:

- Business Objects (class diagrams)
- Scenarios (collaboration/sequence diagrams)
- User Interface Models:
  - Metaphors
  - Mock-ups

System modelling is concerned with *how systems are realised using technology*. System modelling is largely a technological activity that attempts to translate the application model into a concrete, executable system. System modelling has to deal with artificial details that are not an inherent part of the application model, but a by-product of using specific technologies. For example, it has to deal with specific programming constructs, middleware services, data models, and so on. In other words, it produces an **internal** view of the solution, showing how its different parts interact in order to support the external, application view. System modelling is where the non-functional requirements (e.g., platform, performance, throughput, scalability, maintainability) are addressed.

The system model is expressed in technical terms and is for the internal use of the technologists who work on it. It is inappropriate reading material for business users.

System modelling produces the following artefacts:

- User Interface Models:
  - Screen Specifications
    - Data
    - Data Entry Validation Rules
  - Navigation
- Front-end Components
- Application Server Components
- Business Object Server Components
- Data Access Components
- Data Models

It should be emphasised that design decisions are made in *all* three types of modelling. In business modelling we do not simply record the way the business operates now ('as-is' processes), we also consider how it could operate with the potential benefit of introducing information systems that can streamline the business activities ('to-be' processes). In application modelling, we invent metaphors, screens, and abstractions that enable end-users to use the application as an effective and intuitive tool that blends with their work processes, rather than becoming an obstacle to their work. In system modelling, we invent software artefacts that collectively not only realise the functional requirements for the application, but also satisfy its non-functional requirements.

2.2 Lifecycle Reference Models

A software lifecycle is a map, depicting the stages that a software system undergoes, from its original inception, to its final termination (i.e., from cradle to grave). There is, however, no universally agreed

lifecycle that is suited to all software development projects. Various lifecycles have been invented for different purposes. Project characteristics (such as size, timeframe, volatility of requirements, architecture, technologies, expected system lifetime) influence the most appropriate choice of lifecycle for a project.

This section outlines different software lifecycles and their intended usage context. The relationship between the different lifecycles is managed through a gateway reference model. This is described first.

2.2.1 Gateways Reference Model

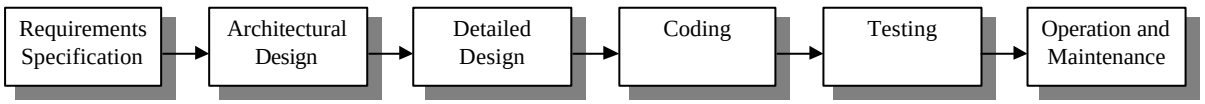
Every software project involves a number of key milestones. Each milestone represents an important event, and provides a review opportunity to decide whether the project should proceed to the next step. These milestones are called gates. Five universal gates are defined:

Gate	Description
0	A business case exists for the proposed solution.
1	Business requirements have been specified and agreed upon.
2	An integration-tested release has been produced, ready for system testing.
3	The release has successfully passed system testing.
4	The release has successfully passed acceptance testing.

Not all gates are relevant to all lifecycles. Also, in some lifecycles a gate may be passed iteratively (i.e., one chunk at a time).

2.2.2 Linear Lifecycle Model

The linear lifecycle model (also called the waterfall lifecycle) views software development as a set of phases that take place linearly (i.e., one after the other).



This lifecycle is largely outdated because of the common problems it suffers from:

- It assumes that it is feasible to produce an accurate specification of requirements before getting involved in the design and implementation of the system. Experience has shown that, in most cases, this is not practical. In practice, requirements often tend to be vague and incomplete. Users are not certain of what they want and, once they see a working system in operation, tend to change their mind.
- It takes too long to produce a demonstrable system, during which time the business climate and hence requirements may change substantially. So by the time the system is delivered, it may be already obsolete.

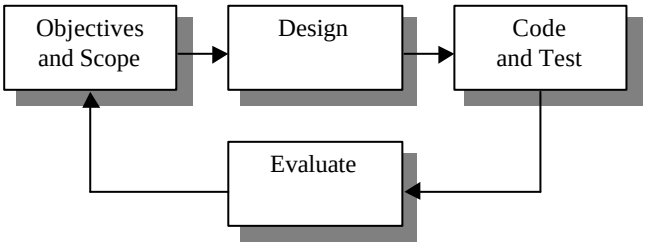
- The impact of defects in earlier phases is far too great on later phases. For example, a requirements defect discovered during the coding phase may cost 100-1000 times a coding defect to correct.
- The rate of rework tends to increase substantially from phase to phase, because each phase tends to uncover defects in the deliverables of earlier phases. This in turn derails the project plan and often puts unrealistic pressure on the development team, which ultimately may result in their demotivation and break up.
- The track record of this lifecycle in the industry is very poor, with over 60% of projects never delivering, and of those delivered, over 50% not being used by the end-users.

Despite these shortcomings, the linear lifecycle does have a place in the software industry. Situations where its application may be sensible include:

- Where the requirements are stable, well understood, and well documented. An example of this is the re-engineering of an existing system that is largely consistent with user requirements, but perhaps technologically obsolete.
- Where the system does not directly interact with end-users (e.g., firmware and communication software).
- Where there is a wealth of existing experience about the problem domain and the project team has had considerable experience of developing similar systems. For example, a company with a good track record of successfully developing payroll applications, is unlikely to hit nasty surprises when developing yet another payroll application.

2.2.3 Proof-of-Concept Lifecycle Model

The Proof of Concept (POC) lifecycle is suited to situations where a proposed concept (e.g., a business process, an architecture) needs to be proven before making further development investment. This is largely a risk management tool: by verifying the suitability and effectiveness of an idea on a small scale, we minimise the potential loss resulting from its failure. In practice, most ideas are partially successful, and the POC provides an opportunity to address their shortcomings before implementing them on a large scale.



The POC lifecycle is simple, but iterative. It begins by establishing the objectives and the scope for the POC. The objectives must be clearly stated and, based on these, a modest scope should be established. Here is an example:

Objectives:

- To prove the technical feasibility of the proposed 3-tier client-server architecture for the proposed retail banking system.
- To verify that the implementation of this architecture can deliver the required performance (< 5 seconds latency, for 1000 concurrent users).

Scope:

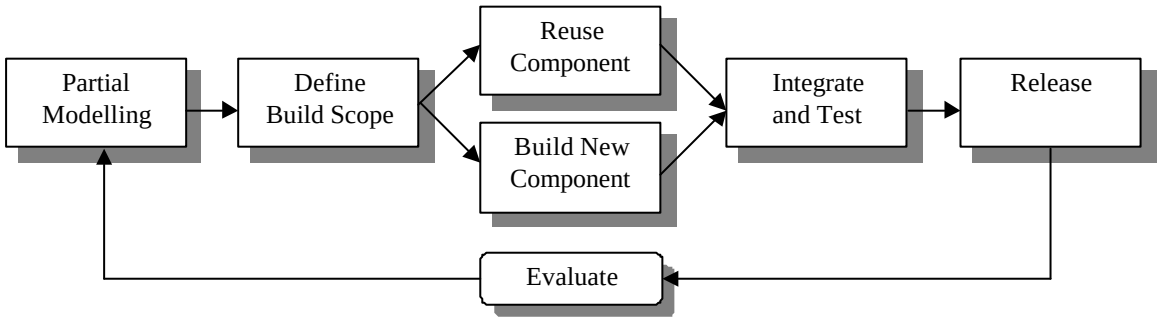
- Implement the one-to-one transfer transaction, end to end. Use ad-hoc shortcuts to populate the database and to simulate 1000 concurrent users.

The scope must be manageable, a good representative of the problem domain, and sufficiently rich to enable the verification of the objectives.

During design and coding in POC, emphasis is typically on speed of construction. Trade-off and corner cutting are acceptable practices, provided they do not conflict with the objectives. For example, in the above scenario, it would be perfectly acceptable to implement only rudimentary error handling. However, if one of the objectives were to verify system robustness under erroneous input data, then this would be unacceptable.

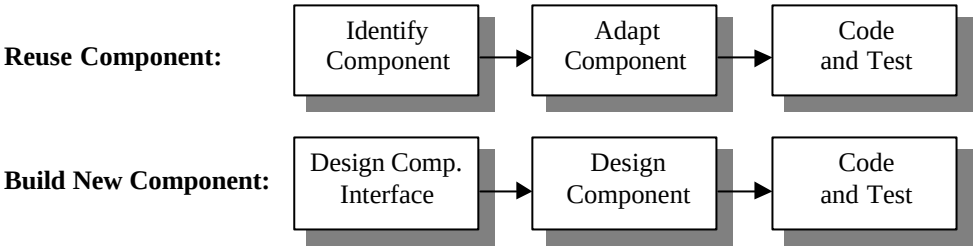
2.2.4 CBD Lifecycle Model

The Component Based Development (CBD) lifecycle is an emerging lifecycle for the development of distributed client-server systems using component technology.



The Partial Modelling phase involves carrying out enough business/application/system modelling to define a meaningful build scope. A build delivers a well-defined set of business functionalities that end-users can use to do real work. In a process-centric information system, for example, a build may represent the realisation of one or more end-to-end business processes. The scope of a build is not a random selection, but rather a logical selection that satisfies specific development objectives.

Once a build scope is established, we need to decide which of the required components can be reused (e.g., already exist in the organisation or can be bought off-the-shelf) and which ones need to be developed. Both these phases have their own mini lifecycles:



Reusing an existing component may require some adaptation. For example, the component interface might not be exactly what is required or some of the method behaviours may need alteration. This is achieved through adaptation, which involves wrapping the component with a thin layer of code that implements the required changes.

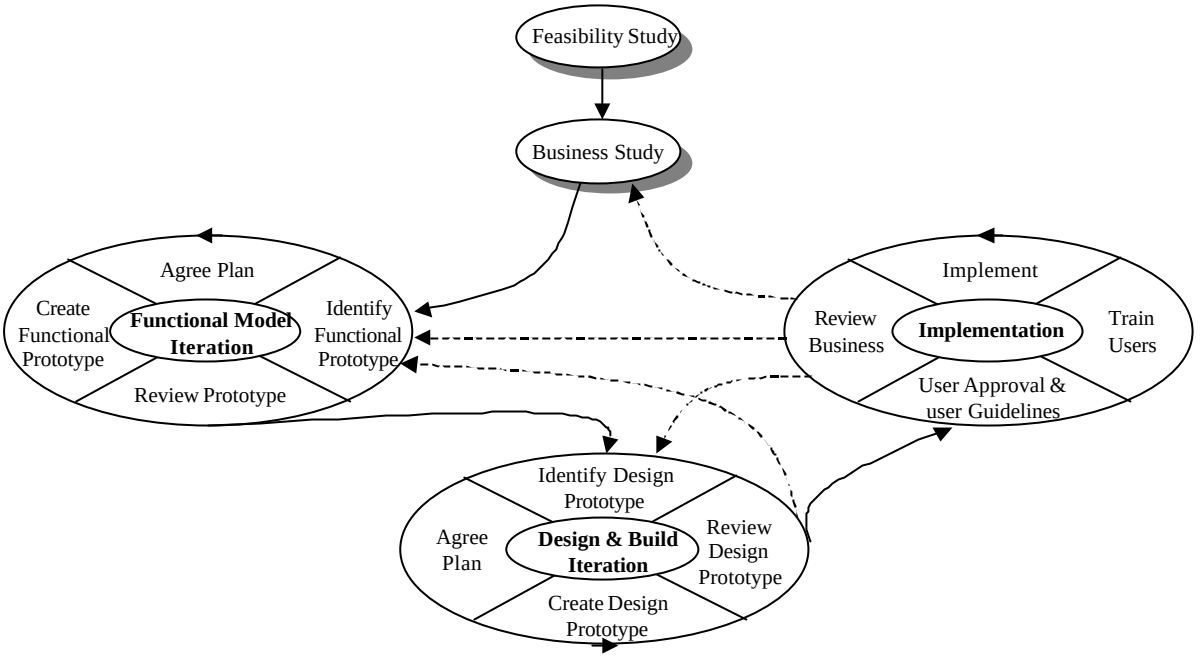
Building a new component should always begin with defining the component interface. This represents a permanent contract between the component and other components. Once the interface is defined and the intent of each method is established, the component can be designed and implemented.

With all the components for a build in place, the components are then integrated and tested. Integration will require the writing of glue code that establishes the interaction between the components. Most component technologies allow this to be done productively using a scripting language.

An integrated build is then formally released (by going through system testing). This is then made available to end-users for evaluation. The evaluation environment may be the same as the development environment (for earlier builds that are not mature), or a pseudo live environment (for later builds that are sufficiently mature). The outcome of the evaluation influences the direction of subsequent builds.

2.2.5 DSDM Lifecycle Model

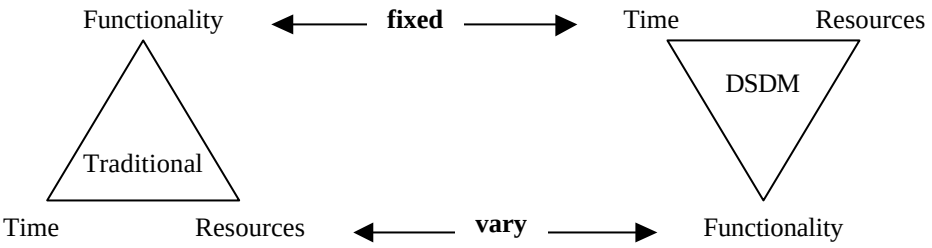
The Dynamic Software Development Method (DSDM) lifecycle is another emerging lifecycle that is suited to the development of information systems (that have vague and/or unstable requirements) to tight time-scales. Unlike CBD, DSDM is independent of any particular technology, but like CDB, it relies on an iterative approach to development.



DSDM is also often referred to as RAD (Rapid Application Development). It consists of five main phases:

- **Feasibility study**, which typically lasts a couple of weeks and assesses the suitability of the RAD approach to the business problem.
- **Business study** which scopes the overall activity and provides the baseline for subsequent work, including business functionality, system architecture, and development objectives.
- **Functional model iteration** which, through a series of prototyping iterations, establishes the application functionality. The prototypes are created for their functionality and not intended to be maintainable.
- **Design and build iteration**, which generates well-engineered prototypes for use in the intended environment.
- **Implementation**, which involves putting the latest increment into the operational environment and training the users.

The following diagram illustrates a key difference between DSDM and traditional methods. Whereas in the traditional methods, the functionality to be delivered is fixed and time and resources may vary to meet that functionality, in DSDM this is turned upside down: functionality is delivered to a fixed time and resource plan. This is based on a fundamental assumption that 80% of business functionality can be delivered in 20% of the time it takes to deliver the whole thing.

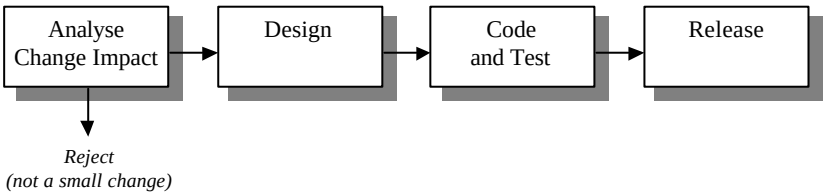


To achieve this, DSDM employs the **timebox** mechanism. For a given project, there is an overall timebox for the work to be done. This is hierarchically broken into shorter timeboxes of 2 to 6 weeks, which are the focus of monitoring and control activities. Each timebox has an immovable end date and a prioritised set of requirements assigned to it. Some of these requirements are mandatory and some of less priority. A timebox will produce something visible in order for progress to be assessed (e.g., a model or a component). Each timebox is inclusive of all its effort, and is divided into 3 parts:

- **Investigation**, which is a quick pass to check that the team is taking the right direction.
- **Refinement**, which builds on the comments resulting from the review at the end of the investigation.
- **Consolidation**, which ties up any loose ends.

2.2.6 Small Change Lifecycle Model

The Small Change (SC) lifecycle is a streamlined lifecycle suitable for making small changes to an existing system.



SC is especially suitable for implementing the type of changes undertaken in a minor release. For example:

- Addition of a new piece of functionality that is fairly similar, in its design, to existing functionality.
- Enhancing or enriching an existing piece of functionality without changing its underlying design.
- Making cosmetic changes to the front-end.
- Making changes to the format (but not the logical structure) of input or output data (e.g., changing a report format).

SC is not suitable for the following types of change:

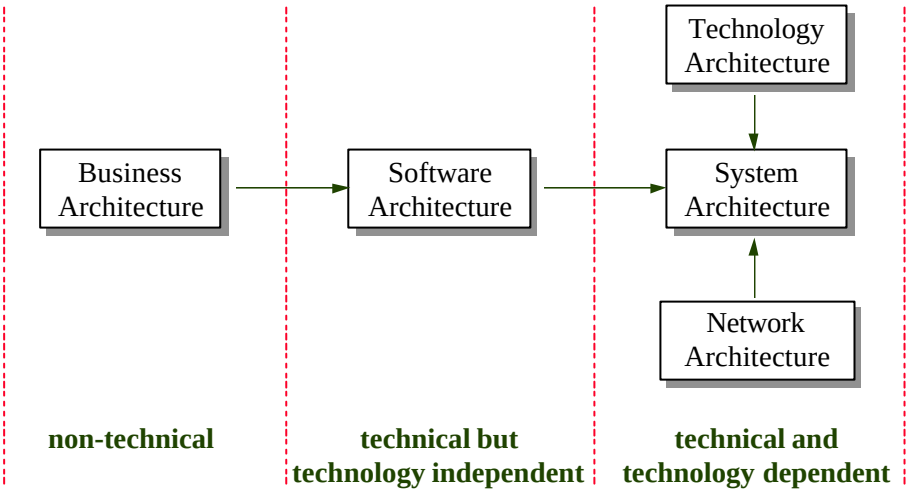
- Architectural changes (e.g., changing the tiering model).

- Changing a key piece of technology in the system (e.g., porting the system from one programming language or platform to another).
- Major addition or revision of functionality (i.e., changes undertaken in a major release).

2.3 Architectural Reference Models

2.3.1 Architectural Domains Reference Model

The following reference model illustrates the key architectural domains of an information system and their relationships.



The primary aim of this model is to achieve a clear separation between the technical and non-technical, and between the technology dependent and technology independent. This ensures that technical changes do not invalidate the business architecture, and that technological changes do not impact the business and software architectures.

A **business architecture** models the key elements of a business, their relationships, and interactions. Its main focus is **analysis**. Given that business processes provide the most reliable and relevant foundation for articulating the desired behaviour of an information system, a process-centric approach to business modelling would be ideal.

A **software architecture** (including data architecture and security architecture) models a software system in terms of its key layers, components, and interfaces between them. Its main focuses are **design** and **ease of maintenance**. A component-based approach is now the industry-wide standard for this purpose.

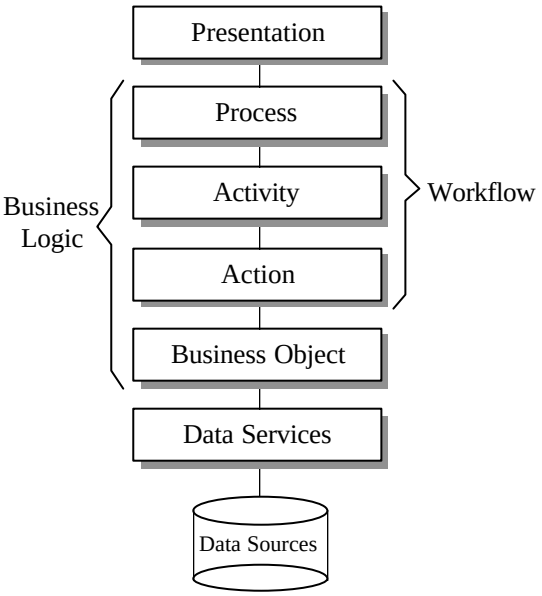
A **technology architecture** models the technological framework for building information systems (including: tools, languages, middleware, operating systems, standards, protocols, third-party components, etc.). Its main focus is **construction**. The technology architecture needs to continuously evolve to keep up with the relevant and mature offerings that gain acceptance in the industry.

A **network architecture** models the computing infrastructure that is used to deploy information systems. Its main focus is **deployment**.

A **system architecture** maps a software architecture to a network architecture using a given technology architecture. Its main focus is **operation**. One of the key objectives of a system architecture is to provide a tiering model that best fits the operational requirements of the system (e.g., scalability, physical distribution, and fault tolerance).

2.3.2 Layered Architecture Reference Model

The following reference model illustrates the key software architecture layers of a process-centric information system. This model is suitable for information systems used in the finance industry because the great majority of such systems support specific business processes.



The **presentation** layer is concerned with the displaying of data to and accepting input from the user. This layer contains no business functionality.

The **process** layer defines the end-to-end business processes of the organisation (e.g., 'open bank account'). A process typically involves a number of persons and is specified in terms of activities and queues (a queue is a place where an activity may deposit data for other activities to withdraw from later).

The **activity** layer defines the activities that comprise the processes. Each activity is a process segment that is performed by one person. For example, the 'open bank account' process may have an activity called 'setup account information'.

The **action** layer defines the specific actions that each activity is broken into. An **action** represents an atomic step: it is either performed fully or not performed at all. For example, the 'setup account information' activity may involve an 'enter customer details' action.

The **business object** layer specifies the business objects that ultimately realise the business functionalities behind the business processes. Each business object represents a key entity in the business domain. When an action is performed, it affects one or more business objects. For example, the 'enter customer details' action will affect the 'Customer' business object.

The **data services** layer handles the querying and updating of external data sources. The data sources provide persistent storage for the business objects (e.g., for 'Customer').

The process, activity, and action layers are collectively called the **workflow** layer. These layers can be supported using a workflow engine.

The process, activity, action, and business object layers are collectively called the **business logic** layer. All of the business logic of the application (e.g., validation, sequencing, and updating rules) are contained by these layers. Consequently, the implementation of a transaction may encompass all these layers.

3. Business Modelling

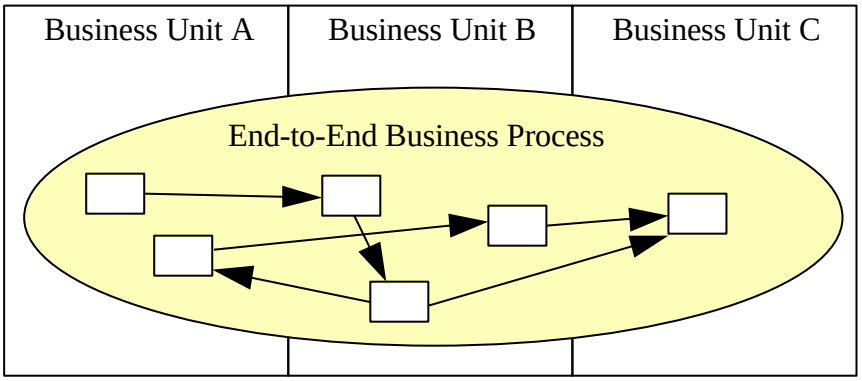
As stated earlier, business modelling is concerned with *what the business does*, and the business model is described in purely business terms.

3.1 Introduction to Business Processes

3.1.1 What Constitutes a Business Process?

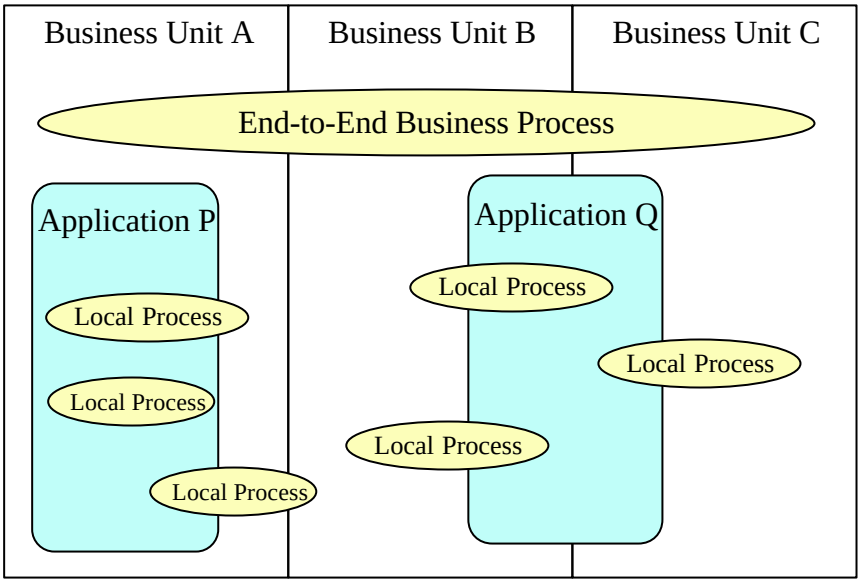
A business process is simply a set of well-defined and inter-related activities that collectively transform a set of inputs into a set of outputs (goods or services) using people and tools. Business processes can be observed at many levels in an organisation: within a business unit, across business units, or across the whole organisation.

Of particular importance are **end-to-end** business processes. These go right across the business units and are typically triggered by external sources (e.g., a customer requesting a service).



To appreciate the importance of end-to-end processes, consider the way information solutions are often used in the finance industry.

Business units generally have an internal focus that narrows their view to the local processes that exist within the unit. As a consequence, these local processes drive their requirements for information solutions, and this results in applications that are built to specifically support them. This gives them what they want: solutions that are tailored to their needs, and solutions that they can exercise control over.

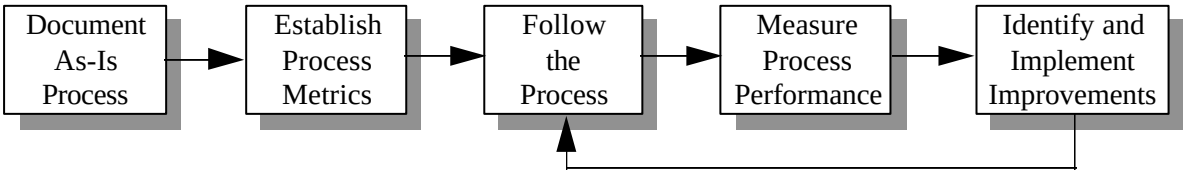


The problem with this approach is that it results in a silo-style organisation that is too rigid in its boundaries. A business never stays stationary. Business opportunities change, markets change, customers change, technologies change, and so on. Given that change is always present, an organisation's success will largely depend on its ability to accommodate change and to use it to its advantage. The way an organisation designs its business processes and the information systems that support them will, to a large degree, determine its ability to deal with change. Under the silo approach, there is very little scope for moving the boundaries between the business units. Moving a boundary too much will break the local processes and even split applications (as illustrated by the above diagram).

Over time, however, boundaries must move so that the business can adapt itself to change. The accumulated effect of these changes is that processes become patchy and applications deteriorate under pressure to conform to conflicting requirements. In other words, the more change takes place, the less the organisation is able to cope with it.

3.1.2 Business Process Improvement

Rising customer demand for better and cheaper products and services has forced most businesses to seriously consider process improvement in order to stay competitive. Most companies that have embarked on process improvement have adopted the **continuous improvement model**, as illustrated below.



This method is effective in generating gradual, incremental process improvements. Over the last decade, however, several factors have accelerated the need to improve processes faster; most notably:

- New technologies (e.g., Internet) are rapidly bringing new capabilities to businesses, and hence raising the competitive bar.
- The opening of world markets and increased free trade are bringing more companies into the marketplace, which is in turn increasing competition.

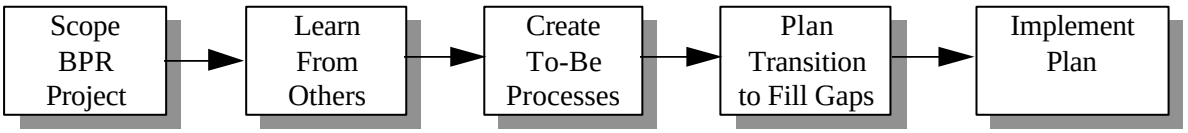
As a result, most companies that are hard-squeezed are no longer content with gradual improvement, but are looking for breakthrough performance leaps. This demand has led to the emergence of a more radical approach called Business Process Re-engineering (BPR).

3.1.3 Business Process Re-engineering (BPR)

BPR takes a different approach to process improvement which, at the extreme, assumes that the current process is irrelevant and should be redesigned from scratch. BPR proponents argue that we should disassociate ourselves from the present, project ourselves into the future, and ask some fundamental questions:

- What should the process look like?
- What do our customers want it to look like?
- What do other employees want it to look like?
- How do the best-in-class companies do it?
- What benefit can we get by using new technology?

The following diagram illustrates the BPR approach. It begins with defining the scope and objectives of the BPR project. A learning process next follows that involves customers, employees, competitors, non-competitors, and the use of new technology. Based on this improved understanding, a new set of 'to-be' processes are designed. A transition plan is then formulated which aims to close the gap between the 'to-be' processes and present. This plan is then implemented.



Because of its clean slate approach, BPR offers a far greater potential for realising breakthrough improvements.

3.2 Business Modelling Concepts

3.2.1 Abstraction versus Instance

Those with an understanding of object-orientation concepts would be familiar with the distinction between an abstraction (e.g., class) and its instances (i.e., objects). An **abstraction** depicts a general concept that has no physical existence, whereas an **instance** is a specific manifestation of the abstraction that has physical existence. For example, 'book' is an abstraction that may be

defined as 'a collection of pages with text printed on them', whereas "my copy of 'A Brief History of Time' by Stephen Hawkins" is an instance of the book concept.

This distinction is equally important when dealing with business processes, and allows us to differentiate between:

- a **process definition**, which is provided as a recipe that describes the process in terms of its constituent parts, and
- a **process instance**, which represents a specific case of performing the process.

The distinction is similar to that between a cuisine recipe and someone's specific attempt of following the recipe to produce a dish. It is important to note that any given unique abstraction may have many (potentially infinite) instances. Abstractions used in an information system (e.g., classes, business processes) are defined only once, during the development of the system. Subsequently, these abstractions are instantiated numerous times during the operational lifetime of the system.

3.2.2 Business Process Definition

A business process is defined in terms of these abstractions:

- **Triggers**. These are events outside the process that cause the process to be instantiated (i.e., kick-off the process).
- **Activities**. These are the building blocks of the process. The key difference between an activity and a process is that, whereas multiple individuals typically perform a process, only one person performs an activity. Therefore, it requires no further collaboration.
- **Queues**. As a process progresses from one activity to the next, there is a need to hold the work somewhere, until the person doing the next activity is free to pick it up. This is very similar to the concept of in-trays in an office. You receive new work in your in-tray, where it piles up until you can pick it up, do your bit to it, and put it in someone else's in-tray. Queues enable a process to be performed in an asynchronous fashion.

We will use the following symbols to depict these abstractions:

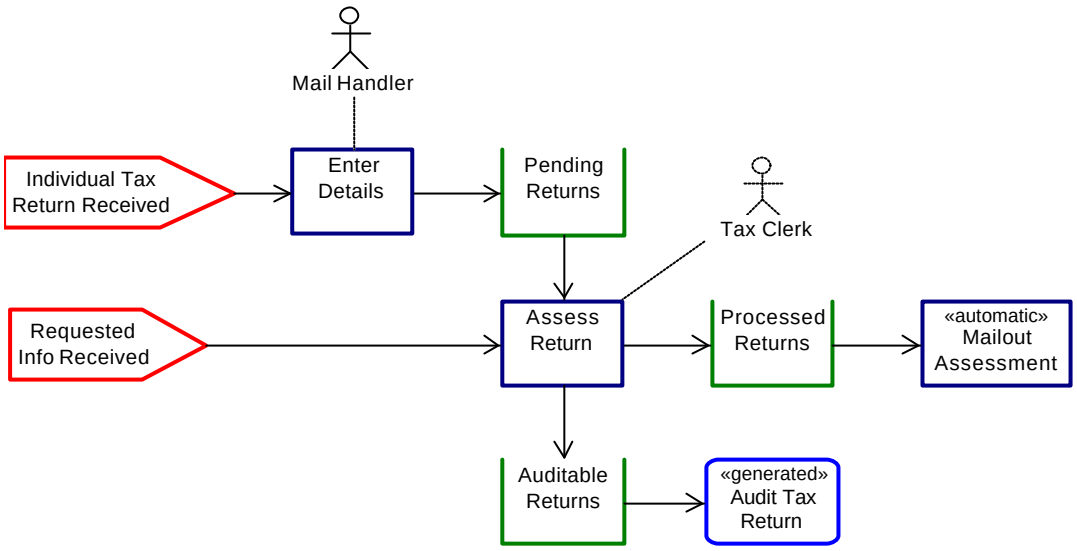
The ubiquitous arrow depicts the flow between these abstractions. *Process* and *Activity* symbols may have one of the following stereotypes:

- *Manual*, implying that the process/activity is totally manual (i.e., is done by the user and involves no interaction with a system).
- *Automatic*, implying that it is totally automatic (i.e., is done by the system and involves no further interaction).
- *Semi-auto*, implying that it is partially done by the user (e.g., checking paper forms) and partially by the system (e.g., the system checking certain calculations).

- *Generated*, implying that it is automatically generated by the system.

If no stereotype is specified then this means that the process/activity is performed through the interaction of the user with the system.

For example, consider an Individual Tax Return process at the Taxation Office:



This process map states that the process is kicked-off by the *Individual Tax Return Received* trigger. The *Enter Details* activity is performed by a *Mail Handler*. Once this activity is performed, the work is deposited into the *Pending Returns* queue. In the *Assess Return* activity, a *Tax Clerk* takes the work from this queue and assesses the tax return. If the return has missing information, then this information is requested from the return filer, and the activity is suspended until the filer provides the requested information (i.e., *Requested Info Received* trigger). The outcome of the assessment is either a fully processed return, or the initiation of a tax audit. Processed returns go into the *Processed Returns* queue and are then handled by the automatic *Mailout Assessment* activity. Auditable returns go into the *Auditable Returns* queue and result in the automatic generation of an *Audit Tax Return* process (defined elsewhere).

3.2.3 Activity Definition

As stated earlier, an activity is a process segment that is performed by one person. It is defined in terms of these abstractions:

- **Actions**. These are the building blocks of the activity. Unlike an activity, an action is an atomic step. Consequently, a user can perform an activity partially (i.e., do some of its actions) and complete the rest at some later stage. This is not possible with an action, which offers no further breakdown to the user.
- **Branches**. These can be used to constrain the flow of control between actions. There are 3 types of branches, all of which have one-to-many or many-to-one cardinality:
  - An **and** branch implies that all the 'many' actions to which it connects must be completed.

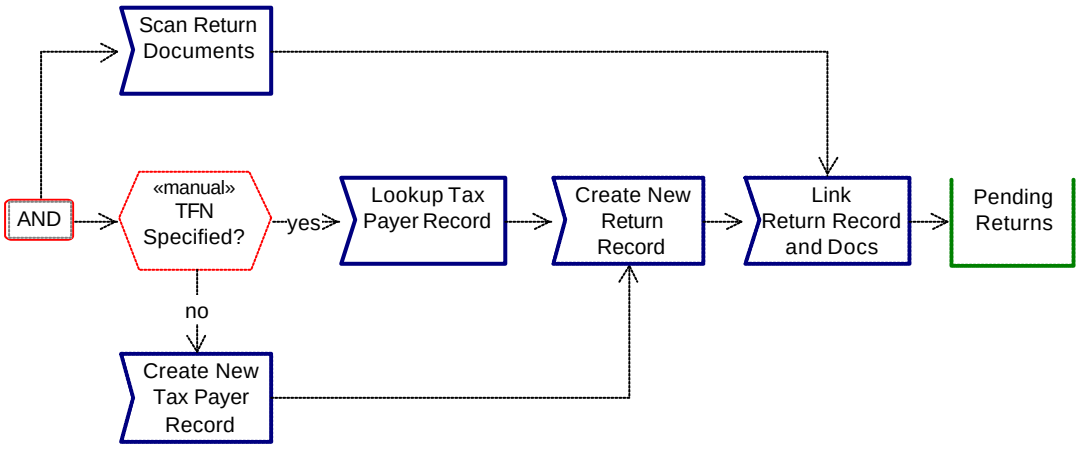
- An **or** branch is similar to an 'and' branch, except that at least one of the 'many' actions to which it connects must be completed.
- An **xor** branch is similar to an 'or' branch, except that exactly one of the 'many' actions to which it connects must be completed.

- **Conditions**. These allow the flow of control to be redirected based on the outcome of certain logical conditions.
- **Documents**. Most processes in the service sector deal with a variety of documents (e.g., letters, bills, invoices). Documents are not only generated by processes/activities, they may also serve as input to other processes/activities.

We will use the following symbols to depict these abstractions:

As before, an arrow depicts the flow between these abstractions. The *manual*, *automatic*, and *semi-auto* stereotypes can be used with *Action* and *Condition* symbols. Additionally, an action may have the *optional* stereotype, implying that, at the user's discretion, it may or may not be performed.

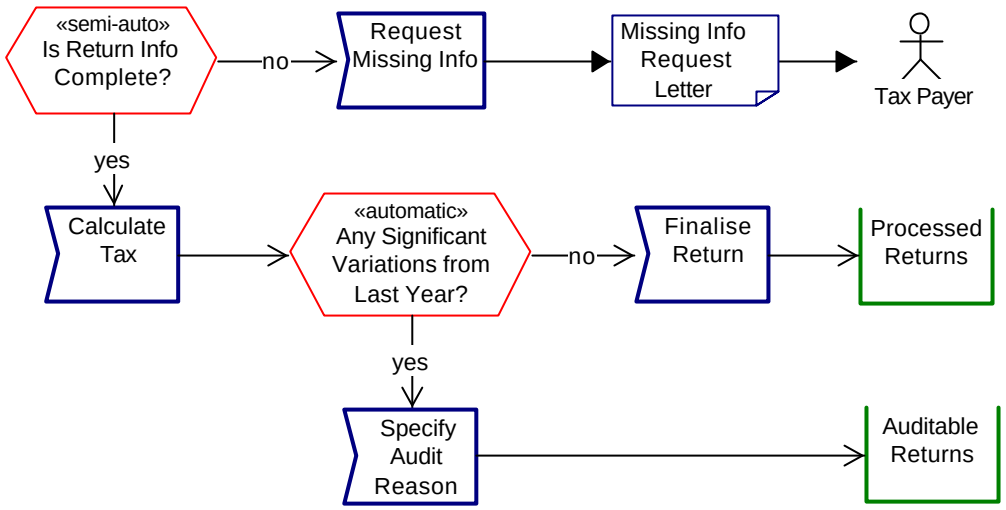
Referring back to our earlier tax return process example, each activity in that process map is refined into an activity map. For example, the *Enter Details* activity is refined into the following activity map:



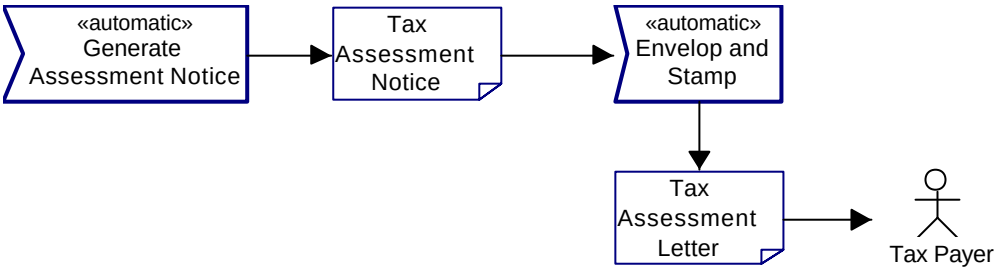
This activity map states that the return documents must be scanned (*Scan Return Documents* action) *and* the user must manually check whether the TFN is specified on the return. If so, then the tax payer's record is looked up on the system (*Lookup Tax Payer Record* action). Otherwise, a new tax record needs to be created on the system (*Create New Tax Payer Record* action). Then a new tax return record is created (*Create New Return Record* action), and the scanned documents are linked with this tax record (*Link Return Record and Docs* action). Work is then passed onto

the *Pending Returns* queue. Note that this matches a similar flow from the activity to the queue in the parent process map shown earlier. The use of the *and* branch here signifies that the scanning and the checking of TFN must both be done, but can be done in either order.

Similarly, the *Assess Return* activity is refined into the following activity map:



Finally, the *Mailout Assessment* activity is refined into this activity map:



Both actions in this activity map are automatic, implying that they are performed by the system without human intervention.

3.2.4 Action Definition

For the purpose of business process modelling, each action is defined informally using a narrative. It is not practical at this stage to formalise the definition of an action, because this will require the business objects on which the action operates to have been defined. These business objects are defined in the next phase (i.e., requirements analysis).

As an example of an action narrative, here is how the *Create New Tax Payer Record* action in our tax example may be defined:

Action:	**4. Create New Tax Payer Record**
Description:	Allows the user to key-in the details of a taxpayer, and creates a record for the

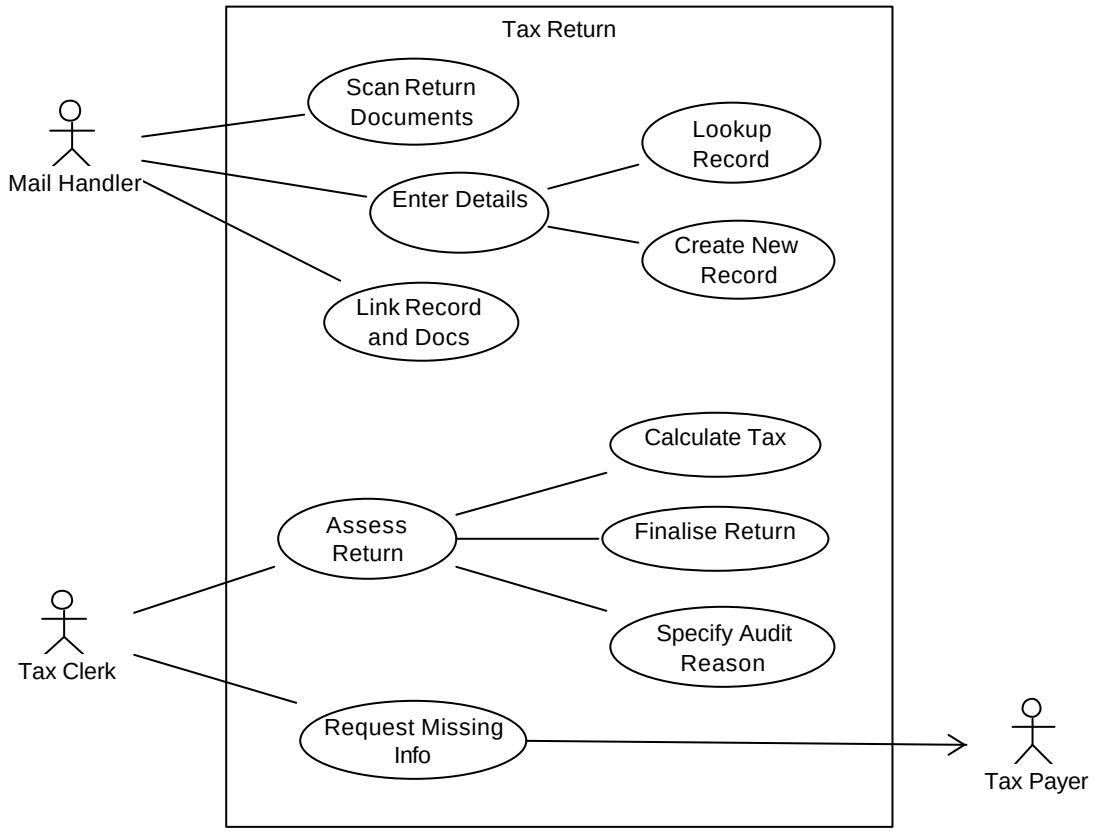
	taxpayer.
Input:	Details of a tax payer (TFN is optional)
Output:	Unique reference number for the record.

3.3 Use-Case Modelling

An alternative approach to producing a business model is to focus on the business functions rather than the business processes. This is essentially what has been popularised as the **use-case** approach.

Each use-case captures a way of using the system, that is, a business function. Because use-cases focus on the functionality that the system will provide rather than the business processes that it should support, the resulting system will be function centric. Consequently, this approach is suitable for developing vertical applications, whereas business process modelling is better suited to enterprise applications.

To illustrate the difference between the two approaches, if we tried to model the Individual Tax Return process described earlier using use-cases, we might come up with the following use-case model.



The key differences with the process model are:

- The activities (*Enter Details* and *Assess Return*) become use-cases that refer to lower-level use-cases (which were actions in the process model).

- Some of the actions (e.g., *Scan Return Documents*) become use-cases in their own right.
- There is no longer a clear notion of flow of control or conditions.
- If developed further, the use-case approach results in an application that does not convey or enforce a business process. The user is expected to know the process and to use the application to perform the steps that reflect his understanding of the process steps and their correct sequence.

4. Application Modelling

As stated earlier, application modelling is concerned with *how systems support the business*. It produces an external view of the solution, and does not assume any specific implementation technology.

In application modelling, we produce 3 types of models:

- **Class diagrams**, which depict the business objects underlying the business processes (or use-cases).
- **Scenarios**, which illustrate how business objects exchange messages in order to support the actions (or uses-cases).
- **User interface models**, which illustrate how the business functionality is presented to the user.

4.1 Business Objects[1]

Business processes (and business functions) involve business objects. Each business object represents a major abstraction in the business domain that is characterised by data and behaviour. The data represents the persistent state of the object, and the behaviours represent what can be done to the object, which in turn determines how the data is manipulated.

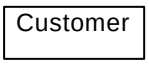
Given a business model (expressed as process maps or use-cases), we must analyse it to identify the business objects. This is not a mechanical task and requires a good understanding of the business domain and a bit of creativity. For each object, we must identify:

- Its **attributes** (these represent the persistent data that record the object state).
- Its **methods** (these represent the behaviour of the object, i.e., the things that you can do to the object).
- Its **relationships** to other objects (e.g., does this object make use of other objects?)

This results in a **static model** (called **class diagram**) that provides a lot of information about the objects, without saying anything about how they work together in supporting the business processes or use-cases.

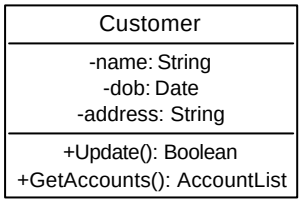
4.1.1 Class Diagrams

The UML notation is used for defining a class diagram, and includes the following symbols:



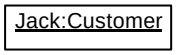
A rectangle represents a **class**, where the name of the class (*Customer* here) appears inside the box.

[1] The term *object* is somewhat misleading here; what we really mean is 'business class', because we are referring to an abstraction rather than an instance. Unfortunately, the IT literature tends to use the term 'object' to mean 'object' and 'class'; the exact intention being implied by the context.

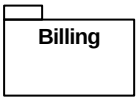


Attributes and **methods** of a class may optionally be displayed inside it. Attributes appear immediately below the class name, separated by a horizontal line from it. Methods appear beneath the attributes, again separated by a horizontal line. The amount of information displayed can vary. For example, we can display just an attribute name, or the name and its type. An attribute/method may have one of the following signs before it:

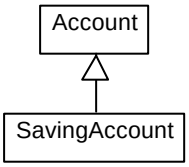
- A minus (-) sign means that it is **private**.
- A hash (#) sign means that it is **protected**.
- A plus (+) sign means that it is **public**.



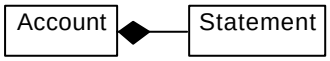
A **class instance** uses the same symbol as a class, except that a colon precedes the class name (and the colon may be optionally preceded by the instance name). The resulting string is underlined to highlight the fact that it is an instance.



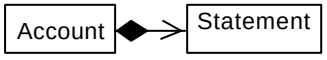
This symbol represents a **package**. Packages are used to organise diagrams into logical hierarchies. For example, we may have a class diagram that captures billing-related classes, another diagram that captures accounts payable-related classes, and so on. Each of these diagrams can be denoted by an appropriate package.



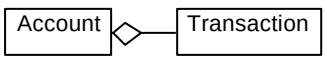
A link with a hollow arrow-head denotes **inheritance**. Here, for example, *SavingAccount* inherits from *Account*, which means that *SavingAccount* inherits the attributes and methods of *Account* and may additionally have its own attributes and methods. *Account* is a **generalisation** and is referred to as a **superclass**. *SavingAccount* is a **specialisation** and is referred to as a **subclass**.



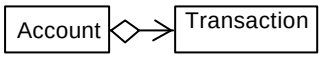
A link with a solid diamond denotes **composition**. Here, for example, an *Account* is composed of (i.e., **has**) a *Statement*. This means that *Statement* is an integral part of *Account* and cannot exist independently of it. Composition implies bi-directional navigation: you can navigate from *Account* to *Statement*, and vice-versa.



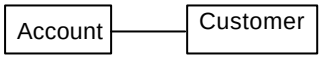
This is a variation of composition, called **uni-composition**. It implies that you can only navigate from *Account* to *Statement*, and not the other way round.



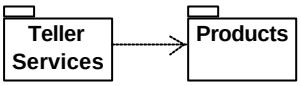
A link with a hollow diamond denotes **aggregation**. Here, for example, an *Account* is an aggregation of (i.e., **refers to**) a *Transaction*. This means that *Transaction* can exist independently of *Account*. Aggregation implies bi-directional navigation: you can navigate from *Account* to *Transaction*, and vice-versa.



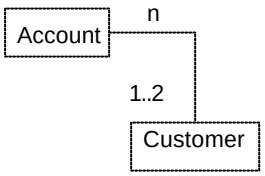
This is a variation of aggregation, called **uni-aggregation**. It implies that you can only navigate from *Account* to *Transaction*, and not the other way round.



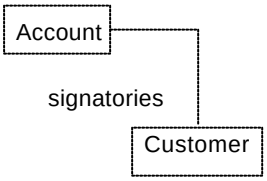
A plain link denotes an **association**. Here, for example, *Account* is associated with *Customer*.



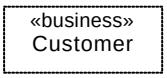
A dotted link with an arrow-head denotes **dependency**. Here, for example, the *Teller Services* package is dependent on the *Products* package.



Composition, aggregation, and association relationships may have **cardinalities**. These appear as labels near the end-points of the link. Here, for example, the cardinalities state that an *Account* is associated with one or two *Customer*s, and a *Customer* may be associated with any number of *Account*s.



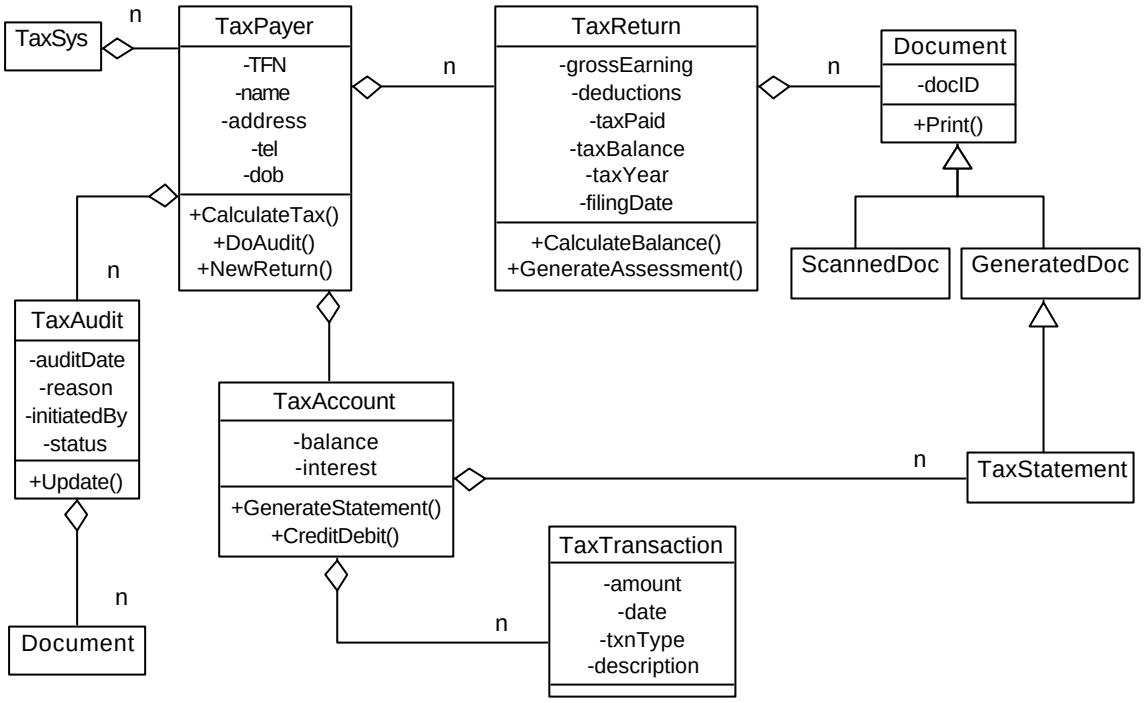
A relationship may also have specified **roles** for the objects to which it is connected. Here, for example, *Customer*(s) have the role of *signatories* in their relationship with *Account*.



A word enclosed in «» is called a **stereotype**. It provides a way of extending the UML notation. Stereotypes may appear on classes, packages, and relationships. Here, for example, a *«business»* stereotype is used to distinguish *Customer* as a business class.

4.1.2 Example

An object analysis of our tax return process example might produce the following class diagram.



This diagram states that the tax system (*TaxSys* class) is an aggregation of *TaxPayer*s. Each *TaxPayer* has zero or more *TaxReturn*s, and each *TaxReturn* has an associated set of *Document*s. A *Document* may be a *ScannedDoc*ument or a *GeneratedDoc*ument; furthermore, a *TaxStatement* is an example of a *GeneratedDoc*. For each *TaxPayer* a *TaxAccount* is held which records the tax owed by/to the *TaxPayer*. This *TaxAccount* has *TaxTransaction*s recorded against it. Also, *TaxStatement*s may be generated against this account. Finally, a *TaxPayer* may be subjected to *TaxAudit*s. Each such audit may have various associated *Document*s.

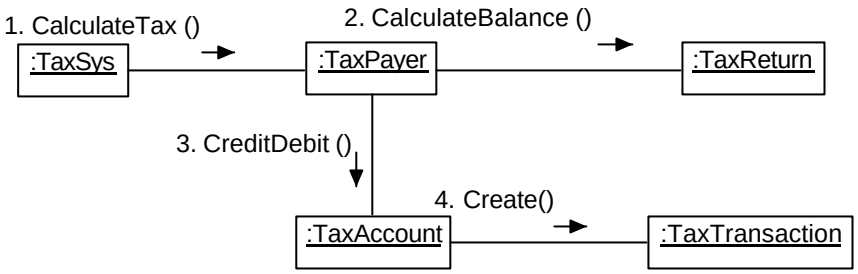
It is worth noting that not all the information captured by this class diagram comes directly from the process model. For example, additional questions need to be asked in order to identify the class attributes and their relationships. Also, *TaxAccount*, *TaxTransaction*, and *TaxStatement* are not even referred to by the process model, but emerge out of further analysis of the problem domain. In summary, to identify business objects, one should not be limited by the information provided by the business processes; other useful sources of information should also be considered.

4.2 Scenarios

Having identified the business objects, we can now attempt to describe how each action (or use-case) is supported by these objects. These descriptions are called scenarios and can be expressed in two forms: as collaboration diagrams or as sequence diagrams.

4.2.1 Collaboration Diagrams

A collaboration diagram describes a scenario as the exchange of messages between objects. For example, the *CalculateTax* action in our process example can be described by the following collaboration diagram.

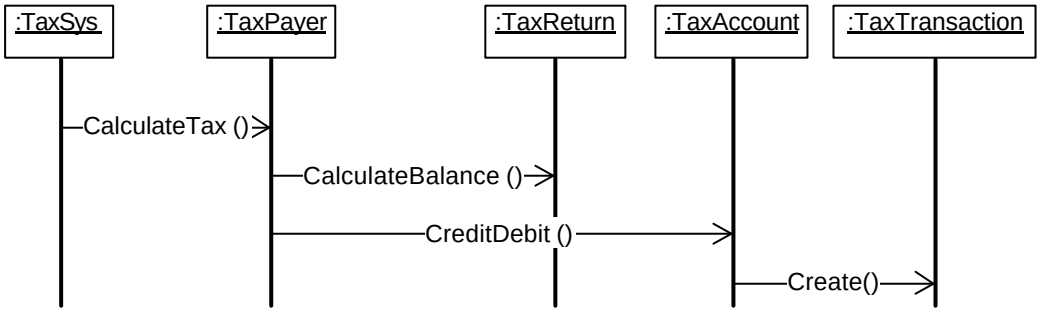


Each message is depicted by a small arrow, emanating from the object that issues it, and directed to the object that receives it. The receiving object must support the message (i.e., the message must be one that is exposed by the receiving object). The messages are numbered to depict their logical sequence.

The above diagram, therefore, states that *TaxSys* issues a *CalculateTax* message to *TaxPayer*, which in turn issues a *CalculateBalance* message to *TaxReturn*. When the latter returns, *TaxPayer* uses the calculated tax balance to issue a *CreditDebit* message to *TaxAccount*, which in turn *Create*s a *TaxTransaction* object to record the credit/debit. Upon completion of these messages, control returns to *TaxSys*.

4.2.2 Sequence Diagrams

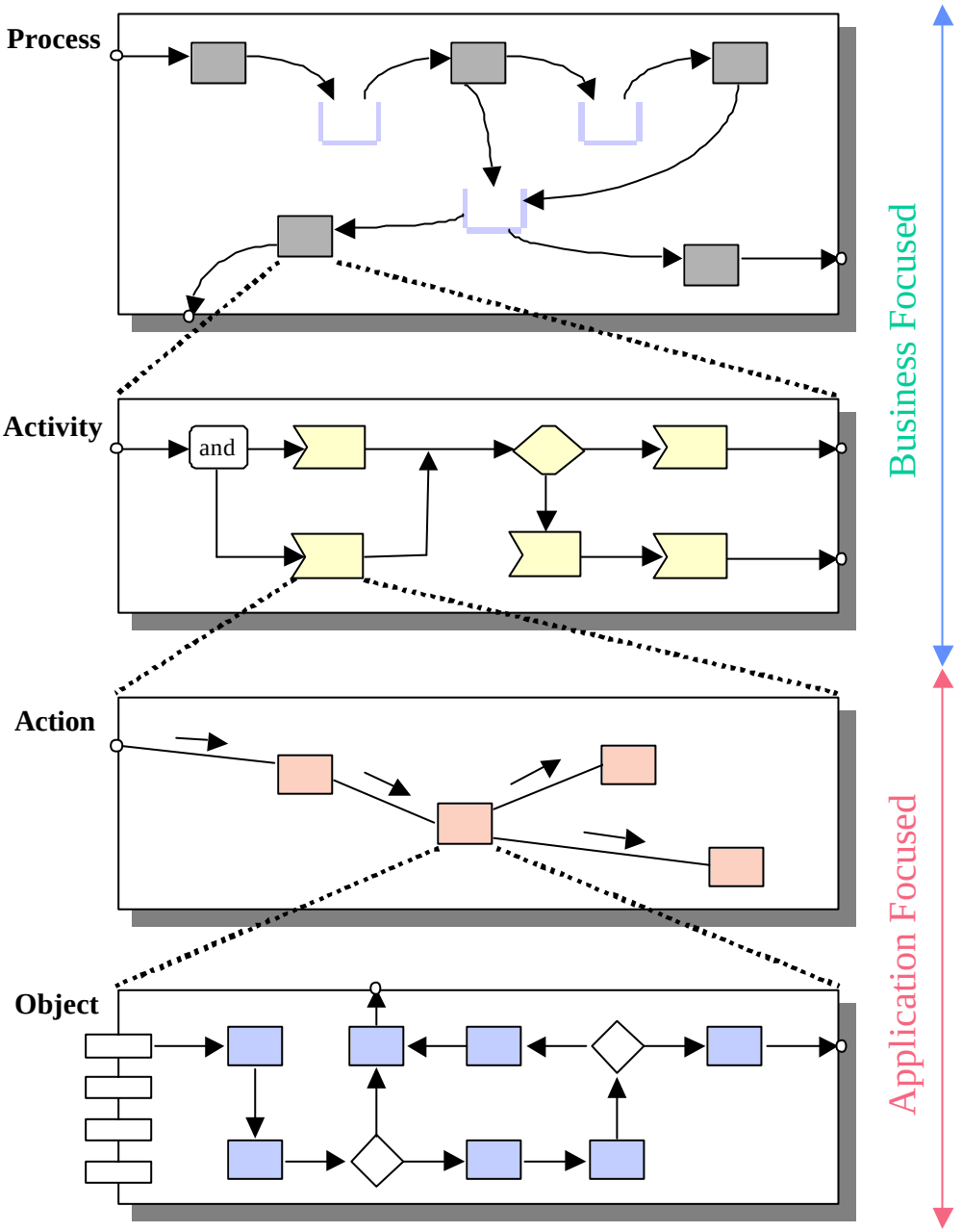
A sequence diagram describes a scenario as the interaction among objects in time sequence. For example, the above collaboration diagram can be represented as the following sequence diagram.



The time sequence is denoted by the vertical line below each object. Time flow is downwards.

4.2.3 Completed Business Model

With the addition of business objects and scenarios, we now have a complete business model that consists of four layers, as illustrated below. Each lower layer contains a refinement for each of the 'abstractions' in the layer above it.



The **process** layer defines the end-to-end business processes covered by the model. The **activity** layer specifies each of the activities that comprise the business processes in the process layer. The **action** layer specifies each of the actions that comprise the activities in the activity layer. When an action is performed, it affects one or more business objects (denoted by boxes in the third layer).

The **object** layer specifies the business objects that ultimately realise the business functionalities behind the business processes.

4.3 User Interface Models

Two things make the user interface a crucial part of any business application:

- The user interface is the means by which the user is given access to the application's business functionality. Regardless of how sophisticated and comprehensive the underlying business functionality might be, if the user interface is poor then all this will go to waste – the user will never have the opportunity to benefit from it.
- Ultimately it is the user interface that shapes the way the user works. If the conceptual model portrayed by the user interface is a close reflection of the user's work practices, then it will be quickly understood and accepted by the user. On the other hand, if the conceptual model is so complex, confused, or alien that the user finds it very difficult to relate to, it will put unnecessary burden upon the user and reduce his/her productivity. In other words, the user interface should be designed to match the user, not the other way round.

Common sense dictates that user interfaces should be designed based on the principle of recognition rather than recollection. A successful user interface is one that draws upon the background and business experience of its users to provide an intuitively obvious way of using it. A poor user interface is one that requires extensive formal training and/or memorisation of an extensive list of commands and procedures.

4.3.1 Metaphors

Sensible use of metaphors can make a user interface much more accessible to its target audience. A metaphor is "a figure of speech in which an expression is used to refer to something that it does not literally denote, in order to suggest a similarity".

A user interface metaphor relies on a user's familiarity with a common concept in order to suggest how to do something unfamiliar. Graphical User Interfaces (GUIs) are full of such examples:

- Use of a 'trash can' as a means for deleting files.
- Use of 'drag and drop' as a means for moving objects around.
- Use of 'desktop' as a way of visualising and organising your work.
- Use of 'directories' as a way of organising files.
- Etc.

Before we go about designing a new user interface, we should spend some time exploring potentially useful metaphors that can enhance the design. These should then influence the design.

4.3.2 Mock-ups

There are two ways to present a user interface design:

- As static images (hand-drawn or created using a desktop tool), with associated prose that describes the dynamic behaviour of the interface.
- As a live mock-up created using a visual development environment (e.g., VB or Java).

The latter is always preferred, for two reasons. Firstly, people find it easier to relate to a mock-up. They can play with it and quickly come up with feedback about its good and bad aspects. Secondly, experience suggests that a good user interface is almost never created in one go. It often involves many iterations, during each of which the interface will undergo (sometimes major) changes. It is invariably more productive and more effective to do these changes in a visual development environment.

The creation of a mock-up should be guided by the following principles:

- The **scope** of the mock-up should be the business processes (or use-cases) that the application will support. If someone suggests a screen or GUI element that cannot be directly or indirectly justified in terms of this scope, it should be excluded.
- The **objective** of the mock-up should be to show how a business process (or use-case) is performed both in terms of the static information presented (i.e., screens) and the dynamic flow between the screens.
- Under no circumstances, a mock-up should be allowed to become anything more than a mock-up. There is always the risk of some users or business decision makers concluding that simply enhancing or 'finishing' the mock-up will result in the final application. That is why that the **purpose** of the mock-up should be clearly established and communicated up-front: to agree on a user interface design, and nothing more. The mock-up must be understood as a **throw-away** artefact.
- **Involving** potential end-users in the design process can save a lot of time and energy. They not only can come up with valuable insights and ideas that would have been inaccessible to the technologists, they can also become potential champions in introducing the user community to the new application.

5. System Modelling

As stated earlier, system modelling is concerned with *how systems are realised using technology*. It produces an *internal* view of the solution, showing how its different parts interact in order to support the external, application view.

5.1 Multi-tier Architectures

Modern information systems tend to be distributed: they use a multi-tier, client-server architecture that can support the non-functional requirements of the system. These requirements often involve:

- *Geographic distribution*. Most large businesses are geographically distributed and require their information systems to be accessible irrespective of geographic boundaries.
- *Scalability*. Once implemented, an information system may be required to serve the needs of a growing user base. It should be possible to scale the system by increasing the computing resources, and without resorting to design changes.
- *Heterogeneous computing environment*. Most organisations have a multi-vendor computing environment, consisting of incompatible hardware and system software. An information system may be required to operate across the boundaries of such incompatible domains.

Distribution of an information system involves tiering, which divides the system into separate partitions that can run on separate (but networked) physical resources.

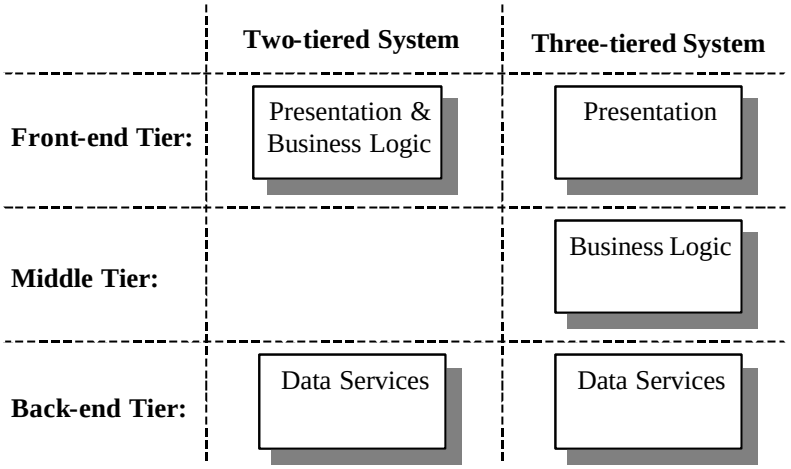
Tiering should not be confused with layering or, put differently, a software tier is not the same as a software layer. A software **layer** is a conceptual abstraction, which packages a defined set of functionalities and makes it accessible through a logical interface. A software **tier**, on the other hand, is the physical packaging of the implementation of one or more software layers.

Layering and tiering are the means through which an architect logically and physically partitions a system, respectively. The software layers are defined during application modelling, whereas the software tiers are established during system modelling. Layering affects tiering. By defining layers, an architect is providing opportunities for creating tiers.

Early client-server systems were mainly **two-tiered**, consisting of a front-end and a back-end tier. Under this model, the front-end (also known as a **fat client**) contains the presentation as well as the business logic, and the back-end consists of a database system. A two-tiered system is inflexible in a number of ways:

- There is no logical separation between presentation and business logic, which makes maintenance difficult.
- The system is not scalable.
- In most cases, the system lacks openness, because it is based on a vendor's proprietary technology (e.g., database vendor's technology).

Modern client-server systems are mainly **three-tiered**, where the business logic is separated from presentation and occupies its own middle tier. This approach provides a basis for overcoming the shortcomings of two-tiered systems.



The main challenge of building an effective three-tiered system is in designing and building the middle tier. This tier needs to be designed in a way that delivers all the promised advantages of three-tiered systems:

- It needs to remove the business logic from the front-end, so that the front-end only deals with presentation concerns. This is known as the **thin client** approach.
- It needs to componentize the business logic in a way that can be distributed across hardware resources so that it becomes scalable.
- It needs to provide open interfaces so that other systems can be integrated with it with minimal effort.

The middle tier is often designed such that it can be subdivided into further tiers – hence the term **n-tiered** – to achieve greater scalability and to provide additional open interfaces. For example, it may be broken into these three sub-tiers:

- A *dynamic content* tier that generates the content to be presented by the front-end.
- An *application* tier that realises the business processes offered by the system.
- A *business object* tier that realises the business objects underlying the system.

Productive development of the middle tier requires the use of **middleware** technology, which provides the necessary tools for packaging the tiers, defining the interfaces between them, communication across the tiers, and transaction management. These are complex, system level activities that are well beyond the scope of a typical project. Middleware eliminates the need to work at this level, and provides a reliable basis for the project to focus on developing business functionality.

Component technology goes a step further from middleware technology. It also provides a means for packaging functionality so that it is highly independent and reusable. Whereas middleware allows one to develop business functionality without dealing with low-level system issues, component technology aims to provide a basis for productively developing new functionality by reusing an existing set of components (as well as creating new ones for future reuse).

System modelling must deal with all these issues. Specifically, it must:

- Deliver a proper tiering model into which the software layers map.
- Define the abstractions (e.g., components) that comprise each tier, and show how they support the application model.
- Select the technologies that are to be used to build the system (e.g., middleware/ component technology, persistence technology, presentation technology, scripting technology).

5.2 Front-End Models

Under the thin client architecture, the front-end deals with presentation only. Based on the requirements, the front-end may be designed according to one or more of the following models:

- **Conventional client**. This is a traditional style GUI client that involves intensive interaction between the user and the system (e.g., as in a spreadsheet, a word processor, or a CAD system). These interfaces go beyond the simple input/output of data in most business applications, and require the lower-level controlling of the windowing system to deliver the expected performance and ease of use. Although the client is not fat, it does not have a zero footprint, as it needs to contain components that can support the interaction and communicate with the middle tier.
- **Page-based client**. This is a browser-based client (i.e., runs within a HTML browser) with zero footprint. The interface consists of a set of web pages, through which the user enters and/or views data. A web server in the middle tier receives the data entered by the user, and generates the dynamic HTML to be displayed by the browser.
- **Content-based client**. This is also a browser-based client, but is implemented as a downloaded **applet** that runs in the browser and communicates with a **servlet** in the middle tier's web server. This is useful when we need to provide functionality directly at the front-end that helps a user organise and interact with the system. In this case, the server exchanges raw content (typically expressed in XML) with the client instead of HTML pages.

Because of their simplicity, flexibility, and web-readiness, more and more business applications are adopting the page-based and content-based styles. The key difference between these two is the point of control for presentation. With a page-based interface, the client receives preformatted presentation (i.e., HTML) from the middle-tier, which it cannot alter. In this case therefore, the middle-tier determines presentation. In a content-based client, however, the middle-tier delivers raw data (i.e., content) to the client, and it is up to the client to determine how to display it. This is closer to the conventional client model.

The following table provides a useful comparison of the three styles of clients. Your choice should be guided by the extent to which a given style best matches your requirements. It is not uncommon for a system to provide two or all of these styles for the same system, but targeted at different users.

Client Type	Presentation Determined By	Can Deliver Dynamic Content	Is Web Enabled	Requires Client Installation	Interaction Overheads
Conventional Client	Client	No	No	Yes	Low
Page-based Client	Middle-tier	Yes	Yes	No	High
Content-based Client	Client	Yes	Yes	Dynamic	Medium

5.2.1 Screen Specifications

By the end of application modelling, the user interface concept (including mock ups) should have been established, so that once system modelling commences, the expected look and feel of the user interface (and the underlying metaphors) is well understood.

Although the user interface concepts developed during application modelling include the actual screens, they do not get into the details of each screen. The layout of each screen is rough and not finalised, the widget types and the type of data to be displayed in each widget is not fully specified, and the data entry validation rules are excluded. These details are specified during system modelling.

Each screen in the user interface (e.g., a window, a dialog box, or a web page) needs to be specified, including:

- A picture of the actual physical layout of the screen. If possible, this is best produced using the tools that come with the development environment. Most such environments are visual and provide resource editors for painting the GUI screens in an interactive fashion. This has the advantage that the screens can be used ‘as is’ later in the development cycle.
- For each element (i.e., widget) in the screen, the data associated with that element needs to be identified and its type specified. In most cases, this data is expected to map directly to an attribute in a business object.
- The validation rules for the screen elements need to be defined. These apply at two levels: (i) field-level validation rules applicable to a given element (e.g., valid date), and (ii) screen-level validation rules that operate across elements (e.g., ‘start date’ should predate ‘end date’).

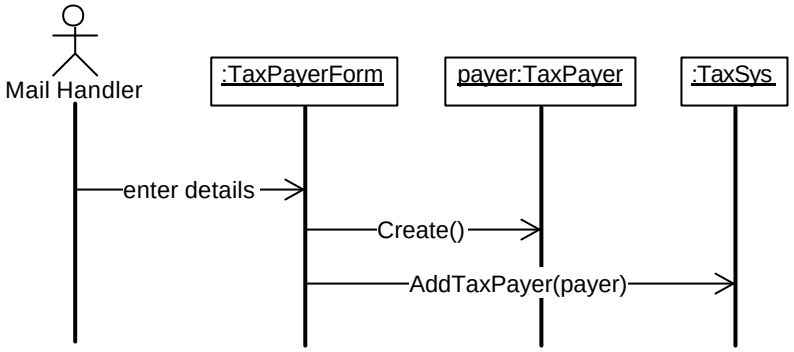
5.2.2 Navigation

The navigation paths across user interface screens need to also be defined. In a conventional interface, for example, this may involve specifying how an action performed in one screen (e.g., pressing a button) leads to the display of another screen. In a web-based interface, this will also involve the specification of the hyper links.

5.2.3 Boundary Objects

Under object technology, the user interface is developed as a collection of **boundary** objects. A boundary object is a visual object that handles the interaction between the system and the end-user.

For example, consider the 'Create New Tax Payer Record' action in our tax office model. In this action, the user (a *MailHandler*) enters the details of the taxpayer into a form, which in turn creates a new *TaxPayer* object and adds it to the system. The following sequence diagram illustrates this scenario.



TaxPayerForm is an example of a boundary object. It collects the required information from the user and communicates it to the rest of the system.

In this case, it should be obvious that the contents of the form will closely resemble the attributes of the business object it corresponds to. The following design considerations are worth noting.

- Given that a change to a business object attribute will result in a similar change to the boundary object, a good design will attempt to minimise this overhead. Under the conventional client model, there is not much that can be done. A browser-based client, however, can alleviate this problem by using an intermediate generator object that can take a meta-data specification of the object and generate a matching boundary object for it.
- Field-level validation rules should be the responsibility of the boundary object. Simple field-level validation rules (e.g., numeric value) can be directly encoded into the boundary object. More complex field validation rules should be handled by invoking methods on objects that can perform the validation on behalf of the boundary object. Any form of validation that requires database access (e.g., valid postcode) is best handled by an object in the middle tier. Other complex filed validation rules (e.g., valid date) can be implemented by utility objects available in the front-end and shared across the user interface.
- Screen-level validation rules should be the responsibility of the underlying business object. For example, when requested to create or modify a business object, the business object should perform the necessary validations to ensure that the resulting object will be valid.

In favour of better usability, more and more modern user interfaces incorporate **dynamic** behaviour. A typical example of this is when the value entered into a given form field affects the remaining fields, which may be enabled/disabled or displayed/hidden as a result. For example, in an account form,

choosing a certain 'account type' may result in the rest of the form being altered so that only information relevant to that account type is displayed.

Also, there are cases where the relationship between a screen and the underlying business objects is **one to many**. For example, a 'fund transfer' screen typically involves two account objects, and an 'account summary' screen may involve an account object and a list of transaction objects.

Dynamic screens and screens with one-to-many business object relationship are best supported through **proxy** objects in the middle tier. Coding these complexities in the proxy object rather than the boundary object results in greater decoupling of the two tiers.

Finally, most modern interfaces are required to support pick lists. A **pick list** is a list of possible values for a field, which the user can simply choose from, rather than entering the value directly. Pick lists may be static or dynamic. A static pick list provides a list of values that remain forever fixed (e.g., a list of 12 months in the year). A dynamic pick list provides a list of values that change over time (e.g., a list of available products). Static pick lists are simple enough to be coded directly into the boundary object. Dynamic pick lists should be implemented in the middle tier and made available to the front-end through appropriate interfaces.

5.3 Middle-Tier Models

The middle tier is by far the most complex part of any 3-tier client-server system. This is where the bulk of the development effort should go to ensure a resilient design that can best cope with future changes. The scalability of the system, in particular, is directly impacted by the design of the middle tier.

The middle tier must achieve the following:

- Realise the business processes defined during business modelling.
- Release the business objects defined during application modelling.
- Implement transaction control (using the chosen middleware).
- Provide the services required by the front-end through a well-defined interface.
- Manage efficient communication with the back-end to ensure persistence for the relevant objects.
- Partition the middle tier into components that can be distributed across physical resources with minimal overheads.

The middle tier is constructed using three types of objects: entity objects, control objects, and boundary objects. These are discussed below.

5.3.1 Entity Objects

An entity object is a persistent object that participates in transactions. The main entity objects in a system are the business objects themselves. Ideally, the persistence of an entity object is managed

by the middleware. Where this is not the case, the entity object itself is responsible for issuing back-end requests to maintain its own persistence.

Each business object specified during application modelling is implemented as an entity object. For example, in our tax office model, *TaxPayer*, *TaxReturn*, and *TaxAccount* are all entity objects. If these objects are specified fully during application modelling then their implementation as entity objects is fairly straightforward. In practice, however, this is rarely the case. Often additional analysis is required to pin down the business rules for each such object, and to devise appropriate algorithms for the non-trivial methods.

The implementation of an entity object should satisfy two things: (i) it should conform to the object specification, and (ii) it should preserve the relationships between the objects. For example, in our tax office model, there is a one-to-many aggregation relationship between *TaxPayer* and *TaxReturn*. The methods of these objects must ensure that this relationship is not violated.

Another important implementation consideration is that, ideally, an entity object should make no assumptions about the underlying data representation for object persistence. This ensures that changes to the data model will not directly impact the entity objects.

It is worth noting that not all entity objects are business objects. For example, consider a 'workspace' object that remembers the position of the windows for a given user and their preferences. This object needs to be persistent so that the next time the user logs in, his/her workspace can be restored. Another example is an object that records transaction related statistics for audit or tuning purposes.

An important property of an entity object is that it is shared by all users, i.e., different users can engage in different transactions that potentially impact the same entity object. Of course, such changes cannot occur concurrently. It is the responsibility of the transaction management mechanism of the middleware to manage this.

From a design point of view, because entity objects are persistent (i.e., require database IO), they consume precious resources. The middle tier design, therefore, should try to exercise control over the number of entity objects that need to be kept in memory at any point in time.

Most middle tier designs employ techniques such as smart pointers, object caching, and database connection pooling to reduce the load on the resources.

5.3.2 Control Objects

A control object sits between boundary and entity objects, and performs a task on behalf of the boundary object. In other words, a control object is an extension of a boundary object. Rather than implementing the task directly inside the boundary object (tight coupling), the task is implemented independently (loose coupling). This provides enormous flexibility. Specifically:

- Changes to the boundary object will not necessarily impact the control object, and vice versa.

- The control object can be remoted, so that it runs on a server, independently of the boundary object. This has the added advantage that the control object can be shared between a number of clients.

Unlike entity objects, control objects are non-persistent, i.e., they do not need to maintain state beyond the scope of the task they perform. However, some control objects may need to maintain state for the length of the task (i.e., duration of conversation with a client). Based on this, control objects are divided into two categories:

- A **stateful** control object needs to maintain the state of some (or all of) its attributes for the duration of the conversation with its client. When the client finishes with the control object, the state disappears. For example, a *PolicyValue* control object would need to remember the *policy number* attribute of the control object for the duration of the conversation.
- A **stateless** control object does not maintain a conversational state for a particular client. When a client invokes the method of a stateless object, the object's instance variables may contain a state, but only for the duration of the invocation. When the method is finished, the state is no longer retained. For example, a *CommissionPayment* control object would require no state due to its atomic nature.

Except during method invocation, all instances of a stateless control object are equivalent, and hence can be assigned to any client. Because stateless control objects can support multiple clients and require no persistence, they can offer better performance and scalability for applications that require large numbers of clients. As a rule of thumb, therefore, you should avoid using stateful control objects unless you really need to.

In general, each **transaction** is implemented in the middle-tier by a control object. The control object is itself responsible for transaction control (i.e., commit and rollback). In some cases, this responsibility may be abstracted by the middleware through deployment descriptors.

However, not all control objects implement transactions. A control object may also implement a **query**. For example, a query to retrieve the last 20 transactions for a given account may be handled by a control object. This control object retrieves the account and the transactions (all entity objects) and makes them available to the client as required. This approach is particularly useful for queries that may return very large lists (e.g., retrieve all the transactions for a given account). Given that entity objects are expensive, the control object can be implemented to do this smartly. For example, it may retrieve only a limited number of transactions at a time, and retrieve additional transactions only when the client needs them.

Proxy objects that support boundary objects in the front-end tier (as described in Section 5.2.3) are also examples of control objects.

5.3.3 Boundary Objects

The interface between the middle-tier and the other tiers may involve boundary objects. For example, where the middle-tier is implemented as a CORBA server, the CORBA interface exposed

by the middle tier consists of boundary objects. Such boundary objects, however, are essentially wrappers and provide no further functionality. They simply adapt a tier's interface to a format that is agreed with another tier.

5.3.4 Long Transactions

The implementation of transactions as control objects was discussed earlier. These transactions are known as short transactions, i.e., they correspond to a task performed by a client at a specific time (e.g., transfer funds from one account to another).

There is another class of transactions, known as long transactions, which span beyond one task. A long transaction consists of a number of tasks, performed by potentially different users, and at different points in time. For example, in a process-centric system, an end-to-end process (e.g., home loan application) may be regarded as a long transaction.

Long transactions are generally very complex and pose a number of challenges:

- Two or more long transactions can overlap by operating on the same entity objects, with no guarantee that all will commit.
- Because a long transaction is performed in a piecemeal fashion, it has to cope with possibly modified entity objects between its successive steps. Also, when the transaction is about to commit, it needs to verify that interim modifications to the entity objects have not invalidated earlier steps.
- Rolling back a long transaction may be a non-trivial task, because other transactions may now be relying on the modifications made by the transaction to entity objects.
- Unlike a short transaction, a long transaction needs to be implemented as a persistent object. The transaction may take hours, days, or even months to complete, during which time the users participating in the transaction may login and out a number of times, and the system may be restarted.

These complexities go beyond virtually all middleware products' transaction management capabilities. Also, effective management of a long transaction often requires access to relevant business rules, which reside well beyond the middleware domain. As a result, when long transactions are used, the middle-tier needs to implement its own long transaction management facility. This facility needs to implement the following:

- *Transaction persistence* (i.e., realisation as an entity object).
- *Versioning of entity objects.* The entity objects modified by a long transaction need to be versioned to avoid the problem of overlapping transactions modifying the same entity object in inconsistent ways. Versioning can ensure that changes made by overlapped transactions are mutually excluded.
- *Version reconciliation*. If a transaction is performed on the basis of an old version of an entity object, when committing, the transaction needs to reconcile itself against the latest version of the entity object.

- *Transaction rollback*. With proper versioning of entity objects, this will be straightforward (i.e., simply throw away the changes made by the transaction).

An unusual aspect of long transactions is that occasionally there may not be enough information to reconcile entity object versions at commit time. This will necessitate asking the user to make the decision.

5.4 Back-End Models

The back-end tier of a three-tier client-server system is conceptually quite simple: it provides the persistent storage for the entity objects of the middle tier. This tier provides two things:

- A data model for the storage of the objects.
- Adapter objects for accessing and updating the data.

5.4.1 Data Models

For a bespoke system, a new data model needs to be synthesized. The input to this process is the object model created during application modelling and later enriched by system modelling. In most cases, this is achieved by:

- Mapping each entity object to a **table**.
- Identifying appropriate **keys** for each table, based on the access paths required by object methods.
- Modelling the **relationships** between objects either using keys or additional tables. There are three possible cases:
    - A *one-to-one* relationship can be modelled using a key in either or both tables. For example, there is a one-to-one relationship between *TaxPayer* and *TaxAccount*, and this can be represented by having a *TaxPayer* key in the *TaxAccount* table, and/or vice versa.
    - A *one-to-many* relationship can be modelled using a key in the 'many' table. For example, a one-to-many relationship between *TaxPayer* and *TaxReturn* can be represented by having a *TaxPayer* key in the *TaxReturn* table.
    - A *many-to-many* relationship can be modelled using an additional table, which combines the keys for both tables. For example, a many-to-many relationship between *Customer* and *Account* can be represented by a *CustAccRel* table that has attributes for recording the keys of both tables.

Any non-trivial system, however, poses further data modelling challenges that need to be addressed. One of these involves the issue of inheritance and how to model it at a data level. There are no strict rules for handling inheritance, but the following two guidelines cover almost all cases:

- Where an inheriting object adds very few attributes to the inherited object, use the same table to represent both. Obviously the table needs to include the attributes of both objects, and where an object does not use a certain attribute, that attribute is simply ignored. For example, specialisations of an *Account* object (e.g., *LoanAccount*, *SavingAccount*, *ChequeAccount*)

are likely to add very few additional attributes. In this case, it makes sense to have one *Account* table to represent all account types. An additional attribute in the table can be used to denote the account type.

- Where the inheriting object adds many more attributes to the inherited object, use a separate table for either, and include a key in the 'inheriting' table to refer to the 'inherited' object's table. For example, a *ContactPoint* object may have specialisations such as *PhysicalAddress*, *TelecomAddress*, and *WebAddress*. These are fairly disjoint, so it makes sense to have a table for each.

The degree to which the data model needs to be normalised is a database design issue and should be determined by the DBA. One of the advantages of OO modelling is that it tends to result in data models that are highly normalised.

The ER data model synthesized from the object model needs to be kept in sync with it. Processes need to be put in place to ensure that developers work off a consistent set of object and data models. Experience has shown that this is an area where most projects encounter avoidable problems.

Where legacy systems are involved (as is the case in most client-server projects), additional constraints are imposed. If all the data for the system is to be sourced from legacy systems, then this rules out the possibility of developing a brand new and clean data model. Instead one has to adapt the data provided to serve the needs of the middle-tier (and vice versa).

These constraints should be absorbed underneath the business object layer and should never be exposed beyond it. Any higher-level component that uses the business objects should not have to (or be allowed to) assume any knowledge about the underlying data model. This decoupling minimises the impact of a data model change on the rest of the system.

5.4.2 Data Access Objects

Depending on the data model, there may or may not be a need to have an additional layer to manage access to data. For example, in a bespoke system with a clean, new relational data model, the business objects may access this data through an open interface such as ODBC or JDBC. No additional processing is required.

However, where legacy systems are involved, there may be a need to perform additional processing to adapt the legacy data to the format required by the business objects. For example, a business object layer that talks XML is incompatible with a data layer consisting of CICS/COBOL legacy systems. This can be overcome by developing adapter objects that map the data between the format used by the business objects and the format required by the legacy systems. These objects may also perform additional house keeping as relevant to the legacy systems involved.

6. Testing

There are two philosophical views on the purpose of software testing:

- The purpose of software testing is to demonstrate that there are no defects.
- The purpose of software testing is to detect defects.

The problem with the first view is that it promotes the wrong psychology – it encourages the testers to come up with test cases that are likely to run successfully, rather than ones that break the software.

The second view is based on the premise that any non-trivial application will always contain defects. The true value of testing is to detect as many defects as is economically feasible (and then to fix them) in order to increase confidence in the reliability of the application.

6.1 Introduction

Testing should take place throughout the development lifecycle, so that defects are detected and fixed at the earliest opportunity. Most artefacts produced during the development lifecycle can be tested if they are expressed in appropriate notations. For example, we can test a business process by passing hypothetical cases through it to check if there are any gaps in the flow or logic.

Extensive testing, however, cannot be undertaken until executable code has been produced. Because code is the ultimate artefact of software development, it should be subject to more testing than other artefacts.

6.1.1 Testing Process

The underlying process for all forms of software testing is the same, and should adhere to the following principles.

Before testing can begin, a **test plan** must be produced. The test plan defines a set of test cases, the completion criteria for the tests, and the environment required for performing the tests.

Each **test case** in a test plan consists of two things: **test data** and **expected result**. When a test case is performed, the software is exercised using the test data and the actual result is compared against the expected result. A match or discrepancy is then recorded in the **test log**. The test log is a record of the test cases performed and their outcome.

A test plan must conform to a defined **test strategy**, which provides an overall framework for all the different forms of testing for an application.

6.1.2 Testing Approaches

There are two general approaches to testing: white box testing and black box testing.

In **white box** testing, test cases are formulated based on the internal design of the artefact being tested. For example, if the artefact is the code for a class, then we can use the internal logic of the class to create test cases that exercise all control flow paths through the code, so that every statement is executed at least once.

In **black box** testing, the artefact being tested is treated as a black box that, given a set of inputs, produces some output. This means that no knowledge of the internal design of the artefact is assumed. Test cases are created based on the range of the input values that the artefact should accept (or reject) and their relationships to the expected output values.

White box testing is more applicable to lower-level artefact, such as functions, classes, and components. Black box testing is better suited to higher-level artefacts, such as application modules, applications, and integrated systems. Given that white box and black box testing tend to expose different types of defects, it is generally recommended that a combination of the two be used in the development lifecycle in order to maximise the effectiveness of testing.

6.1.3 Testing Techniques

A number of different testing techniques have been devised for use at various stages of development. On its own, no one technique is sufficient to produce adequate test cases. The techniques, rather, serve as a toolbox that test designers can utilise to design effective test cases. The most widely recognised techniques are:

- **Coverage testing**. This is a white box technique that attempts to achieve a certain level of code coverage. Typical types of coverage considered are:
  - *Statement coverage* requires that enough test cases be created to exercise every statement in code at least once.
  - *Branch coverage* requires that all the alternatives of every decision branch in the code be exercised at least once.
  - *Condition coverage* requires the true/false outcome of every condition in the code is exercised at least once. This is not the same as branch coverage, since the logical expression for a branch may, for example, be a conjunction of multiple conditions.
- **Boundary value testing**. This is a black box testing technique, where test cases are written such that they involve input or output values that are around the boundary of their permissible range. For example, if a function takes a ‘day of month’ argument then values such as -1, 0, 1, 30, 31, 32 are around the boundary of permissible values and would be good input test data.
- **Cause effect graphing**. This is a white box technique that involves mapping out the logical relationships between causes (input values representing a meaningful condition) and effects (output values representing a corresponding meaningful outcome). An example of a cause might be a ‘positive amount in a transaction’ and its corresponding effect may be ‘an account being credited’. Once all the possible causes and effects have been listed, then test cases are designed such that each cause and all its potential effects (and each effect and its potential causes) are exercised at least once.

- **Domain analysis**. This is a black box technique that involves analysing the domain of each input value, and subdividing it into sub-domains, where each sub-domain involves 'similar values' in the sense that if you use one of these values in a test-case, it will be as good as any other value in that sub-domain. For example, the domain of an 'amount' input value might be subdivided into the ranges 0-1000, 1001-100,000, and >100,000; these three sub-domains being different from an authorisation point of view. Based on the outcome of domain analysis, a minimum number of test cases can be designed that will have a high yield, by avoiding the use of 'similar' values in separate test cases.
- **Error guessing**. Error guessing involves using your imagination to come up with test cases that are likely to break the artefact being tested. There are no particular rules in this technique, except that the more unusual and nonsensical the test-cases, the more likely is their effectiveness.

One point that is often overlooked by test case designers is the use of erroneous input data. It is important that testing should involve at least as many invalid or unexpected input values as valid or expected ones. Once an application goes live, it will be used in ways that go beyond the original expectations of its designers. It is important that the application behaves gracefully in face of invalid or unexpected input data.

6.1.4 Testing Stages

During its development lifecycle, software is subjected to testing at a number of stages, as summarised by the following table.

Stage	Type of Testing	By Who	Purpose
After a 'unit' has been coded.	Unit Testing	Developers	To detect any variances between the 'unit' behaviour and its specification.
When a number of units are combined to create an executable module (e.g., a tier in a distributed application).	Integration Testing	Developers	To detect any discrepancies in the interfaces between the units (e.g., mismatching message format between two components).
When the modules are combined to create an integrated application (e.g., all the tiers in a client-server application).	Integration Testing	Developers	To detect any discrepancies in the modules that makes up the application (e.g., a client expecting a server to send a single record in response to a request, whereas it actually sends a list of records).
When an integrated application is robust enough to undergo extensive testing (e.g., after all cross-tier mismatches have been fixed).	System Testing	Test Team	To detect any variances between the way the application behaves and its official requirements model (e.g., a request should take < 5 seconds, whereas in practice it takes 30 seconds to complete).
When an application is required to inter-operate	Integration Testing	Test Team	To detect any discrepancies in the interfaces between applications

with other applications in an enterprise solution.			(e.g., a change of employee address is reflected in the company Intranet, but not in the payroll application).
When a solution is delivered to its intended customer.	Acceptance Testing	Customer	This is the only type of testing performed by the customer. It gives the customer the opportunity to verify the fit of the application with respect to their requirements, before officially accepting the application.

As indicated by this table, integration testing comes in various forms and happens at a number of stages. Unfortunately, most publications on testing refer to integration testing as if it only happens once in the development lifecycle, and this has led to much confusion. The confusion can be avoided by bearing in mind that integration testing should happen whenever a number of artefacts of similar characteristics are combined, be they classes, components, modules, application tiers, or entire applications.

System testing is by far the most labour intensive testing stage, because there are so many different types of tests that need to be performed. Because of the specialised nature of system testing, it must be performed by a dedicated test team that specialises in this area. In particular, it should never be done by the developers themselves, since they neither have the required expertise, nor the appropriate psychological profile to do it effectively.

In most projects, the customer relies on the developers to help them with creating an acceptance test plan. Because of the extensive nature of system testing, virtually everything that needs to be verified in acceptance testing is likely to have been tested for in system testing. As a result, acceptance testing often involves a subset of the test cases used for system testing.

6.1.5 Regression Testing

With any type or stage of testing, one has to deal with the problem of tested artefacts being modified. The dilemma is that, on the one hand, we are aware that the modifications may well have introduced new defects and, on the other hand, we do not want to incur the overhead of completely retesting the artefact every time we make changes to it.

The aim of regression testing is to check if the modifications have caused the artefact to regress (i.e., have introduced new defects into it). It should be obvious that unless regression testing can be done quickly, the whole development cycle grinds to a halt. There are two ways of regression testing productively:

- By selecting a high yield subset of the original tests and only running these.
- By using appropriate testing tools to automate the testing process, so that they can be performed with minimal human intervention.

The latter is much more effective, but does involve a greater upfront investment in creating the test scripts required by the automated testing tools.

6.2 Test Planning

Successful testing requires appropriate planning. Given that test planning requires substantial amount of effort and a considerable length of time, the actual planning must begin well before the artefacts to be tested are ready for testing.

Test planning covers four key activities:

- The creation of a **test strategy** that will guide all testing activities.
- The creation of **test plans** for the different stages of testing.
- The setting up of the **test environment** so that the test plan can be carried out.
- The creation of **test scripts** for automated testing.

These are separately discussed below.

6.2.1 Test Strategy

The test strategy provides an overall framework for all testing activities in a project (or group of related projects). This may sound motherhood and unnecessary but, in practice, it can have a significant impact on the way testing is carried out.

The primary objectives of a test strategy are to:

- Ensure a consistent approach to testing at all stages.
- Spell out the things that are crucial to the project as far as testing is concerned (e.g., iteration speed, robustness, completeness).
- Provide guidelines on the relevant testing techniques and tools to be used.
- Provide guidelines on test completion criteria (i.e., define what is 'good enough testing'), and the expected amount of effort that should go into testing.
- Provide a basis for reusing test cases and identifying areas that can benefit from automation.
- Establish standards, templates, and the deliverables that need to be used/produced during testing.

6.2.2 Test Plan

A test plan is a documentation of the test cases to be performed and associated instructions for performing the tests. Two levels of test plans are often used:

- A **master test plan** is used to identify the high-level objectives and test focus areas.
- A **detailed test plan** is used to document the test cases produced as a result of analysing the test focus areas identified in the master test plan.

6.2.3 Test Environment

The computing environment to be used for conducting the tests needs to be planned, so that it is ready for use when testing commences. Issues to be considered include:

- *Construction of test harnesses*. A test harness is a software tool that can be used to invoke the software being tested and feed test data to it. A test harness is necessary when the software being tested cannot be executed on its own (e.g., a component). Test harnesses are particularly valuable during the earlier stages of testing (e.g., unit testing).
- *Setting up of test boxes*. Later stages of testing (e.g., system testing) require the use of 'clean' test machines that are set up specifically for the purpose of testing. Unless the test machine is 'clean', when an error occurs, it is difficult to determine whether it is due to the effect of existing software and historical state of the machine or it is genuinely caused by the application being tested. This level of isolation is essential in order to have any faith in the test outcome.
- *Creation of test databases*. Most applications use a database of some form. The database schemas need to have been set up and the database populated with appropriate test data so that the tests can be carried out.
- *Setting up of security access and accounts.* Access to most applications is subject to security rights and having appropriate user accounts. Additional accounts may also be needed to access backend systems, databases, proxy servers, etc. These accounts and access rights need to be properly set up to enable the tests to be carried out without unnecessary obstacles.

6.2.4 Automated Testing

Given the extensive effort that usually goes into testing (especially regression testing), it often makes economic sense to use automated testing tools to cut down the effort. Most such tools have a built-in scripting language, which can be used by test designers to create test scripts. The test tool uses the test script to invoke the application being tested and to supply it with specific test data, and then to compare the outcome of the test against expected test results. Once set up with appropriate scripts, the test tool can rapidly perform the test cases (often with no human involvement) and to record their success/failure outcome.

With automated testing, the bulk of the effort goes into the creation and debugging of the test scripts. This can require substantial development effort and must be planned in advance.

6.3 System Testing

As stated earlier, system testing is the most labour intensive testing stage and of direct relevance to acceptance testing. It largely involves black box testing, and is always performed with respect to the requirements baseline (i.e., it tests the application's implementation of the requirements). There are many different types of tests that need to be planned and performed to test every aspect of the application, as summarised by the following table.

Type of System Test	Baseline	Purpose
Function Testing	Business/Application Model	Identify defects in the realisation of the business functions/processes.
Exception Testing	Business/Application Model	Identify incorrectly handled exception

		situations.
Stress Testing	Non-functional Requirements	Identify stress levels beyond which the application cannot operate.
Volume Testing	Non-functional Requirements	Identify data volume levels beyond which the application cannot operate.
Scalability Testing	Non-functional Requirements	Identify the limit beyond which the application will not scale.
Availability Testing	Non-functional Requirements	Measure the availability of the application over a prolonged period of time.
Usability Testing	Business/Application Model Non-functional Requirements	Identify problems that reduce the application's ease of use.
Documentation Testing	Business/Application Model Non-functional Requirements	Identify problems in the user documentation for the application.
Installation Testing	System Model	Identify problems in the installation process for the application.
Migration Testing	System Model	Identify problems in the migration of data from the legacy system to the application.
Coexistence Testing	System Model	Identify problems caused by the coexistence of the application with other applications in the same live environment.

Each of system tests is separately described below.

6.3.1 Function Testing

The purpose of function testing is to identify defects in the 'business functions' that the application provides, as specified by the business/application model. If this model is specified as business processes (as recommended earlier in this handbook), then the test cases are built around these business processes. If it is specified as use-cases, then the test cases are built around the use-cases.

6.3.2 Exception Testing

An exception refers to a situation outside the normal operation of an application (as represented, for example, by invalid input data or incorrect sequence of operations). For example, a mortgage application may require that before a mortgage account can be created, the mortgagee record must have been created. Therefore, an attempt to create a mortgage account when the mortgagee is unknown to the system is an example of an exception.

Exception testing is, in a way, the opposite of function testing – it tests for dysfunctional behaviour. If an application successfully passes function testing, then it is not necessarily fit for business. It may be the case that it does allow dysfunctional operations to be performed, which, from a business point of view, can lead to liability or financial loss. The purpose of exception testing is to identify exception situations that are not satisfactorily handled by the application.

6.3.3 Stress Testing

Stress testing involves observing the behaviour of the application under 'stress conditions'. Exactly what these conditions are depends on the nature of the application. For example, in a web-based

financial application, a very large number of transactions and a very large number of end-users would represent stress conditions. 'Large' in this context should be interpreted as 'equal to, greater than, and much greater than' what has been specified in the non-functional requirements. If the requirements call for the support of up to 500 concurrent users, then we should, for example, test for 500, 600, 1000, etc.

The purpose of stress testing is to identify the stress conditions that cause the application to break. If these conditions are within the expected operational range of the application, then we conclude that the application has failed the stress tests.

Stress conditions can and do arise when the application goes live. It is therefore important to know what the stress limits of the application are, so that contingency plans can be made.

6.3.4 Volume Testing

Volume testing involves observing the behaviour of the application when it is subjected to very large volumes of data. For example, if the application is a banking system having an underlying database for storing account records, volume testing will attempt to populate this database with maximum capacity records and beyond. For example, if the requirement is for the system to store up to a million accounts, then we may try to populate the database with 1 million, 2 million, and 5 million records.

The purpose of volume testing is to identify the volume levels beyond which the application will be unable to operate properly (e.g., physical storage limit or acceptable performance level).

As with stress testing, unanticipated volume levels can occur when the application goes live, and it is therefore important to know the volume limits for contingency reasons.

6.3.5 Scalability Testing

Most modern business applications are multi-user, distributed, client-server systems that serve a large user base. The 'growth' (i.e., increased use) of an application in a business is often difficult to predict. Some applications that are originally designed for a handful of users, later end up being used by hundreds or thousands of users. The ability of an application to serve a growing user base (and growing transaction volume) should therefore be an important consideration.

The degree to which the use of an application can grow without making any design changes is called scalability. A scalable architecture can ensure that the application can grow by simply adding more hardware resources to distribute the application across more and more boxes.

The purpose of scalability testing is to identify the boundaries beyond which the application will not be able to grow. Scalability testing is environmentally complex because it involves making changes to the distribution model of the application for its test cases.

There is an obvious interplay between scalability and stress/volume testing, and this needs to be taken into account when test planning.

6.3.6 Availability Testing

Each application has certain availability requirements, as determined by the business environment within which it runs. This is often expressed as a percentage for a given duration (e.g., 98% availability for the month of July).

The purpose of availability testing is to determine if the application availability falls below the minimum acceptable level. This is measured by running the application over a long duration (e.g., a week, a month, or until it falls over) while it is subjected to a realistic load.

6.3.7 Usability Testing

The purpose of usability testing is to identify any features or obstacles in the design of the application (mainly its user interface) that makes the application difficult to use. Although usability is a subjective notion, there are meaningful measures that can be employed to establish the relative usability of an application. For example, given a certain level of initial training, users can be observed with the purpose of recording measures such as:

- The average length of time needed to complete a business process/activity/action.
- The number of errors made during a business process/activity/action.
- The amount of time spent on rework due to errors.
- Given a certain task, the length of time it takes a user to find out how to use the application to do it.

6.3.8 Documentation Testing

The purpose of documentation testing is to establish the relevance, usefulness, readability, and accuracy of end-user documentation for the application. This is performed using just the supplied documentation in order to operate the application. So for each given test case, the tester will refer to the documentation to find out the instructions for performing it. In other words, the testing process will simulate the actions of an untrained user who has to use the application on basis of the information provided.

Documentation testing will detect defects in the user documentation, such as: gaps (no explanation of how to do a certain task), factual errors, ambiguity, technical jargon, and out of date information.

6.3.9 Installation Testing

The purpose of installation testing is to detect defects in the installation process for the application. Modern applications are supplied with installation tools/scripts that automate the installation process. Installation test cases involve attempting to install the application (in a clean environment) using the installation package provided (i.e., installation scripts, documentation, and release notes).

6.3.10 Migration Testing

Most modern applications are replacements for legacy systems. For business continuity, often the legacy data needs to be preserved and migrated to the new application. This is usually a very

complex problem that is addressed through development: the creation of the necessary tools/scripts to migrate the data is part of the same project.

The purpose of migration testing is to identify defects in the data migration process. Migration test cases involve attempts to migrate the legacy data to a clean installation of the application.

6.3.11 Coexistence Testing

In a live environment, most applications run in conjunction with other applications. The system test environment, however, is usually isolated and not as complex as the live environment. Potential interplay between applications (that compete for resources and interact with each other) cannot be ruled out in a live environment. The purpose of coexistence testing is to establish whether the application can successfully coexist with other applications.

Coexistence testing is usually carried out first in a pseudo live environment, and then in the live environment itself, but with restricted access.

6.4 Test Case Design

The design of test cases is, by far, the most important part of testing, since it is the quality of the test cases that determines the overall effectiveness of testing.

The layered architecture reference model (see Section 2.3.2) provides a sound basis for organising the design of test cases. Using this model, each layer is considered separately, in order to design test cases that cover that layer. This results in the test case design effort being divided into 4 categories, as summarised by the following table:

Architecture Layer	Test Case Design Effort Category
Presentation	Presentation oriented test case design
Workflow	Workflow oriented test case design
Business Object	Business object oriented test case design
Data Services	Data oriented test case design

This logical separation ensures that every important architectural consideration is fully tested. It does not, however, provide complete coverage for all system test types. Additional test cases need to be created for these, especially those that involve non-functional requirements.

All 4 categories use the same source material for designing the test cases, which consists of the following:

- **Business model**. This is probably the most important source, as it describes the business activities that the application supports.
- **Application model**. This is also important in that it provides a picture of what the application is supposed to do, and can compensate for gaps in the user documentation.
- **System model**. This is important as a formal and detailed technical source, and is especially useful in relation to non-functional requirements.

- **Non-functional requirements**. This covers the important constraints that the application should satisfy (e.g., performance requirements).
- **User documentation concept**. It is unlikely that by the time system testing commences, the user documentation would be ready. However, it is reasonable to expect that by then concept documents be at least produced, providing a terse version of the intended documentation.

We will look at each category in turn, and provide a table for showing how the source material relates to each major test focus area, and to which system test type the resulting test cases belong.

6.4.1 Presentation Oriented Test Case Design

These tests are concerned with all those aspects of the application that are manifested through the user interface. Major test focus areas are:

- *Presentation layout*, which uses the actual design of the user interface to design test cases that assess the ease of comprehension of the presentation.
- *Input data validation*, which uses the specification of the validation rules for input data to design test cases that assess the correctness of the implementation of these rules.
- *Interaction dynamics*, which uses the rules for dynamic feedback to the user (e.g., enabling/disabling of GUI elements) to design test cases that assess the correctness of the implementation of these rules.
- *Navigation*, which uses the specification of navigation paths from one window to another to design test cases that assess the correctness of the implementation of these paths.
- *Productivity*, which considers things that can affect user productivity (e.g., response time, ease of use, rework frequency) to design test cases that can identify barriers to user productivity.
- *Documentation*, which uses the user documentation concept to design test cases that can identify potential problems in the user interface documentation.

MAJOR TEST FOCUS AREA	SOURCE MATERIAL FOR TEST CASE DESIGN					INCLUDE IN SYSTEM TEST TYPE
	Business Model	Application Model	System Model	Non-fun'l Requirem'ts	User Doco Concept	
Presentation Layout		×	×			Usability Testing
Input Data Validation			×			Exception Testing
Interaction Dynamics			×			Usability Testing
Navigation		×	×			Usability Testing
Productivity		×		×	×	Usability Testing
Documentation					×	Doco. Testing

The above table summarises the major test focus areas for presentation oriented test case design. For each focus area, the source materials to be used for test case design and the system test types under which the test cases are to be documented are identified.

6.4.2 Workflow Oriented Test Case Design

These tests are concerned with the instantiation and execution of business processes. Major test focus areas are:

- *Workflow Logic*, which uses process/activity maps to design test cases that exercise the various paths through the process.
- *Workflow Data*, which involves test cases that will handle the specific data items (e.g., documents) created/manipulated by the process.
- *Security*, which involves test cases that ensure that those aspects of a business process that have restricted access are only available to users with the relevant security rights.
- *Workflow Validation*, which involves test cases that attempt to invoke exception situations for the process to see how they get handled.
- *Documentation*, which involves test cases that attempt to perform a process, based on its documentation.

MAJOR TEST FOCUS AREA	SOURCE MATERIAL FOR TEST CASE DESIGN					INCLUDE IN SYSTEM TEST TYPE
	Business Model	Application Model	System Model	Non-fun'l Requirem'ts	User Doco Concept	
Workflow Logic	×					Function Testing
Workflow Data	×	×				Function Testing
Security		×				Function Testing
Workflow Validation			×			Exception Testing
Documentation					×	Doco. Testing

6.4.3 Business Object Oriented Test Case Design

These tests are concerned with the instantiation and manipulation of business objects. Major test focus areas are:

- *Object Behaviour*, which uses the business object models to design test cases that exercise the methods of each business object.
- *Object Validation*, which involves test cases that attempt to create invalid business objects.

MAJOR TEST FOCUS AREA	SOURCE MATERIAL FOR TEST CASE DESIGN					INCLUDE IN SYSTEM TEST TYPE
	Business Model	Application Model	System Model	Non-fun'l Requirem'ts	User Doco Concept	
Object Behaviour		×				Function Testing
Object Validation		×	×			Exception Testing

6.4.4 Data Oriented Test Case Design

These tests are concerned with the storage and retrieval of persistent objects. Major test focus areas are:

- *Object Persistence*, which involves test cases that verify the correct persistence of entity objects.
- *Persistence Efficiency*, which involves test cases that measure the time required to store/retrieve persistent objects.
- *Storage Capacity*, which involves test cases that attempt to exercise the storage limits of the application.
- *Backup & Recovery*, which involves test cases for backing up the application database and then restoring it from the backup image.

MAJOR TEST FOCUS AREA	SOURCE MATERIAL FOR TEST CASE DESIGN					INCLUDE IN SYSTEM TEST TYPE
	Business Model	Application Model	System Model	Non-fun'l Requirem'ts	User Doco Concept	
Object Persistence		×				Function Testing
Persistence Efficiency			×	×		Usability Testing
Storage Capacity				×		Volume Testing
Backup & Recovery			×		×	Availability Testing