Architectural Styles

An architectural style is an attribute of software architecture which reduces the set of possible forms to choose from, and imposes a certain degree of uniformity to the architecture. It is a set of constraints on an architecture that defines a set or family of architectures that satisfies them. It is a meta-model providing the software's high-level organisation and defines ways of selecting and presenting architectural building blocks

Architectural Design Patterns are tried and tested high-level designs of architectural building blocks.

The software built for computer-based systems exhibits one of many architectural styles. Each style describes a system category that encompasses:

1. A set of components (e.g. a database, computational modules) that perform a function required by a system;
2. A set of connectors that enable "communication, coordination and cooperation" among components - interaction;
3. Constraints that define how components can be integrated to form the system;
4. Semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

Architectural Styles vs Architectural Patterns

An *architectural style* defines a family of architectures constrained by

- Component/connector vocabulary, e.g. layers and calls between them
- Topology, e.g. stack of layers
- Semantic constraints, e.g. a layer may only talk to its adjacent layers

For each architectural style, an *architectural pattern* can be defined

- It's basically the architectural style cast into the pattern form
- The pattern form focuses on identifying a problem, context of a problem with its forces, and a solution with its consequences and tradeoffs; it also explicitly highlights the composition of patterns

1. Data-flow architectures

This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. E.g. Pipes and Filters, Batch Sequential

Case Study: The P*ipe and Filter pattern*

This has a set of components, called *filters*, connected by pipes that transmit data from one component to the next. Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form. However, the filter does not require knowledge of the working of its neighbouring filters.

If the data flow degenerates into a single line of transforms, it is termed *batch sequential.* This pattern accepts a batch of data and then applies a series of sequential components (filters) to transform it.

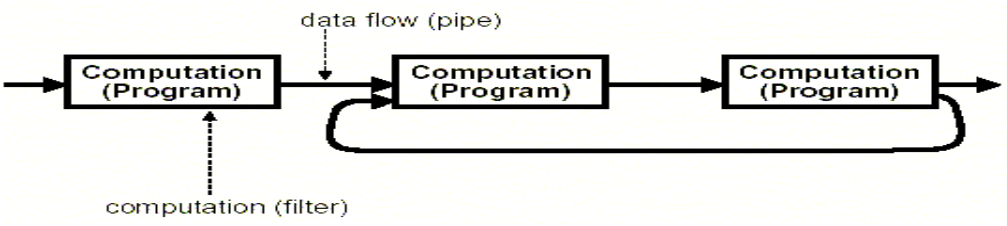


Figure 1: Pipes and Filters

Examples of Pipe and Filter Systems: Unix pipes, Image processing, Signal processing, Voice and video streaming, Compilers.

Strengths

- Makes it easy to understand overall function of the system as a composition of filter functions
- Encourages reuse of filters
- Facilitates maintenance
- Facilitates deadlock and throughput analysis

Weaknesses

- Often leads to batch-type processing
- Not good for interactive applications where incremental computations are done, e.g. incremental display updates
- Can't coordinate stream inputs
- Data transmission critical for system performance

2. Call and return architectures.

This architectural style enables a software designer (system architect) to achieve a program structure that is relatively easy to modify and scale.

A number of sub-styles exist within this category:

- Main program/subprogram architectures. This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components, which in turn may invoke still other components.
- Remote procedure call architectures. The components of a main program/ subprogram architecture are distributed across multiple computers on a network
- Other sub-styles include Object Oriented

Strengths

- Architecture based on well-identified parts of the task
- Change implementation of routine without affecting clients
- Reuse of individual operations
- Can break problems into interacting agents
- Can distribute across multiple machines or networks

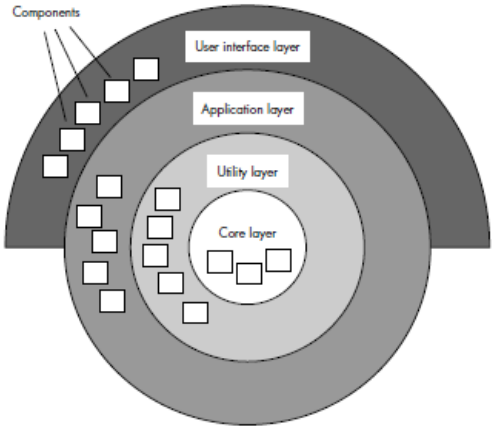
Weaknesses

- Must know which exact routine to change
- Hides role of data structure
- Does not take into account commonalities between variants
- Bad support for extendibility
- Side effects: if A uses B and C uses B, then C's effects on B can be unexpected to A

3. Layered architectures.

The basic structure of a layered architecture is as illustrated below. A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set - Each layer collects services at a particular level of abstraction. At the outer layer, components service user interface operations. At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions.

"A layered system is organised hierarchically, each layer providing service to the layer above it and serving as a client to the layer below."



Examples of Layered Systems

- ISO defined the OSI 7-layer architectural model with layers: Application, Presentation, …, Data, Physical.
- TCP/IP is the basic communications protocol used on the internet. POSA book describes 4 layers: ftp, tcp, ip, Ethernet. The same layers in a network communicate 'virtually'.
- Operating systems e.g. hardware layer, …, kernel, resource management, … user level.

Strengths

- Increasing levels of abstraction as we move up through layers means partitioning of complex problems.
- Maintenance - in theory, a layer only interacts with layers above and below, therefore change in one layer has minimum effect on another.
- Reuse - different implementations of the same level can be interchanged

Weaknesses

- Not all systems are easily structured in layers (e.g., mobile robotics)
- Performance - communicating down through layers and back up, hence bypassing may occur for efficiency reasons

4. Independent Components (Loosely Coupled)

This style consists of independent processes or objects communicating through messages. There is no control between components and modifiability is achieved by decoupling various portions of computation. Sub-styles include:

- Communicating Processes e.g. Peer to Peer, Client /Server
- Event- Based Systems which might be by Implicit Invocation.

Case Study: Implicit Invocation

With this style, rather than invoking a procedure directly or sending a message, a component announces, or broadcasts, one or more events. Other components then register interest in events. Each event has an associated routine which is automatically executed when the event is produced.
When the event is announced the system implicitly invokes all procedures that have been registered for the event.

Examples of this style are the Model-View Controller (MVC) architecture which is used for User Interfaces, Programming environment tool integration, Syntax-directed editors to support incremental semantic checking.

Strengths

- Strong support for reuse – one can plug in a new component by registering it for events.
- Maintenance - components can be added and replaced with minimum affect on other components in the system.

Weaknesses

- Loss of control of system - when a component announces an event, it has no idea what components will respond to it, cannot rely on the order in which the components will be invoked and cannot tell when they are finished.
- Ensuring correctness is difficult because it depends on context in which invoked as interactions are unpredictable.

Structured Design

A.K.A Function Oriented Design, Structured design was developed by Ed Yourdon and Larry Constantine and is generally used after structured analysis.

The concept of structured design is that a program is designed as a top-down hierarchy of modules i.e. decomposition centers on identifying the major software functions and then elaborating and refining them in a top-down manner. The top-down structure of these modules is developed according to various design rules and guidelines.

The resulting hierarchy of modules can then be evaluated according to certain quality acceptance criteria to ensure the best modular design for the program. Upon completion, the modules are to be implemented using structured programming principles.

The primary tool used for structured design is the *Structure Charts.*

Structured design is often characterized as a data flow-oriented design method because it provides a convenient transition from a data flow diagram to software architecture. The transition from information flow (represented as a DFD) to program structure is accomplished as part of a six-step process:
(1) The type of information flow is established;
(2) Flow boundaries are indicated;
(3) The DFD is mapped into program structure;
(4) Control hierarchy is defined;
(5) Resultant structure is refined using design measures and heuristics; and
(6) The architectural description is refined and elaborated.

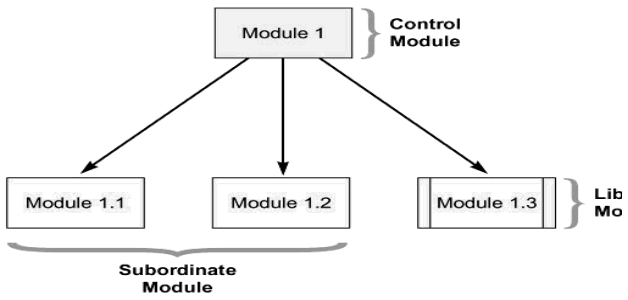
Structure Charts

Structure charts are used to graphically depict a modular design of a program. Specifically, they show how the program has been partitioned into smaller more manageable modules, the hierarchy and organization of those modules, and the communication interfaces between modules. They are based on DFD and Data dictionary.

Structure charts, however, do not show the internal procedures performed by the module or the internal data used by the module.

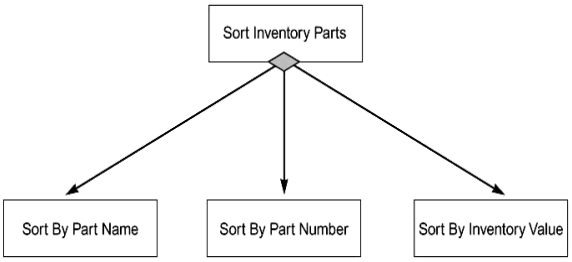
Structure Chart Notation

1. Modules (sequential logic)



- The rectangle represents a module (program or subroutine)
- Control Modules (mainline) branch to sub-modules
- Library modules are reusable and can be invoked from more than one Control Module elsewhere in the system

2. Conditions (decision logic)



- A diamond symbol located at the bottom of a module means that the module calls one and only one of the other lower modules that are connected to the diamond.
- It indicates that a control module determines which subordinate module will be invoked

3. Loops (iterative logic)

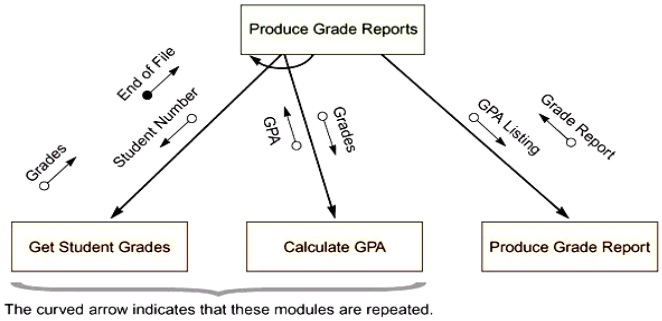
These are represented by a curved (arc shaped) arrow located across a line (representing a module call) and mean that the module makes iterative calls i.e. one or more modules are repeated.

4. Data Couple

Represented by an arrow with an empty circle, it shows the data one module passes to another.

5. Control Couple

Represented by an arrow with a filled circle, it shows a message (flag) which one module sends to another. Modules use flags to signal a specific condition or action to another module. e.g. the 'Get Student Grades' module sends a flag 'End of file' to the 'Produce Grade Reports' module.



From Analysis Model to Design Model

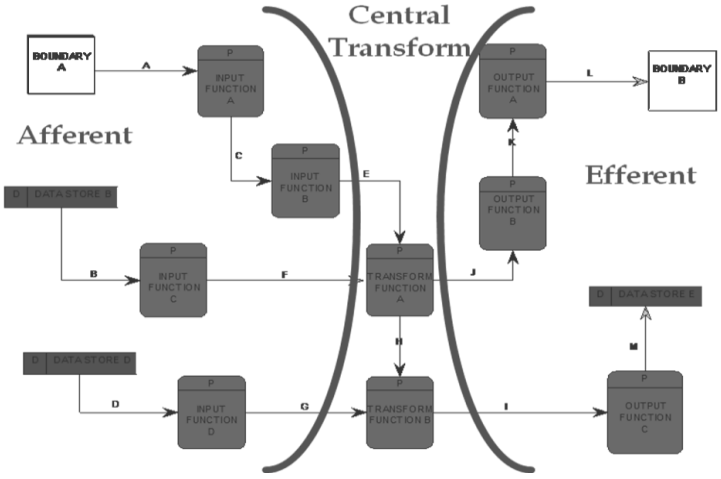
The type of information flow is the driver for the mapping approach required in step 3.
There is no practical mapping for some architectural styles, and the designer must approach the translation of requirements to design for these styles in an ad hoc fashion.

However, a data flow diagram can be mapped into program structure using one of two mapping approaches—transform mapping or transaction mapping.

Transform mapping/analysis

Transform analysis is an examination of the DFD to divide the processes into those that perform input and editing, those that do processing or data transformation (e.g., calculations), and those that do output.
It is applied to an information flow that exhibits distinct boundaries between incoming and outgoing data. The DFD is thus mapped into a structure that allocates control to input, processing, and output along three separately factored module hierarchies.

- The portion consisting of processes that perform input and editing is called the afferent.
- The portion consisting of processes that do actual processing or transformations of data is called the central transform.
- The portion consisting of processes that do output is called the efferent.



The strategy for identifying the afferent, central transform and efferent portions begins by first tracing the sequence of processing for each input. However,

- There may be several sequences of processing.
- A sequence of processing for a given input may actually split to follow different paths.

Once sequence paths have been identified, each sequence path is examined to identify processes along that path that are central transforms, afferent or efferent processes;

- Beginning with the input data flow, the data flow is traced through the sequence until it reaches a process that does processing (transformation of data) or an output function e.g. TRANSFORM FUNCTION A.
- Beginning with an output data flow from a path, the data flow is traced backwards through connected processes until a transformation processes is reached (or a data flow is encountered that first represents output) e.g. TRANSFORM FUNCTION A.

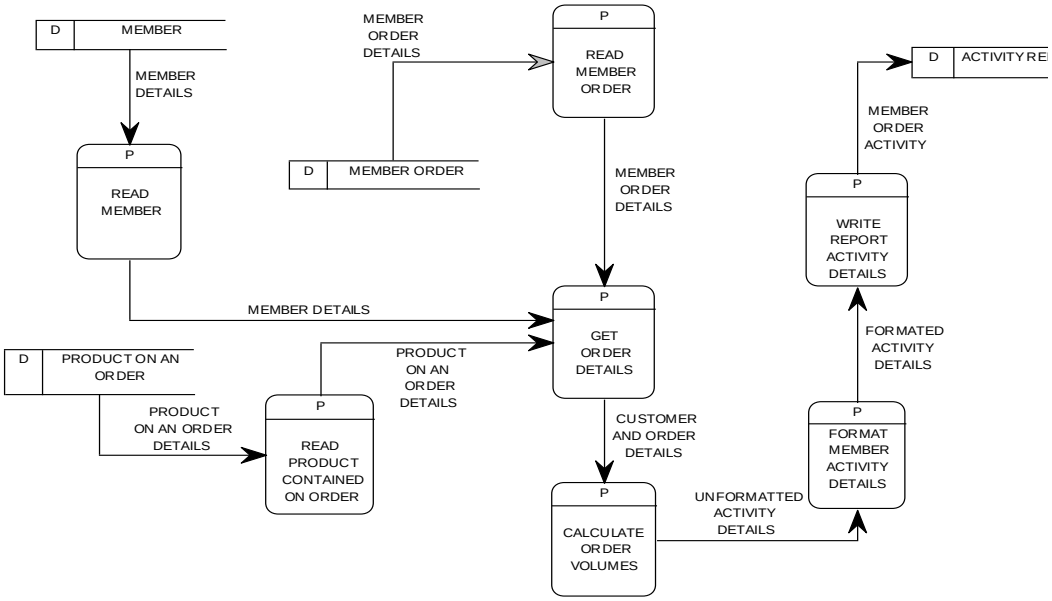
- All other processes are then considered to be part of the central transform! The processes that do the real work — making decisions or transform data (such as checking a customer's credit or calculating an employee's pay). They can be characterized as processes that produce an output that can clearly be distinguished from the input data flow(s). In other words, the output data flow clearly is different in content or meaning from the incoming data flow.

Creating the STC after partitioning the DFD

Once the DFD has been partitioned, a structure chart can be created that communicates the modular design of the program.

1. Create a process that will serve as a "commander and chief" of all other modules. This module manages or coordinates the execution of the other program modules.
2. The last process encountered in a path that identifies afferent processes becomes a second-level module on the structure charts.
3. Beneath that module should be a module that corresponds to its preceding process on the DFD. This continues until all afferent processes in the sequence path are included on the structure chart.
4. If there is only one transformation process, it should appear as a single module directly beneath the boss module. Otherwise, a coordinating module for the transformation processes should be created and located directly above the transformation process.
5. A module per transformation process on the DFD should be located directly beneath the controller module.
6. The last process encountered in a path that identifies efferent processes becomes a second-level module on the structure chart.
7. Beneath the module (in step 6) should be a module that corresponds to the succeeding process appearing on the sequence path.
8. Likewise any process immediately following that process would appear as a module beneath it on the structure chart.

Example: Ordering System



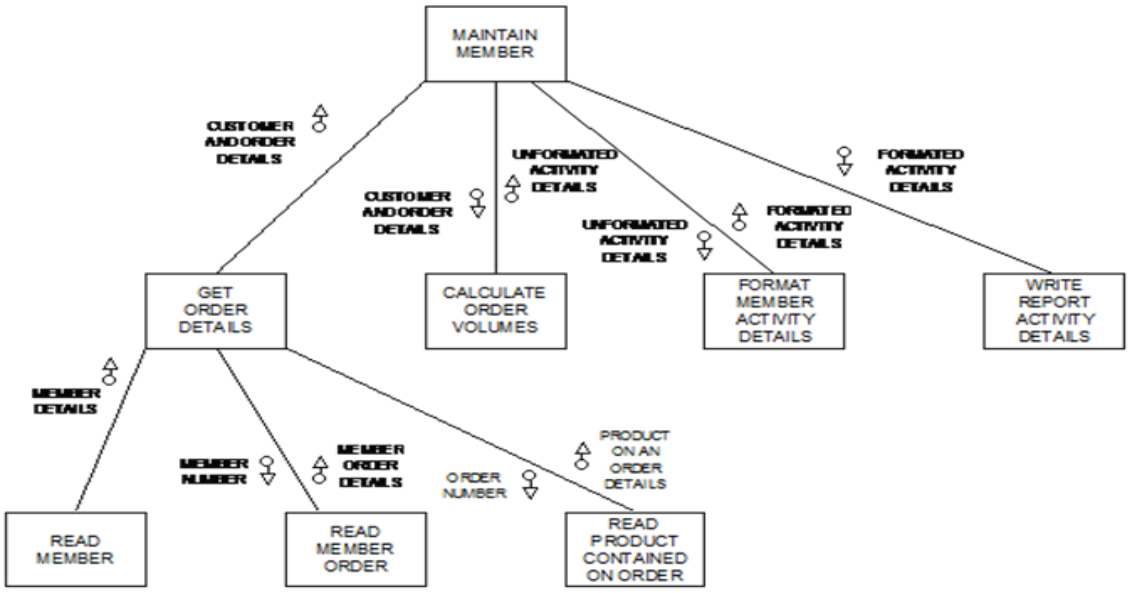


Figure 2: Sample SoundStage Structure Chart from Transform Analysis

On the structure chart above, the sequence for which GET ORDER DETAILS calls the three input processes is very significant:

It is essential that READ MEMBER precedes READ MEMBER ORDER, which in turn must precede READ PRODUCT CONTAINED ON ORDER.

The design of this output would communicate that data on the output is organized by member, then by orders for each member, followed by details regarding products appearing on each order. The fact that READ MEMBER ORDER must precede READ PRODUCT CONTAINED ON ORDER is also reinforced by examining our data model diagram and noticing the relationship paths (for example, MEMBER is related to MEMBER ORDER, which is related to PRODUCT CONTAINED ON ORDER).

Transaction Mapping/ Analysis

Transaction analysis is the examination of the DFD to identify processes that represent transaction centers. It is an alternative structured design strategy for developing structure charts.

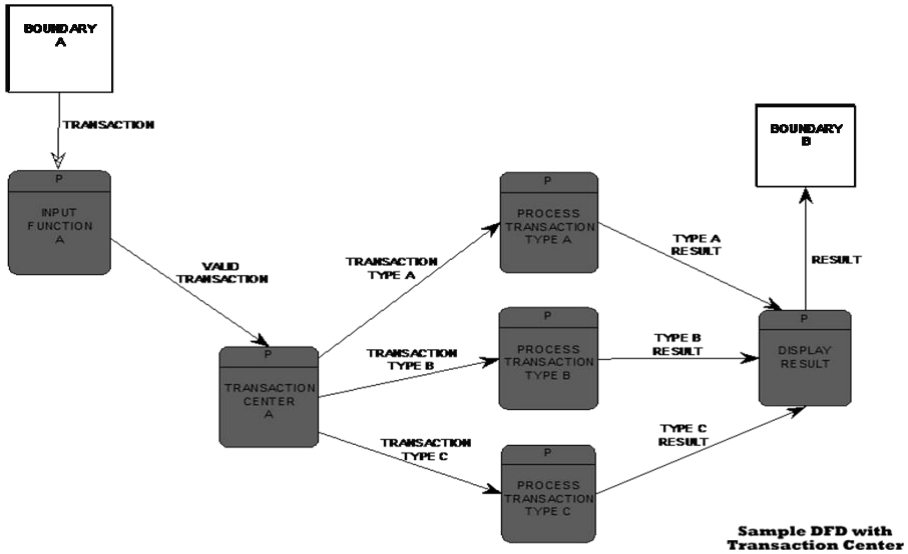
It is applied when a single information item causes (triggers) data flow to branch along one of many paths. The DFD is mapped into a structure that allocates control to a substructure that acquires and evaluates a transaction. Another substructure controls all potential processing actions based on a transaction.

Transaction flow is characterized by data moving along an incoming path that converts external world information into a transaction. The transaction is evaluated and, based on its value, flow along one of many *action paths* is initiated

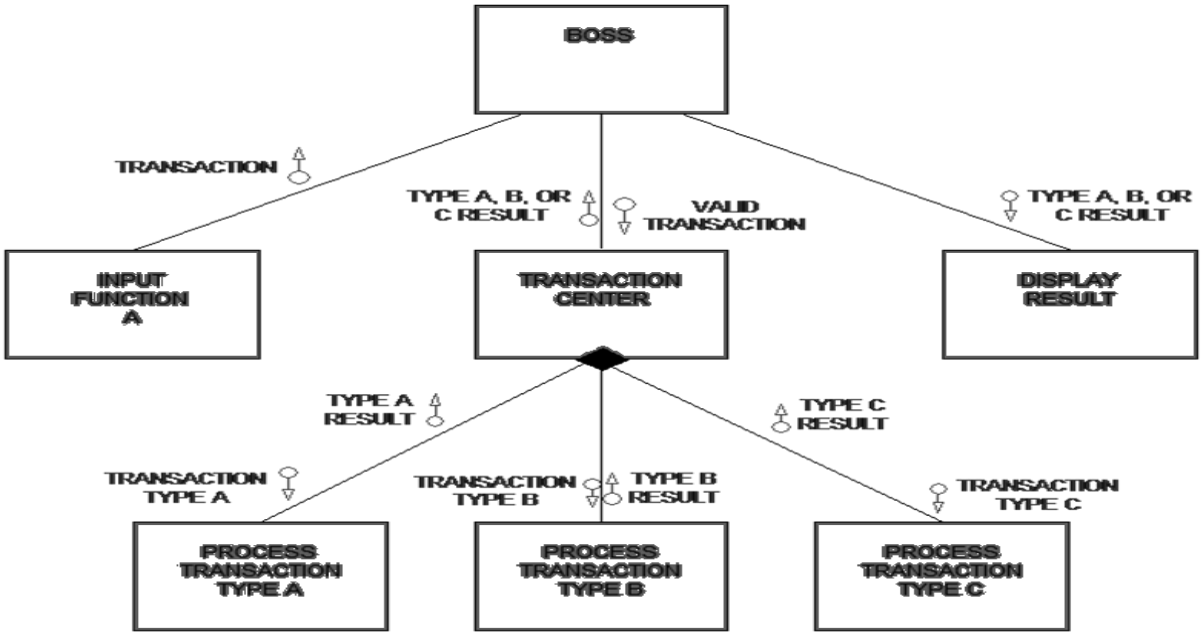
A transaction center is a process that does not do actual transformation upon the incoming data (data flow); rather, it serves to route the data to two or more processes. It is the hub of information flow from which many action paths emanate is called a *transaction center*.

Analogy: *a traffic cop that directs traffic flow.*

These processes are usually easy to recognize on a DFD, because they usually appear as a process containing a single incoming data flow to two or more other processes.



Sample DFD with Transaction Center



The primary difference between transaction analysis and transform analysis is that transaction analysis recognizes that modules can be organized around the transaction center rather than a transform center.

Note: Within a DFD for a large system, both transform and transaction flow may be present. For example, in a transaction-oriented flow, information flow along an action path may have transform flow characteristics.

Structure Chart Quality Assurance Checks

Structure charts developed through structured design are evaluated for quality. By using the Yourdon/Constantine strategy to divide a program into modules, you are able to end up with modules that are said to be loosely coupled and highly cohesive.

In the following sections we examine these two measures of program design quality. As you study these two measures, recognize that the data and control flow symbols depicted on a structure chart can serve as aids in determining the degree of coupling and cohesion of modules.

Refining The Architectural Design

Successful application of transform or transaction mapping is supplemented by additional documentation that is required as part of architectural design. After the program structure has been developed and refined, the following tasks must be completed:

- A processing narrative must be developed for each module.
  This is (ideally) an unambiguous, bounded description of processing that occurs within a module. The narrative describes processing tasks, decisions, and I/O.
- An interface description is provided for each module.
  This describes the design of internal module interfaces, external system interfaces, and the human/computer interface.
- Local and global data structures are defined.
  The design of data structures can have a profound impact on architecture and the procedural details for each software component.
- All design restrictions and limitations are documented.
  For example: restriction on data type or format, memory or timing limitations; bounding values or quantities of data structures;
  The purpose of a restrictions and limitations section is to reduce the number of errors introduced because of assumed functional characteristics.
- Refinement is considered (if required and justified).
  Design refinement should strive for the smallest number of modules that is consistent with effective modularity and the least complex data structure that adequately serves information requirements.

Structured Analysis and Design - Summary

Structured analysis is a set of techniques and graphical tools that allow the analyst to develop a new kind of system specification that are easily understandable to the user.

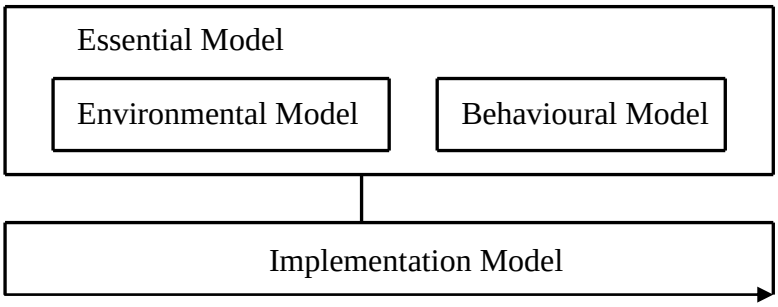
Goals of Structured Analysis and Design

- Improve Quality and reduce the risk of system failure
- Establish concrete requirements specifications and complete requirements documentation
- Focus on Reliability, Flexibility, and Maintainability of system

Characteristics of a Good analysis method

- Graphical with supporting text.
- Allow system to be viewed in a top-down and partitioned fashion.
- Minimum redundancies.
- Reader should be able to predict system behavior.
- Easy to understand by user.

Elements of Structured Analysis and Design



A. Structured Analysis

Essential Model

This is a model of what the system must do. However, it does NOT define HOW the system will accomplish its purpose. It is a combination of the environmental and behavioural model.

i. Environmental Model

- Defines the scope of the proposed system
- Defines the boundary and interaction between the system and the outside world
- Composed of: Statement of Purpose, Context diagram, and Event List

ii. Behavioural Model

- Model of the internal behaviour and data entities of the system
- Models the functional requirements
- Composed of Data Dictionary, Data Flow Diagram, Entity Relationship Diagram, Process Specification and State Transition Diagram

The Essential Model can be thought of as the analysis model of the software/system

B. Structured Design

Implementation Model

This model maps the functional requirements to the hardware and software. It thus minimizes the cost of development and maintenance as it determines which functions should be manual vs. automated. The tools used for this model are the Structure Charts.
The Structure Chart representation can be easily implemented using a programming language.

Analysis and Design Process